

Earliest Query Answering for Regular Queries with Complete Subhedge Projection

Antonio AL SERHALI

Doctoral Dissertation

submitted to obtain the degree

DOCTOR OF THE UNIVERSITY OF LILLE

in Computer Science

Committee members

<i>Reporters</i>	Sebastian MANETH	Professor at University of Bremen
	Sylvain SCHMITZ	Professor at University Paris-Cité
<i>Thesis director</i>	Joachim NIEHREN	Research director at Inria
<i>President</i>	Sophie TISON	Professor emeritus at University of Lille

defended on December 12, 2024

Evaluation au Plus Tôt de Requêtes Régulières avec Projection de Sous-Haies Complète

Antonio AL SERHALI

Thèse de doctorat

présentée en vue de l'obtention du grade de

DOCTEUR DE L'UNIVERSITÉ DE LILLE

en Informatique

Composition du jury

<i>Rapporteurs</i>	Sebastian MANETH	Professeur à l'Université de Brême
	Sylvain SCHMITZ	Professeur à l'Université Paris-Cité
<i>Directeur de thèse</i>	Joachim NIEHREN	Directeur de recherche Inria
<i>Présidente</i>	Sophie TISON	Professeure émérite à l'Université de Lille

soutenue le 12 décembre 2024

Abstract

Logical queries are at the core of graph databases, complex event processing, and XML stream processing. The efficiency of query answering algorithms is crucial in practice, despite the inherent theoretical complexity of the underlying algorithmic problem, which rules out any solution that is at the same time fully general and efficient.

We reconsider the problem of how to answer regular queries on sequences of data trees – often called hedges – in an earliest manner. This problem was first studied by Gauwin et al. in 2011. Their original motivation, and still our main application, is to answer regular XPATH queries on XML streams with the lowest possible latency. Query answers should be output immediately whenever they become certain independently of the continuation of the stream.

Earliest query answering for deterministic nested word automata was shown to be in polynomial time by Gauwin et al. (2011). Unfortunately, their earliest query answering algorithm was not successful in practice when applied to regular XPATH queries on XML streams: the deterministic nested word automata obtained were huge even for small XPATH queries, the processing time per event of the stream was too high, and no projection of irrelevant events was available. Therefore, the best current tools for evaluating regular XPATH queries on XML streams – obtained by Sebastian et al. (2015) – are based on an approximation of earliest query answering that avoids the need to determinize nested word automata.

In this thesis, we show that earliest query answering for regular XPATH queries is feasible in practice. For this, we develop a new earliest query answering algorithm for deterministic stepwise hedge automata (dSHAS). This more recent automaton model from Sakho et al. (2021) combines deterministic finite state automata for words and trees in a natural manner. We enhance our algorithm with complete subhedge projection in order to project irrelevant subhedges maximally. We show how to obtain small dSHAS for regular XPATH queries harvested from practical XSLT and XQUERY programs by Lick and Schmitz by schema-based determinization, develop a streaming algorithm for monadic queries defined by dSHA with better worst case complexity, and the first complete subhedge projection algorithms for dSHAS. It turns out that complete subhedge projection makes earliest query answering with dSHAS competitive in time efficiency with the best existing streaming tools for general regular XPATH queries, while being more memory efficient in cases where these tools are not earliest. We believe that the algorithmic progress made on earliest query answering on streams in the present thesis will eventually enable earliest query answering on hyperstreams as proposed by Sakho et al. (2021).

Résumé

Les requêtes logiques sont au cœur des bases de données graphes, du traitement d'événements complexes et du traitement de flux XML. L'efficacité des algorithmes de réponse à ces requêtes est cruciale en pratique, malgré la complexité théorique inhérente du problème algorithmique sous-jacent, qui exclut toute solution à la fois pleinement générale et efficace.

Nous réexaminons le problème de la réponse au plus tôt aux requêtes régulières sur des séquences d'arbres de données, souvent appelées haies. Ce problème a été étudié pour la première fois par Gauwin et al. en 2011. Leur motivation initiale, qui demeure notre principale application, est de répondre aux requêtes régulières `XPATH` sur des flux XML avec la latence la plus faible possible. Les réponses aux requêtes doivent être produites immédiatement dès qu'elles deviennent certaines, indépendamment de la continuation du flux.

Gauwin et al. (2011) ont montré que la réponse au plus tôt aux requêtes pour les automates à mots imbriqués déterministes est en temps polynomial. Malheureusement, leur algorithme de réponse n'était pas suffisamment performant en pratique lorsqu'il était appliqué aux requêtes régulières `XPATH` sur des flux XML : les automates obtenus étaient énormes même pour de petites requêtes `XPATH`, le temps de traitement par événement du flux était trop élevé, et aucune projection d'événements non pertinents n'était disponible. Par conséquent, les meilleurs outils actuels pour évaluer les requêtes régulières `XPATH` sur des flux XML – obtenus par Sebastian et al. (2015) – reposent sur une approximation des réponses au plus tôt qui évite la nécessité de déterminer des automates à mots imbriqués.

Dans cette thèse, nous montrons que la réponse au plus tôt aux requêtes régulières `XPATH` est réalisable en pratique. Pour ce faire, nous développons un nouvel algorithme pour les automates de haies déterministes pas-à-pas (`dSHAS`). Ce modèle d'automate plus récent de Sakho et al. (2021) combine de manière naturelle les automates à états finis déterministes pour les mots et les arbres. Nous renforçons notre algorithme avec une projection complète de sous-haies afin de projeter au maximum des sous-haies non pertinentes. Nous montrons comment obtenir de petits `dSHAS` pour des requêtes régulières `XPATH` extraites de programmes `XSLT` et `XQUERY` pratiques par Lick et Schmitz, par détermination guidée par schéma, développons un algorithme de streaming pour des requêtes monadiques définies par `dSHA` avec une meilleure complexité dans le pire des cas, et introduisons les premiers algorithmes complets de projection de sous-haies pour les `dSHAS`.

Il s'avère que la projection complète de sous-haies rend la réponse au plus tôt avec les `dSHAS` compétitive en termes d'efficacité temporelle par rapport aux meilleurs outils de streaming existants pour les requêtes régulières `XPATH` générales, tout en étant plus efficace en mémoire dans les cas où ces outils ne fonctionnent pas au plus tôt. Nous croyons que les progrès algorithmiques réalisés sur l'approche au plus tôt sur les flux de données dans la présente thèse permettront finalement d'avoir cette même approche sur les hyperflux, comme proposé par Sakho et al. (2021).

Acknowledgments

First and foremost, I sincerely thank Joachim for giving me the opportunity to work in research and pursue a PhD. It has been a long and fruitful journey. My gratitude also goes to Professors Sebastian Maneth and Sylvain Schmitz for reviewing my dissertation and providing valuable feedback, and to Professor Sophie Tison for being part of the committee and presiding over it. Finally, a heartfelt thanks to the Links team for the coffee chats, seminar discussions, and collaborations (after all, I started as an R&D engineer in the team).

Now, since this is probably the only section where I can dodge science and readings, it's dedicated to some of the experiences, things, and places that made this journey bearable—and even memorable. No algorithms, no automata, and no names (a quirky habit of mine to link memories to places, moments, and, yes, even the weather). But trust me—you'll know who you are!

It began in 2020 with the "delightful" outbreak of Covid-19. Apparently, a PhD wasn't challenging enough without confinement and a looming death threat. Room number 5, 4 a.m. dinner cooking (don't judge), a lovely soul, and long nights on "Clubhouse" with friends from all over the world made surviving that period possible.

Coffee. The coffee machine. The second coffee machine (don't ask). Beer—Belgian beer (arguably the best). Wine—French, obviously. German wine—yes, it exists, and I know at least one person who argues it's the best. Local markets—no visit to Lille is complete without Nicola's pizza from the Minitalie food truck. Senegalese food (ask J.), Molla's Kurdish dishes, the clubs on Rue Royale, and more. Each was a much-needed escape to recharge.

Lille and its weather—enough said. Dijon, the bittersweet. Lyon, an old childhood friendship. Rouen, the newest piece of my precious people puzzle finding its place. Nearby Ghent, Bruges, the Belgian coast, and the Normandy coast where you can still see traces of Europe's "affection" for each other (I couldn't resist—sorry!). On a side note, the universe seemed to enjoy throwing heavy events at us—from the Middle East crises to the Russian-Ukrainian war. Away from stereotyping, and while the sample size is not statistically relevant for a valid conclusion, but let's just say a political discussion between a Middle Easterner and a German might be best avoided heh! Hopefully, the universe will someday find a way to "expand" without throwing in more wars. The long car trips (I know, not the most eco-friendly choice, and no, Paris didn't make the cut—too noisy for the brain) to the Netherlands, Germany, Belgium, Switzerland, Luxembourg, and the rest of France. Unforgettable, thanks to all who tagged along and turned them into cherished memories.

Lastly, Lebanon—my roots, my heart, my nostalgia, and my backbone. No matter how far I go, how long I'm gone, or what arguably questionable choices I make, it's always there—never judging, always supporting, and always welcoming me back.

Funding Acknowledgments

I would like to express my gratitude to the "ANR" and the "Région Hauts-de-France" for funding my PhD project and supporting my research endeavors. I am also deeply thankful to the Inria Research Center of Lille for providing an outstanding working environment as part of the Links team, where I benefited from enriching collaborations, insightful discussions, and access to all the resources I needed throughout my PhD journey—even during the stay-at-home challenges of the Covid period. Finally, I sincerely thank the BioComputing team at Cristal, who graciously welcomed me into their offices during my final year of research and the most challenging phase of thesis writing.

Contents

1	Introduction	1
1.1	Context	1
1.1.1	Logical Queries	1
1.1.2	Streaming Query Answering	3
1.2	Open Challenges	6
1.3	Contributions	9
1.3.1	Small dSHAS for Regular XPATH Queries	9
1.3.2	Subhedge Projection Algorithms for dSHAS	10
1.3.3	Earliest Query Answering for dSHAS	11
1.4	Further Related Work	13
1.4.1	Complex Event Processing	13
1.4.2	XML Stream Processing	14
1.4.3	Projection for XPATH Queries	15
1.4.4	Parallelism and Streams	16
1.5	Outline	17
1.6	Publication Comments	17
Part I	Starting	19
2	Preliminaries	21
2.1	Mathematical Notation	22
2.2	Words, Hedges and Nested Words	22
2.2.1	Words	22
2.2.2	Hedges	23
2.2.3	Nested Words	26
2.2.4	Hedge Traversals	27
2.2.5	Nested Words Prefixes	29
2.3	Regular Languages and Queries	30
2.3.1	Regular Expressions	30

2.3.2	Nested Regular Expressions	32
2.3.3	Regular Monadic Queries	34
2.3.4	Schema Constraints for x-Annotations	37
3	Hedge Automata	39
3.1	Finite State Automata on Words (NFAs)	40
3.1.1	Syntax	41
3.1.2	Semantics	41
3.1.3	Size Measures	42
3.1.4	Graphs	42
3.1.5	Completion	42
3.1.6	Runs	44
3.1.7	Determinization	44
3.1.8	Complementation	46
3.1.9	Product and Intersection	46
3.1.10	Accessibility	48
3.1.11	Minimization	49
3.1.12	Cleaning	49
3.1.13	Infinitary NFAs	50
3.1.14	Adding Else Rules	51
3.2	Stepwise Hedge Automata (SHAs)	53
3.2.1	Syntax	54
3.2.2	Semantics	56
3.2.3	Size Measures	58
3.2.4	Completion	58
3.2.5	Runs	58
3.2.6	Determinization	62
3.2.7	Complementation	62
3.2.8	Product and Intersection	63
3.2.9	Hedge Accessibility	63
3.2.10	Minimization	64
3.2.11	Cleaning	64
3.2.12	Infinitary SHAs	65
3.2.13	Else Rules	65
3.2.14	Related Automata Models	65
3.3	Downward Stepwise Hedge Automata (SHA [↓] s)	66
3.3.1	Syntax	67
3.3.2	Semantics	68

3.3.3	Completion	69
3.3.4	Runs	69
3.3.5	Conversion between SHAS and SHA^\downarrow s	70
3.3.6	Determinization	71
3.3.7	Minimization	72
3.3.8	Relationship to Nwas	72
3.4	Membership Testing	75
3.4.1	In-Memory	76
3.4.2	Streaming	77
3.5	Schema-Completeness	79
3.6	Schema-Based Cleaning	82
3.7	Two-sorted Automata	82
3.7.1	2-Sorted SHAS	83
3.7.2	2-Sorted SHA^\downarrow s	84
4	XML Documents and XPATH Queries	85
4.1	XML Documents	86
4.2	Hedge Encoding of XML Documents	87
4.2.1	General Encoding	87
4.2.2	Schema of Hedge Encodings	88
4.3	XPATH	89
4.3.1	Regular Fragment	89
4.3.2	Non-regular Queries	90
4.3.3	Types and Functions	91
4.3.4	Answer Sets	91
4.3.5	Variables	91
4.4	XPATH Benchmarks	92
4.4.1	XPATHMARK Benchmark	92
4.4.2	Lick and Schmitz' Benchmark	93
4.5	Schema Constraints for x-Annotations	94
4.6	Available Deterministic SHAS	95
Part II	Determinization	105
5	Schema-based Determinization	107
5.1	Introduction	108
5.2	Accessible Determinization	112
5.3	Schema-Based Cleaning for NFAs	114

5.4	Schema-Based Determinization for NFAs	117
5.5	Schema-Based Cleaning and Determinization for SHAs	124
5.6	Correctness Proof	127
5.7	Scaling Experiments	131
6	Benchmark of XPATH Queries	137
6.1	Introduction	138
6.2	Subcorpus of Lick and Schmitz' Benchmark	140
6.3	A Schema for XML Documents	146
6.4	Compiler to Automata	146
6.4.1	Parser	147
6.4.2	Nested Regular Expressions	147
6.4.3	Compiler to SHAs	148
6.4.4	Schema-Based Determinization	150
6.4.5	Minimization	151
6.4.6	Examples	151
6.5	Testing Automata	153
6.5.1	Membership for Samples	153
6.5.2	Query Evaluation	154
6.6	Automata Statistics	155
6.6.1	XPATHMARK Benchmark	155
6.6.2	Lick and Schmitz' Benchmark	156
6.7	Example Automata for Lick and Schmitz' Benchmark	159
Part III	Projection	167
7	Complete Subhedge Projection	171
7.1	Introduction	172
7.2	Definitions and Properties	174
7.2.1	Irrelevant Subhedges	174
7.2.2	Basic Properties	176
7.2.3	Completeness for Subhedge Projection	178
7.3	Safe-No-Change Projection	183
7.3.1	Algorithm	183
7.3.2	Soundness	186
7.3.3	Incompleteness	196
7.4	Congruence Projection	197
7.4.1	Motivation	198

7.4.2	Approach	198
7.4.3	Algorithm	200
7.4.4	Soundness	207
7.4.5	Completeness	217
7.4.6	Automata Sizes	222
8	Complete Suffix Projection	223
8.1	Introduction	224
8.2	Certainty	226
8.2.1	Σ -Certain Membership	226
8.2.2	Certain Non-membership	227
8.3	Schema-Safety	227
8.4	Certainty Automata	229
8.4.1	Membership	229
8.4.2	Non-membership	232
8.4.3	Combining Both	232
8.5	Earliest $d\text{SHA}^\downarrow$ s with Complete Suffix Projection	233
9	Combining Subhedge and Suffix Projection	237
9.1	Combination Algorithm	237
9.2	Soundness and Completeness	238
9.3	Automaton Size	240
9.4	Benchmark $d\text{SHA}$ for XPATH Queries	240
10	Projecting Evaluators for Earliest Membership	245
10.1	Early Evaluators with Projection	245
10.1.1	In-Memory Evaluator	246
10.1.2	Streaming Evaluator	249
10.2	Earliest Membership with Projection	254
10.2.1	In-Memory Complexity	254
10.2.2	Streaming Complexity	255
Part IV	Querying	257
11	Earliest Query Answering for Regular Monadic Queries	261
11.1	Introduction	262
11.2	Certainty for Monadic Queries	263
11.2.1	Certain Answers	264
11.2.2	Certain Non-answers	264

11.2.3 Deciding Certainty	265
11.3 Candidate Automata	267
11.3.1 Construction	267
11.3.2 In-Memory Correctness	269
11.3.3 Streaming Correctness	273
11.4 Earliest Monadic Query Answering	277
11.4.1 Earliest Candidate Automata	277
11.4.2 Adding Subhedge Projection	280
12 Experiments on Regular XPath Evaluation on XML Streams	283
12.1 Introduction	284
12.2 Streaming Evaluation Tool: ASTREAM	284
12.3 Experiments without projection: Linear increase with size	286
12.3.1 Scaling-Up Document Sizes without Subhedge Projection	286
12.3.2 Being Earliest	287
12.3.3 Factorization	287
12.4 Experiments with projection	288
12.4.1 Evaluation Measures	288
12.4.2 Earliest Congruence Projection	290
12.4.3 Earliest Safe-No-Change Projection	291
12.4.4 Comparison to External Tools	292
12.4.5 Experiments with Lick and Schmitz' Benchmark	293
13 Conclusion and Future Work	297
Bibliography	301

Introduction

1.1 Context

In this thesis, we study the problem of answering logical queries on streams containing linearizations of data trees or hedges. We start by recalling logical languages in which logical queries can be specified and then discussing how they are used in stream processing.

1.1.1 Logical Queries

Logical queries are at the core of database theory [Arenas *et al.* 2022, Abiteboul *et al.* 1995], knowledge representation [Fagin *et al.* 2003], complex event processing [Cugola & Margara 2012], data exchange [Arenas *et al.* 2014], Web information extraction [Abiteboul *et al.* 2000], and XML document transformation [Kay 2011]. Typical examples of logical languages are first-order logic [Smullyan 2012], monadic second-order logic [Comon *et al.* 2002], and Datalog [Ceri *et al.* 2012]. The formulas of a logical language can specify conditions – based on the usual logical connectives and possibly with recursion – upon which information may be extracted from relational structures, such as graphs, hypergraphs, trees, and relations (tables).

For many logical languages, however, the algorithmic problem of query answering is highly complex. Even the simpler problem of deciding whether a given query has some answer on a given relational structure is often computationally hard. The most well-known example is first-order logic – the foundation of relational databases – where satisfiability is PSPACE-complete in terms of combined complexity

[Stockmeyer 1974, Vardi 1982]. Even worse, satisfiability remains NP-complete for conjunctive first-order queries [Cook 1971]. As a result, the satisfiability of SQL's widely-used select-project-join queries is also NP-complete when both the query and the database are provided as inputs. Thus, under the common complexity assumption that $P \neq NP$, query answering is not in polynomial time even for the most frequent classes of database queries in practice.

A key idea for reducing the algorithmic complexity of logical queries is to restrict the use of variables. For instance, one might consider various notions of acyclic queries [Yannakakis 1981, Durand & Grandjean 2004, Bergougnoux *et al.* 2019], or else, limit the number of free variables in sub-formulas [Immerman & Kozen 1989, Schwentick & Zeume 2012]. An interesting alternative is to use variable-free logics, such as modal logic [Kripke 1959, Blackburn *et al.* 2001] or temporal logic [Barcelo & Libkin 2005, Hustadt 2001]. A particularly powerful variable-free logic for graphs is the language of Nested Regular Path Queries (NRPQs), first introduced in the seventies as formulas of propositional dynamic logic [Fischer & Ladner 1979]. For NRPQs, fortunately, the query answering problem is known to be in combined linear time $O(|Q| |G|)$, where $|Q|$ represents the size of the query and $|G|$ the size of the graph. This complexity upper bound is folklore in the context of database theory and has also been shown for the even richer logical language called the alternation-free modal μ -calculus [Cleaveland & Steffen 1991].

When restricting graphs to data trees or hedges, NRPQs can be identified with the navigational core of XPATH 3.0. In contrast to the navigational cores of XPATH 2.0 [Marx 2004] and XPATH 1.0 [Gottlob *et al.* 2003], NRPQs not only support filters with logical operators but also Kleene's star operator without any restriction. The latter cannot be expressed natively in XPATH 3.0; however, it is still expressible by using recursive functions, which are one of the important additions in XPATH 3.0. For instance, if one wants to select the name of all persons in an XML document, for which an address is provided, as well as a phone number or a homepage, and also a credit card number or a profile, the query can be formulated as follows in the navigational core of XPATH 1.0:

```
//person
  [address and (phone or homepage)
    and (creditcard or profile)]/name
```

Here, the operator $/$ hides a *child* axis, and the operator $//$ a descendant axis, i.e., the transitive closure $child^+$ of the *child* relation. Beside these navigational

operators, this XPATH query includes a filter `[address and ...]` that contains both conjunctions and disjunctions. Note that further *child* axis are hidden in the above filter, which could be equivalently written as:

```
[child::address and (child::phone or child::homepage)
and (child::creditcard or child::profile)]
```

This should make it clearer that `address` is a label test for some child node of the current node of the hedge. On the other hand, the core of the query language XPATH 3.0 rules out the comparisons between data values, which are otherwise permitted in full XPATH. For instance, if one wants to know whether the same *firstname* is used by both a male and by a female in the database, it can be done by the following comparison of data values:

```
//male/firstname = //female/firstname
```

NRPQs on data trees or hedges can be compiled into nested regular expressions with an arbitrary but fixed selection variable x , and thus, into hedge automata [Niehren & Sakho 2021]. This shows that all NRPQs for hedges define regular queries, while the addition of comparisons of data values enable definitions of non-regular queries. For this, the language of conjunctive regular path queries (CRPQs) is more widely used as logical foundation of graph databases [Florescu *et al.* 1998, Angles *et al.* 2017], but at the cost of variables coming back.

In this thesis, however, we will focus exclusively on regular monadic queries on hedges with data values. Still, we permit to compare data values to constants, such as in the query that selects all first names in the database containing the data value “Marie”:

```
//male/firstname[contains(., 'Marie')]
```

Comparisons with constant data values enable a restrictive usage of the data values in the hedges without compromising the regularity of the queries.

1.1.2 Streaming Query Answering

Query answering on data streams has been studied extensively in the fields of XML document processing and Complex Event Processing (CEP), where a complex event is nothing else than a query. The data streams contain words with or without

parenthesis, with the main restriction being that the word on the stream can be read only once from the left to the right.

Query answering on hedges becomes even more challenging when the hedge can only be read in streaming mode, meaning that a linearization of the hedge as a nested word is received through an input stream. Users often expect that query answers are produced with low latency, i.e., as soon as sufficient information has been observed in the stream, rather than having to wait for the end of the stream to conclude. Consequently, streaming algorithms should operate incrementally, processing streams event by event (i.e. letter by letter), immediately as they arrive. For certain applications, such as web advertising placement or detecting web service attacks, answers must not only be provided with low latency but also in real time, often within milliseconds.

From an algorithmic perspective, non-answers must be discovered and rejected eagerly once they become certain. These are answer candidates that can no longer be selected in any continuation of the stream. If not addressed, these non-answers may require excessive space and processing time. For monadic queries, the number of certain non-answers can be linear in the size of the stream, making it impossible to store all of them for large streams (stream size exceeding memory capacity, which can reach tens of gigabytes on typical modern machines). Furthermore, examining each certain non-answer for every event may require quadratic time in the size of the stream, which is unacceptable, even for streams of moderate size (typically on the order of several tens of megabytes).

A certain query answer is an answer candidate in a stream that will be selected in all its continuations. Low-latency query answering requires the eager output of certain query answers. An alive answer candidate is an answer candidate in a stream that is neither certain for selection nor certain for rejection. The concurrency of a query at an event in a stream refers to the number of its alive candidates. Many practical queries exhibit only a small concurrency. This holds for instance for 0-delay regular path queries, such as:

`//a/b/c`

Whether a c-event on the stream can be selected can be decided without looking into the future of the stream. The situation becomes more complicated for NRPQs with filter from Core XPath 1.0 such as:

`//a/b/c[child::d]`

In many practical complex event processing systems [Carney *et al.* 2002, Chandrasekaran *et al.* 2003], queries are specified by using sliding windows. The delay of such queries is the window’s size w , so the concurrency of the query is bounded by $w + 1$. In XML stream processing, however, regular XPATH queries with unbounded delay and concurrency cannot be excluded, as illustrated already by the above NRPQ example from Core XPATH 1.0 with a filter. Therefore, all existing streaming tools for regular XPATH queries with large coverage such as Olteanu’s SPeX [Olteanu 2007] and Sebastian’s QUIXPATH [Sebastian 2016, Debarbieux *et al.* 2015] need clever strategies to remove certain non-answers and detect certain answers as quickly as possible.

This brings us to the main problem addressed in this thesis: *Earliest Query Answering (EQA)*. EQA refers to the problem of answering a given query on a given stream in an earliest manner, i.e., determining certain answers with the lowest possible latency, while discarding certain non-answers as soon as possible. In this work, we specifically focus on EQA for regular queries on streams containing data hedges, i.e., sequences of letters and trees containing data hedges themselves. In other words, the streams send the letters of some nested word.

Unfortunately, EQA is computationally hard, even for small logical query languages for which query answering is in polynomial time. More precisely, the decision problem for a query language – i.e. determining whether a candidate in the stream is a certain answer to a query in that language – is a universality problem [Gauwin & Niehren 2011]. Since EQA is a universality problem, both EQA and universality are coNP-hard problems for the language of forward regular path queries with the axes *child* and *child**, including filters and negation (see Proposition 17 of [Gauwin 2009]).

On the positive side, Gauwin *et al.* [Gauwin *et al.* 2009b, Gauwin 2009] showed that EQA is in polynomial time for regular queries on nested words (i.e., hedges) defined by deterministic nested word automata (NwAs). Determinism is crucial here, as it renders the language universality problem feasible, and thus also EQA. Gauwin’s EQA algorithm requires quadratic time per event $O(n^2)$, where n is the number of states of the deterministic NwA, following a cubic precomputation in time $O(n^3)$. The first bottleneck of this approach in practice is the determinization algorithm for NwAs, which may produce huge results even for simple regular XPATH queries, or may not even terminate after several hours [Debarbieux *et al.* 2015].

To avoid the bottleneck of NwA determinization, Sebastian *et al.* proposed compiling forward regular XPATH queries to nondeterministic early NwAs for approxi-

matizing *EQA* [Sebastian 2016, Debarbieux *et al.* 2015]. Additionally, they presented subtree and descendant projection algorithms for *NWAs* to improve the efficiency of their streaming algorithms. Their *QUIXPath* tool, based on these algorithms, is currently the most efficient streaming tool for answering *XPATH* queries on streams with large coverage.

The time per event of *QUIXPath* is limited to $O(n)$, even for queries with large concurrency, due to stack-and-state sharing. However, for some queries with low concurrency, where their *EQA* approximation is not exact, *QUIXPath* may run out of memory on huge streams, even though the concurrency c is low. In contrast, for queries for which *QUIXPath* is earliest, the required memory is bounded by

$$O(c + m + n \text{ depth}(h))$$

where m is the overall size of the input early *NWA*, n the number of its states, and c the concurrency the automaton's query on h , and $\text{depth}(h)$ the depth of hedge h . So when bounding the concurrency c and $\text{depth}(h)$, *QUIXPath* should not run out of memory for queries where it is earliest even for huge input streams.

1.2 Open Challenges

Following the above, several open challenges and questions emerge. These highlight the problems for which we will present solutions in this dissertation. Here, we lay them out:

Question 1. Does there exist sufficiently small deterministic automata for regular path queries, that allow to answer regular *XPATH* queries on *XML* streams in an earliest manner in practice? And if so, how to compute them?

As discussed above, *XML* stream processing has been widely studied, with substantial research efforts aimed at improving its practical efficiency, particularly in terms of large coverage, minimal latency, and low memory consumption. Large coverage, combined with streaming, calls for automata-based approaches, as shown in the best previous solutions. Optimal latency in all cases, calls for deterministic automata due to complexity reasons.

The determinization of automata for words and trees is usually based on the subset construction, which may raise an exponential blow-up of the automaton size. The usual determinization algorithm for nested word automata [von Braunmühl & Verbeek 1985, Alur & Madhusudan 2004,

Okhotin & Salomaa 2014] combines a pair and a subset construction, so it may raise an exponential blow-up too. For NwAs, however, this may happen much more frequently than for NFAs as noticed by Debarbieux et al. [Debarbieux *et al.* 2015]. For instance, they report a huge blow-up for their attempt to determinize an NwA for the XPATH query:

```
//a[following-sibling::b[./c][./d]]/e
```

The nondeterministic NwA that they compiled from the query has 38 states and 7,719 transitions. The determinization of this NwA runs out of memory, after having discovered more than 5,000 accessible states and 20 million transitions.

Niehren and Sakho [Niehren & Sakho 2021] revisited the problems of determinization and minimization of automata for nested words. They propose to reduce NwA determinization to the determinization of stepwise hedge automata (SHAs), in order to obtain smaller deterministic NwAs. SHAs are a more recent automaton model that they introduce, by combining finite state automata for words and trees in a natural manner. They have the advantage that they can be determinized by the usual subset construction, avoiding the difficulties of classical NwA determinization.

Niehren and Sakho also noticed an important role of schemas of queries, i.e., to which hedges the query may be applied. Which regular languages represent a given regular query may depend on the schema. Additionally, they propose schema-based automata cleaning – which may change the automaton’s language outside the schema – in order to reduce the size of minimal deterministic automata. In this way, they obtained promising results, producing smaller deterministic automata for all regular XPATH queries of the XPATHMARK benchmark.

However, their approach still falls short when applied to the practical queries from the benchmark harvested by Lick and Schmitz [Lick & Schmitz 2022], as we will report in Chapter 5: SHA determinization fails to terminate within a timeout of 100 seconds for 37% of the queries there. Even without the timeout and with processing time extending to some hours, the resulting determinized automata for some queries still reached the order of tens of thousands of states and millions of transition rules. So the question remains open whether one can improve on this situation. What gives hope is that the schema-based cleaning of the determinization of an SHA may still be way much smaller than the determinization.

Question 2. Is EQA for regular queries feasible in practice, particularly in the application of XPATH query answering on XML streams? Can the quadratic factor

n^2 in the time upper bound per event of Gauwin’s *EQA* algorithm for dNwas be avoided?

As input, the *EQA* problem receives a stream that contains a nested word – i.e. a linearization of some hedge sent letter by letter from the left to the right – and a deterministic automaton for nested words that defines the regular query. We assume that the deterministic automaton is of a reasonable size, so the number of states n is no more than few hundreds, and the number of transition rules m at most few thousands.

Gauwin’s *EQA* algorithm requires time $O(c n^2)$ per event of the stream, where c is the concurrency of the query at the event, i.e., the number of alive candidates of the query on the event that are neither certain for selection nor rejection. In practice of regular *XPATH* queries, the concurrency c is often bounded by 2. When assuming bounded concurrency, the critical factor is the remaining $O(n^2)$. This quadratic factor comes from dNwa universality checking, under the assumption that the binary hedge accessibility relation of the dNwa has been precomputed in time $O(n^3)$. So a key question is how to reduce this quadratic factor (and may be also the cubic precomputation?).

We also note that for queries with unbounded concurrency, one should be able to apply dNwa evaluation with stack-and-state sharing from Debarbieux et al. [Debarbieux *et al.* 2015], in order to improve Gauwin’s *EQA* algorithm. Thereby, the concurrency factor c in the upper bound can be reduced to n . However, even in this case, the question remains: under which conditions can one reduce the quadratic factor n^2 ? More concretely, can we reduce the quadratic factor n^2 to m – i.e. the overall automaton size – when using dSHAS instead of dNwas as inputs? This was suggested already by the PhD thesis of Sakho [Sakho 2020], but could not be proven there. As we will show, the answer is indeed positive.

Question 3. Can *EQA* algorithms for regular monadic queries be made so efficient, that they becomes competitive with the best existing streaming algorithms for answering regular *XPATH* queries on XML streams? Given that projection is key for the efficiency of these algorithms, the question is how to obtain sufficiently powerful projection algorithms for dSHAS.

Projection means processing only the relevant parts of the input document while ignoring others, thereby reducing unnecessary computation. In-memory projection can be obtained by jumping over irrelevant parts of the input hedge. In streaming mode, projection means to only parse irrelevant parts of the input hedge and not to

send them to the evaluator. This optimization is critical, given that pure parsing is usually done two or three orders of magnitude faster than query evaluation.

The best existing streaming algorithms that answer `XPATH` queries on `XML` streams rely on projection to drastically reduce the processing time. `SAXON`'s in-memory evaluator for all of `XSLT` and thus `XPATH` must use some kind of projection given its high efficiency. However, we couldn't find information on which precise projection algorithm it applies. Projection for alternating tree automata is used for the regular `XPATH` evaluator of [Maneth & Nguyen 2010]. The `QUIXPATH` evaluator for `XML` streams, relies on subtree and descendant projection for `NWAS` [Sebastian & Niehren 2016]. The idea in the latter is to recognize loops, in which an `NWA` does not change the state, when either reading a subtree, or else moving down to some descendant. Their approach may require choices between the two strategies for subtree projection or descendant projection. However, it does not offer any guarantee that projection is complete, i.e., that *all* irrelevant parts of the stream are ignored.

When moving from `dNWAS` to `dSHAS`, one problem is that subtree projection becomes much weaker, since `dSHAS` operate in a bottom-up manner, so that the loops in which `dNWAS` don't change the state in subtrees do not exist anymore. Therefore, it is unclear how one could obtain to powerful projection methods for `dSHAS`. It is sure, though, that simple loop detection will not do the job. What is less clear is which alternative one could develop instead.

1.3 Contributions

We next present our contributions to the open challenges, on schema-based determinization, subhedge projection, and earliest query answering (*EQA*). Each contribution comes with a theoretical and a practical result.

1.3.1 Small `dSHAS` for Regular `XPATH` Queries

We contribute the first schema-based determinization algorithm for stepwise hedge automata (`SHA`). In a nutshell, it consists of integrating schema-based cleaning as presented in [Niehren & Sakho 2021] directly into the accessible determinization of the automaton. Thereby, generating the possibly exponentially larger accessible determinization as an intermediate result is avoided.

We show an upper-bound for the time needed to compute schema-based deter-

minized automata for SHAS, which is equivalent to the accessible determinized one followed by schema-cleaning. This upper bound depends quadratically on the size of the product of the accessible determinization automaton with the deterministic automaton for the schema, which, in turn, may be exponentially smaller than the accessible determinization alone.

We implemented our schema-based determinization algorithm for SHAS and tested it on the SHAS of 78 regular forward XPATH queries, that we selected from a corpus of over 21,000 queries extracted by Lick and Schmitz [Lick & Schmitz 2022] from real-world XSLT and XQUERY programs (docbook, htmlbook, teixsl, treedown, and histei). The selection aimed to capture the most complex regular forward XPATH queries in the corpus. The schema we employed integrates the XML data model and the fact that the selection variable of SHAS of the query must occur exactly once in each accepted hedge. As a result, we could successfully determinize the SHAS for 78 queries, obtaining small dSHA with at most 58 states for all queries and an overall size of at most 358. We made these dSHAS freely available in the Software Heritage archive (see <https://gitlab.inria.fr/aalserha/xpath-benchmark>), as well as the corresponding dNwas of similar sizes.

This contribution yields a positive answer to Question 1. The schema-based determinization algorithm directly produces small deterministic SHAS by avoiding large intermediate automata. The complexity result shows that this method can generate significantly smaller automata, and the implementation tested on real XPATH queries confirms its practical effectiveness. This result is game changer for the practical problem of earliest query answering of regular XPATH queries on XML streams.

1.3.2 Subhedge Projection Algorithms for dSHAS

We developed subhedge projection algorithms for dSHAS that answer Question 3, at least partially.

We start by formalizing the concept of irrelevant subhedges for regular languages. Given a language L , it identifies which subhedges of a given hedge h are not relevant for language membership $h \in L$. This notion is the foundation of a subhedge projection algorithm, that we present next. We will apply subhedge projection to earliest membership testing algorithms for regular hedge languages – corresponding to answering regular Boolean queries – and to earliest monadic query answering.

Our first algorithm for subhedge projection for dSHAS is called safe-no-change

projection. It relies on compiling dSHAS representing regular hedge languages to $d\text{SHA}^\downarrow$ s. These are an extension of SHAS with top-down processing that we introduce, necessary to identify states that allow for subhedge projection. SHA^\downarrow s for hedges are roughly like NWA for nested words [Alur 2007, Okhotin & Salomaa 2014]. Formally, SHAS are even closer to Neumann and Seidl’s pushdown forest automata [Neumann & Seidl 1998], since hedges extend on (ordered) forests by permitting unlabeled nodes.

The key idea behind the safe-no-change projection is to propagate information top-down, allowing the identification of looping states that are guaranteed to remain unchanged. Although being sound, safe-no-change projection is not complete for subhedge projection, i.e., it may not detect all irrelevant subhedges.

This leads us to the question, whether complete subhedge projection for dSHAS is possible. To formalize what that means precisely, we introduce the notion of strongly irrelevant subhedges. Only these subhedges need to be projected by a complete subhedge projection. Projecting all strongly irrelevant subhedges makes it impossible to project those that are irrelevant but not strongly irrelevant. The most technically challenging contribution of this thesis is an algorithm for dSHAS that is complete for subhedge projection: the congruence projection algorithm. It resolves the incompleteness of the safe-no-change projection, while using congruence as known from Myhill-Nerode’s theorem for NFA minimization. In the case of general dSHAS, the algorithm maintains a difference relation on the states of the input dSHA, for compiling it to a $d\text{SHA}^\downarrow$, while also depending on a regular schema.

This contribution shows that powerful subhedge projection algorithms for dSHAS are indeed achievable, providing a partial answer to Question 3. How to incorporate complete subhedge projection into EQA for regular monadic queries is studied next, as well as reporting some practical results.

1.3.3 Earliest Query Answering for dSHAS

We finally study earliest query answering for dSHAS in order to answer Question 2. In combination with complete subhedge projection, this also closes most of the open parts of Question 1 and Question 3.

We contribute a new compiler from dSHAS to earliest $d\text{SHA}^\downarrow$ s which can detect certain language membership or non-membership at the earliest possible prefix of hedge. These earliest automata enable complete suffix projection, i.e. earliest selection and earliest rejection, so that all irrelevant suffixes of the nested word of a

hedge can be ignored.

We then show how to obtain an earliest $d\text{SHA}^\downarrow$ s with complete subhedge projection for any $d\text{SHA}$. For this, we provide a general method for combining subhedge projection with suffix projection, i.e. earliest $d\text{SHA}^\downarrow$ s with $d\text{SHA}^\downarrow$ s with complete subhedge projection.

We then introduce earliest top-down evaluators for $d\text{SHA}^\downarrow$ s with subhedge projection, which can operate either in-memory or in streaming modes. Both evaluators detect certain membership upon reaching some selection state. For each non-projected event of the input hedge, they require $O(1)$ time, when assuming a potentially exponential precomputation.

Finally, we present a new earliest query answering (EQA) algorithm with full subhedge projection for monadic queries defined by $d\text{SHA}$. The algorithm operates in time $O(c)$ per non-projected event, where c is the concurrency of the query at the event of the hedge. Our algorithm efficiently supports both streaming and top-down in-memory evaluation of regular monadic queries defined by $d\text{SHA}$. It requires polynomial preprocessing time relative to the size of the earliest $d\text{SHA}^\downarrow$ with complete subhedge projection, which itself may require exponential time to compute in the worst case. This exponential preprocessing time can be avoided by computing only the needed part of the $d\text{SHA}^\downarrow$ s on the fly, at the cost of admitting $O(c \cdot m)$ time per event, where m is the overall size of the $d\text{SHA}$. In this case, the preprocessing time can be reduced to $O(n^3 d)$, where d is the number of needed difference relations, which is linearly bounded by the size of the stream.

We fully implemented the EQA algorithm for $d\text{SHAS}$ with complete subhedge and suffix projection in *ASTREAM*, a currently internal tool that we developed to test our query answering algorithms for monadic queries defined by $d\text{SHAS}$. It supports both safe-no-change and congruence projection, and is currently limited to the streaming evaluation mode.

Since we aim to answer regular *XPATH* queries on XML streams, we tested *ASTREAM* on $d\text{SHAS}$ using the *XPATHMARK* benchmark, which provides an XML document generator for scaling document sizes. This allows us to compare the performances of *ASTREAM* and *QUIXPATH* on the same XML documents and *XPATH* queries. Congruence projection proved more efficient than safe-no-change, significantly reducing runtime by projecting 75.7% to 100% of the input streams. For queries with only child axes, *ASTREAM*'s congruence projection is 1.3 to 2.9 times slower than *QUIXPATH*, but remains competitive even with descendant axes, where it is at most 13.8 times slower. We also tested *ASTREAM*'s congruence projection on the regular

queries collected from Lick and Schmitz' benchmark, on documents we gathered for this purpose. The test showed the correctness of the answer sets after comparing them to those obtained with SAXON evaluator.

1.4 Further Related Work

We describe some approaches on complex event processing, XML stream processing, and projection in more details.

1.4.1 Complex Event Processing

A complex event is a query on streams that selects tuples of events, i.e., tuples of positions of the stream. Complex event processing (CEP) is the problem of answering such a query on a stream. Most approaches to CEP [Cugola & Margara 2012] use restricted classes of queries, so that whether a tuple of events answers the query depends only on the past and on a window of bounded size of the future (following the last event of the tuple). Most CEP systems do allow for non-regular queries, which compare data values. Moreover, answers are expected to be reported with low latency, meaning the output must be produced incrementally, no later than when the processing of the window following the selected tuple is finished.

In the most extreme case, where the window size is 0, one refers to 0-delay queries [Benedikt & Jeffrey 2007]. For monadic 0-delay queries, whether an event is an answer depends solely on the past of the event, not on its future. Bounded delay queries allow windows with a bounded number of events [Gauwin *et al.* 2011]. However, there are also many other kinds of windows, for instance defined by the real time delay [Cugola & Margara 2012].

In [Mozafari *et al.* 2012], a query language for CEP over XML streams was developed and applied to various domains such as stock analysis, social networks, genetics, and more. Whether an XML element is selected by a query of this language depends only on the attributes of the event and its past, i.e., the preceding elements. The language is inspired by XPATH. It uses a syntax for NRPQs including the Kleene-star natively (unlike XPATH), and thus allowing a recursive look back into the past. They compile their queries to nondeterministic visibly pushdown automata – yet another alternative to nested word automata [Pitcher 2005, Okhotin & Salomaa 2014] – which are used for query answering.

In [Grez *et al.* 2019], a formal framework for CEP was introduced, along with a

formal language for specifying complex events and its complete semantics. Streams are lists of tuples of data values. This language permits to select tuples of events while using comparisons of data values. Selection depends only on the past of the last event bound in the tuple. In [Muñoz & Riveros 2022a], streams were generalized to nested words, and filter with tests about the future were added. In this case, the answer set can be produced only at the very end of the stream.

1.4.2 XML Stream Processing

XML stream processing has been extensively studied [Green *et al.* 2004, Koch *et al.* 2004, Olteanu 2007, Kumar *et al.* 2007, Benedikt *et al.* 2008, Kay 2010] both theoretically and practically. Many approaches are restricted to small fragments of XPATH, with a primary focus on algorithms and tools that can handle XML streams efficiently, with low memory, low time per event, and low latency. But some approaches can deal with large fragment of XPATH too.

Olteanu’s SPEX tool [Olteanu 2007], was the first to deal with all regular XPATH 1.0 queries. He propose to use transducer networks to represent and process regular XPATH queries. His tool is freely available and operational. It yields decent time and space efficiency for most queries. His algorithm approximates EQA but is not earliest in general. The idea is that all answer candidates nodes of a regular XPATH query, with only descending axis (`child` and `descendant`), become certain at closing time.

Sebastian QUIXPATH tool [Sebastian 2016, Debarbieux *et al.* 2015] also supports all regular XPATH 1.0 queries. In addition, it support some non-regular queries of XPATH 3.0 based on networks of automata. QUIXPATH approximates EQA by compilation of forward regular XPATH queries to nondeterministic Early Nwas. These are Nwas with distinguished subsets of selection and rejection states that permit to select certain answers and non-answer in an early manner.

QUIXPATH achieved the largest coverage compared to all other tools benchmarks and demonstrated the highest time efficiency since using subtree and descendant projection [Sebastian & Niehren 2016]. Queries with high concurrency can be treated in time $O(n)$ per event, where n is the number of states of the Nwa representing the query. This is possible by using stack-and-state sharing for the concurrent candidates [Debarbieux *et al.* 2015]. For queries, where QUIXPATH is not earliest, it still has high time efficiency per event, but may run out of memory for huge documents that do not fit into memory (even with low concurrency).

1.4.3 Projection for XPATH Queries

Maneth and Nguyen [Maneth & Nguyen 2010] defined notions of relevant nodes for bottom-up and top-down tree automata. These are based on the idea the automaton does not change the state on these nodes, up to equivalence class with respect to the Myhill-Nerode congruence. They then use an approximation of these notions of relevant nodes for nondeterministic alternating tree automata, in order to evaluate regular XPATH queries in-memory with subtree and descendant projection. Their experimental results show very good performances compared to XML database systems (MonetDB). Comparisons with SAXON's in-memory evaluation are not given.

The idea underlying this approach are closely related to the approach to projection applied in the present thesis. Our notion of irrelevant subhedges for SHA^\downarrow s seems to generalize on the congruence-based notion of irrelevant nodes of Maneth and Nguyen, in that SHA^\downarrow s generalize on top-down and bottom-up tree automata in a uniform framework. With the difference that we consider only subhedge projection, while they also treat descendant projection. Our safe-no-change algorithm for subhedge projection for SHAS yields an approximation of relevant subhedges based on looping states, in the same spirit as their approximation of relevant nodes for nondeterministic alternating tree automata. Our notion of congruence projection is based on the idea that irrelevant suffixes for DFAS can be discovered on the basis of the Myhill-Nerode congruence, which was already noticed in their paper. But no completeness results for subhedge projection were obtained there that could be applied to general regular XPATH queries.

Niehren and Sebastian [Sebastian & Niehren 2016] propose projection algorithms for Nwas supporting both, subtree and descendant projection. These algorithms use loop detection to skip subtrees or jump to descendants while ignoring irrelevant parts. As we do, they consider streaming evaluation, in contrast to Maneth and Nguyen who study in-memory evaluation. But similar to [Maneth & Nguyen 2010], their approach does not guarantee complete subtree projection, but has the advantage to also perform descendant projection.

Niehren et al. [Niehren *et al.* 2022b] study projection algorithms for NRPQs on graphs. They introduce a notion of top-irrelevant nodes for NRPQs and show that it can be captured by the top-down evaluation of Datalog programs compiled from the NRPQs. Even though applicable to graphs, they tested their evaluation with projection on regular XPATH queries. Moreover, they overcome some limitations of Maneth and Nguyen's. Their algorithm is not bound to trees and applies to graphs.

Furthermore, it is not limited to forward navigational `XPATH` but can treat any NRPQ also with backward axis. Finally, it was implemented efficiently without any dedicated techniques based on any top-down Datalog evaluator (such as provided by LogicBlox for instance).

Gienieczko, Murlak, and Paperman [Gienieczko *et al.* 2024] present an efficient algorithm for querying JSON streams. It supports simple NRPQs without filters, i.e., path queries with forward axis *child*, *descendant*, label test, and *wildcard* selectors. For such queries, any answer or non-answer becomes certain with 0-delay. They consider machines with *Single Instruction, Multiple Data (SIMD)* architectures. Based on vector operations of such machines, the part of the stream subject to descendant projection can be processed blockwise (using *memchr*), speeding up the parsing of the projected part up to an order of magnitude.

1.4.4 Parallelism and Streams

Stream processing can be sped up by hyperstreaming. The idea is to decompose the stream into factors that are sent and processed in parallel. Hyperstreams containing XML documents are also called XML streams with references [Maneth *et al.* 2015]. Hyperstreams containing trees are also called compressed tree patterns [Sakho *et al.* 2017].

When evaluating a factor by an automaton, however, one cannot know the state into which the preceding suffix got evaluated. Nonetheless, one can still start in all states in a speculative manner, and decide on the correct state later once the missing information becomes available. The whole stream can then be evaluated by composing the transitions of its factors.

Paperman *et al.* [Murlak *et al.* 2016] considered hyperstreams with fixed-size blocks for the problem of regular schema validation. They used circuits to evaluate blocks of size b in time $\log(b)$.

Sakho *et al.* [Sakho 2020, Sakho *et al.* 2017] studied hyperstreams of trees for the problem of earliest query answering for regular monadic queries. They considered automata techniques to evaluate the nested word factors, with the objective of reducing the output latency of certain query answers.

Maneth, Seidl, *et al.* [Maneth *et al.* 2015] studied the problem of transforming XML streams with references by top-down tree transducers with low memory costs.

1.5 Outline

This thesis is split into 4 parts:

Part I starts with preliminaries. The only new contribution there is the notion of downward stepwise hedge automata in Section 3.3.

Part II is about schema-based determinization and how to use it to obtain small deterministic automata for regular `XPATH` queries in practice.

Part III presents the new results on subhedge and suffix projection. This is the core of the thesis.

Part IV shows how to use these projection algorithms for earliest query answering on XML streams, and provides experimental results.

1.6 Publication Comments

During my PhD, I contributed to 5 publications, 4 at international conferences (of which one is of industrial character), and 1 extended to a journal article. The content of all these publications constitute the basis of the present dissertation:

- *Schema-Based Automata Determinization* [Niehren *et al.* 2022a]: This Gandalf'22 conference paper introduces our schema-based determinization algorithms for both `NFAS` and `SHAS`. It is included in Chapter 5.
- *A Benchmark Collection of Deterministic Automata for XPATH Queries* [Al Serhali & Niehren 2022]: This XML Prague'22 conference paper presents the application of our newly introduced determinization algorithm on benchmarks of `XPATH` queries, where we were able to obtain small deterministic `SHAS`. It constitutes the content of Chapter 6.
- *Complete Subhedge Projection for Stepwise Hedge Automata* [Al Serhali & Niehren 2024] and *Subhedge Projection for Stepwise Hedge Automata* [Al Serhali & Niehren 2023b]: This article in the Algorithms'24 journal was partially presented previously at the FCT'23 conference. In FCT, the safe-no-change projection algorithm was given and in Algorithms, the congruence projection algorithm was added. These contribution form the backbone of the present dissertation. They can be found in Chapters 7, 9, 10, and 12.

- *Earliest Query Answering for Deterministic Stepwise Hedge Automata* [Al Serhali & Niehren 2023a]: This conference paper presents our Earliest Query Answering (*EQA*) algorithm for $\text{dSHA}s$, which achieves a combined linear complexity of $O(c \cdot m)$ time per event. The algorithm used infinitary, on-the-fly, deterministic nested word automata (dNwAs) as streaming machines, without considering schema. In this dissertation, we now use downward stepwise hedge automata (dSHA^\downarrow s), which we construct statically, incorporating schema into the construction. This work is included in Chapters 8, 10, and 11.

Part I

Starting

We present preliminaries on formal language theory, finite state automata on words, regular hedge languages, and stepwise hedge automata. We introduce downward stepwise hedge automata, an extension of stepwise hedge automata that was proposed during the present doctoral thesis [Al Serhali & Niehren 2023b, Al Serhali & Niehren 2024], but can also be considered as a variant of nested word automata, which have been reinvented multiple times under various different names since the eighties. We finally discuss regular monadic queries on hedges and their relationship to regular `XPATH` queries on `XML` documents.

Chapter 2

Preliminaries

Abstract

We introduce the basic concepts from formal language theory and regular queries on hedges that will be needed in this dissertation. We start with the structures of words, hedges, and nested words, with particular emphasis on the graphs of hedges and their top-down traversals. Additionally, we discuss the relationship between hedges, nested words, and stream processing. We then recall how to define regular hedge languages by nested regular expressions, and explain how the latter can be used to define regular monadic queries that select nodes of hedges as well.

Contents

2.1	Mathematical Notation	22
2.2	Words, Hedges and Nested Words	22
2.2.1	Words	22
2.2.2	Hedges	23
2.2.3	Nested Words	26
2.2.4	Hedge Traversals	27
2.2.5	Nested Words Prefixes	29
2.3	Regular Languages and Queries	30
2.3.1	Regular Expressions	30
2.3.2	Nested Regular Expressions	32
2.3.3	Regular Monadic Queries	34
2.3.4	Schema Constraints for x-Annotations	37

2.1 Mathematical Notation

Let \mathbb{N} be the set of natural numbers including 0. For any set A and natural number $n \in \mathbb{N}$, we define the set of n -tuples of elements of A by:

$$A^n = \{(a_1, \dots, a_n) \mid a_1, \dots, a_n \in A\}$$

Note that $A^0 = \{()\}$ is a singleton and thus \emptyset^0 is nonempty. Let A, B be sets. If $A \subseteq B$ then the complement of A in B is denoted by

$$\overline{A} = B \setminus A$$

Note that the set relative to which the complement is taken is kept implicit when writing \overline{A} . The *domain* of a binary relation $r \subseteq A \times B$ is:

$$\text{dom}(r) = \{a \in A \mid \exists b \in B. (a, b) \in r\}$$

A *partial function* $f : A \hookrightarrow B$ is a binary relation $f \subseteq A \times B$ that is functional, i.e., for all $a \in A$ there exists at most one $b \in B$ such that $(a, b) \in f$. In this case we define $f(a) = b$. A *total function* $f : A \rightarrow B$ is a partial function $f : A \hookrightarrow B$ with $\text{dom}(f) = A$.

2.2 Words, Hedges and Nested Words

We start with recalling the definition for words, then introduce hedges and their linearizations to nested words.

2.2.1 Words

Let Σ be a finite set. The set of words with alphabet Σ is $\Sigma^* = \bigcup_{n \in \mathbb{N}} \Sigma^n$. A word $(a_1, \dots, a_n) \in \Sigma^n$ where $n \in \mathbb{N}$ is written as $a_1 \dots a_n$. We denote the empty word of length 0 by $\epsilon = () \in \Sigma^0$ and by $v_1 \cdot v_2 \in \Sigma^*$ the concatenation of two words $v_1, v_2 \in \Sigma^*$.

If $v = u \cdot v' \cdot w$ is a word, then we call u a prefix of v and v' a factor of v and w a suffix of v . Given any subset $L \subseteq \Sigma^*$, we denote the set of prefixes of words in L by $\text{pref}(L)$ and the set of suffixes of words in L by $\text{suff}(L)$.

For any subalphabet $\Sigma' \subseteq \Sigma$, the projection of a word $v \in \Sigma^*$ to Σ' is the word $\text{proj}_{\Sigma'}(v)$ in $(\Sigma')^*$ that is obtained from v by removing all letters from $\Sigma \setminus \Sigma'$.

2.2.2 Hedges

Intuitively, hedges are sequences of letters and unlabeled trees $\langle h \rangle$, that again contain some hedge h . More formally, a hedge $h \in \mathcal{H}_\Sigma$ is an equivalence class of terms with the following abstract syntax:

$$h, h' \in \mathcal{H}_\Sigma ::= a \mid \langle h \rangle \mid \epsilon \mid h \cdot h' \quad \text{where } a \in \Sigma$$

The terms are constructed from the constants $a \in \Sigma$, the unary tree constructor $\langle \rangle$, and, from a constant ϵ , and the binary composition operator \cdot . The application of the tree constructor $\langle \rangle$ to some hedge h is written as $\langle h \rangle$, while the application of the binary constructor \cdot to hedge h and h' is written in infix notation as $h \cdot h'$. We assume that $(\mathcal{H}_\Sigma, \epsilon, \cdot)$ is a monoid, i.e. that hedge composition \cdot is associative with the empty hedge ϵ as neutral element, so for all $h, h', h'' \in \mathcal{H}_\Sigma$:

$$\begin{aligned} \epsilon \cdot h &= h \cdot \epsilon = h \\ (h \cdot h') \cdot h'' &= h \cdot (h' \cdot h'') \end{aligned}$$

That means that hedges are equivalence classes of terms modulo the monoid axioms. Since \cdot is associative, we can omit parenthesis in our abstract syntax, and simply write $h \cdot h' \cdot h''$ instead of $(h \cdot h') \cdot h''$ or $h \cdot (h' \cdot h'')$.

Example 2.1. Let $\Sigma = \{a, b\}$. Then $\langle a \cdot \epsilon \cdot \langle b \cdot a \rangle \cdot \langle a \cdot b \rangle \rangle \cdot a \cdot \langle a \rangle$ is a hedge in \mathcal{H}_Σ that is equal to the hedge $\langle a \cdot \langle b \cdot a \rangle \cdot \langle a \cdot b \rangle \rangle \cdot a \cdot \langle a \rangle$

Words are different from hedges. The only words that are also hedges are the empty word ϵ and the single letter words $a \in \Sigma$. Still any word in Σ^* can be mapped to some hedge in \mathcal{H}_Σ by adding the composition operator \cdot between the letters. The word $w = aab \in \Sigma^*$, for instance, can be mapped to the hedge $hdg(w) = a \cdot a \cdot b \in \mathcal{H}_\Sigma$.

Note that the composition operator \cdot for hedge construction is different from the concatenation operator \cdot for words, which is typeset in bold face to make this apparent. As a consequence, $a \cdot a \cdot b = hdg(aab)$ is a hedge in \mathcal{H}_Σ , whereas $a \cdot a \cdot b = aab$ is a word in Σ^* .

Given a subset $\Sigma' \subseteq \Sigma$ and a hedge $h \in \mathcal{H}_\Sigma$, we define the projection $\Pi_{\Sigma'}(h) \in \mathcal{H}_{\Sigma'}$ as the hedge obtained by removing all letter outside Σ' from h .

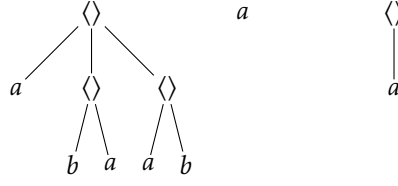


Figure 2.1: A graphical representation of the hedge $\langle a \cdot \epsilon \cdot \langle b \cdot a \rangle \cdot \langle a \cdot b \rangle \rangle \cdot a \cdot \langle a \rangle$.

2.2.2.1 Hedge Sizes

We define the size $|h|$ of a hedge as its term size, while ignoring the constructors \cdot and ϵ . The size can be defined recursively over the structure of hedges such that for all $h, h' \in \mathcal{H}_\Sigma$ and $a \in \Sigma$:

$$\begin{aligned} |\epsilon| &= 0 & |a| &= 1 \\ |h \cdot h'| &= |h| + |h'| & |\langle h \rangle| &= 1 + |h| \end{aligned}$$

Example 2.2. The hedge $h = \langle a \cdot \epsilon \cdot \langle b \cdot a \rangle \cdot \langle a \cdot b \rangle \rangle \cdot a \cdot \langle a \rangle$ from Figure 2.1 has size $|h| = 11$.

2.2.2.2 Hedge Nodes

Since any hedge is a term modulo the monoid axioms, it can be represented by some directed acyclic graph (DAG).

Example 2.3. A graphical representation of the hedge $\langle a \cdot \epsilon \cdot \langle b \cdot a \rangle \cdot \langle a \cdot b \rangle \rangle \cdot a \cdot \langle a \rangle$, which that is equal to $\langle a \cdot \langle b \cdot a \rangle \cdot \langle a \cdot b \rangle \rangle \cdot a \cdot \langle a \rangle$ by the monoid axioms, is given in Figure 2.1. Note that the operators \cdot and ϵ are not shown there explicitly. They impose an ordering on the nodes of any subhedge of the hedge, which we drawn by placing them from the left to the right.

A node of a hedge in the graphical representation is an occurrence of a hedge constructor in $\Sigma \cup \{\langle \rangle\}$. We can identify any node of a hedge $h \in \mathcal{H}_\Sigma$ by a natural number, numbering the occurrences of hedge constructors in $\Sigma \cup \{\langle \rangle\}$ from 1 to $|h|$. So the set of all nodes of a hedge h is:

$$\text{nod}(h) = \{1, \dots, |h|\}$$

Example 2.4. The hedge $h = \langle a \cdot \epsilon \cdot \langle b \cdot a \rangle \cdot \langle a \cdot b \rangle \rangle \cdot a \cdot \langle a \rangle$ from Figure 2.1, has the node set $\text{nod}(h) = \{1, \dots, 11\}$.

Various alternative naming schemes for nodes of trees or hedges were proposed in the literature, such as Dewey notation for paths addressing nodes. We subscribe

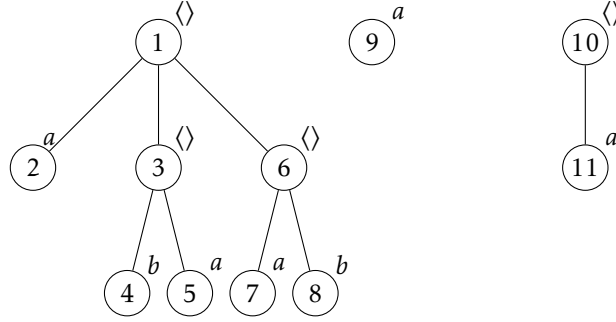


Figure 2.2: Illustration of the graph of the hedge $\langle a \cdot \epsilon \cdot \langle b \cdot a \rangle \cdot \langle a \cdot b \rangle \rangle \cdot a \cdot \langle a \rangle \cdot$

$$\begin{aligned}
 V = \text{nod}(h) &= \{1, \dots, 11\} \\
 E = \text{ch}^h &= \{(1, 2), (1, 3), (1, 6), (3, 4), (3, 5), (6, 7), (6, 8), (10, 11)\} \\
 \text{ns}^h &= \{(1, 9), (9, 10), (2, 3), (3, 6), (4, 5), (7, 8)\} \\
 \text{lab}^h &= [1/\langle \rangle, 2/a, 3/\langle \rangle, 4/b, 5/a, 6/\langle \rangle, 7/a, 8/b, 9/a, 10/\langle \rangle, 11/a]
 \end{aligned}$$

Figure 2.3: The graph $(V, E, \text{lab}^h, \text{ns}^h)$ of the hedge h from Figure 2.2.

to the identification of nodes by integers. This has the advantage that each node can be represented with logarithmic space in the size of the hedge, while Dewey addresses would require linear space. As a consequence, any subset of nodes of a hedge of size $|h|$ can be represented by an array of size $|h|$. With the Dewey notation, a representation of size $O(|h|^2)$ may become necessary.

The edges of the DAG of a hedge correspond to the child relation on the hedge's nodes. Note that DAGs of hedges are always sharing-free, so none of the nodes may have more than one incoming edge. The DAG may have several roots but these are totally ordered. The left-most root is thus unique if it exists.

Example 2.5. *The example hedge in Figure 2.1 has three roots, which are the nodes $1 < 9 < 10$. The left-most and right-most root are labeled by the tree constructor $\langle \rangle$ and the root in the middle is labeled by letter a .*

2.2.2.3 Graphs of Hedges

The graphical representation of a hedge can be turned into a formal definition of the graph of a hedge, which in turn induces a logical structure. We next explain how this can be done, while still leaving parts of the construction informal. We continue to identify the nodes of a hedge h by the natural numbers in $\text{nod}(h)$.

Example 2.6. *For illustration, the node identifiers of the example hedge*

$\langle a \cdot \epsilon \cdot \langle b \cdot a \rangle \cdot \langle a \cdot b \rangle \rangle \cdot a \cdot \langle a \rangle$ from Figure 2.1 are added to the graph in Figure 2.2. One can see there that the example hedge has indeed three roots $1 < 9 < 10$.

The edges of the graph are directed and drawn top-down. They represent the child relation between the nodes of the hedge h that we denote by ch^h . For each node, the children are ordered from the left to the right. Also the roots of the hedge are ordered from the left to the right. The set of single steps in this ordering is called the next-sibling relation and denoted by ns^h . So the graph of a hedge h is a tuple (V, E, lab^h, ns^h) where $V = nod(h)$, $E = ch^h \subseteq V \times V$, $lab^h : V \rightarrow \Sigma \cup \{\langle \rangle\}$, and $ns^h \subseteq V \times V$. The elements of $V = nod(h)$ are the nodes of the hedge, the elements of E the edges for the child relation on nodes ch^h , $lab^h : V \rightarrow \Sigma \cup \{\langle \rangle\}$ is the node labeling, and ns^h is the next sibling relation, that is compatible with the total ordering on the nodes.

2.2.2.4 Logical Structures and Queries for Hedge Graphs

The graph of a hedge $h \in \mathcal{H}_\Sigma$ induces a relational structure with domain $nod(h)$ and relational signature $\Gamma = \{lab_a \mid a \in \Sigma \cup \{\langle \rangle\}\} \cup \{ch, ns\}$. The arity of all relation symbols lab_a is 1 where $a \in \Sigma \cup \{\langle \rangle\}$ and the arities of ch and ns are both 2. For any hedge h , the logical structure of h interprets the monadic relation symbols lab_a as the set of a -labeled nodes $\{\pi \in nod(h) \mid lab^h(\pi) = a\}$ and the binary relation symbols ch and ns by the relations ch^h and respectively ns^h .

The view of hedge graphs as relational structures permits us to define n -ary node selection queries on hedges in \mathcal{H}_Σ by logical formulas with n -free variables and relation symbols in $\Gamma = \{lab_a \mid a \in \Sigma \cup \{\langle \rangle\}\} \cup \{ch, ns\}$.

Example 2.7. Consider the signature $\Sigma = \{a, b\}$. The formula below with the single free variable x selects all a -nodes that have some next sibling y with some child z :

$$\exists y. \exists z. (lab_a(x) \wedge ns(x, y) \wedge ch(y, z))$$

This answer set of this monadic node selection query on the graph of the hedge from Figure 2.3 is the subset of nodes $\{2, 9\}$.

2.2.3 Nested Words

A nested word over Σ is a word over the alphabet $\hat{\Sigma} = \Sigma \cup \{\langle, \rangle\}$ in which all parentheses are well-nested. Intuitively, this means that for any opening parenthesis there is a matching closing parenthesis and vice versa. An example of a nested

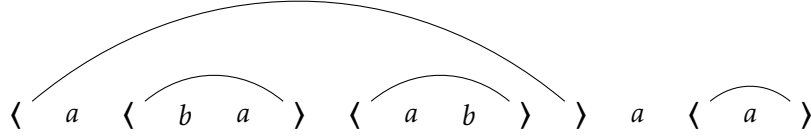


Figure 2.4: The nested word $nw(h) = \langle a \langle ba \rangle \langle ab \rangle \rangle a \langle a \rangle$ linearizing the hedge $h = \langle a \cdot \epsilon \cdot \langle b \cdot a \rangle \cdot \langle a \cdot b \rangle \rangle \cdot a \cdot \langle a \rangle$ with the matching relation of the parenthesis, for illustrating its well-nestedness.

word with the matching relation on parenthesis, illustrating its well-nestedness, is given Figure 2.4.

The set of nested words over Σ can be defined as the subsets of words over $\hat{\Sigma}$ that are a linearization of some hedge, where the linearization function $nw : \mathcal{H}_{\Sigma} \rightarrow (\Sigma \cup \{\langle, \rangle\})^*$ is defined by induction on the structure of h such that for all $h, h' \in \mathcal{H}_{\Sigma}$ and $a \in \Sigma$:

$$\begin{aligned} nw(\epsilon) &= \epsilon & nw(\langle h \rangle) &= \langle \cdot nw(h) \cdot \rangle \\ nw(a) &= a & nw(h \cdot h') &= nw(h) \cdot nw(h') \end{aligned}$$

So the set of all nested words over Σ is:

$$\mathcal{N}_{\Sigma} = nw(\mathcal{H}_{\Sigma})$$

Example 2.8. For our example hedge $h = \langle a \cdot \epsilon \cdot \langle b \cdot a \rangle \cdot \langle a \cdot b \rangle \rangle \cdot a \cdot \langle a \rangle$, we obtain the nested word $nw(h) = \langle a \langle ba \rangle \langle ab \rangle \rangle a \langle a \rangle$ illustrated in Figure 2.4.

Let hdg be the inverse of the injective function nw restricted to its image, so the mapping from nested words to hedges with $hdg(nw(h)) = h$ for all $h \in \mathcal{H}_{\Sigma}$.

2.2.4 Hedge Traversals

A depth-first search on the graph of our example hedge is illustrated in Figure 2.5. Each node labeled $\langle \rangle$ is visited twice, one when entering the subtree and once when leaving it. Labels in leafs are visited only once. The visits are also called events. These are totally ordered. In our example in Figure 2.5 they range from 1 to 15, in green for opening events, red for closing events and yellow for letter events.

We note that the preorder of the depth-first search is equal to the total ordering of the nodes:

$$1 < 2 < 3 < 4 < 5 < 6 < 7 < 8 < 9 < 10 < 11$$

In other words, if the first visits of a node π precedes the first visit of a node π'

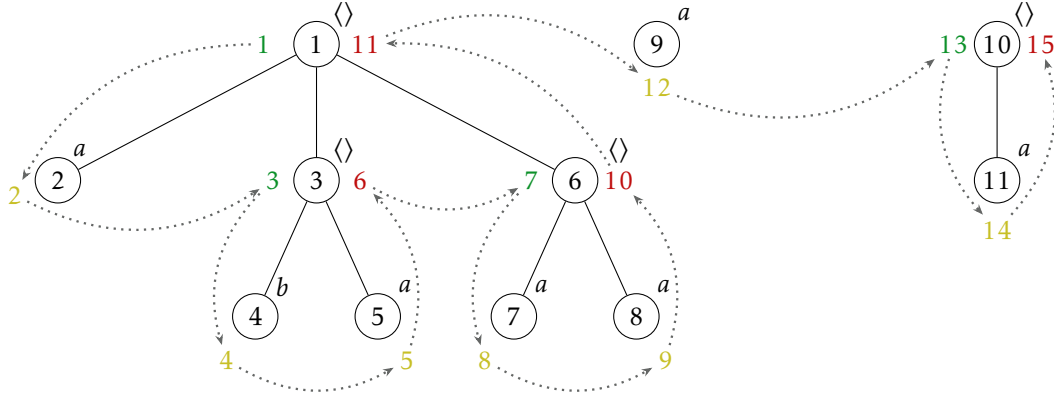


Figure 2.5: The depth-first search of the graph of the hedge $\langle a \cdot \langle b \cdot a \rangle \cdot \langle a \cdot b \rangle \rangle \cdot a \cdot \langle a \rangle$ illustrated in Figure 2.2.

then $\pi < \pi'$.

A *traversal* of a hedge is a list of nodes that contains all the nodes of the hedge at least once. We will see lists as words. The *top-down traversal* of a hedge is the list of its nodes obtained when mapping each event in the list of events to the node that it affects. The top-down traversal of our example hedge, for instance, is:

1 2 3 4 5 3 6 7 8 6 1 9 10 11 10

The node 1, for instance, occurs twice in this top-down traversal, since it is the node of both events 1 and 8. For this reason the top-down traversal is not an order.

We note that the top-down traversal of a hedge can also be obtained by mapping the letters of its nested word to the corresponding node of the hedge. In the running example, this can be illustrated as follows:

\langle	a	\langle	b	a	\rangle	\langle	a	b	\rangle	\rangle	a	\langle	a	\rangle
1	2	3	4	5	3	6	7	8	6	1	9	10	11	10

This means that a top-down traversal lists all the nodes of a hedge in the ordering in which they are met when reading the hedge in streaming mode. But, of course, streaming is not needed: top-down traversals of hedge graphs can also be computed from the graph of a hedge when stored in memory.

The depth-first search of a hedge also induce a postorder on the nodes of the hedge, which depends on the order of the last visiting events of the nodes. In our

example hedge, the postorder is:

$$2 < 4 < 5 < 3 < 7 < 8 < 6 < 1 < 9 < 11 < 10$$

The *left-most bottom-up traversal* of a hedge is the list of all states in postorder. For instance, the bottom-up traversal of our running example is:

$$2\ 4\ 5\ 3\ 7\ 8\ 6\ 1\ 9\ 11\ 10$$

It should be noticed that the left-most-bottom up traversal is contained in the top-down traversal for any hedge. In our example, we have to remove all elements from the top-down traversal coming from opening events, while we have to keep those coming from closing and letter events. The latter are highlighted in boldface below:

$$1\ 2\ 3\ 4\ 5\ 3\ 6\ 7\ 8\ 6\ 1\ 9\ 10\ \mathbf{11}\ \mathbf{10}$$

2.2.5 Nested Words Prefixes

Prefixes, suffixes, and factors of nested words may not be nested words themselves. For instance, the hedge $h = a \cdot \langle b \cdot c \rangle$ has the linearization $nw(h) = a\langle bc \rangle$. Its prefix $a\langle b$ is not well-nested since it has a dangling opening parenthesis. Neither its suffix $c \rangle$ is well-nested since it has a dangling closing parenthesis.

Any algorithm that traverses a hedge h top-down inspects all the prefixes of $nw(h)$. The end of any prefix v of $nw(h)$ locates some position in the top-down traversal of the graph of h , i.e., a position of the nested word of h , or equivalently an event of the depth-first search of the graph of h . Furthermore, the prefix specifies the part of the hedge that is located before this *event* in the linearization $nw(h)$. An example is illustrated graphically in Figure 2.6.

This particularly holds for streaming algorithms that receive the nested word $nw(h)$ as an input on a stream, and may be inspected only once from the left to the right. But it holds equally for in-memory algorithms that receive the input hedge h as a hierarchical structure whose graph is stored in-memory, and then traverses the hedge's graph top-down. Any $\langle \rangle$ node will then be visited twice, once when reading an opening parenthesis \langle and going down into a subhedge, and another time when going up to the closing parenthesis \rangle after having processed the subhedge.

The set of positions $pos(w)$ of a nested word prefix $w \in \text{prefs}(\mathcal{N}_\Sigma)$ is $\{1, \dots, k\}$ where k is the number of occurrences in w of symbols from $\Sigma \cup \{\langle \rangle\}$. Note that for any

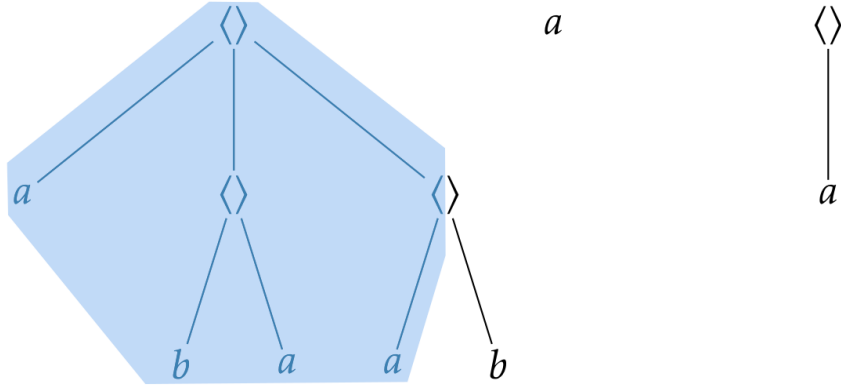


Figure 2.6: The part of the hedge from the nested word prefix $\langle a \langle ba \rangle \langle a \rangle$.

nested word $w \in \mathcal{N}_\Sigma$ we have $\text{pos}(w) = \text{nod}(\text{hdg}(w))$.

2.3 Regular Languages and Queries

We show how to define regular languages and queries by regular expressions. We are interested in boolean queries for hedges, i.e., in hedge languages, and in monadic queries for hedges that select nodes in hedges.

2.3.1 Regular Expressions

Let Σ be a set. A *language* $L \subseteq \Sigma^*$ is a subset of words with alphabet Σ . Finite languages are always regular, while infinite languages may be non-regular. For instance consider the alphabet $\Sigma = \{a, b\}$.

- The finite language $\Sigma^2 = \{aa, ab, ba, bb\}$ is regular. It contains all words of length exactly 2.
- The infinite language $\{a^n b^n \mid n \geq 0\}$ is non-regular. For each natural number n it contains the word $a^n b^n$. So this language is equal to the set $\{\epsilon, ab, aabb, aaabbb, \dots\}$.

A language $L \subseteq \Sigma^*$ is regular if it can be defined by some regular expression, where a *regular expression* is a term $e \in \text{RegExp}_\Sigma$ with the following abstract syntax:

$$e, e' \in \text{RegExp}_\Sigma ::= \epsilon \mid \emptyset \mid a \mid e \cdot e' \mid e + e' \mid e^* \quad \text{where } a \in \Sigma \\ \mid e \& e' \mid \bar{e}$$

The terms $e \in \text{RegExp}_\Sigma$ are constructed from the constants ϵ , \emptyset , and $a \in \Sigma$, the binary concatenation operator \cdot , a binary union operator $+$, and the unary repetition operator $*$, that is usually called the Kleene star. Furthermore, we admit a binary intersection operator $\&$ and a unary complement operator $\bar{}$ which are known not to add expressiveness. We include them nevertheless, since they will make it easier to define regular queries.

Semantically, any regular expression $e \in \text{RegExp}_\Sigma$ defines a language $\llbracket e \rrbracket \subseteq \Sigma^*$. The definition is by induction on the structure of regular expressions such that for all $a \in \Sigma$ and $e, e' \in \text{RegExp}_\Sigma$:

$$\begin{array}{ll} \llbracket \emptyset \rrbracket = \{\} & \llbracket e + e' \rrbracket = \llbracket e \rrbracket \cup \llbracket e' \rrbracket \\ \llbracket \epsilon \rrbracket = \{\epsilon\} & \llbracket e^* \rrbracket = \llbracket e \rrbracket^* \\ \llbracket a \rrbracket = \{a\} & \llbracket e \& e' \rrbracket = \llbracket e \rrbracket \cap \llbracket e' \rrbracket \\ \llbracket e \cdot e' \rrbracket = \llbracket e \rrbracket \cdot \llbracket e' \rrbracket & \llbracket \bar{e} \rrbracket = \Sigma^* \setminus \llbracket e \rrbracket \end{array}$$

It should also be noticed that some of the operators could be removed due to redundancy. For example, the empty set constant can be eliminated since the complement operator is available. Let $\Sigma = \{a_1, \dots, a_n\}$, then $\Sigma^* = \llbracket (a_1 + \dots + a_n)^* \rrbracket$ so that $\llbracket \emptyset \rrbracket = \overline{\llbracket (a_1 + \dots + a_n)^* \rrbracket}$. Conversely, the complement operator can be removed – by a detour via the complementation of finite state automata – but only if the empty set constant remains present. Furthermore, the intersection is redundant in the presence of complement, since $\llbracket e \& e' \rrbracket = \overline{\llbracket \bar{e} + \bar{e}' \rrbracket}$.

Definition 2.9. A language $L \subseteq \Sigma^*$ is regular if there exists a regular expression $e \in \text{RegExp}_\Sigma$ such that $L = \llbracket e \rrbracket$.

The class of all regular languages can be built from the finite languages by using the boolean operators union and concatenation and the Kleene star. A proof that the language $\{(ab)^n \mid n \geq 0\}$ is non-regular can be found in any standard text books on theoretical computer science. It can be based on a pumping Lemma for finite state automata and the fact that these capture regular languages.

Star-free regular languages are regular languages that can be defined by some expression in RegExp_Σ without the Kleene star. For this definition, the availability of the complement operator is essential. For instance, if $\Sigma = \{a, b\}$, then the language $\llbracket a^* \rrbracket$ is star-free since it is equal to the language $\llbracket \overline{\emptyset \cdot b \cdot \emptyset} \rrbracket$. In contrast, the language $\llbracket (a \cdot a)^* \rrbracket$ is known not to be star-free, since counting modulo 2 is needed for testing membership. Still, how to test whether a regular language is star-free is not obvious.

For instance, the language $\llbracket (a \cdot b)^* \rrbracket$ is again star-free since it is equal to:

$$\llbracket \epsilon + (a \cdot b \cdot \bar{\emptyset} \cap \bar{\emptyset} \cdot a \cdot b \cap \overline{\bar{\emptyset} \cdot a \cdot a \cdot \bar{\emptyset}} \cap \overline{\bar{\emptyset} \cdot b \cdot b \cdot \bar{\emptyset}}) \rrbracket$$

So, for testing membership to this language of a nonempty word, it is sufficient to test locally whether it starts and ends with ab , and that it does not contain aa nor bb somewhere in the middle. Counting is not needed, local tests are sufficient. We note that intersections of star-free languages are always star-free by definition, even though they may not be locally testable. An example is $\llbracket (a \cdot b)^* \cap (b \cdot a)^* \rrbracket$.

A classic result in formal language theory is the equivalence of star-free regular languages to non-counting or aperiodic regular languages, and to languages definable in first-order logic [McNaughton & Papert 1971], with the successor predicate on positions of the word but without the order predicate.

2.3.2 Nested Regular Expressions

We recall *nested regular expression* ($nRegExp_\Sigma$) [Hosoya & Pierce 2003] for defining regular hedge languages in the same way regular expressions define regular languages of words. For this, we follow the presentation and choices from [Niehren & Sakho 2021] rather than the original definition. We would like to note that similar regular expressions for defining regular languages of ranked trees were already used in the eighties [Gécseg & Steinby 1984].

Beside the signature Σ , we fix a set V of recursion variables and assume that it is disjoint from Σ . A nested regular expression $e \in nRegExp_\Sigma$ is then a term with the following abstract syntax:

$$\begin{aligned} e, e' \in nRegExp_\Sigma \quad ::= \quad & \epsilon \mid \emptyset \mid a \mid e \cdot e' \mid e + e' \mid e^* \quad \text{where } a \in \Sigma \\ & \mid e \& e' \mid \bar{e} \\ & \mid \langle e \rangle \mid z \mid \mu z. e \quad \text{where } z \in V \end{aligned}$$

The first two lines are the same as for regular expressions. The third line provides the application of a unary constructor for unlabeled trees $\langle . \rangle$, and a μ -operator for vertical recursion, that binds a recursion variable $z \in V$. We adopt the usual restriction that in any nested regular expression $\mu z. e$, all occurrences of z in e must be in some subexpression $\langle e' \rangle$ of e , i.e., any free occurrence of z must be guarded by some tree constructor.

Any $e \in nRegExp_\Sigma$ represents a hedge language $\llbracket e \rrbracket \subseteq \mathcal{H}_{\Sigma \cup V}$. We define it by induction on the structure of nested regular expressions such that for all $a \in \Sigma$,

$e, e' \in nRegExp_{\Sigma}$ and $z \in V$:

$$\begin{array}{lll}
\llbracket \emptyset \rrbracket = \{\} & \llbracket e + e' \rrbracket = \llbracket e \rrbracket \cup \llbracket e' \rrbracket & \llbracket z \rrbracket = \{z\} \\
\llbracket \epsilon \rrbracket = \{\epsilon\} & \llbracket e^* \rrbracket = \llbracket e \rrbracket^* & \llbracket \langle e \rangle \rrbracket = \langle \llbracket e \rrbracket \rangle \\
\llbracket a \rrbracket = \{a\} & \llbracket e \& e' \rrbracket = \llbracket e \rrbracket \cap \llbracket e' \rrbracket & \llbracket \mu z. e \rrbracket = \bigcup_{n \geq 0} \llbracket \mu^n z. e \rrbracket \\
\llbracket e \cdot e' \rrbracket = \llbracket e \rrbracket \cdot \llbracket e' \rrbracket & \llbracket \bar{e} \rrbracket = \Sigma^* \setminus \llbracket e \rrbracket &
\end{array}$$

The first two columns are literally the same as for the regular expressions. In the last column, the tree constructor is treated without surprise. For the μ -operator, we have to define iterations $\mu^n z. e$. We set:

$$\begin{array}{ll}
\mu^0 z. e = \emptyset & \text{zero iterations} \\
\mu^n z. e = e[z/\mu^{n-1} z. e] & n \text{ iterations where } n \geq 1
\end{array}$$

For the language of a μ -expression $\mu z. e$, we have to replace the recursion variable z in e repeatedly by e until eventually replacing it by \emptyset . For instance, consider the μ -expression $\mu z. (a + \langle z \rangle)$. We then have $\mu^0 z. e = \emptyset$, $\mu^1 z. e = a$, $\mu^2 z. e = a + \langle a \rangle$, $\mu^3 z. e = a + \langle a + \langle a \rangle \rangle$, etc.

Note that for μ -expressions like $\mu z. (a \cdot z \cdot b + \epsilon)$, this semantic definition would lead to non-regular languages. Fortunately, this μ -expression is forbidden, since the bound variable z does not have free occurrences that are not guarded by any tree constructor.

Example 2.10. We can model nested lists as hedges over the signature $\Sigma = \{\text{list}, \text{item}\}$. For instance, the following hedge in \mathcal{H}_{Σ} is a nested list:

$$\langle \text{list} \cdot \langle \text{item} \rangle \cdot \langle \text{list} \cdot \langle \text{item} \rangle \cdot \langle \text{item} \rangle \rangle \cdot \langle \text{item} \rangle \rangle$$

More generally, the set of all nested lists can be defined by the nested regular expression N-List as follows:

$$\begin{array}{ll}
\text{N-List} & =_{\text{def}} \text{Tree}^* \\
\text{Tree} & =_{\text{def}} \mu z. \langle (\text{list} + \text{item}) \cdot z^* \rangle
\end{array}$$

Note that $\llbracket \emptyset \rrbracket^* = \{\epsilon\}$ and that $\mu^1 z. \langle (\text{list} + \text{item}) \cdot z^* \rangle = \langle (\text{list} + \text{item}) \cdot \emptyset^* \rangle$. Hence,

$$\llbracket \mu^1 z. \langle (\text{list} + \text{item}) \cdot z^* \rangle \rrbracket = \llbracket \langle (\text{list} + \text{item}) \rangle \rrbracket = \{\langle \text{list} \rangle, \langle \text{item} \rangle\}$$

Therefore, the set $\llbracket \text{Tree} \rrbracket$ of all trees that are nested lists is nonempty.

The nested regular expression N-List will serve us in the running examples of

the next chapters and parts. This is since XML documents are essentially nested lists with data values, where `list` corresponds to XML's element constructor and `item` to XML's text node constructor.

2.3.3 Regular Monadic Queries

We start defining what are boolean and monadic queries on hedges and then discuss what it means for a query to be regular.

2.3.3.1 Boolean Queries

A Boolean query on hedges in \mathcal{H}_Σ is nothing else than a hedge language $L \subseteq \mathcal{H}_\Sigma$. So any schema is itself a Boolean query.

Definition 2.11. *Let $S \subseteq \mathcal{H}_\Sigma$. A boolean query with schema S is a language $L \subseteq S$.*

For instance, $\llbracket Tree \rrbracket$ is a boolean query with schema $\llbracket N-List \rrbracket$. The boolean query answering problem for schema S then inputs a hedge $h \in S$ and a Boolean query $L \subseteq S$ and returns the truth value of $h \in L$. The same problem is also called the language membership problem for hedges with schema S .

A boolean query is called regular if it can be defined by some nested regular expression. So $\llbracket Tree \rrbracket$ is a regular Boolean query with schema $\llbracket N-List \rrbracket$.

2.3.3.2 Monadic Queries

Monadic queries select nodes in hedges satisfying the schema. More formally:

Definition 2.12. *Let $S \subseteq \mathcal{H}_\Sigma$. A monadic query with schema S is a function $Q : S \rightarrow 2^{\mathbb{N}}$ mapping any hedge $h \in S$ to some subset of nodes $Q(h) \subseteq \text{nod}(h)$.*

For any hedge $h \in S$, the set $Q(h)$ is called the *answer set* of Q on h .

Example 2.13. *For instance, consider the query Q that mimics the XPATH query*

`descendant-or-self::list[child::item]`

on nested lists in $\llbracket N-List \rrbracket$, where it selects all `list` nodes with at least one `item` child. On the nested list

$h = \langle \text{list} \cdot \langle \text{item} \rangle \cdot \langle \text{list} \cdot \langle \text{list} \rangle \cdot \langle \text{item} \rangle \cdot \langle \text{item} \rangle \rangle \cdot \langle \text{list} \rangle \rangle$

for instance, it has the answer set:

$$Q(h) = \{2, 6\}$$

In order to specify regular monadic queries on hedges, we can identify them with languages of annotated hedges. For this, we need to introduce annotations of hedges first.

2.3.3.3 Annotating Hedges with Node Identifiers

For any $h \in \mathcal{H}_\Sigma$, we want to define $ann-nod(h) \in \mathcal{H}_{\Sigma \cup nod(h)}$ as its annotation with node identifiers. For this, we define more generally for any $n \in \mathbb{N}$ a hedge $ann-nod_n(h) \in \mathcal{H}_{\Sigma \cup \{n, \dots, n+|h|-1\}}$ where the annotation starts with n . The definition is by induction on the structure of h , so that for all $a \in \Sigma$, $n \in \mathbb{N}$, and $h, h' \in \mathcal{H}_\Sigma$:

$$\begin{aligned} ann-nod_n(\epsilon) &= \epsilon \\ ann-nod_n(a) &= a \cdot n \\ ann-nod_n(h \cdot h') &= ann-nod_n(h) \cdot ann-nod_{n+|h|}(h') \\ ann-nod_n(\langle h \rangle) &= \langle n \cdot ann-nod_{n+1}(h) \rangle \end{aligned}$$

Since the identifier of the left-most root of h is 1, we define the annotation of h by:

$$ann-nod(h) = ann-nod_1(h)$$

For instance:

$$ann-nod(a \cdot a \cdot \langle \rangle \cdot a) = a \cdot 1 \cdot a \cdot 2 \cdot \langle 3 \rangle \cdot a \cdot 4$$

2.3.3.4 Regular Monadic Queries

We next relate monadic queries on hedges to hedge languages. For this, we fix a selection variable $x \notin \Sigma$ arbitrarily and consider hedge languages over signature $\Sigma^x = \Sigma \cup \{x\}$.

For any variable assignment $\alpha : \{x\} \hookrightarrow nod(h)$, we define the hedge $h * \alpha \in \mathcal{H}_{\Sigma^x}$ annotated with x by substituting in $ann-nod(h)$ the node $\alpha(x)$ by x and removing all other node annotations. That is:

$$h * \alpha = ann-nod(h)[\alpha(x)/x][\pi/\epsilon \mid \pi \neq \alpha(x), \pi \in \mathbb{N}]$$

Example 2.14. For instance, if $h = a \cdot a \cdot \langle \rangle \cdot a$ then $h * [x/2] = a \cdot a \cdot x \cdot \langle \rangle \cdot a$ and $h * [] = h$.

We note that all hedges of the form $h * [x/\pi]$, with $\pi \in \mathbb{N}$, contain a single occurrence of x that was inserted after node π . Such structures are also called V -structures where $V = \{x\}$ (see e.g [Straubing 1994]). The set of all $\{x\}$ -structures with schema $\mathbf{S} = \mathcal{H}_\Sigma$ can be defined the following nested regular expression One- x :

$$\begin{aligned} \text{One-}x &=_{\text{def}} \mu z. (\text{Zero-}x \cdot (x + \langle z \rangle) \cdot \text{Zero-}x) \\ \text{Zero-}x &=_{\text{def}} \mu z. (+_{a \in \Sigma} a + \langle z \rangle)^* \end{aligned}$$

For any schema $\mathbf{S} \subseteq \mathcal{H}_\Sigma$ we define a schema of $\{x\}$ -structures $x\text{-str}(\mathbf{S}) \subseteq \mathcal{H}_{\Sigma^x}$ by:

$$x\text{-str}(\mathbf{S}) = \{h * [\pi/x] \mid h \in \mathbf{S}, \pi \in \text{nod}(h)\}$$

Clearly $x\text{-str}(\mathcal{H}_\Sigma) = \llbracket \text{One-}x \rrbracket$. The monadic query with schema $\mathbf{S} \subseteq \mathcal{H}_\Sigma$ defined by a language $L \subseteq \mathcal{H}_{\Sigma^x}$ is the function $\text{qry}_\mathbf{S}(L) : \mathbf{S} \rightarrow 2^\mathbb{N}$ such that for all hedges $h \in \mathbf{S}$:

$$\text{qry}_\mathbf{S}(L)(h) = \{\pi \mid \pi \in \text{nod}(h), h * [x/\pi] \in L\}$$

For any $L \subseteq \mathcal{H}_{\Sigma^x}$, note that $\text{qry}_\mathbf{S}(L) = \text{qry}_\mathbf{S}(L \cap \llbracket \text{One-}x \rrbracket) = \text{qry}_\mathbf{S}(L \cap x\text{-str}(\mathbf{S}))$. So the languages L and $L \cap x\text{-str}(\mathbf{S})$ define the same query with schema \mathbf{S} . This means that only the $\{x\}$ -structures in L are relevant for the query $\text{qry}_\mathbf{S}(L)$. Furthermore, only the x -annotations of hedges in \mathbf{S} in L matter for the query.

Definition 2.15. A monadic query $\mathbf{Q} : \mathbf{S} \rightarrow 2^\mathbb{N}$ on a subset of hedges $\mathbf{S} \subseteq \mathcal{H}_\Sigma$ is called *regular* if \mathbf{S} is regular and there exists a nested regular expression $e \in n\text{RegExp}_{\Sigma^x}$ such that $\mathbf{Q} = \text{qry}_\mathbf{S}(\llbracket e \rrbracket)$.

Example 2.16. In Example 2.13, we considered the monadic query \mathbf{Q} on nested lists that mimics the XPATH query `self::list[child::item]` on nested lists, where it selects all `list`-labeled nodes with at least one child labeled `item`. This query has the schema $\mathbf{S} = \llbracket \text{N-List} \rrbracket$ and the signature $\Sigma = \{\text{list}, \text{item}\}$. For instance, on the nested list

$$h = \langle \text{list} \cdot \langle \text{item} \rangle \cdot \langle \text{list} \cdot \langle \text{list} \rangle \cdot \langle \text{item} \rangle \cdot \langle \text{item} \rangle \rangle \cdot \langle \text{list} \rangle \rangle$$

the query selects the set of nodes $\mathbf{Q}(h) = \{2, 6\}$. Query \mathbf{Q} is regular, since it can be defined by the following nested regular expression $\text{child}^*\text{-list}[\text{child-item}] \in n\text{RegExp}_{\Sigma^x}$ where:

$$\text{child}^*\text{-list}[\text{child-item}] =_{\text{df}} \mu z. \left(\begin{array}{c} \langle \text{list} \cdot x \cdot \text{N-List} \cdot \langle \text{item} \rangle \cdot \text{N-List} \rangle \\ + \\ \langle (\text{list} + \text{item}) \cdot \text{N-List} \cdot z \cdot \text{N-List} \rangle \end{array} \right)$$

This means that $\text{qry}_{\llbracket \text{N-List} \rrbracket}(\llbracket \text{child}^*\text{-list}[\text{child-item}] \rrbracket) = \mathbf{Q}$. For instance, the following two x -annotations of hedge h belong to the language $\llbracket \text{child}^*\text{-list}[\text{child-item}] \rrbracket$ defining

query Q :

$$\begin{aligned} h*[x/2] &= \langle \text{list} \cdot x \cdot \langle \text{item} \rangle \cdot \langle \text{list} \cdot \langle \text{list} \rangle \cdot \langle \text{item} \rangle \cdot \langle \text{item} \rangle \rangle \cdot \langle \text{list} \rangle \rangle \\ h*[x/6] &= \langle \text{list} \cdot \langle \text{item} \rangle \cdot \langle \text{list} \cdot x \cdot \langle \text{list} \rangle \cdot \langle \text{item} \rangle \cdot \langle \text{item} \rangle \rangle \cdot \langle \text{list} \rangle \rangle \end{aligned}$$

2.3.4 Schema Constraints for x-Annotations

When specifying queries $Q : \mathbf{S} \rightarrow 2^{\mathbb{N}}$ we may want to impose constraints where x -annotations may be inserted. This can be done by choosing some schema constraints, that is a schema for x -annotated hedges:

$$\mathbf{C} \subseteq \mathcal{H}_{\Sigma^x}$$

The query specified by a language $L \subseteq \mathcal{H}_{\Sigma^x}$, a schema for the query $\mathbf{S} \subseteq \mathcal{H}_{\Sigma}$, and a schema constraint for x -annotation $\mathbf{C} \subseteq \mathcal{H}_{\Sigma}$ then is:

$$qry_{\mathbf{S}}^{\mathbf{C}}(L) = qry_{\mathbf{S}}(L \cap \mathbf{C})$$

rather than $qry_{\mathbf{S}}(L)$ without the schema constraints on the x -annotations.

We note that $\Pi_{\Sigma}(\mathbf{C}) \neq \mathbf{S}$ is possible. In this case, one may not be able to infer the schema of the query \mathbf{S} from the schema constraints \mathbf{C} on the x -annotated hedges, nor vice versa.

Example 2.17. When considering monadic queries on nested lists, i.e., with the schema:

$$\mathbf{S} = \llbracket \text{N-List} \rrbracket$$

we may want to select `list` or `item` nodes only. In order to do so, we can impose the schema constraints \mathbf{C} on the x -annotations of nested lists

$$\mathbf{C} = \llbracket \text{N-List-}x \rrbracket$$

where:

$$\begin{aligned} \text{N-List-}x &=_{\text{def}} \text{Tree-}x^* \\ \text{Tree-}x &=_{\text{def}} \mu z. \langle (\text{list} + \text{item}) \cdot (x + \epsilon) \cdot z^* \rangle \end{aligned}$$

This means that variable x matters only when placed in a nested list only after a letter (`list` or `item`) following some opening parenthesis. Thus, the positions of some opening or closing parenthesis themselves cannot be selected. Clearly, $\Pi_{\Sigma}(\llbracket \text{N-List-}x \rrbracket) = \llbracket \text{N-List} \rrbracket$. Note however, that $\llbracket \text{N-List-}x \rrbracket$ does contain hedges, in which x occurs more than once or not at all, such as ϵ and $\langle \text{list} \cdot x \rangle \cdot \langle \text{list} \cdot x \rangle$. These are irrelevant for

queries, so one may want to impose the more restrictive constraint, without changing the query defined:

$$C' = C \cap \llbracket \text{One-}x \rrbracket$$

So for any query specification $L \subseteq \mathcal{H}_{\Sigma^x}$, we are not really interested in the query $\text{qry}_S(L \cap C)$ but by the query:

$$\text{qry}_S^C(L) = \text{qry}_S^{C'}(L)$$

Chapter 3

Hedge Automata

Abstract

We introduce stepwise hedge automata (SHAS) for the bottom-up evaluation of hedges. We then extend SHAS to downward stepwise hedge automata for the top-down evaluation of hedges, either in-memory or in streaming mode. We introduce the notion of schema-completeness and discuss two-sorted extensions for SHAS. The relationships to tree automata, forest automata, standard hedge automata and nested word automata are also discussed.

Contents

3.1	Finite State Automata on Words (NFAS)	40
3.1.1	Syntax	41
3.1.2	Semantics	41
3.1.3	Size Measures	42
3.1.4	Graphs	42
3.1.5	Completion	42
3.1.6	Runs	44
3.1.7	Determinization	44
3.1.8	Complementation	46
3.1.9	Product and Intersection	46
3.1.10	Accessibility	48
3.1.11	Minimization	49
3.1.12	Cleaning	49
3.1.13	Infinitary NFAs	50

3.1.14	Adding Else Rules	51
3.2	Stepwise Hedge Automata (SHAS)	53
3.2.1	Syntax	54
3.2.2	Semantics	56
3.2.3	Size Measures	58
3.2.4	Completion	58
3.2.5	Runs	58
3.2.6	Determinization	62
3.2.7	Complementation	62
3.2.8	Product and Intersection	63
3.2.9	Hedge Accessibility	63
3.2.10	Minimization	64
3.2.11	Cleaning	64
3.2.12	Infinitary SHAS	65
3.2.13	Else Rules	65
3.2.14	Related Automata Models	65
3.3	Downward Stepwise Hedge Automata (SHA^\downarrow s)	66
3.3.1	Syntax	67
3.3.2	Semantics	68
3.3.3	Completion	69
3.3.4	Runs	69
3.3.5	Conversion between SHAS and SHA^\downarrow s	70
3.3.6	Determinization	71
3.3.7	Minimization	72
3.3.8	Relationship to Nwas	72
3.4	Membership Testing	75
3.4.1	In-Memory	76
3.4.2	Streaming	77
3.5	Schema-Completeness	79
3.6	Schema-Based Cleaning	82
3.7	Two-sorted Automata	82
3.7.1	2-Sorted SHAS	83
3.7.2	2-Sorted SHA^\downarrow s	84

3.1 Finite State Automata on Words (N_{FAS})

We recall finite automata on words, their closure properties, minimization, and cleaning methods.

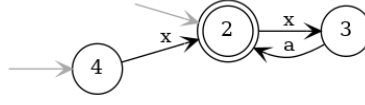


Figure 3.1: An NFA A_0 for the regular expression $(x + \epsilon) \cdot (x \cdot a)^*$. Note that it is not deterministic since it has two initial states.

3.1.1 Syntax

We start with the definition of nondeterministic finite state automata on words.

Definition 3.1. A NFA is a tuple $A = (\Sigma, Q, \Delta, I, F)$ such that Q is a finite set of states, the alphabet Σ is a finite set, $I, F \subseteq Q$ are subsets of initial and final states, and $\Delta \subseteq (Q \times \Sigma) \times Q$ is the set of transition rules. We call an NFA deterministic or equivalently a DFA, if I contains at most one initial state and Δ is a partial function, i.e., $|I| \leq 1$ and $\Delta : (Q \times \Sigma) \hookrightarrow Q$.

We write $q \xrightarrow{a} q' \in \Delta$ instead of a transition rule $(q, a, q') \in \Delta$. For any letter $a \in \Sigma$ we define a finite interpretation as a subset of transition rules:

$$a^\Delta = \{(q, q') \in Q^2 \mid q \xrightarrow{a} q' \in \Delta\}$$

If A is a DFA, then all interpretations a^Δ are partial functions, so $a^\Delta : Q \hookrightarrow Q$.

3.1.2 Semantics

We define transitions $q \xrightarrow{w} q'$ wrt Δ for arbitrary words $w \in \Sigma^*$ by the following inference rules:

$$\frac{q \in Q}{q \xrightarrow{\epsilon} q \text{ wrt } \Delta} \quad \frac{q \xrightarrow{a} q' \in \Delta}{q \xrightarrow{a} q' \text{ wrt } \Delta} \quad \frac{q_0 \xrightarrow{w_1} q_1 \text{ wrt } \Delta \quad q_1 \xrightarrow{w_2} q_2 \text{ wrt } \Delta}{q_0 \xrightarrow{w_1 \cdot w_2} q_2 \text{ wrt } \Delta}$$

The language of words recognized by a NFA then is:

$$\mathcal{L}(A) = \{w \in \Sigma^* \mid q \xrightarrow{w} q' \text{ wrt } \Delta, q \in I, q' \in F\}$$

3.1.3 Size Measures

We will use two measures for the size of automata. The number of states of an automata will be denoted by

$$n = |\mathcal{Q}|$$

while its overall size will be denoted by m , for an NFA:

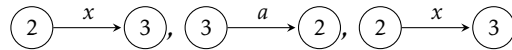
$$m = n + |\Sigma| + |\Delta| + |F| + |I|$$

For any DFA note that $m \in \Theta(|\Sigma| n)$. Therefore there is no significant difference between n and m for fixed signature Σ . This may not hold for NFAs though, where we only have $m \in \Theta(|\Sigma| n^2)$.

3.1.4 Graphs

We draw NFAs as graphs whose nodes are the states. A state $q \in \mathcal{Q}$ is drawn with a circle (q) , an initial state $q \in I$ with an incoming arrow $\rightarrow (q)$, and a final state with a double circle \textcircled{q} . A letter transition rule $(q_1, a, q_2) \in \Delta$ is drawn as a black edge $(q_1) \xrightarrow{a} (q_2)$ that is labeled by a letter $a \in \Sigma$. As a running example, an NFA A_0 for the regular expression $(x + \epsilon).(x.a)^*$ that is drawn in Figure 3.1.

Note that there exists a transition $q \xrightarrow{w} q'$ wrt Δ if and only if there exists a path in the graph of A over edges labeled by the letters in w , that starts with q and ends in q' . In the graph in Figure 3.1, for instance, the transition $2 \xrightarrow{xax} 3$ wrt Δ is coming from the following path:



3.1.5 Completion

Intuitively, an automaton is complete if for all states and letters in the signature, it has some transition rule. In addition, it must have at least one initial state.

Definition 3.2. We call a NFA $A = (\mathcal{Q}, \Sigma, \Delta, I, F)$ complete if the binary relation $\Delta \subseteq (\mathcal{Q} \times \Sigma) \times \mathcal{Q}$ is total and the set I is nonempty.

Lemma 3.3. If A is complete then for any word $w \in \Sigma^*$ there exists some transition $q \xrightarrow{w} q'$ wrt Δ for some $q \in I$ and $q' \in \mathcal{Q}$.

Any NFA can be completed. It is sufficient to add a fresh sink state and transition

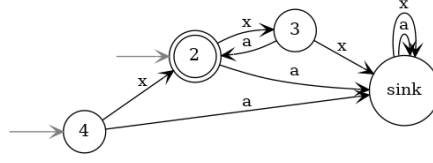


Figure 3.2: The completion $\text{complete}(A_0)$ adds the state sink to A_0 and transitions rules leading to the sink state.

rules into this sink whenever no other transition rule exists. More precisely, let $A = (\Sigma, \mathcal{Q}, \Delta, I, F)$ be an NFA and sink a fresh state. The completion $\text{complete}(A)$ is then defined as follows:

$$\text{complete}(A) = (\Sigma, \mathcal{Q} \uplus \{\text{sink}\}, \Delta', I, F)$$

where Δ' contains the following transition rules:

$$\frac{q \xrightarrow{a} q' \in \Delta}{q \xrightarrow{a} q' \in \Delta'} \quad \frac{a \in \Sigma \quad q \in \mathcal{Q} \quad \nexists q' \in \mathcal{Q}. q \xrightarrow{a} q' \in \Delta}{q \xrightarrow{a} \text{sink} \in \Delta'} \quad \frac{a \in \Sigma}{\text{sink} \xrightarrow{a} \text{sink} \in \Delta'}$$

Example 3.4. The completion $\text{complete}(A_0)$ of the automaton A_0 is shown in Figure 3.2. The added sink state cannot reach any final state and is thus irrelevant for the language.

Proposition 3.5. For any NFA A : $\mathcal{L}(A) = \mathcal{L}(\text{complete}(A))$.

Proof. The sink cannot be used in any successful run of $\text{complete}(A)$. Since the sink state cannot access any final state, the language of A remains unchanged when adding a sink. \square

Lemma 3.6. If A is deterministic then $\text{complete}(A)$ is deterministic too.

Proof. The completion adds transition rules for letters in a deterministic manner only when none existed from the original states with the same letters. This preserves determinism. \square

3.1.6 Runs

A run of NFA A on a word $w \in \Sigma^*$ is a word in $(\Sigma \uplus \mathcal{Q})^*$. We define the set of runs $R \in \text{run}^\Delta(w)$ by the following rules, such that for all $w, w' \in \Sigma^*$:

$$\frac{\text{true}}{q \in \text{run}^\Delta(\epsilon)} \quad \frac{q \xrightarrow{a} q' \in \Delta}{q \cdot a \cdot q' \in \text{run}^\Delta(a)} \quad \frac{q \cdot R \cdot q' \in \text{run}^\Delta(w) \quad q' \cdot R' \cdot q'' \in \text{run}^\Delta(w')}{q \cdot R \cdot q' \cdot R' \cdot q'' \in \text{run}^\Delta(w \cdot w')}$$

A run of $A = (\mathcal{Q}, \Sigma, \Delta, I, F)$ is called *successful* if starts in some state in I it ends with some state in F .

Lemma 3.7. *Any word $w \in \Sigma^*$ satisfies that $w \in \mathcal{L}(A)$ if and only if there exists a successful run $R \in \text{run}^\Delta(w)$ of A .*

Proof. Straightforward by induction on w . □

Definition 3.8. A partial run of A on a word w is a prefix r of some run of $\text{compl}(A)$ on w such that r does not contain the state sink added by $\text{compl}(A)$. A partial run is *blocking* if is maximal and ends with some letter from Σ .

Example 3.9. A successful run of the NFA A_0 in Figure 3.1 on the word xxa is $4x2x3a2$. On the same word, there is also a blocking partial run $2x3x$ of A_0 . It starts from the other initial state of A_0 than the successful run on xxa . It is blocking since there is no transition rule starting in 3 reading x .

3.1.7 Determinization

We next remind that any NFA can be made deterministic while preserving the language. So let $A = (\mathcal{Q}, \Sigma, \Delta, I, F)$ be an NFA. We construct its (full) determinization:

$$A^{\text{det-f}} = (\Sigma, \mathcal{Q}^{\text{det-f}}, \Delta^{\text{det-f}}, I^{\text{det-f}}, F^{\text{det-f}})$$

by the usual subset construction. We call this determinization "full" in order to distinguish it from the accessible determinization A^{det} , that will come later on. The state set of the determinization is:

$$\mathcal{Q}^{\text{det-f}} = 2^{\mathcal{Q}} \setminus \{\emptyset\}$$

Note that the empty set is excluded from $\mathcal{Q}^{\text{det-f}}$. Otherwise, it would become a sink state, that is useless since not co-accessible. The initial and final subsets of the full

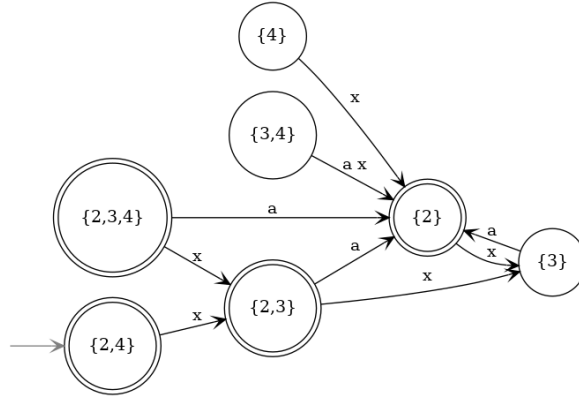


Figure 3.3: The full determinization A_0^{det-f} for the NFA A_0 of Figure 3.1. Note that the states $\{2, 3, 4\}$, $\{3, 4\}$ and $\{4\}$ are not accessible from the initial state and thus useless.

determinization are the following:

$$\begin{aligned} I^{det-f} &= \{I \mid I \neq \emptyset\} \\ F^{det-f} &= \{Q \subseteq \mathcal{Q} \mid Q \cap F \neq \emptyset\} \end{aligned}$$

The set of transition rules Δ^{det-f} can be inferred by the following rule:

$$\frac{Q \in \mathcal{Q}^{det-f} \quad a \in \Sigma \quad Q' = \{q' \in \mathcal{Q} \mid q \xrightarrow{a} q' \in \Delta, q \in Q\} \neq \emptyset}{Q \xrightarrow{a} Q' \in \Delta^{det-f}}$$

For each nonempty subset of states $Q \subseteq \mathcal{Q}$ and letter $a \in \Sigma$, the full determinization has a transition rule $Q \xrightarrow{a} Q' \in \Delta^{det-f}$ where Q' is the set of all states q' to which there exists a transition rule $q \xrightarrow{a} q'$ for the same state $q \in Q$.

Example 3.10. In Figure 3.3, we show the DFA A_0^{det-f} obtained by the full determinization of NFA A_0 from Figure 3.1. Note that all subsets of states of A_0 are states of A_0^{det-f} except for the empty set \emptyset . Note also, that some of the subsets – $\{4\}$, $\{3, 4\}$, and $\{2, 3, 4\}$ – are not accessible from the initial state $\{2, 4\}$ in A_0^{det-f} , so they cannot be used in any run of A_0^{det-f} .

Proposition 3.11 (Folklore). Any NFA A can be determinized while preserving the language:

$$\mathcal{L}(A) = \mathcal{L}(A^{det-f})$$

Proof. It is sufficient to show for any nonempty subsets $Q, Q' \subseteq \mathcal{Q}$ and word $w \in \Sigma^*$:

$$Q \xrightarrow{w} Q' \text{ wrt } \Delta^{det-f} \Leftrightarrow Q' = \{q' \in \mathcal{Q} \mid \exists q \in Q. q \xrightarrow{w} q' \text{ wrt } \Delta\}$$

This can be shown by induction on the size of w . □

3.1.8 Complementation

We next show that regular languages are closed under complement. For any complete DFA A , we define its complement automaton by \bar{A} by flipping the final states of A , i.e.:

$$\bar{A} = (\Sigma, \mathcal{Q}, \Delta, I, \bar{F})$$

.

Proposition 3.12 (Folklore). *For any complete DFA A : $\overline{\mathcal{L}(A)} = \mathcal{L}(\bar{A})$.*

Proof. Let $A = (\mathcal{Q}, \Sigma, \Delta, I, F)$ be a complete DFA and $w \in \Sigma^*$ an arbitrary word. Since A is complete there exists $q \in I$ and $q' \in \mathcal{Q}$ such that $q \xrightarrow{w} q' \text{ wrt } \Delta$. Since A is deterministic, the states q and q' unique with this property. Therefore:

$$w \in \mathcal{L}(A) \Leftrightarrow q' \in F$$

Since $\bar{A} = (\Sigma, \mathcal{Q}, \Delta, I, \bar{F})$, there is also a unique transition of \bar{A} on w that starts in I . Since A and \bar{A} have the same set of initial states and the same set of transition rules, this transition is $q \xrightarrow{w} q' \text{ wrt } \Delta$. Hence:

$$w \in \mathcal{L}(\bar{A}) \Leftrightarrow q' \notin F$$

As a consequence, $w \in \mathcal{L}(A) \Leftrightarrow w \notin \mathcal{L}(\bar{A})$. Since w was arbitrary, this yields $\overline{\mathcal{L}(A)} = \mathcal{L}(\bar{A})$. □

3.1.9 Product and Intersection

For any two NFAs A and B we can construct an NFA $A \times^f B$ that runs A and B in parallel, and accepts if both accept. For this, we define the (full) product of two NFAs $A = (\Sigma, \mathcal{Q}^A, \Delta^A, I^A, F^A)$ and $B = (\Sigma, \mathcal{Q}^B, \Delta^B, I^B, F^B)$ as the NFA

$$A \times^f B = (\Sigma, \mathcal{Q}^{A \times^f B}, I^{A \times^f B}, F^{A \times^f B}, \Delta^{A \times^f B})$$

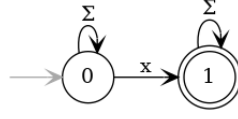


Figure 3.4: The DFA *word-one- x_Σ* with signature $\Sigma \uplus \{x\}$.

as follows. The states of $A \times^f B$ are all sets of pairs of states from A and B , i.e.:

$$Q^{A \times^f B} = Q^A \times Q^B$$

The initial and final states are defined as follows:

$$\begin{aligned} I^{A \times^f B} &= I^A \times I^B \\ F^{A \times^f B} &= F^A \times F^B \end{aligned}$$

Furthermore, the set of transition rules $\Delta^{A \times^f B}$ is defined by the following inference rule:

$$\frac{q_1 \xrightarrow{a} q_2 \in \Delta^A \quad p_1 \xrightarrow{a} p_2 \in \Delta^B}{(q_1, p_1) \xrightarrow{a} (q_2, p_2) \in \Delta^{A \times^f B}}$$

Proposition 3.13. $\mathcal{L}(A \times^f B) = \mathcal{L}(A) \cap \mathcal{L}(B)$.

Proof. Let $w \in \Sigma^*$ and $p, p' \in Q^A$ and $q, q' \in Q^B$ be arbitrary. We then have:

$$(p, q) \xrightarrow{w} (p', q') \in \Delta^{A \times^f B} \Leftrightarrow p \xrightarrow{w} p' \in \Delta^A \wedge q \xrightarrow{w} q' \in \Delta^B$$

By construction of $I^{A \times^f B}$ and $F^{A \times^f B}$ it follows that $w \in \mathcal{L}(A \times^f B)$ if and only if $w \in \mathcal{L}(A)$ and $w \in \mathcal{L}(B)$. \square

Example 3.14. The DFA *word-one- x_Σ* in Figure 3.4 recognizes all words with $\Sigma \uplus \{x\}$ that contain exactly one occurrence of x . This automaton will be used later on as a schema for monadic queries on words. In Figure 3.5, we show the full product $A_0 \times^f \text{word-one-}x_{\{a\}}$. It accepts the words of $\mathcal{L}(A_0)$ with exactly one occurrence of x , i.e., $\mathcal{L}(A_0 \times^f \text{word-one-}x_{\{a\}}) = \{x, x \cdot a\}$. Note that the loop $2 \xrightarrow{x} 3 \xrightarrow{a} 2$ of A_0 is no more present in the full product, since it would accumulate more than one x , so that *word-one- $x_{\{a\}}$* blocks.

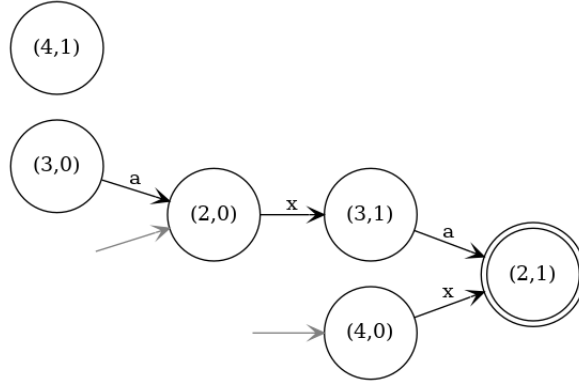


Figure 3.5: The full product $A_0 \times^f \text{word-one-}x_{\{a\}}$ where A_0 is given Figure 3.1 and $\text{word-one-}x_{\{a\}}$ in Figure 3.4. Note the states $(4,1)$ and $(3,0)$ of the full product are useless since not accessible from the initial states.

3.1.10 Accessibility

Let $A = (\mathcal{Q}, \Sigma, \Delta, I, F)$ be an NFA and $Q \subseteq \mathcal{Q}$ a subset of states. The set of states accessible from Q is then defined as follows:

$$\text{acc}^\Delta(Q) = \{q' \in \mathcal{Q} \mid \exists q \in Q. \exists w \in \Sigma^*. q \xrightarrow{w} q' \text{ wrt } \Delta\}$$

So $q \in \text{acc}^\Delta(Q)$ if and only if there is some path in the graph of A that starts from some state in Q and ends in q . For any set Q , we can compute the set of accessible states $\text{acc}^\Delta(Q)$ in time $|\Delta| + |Q|$ by a traversal of the graph of A starting from Q . Alternatively, we can compute $\text{acc}^\Delta(Q)$ as the least fixed point of a ground Datalog program of size $|\Delta| + |Q|$:

$$\frac{q \in Q}{q \in \text{acc}^\Delta(Q)}. \quad \frac{q \xrightarrow{a} q' \in \Delta}{q' \in \text{acc}^\Delta(Q) :- q \in \text{acc}^\Delta(Q)}.$$

Since the size of this Datalog program is linear in the size of A , the set of all accessible states can be computed in time $O(m)$. The inverse accessibility relation for a set of states Q is defined as follows:

$$\text{invacc}^\Delta(Q) = \{q' \mid q \in \text{acc}^\Delta(\{q'\}), q \in Q\}$$

Note that $invacc^\Delta(Q) = acc^{\Delta^{-1}}(Q)$ where Δ^{-1} is the set of inverse transition rules of Δ , so for all $q, q' \in Q$ and $a \in \Sigma$:

$$\frac{q \xrightarrow{a} q' \in \Delta}{q' \xrightarrow{a} q \in \Delta^{-1}}$$

Therefore, $invacc^\Delta(Q)$ can also be computed in linear time.

3.1.11 Minimization

We are generally interested in reducing the size nondeterministic automata as much as possible while preserving the language. In the case of D_{FAS} this can be done by automata minimization.

It is well known, D_{FAS} enjoy unique minimization, i.e., for any regular language L , the DFA A with $\mathcal{L}(A) = L$ with a minimal number of states is unique up to state renaming.

3.1.12 Cleaning

For more general N_{FAS}, it is still possible to clean the automata by removing useless states, i.e., states that are not used in any successful run. Clearly, the states in $\overline{acc^\Delta(I)}$ and $invacc^\Delta(F)$ are useless.

Lemma 3.15. *A state $q \in Q$ of an NFA $A = (Q, \Sigma, \Delta, I, F)$ is useless if and only if $q \notin acc^\Delta(I) \cap invacc^\Delta(F)$.*

For any NFA $A = (Q, \Sigma, \Delta, I, F)$ and subset of states $Q \subseteq Q$, we define the cleaning of A with respect to Q , removing the states in \overline{Q} , by:

$$clean(A, Q) = (\Sigma, Q, \Delta \cap (Q \times \Sigma \times Q), I \cap Q, F \cap Q)$$

Lemma 3.16. *If all states in a subset $Q \setminus Q$ are useless then $\mathcal{L}(A) = \mathcal{L}(clean(A, Q))$.*

We define the accessibility cleaning, the co-accessibility cleaning, and the trimming of A as follows:

$$\begin{aligned} acc-clean(A) &= clean(A, acc^\Delta(I)) \\ coacc-clean(A) &= clean(A, invacc^\Delta(F)) \\ trim(A) &= clean(A, acc^\Delta(I) \cap invacc^\Delta(F)) \end{aligned}$$

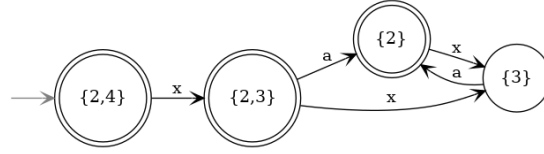


Figure 3.6: The trimmed automaton $\text{trim}(A_0^{\text{det-f}}) = \text{acc-clean}(A_0^{\text{det-f}})$.

By Lemma 3.15, that automaton $\text{trim}(A)$ removes all useless states of NFA A . The accessibility cleaning and co-accessibility cleaning each remove only a subset of these states.

Example 3.17. The accessibility cleaning and trimming of $A_0^{\text{det-f}}$ result in the same automaton, i.e., $\text{trim}(A_0^{\text{det-f}}) = \text{acc-clean}(A_0^{\text{det-f}})$, shown in Figure 3.6. Note that this DFA is minimal.

We can now refine the notions of full determinization and product to the accessible determinization and product. For any NFAs A and B , we define the accessible determinization as the accessibility cleaning of the full determinization:

$$\text{det}(A) = \text{acc-clean}(A^{\text{det-f}})$$

and the accessible product by the accessibility cleaning of the full product:

$$A \times B = \text{acc-clean}(A \times^f B)$$

It is not difficult to see how to compute $\text{det}(A)$ from A without computing $A^{\text{det-f}}$, as well as how to compute $A \times B$ from A and B without computing $A \times^f B$. Co-accessibility cleaning, in contrast, can be done only a posteriori. This is the reason, why accessibility cleaning is of particular interest, even though trimming may be stronger.

3.1.13 Infinitary NFAs

We sometimes need to admit infinitary NFAs, which are similar to NFAs expect that the sets of states and transition rules may be infinite.

Definition 3.18. An NFA^∞ is a tuple $A = (Q, \Sigma, \Delta, I, F)$ such that Σ and Q are sets, $I, F \subseteq Q$ and $\Delta \subseteq (Q \times \Sigma) \times Q$. An NFA^∞ is called deterministic or equivalently a DFA^∞ if $\Delta \subseteq (Q \times \Sigma) \times Q$ is a partial function.

In other words, there are no finiteness restrictions any more, so any component of infinitary NFAS may be infinite. Such general infinitary automata can be used for instance to define query answering algorithms, that can store integers in their states, i.e., the nodes of hedges.

3.1.14 Adding Else Rules

In order to deal with infinite signatures, we will consider infinitary NFAS with else rules, but with finite sets of states and transition rules. More general kinds of symbolic automata [Veanes *et al.* 2012] will not be needed, neither in the present thesis, nor in the practical tools issuing from this thesis.

An NFA with untyped else rules is an infinitary NFA with a special symbol $_$ added to the signature.

Definition 3.19. *An NFA over Σ with untyped else rules is an $NFA^\infty A = (\Sigma \uplus \{_ \}, Q, \Delta, I, F)$ such that the sets Q and Δ are finite.*

Semantically, an else rule $q \rightrightarrows q' \in \Delta$ means that the automaton in state q can go to state q' when reading any letter $a \in \Sigma$ such there exists no state $q'' \in Q$ with $q \xrightarrow{a} q'' \in \Delta$. Even in the case of finite signatures, this extension is relevant, since large numbers of transition rules with labels in Σ can be represented symbolically in a more concise manner by using else rules $q \rightrightarrows q' \in \Delta$.

Any else rule can be expanded to a possibly infinite set of letter transition rules $\Delta^{exp} \subseteq Q \times \Sigma \times Q$ as follows:

$$\frac{q \rightrightarrows q' \in \Delta \quad a \in \Sigma \quad \neg \exists q'' \in Q. q \xrightarrow{a} q'' \in \Delta}{q \xrightarrow{a} q' \in \Delta^{exp}} \quad \frac{a \in \Sigma \quad q \xrightarrow{a} q' \in \Delta}{q \xrightarrow{a} q' \in \Delta^{exp}}$$

By contrast, Δ was assumed to be finite. In this sense, else rules represent infinite set of letter rules symbolically.

3.1.14.1 Typed Else Rules

We next consider NFAS with typed signatures, leading us to the concept of NFAS with typed else rules. A typing for a signature Σ is a function

$$type : \Sigma \rightarrow T$$

with some finite range T , that is called the set of types. The typing function assigns some type from T to each letter of Σ .

Definition 3.20. Let $type : \Sigma \rightarrow T$ be a typing function. An NFA over Σ with typed else rules is an $NFA^\infty A = (\Sigma \uplus \{ _ : \tau \mid \tau \in T \}, Q, \Delta, I, F)$ such that Q and Δ are finite.

Semantically, an typed else rule $q \xrightarrow{\cdot \tau} q' \in \Delta$ means that the automaton in state q can go to state q' when reading any letter $a \in \Sigma$ with $type(a) = \tau$ such there exists no state $q'' \in Q$ with $q \xrightarrow{a} q'' \in \Delta$. A typed else rule can be expanded with all letters $a \in \Sigma$ with $type(a) = \tau$:

$$\frac{q \xrightarrow{\cdot \tau} q' \in \Delta \quad type(a) = \tau \quad \neg \exists q'' \in Q. q \xrightarrow{a} q'' \in \Delta}{q \xrightarrow{a} q' \in \Delta^{exp}}$$

3.1.14.2 Lifting Automata Constructions

All above automata constructions for NFAs can be lifted to NFAs with else rules. This has to be done carefully in order to avoid automata with infinitely many transition rules, i.e., the infinite expansions Δ^{exp} . It is sufficient to treat the case of typed else rules, since any untyped else rule can be encoded by a finite set of typed else rule, one for each type, as follows:

$$\frac{\tau \in T \quad q \rightarrow q' \in \Delta^{untyped}}{q \xrightarrow{\cdot \tau} q' \in \Delta^{typed}}$$

Determinization. The full determinization A^{det-f} can be defined for any NFA with typed else rules A , by lifting the full determinization of NFAs as follows:

$$\frac{a \in \Sigma \quad \tau = type(a) \quad Q \subseteq Q \quad \{q \xrightarrow{a} q' \in \Delta \mid q \in Q, q' \in Q\} \neq \emptyset}{Q \xrightarrow{a} \{q' \in Q \mid q \xrightarrow{a} q' \in \Delta^{exp}, q \in Q\} \in \Delta^{det-f}}$$

$$\frac{\tau \in T \quad Q \subseteq Q \quad Q' = \{q' \in Q \mid q \xrightarrow{\cdot \tau} q' \in \Delta, q \in Q\} \neq \emptyset}{Q \xrightarrow{\cdot \tau} Q' \in \Delta^{det-f}}$$

It should be noticed that the set of transition rules Δ^{det-f} remains finite since Δ was finite, even though Δ^{exp} may be infinite.

Product. The full product $A \times^f B$ can also be lifted to NFAS with typed else rules as follows:

$$\frac{q_1 \xrightarrow{a} q_2 \in \Delta^A \quad p_1 \xrightarrow{a} p_2 \in \Delta^{B^{exp}}}{(q_1, p_1) \xrightarrow{a} (q_2, p_2) \in \Delta^{A \times^f B}} \quad \frac{q_1 \xrightarrow{a} q_2 \in \Delta^{A^{exp}} \quad p_1 \xrightarrow{a} p_2 \in \Delta^B}{(q_1, p_1) \xrightarrow{a} (q_2, p_2) \in \Delta^{A \times^f B}}$$

$$\frac{q_1 \xrightarrow{\cdot:\tau} q_2 \in \Delta^A \quad p_1 \xrightarrow{\cdot:\tau} p_2 \in \Delta^B}{(q_1, p_1) \xrightarrow{\cdot:\tau} (q_2, p_2) \in \Delta^{A \times^f B}}$$

Note that $\Delta^{A \times^f B}$ remains finite since Δ^A and Δ^B are finite, even though $\Delta^{A^{exp}}$ and $\Delta^{B^{exp}}$ may be infinite. It should also be clear that the product automaton $A \times^f B$ recognizes the intersection $\mathcal{L}(A) \cap \mathcal{L}(B)$.

Completion. An NFA with else rules is complete if I is nonempty and for all $q \in \mathcal{Q}$ and $a \in \Sigma$ there exists $q' \in \mathcal{Q}$ such that $q \xrightarrow{a} q' \in \Delta^{exp}$. The idea for completing an NFA with else rules, is to add else rules leading to the added sink state. More formally, let A be a NFA with typed else rules for the typing $type : \Sigma \rightarrow T$. We then define the completion

$$complete'(A) = (\Sigma, \mathcal{Q} \uplus \{sink\}, \Delta', I, F)$$

as an NFA with typed else rules and the same typing such that:

$$\frac{a \in \Sigma \quad q \xrightarrow{a} q' \in \Delta}{q \xrightarrow{a} q' \in \Delta'} \quad \frac{\tau \in T \quad q \xrightarrow{\cdot:\tau} q' \in \Delta}{q \xrightarrow{\cdot:\tau} q' \in \Delta'}$$

$$\frac{\tau \in T \quad q \in \mathcal{Q} \quad \nexists q'. q \xrightarrow{\cdot:\tau} q' \in \Delta}{q \xrightarrow{\cdot:\tau} sink \in \Delta'} \quad \frac{\tau \in T}{sink \xrightarrow{\cdot:\tau} sink \in \Delta'}$$

Complementation. Complete DFAS with typed else rules can be complemented by flipping final states, exactly as for DFAS without else rules.

3.2 Stepwise Hedge Automata (SHAS)

We show how to capture regular hedge languages by stepwise hedge automata (SHAS).

3.2.1 Syntax

SHAS are a more recent notion of automata for hedges [Niehren & Sakho 2021] that mix up bottom-up tree automata and left-to-right word automata in a natural manner. They extend stepwise tree automata [Carme *et al.* 2004] in that, they operate on hedges rather than unranked trees, i.e., on sequences of letters and trees containing hedges. They differ from nested word automata (Nwas) [Okhotin & Salomaa 2014, Alur 2007] in that, they process hedges directly rather than their linearizations to nested words.

Definition 3.21. A stepwise hedge automata (SHA) is a tuple $A = (\Sigma, \mathcal{Q}, \Delta, I, F)$ such that the alphabet Σ is a finite set of letters, \mathcal{Q} is a finite set of states, $I, F \subseteq \mathcal{Q}$ are subsets of initial and final states, $\Delta = (\Delta', \langle \rangle^\Delta, @^\Delta)$ is the set of transition rules, with $\Delta' \subseteq (\mathcal{Q} \times \Sigma) \times \mathcal{Q}$ a set of letter rules, $\langle \rangle^\Delta \subseteq \mathcal{Q}$ a set of tree initial states, and $@^\Delta \subseteq (\mathcal{Q} \times \mathcal{Q}) \times \mathcal{Q}$ a set of apply rules. A SHA is deterministic or equivalently a $dSHA$ if I and $\langle \rangle^\Delta$ contain at most one element, and all relations in Δ are partial functions.

Any SHA $A = (\Sigma, \mathcal{Q}, \Delta, I, F)$ where $\Delta = (\Delta', \langle \rangle^\Delta, @^\Delta)$ extends on the NFA $(\Sigma, \mathcal{Q}, \Delta', I, F)$, by adding a set of tree initial states $\langle \rangle^\Delta \subseteq \mathcal{Q}$ and a set of apply rules $@^\Delta \subseteq (\mathcal{Q} \times \mathcal{Q}) \times \mathcal{Q}$. Whenever $q \in \langle \rangle^\Delta$, we also write $\xrightarrow{\langle \rangle} q \in \Delta$, and alternatively to $(q, p, q') \in @^\Delta$ we write $q@p \rightarrow q' \in \Delta$.

Example 3.22. An $dSHA$ for the nested regular expression $N\text{-List}$ from Section 2.3.2 is defined in Figure 3.7 and graphically represented in Figure 3.8. This $dSHA$ recognizes for instance the hedge $\langle list \cdot \langle list \rangle \cdot \langle item \rangle \rangle \cdot \langle list \rangle$ but not the hedge $\langle \langle list \rangle \cdot \langle item \rangle \rangle$. Its transition rules in Δ can be rewritten as the following set:

$$\{0 \xrightarrow{\text{list}} 1, 0 \xrightarrow{\text{item}} 1, 0@1 \rightarrow 4, 1@1 \rightarrow 1, 4@1 \rightarrow 4, \xrightarrow{\langle \rangle} 0\}$$

The graphs of SHA can be inferred and drawn similarly to $NFAS$, see e.g. Figure 3.8. The states of the automaton are the nodes of the graph and its transition rules are the edges. Any apply rule $q_1@p \rightarrow q_2 \in \Delta$ is represented by a blue edge $\textcircled{q_1} \xrightarrow{p} \textcircled{q_2}$ that is annotated with state p (and not with some letter from Σ). Tree initial states $q \in \langle \rangle^\Delta$ are indicated by an incoming $\langle \rangle$ -edge as follows $\xrightarrow{\langle \rangle} \textcircled{q}$.

Example 3.23. In Figure 3.9, another $dSHA$ for the nested regular expression $N\text{-List}$ is given. It has two disjoint levels, the subautomaton accessible from I and subautomaton accessible from $\langle \rangle^\Delta$. It has the same number of states, but nevertheless is not equal to the $dSHA$ in Figure 3.8 up to renaming of states. This shows that unique minimization does not hold for $dSHAS$.

$$\begin{array}{ll}
\Sigma &= \{\text{list}, \text{item}\} & \mathcal{Q} &= \{0, 1, 4\} \\
I &= \{0\} & F &= \{0, 4\} \\
\text{list}^\Delta &= \{(0, 1)\} & \text{item}^\Delta &= \{(0, 1)\} \\
\langle \rangle^\Delta &= \{0\} & @^\Delta &= \{(0, 1, 4), (1, 1, 1), (4, 1, 4)\}
\end{array}$$

Figure 3.7: A dSHA $A = (\Sigma, \mathcal{Q}, \Delta, I, F)$ for the nested regular expression N-List from Section 2.3.2. This dSHA is drawn graphically in Figure 3.8.

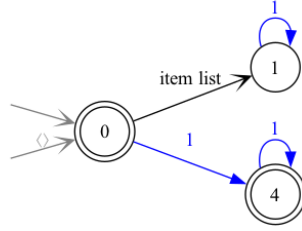


Figure 3.8: A graphical presentation of the dSHA in Figure 3.7 for the nested regular expression N-List from Section 2.3.2.

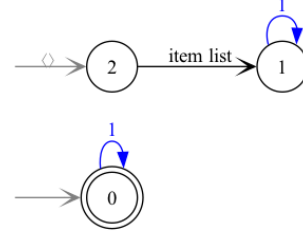


Figure 3.9: Another state-minimal dSHA for the nested regular expression N-List from Section 2.3.2 with two separated levels.

As shown in [Niehren & Sakho 2021], unique minimization holds for the subclass of dSHAs with $I = \langle \rangle^\Delta$. In [Niehren 2024], it is shown that unique minimization also holds for the subclass dSHAs with two levels, so where top-level (the states sets accessible from I) and the tree level (the states accessible from $\langle \rangle^\Delta$) are disjoint. The unique minimal deterministic SHAs for these two classes are exactly those shown in Figures 3.8 and 3.9 respectively.

Example 3.24. We consider the boolean query with schema $\llbracket \text{N-List} \rrbracket$ that is defined by the following regular expression mimicking the XPATH filter $[\text{self::list/child::item}]$:

$$[\text{self-list-child-item}] =_{\text{def}} \langle \text{list} \cdot \text{N-List} \cdot \langle \text{item} \cdot \text{N-List} \rangle \cdot \text{N-List} \rangle \cdot \text{N-List}$$

The language $\llbracket [\text{self-list-child-item}] \rrbracket$ contains all nested lists, where the first tree starts with a list-node that contains some item-child. This language can be defined equivalently by the dSHA in Figure 3.10. In order to accept a nested list, it has to move from the initial state 0 to the final state 4 but reading some tree in state 3, by using the apply rule $0@3 \rightarrow 4$ that is drawn as $(0) \xrightarrow{3} (4)$. For a subtree to get into state 3 it has to start with the tree initial state 0, read a letter list to go to state 1, by using the rule $0 \xrightarrow{\text{list}} 1$, and then eventually read some tree in state 2 with the apply rule drawn as $(1) \xrightarrow{2} (3)$.

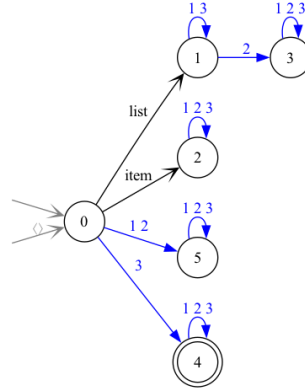


Figure 3.10: A dSHA recognizing the language of the nested regular expression $[\text{self-list-child-item}] = \langle \text{list} \cdot \text{N-List} \cdot \langle \text{item} \cdot \text{N-List} \rangle \cdot \text{N-List} \rangle \cdot \text{N-List}$.

For a subtree to get into state 2 it has to start with the tree initial state 0, read a letter *item* there by using the rule $0 \xrightarrow{\text{item}} 2$.

This automaton can also be run on all hedges from the schema that are not accepted, i.e., the hedges in $\llbracket \text{N-List} \rrbracket \setminus \llbracket [\text{self-list-child-item}] \rrbracket$ for which, the run will end in the sink state 5. So this automaton will never block when run on some hedge satisfying schema $\llbracket \text{N-List} \rrbracket$. It does block, however, for hedges outside the schema such as $\langle \langle \text{list} \rangle \rangle$, since it is missing an apply rule with left hand side $0@5$. We therefore call this automaton *schema-complete* for schema $\llbracket \text{N-List} \rrbracket$, while not being complete in general.

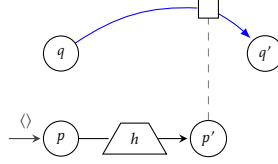
3.2.2 Semantics

We next define the semantics of SHAS, i.e., the transitions on hedges that it defines and the hedge language that it accepts. For any hedge $h \in \mathcal{H}_\Sigma$, we define the transition steps $q \xrightarrow{h} p$ wrt Δ such that for all $q, q', p, p' \in \mathcal{Q}$, $a \in \Sigma$, and $h, h' \in \mathcal{H}_\Sigma$:

$$\begin{array}{c}
 \frac{q \xrightarrow{a} q' \in \Delta}{q \xrightarrow{a} q' \text{ wrt } \Delta} \qquad \frac{q \xrightarrow{h} q' \text{ wrt } \Delta \quad q' \xrightarrow{h'} q'' \text{ wrt } \Delta}{q \xrightarrow{h \cdot h'} q'' \text{ wrt } \Delta} \\
 \\
 \frac{q \in \mathcal{Q}}{q \xrightarrow{\epsilon} q \text{ wrt } \Delta} \qquad \frac{\langle \rangle \rightarrow p \in \Delta \quad p \xrightarrow{h} p' \quad q@p' \rightarrow q' \in \Delta}{q \xrightarrow{\langle h \rangle} q' \text{ wrt } \Delta}
 \end{array}$$

The transitions can be used to evaluate hedges nondeterministically bottom-up and left-to-right by SHAS. The first three rules state how to evaluate sequences of trees and letters as a usual finite state automaton, while assuming that the trees were already evaluated to states. The last rule states how to evaluate a tree $\langle h \rangle$ from

some state q to some state q' . This can be visualized as follows:



For any tree initial state $p \in \langle \rangle^\Delta$, one has to evaluate the subhedge h to some state p' nondeterministically. For each p' obtained, one then returns some state q' such that $q@p' \rightarrow q' \in \Delta$ non-deterministically.

A hedge is accepted by A if it permits a transition from some initial state to some final state. The language $\mathcal{L}(A)$ is the set of all accepted hedges:

$$\mathcal{L}(A) = \{h \in \mathcal{H}_\Sigma \mid q \xrightarrow{h} q' \text{ wrt } \Delta, q \in I, q' \in F\}$$

Proposition 3.25 (Theorem 2 of [Niehren & Sakho 2021]). *For any nested regular expression $e \in nRegExp$, we can construct a nondeterministic SHA A in linear time such that $\mathcal{L}(A) = \mathcal{L}(e)$.*

It should be noticed that the proof of this Proposition uses 2-level SHAS, whereas, for most of our example SHAS, we will satisfy $I = \langle \rangle^\Delta$.

Theorem 1 (Kleene's Theorem for SHAS). *A hedge language is regular if and only if it can be defined by some SHA.*

Proof. The one direction is from Proposition 3.25. Conversely, it is not difficult to see that the language of any SHA can be defined by some nested regular expression. \square

To the best of our knowledge, the analogous result has not been formulated for standard hedge automata before (see e.g. Chapter 8 of [Comon *et al.* 2007]). However, analogous results are folklore for both tree automata [Gécseg & Steinby 1984] and word automata [Yu 1997], where it is known as Kleene's theorem from the fifties.

Corollary 3.26. *A monadic query on hedges is regular if and only if it can be defined by some SHA.*

Proof. Immediate from Theorem 1. \square

3.2.3 Size Measures

The number of states of a $\text{SHA } A = (\mathcal{Q}, \Sigma, \Delta, I, F)$ is denoted by $n = |\mathcal{Q}|$. Suppose that $\Delta = (\Delta', @^\Delta, \langle \rangle^\Delta)$. Then the overall size m of A is defined by:

$$m = n + |\Sigma| + |\Delta'| + |@^\Delta| + |\langle \rangle^\Delta| + |I| + |F|$$

For dSHAS note that $m \in \Omega(n^2 + |\Sigma| n)$. In contrast to DFAS (see Section 3.1.3), note that m may grow quadratically in n for dSHAS due to $|@^\Delta|$, even when fixing the signature Σ . For general SHAS , there may even be a cubic dependency, since then $m \in \Omega(n^3 + |\Sigma| n^2)$. Again, this size increase is due to the addition of apply rules $@^\Delta$ to NFAS .

3.2.4 Completion

Similarly to NFAS , any SHA can be completed with the same intuition, with one addition: it must also have at least one tree initial state. Completion works as before by adding a sink state. We now also have to add apply rules to the sink as follows, where Δ is original set of transition rules and Δ' that of the completion:

$$\frac{q, p \in \mathcal{Q} \cup \{\text{sink}\} \quad \neg \exists q' \in \mathcal{Q}. q @ p \rightarrow q' \in \Delta}{q @ p \rightarrow \text{sink} \in \Delta'}$$

3.2.5 Runs

Runs can represent the whole history of a single choice of the nondeterministic evaluator on a hedge. We define runs of SHAS on hedges formally. Whether a hedge with letters and states $R \in \mathcal{H}_{\Sigma \cup \mathcal{Q}}$ is a Δ -run or simply a run on a hedge $h \in \mathcal{H}_\Sigma$ – written $R \in \text{run}^\Delta(h)$ – is defined by the following rules:

$$\begin{array}{c} \frac{q \xrightarrow{a} q' \in \Delta}{q \cdot a \cdot q' \in \text{run}^\Delta(a)} \quad \frac{q \cdot R \cdot q' \in \text{run}^\Delta(h) \quad q' \cdot R' \cdot q'' \in \text{run}^\Delta(h')}{q \cdot R \cdot q' \cdot R' \cdot q'' \in \text{run}^\Delta(h \cdot h')} \\[10pt] \frac{\text{true}}{q \in \text{run}^\Delta(\epsilon)} \quad \frac{\langle \rangle \xrightarrow{\quad} p \in \Delta \quad p \cdot R \cdot p' \in \text{run}^\Delta(h) \quad q @ p' \rightarrow q' \in \Delta}{q \cdot \langle R \rangle \cdot q' \in \text{run}^\Delta(\langle h \rangle)} \end{array}$$

Note that if $R \in \text{run}^\Delta(h)$ then h can be obtained by removing all states from R , i.e., $\text{proj}_\Sigma(R) = h$.

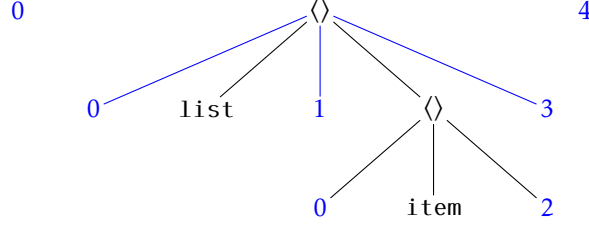


Figure 3.11: The unique run $0 \cdot \langle 0 \cdot \text{list} \cdot 1 \cdot \langle 0 \cdot \text{item} \cdot 2 \rangle \cdot 3 \rangle \cdot 4$ of the dSHA in Figure 3.10 for the nested regular expression [self-list-child-item] on the hedge $\langle \text{list} \cdot \langle \text{item} \rangle \rangle$.

Example 3.27. The dSHA for the nested regular expression [self-list-child-item] in Figure 3.10 on the nested list $\langle \text{list} \cdot \langle \text{item} \rangle \rangle$ has the following successful run:

$$0 \cdot \langle 0 \cdot \text{list} \cdot 1 \cdot \langle 0 \cdot \text{item} \cdot 2 \rangle \cdot 3 \rangle \cdot 4$$

This run is a hedge over a signature extended with states. It can be drawn graphically as any hedge, see Figure 3.11. It is unique since the automaton is deterministic.

A run of $A = (\mathcal{Q}, \Sigma, \Delta, I, F)$ is a Δ -run that starts with some state in I . A run of A is called *successful* if it ends with some state in F .

Lemma 3.28. Any hedge h satisfies that $h \in \mathcal{L}(A)$ if and only if there exists a successful run $R \in \text{run}^\Delta(h)$ of A .

Proof. Straightforward by induction on h . □

3.2.5.1 States of Prefixes

Any run $R \in \text{run}^\Delta(h)$ satisfies $h = \text{proj}_\Sigma(R)$. Note that R annotates all events of the depth first-search of the graph of h by some state. These events do correspond to prefixes of $nw(h)$, so R can be identified with a mapping of prefixes of the nested word $nw(h)$ to states in \mathcal{Q} . Therefore, for any prefix v of $nw(h)$, there exists a unique prefix r of the nested word $nw(R)$ that ends with some state $q \in \mathcal{Q}$ and satisfies $\text{proj}_\Sigma(r) = v$. We then call q the state of R at prefix v .

Example 3.29. Reconsider the run $R = 0 \cdot \langle 0 \cdot \text{list} \cdot 1 \cdot \langle 0 \cdot \text{item} \cdot 2 \rangle \cdot 3 \rangle \cdot 4$ on the hedge

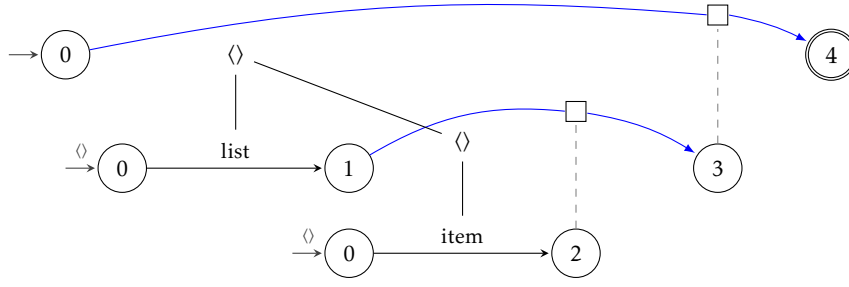


Figure 3.12: A top-down representation of the run the dSHA for the nested regular expression $[self-list-child-item]$ in Figure 3.10 on the hedge $\langle list \cdot \langle item \rangle \rangle$.

$\langle list \cdot \langle item \rangle \rangle$. It corresponds to the following mapping on $prefs(\langle list \rangle \langle item \rangle)$:

$$\left[\begin{array}{ll} \epsilon & / 0, \quad \langle list \langle item \rangle / 2, \\ \langle & / 0, \quad \langle list \langle item \rangle / 3, \\ \langle list & / 1, \quad \langle list \langle item \rangle \rangle / 4 \\ \langle list \langle & / 0 \end{array} \right]$$

So, for instance, run R starts in state 0, eventually goes at prefix $\langle list \langle item \rangle$ to state 2, and then ends for the full nested word $\langle list \langle item \rangle \rangle$ in state 4.

3.2.5.2 Bottom-Up versus Top-Down Vision

Runs of dSHAs can be computed deterministically when traversing the hedge in a bottom-up and left-to-right manner. In other words, the states of the unique run of a hedge can be computed during a left-most bottom up traversal of the hedge.

As argued in Section 2.2.4, the left-most bottom-up traversal of a hedge is part of its top-down traversal. In this thesis, we prefer the top-down vision when running automaton on hedges. For emphasizing the top-down vision, we will often draw runs as in Figure 3.11, alternatively as in Figure 3.12. This way, we separate the contributions of the hedge and of the automaton in a clear manner (without the need of coloring the contribution of the automaton). We added some more processing comments to the top-down representation of the same run in Figure 3.13: the depth-first search of the hedge $\langle list \cdot \langle item \rangle \rangle$ is indicated within the run by a line that is colored first green and then changes to red. This shows that the states of the run can be assigned to the events of the depth-first search, i.e., to the elements of the top-down traversal, or yet alternatively, to the prefixes of the nested word of the hedge. The green parts of the line indicate the events in descending direction, and the red part those events in ascending direction. Note

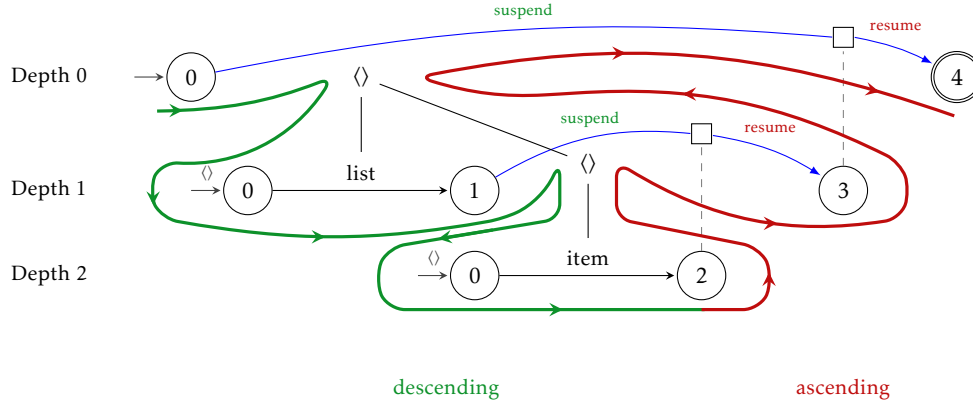


Figure 3.13: Adding processing comments to the top-down vision of the run in Figure 3.12.

that there could be multiple changes of directions, i.e., from green to red and vice versa, even though there is only one in the present example.

The run starts at depth zero before the left-most root (there is only one in this example) in the unique initial state $\rightarrow(0)$. It then encounters a tree constructor, so it must be suspended in state 0 until the subhedge of the tree is evaluated. The suspension is indicated by the blue edge to the suspension box $\rightarrow(0) \square$, which represents a future value that will be computed on the lower level. The run then restarts one level down at depth 1: it has to evaluate the subhedge $h' = \text{list} \cdot \langle \text{item} \rangle$ of the tree, starting with the unique tree initial state of the automaton $\xrightarrow{\langle \rangle}(0)$. For h' , the run reads letter *list* reaching state 1 and must suspend again at $(1) \square$ when encountering a second tree constructor. It then goes down to depth 2 where it is restarted again in state $\xrightarrow{\langle \rangle}(0)$, for evaluating the last subhedge $h'' = \text{item}$. From state 0, the run of h'' eventually ends in state 2 concluding the descending part.

It should be noticed that no information is passed down in the descending phases of any run of any dSHA. Instead, at each step going properly down into some subhedge at a lower level, the run is restarted in the tree initial state, while the run on the upper level will be resumed once the evaluation of the lower level is finished. The only state changes in the descending phases are due to left-to-right processing, when reading letters and staying at the same depth.

Once reaching the last node at depth 2, the run changes direction and becomes ascending. It goes up revisiting the tree constructors, filling the boxes left above with the states computed at the lower level, and consequently triggering the resumption of the runs at higher depths. Therefore, the box at depth 1 is filled with state 2,

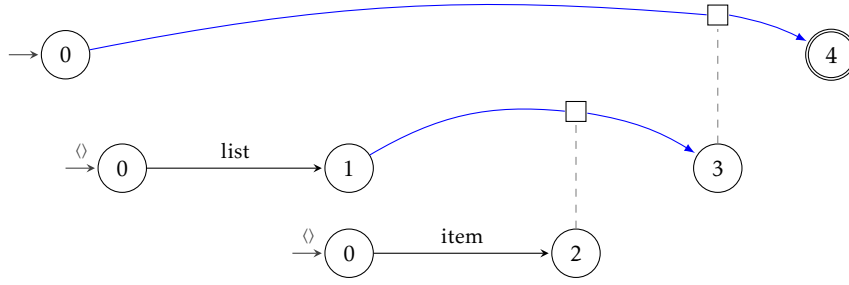


Figure 3.14: A simplified top-down representation of the run the dSHA for the nested regular expression $[\text{self-list-child-item}]$ in Figure 3.10 on the hedge $\langle \text{list} \cdot \langle \text{item} \rangle \rangle$ without tree constructors and child edges.

which we mark by $\textcircled{2} - \square$, giving $\textcircled{1} - \square \rightarrow \textcircled{3}$. This transition is justified by the apply rule $1@2 \rightarrow 3$. Similarly at the top level, the resumption is now triggered, the box is filled with state 3 and the computation finally goes to the final state 4 by using the apply rule $0@3 \rightarrow 4$.

In the top-down drawings of runs, we can safely remove the tree constructors and child edges without losing any information. This leads as to the simplified version of the graphical representation in Figure 3.14.

3.2.6 Determinization

The full determinization of an SHA A extends on that of the NFA in subsection 3.1.7, with the below two inference rules:

$$\frac{\text{true}}{\langle \rangle^{\Delta^{det-f(A)}} = \{\langle \rangle^{\Delta} \mid \langle \rangle^{\Delta} \neq \emptyset\}}$$

$$\frac{Q_1, Q_2 \subseteq \mathcal{Q}^A \quad Q' = \{q' \in \mathcal{Q}^A \mid q_1 @ q_2 \rightarrow q' \in \Delta^A, q_1 \in Q_1, q_2 \in Q_2\} \neq \emptyset}{Q_1 @ Q_2 \rightarrow Q' \in \Delta^{det-f(A)}}$$

3.2.7 Complementation

Analogously to NFAs, for any complete dSHA A , its complement automata \overline{A} can be obtained by flipping the final states.

Moreover, Proposition 3.12 still holds when applied for some dSHA A .

3.2.8 Product and Intersection

The full product $A \times^f B$ of two SHAS A and B can be computed by an extension of the construction for NFAS from Section 3.1.9. It is sufficient to add the following two inference rules:

$$\frac{q \in \langle \rangle^{\Delta^A} \quad p \in \langle \rangle^{\Delta^B}}{(q, p) \in \mathcal{Q}^{A \times^f B}} \quad \frac{q_1 @ q \rightarrow q_2 \in \Delta^A \quad p_1 @ p \rightarrow p_2 \in \Delta^B}{(q_1, p_1) @ (q, p) \rightarrow (q_2, p_2) \in \Delta^{A \times^f B}}$$

As for Nwas the product of SHAS continues to compute the intersection of their languages, so Proposition 3.13 remains true for SHAS too.

3.2.9 Hedge Accessibility

The set of states that are accessible from a state in a subset $Q \subseteq \mathcal{Q}$ through some hedge is:

$$\text{acc}^\Delta(Q) = \{q' \mid q \in Q, q \xrightarrow{h} q' \text{ wrt } \Delta, h \in \mathcal{H}_\Sigma\}$$

For any set Q we can compute the set of accessible states by the least fixed point of a ground Datalog program of size $O(|\Delta| + |Q|)$.

$$\begin{array}{c} \frac{q \in Q}{q \in \text{acc}^\Delta(Q)}. \quad \frac{q \in \langle \rangle^\Delta}{q \in \text{acc}^\Delta(\langle \rangle^\Delta)}. \\[10pt] \frac{q \xrightarrow{a} q' \in \Delta}{q' \in \text{acc}^\Delta(Q) :- q \in \text{acc}^\Delta(Q)}. \quad \frac{q_1 @^\Delta p = q_2}{q_2 \in \text{acc}^\Delta(Q) :- q_1 \in \text{acc}^\Delta(Q), p \in \text{acc}^\Delta(\langle \rangle^\Delta)}. \\[10pt] \frac{q \xrightarrow{a} q' \in \Delta}{q' \in \text{acc}^\Delta(\langle \rangle^\Delta) :- q \in \text{acc}^\Delta(\langle \rangle^\Delta)}. \quad \frac{q_1 @^\Delta p = q_2}{q_2 \in \text{acc}^\Delta(\langle \rangle^\Delta) :- q_1 \in \text{acc}^\Delta(\langle \rangle^\Delta), p \in \text{acc}^\Delta(\langle \rangle^\Delta)}. \end{array}$$

For any $Q \subseteq \mathcal{Q}$, the set $\text{acc}^\Delta(Q)$ can be computed in time $O(|A|)$. This also applies to inverse accessibility:

$$\text{invacc}^\Delta(Q) = \{q' \mid q \in \text{acc}^\Delta(\{q'\}), q \in Q\}$$

This is since $\text{invacc}^\Delta(Q) = \text{acc}^{\Delta^{-1}}(Q)$ where Δ^{-1} is the set of inverse transition rules

for Δ , which is such that for all $q, p, q' \in Q$ and $a \in \Sigma$:

$$\frac{q \xrightarrow{a} q' \in \Delta}{q' \xrightarrow{a} q \in \Delta^{-1}} \quad \frac{q@p \rightarrow q' \in \Delta}{q'@p \rightarrow q \in \Delta^{-1}}$$

3.2.10 Minimization

As illustrated by example in Section 3.2.1, unique minimization does not hold for dSHAS in general. But two subclasses of dSHAS enjoy unique minimization. The first is the subclass of dSHAS where the initial state is equal to the tree initial state. The second is the subclass of 2-level dSHAS.

While our SHA construction from nested regular expressions needs 2-level SHAS, our determinizer for SHAS will produce SHAS where the initial state is equal to the tree initial state. So when considering deterministic SHAS we will work with the first subclass and use the minimization algorithm from [Niehren & Sakho 2021] for their minimization.

3.2.11 Cleaning

The following Datalog program defines the set of useful states of a SHA by using accessibility and co-accessibility. Note that determinism is not needed.

For this, we introduce monadic predicates acc_τ and $coacc_\tau$ for accessibility and co-accessibility for both types $\tau \in \{h, t\}$, depending on whether we talk about the upper hedge level, or the lower tree level. The following monadic ground Datalog program defines these predicates:

$$\begin{array}{c} \frac{q \in I}{acc_h(q)}. \quad \frac{q \in \langle \rangle^\Delta}{acc_t(q)}. \quad \frac{q \in F}{coacc_h(q)}. \quad \frac{q@p \rightarrow q' \in \Delta}{coacc_t(p) :- acc_h(q), coacc_h(q')}. \\[10pt] \frac{q@p \rightarrow q' \in \Delta \quad \tau \in \{h, t\}}{coacc_\tau(q) :- acc_t(p), coacc_\tau(q'). \quad acc_\tau(q') :- acc_\tau(q), acc_t(p)}. \quad \frac{q \xrightarrow{a} q' \in \Delta \quad \tau \in \{h, t\}}{coacc_\tau(q) :- coacc_\tau(q'). \quad acc_\tau(q') :- acc_\tau(q)}. \end{array}$$

$$\frac{\tau \in \{h, t\}}{\text{useful}(q) :- \text{acc}_\tau(q), \text{coacc}_\tau(q).}$$

We define the accessibility cleaning, the co-accessibility cleaning, and the trimming of A as follows:

$$\begin{aligned} \text{acc-clean}(A) &= \text{clean}(A, \{q \in \mathcal{Q} \mid \text{acc}_\tau(q), \tau \in \{h, t\}\}) \\ \text{coacc-clean}(A) &= \text{clean}(A, \{q \in \mathcal{Q} \mid \text{coacc}_\tau(q), \tau \in \{h, t\}\}) \\ \text{trim}(A) &= \text{coacc-clean}(\text{acc-clean}(A)) \end{aligned}$$

So $\text{trim}(A) = \text{clean}(A, \{q \in \mathcal{Q} \mid \text{useful}(q)\})$ keeps only the useful states of the SHA, i.e., those that are used in some successful run.

3.2.12 Infinitary SHAS

In analogy to the definition of NFA^∞ s in Section 3.1.13, we can define infinitary SHAS or equivalently SHA^∞ s such as SHAS while dropping all finiteness restrictions. In other words, any SHA^∞ equips some NFA^∞ with a set of set of apply rules and tree initial states.

3.2.13 Else Rules

An SHA with else rules is an SHA^∞ with finite sets of states and transitions rules, but possibly an infinite signature, that is extended by typed or untyped else rules. So any SHA with else rule extends on some NFA with else rules (see Section 3.1.14 by apply rules and tree initial states.

3.2.14 Related Automata Models

We relate the model of SHAS to the existing automata models for unranked trees, hedges, and nested words in the literature.

3.2.14.1 Standard Hedge Automata

Standard hedge automata [Thatcher 1967, Comon *et al.* 2007, Pair & Quéré 1968, Murata 2000] operate on labeled hedges in a bottom-up manner similarly to SHAS. Also they have the same expressiveness, but horizontal languages are specified dif-

ferently, leading to a problematic notion of determinism [Martens & Niehren 2007]. For this reason, unique minimization fails for deterministic standard hedge automata. Still, they have the same expressiveness as SHAS when restricted to labeled hedges.

3.2.14.2 Standard Tree Automata

Syntactically, any SHA A is a standard tree automaton over the following ranked signature:

- a unary symbol for any letter $a \in \Sigma$,
- a binary symbol $@$, and
- a constant $\langle \rangle$.

Semantically, however, there is no perfect general correspondence. Only for subclasses a perfect correspondence can be made up via binary encoding. This was shown in [Niehren & Sakho 2021] for the subclass of dSHAS with $I = \langle \rangle^\Delta$. In [Niehren 2024], it could also be shown for the subclass of multi-module dSHAS. This leads to unique minimization results for both subclasses of deterministic SHAS.

3.2.14.3 Bojanczyk’s Forest Automata

Standard hedge automata also have the same expressiveness as Bojanczyk’s forest automata (see Section 3.3 of [Bojanczyk & Walukiewicz 2008]). These are based on the idea of transition monoids, rather than on the idea of transition rules such as SHAS or finite state automata or Neuman’s and Seidl’s forest automata. As a consequence, however, there is an exponential difference in succinctness between SHAS and Bojanczyk’s forest automata [Niehren 2024].

3.3 Downward Stepwise Hedge Automata (SHA^\downarrow s)

The bottom-up evaluation of SHAS cannot pass any information top-down. Even with the top-down vision of runs of SHAS— as presented in the previous section — no information is passed down when a SHAS moves down by one level. Instead, the computations is always restarted in some tree initial state.

We therefore propose an extension of SHAS to SHA^\downarrow s, that adds the ability to SHAS to pass information top-down. SHA^\downarrow s are basically equal to Neumann and

Seidl's pushdown forest automata [Neumann & Seidl 1998] but lifted from (labeled) forests to (unlabeled) hedges. These, in turn, are closely related to Nwas too, as already noticed in [Gauwin *et al.* 2008] and discussed in Section 3.3.8.

3.3.1 Syntax

SHA^\downarrow s are like SHAs except that tree initial states are generalized to tree initial rules. Thereby, the state in which the evaluation of the hedge in a subtree restarts may depend on the state of its parent.

Definition 3.30. A downward stepwise hedge automaton (SHA^\downarrow) is a tuple $A = (\Sigma, \mathcal{Q}, \Delta, I, F)$ where Σ and \mathcal{Q} are finite sets, $I, F \subseteq \mathcal{Q}$, and $\Delta = (\Delta', \langle \rangle^\Delta, @^\Delta)$. Furthermore, $\Delta' \subseteq (\mathcal{Q} \times \Sigma) \times \mathcal{Q}$, $\langle \rangle^\Delta \subseteq \mathcal{Q} \times \mathcal{Q}$, and $@^\Delta \subseteq (\mathcal{Q} \times \mathcal{Q}) \times \mathcal{Q}$. A SHA^\downarrow is called deterministic or equivalently a $d\text{SHA}^\downarrow$ if I contains at most one element, and all relations $\langle \rangle^\Delta$, Δ' , and $@^\Delta$ are partial functions.

The only difference to SHAs is the form of the tree opening rules. If $(q, q') \in \langle \rangle^\Delta$ then we have a tree initial rule that we denote as $q \xrightarrow{\langle \rangle} q' \in \Delta$. So here the state q' where the evaluation of a subhedge starts depends on the state q of the parent.

Graphically, a SHA^\downarrow is drawn like an SHA, except that tree initial rules are represented by edges $\textcircled{q} \xrightarrow{\langle \rangle} \textcircled{q'}$.

Example 3.31. A $d\text{SHA}^\downarrow$ recognizing the language of expression [self-list-child-item] from Example 3.24 is given in Figure 3.15. The same language was recognized by the $d\text{SHA}$ in Figure 3.10. The $d\text{SHA}^\downarrow$, however, passes level information top-down. Its states on level 0 have 0 primes, states on level 1 have 1 prime, and states on level 2 and more have 2 primes. Furthermore, each state has a type which is a state of the $d\text{SHA}$ in Figure 3.10. For instance, the state $0''$ of the $d\text{SHA}^\downarrow$ has type 0 and 2 primes. For example, the tree opening rule $\textcircled{1'} \xrightarrow{\langle \rangle} \textcircled{0''}$ means that the automaton in a node of type 1 at level 1 goes to a node of type 0 at level 2. We notice that the automaton needs to inspect only 2 levels of the input hedge, in order to verify whether the first subtree has label `list` on level 1 and a child with label `item` on level 2.

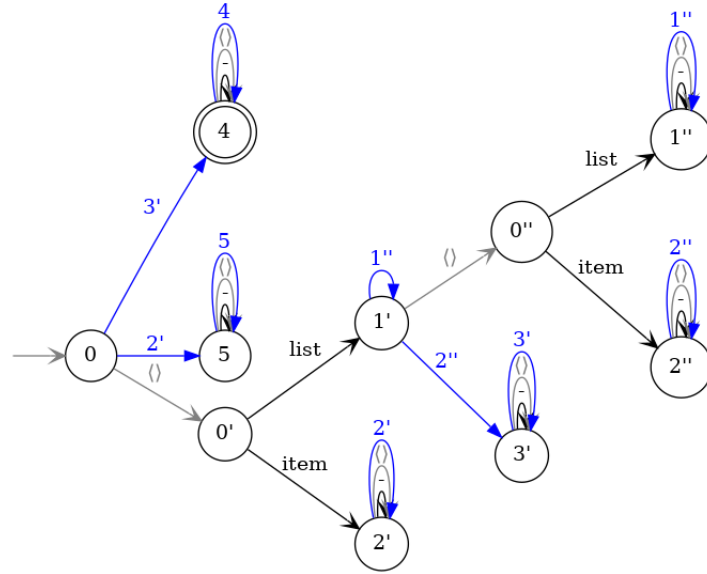


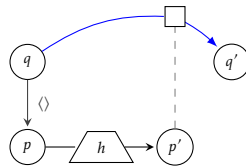
Figure 3.15: A deterministic SHA^\downarrow recognizing the language of the nested regular expression $[\text{self-list-child-item}] = \langle \text{list} \cdot \text{N-List} \cdot \langle \text{item} \cdot \text{N-List} \rangle \cdot \text{N-List} \rangle \cdot \text{N-List}$.

3.3.2 Semantics

The definition of transition steps for SHA^\downarrow s differs from that of SHAs only in the usage of opening rules in the following inference rule:

$$\frac{q \xrightarrow{\langle \rangle} p \in \Delta \quad p \xrightarrow{h} p' \text{ wrt } \Delta \quad q @ p' \rightarrow q' \in \Delta}{q \xrightarrow{\langle h \rangle} q' \text{ wrt } \Delta}$$

This means that the evaluation of the subhedge h starts in some state p such that $q \xrightarrow{\langle \rangle} p \in \Delta$.



So the restart state p that is chosen may now depend on the state q above. This is how finite state information is passed top-down by SHA^\downarrow s. SHAs in contrast, operate purely bottom-up and left-to-right.

3.3.3 Completion

Completion for SHA^\downarrow s is done similarly as for SHAS , except that now we need rules to the sink when starting subtrees, where Δ is original set of transition rules and Δ' that of the completion:

$$\frac{q \in \mathcal{Q} \cup \{\text{sink}\} \quad \nexists q' \in \mathcal{Q}. q \xrightarrow{\langle \rangle} q' \in \Delta}{q \xrightarrow{\langle \rangle} \text{sink} \in \Delta'}$$

3.3.4 Runs

The notion of runs can be adapted straightforwardly from SHAS to SHA^\downarrow s. When in state q , it is sufficient to restart the computation in subhedges on the state p such that $q \xrightarrow{\langle \rangle} p \in \Delta$. In this way, finite state information is passed down (while for SHAS some tree initial is to be chosen that is independent of q). The only rule of runs to be changed is the following:

$$\frac{q \xrightarrow{\langle \rangle} p \in \Delta \quad p \cdot R \cdot p' \in \text{run}^\Delta(h) \quad q @ p' \rightarrow q' \in \Delta}{q \cdot \langle R \rangle \cdot q' \in \text{run}^\Delta(\langle h \rangle)}$$

Example 3.32. An example of a run of the $d\text{SHA}^\downarrow$ in Figure 3.15 is shown in Figure 3.16 for the hedge $\langle \text{list} \cdot \langle \text{list} \cdot h_1 \rangle \cdot \langle \text{item} \cdot h_2 \rangle \rangle$ with arbitrary $h_1, h_2 \in \mathcal{H}_\Sigma$. It naturally shows the top-down traversal of the SHA^\downarrow on the hedge. The difference with the run of SHAS concerns the tree initial rules used to pass level information top-down as already described in the Example 3.31.

For instance, consider the tree whose first node is labeled `list`: reading such a tree at depth 1 is done through the sequence of states $0, 0', 1'$, whereas at depth 2, the same tree is read through a different sequence $1', 0'', 1''$. Finally, going beyond depth 2, it read in states $1''$ or $2''$ depending whether the first node of the parent tree is respectively labeled `list` or `item`.

More generally, notice the parts marked in red, where all nodes of the arbitrary subhedges h_1 and h_2 does not change states after evaluation and stay respectively in states $1''$ and $2''$. This highlights a key notion of our thesis that will be introduced going forward: Subhedge irrelevance.

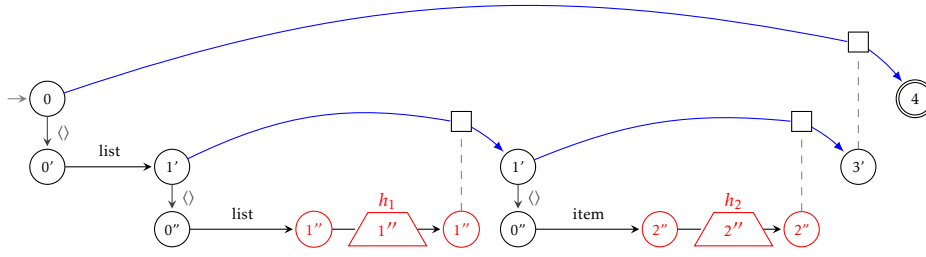


Figure 3.16: A successful run of the SHA^\downarrow in Figure 3.15 on the hedge $\langle \text{list} \cdot \langle \text{list} \cdot h_1 \rangle \cdot \langle \text{item} \cdot h_2 \rangle \rangle$.

3.3.5 Conversion between SHAS and SHA^\downarrow s

SHA^\downarrow s have the same expressiveness as SHAS . Basically, it is for the same reason why NWS have the same expressiveness as SHAS (see e.g. [Niehren & Sakho 2021]). We shall describe how to get from a SHA to an equivalent SHA^\downarrow and vice versa

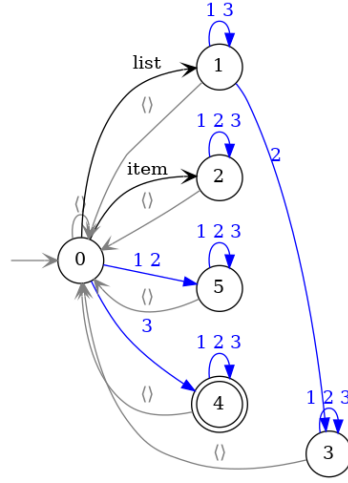
Downward Insertion Any SHA $A = (\Sigma, \mathcal{Q}, \Delta, I, F)$ can be mapped to a $\text{SHA}^\downarrow A^{\text{down}} = (\Sigma, \mathcal{Q}, \Delta^{\text{down}}, I, F)$ while preserving the runs and determinism. The only change of Δ^{down} compared to Δ is described by the following rule:

$$\frac{\langle \rangle \xrightarrow{\quad} p \in \Delta \quad q \in \mathcal{Q}}{q \xrightarrow{\langle \rangle} p \in \Delta^{\text{down}}}$$

Independently of the current state $q \in \mathcal{Q}$, the $\text{SHA}^\downarrow A^{\text{down}}$ can start the evaluation of the subhedge of a subtree in any open tree state $p \in \langle \rangle^\Delta$. For instance, the $d\text{SHA}^\downarrow A^{\text{down}}$ for $d\text{SHA}$ A from Figure 3.10 from the introduction is given in Figure 3.17. It is clear by this conversion, no information has been passed down in contrast to the example in Figure 3.15 and the run on this automaton will be the same as the its equivalent SHA , where it restarts in the same tree initial state 0 at each encounter of a tree constructor.

Downward Elimination Conversely, we can convert any $d\text{SHA}^\downarrow A$ into an equivalent $\text{SHA} \text{elim}^\downarrow(A)$ while possibly introducing nondeterminism as follows:

$$\begin{array}{l} I^{\text{elim}^\downarrow} = I \times \mathcal{Q} \\ F^{\text{elim}^\downarrow} = F \times \mathcal{Q} \end{array} \quad \frac{q \xrightarrow{\langle \rangle} q' \in \Delta}{\langle \rangle \xrightarrow{\quad} (q, q') \in \Delta^{\text{elim}^\downarrow}}$$

Figure 3.17: The $\text{SHA}^\downarrow A^{\text{down}}$ for the dSHA A in Figure 3.10.

$$\frac{q \xrightarrow{a} q' \in \Delta \quad r \in \mathcal{Q}}{(r, q) \xrightarrow{a} (r, q') \in \Delta^{\text{elim}^\downarrow}} \quad \frac{q@p \rightarrow q' \in \Delta \quad r \in \mathcal{Q}}{(r, q)@(q, p) \rightarrow (r, q') \in \Delta^{\text{elim}^\downarrow}}$$

Proposition 3.33. $\mathcal{L}(A) = \mathcal{L}(\text{elim}^\downarrow(A))$.

The construction is analogous to the conversion of NWAS to SHAS [Niehren & Sakho 2021] or to hedge automata [Gauwin *et al.* 2008]. The correctness proofs for these compilers are standard.

3.3.6 Determinization

The determinization procedure for SHA^\downarrow s is more complicated than for SHAS, since SHA^\downarrow s can pass information top-down and not only bottom-up and left-to-right. Determinization therefore does not only rely on the pure subset construction, but also uses pairs of states in the subsets, basically in the same way as for nested word automata (NWAS) [Okhotin & Salomaa 2014, von Braunmühl & Verbeek 1985]. This is needed to deal with the stack of states that were seen above. Therefore, each determinization step may cost time $O(|A|^5)$ as stated for instance in [Debarbieux *et al.* 2015]. The upper time bound for membership testing for SHA^\downarrow s is thus in time $O(|h||A|^5)$, and no longer combined linear as it is for SHAS.

On the other hand, one can take the approach done for NWAS in [Niehren & Sakho 2021] and apply it to SHA^\downarrow . It would require to transform the SHA^\downarrow to its equivalent SHA by downward elimination. Since the latter procedure may introduce nondeterminism, the resulting SHA should be determinized and then

converted back to SHA^\downarrow . This approach has proved to be efficient in practice and the resulting automata did not explode in size as with direct determinization. In any case, and during the course of this thesis, we do not attempt to directly determinize a SHA^\downarrow but rather get the needed parts through the determinization of SHA .

3.3.7 Minimization

Unique minimization fails for $d\text{SHA}^\downarrow$ s for two reasons. First, since it fails for $d\text{SHAS}$ (as discussed in Example 3.23), and second, since it fails for deterministic multiway automata.

3.3.8 Relationship to NwAs

SHA^\downarrow s are closely related to Neumann and Seidl's pushdown forest automata, which in turn are closely related to nested word automata, as first noticed in [Gauwin *et al.* 2008]. We here present the relationship between SHA^\downarrow s and NwAs in a direct manner.

3.3.8.1 Syntax

A nested word automata reads nested words rather than hedges. So it provides rules for all letters of nested words.

Definition 3.34. A nested word automata (NwA) is a tuple $(\Sigma, \mathcal{Q}, \Gamma, \Delta, I, F)$, where Σ , Γ and \mathcal{Q} are sets, $I, F \subseteq \mathcal{Q}$, and $\Delta = (\Delta', \langle^\Delta, \rangle^\Delta)$ contains relations: $\Delta' \subseteq \mathcal{Q} \times \Sigma \times \mathcal{Q}$, $\langle^\Delta \subseteq \mathcal{Q} \times (\Gamma \times \mathcal{Q})$ and $\rangle^\Delta \subseteq (\mathcal{Q} \times \Gamma) \times \mathcal{Q}$. A NwA is deterministic or equivalently a $d\text{NwA}$ if I contains at most one element and all above relations are partial functions.

The elements of Γ are called stack symbols. The transition rules in Δ again have three forms: Internal rules $q \xrightarrow{a} q'$ in Δ as for SHAS , opening rules $q \xrightarrow{\langle^\Delta \gamma} q'$ in Δ if $\langle^\Delta(q) = (q', \gamma)$ and closing rules $q \xrightarrow{\gamma \rangle^\Delta} q'$ in Δ if $\rangle^\Delta(q, \gamma) = q'$.

3.3.8.2 Semantics

We can define transitions $q \xrightarrow{v} q'$ wrt Δ for all NwAs $A = (\Sigma, \mathcal{Q}, \Gamma, \Delta, I, F)$ with states $q, q' \in \mathcal{Q}$ and nested words $v \in \mathcal{N}_\Sigma$. These are such that for all $q, q', p, p' \in \mathcal{Q}$, $a \in \Sigma$,

and $v, v' \in \mathcal{N}_\Sigma$:

$$\frac{q \xrightarrow{a} q' \in \Delta}{q \xrightarrow{a} q' \text{ wrt } \Delta} \quad \frac{q \xrightarrow{v} q' \text{ wrt } \Delta \quad q' \xrightarrow{v'} q'' \text{ wrt } \Delta}{q \xrightarrow{v \cdot v'} q'' \text{ wrt } \Delta}$$

$$\frac{q \in \mathcal{Q}}{q \xrightarrow{\epsilon} q \text{ wrt } \Delta} \quad \frac{q \xrightarrow{\langle \downarrow \gamma \rangle} p \in \Delta \quad p \xrightarrow{v} p' \text{ wrt } \Delta \quad p' \xrightarrow{\gamma \uparrow} q' \in \Delta}{q \xrightarrow{\langle v \rangle} q' \text{ wrt } \Delta}$$

Note that the inference rules for NWA transitions are in full analogy to the inference rules for SHA^\downarrow transitions, when adopting the restriction $\Gamma = \mathcal{Q}$. It is sufficient to identify hedges h with their nested words $nw(h)$, open tree rules $q \xrightarrow{\langle \rangle} p \text{ wrt } \Delta$ with opening rules $q \xrightarrow{\langle \downarrow q \rangle} p \in \Delta$ that push the current state to the stack, and apply rules $q@p \rightarrow q'$ with closing rules $q \xrightarrow{\gamma \uparrow p} q' \in \Delta$. The language of nested words recognized by a NWA A can be defined literally as for SHA^\downarrow :

$$\mathcal{L}(A) = \{v \in \mathcal{N}_\Sigma \mid q_0 \in I, q_0 \xrightarrow{v} q_1 \text{ wrt } \Delta, q_1 \in F\}$$

Lemma 3.35. $\mathcal{L}(A^{nwa}) = nw(\mathcal{L}(A))$.

Proof. The transitions of a $\text{SHA}^\downarrow A$ on a hedge h and of the NWA A^{nwa} on the nested word $nw(h)$ can be identified. \square

3.3.8.3 Streaming Evaluator

A streaming evaluator for dNWA can be defined by a visibly pushdown machine (or by compilation to a visibly pushdown automaton [Alur & Madhusudan 2004] known earlier as input-driven pushdown automata [Alur & Madhusudan 2004, Okhotin & Salomaa 2014]).

A configuration of this machine is a pair of a state and a stack, i.e., a word of stack symbols:

$$\mathcal{K} = \mathcal{Q} \times \Gamma^*$$

For any factor of some nested word in \mathcal{N}_Σ , i.e., for any $v \in \hat{\Sigma}^*$, we define streaming transitions on configurations

$$\llbracket v \rrbracket_{str} : \mathcal{K} \hookrightarrow \mathcal{K}$$

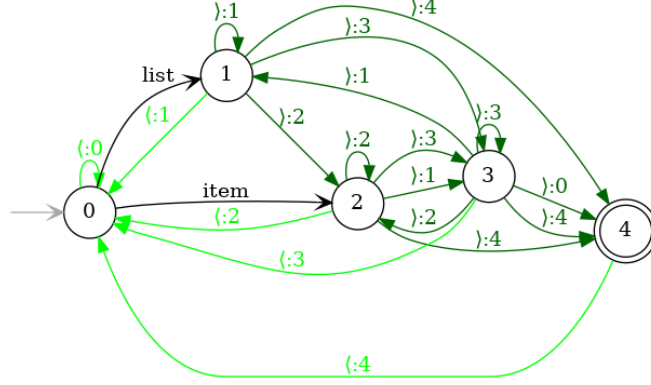


Figure 3.18: The dNWA B^{nwa} for the $d\text{SHA}^\downarrow B = A^{nwa}$ in Figure 3.17, where A is the earliest dSHA in Figure 3.10 for the nested regular expression [self-list-child-item].

such that for all $q, q' \in \mathcal{Q}$, $v, v' \in \hat{\Sigma}^*$, $\gamma \in \Gamma$ and $\sigma \in \Gamma^*$:

$$\begin{aligned}
 \llbracket a \rrbracket_{str}(q, \sigma) &= (a^\Delta(q), \sigma) & \llbracket \langle \rrbracket_{str}(q, \sigma) &= (q', \sigma \cdot \gamma) \text{ where } q \xrightarrow{\langle \downarrow \gamma} q' \\
 \llbracket \epsilon \rrbracket_{str}(q, \sigma) &= (q, \sigma) & \llbracket \rangle \rrbracket_{str}(q, \sigma \cdot \gamma) &= (q', \sigma) \text{ where } q \xrightarrow{\rangle \uparrow \gamma} q' \\
 \llbracket v \cdot v' \rrbracket_{str}(q, \sigma) &= \llbracket v' \rrbracket_{str}(\llbracket v \rrbracket_{str}(q, \sigma))
 \end{aligned}$$

3.3.8.4 From SHA^\downarrow s to NWAs

A streaming evaluator for $d\text{SHA}^\downarrow$ s can be derived from that of dNWAs by compiling the former to the latter. More generally, we can also compile nondeterministic SHA^\downarrow s to nondeterministic NWAs while preserving determinism.

For any $\text{SHA}^\downarrow A = (\Sigma, \mathcal{Q}, \Delta, I, F)$ we can define an NWA $A^{nwa} = (\Sigma, \mathcal{Q}, \Gamma, \Delta^{nwa}, I^{nwa}, F^{nwa})$ while preserving determinism, such that $\Gamma = \mathcal{Q}$ and such that Δ^{nwa} contains for all $a \in \Sigma$ and $q, p \in \mathcal{Q}$ the transition rules:

$$\frac{q \xrightarrow{a} q' \in \Delta}{q \xrightarrow{a} q' \in \Delta^{nwa}} \quad \frac{q \xrightarrow{\langle \rangle} q' \in \Delta}{q \xrightarrow{\langle \downarrow q} q' \in \Delta^{nwa}} \quad \frac{q @ p \rightarrow q' \in \Delta}{p \xrightarrow{\rangle \uparrow q} q' \in \Delta^{nwa}}$$

For instance, the dNWA B^{nwa} in Figure 3.18 is obtained from the $d\text{SHA}^\downarrow B = A^{down}$ in Figure 3.17. It captures the nested regular expression [self-list-child-item].

3.3.8.5 From NwAs to SHA^\downarrow s

Conversely, any NwA $A = (\mathcal{Q}, \Sigma, \Delta, I, F)$ can be compiled back to a $\text{SHA}^\downarrow A^{\text{sha}^\downarrow} = (\Sigma, \mathcal{Q}, \Delta^{\text{sha}^\downarrow}, I, F)$, which has the same transitions up to mapping nested words v to hedges $\text{hdg}(v)$, while preserving determinism:

$$\frac{q \xrightarrow{a} q' \in \Delta}{q \xrightarrow{a} q' \in \Delta^{\text{sha}^\downarrow}} \quad \frac{q \xrightarrow{\langle \downarrow \gamma \rangle} p \in \Delta}{q \xrightarrow{\langle \rangle} p \in \Delta^{\text{sha}^\downarrow}} \quad \frac{p \xrightarrow{\rangle \uparrow q} q' \in \Delta \quad q \in \mathcal{Q}}{q @ p \rightarrow q' \in \Delta^{\text{sha}^\downarrow}}$$

This translation is in linear time for deterministic NwAs, since for these, the stack symbol γ pushed when opening $q \xrightarrow{\langle \downarrow \gamma \rangle} p \in \Delta$ is determined by the current state q .

Lemma 3.36. *If A is a deterministic NwA then $(A^{\text{sha}^\downarrow})^{\text{nwa}} = A$.*

For this reason, there is no essential difference between $d\text{SHA}^\downarrow$ s and $d\text{NwAs}$ when it comes to their streaming evaluator, also with respect to complexity issues. For nondeterministic NwAs, however, note the translation to SHA^\downarrow s may be in quadratic time. So the correspondence between NwAs and SHA^\downarrow s is less perfect.

Also notice that the graphs of both kinds of automata are drawn differently, similarly to the graphs of hedges and nested words. So the perspectives taken are not the same: Hedge automata may seem more natural when focusing on in-memory processing, while nested word automata are slightly more streaming oriented. Still there is no essential difference between $d\text{SHA}^\downarrow$ s and $d\text{NwAs}$.

The main advantage of SHA^\downarrow s is their syntactical similarity to the bottom-up model of SHAs, for which determinization is less problematic than for NwAs. As stated earlier, SHAs correspond to weak single-entry NwAs. What is not yet clear is whether the usual determinization algorithm for NwAs restricted to weak single entry NwAs is bisimilar to the determinization algorithm for SHAs.

3.4 Membership Testing

The membership $h \in \mathcal{L}(A)$ can be decided for any $d\text{SHA}^\downarrow A$ and hedge h by computing the unique transition (or run) of A on h starting from I , if it exists, and testing whether it ends in a final state. This can also be applied to any $d\text{SHA} B$ via conversion to a $d\text{SHA}^\downarrow A = B^{\text{down}}$.

We next show how membership can be tested during a top-down traversal either in-memory or in streaming mode.

We recall that the top-down traversals of hedges have top-down, left-to-right, and bottom-up steps. When applied to some $d\text{SHA}^\downarrow A = B^{\text{down}}$ obtained from some $d\text{SHA} B$, note that the top-down steps $q \xrightarrow{\langle \rangle} \langle \rangle^B \in \Delta^A$ will always restart the computation in the same state $\langle \rangle^B$, so no information from the current state q is ever passed down. Hence, all the work will be done during bottom-up and left-to-right steps of the $d\text{SHA} B$ during the top-down traversal.

3.4.1 In-Memory

Let $A = (\Sigma, \mathcal{Q}, \Delta, I, F)$ be a $d\text{SHA}^\downarrow$ and $h \in \mathcal{H}_\Sigma$ a hedge. For the deterministic in-memory evaluation of A one h , we define a partial function

$$\llbracket h \rrbracket^\Delta = \llbracket h \rrbracket : \mathcal{Q} \hookrightarrow \mathcal{Q}$$

recursively on the structure of h , such that for all $q \in \mathcal{Q}$, $a \in \Sigma$, and $h, h' \in \mathcal{H}_\Sigma$:

$$\begin{aligned} \llbracket \epsilon \rrbracket(q) &= q & \llbracket a \rrbracket(q) &= a^\Delta(q) \\ \llbracket h \cdot h' \rrbracket(q) &= \llbracket h' \rrbracket(\llbracket h \rrbracket(q)) & \llbracket \langle h \rangle \rrbracket(q) &= q @^\Delta \llbracket h \rrbracket(\langle \rangle^\Delta(q)) \end{aligned}$$

The next lemma relates the deterministic transitions of $d\text{SHA}^\downarrow$ s to their deterministic evaluation, showing that they are operating in the same way.

Lemma 3.37. *For any $d\text{SHA}^\downarrow A = (\Sigma, \mathcal{Q}, \Delta, I, F)$, hedge $h \in \mathcal{H}_\Sigma$, and states $q, q' \in \mathcal{Q}$:*

$$q \xrightarrow{h} q' \text{ wrt } \Delta \Leftrightarrow q' = \llbracket h \rrbracket^\Delta(q)$$

Proof. By induction on the structure of hedge h . □

As a consequence a hedge h is accepted by a $d\text{SHA}^\downarrow A$ if and only if the state $\llbracket h \rrbracket(I)$ is well-defined and belongs to F .

Proposition 3.38 (Efficiency of in-memory membership testing). *For any $d\text{SHA}^\downarrow A$ and hedge h , language membership $h \in \mathcal{L}(A)$ can be decided in-memory in time and space of $O(|h| + m)$ where m is the overall size of A .*

Proof. Suppose that the graph of h is stored in-memory. We first compute hash tables for Δ , assuming perfect hashing, in time $O(|A|)$. We can then run the recursive evaluator to compute $\llbracket h \rrbracket^\Delta(I)$. This requires to compute at most one state transition for each node of the graph of h , each requires constant time. So the recursive

computation of $\llbracket h \rrbracket^\Delta(I)$ costs time $O(|h|)$. Finally, it is to test whether $\llbracket h \rrbracket^\Delta(I) \in F$ by Lemma 3.37, which is also in constant time. \square

3.4.2 Streaming

We next adapt the deterministic membership tester of $d\text{SHA}^\downarrow$ s so that it can read the nested word of the input hedge in streaming mode, i.e., once from the left to the right. So at each time point, it will have to read a prefix of a nested word, and store a configuration consisting of a state and a stack, which is a word of states. It then reads further letters, so a factor of a nested word, leading to new configurations after each step.

As we will show in Lemma 3.41 the same streaming evaluator can be inherited from dNwas and the compilation of $d\text{SHA}^\downarrow$ to dNwas . For reasoning about complexity, we prefer to present a direct construction though.

A factor of a nested word over Σ may be any word with parenthesis, so a word in $\hat{\Sigma}^*$ where $\hat{\Sigma} = \Sigma \cup \{\langle, \rangle\}$. The deterministic streaming evaluator for a factor of some nested word $v \in \hat{\Sigma}^*$ is a visibly pushdown machine with configurations of the following type:

$$\llbracket v \rrbracket_{str} : \mathcal{K} \hookrightarrow \mathcal{K} \text{ where } \mathcal{K} = \mathcal{Q} \times \mathcal{Q}^*$$

It is defined such such that for all $p, q \in \mathcal{Q}$, $a \in \Sigma$, $\sigma \in \mathcal{Q}^*$ and $v, v' \in \hat{\Sigma}^*$:

$$\begin{aligned} \llbracket a \rrbracket_{str}(q, \sigma) &= (a^\Delta(q), \sigma) & \llbracket \langle \rrbracket_{str}(q, \sigma) &= (\langle^\Delta(q), \sigma \cdot q) \\ \llbracket \epsilon \rrbracket_{str}(q, \sigma) &= (q, \sigma) & \llbracket \rangle \rrbracket_{str}(p, \sigma \cdot q) &= (q @^\Delta p, \sigma) \\ \llbracket v \cdot v' \rrbracket_{str}(q, \sigma) &= \llbracket v' \rrbracket_{str}(\llbracket v \rrbracket_{str}(q, \sigma)) \end{aligned}$$

When reading a letter $a \in \Sigma$, the stack is left unchanged and the state q is changed to $a^\Delta(q)$ by applying a letter rule of Δ . When the opening parenthesis \langle is read, the current state q is pushed onto the stack and the computation continues in $\langle^\Delta(q)$. So an open tree rules got applied. When a closing parenthesis is read, the top most state p is popped from the stack and the computation continues in $q @^\Delta p$.

Lemma 3.39. *For any hedge $h \in \mathcal{H}_\Sigma$: $\llbracket nw(h) \rrbracket_{str}(q, \epsilon) = (\llbracket h \rrbracket(q), \epsilon)$.*

Proof. More generally, we prove for any $d\text{SHA}^\downarrow A = (\Sigma, \mathcal{Q}, \Delta, I, F)$, hedge $h \in \mathcal{H}_\Sigma$, $q \in \mathcal{Q}$ and stack $\sigma \in \mathcal{Q}^*$ that:

$$\llbracket nw(h) \rrbracket_{str}(q, \sigma) = (\llbracket h \rrbracket(q), \sigma)$$

The proof is by induction on the structure of h . Let $\sigma \in \Gamma^*$ be a stack and $q \in \mathcal{Q}$ be a state. We distinguish four cases depending on the form of h :

Case $h = \epsilon$. So $nw(h) = \epsilon$, we then have:

$$\llbracket nw(h) \rrbracket_{str}(q, \sigma) = \llbracket \epsilon \rrbracket_{str}(q, \sigma) = (q, \sigma) = (\llbracket \epsilon \rrbracket(q), \sigma) = (\llbracket h \rrbracket(q), \sigma)$$

Case $h = a$. With $nw(h) = a$, we have :

$$\llbracket nw(h) \rrbracket_{str}(q, \sigma) = \llbracket a \rrbracket_{str}(q, \sigma) = (a^\Delta(q), \sigma) = (\llbracket a \rrbracket(q), \sigma) = (\llbracket h \rrbracket(q), \sigma)$$

Case $h = h_1 \cdot h_2$. Thus, $nw(h) = nw(h_1) \cdot nw(h_2)$. The induction hypothesis applied to h_1 yields $\llbracket nw(h_1) \rrbracket_{str}(q, \sigma) = (\llbracket h_1 \rrbracket(q), \sigma)$. Hence:

$$\begin{aligned} \llbracket nw(h) \rrbracket_{str}(q, \sigma) &= \llbracket nw(h_1) \cdot nw(h_2) \rrbracket_{str}(q, \sigma) \\ &= \llbracket nw(h_2) \rrbracket_{str}(\llbracket nw(h_1) \rrbracket_{str}(q, \sigma)) \quad \text{streaming transition} \\ &= \llbracket nw(h_2) \rrbracket_{str}(\llbracket h_1 \rrbracket(q), \sigma) \quad \text{induction hypothesis on } h_1 \\ &= (\llbracket h_2 \rrbracket(\llbracket h_1 \rrbracket(q)), \sigma) \quad \text{induction hypothesis on } h_2 \\ &= (\llbracket h_1 \cdot h_2 \rrbracket(q), \sigma) \quad \text{in-memory transition} \\ &= (\llbracket h \rrbracket(q), \sigma) \end{aligned}$$

Case $h = \langle h_1 \rangle$. We have $nw(h) = \langle \cdot nw(h_1) \cdot \rangle$, thus:

$$\begin{aligned} \llbracket nw(h) \rrbracket_{str}(q, \sigma) &= \llbracket \langle \rangle \rrbracket_{str}(\llbracket nw(h_1) \rrbracket_{str}(\llbracket \langle \rangle \rrbracket_{str}(q, \sigma))) \quad \text{streaming transition} \\ &= \llbracket \langle \rangle \rrbracket_{str}(\llbracket nw(h_1) \rrbracket_{str}(\langle \rangle^\Delta(q), \sigma \cdot q)) \quad \text{streaming transition} \\ &= \llbracket \langle \rangle \rrbracket_{str}(\llbracket h_1 \rrbracket(\langle \rangle^\Delta(q)), \sigma \cdot q) \quad \text{induction hypothesis for } h_1 \\ &= (q @^\Delta \llbracket h_1 \rrbracket(\langle \rangle^\Delta(q)), \sigma) \quad \text{streaming transition} \\ &= (\llbracket \langle h_1 \rangle \rrbracket(q), \sigma) \quad \text{in-memory transition} \\ &= (\llbracket h \rrbracket(q), \sigma) \end{aligned}$$

□

Proposition 3.40 (Efficiency of streaming membership testing). *For any $d\text{SHA}^\downarrow A$ and hedge h , the streaming membership of $h \in \mathcal{L}(A)$ can be decided in time $O(1)$ per event after a preprocessing time in $O(m)$ and with memory of size in $O(\text{depth}(h) + m)$ where m is the overall size of A .*

So the overall time for streaming membership with deterministic SHA^\downarrow s A on hedges h is in time $O(|h| + m)$. The time upper bound is the same as for in-memory

membership testing, while the memory consumption is lower. The whole hedge h needs no more to be stored, but only a stack, whose size is the depth of h .

Proof. The streaming transition on $nw(h)$ with respect to Δ starting with (q, ϵ) can be computed in time $O(1)$ per letter after a precomputation in time $O(|A|)$. So the overall computation time is in $O(|A| + |h|)$. The streaming memory needed to store a configuration is of size $O(\text{depth}(h) + |A|)$ since the size of the visible stack is bounded by the depth of the input hedge. We finally test if $\llbracket h \rrbracket_{str}(I, \epsilon) \in (F, \epsilon)$. \square

The streaming transitions of a $d\text{SHA}^\downarrow A$ are equal to the streaming transitions of the corresponding $\text{dNWA } A^{nwa}$, that we recalled in Section 3.3.8.3 on related automata models.

Lemma 3.41. *For any $d\text{SHA}^\downarrow A = (\mathcal{Q}, \Sigma, \Delta, I, F)$ and factor $v \in \hat{\Sigma}^*$ of nested words:*

$$\llbracket v \rrbracket_{str}^\Delta = \llbracket v \rrbracket_{str}^{\Delta^{nwa}}$$

This shows that the streaming evaluator for $d\text{SHA}^\downarrow$ s is induced via compilation to dNWAS from that the streaming evaluator for dNWAS .

3.5 Schema-Completeness

Schemas are inputs of the membership problem that we consider, while completeness is a fundamental property of automata. To combine both aspects appropriately, we propose the notion of schema completeness, that we define uniformly for both SHAS and SHA^\downarrow s. First, we define what is a partial run on a $d\text{SHA}$ (equivalently a SHA^\downarrow) A and when a partial run is blocking:

Definition 3.42. *A partial run of A on a hedge h is a prefix r of some run of $\text{compl}(A)$ on h such that:*

- r ends with some state of \mathcal{Q} , and
- r does not contain the sink state of $\text{compl}(A)$.

A partial run r of A on h is called blocking if there does not exist any partial run r' of A on h such that r is a strict prefix of r' .

The $d\text{SHA}$ in Figure 3.10 has the unique run $R = 0 \cdot \langle 0 \cdot \text{list} \cdot 1 \cdot \langle 0 \cdot \text{item} \cdot 2 \rangle \cdot 3 \rangle \cdot 4$ on h . The partial runs of h are thus all prefixes of $nw(R)$ that end with some state, i.e., $0, 0 \cdot \langle \cdot 0, 0 \cdot \langle \cdot 0 \cdot \text{list} \cdot 1$, etc. None of these partial runs is blocking.

Definition 3.43. A schema for an automaton A is a hedge language $S \subseteq \mathcal{H}_\Sigma$ such that $\mathcal{L}(A) \subseteq S$. We call the automaton A schema-complete for schema S if no hedge $h \in S$ has some blocking partial run of A .

Schemas S are usually used to restrict the type of hedges that some problem may accept as input. The dSHA pattern matching problem for a schema S takes two inputs: a hedge $h \in S$ and an automaton A – rather than an nested regular expression e . The automaton A then selects those input hedges $h \in S$ that match the pattern, i.e., such that $h \in \mathcal{L}(A)$. For this reason, we assume that S is a schema for A , i.e., $\mathcal{L}(A) \subseteq S$. We will often assume that A is schema-complete for S , so that no partial run of A on any input hedge $h \in S$ may ever block. This can always be achieved based on the next Lemma 3.45.

Example 3.44. The dSHA in Figure 3.10 for the nested regular expression [self-list-child-item], a simplified version of the XPATH filter [self::list/child::item], has signature $\Sigma = \{\text{item}, \text{list}\}$. It is schema-complete for schema $\llbracket \text{N-List} \rrbracket$. To make this happen, we added state 5 to this automaton, which is not used in any successful run. But still, this SHA is not complete. For completing it, we would have to run 5 into a sink states by adding many transitions into it, such as $0@4 \rightarrow 5$ and $0@5 \rightarrow 5$, but also for all other states $q \in \{1, 2, 3, 4, 5\}$ the transition rules $q \xrightarrow{\text{list}} 5$, $q \xrightarrow{\text{item}} 5$, $q@4 \rightarrow 5$ and $q@5 \rightarrow 5$.

Automaton completion may raise problems to safe-no-change projection (Section 7.3). In many examples completion renders safe-no-change projection incomplete, for which it was complete otherwise. This happens for instance for the dSHA in Figure 3.10 for the nested regular expression [self-list-child-item], completed in Example 3.44: the states 2, 3, 5, 6, that otherwise can no more be changed before completion, can be changed after completion into the sink.

Without schema-completeness, however, safe-no-change projection may be unsound: it might overlook the rejection of hedges inside the schema. This is why we have to assume schema-completeness for the input automata of safe-no-change projection, and do not assume full completeness there. For congruence projection (Section 7.4), schema-completeness will be assumed implicitly in the setting taken there. Further automata completion will not affect there completeness for subhedge projection there.

We note that schema-completeness is well-defined even for non-regular schemas. For safe-no-change projection, we indeed don't have to restrict schemas to be regular, while for congruence projection only regular schemas are considered.

Furthermore, if A is schema-complete for the universal schema $\mathbf{S} = \mathcal{H}_\Sigma$ and does not have inaccessible states then A is complete.

Schema-completeness of deterministic automata for regular schemas can always be made to hold. In order to show this, we define that A is *compatible with schema* \mathbf{S} if for any hedge $h \in \mathbf{S}$ there exists a run of A in $\text{run}^\Delta(h)$. Schema-completeness implies compatibility, as shown by the next lemma. The converse is true only when assuming determinism.

Lemma 3.45. *Let A be a deterministic SHA with schema \mathbf{S} . Then A is compatible with \mathbf{S} iff A is schema-complete for \mathbf{S} .*

Proof. Suppose that A is schema-complete for \mathbf{S} . Wlog., we can assume $\mathbf{S} \neq \emptyset$. Note that if $I = \emptyset$, then ϵ would be a blocking partial run for any hedge in $\mathbf{S} \setminus \{\epsilon\}$. Since $\mathbf{S} \neq \emptyset$ it follows that there exists $q_0 \in I$. So q_0 is a partial run for any hedge $h \in \mathbf{S} \setminus \{\epsilon\}$. Since there exists no blocking partial runs by schema-completeness, this run can be extended step by step to a run on any $h \in \mathbf{S}$. So for any hedge $h \in \mathbf{S}$ there exists some run by A , showing compatibility.

For the converse, let A be compatible with \mathbf{S} and deterministic. We have to show that A is schema-complete for \mathbf{S} . Let v be a partial run of A on $h \in \mathbf{S}$. By compatibility, there exists a run $R \in \text{run}^\Delta(h)$ that starts in some initial state of A . By determinism, v must be a prefix of $\text{nw}(R)$. Thus, v is not blocking. \square

Proposition 3.46. *Any SHA A with regular schema \mathbf{S} can be made schema-complete for \mathbf{S} .*

Proof. By Lemma 3.45 it is sufficient to make A compatible with \mathbf{S} . Let B be a dSHA with $\mathbf{S} = \mathcal{L}(B)$. Automaton B is compatible with \mathbf{S} . Since B is deterministic, Lemma 3.45 implies that B is schema-complete for \mathbf{S} . Since \mathbf{S} is a schema for A with have $\mathcal{L}(A) \subseteq \mathcal{L}(B)$. We then compute the completion of $\text{compl}(A)$. The product $C = B \times \text{compl}(A)$ with final states $F^C = F^B \times F^A$ is schema-complete for schema \mathbf{S} . Furthermore, since $\mathcal{L}(A) \subseteq \mathbf{S} = \mathcal{L}(B)$ it follows that $\mathcal{L}(C) = \mathcal{L}(B) \cap \mathcal{L}(A) = \mathcal{L}(A)$. \square

Finally note that the schema \mathbf{S} can be recognized by the same product $C = B \times \text{compl}(A)$ except for replacing the set of final states F^C by the set of schema final states $F^{\mathbf{S}} = F^B \times (Q^A \cup \{\text{sink}\})$. We have $\mathbf{S} = \mathcal{L}(C[F^C/F^{\mathbf{S}}])$ where $C[F^C/F^{\mathbf{S}}]$ is the dSHA obtained from C by replacing the set of final states F^C by $F^{\mathbf{S}}$.

3.6 Schema-Based Cleaning

Schema-based cleaning was introduced recently [Niehren & Sakho 2021] in order to reduce the sizes of automata A with respect to a schema-constraint \mathbf{S} . When interested in the intersection $\mathcal{L}(A) \cap \mathbf{S}$, the idea is to remove all rules and states of A that are not used on any successful run on some hedge from \mathbf{S} . This idea can be applied NFAS , SFAS , $\text{SHA}^\downarrow_{\mathbf{S}}$, and NWS .

If the schema \mathbf{S} is defined by some automaton S , i.e. $\mathbf{S} = \mathcal{L}(S)$, then the strongest version of schema-based cleaning of A wrt. S can be obtained by removing all useless states from full product $A \times^f S$, i.e., by computing $\text{trim}(A \times^f S)$ as described in Section 3.2.11, and then projecting back to A . Alternatively, one can apply accessibility cleaning to the schema product only, i.e., project the accessible product $A \times S = \text{acc-clean}(A \times^f S)$ back to A . We denote this form of schema-based cleaning by:

$$\text{scl}_S(A)$$

It will be studied more deeply in this thesis in the case of NFAS and SHAS .

The advantage of schema-based cleaning over the clean schema-product, is that the schema-based cleaning does always reduce the size of the automaton. In contrast, the cleaning schema-product may become larger. The advantage of the cleaned schema-product is that it recognizes exactly the language of interest $\mathcal{L}(A) \cap \mathbf{S}$, while the schema-based cleaning may recognize a larger language accepting some element of $\bar{\mathbf{S}}$ too.

It should be noticed that schema-based cleaning may change the automaton's language. Therefore, even when working with a class of deterministic automata that enjoys unique minimization, the minimization of $\det(A)$ and $\text{scl}_S(\det(A))$ may be very different. In the most extreme case, where $\mathcal{L}(A) \cap \mathbf{S}$ is empty, the minimal automaton of $\text{scl}_S(\det(A))$ does not contain any state, while $\det(A)$ may be of exponential size in the size of A . This indicates that the size reduction obtained by schema-based cleaning followed by minimization may be much larger than by minimization alone.

3.7 Two-sorted Automata

We will use 2-sorted versions of (downward) stepwise hedge automata with tree states and hedge states. The two sorted versions can be compiled back to the one-sorted versions; so the expressiveness is the same. But this elimination of tree-states may lead to a quadratic size increase. Therefore, the more concise 2-sorted variant

is often preferable in practice.

3.7.1 2-Sorted SHAS

In the 2-sorted version, hedge states (h) must be converted to tree states (t), before they can be applied. So the 2-sorted apply operator has the type $@ : h \times t \rightarrow h$.

Definition 3.47. A 2-sorted SHA is a tuple $A = (\Sigma, \mathcal{Q}, \mathcal{P}, \Delta, I, F)$ where $\Delta = (\Delta', \Delta'')$ so that $A' = (\Sigma, \mathcal{Q}, \Delta', I, F)$ is a NFA. Furthermore, \mathcal{P} is a finite set of tree states and $\Delta'' = (\langle \rangle^\Delta, @^\Delta, T^\Delta)$, such that $\langle \rangle^\Delta \subseteq \mathcal{Q}$ is a subset of tree initial states, $@^\Delta \subseteq (\mathcal{Q} \times \mathcal{P}) \times \mathcal{Q}$ a set of apply rules, and $T^\Delta \subseteq \mathcal{Q} \times \mathcal{P}$ a set of tree final rules.

We draw 2-sorted SHAS as graphs extending on the graphs of NFAs as before. A tree state $p \in \mathcal{P}$ is drawn in gray \textcircled{p} . A tree final rule $(q, p) \in T^\Delta$ is drawn as $\textcircled{q} \rightarrow \textcircled{p}$. It states that if h is a hedge going to state $q \in \mathcal{Q}$ then $\langle h \rangle$ is a tree going to state $p \in \mathcal{P}$.

Transitions of 2-sorted SHAS have the form $q \xrightarrow{h} q'$ wrt Δ where $h \in \mathcal{H}_\Sigma$ and $q, q' \in \mathcal{Q}$. They are defined by the inference rules:

$$\frac{q \in \mathcal{Q}}{q \xrightarrow{\epsilon} q \text{ wrt } \Delta} \quad \frac{q \xrightarrow{a} q' \in \Delta}{q \xrightarrow{a} q' \text{ wrt } \Delta} \quad \frac{q_0 \xrightarrow{h_1} q_1 \text{ wrt } \Delta \quad q_1 \xrightarrow{h_2} q_2 \text{ wrt } \Delta}{q_0 \xrightarrow{h_1 \cdot h_2} q_2 \text{ wrt } \Delta}$$

$$\frac{q' \in \langle \rangle^\Delta \quad q' \xrightarrow{h} q \text{ wrt } \Delta \quad (q, p) \in T^\Delta \quad q_1 @ p \rightarrow q_2 \in \Delta}{q_1 \xrightarrow{\langle h \rangle} q_2 \text{ wrt } \Delta}$$

The last inference rule says that when reading a tree $\langle h \rangle$ the automaton can change from a state q_1 to a state q_2 if with h it can change from some tree initial state q' to q , so that there is some tree final rule $(q, p) \in T^\Delta$ and some apply rule $q_1 @ p \rightarrow q_2 \in \Delta$.

The language $\mathcal{L}(A)$ of a 2-sorted SHA is defined as usual for SHAS. Typed else rules can be added to 2-sorted SHAS as we did for SHAS.

Example 3.48. An example for a deterministic 2-sorted SHA with typed else rules is given in Figure 4.5. It captures the schema of hedges encoding XML documents with a single x -annotation.

One way to compile 2-sorted SHAS to SHAS is to use $\mathcal{Q} \uplus \mathcal{P}$ as state set and to replace tree final transition rules $(q, p) \in T^\Delta$ by epsilon rules $q \xrightarrow{\epsilon} p \in \Delta$. These can be eliminated as usual at the cost of a quadratic size increase. A better way is to use

the following elimination rule:

$$\frac{q_1 @ p \rightarrow q_2 \in \Delta \quad (q, p) \in T^\Delta}{q @ q \rightarrow q_2 \in \Delta^{elim-2-sorted}}$$

Determinism is preserved when eliminating 2-sortedness in this way. Furthermore, the possible size increase is at most quadratic.

3.7.2 2-Sorted SHA^\downarrow s

A 2-sorted SHA^\downarrow can be defined such as a 2-sorted SHA by replacing tree initial states to tree initial rules $\langle \rangle \subseteq \mathcal{Q} \times \mathcal{Q}$. Extensions with typed else rules are as before.

Example 3.49. *An example for a 2-sorted SHA^\downarrow with typed else rules is given in Figure 9.4. It represents the XPATH query $A0$ equal to $child::site$.*

Chapter 4

XML Documents and XPATH Queries

Abstract

We recall the XML data model, discuss an encoding of XML documents as hedges, and show how to formalize the XML data model by a stepwise hedge automaton. We introduce regular XPATH queries for XML documents informally, and discuss how these can be compiled to stepwise hedge automata too. Finally, we recall the 8 regular XPATH queries of the XPATHMARK benchmark that we will be used for testing our algorithms.

Contents

4.1	XML Documents	86
4.2	Hedge Encoding of XML Documents	87
4.2.1	General Encoding	87
4.2.2	Schema of Hedge Encodings	88
4.3	XPATH	89
4.3.1	Regular Fragment	89
4.3.2	Non-regular Queries	90
4.3.3	Types and Functions	91
4.3.4	Answer Sets	91
4.3.5	Variables	91
4.4	XPATH Benchmarks	92
4.4.1	XPATHMARK Benchmark	92
4.4.2	Lick and Schmitz' Benchmark	93
4.5	Schema Constraints for x-Annotations	94

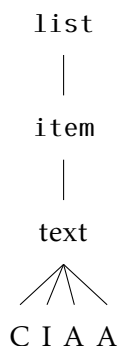
4.1 XML Documents

XML is one of the most used standardized formats for representing and exchanging structured data between various tools and applications. Processing XML documents in both in-memory and streaming modes has been widely studied for many years [Kay 2004] [Labath & Niehren 2015] [Gottlob *et al.* 2003] [Genevès & Layaïda 2006] [Gauwin 2009]. The most frequent tasks are validating, querying and transforming XML documents. In the XML technology, this is done with standardized languages based on XPATH queries, such as XSLT and XQUERY.

Any XML document can be parsed into some labeled unranked tree, or can be serialized to some nested word but with labeled parenthesis. Consider for instance the XML document:

```
<list>
  <item>
    CIAA
  <item>
</list>
```

It can be seen a nested word but with *labeled* parenthesis, the labeled opening parenthesis `<list>` and `<item>` and labeled closing parenthesis `</item>` and `</list>`. Up to removing whitespace, this XML document can be parsed to the following unranked tree:



Its root element is labeled `list`, has a single child element that is labeled by `item`, and which contains a text node with data value $C \cdot I \cdot A \cdot A$. This unranked tree

with labeled nodes can be identified with the following hedge $h \in \mathcal{H}_\Sigma$ with a single unlabeled tree constructor, and the alphabet $\Sigma = \{\text{list}, \text{item}, \text{text}\} \cup \{A, \dots, Z\}$:

$$h = \langle \text{list} \cdot \langle \text{item} \cdot \langle \text{text} \cdot C \cdot I \cdot I \cdot A \rangle \rangle \rangle$$

In turn, this hedge can be linearized to the nested word $nw(h) \in \mathcal{N}_\Sigma$ below – but now with unlabeled parenthesis:

$$nw(h) = \langle \text{list} \langle \text{item} \langle \text{text } C \ I \ A \ A \rangle \rangle \rangle$$

4.2 Hedge Encoding of XML Documents

We present a general encoding of XML documents as hedges, and give a regular schema for the set of all encodings.

4.2.1 General Encoding

Any sequence of XML documents can be encoded by some hedge. This can be done in such a way that the 5 node types of XML documents – element, attribute, text, comment, document – are represented. Furthermore, XML’s qualified names are decomposed into a name and a namespace.

Example 4.1. *In Figure 4.1, we give an example of a sequence of two XML documents. Such a sequence is not an XML document itself, but may be an intermediate result of some XSLT program, to which the XSLT program may then apply some XPATH query. The graph of hedge of this sequence of XML documents is shown in Figure 4.2. This hedge h has the following nested word $nw(h) \in \mathcal{N}_\Sigma$ – but now with unlabeled parenthesis:*

```

<elem default list
  < elem default item < text F C T >
  < elem default item < text C I A A > >
<elem default list
  < elem default item < text C M S B > > >

```

We assume that the signature contains a constant for each of the 5 XML node types: `elem` (elements), `attr` (attributes), `text` (text nodes), `comment` (comments), and `doc` (documents). It must also contain the letters of the characters of text, so usually the set of UTF-8 characters. All names and the namespace must belong to the signature too, but we don’t decompose them into letters.

```

<list>
  <item>FCT</item>
  <item>CIAA</item>
</list>
<list>
  <item>CMSB</item>
</list>

```

Figure 4.1: A sequence of two XML documents.

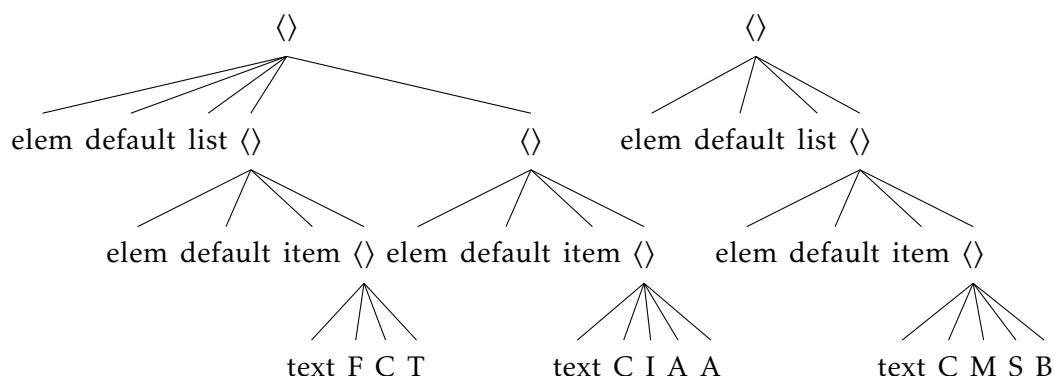


Figure 4.2: The hedge representing the sequence of XML documents in Figure 4.1.

4.2.2 Schema of Hedge Encodings

The schema of hedge encodings of XML documents is defined by the nested regular expression XML-Seq in Figure 4.3.

For this, we assume that there exists a nested regular expression *Name* that specifies the set of XML names, and a nested regular expression *Namespace* that recognizes the set of all namespaces allowed by XML documents. We assume that there is the constant *default* for the case of names without namespace¹. Furthermore, we assume a nested regular expression *Char* that defines the characters of XML data values.

The nested regular expression *Text*, for instance, states that any text subtree starts with XML node type *text* and continues with a sequence of characters. The nested regular expression *Elem* states that an XML element is a subtree that starts with the XML node type *elem*, continues with a *Namespace* or the constant *default*, followed by a *Name*, then a sequence of XML attributes, and finally a sequence of

¹The choice of the name *default* has historical reasons in our tool chain. The value *none* would have been clearer.

$$\begin{aligned}
\text{Attr} &= \langle \text{attr} \cdot (\text{Namespace} + \text{default}) \cdot \text{Name} \cdot \text{Char}^* \rangle \\
\text{Comment} &= \langle \text{comment} \cdot \text{Char}^* \rangle \\
\text{Text} &= \langle \text{text} \cdot \text{Char}^* \rangle \\
\text{Elem} &= \mu z. \langle \text{elem} \cdot (\text{Namespace} + \text{default}) \cdot \text{Name} \\
&\quad \cdot \text{Attr}^* \cdot (z + \text{Comment} + \text{Text})^* \rangle \\
\text{Doc} &= \langle \text{doc} \cdot \text{Elem} \rangle \\
\text{XML-Seq} &= \text{Elem}^* + \text{Doc}
\end{aligned}$$

Figure 4.3: Schema $\llbracket \text{XML-Seq} \rrbracket$ contains all hedges encoding XML documents.

XML elements, text, or comments.

An important issue remains to be discussed though: the sets of namespaces and names may be infinite (while the set of characters is usually finite). Therefore, the assumption of a finite signature is too optimistic. This can be solved by enriching the nested regular expressions with symbols capturing infinitely many letters. These are of the form $_ : \tau$ where $\tau \in \{\text{namespace}, \text{name}, \text{char}\}$. We can then define the missing (infinitary) nested regular expressions as follows:

$$\begin{aligned}
\text{Namespace} &= _ : \text{namespace} \\
\text{Name} &= _ : \text{name} \\
\text{Char} &= _ : \text{char}
\end{aligned}$$

These infinitary nested regular expressions can be compiled to typed else rules of infinitary SHAS.

4.3 XPATH

XPATH is a W3C standard query language for selecting nodes in XML data trees. It plays a key role across XML technologies like XSLT, and XQUERY, enabling data transformations and extractions. XPATH has evolved through three main versions – 1.0, 2.0, and 3.0 – each expanding its syntax while retaining core navigational logic, allowing it to handle a wider variety of queries. The latest version 3.1 had a key update: the addition of support for JSON by introducing maps and arrays, along with new expressions and functions to handle them.

4.3.1 Regular Fragment

In this thesis, we focus on the regular fragment of XPATH 3.1. This is basically the navigational fragment of XPATH 3.0 that we also called Core XPATH 3.0. Its queries

```
<people>
  <female id="f1">
    <firstname>Marie</firstname>
    <lastname>Dupont</lastname>
  </female>
  <male id="m1">
    <firstname>Jean-Marie</firstname>
    <lastname>Martin</lastname>
  </male>
</people>
```

Figure 4.4: An XML document representing a list of people with gender and name information.

are NRPQs, so they have *path* and *filters*, where filters may contain paths again and logical connectives. Note that the Kleene-star is not included natively in XPATH 3.1, but it can still be expressed by recursive functions.

We consider regular XPATH queries with forward axis only. These include *self*, *child*, *descendant*, and *following-sibling*. Backward axis such as *ancestor* or *preceding-sibling* are excluded from our examples, since our current compiler to nested regular expressions does not support them. Also note that *next-sibling* is not an XPATH axis, but still expressible by using positions via *following-sibling*[1]. Positions, however, do not belong to the XPATH fragment that we consider.

Regular XPATH queries permit to compare data values to constants. The following example, for instance, was already given in Section 1.1.1 of the introduction::

```
//male/firstname[contains(., 'Marie')]
```

When applied to the XML document in Figure 4.4, this query selects the second *firstname* node with identifier *m1* of the document, since its parent node is *male*, but not the first, since it has the parent node *female*.

4.3.2 Non-regular Queries

On the other hand, non-regular XPATH query with comparisons between data values are ruled out, in the fragment of interest. An example is the query:

```
//male/firstname[contains(., = //female/firstname)]
```

which selects the node the *firstname* with parent node *male* and text Jean-Marie, since there exists a node *female* whose *firstname* child is Marie.

4.3.3 Types and Functions

We notice that XPATH is a typed language, and that different types of output are possible. It supports subtyping, so that the same object can be given various types of different generality. This permits, for instance, to consider dates as special texts. We restricted ourselves to queries that output sets of nodes. Other queries that output integers, dates, texts, or sequences are not considered.

XPATH 3.1 also support recursive higher order functions. These permit to express the Kleene-star in particular. Having the Kleene star-natively would be of high interest as argued for instance by [Mozafari *et al.* 2012]. The only functions that we permit in our study are the functions on texts: `contains`, `starts-with`, and `ends-with`.

However, since we rule out all other XPATH functions, the only recursive queries permitted are those that use the recursive axis like `descendant` and `following-sibling`.

4.3.4 Answer Sets

XPATH has two different official semantics given by the W3C, depending on whether it is considered as a fragment of XSLT or XQUERY. In the former, the answer set must be output in document order, while in the latter, the answer set can be output in any order. Since we are interested in XML stream processing, it is crucial to adopt the unordered semantics, which is the approach taken in the present thesis.

The nodes of an XML document may be identified with integers when traversing them in document order. In the official semantics, the answer set does *not* refer to the nodes of the XML document, but to the subtrees rooted at these nodes. However, using subtrees instead of nodes may increase the size of the answer set from linear to quadratic and potentially lead to huge delays in streaming mode. For example, if the root is selected, the full input document must be output, which can only be done at the very end of the document. Therefore, we do not follow the choice of outputting subtrees and instead consider the answer sets as sets of nodes (integers).

4.3.5 Variables

Finally notice that XPATH queries may also contain variables that are bound to further XML documents or values of other types. In this way, it may select nodes in several XML documents. In the example we consider, we do not permit any variables

though.

4.4 XPATH Benchmarks

Two benchmarks for XPATH queries were made available, the XPATHMARK benchmark and Lick and Schmitz' benchmark.

4.4.1 XPATHMARK Benchmark

XPATHMARK [Franceschet 2005a] is a widely used benchmark for XPATH 1.0, consisting of two main tests: a functional test and a performance test. Below is a brief description of each:

- **XPATH Functional Test (XPath-FT):** This test focuses on evaluating various functional aspects of XPATH 1.0, such as navigational axes, filters, node tests, operators, and functions. It consists of multiple groups of queries, each targeting specific features of the language, which are executed on a small educational document.
- **XPATH Performance Test (XPath-PT):** This test assesses the performance of an XML query processor. It includes a set of queries grouped by their computational complexity, designed to test both data and query scalability. A key advantage is that these queries are compatible with the XMark benchmark [Schmidt *et al.* 2002], which provides a generator for scalable XML documents, of size up to tens of gigabytes.

In this thesis, we primarily focus on XPath-PT [Franceschet 2005b] due to its scalability benefits and its direct relevance to computational complexity. This test organizes queries into six categories (A–F):

- **A:** queries with forward axes.
- **B:** queries with backward axes.
- **C:** queries with the focus on data comparisons and joins.
- **D:** queries with aggregation functions, primarily counting and summing.
- **E:** queries using positions with different axes, and also string searches.

Query ID	XPATH query
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

Table 4.1: Regular XPATH Queries of XPATHMARK Benchmark.

- **F:** queries with the focus closure types (single, 2-step, long step, transitive).

In this thesis, the starting point of our tests with Category A, which only uses the child and descendant axes. While we could have also considered Category B, as mentioned earlier, we are constrained by certain limitations of existing compilers for nested regular expressions. Some queries from Category C, particularly those involving data comparisons with atomic values, could have been used as well. However, we designed our own set of queries that extend upon Category A (see Table 9.2). In Table 4.1, we present the eight queries from Category A.

The dSHAS for these queries obtained by Niehren and Sakho [Niehren & Sakho 2021] will be shown in Section 4.6.

4.4.2 Lick and Schmitz' Benchmark

Lick and Schmitz [Lick 2019] presented a benchmark of XPATH queries gathered from real-world online XSLT and XQUERY programs [Lick & Schmitz 2022]. The benchmark contains 21,141 XPATH queries from all XPATH 1.0, 2.0, and 3.0.

This benchmark was originally developed for testing verification techniques for XPATH queries, such as satisfiability and containment testing. These problems are undecidable for non-regular XPATH queries in general, so the question was, which percentage of XPATH queries in practice belong to subclasses for which these problems are decidable.

Performance testing or XPATH evaluators was not in the scope of interest. So neither documents nor schemas were provided with the queries. What was given,

however, is a tool to select XPATH queries with particular properties from the benchmark.

4.5 Schema Constraints for x-Annotations

We can encode each node selecting XPATH query as a monadic query:

$$Q : S \rightarrow 2^N$$

The schema S of this query is defined by the nested regular expression XML-Seq:

$$S = \llbracket \text{XML-Seq} \rrbracket$$

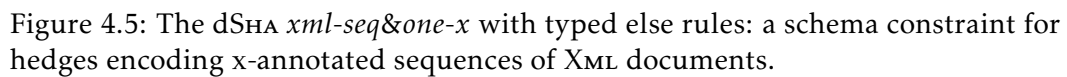
Any monadic query Q for some regular XPATH expression is regular, so Q can be represented by regular languages of x -annotated hedges. We next introduce schema constraints C for x -annotated hedges encoding XML documents. We assume that the selection variable x is annotated after names of `elem` or `attr` nodes, or after the type constants `text`, `comment`, or `doc`. So we impose the following schema XML-Seq- x to constrain the x -annotations of the hedge encodings of XML documents:

$$\begin{aligned} \text{Attr-}x &= \langle \text{attr} \cdot (\text{Namespace} + \text{default}) \cdot \text{Name} \cdot (x + \neg x) \cdot x\text{-Char}^* \rangle \\ \text{Comment-}x &= \langle \text{comment} \cdot (x + \neg x) \cdot \text{Char}^* \rangle \\ \text{Text-}x &= \langle \text{text} \cdot (x + \neg x) \cdot \text{Char}^* \rangle \\ \text{Elem-}x &= \mu z. \langle \text{elem} \cdot (\text{Namespace} + \text{default}) \cdot \text{Name} \cdot (x + \neg x) \cdot \\ &\quad \text{Attr}^* \cdot (z + \text{Comment-}x + \text{Text-}x)^* \rangle \\ \text{Doc-}x &= \langle \text{doc} \cdot (x + \neg x) \cdot \text{Elem-}x \rangle \\ \text{XML-Seq-}x &= \text{Elem-}x^* + \text{Doc-}x \end{aligned}$$

In positions where we may put the selection variable x , we may also put a special word $\neg x$. Most of the time we consider $\neg x$ as a special letter, in particular when determinism matters, but in other place, we identify it with the empty word ϵ .

In addition we may impose the schema constraint on x -annotation $\text{One-}x$ that tests whether x occurs exactly once. So the overall schema constraint for x -annotation that we are going to impose is:

$$C = \llbracket \text{XML-Seq-}x \rrbracket \cap \llbracket \text{One-}x \rrbracket$$



4.6 Available Deterministic SHAs

We present the dSHAS for A1-A8 in the Figures 4.6, 4.7, 4.8, 4.9, 4.10, 4.11, 4.12, 4.13. Each XPATH expression was first compiled to some nested regular expression, which was then compiled to a SHA, that was determinized, cleaned with respect to the schema-constraint C , and then minimized. Each of these dSHA B defines a monadic query

for the corresponding XPATH expression, where schema \mathbf{S} and the schema constraints \mathbf{C} are given above.

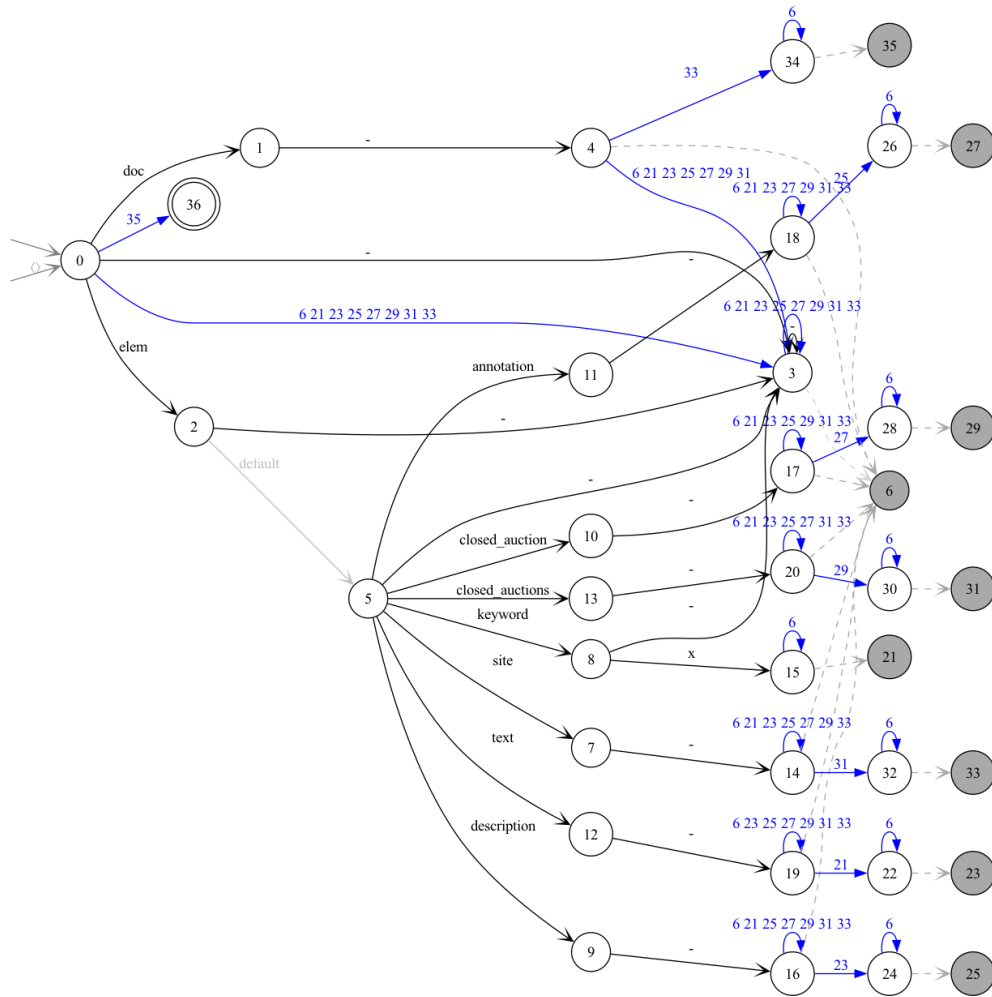


Figure 4.6: The minimization of the schema-cleaned accessible determinization of the SHA compiled from the XPATH query of the XPATHMARK benchmark: $A1 = /site/closed_auctions/closed_auction/annotation/description/text/keyword$.

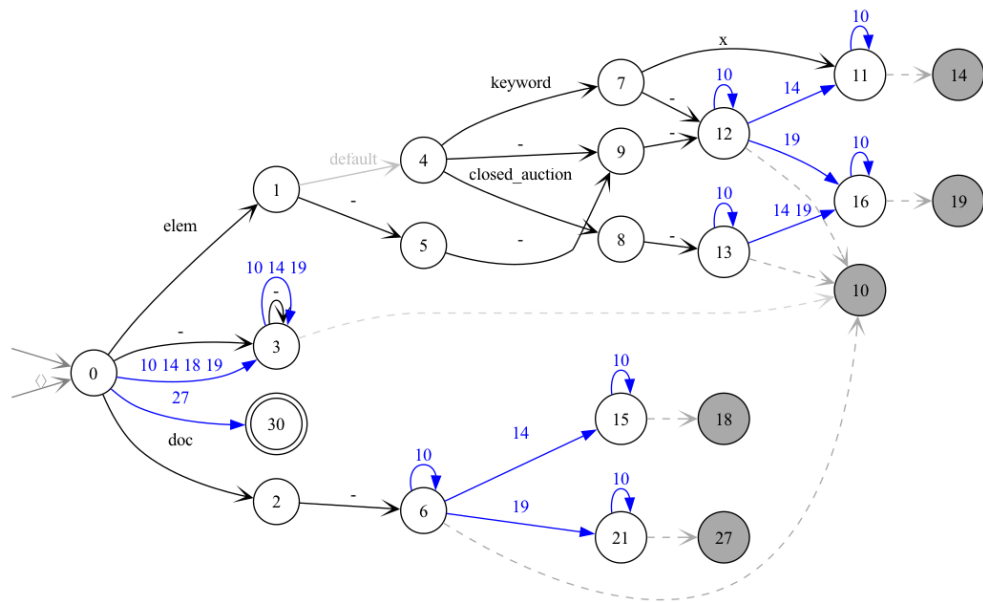


Figure 4.7: The minimization of the schema-cleaned accessible determinization of the SHA compiled from the XPATH query of the XPATHMARK benchmark: $A2 = //closed_auction//keyword$.

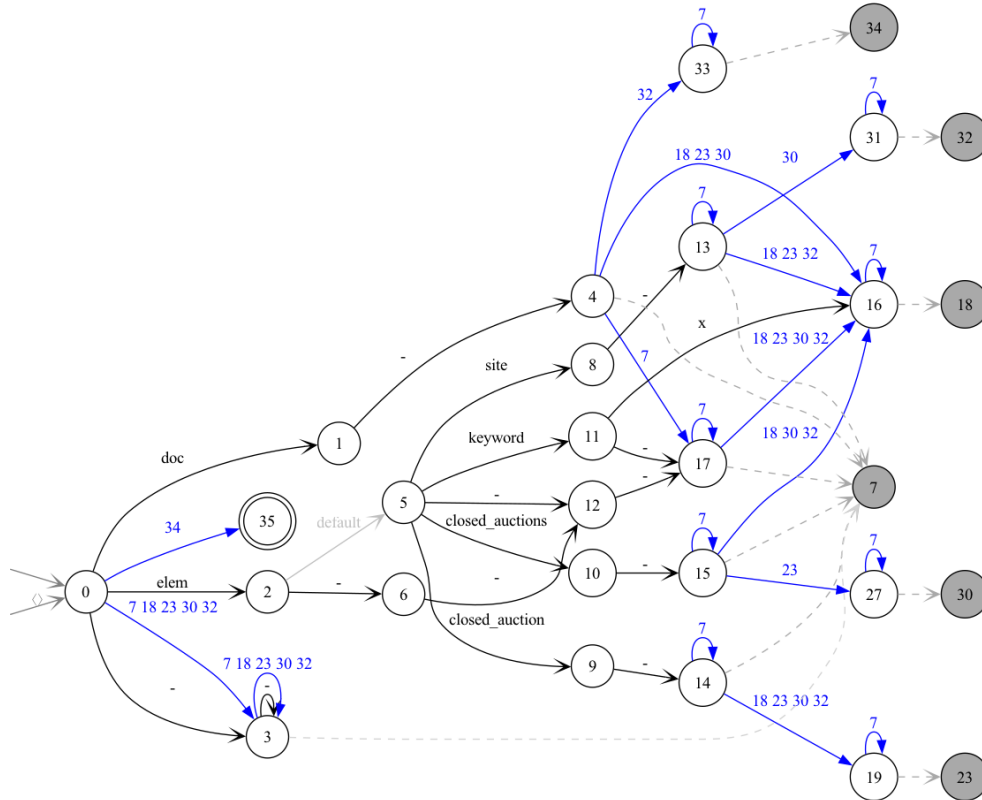


Figure 4.8: The minimization of the schema-cleaned accessible determinization of the SHA compiled from the XPATH query of the XPATHMARK benchmark: $A3 = /site/closed_auctions/closed_auction//keyword$.

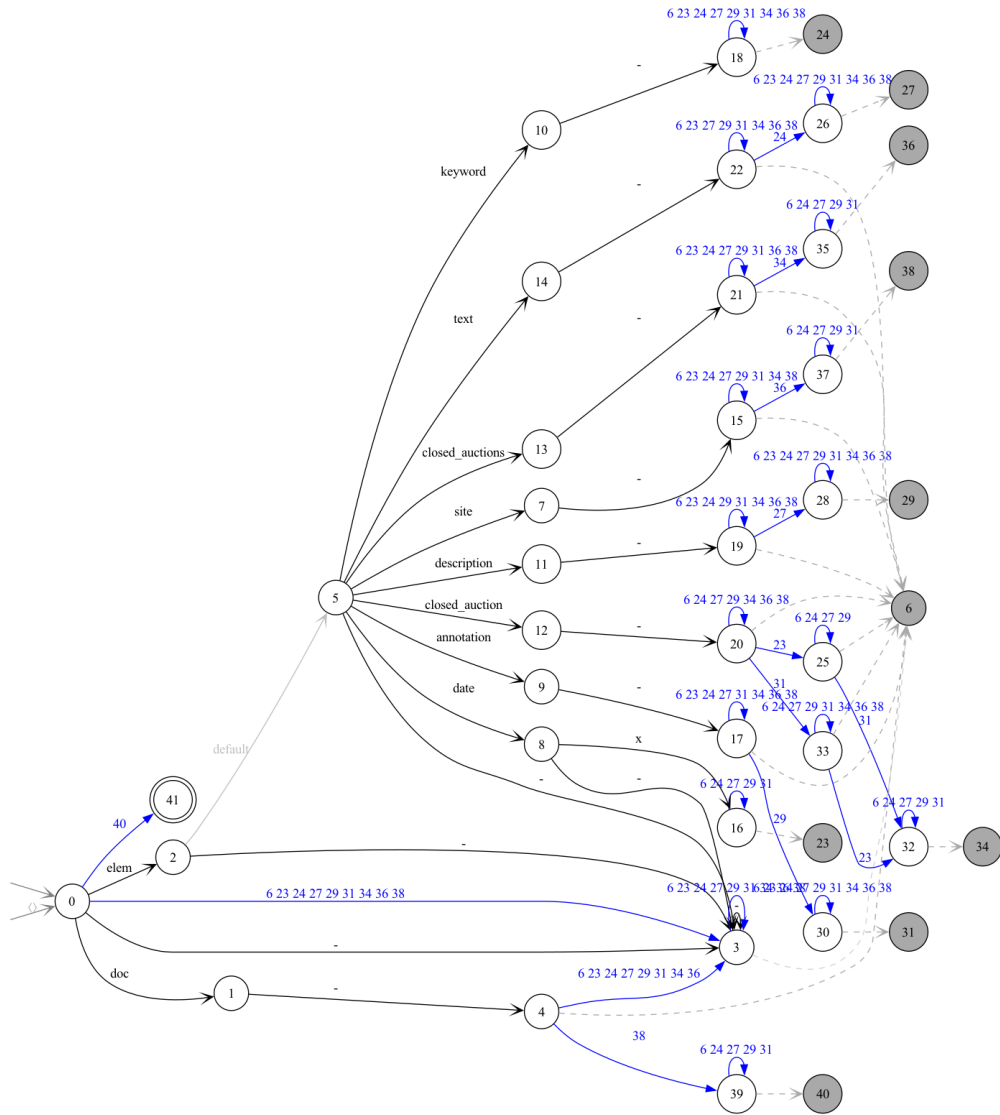


Figure 4.9: The minimization of the schema-cleaned accessible determinization of the SHA compiled from the XPATH query of the XPATHMARK benchmark: $A4 = /site/closed_auctions/closed_auction[annotation/description/ text/keyword]/date$.

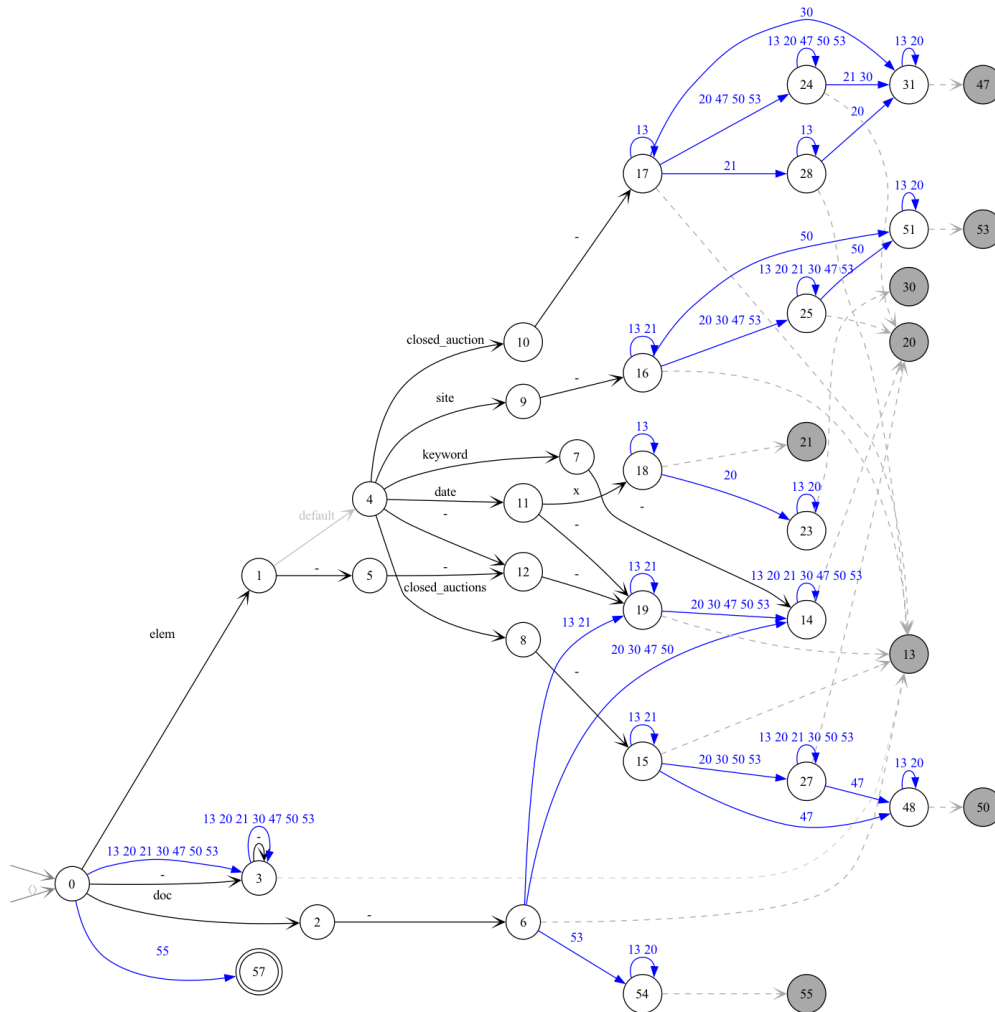


Figure 4.10: The minimization of the schema-cleaned accessible determinization of the SNA compiled from the XPATH query of the XPATHMARK benchmark: A5 = / site / closed_auctions / closed_auction [descendant::keyword] / date.

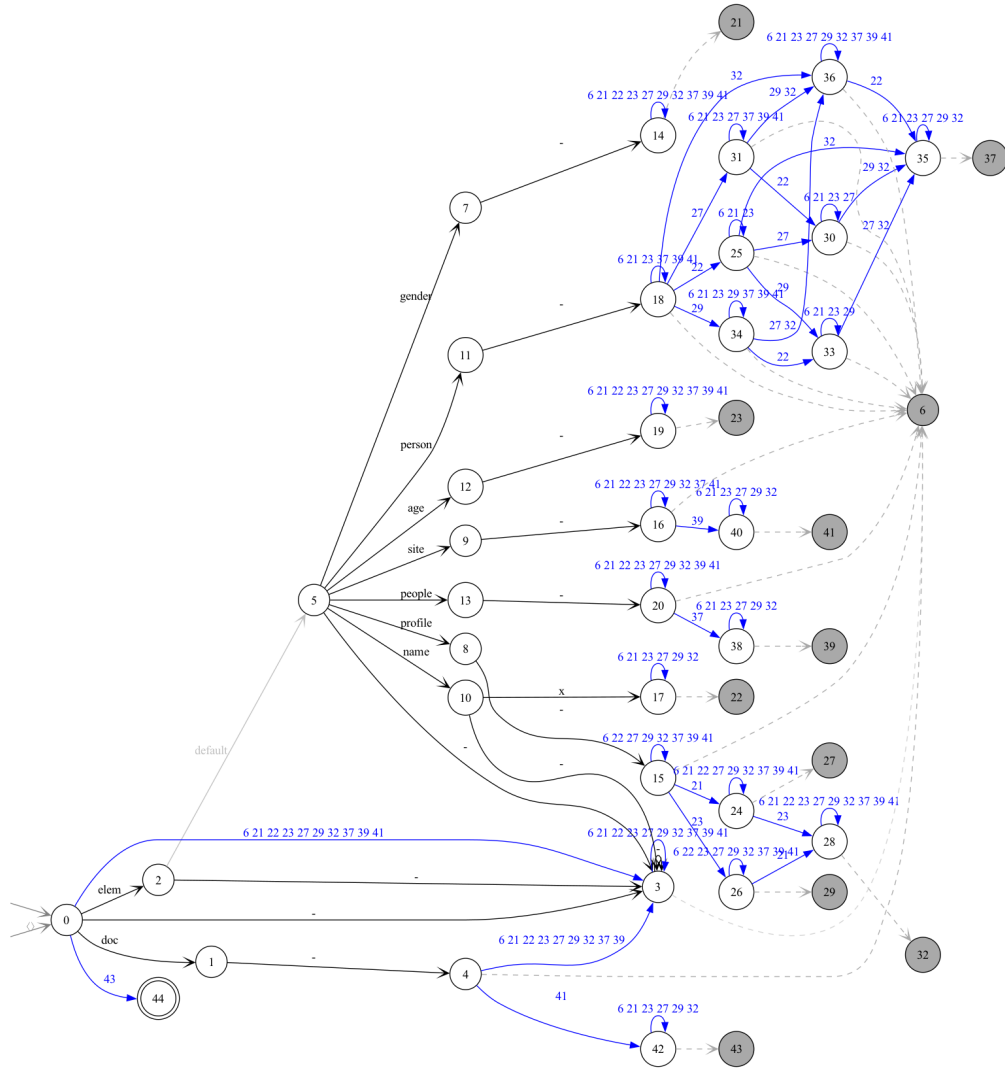


Figure 4.11: The minimization of the schema-cleaned accessible determinization of the SHA compiled from the XPATH query of the XPATHMARK benchmark: $A6 = /site/people/person[profile/gender \text{ and } profile/age]/name$.

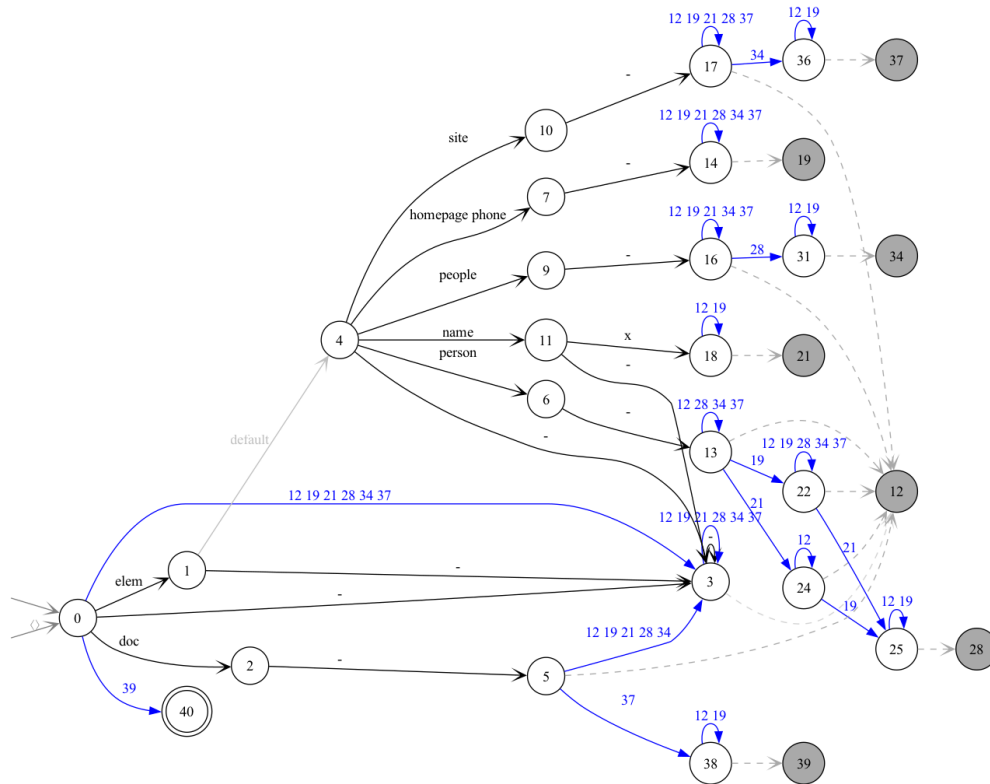


Figure 4.12: The minimization of the schema-cleaned accessible determinization of the SNA compiled from the XPATH query of the XPATHMARK benchmark: $A7 = /site/people/person[phone\ or\ homepage]/name$.

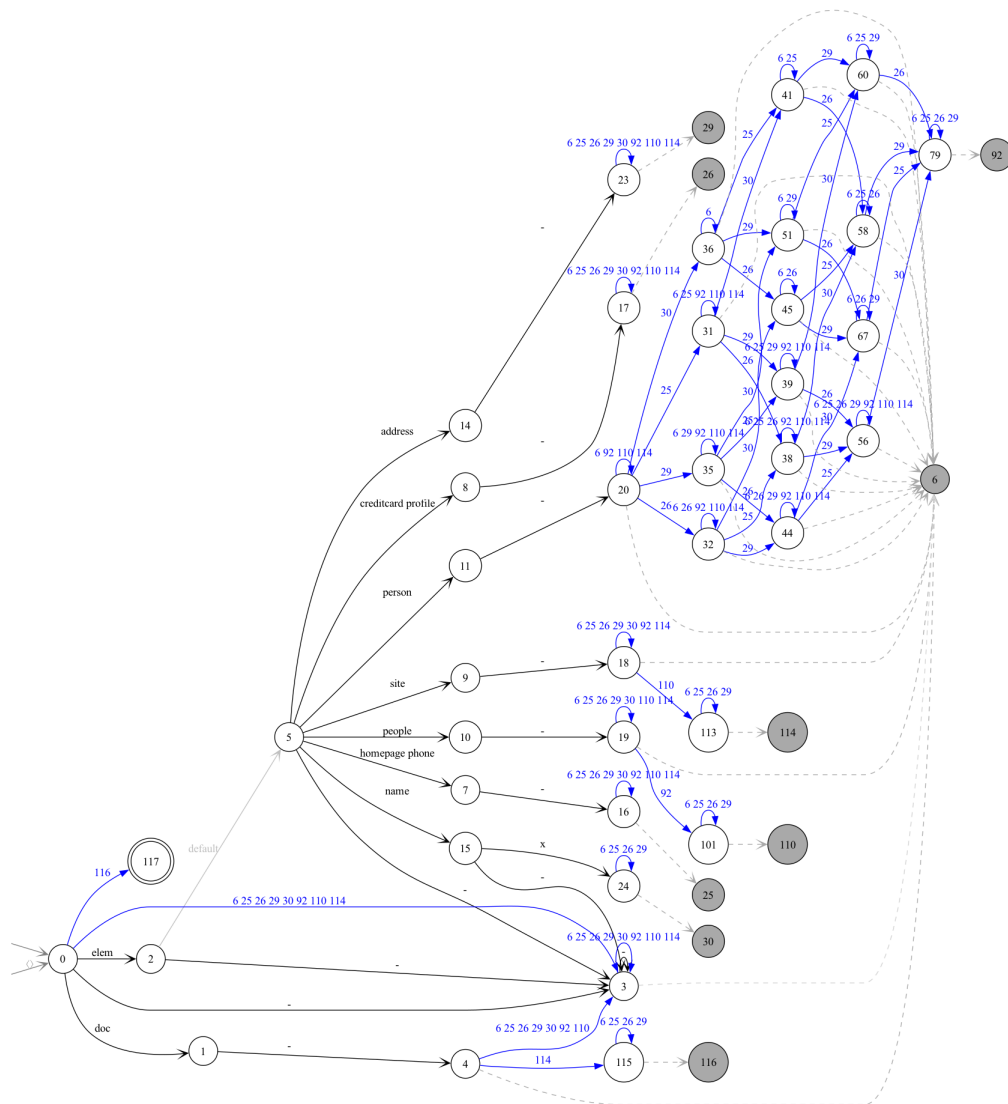


Figure 4.13: The minimization of the schema-cleaned accessible determinization of the `SHA` compiled from the `XPATH` query of the `XPATHMARK` benchmark: `A8 = /site/people/person[address and (phone or homepage) and (creditcard or profile)]/name`.

Part II

Determinization

Abstract

We develop a schema-based determinization algorithm for stepwise hedge automata, in order to obtain small deterministic automata for regular `XPATH` queries in practice. Schema-based determinization may yield exponentially smaller automata than schema-less accessible determinization. We apply schema-based determinization to the benchmark of `XPATH` queries harvested by Lick and Schmitz in practice (from online `XSLT` and `XQUERY` programs transforming `XML` documents with schema (docbook, htmlbook, teixsl, treedown, and histei). We selected the 78 most complex forward regular `XPATH` queries from the more than 4500 regular forward `XPATH` queries in this benchmark, which overall contains more than 21000 queries. We obtain small deterministic stepwise hedge automata for 78 regular `XPATH` queries. The biggest has 58 states and an overall size of 358. We also converted these `dSHAS` to deterministic nested word automata of similar sizes. All queries and automata are made freely available in the Software Heritage archive. They can serve as a benchmark for testing the efficiency of automata-based query evaluators.

Schema-based Determinization

Abstract

We consider automata whose languages are constrained by some regular schema. We propose an algorithm for schema-based determinization of finite state automata. We develop it for stepwise hedge automata (SHAs) but our algorithm is new even for the special case of finite state automata on words. We show that schema-based determinization is always more efficient than the usual accessible determinization followed by schema-based cleaning while yielding the same result. We also present a family of stepwise hedge automata obtained from XPATH queries, for which the experiments with schema-based determinization suggest a more than linear size reduction and thus speed-up compared to accessible determinization.

Contents

5.1	Introduction	108
5.2	Accessible Determinization	112
5.3	Schema-Based Cleaning for NFAs	114
5.4	Schema-Based Determinization for NFAs	117
5.5	Schema-Based Cleaning and Determinization for SHAs	124
5.6	Correctness Proof	127
5.7	Scaling Experiments	131

5.1 Introduction

Automata determinization is a crucial operation for many algorithmic problem on automata like testing universality and inclusion. More generally it is relevant for any automata problem that is in polynomial time when restricted to deterministic automata and in exponential time otherwise. As shown by [Gauwin *et al.* 2009a, Gauwin *et al.* 2009b], this is the case for *EQA* for *NWAS* and *SHAS* (since these are *ExpTIME*-complete), and also unfeasible for *EQA* for *NFAS* (which is *PSPACE*-complete) when assuming $P \neq NP$.

The usual determinization algorithm for *NWAS* – that lifts the subset construction to subsets of pairs of states [Okhotin & Salomaa 2014, Alur & Madhusudan 2004, von Braunmühl & Verbeek 1985] – is problematic for any *NWA* that uses top-down information, i.e., that does not have the weak single entry property. The *NWAS* obtained from regular *XPATH* queries via nested regular expressions and the compiler to *NWAS* from [Niehren & Sakho 2021] are problematic in particular.

In order to solve this problem, Niehren and Sakho proposed an alternative determinization algorithm for *NWAS* that is based on *SHAS* determinization. The *SHAS* obtained from the *NWAS* for the regular *XPATH* queries in the *XPATHMARK* benchmark [Franceschet 2005b] could all be determinized in few seconds. The largest deterministic *SHA* was obtained for *XPATH* query A8. It has 124 states and an overall size of 2464. The size can be reduced further by cleaning the deterministic *SHA* with respect to the schema such as $\llbracket \text{One-}x \rrbracket$ constraining the languages of x -annotated hedges with signature Σ defining *XPATH* queries. When minimizing after schema-based cleaning, the number of states of the *dSHAS* for A8 could be reduced to 48 and the overall size of the automaton to 294. The *dSHAS* obtained could then be compiled to deterministic *NWAS* in linear time.

Unfortunately, the situation becomes worse again, when constructing *dSHAS* for regular *XPATH* queries from practical *XSLT* and *XQUERY* programs, as provided by Lick and Schmitz' benchmark [Lick & Schmitz 2022]. The phenomenon can also be observed at the following example *XPATH* query(outside this benchmark):

```
(QN7)      /a/b//(* | @* | comment() | text())
```

Query QN7 selects all nodes of an XML document that are descendants of a b -element below an a -element at the root. The nodes may have any XML type:

element, attribute, comment, or text. The nondeterministic SHA for QN7 has 145 states and an overall size of 348. Its determinization however leads to an automaton with 10,005 states and an overall size of 1,634,122.

The kick-off question of the present chapter is how to further reduce the size of the dSHA for regular XPath queries, and the time to compute them, in order to obtain sufficiently small dSHAs for practical XPath queries – such as in the benchmark of Lick and Schmitz – efficiently. The idea of the present chapter is to use schema-based cleaning and determinization for this purpose.

Since we are interested in automata defining monadic queries, we consider automata A whose languages are constrained by some regular schema \mathbf{S} . This means that we are less interested in the language of the automaton $\mathcal{L}(A)$ than in its intersection with the schema constraints $\mathcal{L}(A) \cap \mathbf{S}$. A typical example for a schema constraint when defining monadic queries, is the constraint by the schema $\mathbf{S} = \llbracket \text{One-}x \rrbracket$, which accepts all hedges in \mathcal{H}_{Σ^x} that contain a single occurrence x . Schema \mathbf{S} can be defined by the dSHA in Figure 5.9. When x is the selection variable of a monadic query, then it models that any query answer selects a single node for x .

Further schema constraints may define the domain of monadic queries, i.e., the schema of the query. If the query must be applied to some XML documents, then the schema constraint has to specify the XML data model. If the query is applied to hedges encoding XML documents, then the schema constraint has to capture such hedges. See the schema $\llbracket \text{XML-Seq-}x \rrbracket$ from Section 4.2.

The schema-based cleaning from Section 3.6 of the dSHA for QN7 results in a dSHA with only 74 states and 203 transition rules. Applying SHA minimization afterward reduces the automaton further to 27 states and 71 transition rules. What we are really interested in, however, is to compute the schema-based cleaning of $\det(\text{QN7})$ and minimize it. However, our implementation of schema-based cleaning, that computes the accessible product of two SHAs based on OCaml's cli Datalog, quickly runs out of memory for larger automata, so it remains impossible to clean $\det(\text{QN7})$ based on the schema, given that $\det(\text{QN7})$ has 10,005 states and an overall size of 1,634,122. A similar efficiency problem applies to our implementation of dSHA minimization, that is equally based OCaml's cli Datalog for computing difference relations on states, so it is impossible to minimize $\det(\text{QN7})$ in this way.

Therefore, the question remains open how to compute $\text{scl}_5(\det(A))$ efficiently where A is a SHA for some regular XPath query, in order to efficiently produce reasonably small deterministic automata for regular XPath queries as simple as

QN7, and for the regular XPATH queries in the benchmark of Lick and Schmitz.

Given the relevance of schemas for schema-based cleaning, one naive approach could be to determinize the product of the automata for the query and schema. This may look questionable at first sight, given that the schema-product may be bigger than the original automaton, so why could it make determinization more efficient? But in the case of QN7, the determinization of the schema-product yields a deterministic automata with only 92 states and 325 transition rules, and can be computed efficiently. This observation is very promising, motivating three general questions:

Question 4. Why are schemas so important for automata determinization?

Question 5. Can this be established by some complexity result?

Question 6. Is there a way to compute the schema-based cleaning of the determinization of an SHA more efficiently than by accessible determinization followed by schema-based cleaning?

Our main result is a novel algorithm for schema-based determinization SHAS, that integrates schema-based cleaning directly into the usual determinization algorithm. For any SHA A , it computes the schema-based determinization:

$$det_S(A) = scl_S(det(A))$$

i.e., the same result as accessible determinization det applied to A followed by schema-based cleaning scl_S . This is proven by our soundness Theorem 2. Furthermore, we show in Proposition 5.15 that $det_S(A)$ can be computed more efficiently than first computing $B = det(A)$ and then $scl_S(B)$. Our results thus answer Question 6 positively.

The idea of our schema-based determinization algorithm is to refine the usual accessible determinization algorithm based on the subset construction, so that it keeps only those subsets of states that can be aligned to some state of the schema. This algorithm obtained in this way can compute the schema-based cleaning of the usual accessible determinization of QN7 in less than three seconds. In contrast, the accessible determinization is so big such that our implementation of schema-based cleaning of does not terminate on it after a few hours. We also show that the worst case time complexity of schema-based determinization is lower than that of accessible determinization followed by schema-based cleaning.

The answer to Question 4 is that determinizing $A \times S$ may be exponentially less costly than determinizing A , but at most polynomially more costly. This is shown by the answer of Question 5: It starts from the observation that the number of states of $S \times \det(A)$ may be exponentially smaller than the number of states of the accessible determinization $\det(A)$, since the schema automaton S is assumed to be deterministic, and at most polynomially bigger.

In order to see this, we first note that $\det(A \times S) = \det(A) \times S$ up to renaming of states¹ for the deterministic schema automaton S . So for exponentially many subsets of states $Q \in \mathcal{Q}^{\det(A)}$, there may be no state $s \in \mathcal{Q}^S$ such that $(Q, s) \in \mathcal{Q}^{\det(A) \times S}$, since this would require that all states $q \in Q$ can be aligned to s , i.e., that (q, s) in $\mathcal{Q}^{A \times S}$ for all $q \in Q$.

Concerning Question 6, we provide a complexity upper bound for the time to compute $\det_S(A) = \text{scl}_S(\det(A))$ in Proposition 5.15. This bound depends quadratically on the number of states of $S \times \det(A)$, which may be exponentially smaller than for $\det(A)$ since S is deterministic. Furthermore, it is not difficult to see that $\det(A) \times S$ is equal to $\det_S(A) \times S$, and thus:

$$\det(A \times S) = \det_S(A) \times S$$

Hence, any size bound for the schema-based determinization $\det_S(A)$ implies a size bound for the accessible determinization of the schema-product $\det(A \times S)$.

We have implemented schema-based determinization for SHAS with an experimental evaluation. For this, we consider a family of SHAS obtained from a scalable family of XPATH queries. Our experiments confirm a very large reduction of the computation time by the usage of schemas during determinization. These experiments suggest that the speed up is more than linear for this family of queries.

A large scale experiment on practical XPATH queries is provided in Chapter 6 where schema-based algorithms were applied to the regular XPATH queries collected by Lick and Schmitz [Lick & Schmitz 2022] from real-world XQUERY and XSLT programs. Small deterministic SHAS could be obtained by schema-based determinization for all regular XPATH queries in this corpus. By contrast, accessible determinization fails in 37% of the cases with a timeout of 100 seconds. Without this timeout, determinization either runs out of memory or produces very large automata.

¹If $\{(q_1, s_1) \dots (q_n, s_n)\} \in \mathcal{Q}^{\det(A \times S)}$, then there exists a tree that can go into all states $q_1 \dots q_n$ with A and into all states s_1, \dots, s_n with S . Since S is deterministic, we have $s_1 = \dots = s_n$. So there exists a tree going into $\{q_1, \dots, q_n\}$ with $\det(A)$ and also into all s_i . So $(\{q_1, \dots, q_n\}, s_i)$ is a state of $\det(A) \times S$.

$$\begin{array}{c}
\frac{I^A \neq \emptyset}{I^A \in I^{det(A)} \quad I^A \in Q^{det(A)}} \quad \frac{Q \in Q^{det(A)} \quad Q \cap F^A \neq \emptyset}{Q \in F^{det(A)}} \\
\\
\frac{Q \in Q^{det(A)} \quad Q' = \{q' \in Q^A \mid q \xrightarrow{a} q' \in \Delta^A, q \in Q\} \neq \emptyset}{Q \xrightarrow{a} Q' \in \Delta^{det(A)} \quad Q' \in Q^{det(A)}} \\
\\
det(A) = (\Sigma, Q^{det(A)}, \Delta^{det(A)}, I^{det(A)}, F^{det(A)})
\end{array}$$

Figure 5.1: The accessible determinization $det(A)$ of NFA A .

```

1 fun det(A) =
2   let Store = hashset.new(0)
3   let Agenda = list.new() and Rules = hashset.new(0)
4   if I^A ≠ ∅ then Agenda.add(I^A)
5   while Agenda.notEmpty() do
6     let Q = Agenda.pop()
7     let h be an empty hash table with keys from Σ.
8     // the values will be \empty hash subsets of Q^A
9     for q  $\xrightarrow{a}$  q' ∈ Δ^A such that q ∈ Q do
10      if h.get(a) = undef then h.add(a, hashset.new(0))
11      (h.get(a)).add(q')
12     for (a, Q') in h.toList() do Rules.add(Q  $\xrightarrow{a}$  Q')
13     if not Store.member(Q') then Store.add(Q') Agenda.push(Q')
14   let F^{det(A)} = {Q | Q ∈ Store, Q ∩ F^A ≠ ∅}
15   return (Σ, Store.toSet(), Rules.toSet(), I^A, F^{det(A)})

```

Figure 5.2: A program computing the accessible determinization of an NFA A from Figure 5.1.

Outline We start in Section 5.2 by recalling the accessible determinization of NFAs. In Section 5.3, we recall schema-based cleaning for NFAs. In Section 5.4, we contribute our schema-based determinization algorithm in the case of NFAs and show its correctness. In Section 5.5, we lift schema-based determinization to SHAs and proof it soundness in Section 5.6. In Section 5.7, we illustrate a non-linear speed-up of schema-based determinization or by determinizing the schema product experimentally for the SHAs of a family of regular XPATH queries.

5.2 Accessible Determinization

We work out few complexity properties of the accessible determinization of NFAs that are basically folklore.

For this we note that the accessible determinization $det(A)$ of a NFA $A =$

$(\Sigma, \mathcal{Q}^A, \Delta^A, I^A, F^A)$ can be computed by the inference rules in Figure 5.1. The computation works like determinization with the usual subset construction, except that only accessible subsets are created. It is well known that $\mathcal{L}(A) = \mathcal{L}(\det(A))$. Since only accessible subsets of states are added, we have $\mathcal{Q}^{\det(A)} \subseteq 2^{\mathcal{Q}^A}$. Therefore, the accessible determinization may even reduce the size of the automaton and often avoid the exponential worst case where $\mathcal{Q}^{\det(A)} = 2^{\mathcal{Q}^A}$.

Proposition 5.1 (Folklore). *The accessible determinization $\det(A)$ of a NFA A can be computed in expected amortized time $O(|\mathcal{Q}^{\det(A)}| |\Delta^A| + |A|)$.*

Proof. The algorithm for accessible determinization and its complexity upper bound are somehow folklore. We present them nevertheless, since they are to be refined for schema-based determinization later on. A set of inference rules for accessible determinization is given in Figure 5.1, and an algorithm computing the fixed point of these inference rules is presented in Figure 5.2. It uses dynamic perfect hashing [Dietzfelbinger *et al.* 1994] for implementing hash sets, so that set inserting and membership can be done in randomized amortized time $O(1)$. The algorithm has a hash set *Store* to save all discovered states $\mathcal{Q}^{\det(A)}$ and a hash set *Rules* to collect all transition rules. Furthermore, it has a stack *Agenda* to process all new states $Q \in \mathcal{Q}^{\det(A)}$. For each Q popped from the stack *Agenda*, the algorithm uses a hash table h to compute all pairs (a, Q') such that $Q \xrightarrow{a} Q' \in \Delta^{\det(A)}$ and $Q' \neq \emptyset$. This is done by iterating over Δ^A so in time $O(|\Delta^A|)$. By iterating over the hash table h , all transitions $Q \xrightarrow{a} Q'$ will be added to the set *Rules* and Q' will be added to the stack *Agenda* and to the hash set *Store* if it wasn't there yet. The overall number of elements in the *Agenda* is $|\mathcal{Q}^{\det(A)}|$. For each Q , the computation of all Q' is in time $O(|\Delta^A|)$. The preprocessing of A requires time $O(|A|)$. Thus, the total time of the algorithm is in $O(|\mathcal{Q}^{\det(A)}| |\Delta^A| + |A|)$. \square

The graph of the DFA $\det(A_0)$ obtained by accessible determinization is shown in Figure 3.6. For A_0 , it is equal to the trimmed automaton $\text{trim}(A_0^{\det-f}) = \text{acc-clean}(A_0^{\det-f})$. Note that only 4 out of the $2^3 = 8$ subsets are accessible, so the size increases only by a single state and two transitions rules in this example.

Proposition 5.2. *The accessible determinization of a SHA can be computed in expected amortized time $O(|\mathcal{Q}^{\det(A)}|^2 |\Delta^A| + |A|)$.*

Proof. An algorithm for computing the fixed points of the inference rules of accessible determinization of a SHA is presented in Figure 5.3. It extends on the case of NFAs with the same data structures. It uses dynamic perfect hashing for the hash

```

1 fun detSHA(A) =
2   let Store = hashset.new(0)
3   let Agenda = list.new() and Rules = hashset.new(0)
4   if  $I^A \neq \emptyset$  then Agenda.add( $I^A$ )
5   while Agenda.notEmpty() do
6     let Q = Agenda.pop()
7     let h be an empty hash table with keys from  $\Sigma$ .
8     // the values will be \empty hash subsets of  $\mathcal{Q}^A$ 
9     for  $q \xrightarrow{a} q' \in \Delta^A$  such that  $q \in Q$  do
10      if h.get(a) = undef then h.add(a, hashset.new(0))
11      (h.get(a)).add( $q'$ )
12     for (a,  $Q'$ ) in h.toList() do Rules.add( $Q \xrightarrow{a} Q'$ )
13     if not Store.member( $Q'$ ) then Store.add( $Q'$ ) Agenda.push( $Q'$ )
14     for  $Q_1 \in \text{Store}$  do
15       let  $Q' = \{q' \mid q @ q_1 \rightarrow q', q_1 \in Q_1, q \in Q\}$ 
16       if  $Q' \neq \emptyset$  then Rules.add( $Q @ Q_1 \rightarrow Q'$ )
17       if not Store.member( $Q'$ ) then Store.add( $Q'$ ) Agenda.push( $Q'$ )
18       let  $Q'' = \{q'' \mid q_1 @ q \rightarrow q'', q_1 \in Q_1, q \in Q\}$ 
19       if  $Q'' \neq \emptyset$  then Rules.add( $Q_1 @ Q \rightarrow Q''$ )
20   let  $F^{det(A)} = \{Q \mid Q \in \text{Store}, Q \cap F^A \neq \emptyset\}$ 
21   return ( $\Sigma, \text{Store.toSet}(), \text{Rules.toSet}(), I^A, F^{det(A)}$ )

```

Figure 5.3: An algorithm for accessible determinization of SHAS.

sets. The additional treatment of apply rules, that dominates the complexity of the algorithm, works as follows: for each $Q \in \mathcal{Q}^{det(A)}$ in the *Agenda* and each state $Q_1 \in \mathcal{Q}^{det(A)}$ in the *Store*, it computes the sets $Q' = \{q' \mid q @ q_1 \rightarrow q', q_1 \in Q_1, q \in Q\}$ and $Q'' = \{q'' \mid q_1 @ q \rightarrow q'', q_1 \in Q_1, q \in Q\}$ and puts all new nonempty sets in both the *Agenda* and the *Store*, while adding dynamically the generated apply rules in the hash set *Rules*. Again, the overall number of elements in the agenda will be $|\mathcal{Q}^{det(A)}|$, requiring time in $O(|\mathcal{Q}^{det(A)}|^2 |\Delta^A|)$. With a precomputation time of A in $O(|A|)$, the total computation will be in $O(|\mathcal{Q}^{det(A)}|^2 |\Delta^A| + |A|)$. \square

5.3 Schema-Based Cleaning for NFAs

We establish natural properties of schema-based cleaning of NFAs that are based on the accessible schema product. The restriction to accessibility cleaning the schema product is relevant for schema-based determinization. The removal of all useless states from the schema product is not feasible in this context.

Example 5.3. For illustration, the schema-based cleaning of $DFA\ det(A_0)$ in Figure 3.6 with respect to schema automaton $word-one-x_{\{a\}}$ is given in Figure 5.4. The only words recognized by both $det(A_0)$ and $word-one-x_{\{a\}}$ are x and xa . For recognizing these two



Figure 5.4: The schema-based cleaning of $\det(A_0)$ constrained by the schema automaton $\text{word-one-}x_{\{a\}}$, up to renaming the states.

$$\begin{array}{c}
 \frac{q \in I^A \quad s \in I^S}{(q, s) \in I^{A \times S}} \quad \frac{q \in F^A \quad s \in F^S \quad (q, s) \in Q^{A \times S}}{(q, s) \in F^{A \times S}} \\
 \\
 \frac{q_1 \xrightarrow{a} q_2 \in \Delta^A \quad s_1 \xrightarrow{a} s_2 \in \Delta^S \quad (q_1, s_1) \in Q^{A \times S}}{(q_1, s_1) \xrightarrow{a} (q_2, s_2) \in \Delta^{A \times S} \quad (q_2, s_2) \in Q^{A \times S}}
 \end{array}$$

Figure 5.5: Accessible product $A \times S = (\Sigma, Q^{A \times S}, I^{A \times S}, F^{A \times S}, \Delta^{A \times S})$.

words, the automaton $\det(A_0)$ does not need the subset of states $\{3\}$, so it can be removed with all its transition rules. Thereby, the word xxa violating the schema is no more recognized after schema-based cleaning, while it was recognized by $\det(A_0)$. Furthermore, note that the state of subset $\{2, 4\}$ do not need to be final after schema-based cleaning. Therefore the word ϵ , which is recognized by the automaton but not by the schema, is no more recognized after schema-based cleaning. So schema-based cleaning may change the language of the automaton but only outside of the schema.

Interestingly, the NFA A_0 in Figure 3.1 is schema-clean for schema $\text{word-one-}x_{\{a\}}$ too, even though it is not perfect, in that it recognizes the words ϵ and xxa which are rejected by the schema. The reason is that for recognizing the words x and xa , which both satisfy the schema, all 3 states and all 4 transition rules of A_0 are needed. By contrast, we already noticed that the accessible determinization $\det(A_0)$ in Figure 3.6 is not schema-clean for schema $\text{word-one-}x_{\{a\}}$. This illustrates that accessible determinization does not always preserve schema-cleanliness. In other words, schema-based cleaning may have a stronger cleaning effect after determinization than before.

The schema-based cleaning of an automaton can be defined based on the accessible product of the automaton with the schema. The accessible product $A \times S$ of two NFAs A and S with alphabet Σ is defined in Figure 5.5. This is the usual product, except that only accessible states are admitted. Clearly, $\mathcal{L}(A \times S) = \mathcal{L}(A) \cap \mathcal{L}(S)$. Let $\Pi_A(A \times S)$ be obtained from the accessible product by projecting away the second component, as formally defined in Figure 5.6. The schema-based cleaning of A with respect to schema S is this projection.

Definition 5.4. $\text{scl}_S(A) = \Pi_A(A \times S)$.

$$\begin{array}{cccc}
\frac{(q, s) \in I^{A \times S}}{q \in I^{\Pi_A(A \times S)}} & \frac{(q, s) \in Q^{A \times S}}{q \in Q^{\Pi_A(A \times S)}} & \frac{(q, s) \in F^{A \times S}}{q \in F^{\Pi_A(A \times S)}} & \frac{(q_1, s_1) \xrightarrow{a} (q_2, s_2) \in \Delta^{A \times S}}{q_1 \xrightarrow{a} q_2 \in \Delta^{\Pi_A(A \times S)}}
\end{array}$$

Figure 5.6: Projection $\Pi_A(A \times S) = (\Sigma, Q^{\Pi_A(A \times S)}, \Delta^{\Pi_A(A \times S)}, I^{\Pi_A(A \times S)}, F^{\Pi_A(A \times S)})$.

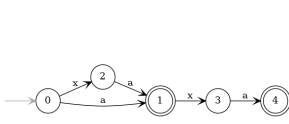


Figure 5.7: A DFA that is schema-clean but not perfect for $word-one-x_\Sigma$.

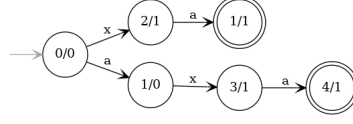


Figure 5.8: The accessible product with $word-one-x_\Sigma$ is schema-clean and perfect for $word-one-x_\Sigma$.

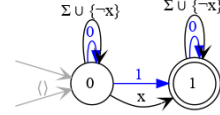


Figure 5.9: The dSHA $one-x_\Sigma$ with alphabet $\Sigma \uplus \{x, \neg x\}$.

The fact that $A \times S$ is restricted to accessible states matches our intuition that all states of $scl_S(A)$ can be used to read some word in $\mathcal{L}(A)$ that satisfies schema S . This can be proven formally under the condition that all states of $A \times S$ are also co-accessible. Clearly, $scl_S(A)$ is obtained from A by removing states, initial states, final states, and transitions rules. So it is smaller or equal in size $|scl_S(A)| \leq |A|$ and language $\mathcal{L}(scl_S(A)) \subseteq \mathcal{L}(A)$. Still, schema-based cleaning preserves the language within the schema.

Proposition 5.5 ([Niehren & Sakho 2021]). $\mathcal{L}(A) \cap \mathcal{L}(S) = \mathcal{L}(scl_S(A)) \cap \mathcal{L}(S)$.

Schema-clean deterministic automata may still not be perfect, in that they may recognize some words outside the schema. This happens for DFAs if some state of is reached, both, by a word satisfying the schema and another word that does not satisfy the schema. An example for a DFA that is schema-clean but not perfect for $word-one-x_\Sigma$ is given in Figure 5.7. It is not perfect since it accepts the non V -structure $xaxa$. The problem is that state 1 can be reached by the words a and xa , so one cannot infer from being in state 1 whether some x was read or not. If one wants to avoid this, one can use the accessible product of the DFA with the schema instead. In the example, this yields the DFA in Figure 5.8 that is schema-clean and perfect for $word-one-x_\Sigma$.

Proposition 5.6 (Folklore). *For any two DFAs A and S with alphabet Σ the accessible product $A \times S$ can be computed in expected amortized time $O(|Q^{A \times S}| |\Sigma| + |A| + |S|)$.*

Proof. An algorithm to compute the fixed points of the inference rules for the

accessible product $A \times S$ in Figure 5.5 can be organized such that only accessible states are considered (similarly to semi-naive Datalog evaluation). This algorithm is presented in Figure 5.10. It dynamically generates the set of rules *Rules* by using perfect dynamic hashing [Dietzfelbinger *et al.* 1994]. Testing set membership is in time $O(1)$ and the addition of elements to the set is in expected amortized time $O(1)$. The algorithm uses a stack, *Agenda*, to memoize all new pairs $(q_1, s_1) \in Q^{A \times S}$ that need to be processed, and a hash set *Store* that saves all processed states $Q^{A \times S}$. We aim not to push the same pair more than once in the *Agenda*. For this, membership to the *Store* is checked before an element is pushed to the *Agenda*. For each pair popped from the stack *Agenda*, the algorithm does the following: for each letter $a \in \Sigma$ it computes the sets $Q = \{q_2 \mid q_1 \xrightarrow{a} q_2 \in \Delta^A\}$ and $R = \{s_2 \mid s_1 \xrightarrow{a} s_2 \in \Delta^S\}$ and then adds the subset of states of $Q \times R$ that were not stored in the hash set *Store* to the agenda. Since A and S are deterministic, there is at most one such pair, so the time for treating one pair on the agenda is in expected amortized time $O(|\Sigma|)$. The overall number of elements in the agenda will be $|Q^{A \times S}|$. Note that Q and R can be computed in $O(1)$ after preprocessing A and S in time $O(|A| + |S|)$. Therefore, we will have a total time of the algorithm in $O(|Q^{A \times S}| |\Sigma| + |A| + |S|)$. \square

Corollary 5.7. *For any two DFAs A and S with alphabet Σ schema-based cleaning $\text{scl}_S(A)$ can be computed in expected amortized time $O(|Q^{A \times S}| |\Sigma| + |A| + |S|)$.*

Proof. By Definition 5.4 it is sufficient to compute the projection of the accessible product $A \times S$. By Proposition 5.6 the product can be computed in time $O(|Q^{A \times S}| |\Sigma| + |A| + |S|)$. Its size cannot be larger than its computation time. The projection can be computed in linear time in the size of $A \times S$, so the overall time is in $O(|Q^{A \times S}| |\Sigma| + |A| + |S|)$ too. \square

5.4 Schema-Based Determinization for NFAs

Schema-based cleaning after determinization becomes impossible in practice if the automaton obtained by determinization is too big. We therefore show next how to integrate schema-based cleaning into automata determinization directly.

The schema-based determinization of A with respect to schema S extends on accessible determinization $\text{det}(A)$. The idea is to run the schema S in parallel with $\text{det}(A)$, in order to keep only those state $Q \in Q^{\text{det}(A)}$ that can be aligned to some state $s \in Q^S$. In this case we write $Q \sim s$.

```

1 fun A × S =
2   let Store = hashset.new(0)
3   let Agenda = list.new() and Rules = hashset.new(0)
4   if initA = {q0} and initS = {s0} then Agenda.add((q0, s0))
5   while Agenda.notEmpty() do
6     let (q1, s1) = Agenda.pop()
7     for a ∈ Σ do
8       let Q = {q2 | q1  $\xrightarrow{a}$  q2 ∈ ΔA} R = {s2 | s1  $\xrightarrow{a}$  s2 ∈ ΔS}
9       for q2 ∈ Q and s2 ∈ R do
10        Rules.add((q1, s1)  $\xrightarrow{a}$  (q2, s2))
11        if not Store.member((q2, s2))
12        then Store.add((q2, s2)) Agenda.push((q2, s2))
13  let initA×S = {(q0, s0) | (q0, s0) ∈ Store} and FA×S = {(q, s) | (q, s) ∈ Store, q ∈ FA, s ∈ FS}
14  return (Σ, Store.toSet(), Rules.toSet(), initA×S, FA×S)

```

Figure 5.10: An algorithm computing the accessible product of DFAS A and S .

$$\begin{array}{c}
\frac{Q \in I^{det(A)} \quad I^S = \{s\}}{Q \in I^{det_S(A)} \quad Q \sim s} \quad \frac{Q \sim s}{Q \in Q^{det_S(A)}} \quad \frac{Q \in F^{det(A)} \quad s \in F^S \quad Q \sim s}{Q \in F^{det_S(A)}} \\
\\
\frac{Q \xrightarrow{a} Q' \in \Delta^{det(A)} \quad Q \sim s \quad s \xrightarrow{a} s' \in \Delta^S}{Q \xrightarrow{a} Q' \in \Delta^{det_S(A)} \quad Q' \sim s'}
\end{array}$$

Figure 5.11: Schema-based determinization $det_S(A) = (\Sigma, Q^{det_S(A)}, \Delta^{det_S(A)}, I^{det_S(A)}, F^{det_S(A)})$.

The schema-determinization $det_S(A)$ is defined in Figure 5.11. The automaton $det_S(A)$ permits to go from any subset $Q \in Q^{det(A)}$ and letter $a \in \Sigma$ to the set of states $Q' = a^{\Delta^{det(A)}}(Q)$, under the condition that there exists schema states $s, s' \in Q^S$ such that $Q \sim s$ and $s \xrightarrow{a} s'$. In this case $Q' \sim s'$ is inferred.

Theorem 1 (Correctness). $det_S(A) = scl_S(det(A))$ for any NFA A and DFA S with the same alphabet.

The theorem states that schema-based determinization yields the same result as accessible determinization followed by schema-based cleaning.

For the correctness proof we collapse the two systems of inference rules for accessible products and projection into a single rule system. This yields the rule systems for schema-based cleaning in Figure 5.12. The rules there define the automaton $\widehat{scl}_S(A)$, that we annotate with a hat, in order to distinguish it from the previous automaton $scl_S(A)$. The rules also infer judgments $(q, s) \in Q^{A \times S}$ that we distinguish by a hat from the previous judgments $(q, s) \in Q^{A \times S}$ of the accessible

$$\begin{array}{c}
\frac{q \in I^A \quad s \in I^S}{q \in I^{\widehat{\text{scl}}_S(A)} \quad (q, s) \in Q^{A \times S}} \quad \frac{q \in F^A \quad s \in F^S \quad (q, s) \in Q^{A \times S}}{q \in F^{\widehat{\text{scl}}_S(A)}} \\
\frac{(q, s) \in Q^{A \times S}}{q \in Q^{\widehat{\text{scl}}_S(A)}} \quad \frac{q_1 \xrightarrow{a} q_2 \in \Delta^A \quad s_1 \xrightarrow{a} s_2 \in \Delta^S \quad (q_1, s_1) \in Q^{A \times S}}{q_1 \xrightarrow{a} q_2 \in \Delta^{\widehat{\text{scl}}_S(A)} \quad (q_2, s_2) \in Q^{A \times S}} \\
\widehat{\text{scl}}_S(A) = (\Sigma, Q^{\widehat{\text{scl}}_S(A)}, \Delta^{\widehat{\text{scl}}_S(A)}, I^{\widehat{\text{scl}}_S(A)}, F^{\widehat{\text{scl}}_S(A)})
\end{array}$$

Figure 5.12: A collapsed rule systems for schema-based cleaning $\widehat{\text{scl}}_S(A)$.

```

1 fun detS(A, S) =
2   let Store = hashset.new(0)
3   let Agenda = list.new() and Rules = hashset.new(0)
4   if IA ≠ ∅ and initS = {s0} then Agenda.add(IA ~ s0)
5   while Agenda.notEmpty() do
6     let (Q1 ~ s1) = Agenda.pop()
7     for a ∈ Σ do
8       let P = {Q2 | Q1  $\xrightarrow{a}$  Q2 ∈ Δdet(A)} and R = {s2 | s1  $\xrightarrow{a}$  s2 ∈ ΔS}
9       for Q2 ∈ P and s2 ∈ R do Rules.add(Q1  $\xrightarrow{a}$  Q2)
10      if not Store.member(Q2 ~ s2)
11      then Store.add(Q2 ~ s2) Agenda.push(Q2 ~ s2)
12   let initdetS(A) = {Q | Q ~ s ∈ Store, Q ∩ IA ≠ ∅} and FdetS(A) = {Q | Q ~ s ∈ Store, Q ∩ FA ≠ ∅}
13   return (Σ, Store.toSet(), Rules.toSet(), initdetS(A), FdetS(A))

```

Figure 5.13: An algorithm for schema-based determinization $\text{det}_S(A)$ of an NFA A and a DFA schema S .

product. The next proposition shows that the system of collapsed inference rules indeed redefines the schema-based cleaning.

Proposition 5.8. *For any two NFAs A and S with the same alphabet:*

$$\text{scl}_S(A) = \widehat{\text{scl}}_S(A) \quad \text{and} \quad Q^{A \times S} = Q^{A \times \widehat{S}}$$

Proof. The two equations are shown by the following four lemmas. The judgments with a hat there are to be inferred by the collapsed system of inference rules in Figure 5.12, while the other judgments are to be inferred with the rule system for accessible products in Figure 5.5.

□

Lemma 5.9. $q \in I^{\widehat{\text{scl}}_S(A)} \text{ iff } q \in I^{\text{scl}_S(A)}.$

Proof. The rule systems of accessible product, projection, and the collapsed system

can be used as following :

$$\frac{q \in I^A \quad s \in I^S}{q \in I^{\widehat{\text{scl}}_S(A)}} \quad \frac{q \in I^A \quad s \in I^S}{\frac{(q,s) \in I^{A \times S}}{q \in I^{\text{scl}_S(A)}}}$$

□

Lemma 5.10. $(q,s) \in Q^{A \times S}$ iff $(q,s) \in Q^{A \times S}$.

Proof. We prove for all $n \geq 0$ that if $(q,s) \in Q^{A \times S}$ has a proof tree of size n then there exists a proof tree for $(q,s) \in Q^{A \times S}$. The proof is by induction on n .

In the case of the rules of the initial states, $(q,s) \in Q^{A \times S}$ is inferred directly whenever $(q,s) \in Q^{A \times S}$ and vice versa, using the following:

$$\frac{q \in I^A \quad s \in I^S}{(q,s) \in I^{A \times S}} \quad \frac{q \in I^A \quad s \in I^S}{(q,s) \in Q^{A \times S}} \quad \frac{q \in I^A \quad s \in I^S}{q \in I^{\widehat{\text{scl}}_S(A)} \quad (q,s) \in Q^{A \times S}}$$

If $(q,s) \in Q^{A \times S}$ is inferred by the internal rule of the collapsed rule system in Figure 5.12. Then the proof tree has the following form for some proof tree T_1 :

$$\frac{q_1 \xrightarrow{a} q_2 \in \Delta^A \quad s_1 \xrightarrow{a} s_2 \in \Delta^S \quad \frac{T_1}{(q_1, s_1) \in Q^{A \times S}}}{(q_2, s_2) \in Q^{A \times S}}$$

This shows that there is a smaller proof tree T_1 for inferring $(q_1, s_1) \in Q^{A \times S}$. So by induction hypothesis applied to T_1 , there exists a proof tree T'_1 for inferring $(q_1, s_1) \in Q^{A \times S}$ with the proof system of accessible products in Figure 5.5:

$$\frac{T'_1}{(q_1, s_1) \in Q^{A \times S}}$$

Therefore, we also have the following proof tree for $(q_2, s_2) \in Q^{A \times S}$ with the internal rule for the accessible product:

$$\frac{q_1 \xrightarrow{a} q_2 \in \Delta^A \quad s_1 \xrightarrow{a} s_2 \in \Delta^S \quad \frac{T'_1}{(q_1, s_1) \in Q^{A \times S}}}{(q_2, s_2) \in Q^{A \times S}}$$

For the inverse direction, if $(q, s) \in Q^{A \times S}$ is inferred by the internal rule of the accessible product rule system in Figure 5.5. Then the proof tree has the following form for some proof tree T_1 :

$$\frac{q_1 \xrightarrow{a} q_2 \in \Delta^A \quad s_1 \xrightarrow{a} s_2 \in \Delta^S \quad \frac{T_1}{(q_1, s_1) \in Q^{A \times S}}}{(q_2, s_2) \in Q^{A \times S}}$$

This means that there is a smaller proof tree T_1 for inferring $(q_1, s_1) \in Q^{A \times S}$. By induction hypothesis applied to T_1 , there exists a proof tree T'_1 for inferring $(q_1, s_1) \in Q^{A \times S}$ with the collapsed system in Figure 5.12:

$$\frac{T'_1}{(q_1, s_1) \in Q^{A \times S}}$$

which leads to the following proof tree for $(q_2, s_2) \in Q^{A \times S}$ with the internal rule for the collapsed system:

$$\frac{q_1 \xrightarrow{a} q_2 \in \Delta^A \quad s_1 \xrightarrow{a} s_2 \in \Delta^S \quad \frac{T'_1}{(q_1, s_1) \in Q^{A \times S}}}{(q_2, s_2) \in Q^{A \times S}}$$

□

Lemma 5.11. $q_1 \xrightarrow{a} q_2 \in \Delta^{\widehat{\text{scl}}_S(A)}$ iff $q_1 \xrightarrow{a} q_2 \in \Delta^{\text{scl}_S(A)}$.

Proof. We prove for all $n \geq 0$ that, if $q_1 \xrightarrow{a} q_2 \in \Delta^{\widehat{\text{scl}}_S(A)}$ has a proof tree of size n , then there exists a proof tree for $q_1 \xrightarrow{a} q_2 \in \Delta^{\text{scl}_S(A)}$ and vice versa. The proof is by induction on n .

If $q_1 \xrightarrow{a} q_2 \in \Delta^{\widehat{\text{scl}}_S(A)}$ is inferred by the internal rule of the collapsed system, the proof tree will have the following for some tree T_1 :

$$\frac{q_1 \xrightarrow{a} q_2 \in \Delta^A \quad s_1 \xrightarrow{a} s_2 \in \Delta^S \quad \frac{T_1}{(q_1, s_1) \in Q^{A \times S}}}{q_1 \xrightarrow{a} q_2 \in \Delta^{\widehat{\text{scl}}_S(A)}}$$

By Lemma 5.10 and the rule of internal rules of the accessible product rule system:

$$\frac{q_1 \xrightarrow{a} q_2 \in \Delta^A \quad s_1 \xrightarrow{a} s_2 \in \Delta^S \quad \frac{T'_1}{(q_1, s_1) \in \mathcal{Q}^{A \times S}}}{(q_1, s_1) \xrightarrow{a} (q_2, s_2) \in \Delta^{A \times S}}$$

For the inverse direction, if $q_1 \xrightarrow{a} q_2 \in \Delta^{\text{scl}_S(A)}$ is inferred by the internal rule of the accessible product, the proof tree will have the following for some tree T_1 :

$$\frac{q_1 \xrightarrow{a} q_2 \in \Delta^A \quad s_1 \xrightarrow{a} s_2 \in \Delta^S \quad \frac{T_1}{(q_1, s_1) \in \mathcal{Q}^{A \times S}}}{\frac{(q_1, s_1) \xrightarrow{a} (q_2, s_2) \in \Delta^{A \times S}}{q_1 \xrightarrow{a} q_2 \in \Delta^{\text{scl}_S(A)}}}$$

By lemma 5.10 and the rule of internal rules of the collapsed system:

$$\frac{q_1 \xrightarrow{a} q_2 \in \Delta^A \quad s_1 \xrightarrow{a} s_2 \in \Delta^S \quad \frac{T'_1}{(q_1, s_1) \in \mathcal{Q}^{A \times S}}}{q_1 \xrightarrow{a} q_2 \in \Delta^{\widehat{\text{scl}_S(A)}}}$$

□

Lemma 5.12. $q \in \widehat{\mathcal{Q}^{\text{scl}_S(A)}}$ iff $q \in \mathcal{Q}^{\text{scl}_S(A)}$ and $q \in \widehat{F^{\text{scl}_S(A)}}$ iff $q \in F^{\text{scl}_S(A)}$.

Proof. We start by proving that $q \in \widehat{\mathcal{Q}^{\text{scl}_S(A)}}$ iff $q \in \mathcal{Q}^{\text{scl}_S(A)}$. By Lemma 5.10, and rules of construction of the accessible product, projection, and collapsed systems, this lemma holds for some proof trees T and T' as follows:

$$\frac{\frac{T}{(q, s) \in \mathcal{Q}^{A \times S}}}{q \in \mathcal{Q}^{\text{scl}_S(A)}} \quad \frac{T'}{q \in \widehat{\mathcal{Q}^{\text{scl}_S(A)}}}$$

Finally, we show that $q \in \widehat{F^{\text{scl}_S(A)}}$ iff $q \in F^{\text{scl}_S(A)}$. Using Lemma 5.10, there exists some proof trees T and T' that infers $(q, s) \in \mathcal{Q}^{A \times S}$ and $(q, s) \in \mathcal{Q}^{A \times S}$ in both ways

$$\begin{array}{c}
\frac{Q \in I^{det(A)} \quad s \in I^S}{Q \in \widehat{scl}_S(det(A)) \quad (Q, s) \in \mathcal{Q}^{det(A) \times S}} \quad \frac{Q \in F^{det(A)} \quad s \in F^S \quad (Q, s) \in \mathcal{Q}^{det(A) \times S}}{Q \in F^{\widehat{scl}_S(det(A))}} \\
\frac{(Q, s) \in \mathcal{Q}^{det(A) \times S}}{Q \in \widehat{scl}_S(det(A))} \quad \frac{Q_1 \xrightarrow{a} Q_2 \in \Delta^{det(A)} \quad s_1 \xrightarrow{a} s_2 \in \Delta^S \quad (Q_1, s_1) \in \mathcal{Q}^{det(A) \times S}}{Q_1 \xrightarrow{a} Q_2 \in \Delta^{\widehat{scl}_S(det(A))} \quad (Q_2, s_2) \in \mathcal{Q}^{det(A) \times S}} \\
\widehat{scl}_S(det(A)) = (\Sigma, \mathcal{Q}^{\widehat{scl}_S(det(A))}, \Delta^{\widehat{scl}_S(det(A))}, I^{\widehat{scl}_S(det(A))}, F^{\widehat{scl}_S(det(A))})
\end{array}$$

Figure 5.14: Instantiation of the collapsed rule system for schema-based cleaning from Figure 5.12 with $det(A)$.

and therefore having the following form of rules:

$$\begin{array}{c}
\frac{q \in F^A \quad s \in F^S \quad \frac{T}{(q, s) \in \mathcal{Q}^{A \times S}}}{q \in F^{\widehat{scl}_S(A)}} \quad \frac{q \in F^A \quad s \in F^S \quad \frac{T'}{(q, s) \in \mathcal{Q}^{A \times S}}}{(q, s) \in F^{A \times S}} \\
\frac{(q, s) \in F^{A \times S}}{q \in F^{\widehat{scl}_S(A)}}
\end{array}$$

□

Proof of Correctness Theorem 1. Instantiating the system of collapsed rules for schema-based cleaning from Figure 5.12 with $det(A)$ for A yields the rule system in Figure 5.14. We can identify the instantiated collapsed system for $\widehat{scl}_S(det(A))$ with that for $det_S(A)$ in Figure 5.11, by identifying the judgments $(Q, s) \in \mathcal{Q}^{det(A) \times S}$ with judgments $Q \sim s$. After renaming the predicates, the inference rules for the corresponding judgments are the same. Hence $\widehat{scl}_S(det(A)) = det_S(A)$, so that Proposition 5.8 implies $scl_S(det(A)) = det_S(A)$. □

Proposition 5.13. *The schema-based determinization $det_S(A)$ for a NFA A and a DFA S over Σ can be computed in expected amortized time $O(|\mathcal{Q}^{det(A) \times S}| |\Sigma| + |\mathcal{Q}^{det_S(A)}| |\Delta^A| + |A| + |S|)$.*

Proof. An algorithm computing the fixed points of the inference rules of schema-based determinization from Figure 5.11 is given in Figure 5.13. It refines the algorithm computing the accessible product with on-the-fly determinization and projection.

On the stack *Agenda*, the algorithm stores alignments $Q \sim s$ such that $(Q, s) \in \mathcal{Q}^{det(A) \times S}$ that were not considered before. Transition rules of $det_S(A)$ are collected in

hash set *Rules*, using the dynamic perfect hashing aforementioned. The alignments $Q_1 \sim s_1$ popped from the agenda are processed as follows: For any letter $a \in \Sigma$, the sets $R = \{Q_2 \mid Q_1 \xrightarrow{a} Q_2 \in \Delta^{det(A)}\}$ and $P = \{s_2 \mid s_1 \xrightarrow{a} s_2 \in \Delta^S\}$ are computed. One then pushes all new pairs $Q_2 \sim s_2$ with $Q_2 \in P$ and $s_2 \in R$ into the agenda, and adds $Q_1 \xrightarrow{a} Q_2$ to the set *Rules*. Since S and $det(A)$ are deterministic there is at most one pair $(Q, s) \in P \times R$ for Q_1 and s_1 . So the time for treating one pair on the agenda is in $O(|\Sigma|)$ plus the time for building the needed transition rules of $det(A)$ from Δ^A on the fly. The time for the on the fly computation of transition rules of $det(A)$ is in time $O(|Q^{det_s(A)}||\Delta^A|)$. The overall number of pairs on the agenda is at most $|Q^{det(A) \times S}|$ so the main while loop of the algorithm requires time in $O(|Q^{det(A) \times S}||\Sigma|)$ apart from on the fly determinization. This will give us an overall complexity for the algorithm in $O(|Q^{det(A) \times S}||\Sigma| + |Q^{det_s(A)}||\Delta^A| + |A| + |S|)$, with consideration of the preprocessing time of A and S . \square

By Proposition 5.1, computing $det(A)$ requires time $O(|Q^{det(A)}||\Delta^A| + |A|)$. Therefore, with Proposition 5.6, the accessible product $det(A) \times S$ can be computed from A and S in time $O(|Q^{det(A) \times S}||\Sigma| + |Q^{det(A)}||\Delta^A| + |A| + |S|)$. Since $Q^{det_s(A)} \subseteq Q^{det(A)}$ the proposition shows that schema-based determinization is at most as efficient in the worst case as accessible determinization followed by schema-based cleaning. If $|Q^{det(A) \times S}||\Sigma| < |Q^{det(A)}||\Delta^A|$ then it is more efficient, since schema-based determinization avoids the computation of $det(A)$ all over. Instead, it only computes the accessible product $det(A) \times S$, which may be way smaller, since exponentially many states of $det(A)$ may not be aligned to any state of S . Sometimes, however, the accessible product may be bigger. In this case, schema-based determinization may be more costly than pure accessible determinization, not followed by schema-based cleaning.

5.5 Schema-Based Cleaning and Determinization for SHAs

We can lift all previous algorithms from NFAS to SHAS while extending the system of inference rules. The additional rules concern tree initial states, that work in analogy to initial states, and also apply rules that works similarly as internal rules. The new inference rules for accessible products $A \times S$ are given in Figure 5.15 and for projection $\Pi_A(A \times S)$ in Figure 5.16. As before we define $scl_S(A) = \Pi_A(A \times S)$. The rules for schema-based determinization $det_S(A)$ are extended in Figure 5.17. The next complexity upper bound, however, now become quadratic with fixed

$$\frac{q \in \langle \rangle^{\Delta^A} \quad s \in \langle \rangle^{\Delta^S}}{(q, s) \in \langle \rangle^{\Delta^{A \times S}}} \quad \frac{\begin{array}{l} (q_1, s_1) \in \mathcal{Q}^{A \times S} \\ (q, s) \in \mathcal{Q}^{A \times S} \end{array} \quad \begin{array}{l} q_1 @ q \rightarrow q_2 \in \Delta^A \\ s_1 @ s \rightarrow s_2 \in \Delta^S \end{array}}{(q_1, s_1) @ (q, s) \rightarrow (q_2, s_2) \in \Delta^{A \times S} \quad (q_2, s_2) \in \mathcal{Q}^{A \times S}}$$

Figure 5.15: Lifting accessible products to SHAs.

$$\frac{(q, s) \in \langle \rangle^{\Delta^{A \times S}}}{q \in \langle \rangle^{\Delta^{\Pi_A(A \times S)}}} \quad \frac{(q_1, s_1) @ (q, s) \rightarrow (q_2, s_2) \in \Delta^{A \times S}}{q_1 @ q \rightarrow q_2 \in \Delta^{\Pi_A(A \times S)}}$$

Figure 5.16: Lifting projections $\Pi_A(A \times S)$ to SHAs.

alphabet:

Proposition 5.14. *If A and S are dSHAs then the accessible product $A \times S$ and the schema-based cleaning $\text{scl}_S(A)$ can be computed in expected amortized time $O(|\mathcal{Q}^{A \times S}|^2 + |\mathcal{Q}^{A \times S}| |\Sigma| + |A| + |S|)$.*

The algorithm in Figure 5.18 is obtained by lifting the algorithm for DFAS in Figure 5.10 to SHAs. For the case of apply rules, we have to combine each pair $(q_1, s_1) \in \mathcal{Q}^{A \times S}$ in the stack *Agenda* with all $(q, s) \in \mathcal{Q}^{A \times S}$ in the hash set *Store*, in both directions. The time to treat these pairs is $O(|\mathcal{Q}^{A \times S}|^2)$, so quadratic in the worst case. As before, no state (q_1, s_1) will be processed twice, due to the set membership test before pushing a pair into the agenda.

Theorem 2 (Correctness). *For any SHA A and dSHA S with the same alphabet:*

$$\text{det}_S(A) = \text{scl}_S(\text{det}(A))$$

The proof presented in Section 5.6 extends on that for NFAS (Theorem 1) in a direct manner.

Proposition 5.15. *The schema-based determinization $\text{det}_S(A)$ of a SHA A with respect to a dSHA S can be computed in expected amortized time $O(|\mathcal{Q}^{\text{det}(A) \times S}|^2 + |\mathcal{Q}^{\text{det}(A) \times S}| |\Sigma| + |\mathcal{Q}^{\text{det}_S(A)}|^2 |\Delta^A| + |A| + |S|)$.*

This proposition follows the result in Proposition 5.13 with an additional quadratic factor in the size of states of the product $\text{det}(A) \times S$ and the states of the schema-based determinized automaton. This is always due to the apply rules of type \mathcal{Q}^3 .

$$\frac{\langle \rangle^{\Delta^S} = \{s\}}{\langle \rangle^{\Delta^A} \in \langle \rangle^{\Delta^{det_S(A)}} \quad \langle \rangle^{\Delta^A} \sim s} \quad \frac{s_1 @ s_2 \rightarrow s' \in \Delta^S \quad Q_1 \sim s_1 \quad Q_2 \sim s_2 \quad Q_1 @ Q_2 \rightarrow Q' \in \Delta^{det(A)}}{Q_1 @ Q_2 \rightarrow Q' \in \Delta^{det_S(A)} \quad Q' \sim s'}$$

Figure 5.17: Extension of schema-based determinization to SHAS.

```

1 fun A × S =
2   let Store = hashset.new(0)
3   let Agenda = list.new() and Rules = hashset.new(0)
4   if initA = {q0} and initS = {s0} then Agenda.add((q0, s0))
5   while Agenda.notEmpty() do
6     let (q1, s1) = Agenda.pop()
7     for a ∈ Σ do
8       let Q = {q2 | q1  $\xrightarrow{a}$  q2 ∈ ΔA} and R = {s2 | s1  $\xrightarrow{a}$  s2 ∈ ΔS}
9       for q2 ∈ Q and s2 ∈ R do Rules.add((q1, s1)  $\xrightarrow{a}$  (q2, s2))
10        if not Store.member((q2, s2))
11          then Store.add((q2, s2)) Agenda.push((q2, s2))
12    for (q, s) ∈ Store do
13      let Q' = {q2 | q1@q → q2 ∈ ΔA} and R' = {s2 | s1@s → s2 ∈ ΔS}
14      for q2 ∈ Q' and s2 ∈ R' do Rules.add((q1, s1)  $\xrightarrow{(q,s)}$  (q2, s2))
15        if not Store.member((q2, s2))
16          then Store.add((q2, s2)) Agenda.push((q2, s2))
17      let Q'' = {q2 | q@q1 → q2 ∈ ΔA} and R'' = {s2 | s@s1 → s2 ∈ ΔS}
18      for q2 ∈ Q'' and s2 ∈ R'' do Rules.add((q, s)  $\xrightarrow{(q_1,s_1)}$  (q2, s2))
19        if not Store.member((q2, s2))
20          then Store.add((q2, s2)) Agenda.push((q2, s2))
21   let initA×S = {(q0, s0) | (q0, s0) ∈ Store} and FA×S = {(q, s) | (q, s) ∈ Store, q ∈ FA, s ∈ FS}
22   return (Σ, Store.toSet(), Rules.toSet(), initA×S, FA×S)

```

Figure 5.18: An algorithm computing the accessible product of dSHAS A and S.

Proof. Analogously to the case of NFAS on words. The algorithm in Figure 5.19 computes the fixed point of the inference rules of schema-based determinization of SHAS. As for NFAS, it stores untreated alignments on a stack *Agenda* and processed alignments in a hash set *Store*. It also collects transition rules in a hash set *Rules*. New alignments can now be produced by the inference rule for apply transitions: for each alignment $Q_1 \sim s_1$ on the *Agenda* and $Q_2 \sim s_2$ in the *Store*, the algorithm computes the sets $\{s \mid s_1 @ s_2 \rightarrow s \in \Delta^S\}$ and $\{Q \mid Q_1 @ Q_2 \rightarrow Q \in \Delta^{det(A)}\}$ and pushes all pairs $Q \sim s$ outside the *Store* to the *Agenda*. There may be at most one such pair since *S* and *det(A)* are deterministic. We also have to consider the symmetric case where $Q_1 \sim s_1$ on the store and $Q_2 \sim s_2$ on the *Agenda*. Thus, it is in time $O(|\mathcal{Q}^{det(A) \times S}|^2)$ which is quadratic in the worst case. Added to the latter, the cost of computing the transition of *det(A)* on the fly which is in worst case $O(|\mathcal{Q}^{det_S(A)}|^2 |\Delta^A| + |A|)$.


```

1 fun detS(A, S) =
2   let Store = hashset.new(0)
3   let Agenda = list.new() and Rules = hashset.new(0)
4   if  $I^A \neq \emptyset$  and  $I^S = \{s_0\}$  then Agenda.add( $I^A \sim s_0$ )
5   while Agenda.notEmpty() do
6     let  $(Q_1 \sim s_1) = \text{Agenda.pop}()$ 
7     for  $a \in \Sigma$  do
8       let  $P = \{Q_2 \mid Q_1 \xrightarrow{a} Q_2 \in \Delta^{det(A)}\}$  and  $R = \{s_2 \mid s_1 \xrightarrow{a} s_2 \in \Delta^S\}$ 
9       for  $Q_2 \in P$  and  $s_2 \in R$  do Rules.add( $Q_1 \xrightarrow{a} Q_2$ )
10      if not Store.member( $Q_2 \sim s_2$ )
11      then Store.add( $Q_2 \sim s_2$ ) Agenda.push( $Q_2 \sim s_2$ )
12    for  $(Q \sim s) \in \text{Store}$  do
13      let  $P' = \{Q_2 \mid Q_1 @ Q \rightarrow Q_2 \in \Delta^{det(A)}\}$  and  $R' = \{s_2 \mid s_1 @ s \rightarrow s_2 \in \Delta^S\}$ 
14      for  $Q_2 \in P'$  and  $s_2 \in R'$  do Rules.add( $Q_1 @ Q \rightarrow Q_2$ )
15      if not Store.member( $Q_2 \sim s_2$ )
16      then Store.add( $Q_2 \sim s_2$ ) Agenda.push( $Q_2 \sim s_2$ )
17      let  $P'' = \{Q_2 \mid Q @ Q_1 \rightarrow Q_2 \in \Delta^{det(A)}\}$  and  $R'' = \{s_2 \mid s @ s_1 \rightarrow s_2 \in \Delta^S\}$ 
18      for  $Q_2 \in P''$  and  $s_2 \in R''$  do Rules.add( $Q @ Q_1 \rightarrow Q_2$ )
19      if not Store.member( $Q_2 \sim s_2$ )
20      then Store.add( $Q_2 \sim s_2$ ) Agenda.push( $Q_2 \sim s_2$ )
21   let  $\text{init}^{det_S(A)} = \{Q \mid Q \sim s \in \text{Store}, Q \cap I^A \neq \emptyset\}$  and  $F^{det_S(A)} = \{Q \mid Q \sim s \in \text{Store}, Q \cap F^A \neq \emptyset\}$ 
22   return ( $\Sigma, \text{Store.toSet}(), \text{Rules.toSet}(), \text{init}^{det_S(A)}, F^{det_S(A)}$ )

```

Figure 5.19: An algorithm for schema-based determinization of an SHA A and a $d\text{SHA}$ schema S

Therefore, having the whole algorithm running, including the time for computing the internal rules, in $O(|Q^{det(A) \times S}|^2 + |Q^{det(A) \times S}| |\Sigma| + |Q^{det_S(A)}|^2 |\Delta^A| + |A| + |S|)$. \square

By Propositions 5.2 and 5.14, computing $\text{scl}_S(det(A))$ by schema-based cleaning after accessible determinization needs time in $O(|Q^{det(A) \times S}|^2 + |Q^{det(A) \times S}| |\Sigma| + |Q^{det(A)}|^2 |\Delta^A| + |A| + |S|)$. This complexity bound is similar to that of schema-based determinization from Proposition 5.15. Since $Q^{det_S(A)} \subseteq Q^{det(A)}$, Proposition 5.15 shows that the worst case time complexity of schema-based determinization is never worse than for schema-based cleaning after determinization.

5.6 Correctness Proof

The proof of Theorem 2 extends on the proof of the case of words (Theorem 1) in a direct manner.

We first lift the collapsed rule system for NFAS from Figure 5.12 to SHAS in Figure 5.20, and then show that collapsed rules also redefine the schema-based cleaning $\widehat{\text{scl}}_S(A) = \text{scl}_S(A)$ in the case of SHAS.

$$\begin{array}{c}
\frac{q \in \langle \rangle^{\Delta^A} \quad s \in \langle \rangle^{\Delta^S}}{q \in \langle \rangle^{\Delta^{\widehat{\text{scl}}_S(A)}} \quad (q, s) \in \mathcal{Q}^{A \times S}} \\
\frac{q_1 @ q \rightarrow q_2 \in \Delta^A \quad s_1 @ s \rightarrow s_2 \in \Delta^S \quad (q_1, s_1) \in \mathcal{Q}^{A \times S} \quad (q, s) \in \mathcal{Q}^{A \times S}}{q_1 @ q \rightarrow q_2 \in \Delta^{\widehat{\text{scl}}_S(A)} \quad (q_2, s_2) \in \mathcal{Q}^{A \times S}}
\end{array}$$

Figure 5.20: Lifting the collapsed rule system from NFAS to SHAS.

Proposition 5.16. *For any two SHAS A and S with the same alphabet:*

$$\Pi_A(A \times S) = \widehat{\text{scl}}_S(A) \quad \text{and} \quad \mathcal{Q}^{A \times S} = \mathcal{Q}^{A \times S}$$

Proof. The two equations are shown either by new lemmas or an extension of the lemmas from the proof of Theorem 1, whereas all unchanged existing lemmas hold (Lemmas 5.9, 5.11 and 5.12). □

Lemma 5.17. $q \in \langle \rangle^{\Delta^{\widehat{\text{scl}}_S(A)}} \text{ iff } \langle \rangle^{\Delta^{\text{scl}_S(A)}}$

Proof. The rule systems of accessible product, projection and the collapsed system can be used as following :

$$\begin{array}{c}
\frac{q \in \langle \rangle^{\Delta^A} \quad s \in \langle \rangle^{\Delta^S}}{q \in \langle \rangle^{\Delta^{\widehat{\text{scl}}_S(A)}}} \quad \frac{q \in \langle \rangle^{\Delta^A} \quad s \in \langle \rangle^{\Delta^S}}{(q, s) \in \langle \rangle^{\Delta^{A \times S}}} \\
\frac{(q, s) \in \langle \rangle^{\Delta^{A \times S}}}{q \in \langle \rangle^{\Delta^{\widehat{\text{scl}}_S(A)}}}
\end{array}$$

□

Lemma 5.18 (extends Lemma 5.10). $(q, s) \in \mathcal{Q}^{A \times S} \text{ iff } (q, s) \in \mathcal{Q}^{A \times S}$.

All proofs for initial states rules and internal rules from the previous lemma hold and we extend it for tree initial rules and apply rules:

Proof. Similarly, we prove for all $n \geq 0$ that if $(q, s) \in \mathcal{Q}^{A \times S}$ has a proof tree of size n then there exists a proof trees for $(q, s) \in \mathcal{Q}^{A \times S}$. The proof is by induction on n .

In the case of tree initial rules, $(q, s) \in \mathcal{Q}^{A \times S}$ is inferred directly whenever $(q, s) \in \mathcal{Q}^{A \times S}$ and vice versa, using the following:

$$\begin{array}{c}
\frac{q \in \langle \rangle^{\Delta^A} \quad s \in \langle \rangle^{\Delta^S}}{(q, s) \in \langle \rangle^{\Delta^{A \times S}}} \quad (q, s) \in \mathcal{Q}^{A \times S} \quad \frac{q \in \langle \rangle^{\Delta^A} \quad s \in \langle \rangle^{\Delta^S}}{q \in \langle \rangle^{\widehat{\text{scl}}_S(A)} \quad (q, s) \in \mathcal{Q}^{A \times S}}
\end{array}$$

In the same spirit, if $(q, s) \in Q^{A\widehat{\times}S}$ is inferred by the apply rule of the same system, then the proof tree has the following form for some proof trees T_1 and T :

$$\frac{q_1 @ q \rightarrow q_2 \in \Delta^A \quad s_1 @ s \rightarrow s_2 \in \Delta^S \quad \frac{T_1}{(q_1, s_1) \in Q^{A\widehat{\times}S}} \quad \frac{T}{(q, s) \in Q^{A\widehat{\times}S}}}{(q_2, s_2) \in Q^{A\widehat{\times}S}}$$

This means that there are smaller proof trees T_1 and T for inferring respectively $(q_1, s_1) \in Q^{A\widehat{\times}S}$ and $(q, s) \in Q^{A\widehat{\times}S}$. Correspondingly, by induction hypothesis applied to T_1 and T , there exists T'_1, T' for inferring $(q_1, s_1) \in Q^{A \times S}$ and $(q, s) \in Q^{A \times S}$:

$$\frac{T'_1}{(q_1, s_1) \in Q^{A \times S}} \quad \frac{T'}{(q, s) \in Q^{A \times S}}$$

Thus allowing the following proof tree for $(q_2, s_2) \in Q^{A \times S}$ with the apply rule of the accessible product:

$$\frac{q_1 @ q \rightarrow q_2 \in \Delta^A \quad s_1 @ s \rightarrow s_2 \in \Delta^S \quad \frac{T'_1}{(q_1, s_1) \in Q^{A \times S}} \quad \frac{T'}{(q, s) \in Q^{A \times S}}}{(q_2, s_2) \in Q^{A \times S}}$$

For the inverse direction of the apply rules, and using the induction hypothesis, we will be able to infer $(q_2, s_2) \in Q^{A\widehat{\times}S}$ with some T'_1 and T' and ending with the following proof tree:

$$\frac{q_1 @ q \rightarrow q_2 \in \Delta^A \quad s_1 @ s \rightarrow s_2 \in \Delta^S \quad \frac{T'_1}{(q_1, s_1) \in Q^{A\widehat{\times}S}} \quad \frac{T'}{(q, s) \in Q^{A\widehat{\times}S}}}{(q_2, s_2) \in Q^{A\widehat{\times}S}}$$

□

Lemma 5.19. $q_1 @ q \rightarrow q_2 \in \Delta^{\widehat{\text{scl}}_s(A)}$ iff $q_1 @ q \rightarrow q_2 \in \Delta^{\text{scl}_s(A)}$.

Proof. Following the same logic in Lemma 5.11, this lemma holds by the following

sequence of rules, for some proof trees T_1 , T , T'_1 and T' :

$$\begin{array}{c}
 \frac{q_1 @ q \rightarrow q_2 \in \Delta^A \quad s_1 @ s \rightarrow s_2 \in \Delta^S \quad \frac{T_1}{(q_1, s_1) \in Q^{A \times S}} \quad \frac{T}{(q, s) \in Q^{A \times S}}}{q_1 @ q \rightarrow q_2 \in \Delta^{\widehat{\text{scl}}_S(A)}} \\
 \\
 \frac{q_1 @ q \rightarrow q_2 \in \Delta^A \quad s_1 @ s \rightarrow s_2 \in \Delta^S \quad \frac{T'_1}{(q_1, s_1) \in Q^{A \times S}} \quad \frac{T'}{(q, s) \in Q^{A \times S}}}{(q_1, s_1) @ (q, s) \rightarrow (q_2, s_2) \in \Delta^{A \times S}} \\
 \\
 \frac{(q_1, s_1) @ (q, s) \rightarrow (q_2, s_2) \in \Delta^{A \times S}}{q_1 @ q \rightarrow q_2 \in \Delta^{\widehat{\text{scl}}_S(A)}}
 \end{array}$$

For the inverse direction, the sequence of rules, for some proof trees T_1 , T , T'_1 and T' will be:

$$\begin{array}{c}
 \frac{q_1 @ q \rightarrow q_2 \in \Delta^A \quad s_1 @ s \rightarrow s_2 \in \Delta^S \quad \frac{T_1}{(q_1, s_1) \in Q^{A \times S}} \quad \frac{T}{(q, s) \in Q^{A \times S}}}{(q_1, s_1) @ (q, s) \rightarrow (q_2, s_2) \in \Delta^{A \times S}} \\
 \\
 \frac{(q_1, s_1) @ (q, s) \rightarrow (q_2, s_2) \in \Delta^{A \times S}}{q_1 @ q \rightarrow q_2 \in \Delta^{\widehat{\text{scl}}_S(A)}} \\
 \\
 \frac{q_1 @ q \rightarrow q_2 \in \Delta^A \quad s_1 @ s \rightarrow s_2 \in \Delta^S \quad \frac{T'_1}{(q_1, s_1) \in Q^{A \times S}} \quad \frac{T'}{(q, s) \in Q^{A \times S}}}{q_1 @ q \rightarrow q_2 \in \Delta^{\widehat{\text{scl}}_S(A)}}
 \end{array}$$

□

Proof of Correctness Theorem 2. For the proof of Theorem 2 for SHA, we extend the instantiation of the rule system for schema-based cleaning from Figure 5.20 with $\det(A)$, yielding the rule system in Figure 5.21. The whole instantiation holds with the previous instantiation for words and we can still identify the rule system for $\widehat{\text{scl}}_S(\det(A))$ with the rule system $\det_S(A)$. With the same identification of judgments and predicate renaming, the two systems are still exactly the same. Having $\widehat{\text{scl}}_S(\det(A)) = \det_S(A)$ implies, by Proposition 5.16 $\text{scl}_S(\det(A)) = \det_S(A)$. □

$$\begin{array}{c}
\frac{Q \in \langle \rangle^{\Delta^{det(A)}} \quad s \in \langle \rangle^{\Delta^S}}{Q \in \langle \rangle^{\Delta^{\widehat{scl}_S(det(A))}} \quad (Q, s) \in \mathcal{Q}^{det(A) \times S}} \\
\\
\frac{Q_1 @ Q \rightarrow Q_2 \in \Delta^{det(A)} \quad s_1 @ s \rightarrow s_2 \in \Delta^S \quad (Q_1, s_1) \in \mathcal{Q}^{det(A) \times S} \quad (Q, s) \in \mathcal{Q}^{det(A) \times S}}{Q_1 @ Q \rightarrow Q_2 \in \Delta^{\widehat{scl}_S(det(A))} \quad (Q_2, s_2) \in \mathcal{Q}^{det(A) \times S}}
\end{array}$$

Figure 5.21: Extending the instantiation of the alt. definition of schema-based cleaning: $\widehat{scl}_S(det(A)) = (\Sigma, \mathcal{Q}^{\widehat{scl}_S(det(A))}, \Delta^{\widehat{scl}_S(det(A))}, I^{\widehat{scl}_S(det(A))}, F^{\widehat{scl}_S(det(A))})$.

5.7 Scaling Experiments

In this section, we present an experimental evaluation of the sizes of the automata produced by the different determinization methods. For this, we consider a scalable family of SHAS that is compiled from the following scalable family of XPATH queries where n and m are natural numbers:

$(Q_{n.m}) \quad // * [self::a0 \text{ or } \dots \text{ or } self::an]$
 $[\text{descendant}:: * [self::b0 \text{ or } \dots \text{ or } self::bm]]$

Query $Q_{n.m}$ selects all elements of an XML document, that are named by either of $a0, \dots, an$ and have some descendant element named by either of $b1, \dots, bm$. We compile those XPATH queries to SHAS based on the compiler from [Niehren & Sakho 2021]. As schema automaton, we chose the one-sorted variant of the dSHA $S = xml\text{-}seq\&one\text{-}x$ from Figure 4.5, so that $\mathbf{S} = \mathcal{L}(S) = \llbracket XML\text{-}Seq\text{-}x \rrbracket \cap \llbracket One\text{-}x \rrbracket$. Note that this SHA has a typed alphabet where each letter has a unique type in $\{node_type, name, namespace, char\}$, and typed else rules. Note also that the special symbol $\neg x$ will be used by this automaton and throughout our experiments, while we could safely ignore it in our theoretical treatments.

The results of our experiments are summarized in Table 5.1. For each automaton we present two numbers, $size(\#states)$, its size and the number of its states. Unless specified otherwise, we use a timeout of 1000 seconds whenever calling some determinization algorithm. Fields of the table are left blank if an exception was raised. This happens when the determinization algorithm reached the timeout, the memory was filled, or the stack overflowed. We conducted all the experiments on a Dell laptop with the following specs: Intel® Core™ i7-10875H CPU @ 2.30 GHz, 16 cores, and 32 GB of RAM.

For illustration, we present the minimization of the schema-based determiniza-

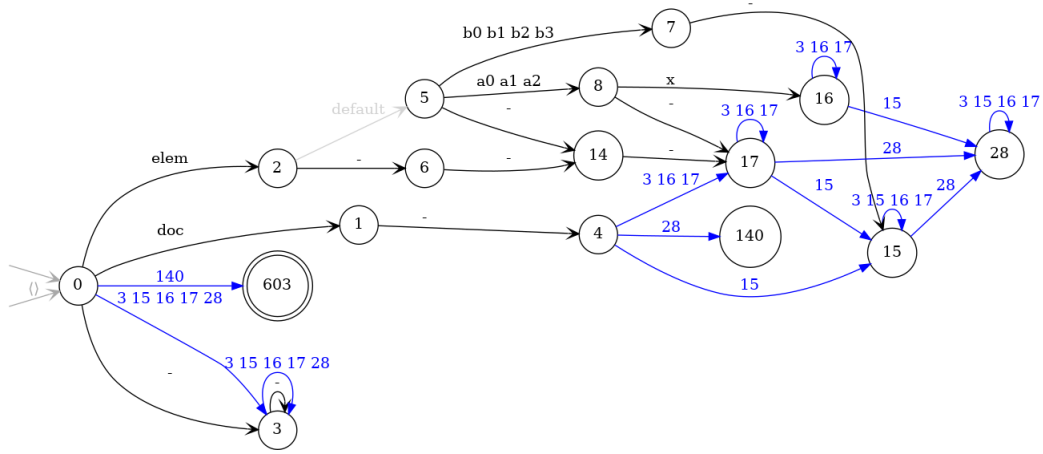


Figure 5.22: The automaton $mini(det_S(A))$ where $A = sha(Q3.4)$.

tion $mini(det_S(A))$ where $A = sha(Q3.4)$ in Figure 5.22. Our tool produces this dSHA from a two-sorted version, with $\#states = 20$ states and $size = 84$, of which we omit the details. The slightly bigger dSHA for the schema product $mini(det(A \times S))$ is given in Figure 5.23. It is computed by our tool from the two-sorted variant of this dSHA with $\#states = 43$ and $size = 168$.

Table 5.1 reports the statistics on the number of states and sizes of the SHAs obtained from the queries $Q = Qn.m$ for $n \in \{1, 2, 3, 4, 5, 6\}$ and $m \in \{1, 2, 3, 4\}$, and for $n = 6$ and $m \in \{5, 6\}$. The first column of the table is about the SHA $A = sha(Q)$ produced by the compiler of [Niehren & Sakho 2021] when applied to query Q . The second column $det(A)$ is obtained from SHA A by accessible determinization. The blank square in column $det(A)$ for query Q4.4 was raised by a timeout of the determinization algorithm. As one can see, this happens for all larger pairs (n, m) . The size of $det(A)$ seems to grow exponentially in $|A| = O(n + m)$.

In the third column $det(A \times S)$, the determinization of the schema product is presented. For all A it yields a much smaller automaton than $det(A)$. For Q4.3 for instance, $det(A)$ has size 53,550 (2161) while $det(A \times S)$ has size 5412 (438). The computation continues successfully until Q6.4. For the larger queries Q6.5 and Q6.6, our determinizer runs out of memory.

The fourth column $det_S(A)$ reports on schema-based determinization. For Q4.3 for instance, we obtain 3534 (329). Here and in all given examples, both measures are always smaller for $det_S(A)$ than for $det(A \times S)$. This may not always be the case, but both approaches yield decent results generally. The numbers for the $det_S(A)$ where $A = sha(Q6.6)$ are marked in gray, since its computation took around one hour, so we obtain it only when ignoring the timeout. In contrast to $det(A \times S)$,

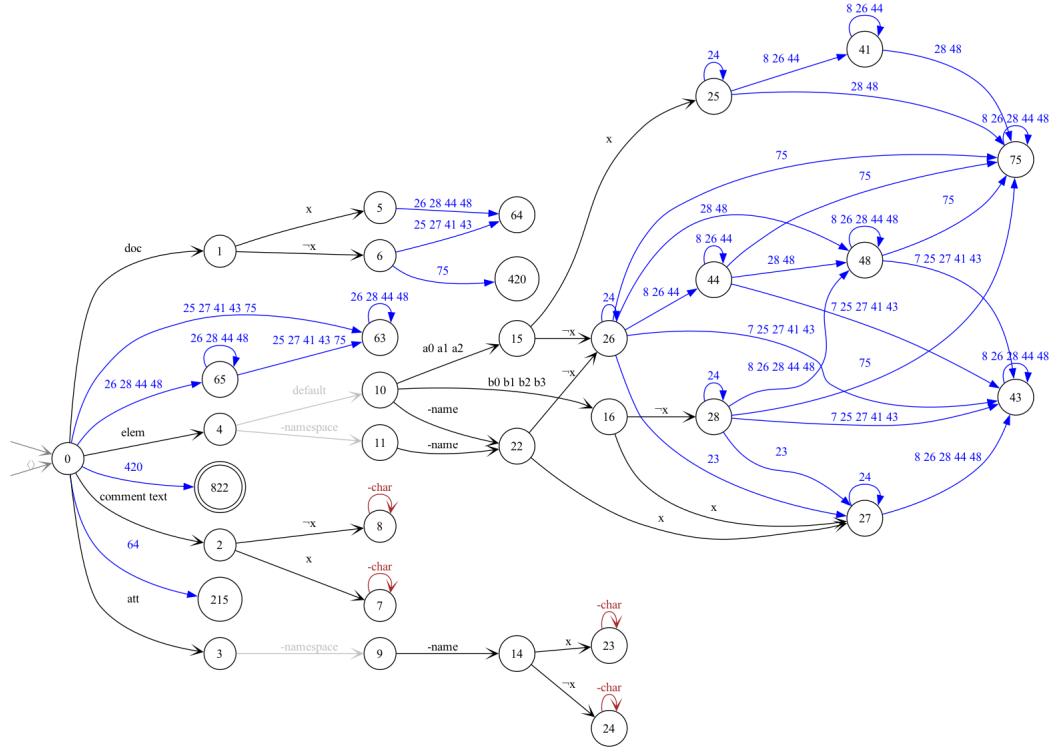


Figure 5.23: The automaton $mini(det(A \times S))$ where $A = sha(Q3.4)$.

however, the computation of $det_S(A)$ did not run out of memory.

The numbers in columns 2-4 of Table 5.1 suggest that the sizes of $det_S(A)$ and $det(A \times S)$ are reduced by more than a linear factor compared to $det(A)$ depending on $|A| = n + m$.

The fifth column $scl_S(det(A))$ contains the schema-based cleaning of $det(A)$. This automaton is equal to $det_S(A)$ by Correctness Theorem 2. Nevertheless, this field is left blank for all queries but the smallest Q2.1, since our Datalog implementation of schema-based cleaning quickly runs out of memory for automata with many states.

In the last two columns for $mini(det(A \times S))$ and $mini(det_S(A))$ we report the size of the minimization of $det(A \times S)$ and $det_S(A)$ respectively. We note that the number of states is independent of n and m , while the number of rules depends linearly of $n + m$. It also turns out that $mini(det_S(A))$ is always smaller than $mini(det(A \times S))$, if both can be computed successfully.

Table 5.2 presents the time in seconds for the determinization of the schema product $det(A \times S)$ and for the schema-based determinization $det_S(A)$. They grow linearly in dependence of the size of the output, from 0.9 seconds until reaching

query Q	$A = sha(Q)$	$det(A)$	$det(A \times S)$	$det_S(A)$	$scl_S(det(A))$	$mini(det(A \times S))$	$mini(det_S(A))$
Q2.1	166 (67)	1380 (101)	540 (92)	284 (53)	284 (53)	160 (43)	73 (20)
Q2.2	199 (79)	3635 (214)	1488 (167)	830 (106)		162 (43)	75 (20)
Q2.3	232 (91)	9574 (471)	4174 (334)	2424 (227)		164 (43)	77 (20)
Q2.4	265 (103)	24813 (1052)	11502 (713)	6826 (504)		166 (43)	79 (20)
Q3.1	203 (81)	3282 (204)	625 (104)	351 (64)		162 (43)	75 (20)
Q3.2	243 (95)	8660 (447)	1716 (191)	1025 (129)		164 (43)	77 (20)
Q3.3	283 (109)	22516 (996)	4793 (386)	2979 (278)		166 (43)	79 (20)
Q3.4	323 (123)	57328 (2225)	13148 (829)	8341 (619)		168 (43)	81 (20)
Q4.1	240 (95)	8020 (435)	710 (116)	418 (75)		164 (43)	77 (20)
Q4.2	287 (111)	20945 (968)	1944 (215)	1220 (152)		166 (43)	79 (20)
Q4.3	334 (127)	53550 (2161)	5412 (438)	3534 (329)		168 (43)	81 (20)
Q4.4	381 (143)		14794 (945)	9856 (734)		170 (43)	83 (20)
Q5.1	277 (109)	19722 (954)	795 (128)	485 (86)		166 (43)	79 (20)
Q5.2	331 (127)	50666 (2129)	2172 (239)	1415 (175)		168 (43)	81 (20)
Q5.3	385 (145)		6031 (490)	4089 (380)		170 (43)	83 (20)
Q5.4	439 (163)		16440 (1061)	11371 (849)			85 (20)
Q6.1	314 (123)	48212 (2113)	880 (140)	552 (97)		168 (43)	81 (20)
Q6.2	375 (143)		2400 (263)	1610 (198)		170 (43)	83 (20)
Q6.3	436 (163)		6650 (542)	4644 (431)		172 (43)	85 (20)
Q6.4	497 (183)		18086 (1177)	12886 (964)			87 (20)
Q6.5	558 (203)			34376 (2169)			
Q6.6	619 (223)			88666 (4862)			

Table 5.1: Statistics of automata for XPATN queries: size(#states)

the timeout. This shows that the times for computing $det(A \times S)$ and $det_S(A)$ can be estimated directly from the sizes of these automata. Note that this is not the case for $scl_S(det(A))$ reported in Table 5.1, in contrast, since the intermediate result $det(A)$ there is larger than the final result, so that the timeout is reached very quickly except for the smallest query Q2.1.

query Q	$det(A \times S)$		$det_S(A)$	
	size(#states)	time in sec	size(#states)	time in sec
Q2.1	540 (92)	1.4	284 (53)	0.9
Q2.2	1488 (167)	3.1	830 (106)	1.5
Q2.3	4174 (334)	11.1	2424 (227)	3.9
Q2.4	11502 (713)	54.5	6826 (504)	16.9
Q3.1	625 (104)	1.6	351 (64)	1.1
Q3.2	1716 (191)	4.1	1025 (129)	1.9
Q3.3	4793 (386)	16.9	2979 (278)	6.2
Q3.4	13148 (829)	84.9	8341 (619)	30.6
Q4.1	710 (116)	1.9	418 (75)	1.3
Q4.2	1944 (215)	5.5	1220 (152)	2.7
Q4.3	5412 (438)	22.9	3534 (329)	9.3
Q4.4	14794 (945)	120	9856 (734)	46.5
Q5.1	795 (128)	2.2	485 (86)	1.5
Q5.2	2172 (239)	7.1	1415 (175)	3.3
Q5.3	6031 (490)	32.2	4089 (380)	13
Q5.4	16440 (1061)	174.6	11371 (849)	71.7
Q6.1	880 (140)	2.8	552 (97)	1.6
Q6.2	2400 (263)	9.2	1610 (198)	4.6
Q6.3	6650 (542)	44.33	4644 (431)	19
Q6.4	18086 (1177)	231.8	12886 (964)	101.4
Q6.5			34376 (2169)	569.7
Q6.6			88666 (4862)	3284.5

Table 5.2: Computation times in seconds for the determinization of the schema product $A \times S$ and for schema-based determinization $det_S(A)$ where $A = sha(Q)$.

Benchmark of Deterministic Automaton for XPATH Queries

Abstract

We provide a benchmark collection of deterministic stepwise hedge automata (dSHAs) for 78 regular forward XPATH queries selected from the corpus of more than 21,000 XPATH queries that Lick and Schmitz extracted from real-world XSLT and XQUERY programs. The objective of the selection of the subcorpus was to find the most complex of regular forward XPATH queries in the corpus, so that these can be used for benchmarking automata based query evaluators. We compile all selected regular XPATH queries to SHAs and determinize them based on a schema, capturing the XML data model and the fact that any answer of a monadic path query must return a single node. The dSHAs obtained are also compiled to deterministic nested word automata of similar size, all having the weak single entry property.

Contents

6.1	Introduction	138
6.2	Subcorpus of Lick and Schmitz' Benchmark	140
6.3	A Schema for XML Documents	146
6.4	Compiler to Automata	146
6.4.1	Parser	147
6.4.2	Nested Regular Expressions	147
6.4.3	Compiler to SHAs	148
6.4.4	Schema-Based Determinization	150

6.4.5	Minimization	151
6.4.6	Examples	151
6.5	Testing Automata	153
6.5.1	Membership for Samples	153
6.5.2	Query Evaluation	154
6.6	Automata Statistics	155
6.6.1	XPATHMARK Benchmark	155
6.6.2	Lick and Schmitz' Benchmark	156
6.7	Example Automata for Lick and Schmitz' Benchmark	159

6.1 Introduction

Algorithms for regular XPATH queries are often based on automata [Muñoz & Riveros 2022b, Debarbieux *et al.* 2015, Mozafari *et al.* 2012, Maneth & Nguyen 2010, Gauwin *et al.* 2009b, Neumann & Seidl 1998]. Compiling regular path queries or nested regular expressions without filters, intersection, and complement to Nwas or SHAS can be done in polynomial time, and is thus feasible.

The compilation of filters can be done by automata intersections. This remains in polynomial time, as long as the number of filters is bounded. In the benchmark of Lick and Schmitz [Lick & Schmitz 2022], the number of filters in regular XPATH queries is bounded by 5 (see Section 4.4.2). The same holds for the regular forward XPATH queries of the XPATHMARK benchmark [Franceschet 2005b] (see Table 4.1). Therefore, the regular XPATH queries with filters to nondeterministic automata is indeed unproblematic in practice. Automata with alternation (as used for instance in [Maneth & Nguyen 2010]) are not needed to represent intersections more concisely.

However, obtaining deterministic automata for regular XPATH queries is often more problematic, as discussed already in the open challenges of the general Introduction (Section 1.2) and also in Chapter 5.

Niehren and Sakho [Niehren & Sakho 2021] proposed to use the determinization algorithm for SHAS to improve on this situation, followed by schema-based cleaning. In this way, deterministic SHAS and Nwas of decent size could be obtained for the forward regular XPATH queries for the XPATHMARK benchmark [Franceschet 2005b] that we reported already in Section 4.4.1.

The determinization of SHAs may still lead to unreasonably large automata for practically relevant XPATH queries, even though the problem doesn't show up for the queries from the XPATHMARK benchmark. This happens in particular for the SHAs obtained for the regular forward XPATH queries in Lick and Schmitz' benchmark (Section 4.4.2). See column $\det(A)$ of Table 6.4 for few examples. The first example in this list is the query Q_{01705} , for which the nondeterministic SHA constructed has 350 states and overall size 772:

```
book | article | part | reference | preface | chapter |
bibliography | appendix | glossary | section | sect1 |
sect2 | sect3 | sect4 | sect5 | refentry | colophon |
bibliodiv[title] | setindex | index
```

The accessible determinization of this SHA needs more than 100 seconds – the timeout we selected – as for 37% of the 78 queries that we selected from Lick and Schmitz' benchmark in Section 6.2.

In Chapter 5, we showed that schema-based SHA determinization may be sped-up exponentially by accessible determinization. The objective of the present chapter is to apply schema-based determinization for the SHAs for the 78 regular XPATH queries in the subcorpus we selected from Lick and Schmitz' benchmark.

For instance, we can compute the schema-based determinization of the SHA of the query Q_{01705} given above in few seconds. The resulting dSHA has 172 states and size 746. Its minimization has only 19 states and overall size 113. It is given in Figure 6.1.

The largest dSHA we obtain for the whole subcollection of 78 regular XPATH queries by schema-based determinization followed by minimization has 58 states. In average, there are 22 states and 71 transition rules per automaton.

The whole automata collection obtained for the subcorpus of 78 regular XPATH queries of Lick and Schmitz' benchmark is published in the software heritage archive at https://archive.softwareheritage.org/browse/origin/?origin_url=https://gitlab.inria.fr/aalserha/xpath-benchmark.

Outline

We present our selection of regular XPATH benchmark queries from the corpus of Lick and Schmitz [Lick & Schmitz 2022] in Section 6.2. A dSHA defining the schema of hedge encodings of valid XML documents is given in Section 6.3. In

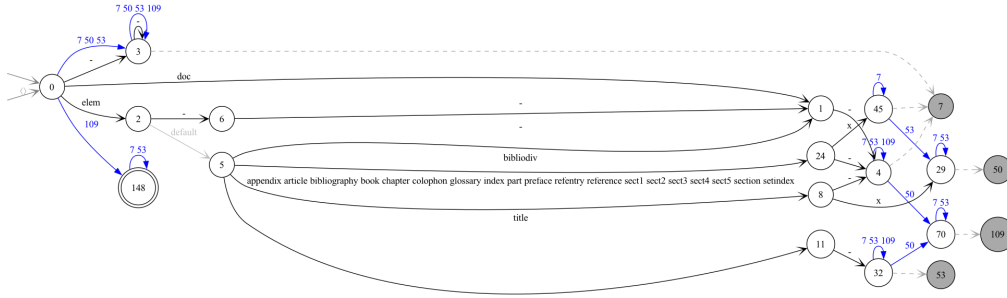


Figure 6.1: The minimization of the schema-based determinization of the SHA compiled from the XPATH query of the Lick and Schmitz' benchmark:

$Q_{01705} = \text{book} \mid \text{article} \mid \text{part} \mid \text{reference} \mid \text{preface} \mid \text{chapter} \mid \text{bibliography} \mid \text{appendix} \mid \dots$

Section 6.4 we discuss our compiler from XPATH expressions to deterministic automata, and illustrate it by example automata from our benchmark collection. In Section 6.5 we discuss how we tested our automata for correctness compared to SAXON's XSLT. The sizes of automata in our regular subset of Lick and Schmitz' benchmark and also for the XPATHMARK benchmark are discussed in Section 6.6. In Section 6.7 we present example automata for some queries of Lick and Schmitz' benchmark.

6.2 Subcorpus of Lick and Schmitz' Benchmark

We start with the collection of 21000 XPATH queries that Lick and Schmitz [Lick & Schmitz 2022] extracted from real-world XQUERY and XSLT programs available on the Web. The purpose of this corpus is to reflect the form and distribution of XPATH queries in practical applications. The much smaller XPATHMARK benchmark [Franceschet 2005b], in contrast, focuses on functional and performance testing.

We then filter the subclass of around 4500 forward navigational XPATH queries of Lick and Schmitz' corpus. The other queries contain comparisons of data values, arithmetics, and functions, including higher-order functions to iterate over sequences, which may be non-regular. We also removed boolean queries and kept only node selection queries. We then selected the 180 largest queries of this subcorpus.

Finally, we removed duplicates of queries up to renaming of XML names-

Id	XPATH Query
18330	/descendant-or-self::node()/child::parts-of-speech
17914	/descendant-or-self::node()/child::tei:back/descendant-or-self::node() /child::tei:interpGrp
10745	*//tei:imprint/tei:date[@type='access']
02091	* ./refentry
00744	./@id ./@xml:id
12060	./attDef
02762	./authorgroup/author ./author
06027	./authorinitials ./author
02909	./bibliomisc[@role='serie']
06415	./email address/otheraddr/ulink

Table 6.1: Some of the 78 queries of the benchmark collection (see Table 6.2).

pace prefixes and local names, and syntactical details, such as `./author` or `descendant-or-self::author` or `descendant-or-self::corppauthor`. This leads us to the collection of 78¹ queries in Table 6.2 of which 10 typical queries are shown in Table 6.1, except that they are rather small.

We note that the XPATH query 18339 is considered as large since it contains the recursive axis `descendant-or-self`. Other queries are considered as large since having a parse tree with more than 15 nodes, for instance 05684.

Table 6.2: The 78 regular XPATH queries selected from Lick and Schmitz' benchmark.

Query ID	XPath query
00744	./@id ./@xml:id
01705	book article part reference preface chapter bibliography appendix glossary section sect1 sect2 sect3 sect4 sect5 refentry colophon bibliodiv[title] setindex index
01847	set book part preface chapter appendix article reference refentry book/glossary article/glossary part/glossary bibliog- raphy colophon
02000	chapter appendix epigraph warning preface index colophon glossary biblioentry bibliography dedication sidebar footnote glossterm glossdef bridgehead part

¹Originally, our filtering scheme selected 79 queries. However, we excluded the query 13896 from the list (`///HEADER//IDNO[@TYPE='evans citation']`). The blank symbol in the data value caused our compiler to fail and not obtain the SHA representing the query.

02086	book article topic part reference preface chapter bibliography appendix glossary section sect1 sect2 sect3 sect4 sect5 refentry colophon bibliodiv[title] setindex index
02091	* ../refentry
02194	//annotation
02697	chapter appendix preface reference refentry article topic index glossary bibliography
02762	../authorgroup/author ../author
02909	../bibliomisc[@role='serie']
03257	../equation[title or info/title]
03325	set book part reference preface chapter appendix article topic glossary bibliography index setindex refentry sect1 sect2 sect3 sect4 sect5 section
03407	set book part reference preface chapter appendix article glossary bibliography index setindex refentry sect1 sect2 sect3 sect4 sect5 section
03410	set book part reference preface chapter appendix article topic glossary bibliography index setindex refentry refsynopsisdiv refsect1 refsect2 refsect3 refsection sect1 sect2 sect3 sect4 sect5 section
03864	guibutton guiicon guilabel guimenu guimenuitem guisubmenu interface
04245	set book part reference preface chapter appendix article glossary bibliography index setindex refentry refsynopsisdiv refsect1 refsect2 refsect3 refsection sect1 sect2 sect3 sect4 sect5 section
04267	descendant::label
04338	refsynopsisdiv/title refsection/title refsect1/title refsect2/title refsect3/title refsynopsisdiv/info/title refsection/info/title refsect1/info/title refsect2/info/title refsect3/info/title
04358	section/title simplesect/title sect1/title sect2/title sect3/title sect4/title sect5/title section/info/title simplesect/info/title sect1/info/title sect2/info/title sect3/info/title sect4/info/title sect5/info/title section/sectioninfo/title sect1/sect1info/title sect2/sect2info/title sect3/sect3info/title sect4/sect4info/title sect5/sect5info/title

04953	set book part reference preface chapter appendix article glossary bibliography index setindex topic refentry refsynopsisdiv refsect1 refsect2 refsect3 refsection sect1 sect2 sect3 sect4 sect5 section
05122	./procedure[title]
05219	set book part preface chapter appendix article topic reference refentry book/glossary article/glossary part/glossary book/bibliography article/bibliography part/bibliography colophon
05226	set book part preface chapter appendix article topic reference refentry sect1 sect2 sect3 sect4 sect5 section book/glossary article/glossary part/glossary book/bibliography article/bibliography part/bibliography colophon
05460	./table//footnote ./informaltable//footnote
05463	table//footnote informaltable//footnote
05684	@abbr @align @axis @bgcolor @border @cellpadding @cellspacing @char @charoff @class @dir @frame @headers @height @id @lang @nowrap @onclick @ondblclick @onkeydown @onkeypress @onkeyup @onmousedown @onmousemove @onmouseout @onmouseover @onmouseup @rules @scope @style @summary @title @valign @width @xml:id @xml:lang
05735	//glossary[@role='auto']
05824	descendant-or-self:*
06027	./authorinitials ./author
06169	article preface chapter appendix refentry section sect1 glossary bibliography
06176	//set //book //part //reference //preface //chapter //appendix //article //colophon //refentry //section //sect1 //sect2 //sect3 //sect4 //sect5 //indexterm //glossary //bibliography //*[@id]
06415	./email address/otheraddr/ulink
06458	info refentryinfo referenceinfo refsynopsisdivinfo refsectioninfo refsect1info refsect2info refsect3info setinfo bookinfo articleinfo chapterinfo sectioninfo sect1info sect2info sect3info sect4info sect5info partinfo prefaceinfo appendixinfo docinfo

06512	//refentry//text()
06639	./tgroup//footnote
06726	//doc:table //doc:informaltable
06794	articleinfo chapterinfo bookinfo doc:info doc:articleinfo doc:chapterinfo doc:bookinfo
06808	personname surname firstname honorific lineage other-name contrib doc:personname doc:surname doc:firstname doc:honorific doc:lineage doc:othername doc:contrib
06856	imageobject imageobjectco audioobject videoobject doc:imageobject doc:imageobjectco doc:audioobject doc:videoobject
06924	authorblurb formalpara legalnotice note caution warning important tip doc:authorblurb doc:formalpara doc:legalnotice doc:note doc:caution doc:warning doc:important doc:tip
06947	anchor areaset audiodata audioobject beginpage constraint indexterm itemset keywordset msg doc:anchor doc:areaset doc:audiodata doc:audioobject doc:beginpage doc:constraint doc:indexterm doc:itemset doc:keywordset doc:msg
07106	dbk:appendix dbk:article dbk:book dbk:chapter dbk:part dbk:preface dbk:section dbk:sect1 dbk:sect2 dbk:sect3 dbk:sect4 dbk:sect5
07113	following-sibling::*[self::dbk:appendix self::dbk:article self::dbk:book self::dbk:chapter self::dbk:part self::dbk:preface self::dbk:section self::dbk:sect1 self::dbk:sect2 self::dbk:sect3 self::dbk:sect4 self::dbk:sect5] following-sibling::dbk:para[@rnd:style = 'bibliography' or @rnd:style = 'bibliography-title' or @rnd:style = 'glossary' or @rnd:style = 'glossary-title' or @rnd:style = 'qandaset' or @rnd:style = 'qandaset-title']
08632	tei:front//tei:titlePart/tei:title
09123	tei:content//rng:ref[@name = 'macro.anyXML']
09138	./rng:ref ./tei:elementRef ./tei:classRef ./tei:macroRef ./tei:dataRef
10337	./tei:note[@place='end']
10745	*//tei:imprint/tei:date[@type='access']

11160	html:table html:tr html:thead html:tbody html:td html:th html:caption html:li
11227	/tei:TEI/tei:text//tei:note[@type='action']
11368	descendant-or-self::tei:TEI/tei:text/tei:back
11478	//xhtml:p[@class]
11780	//tei:ref[@type='cite'] //tei:ptr[@type='cite']
11958	biblStruct//note
12060	./attDef
12404	./tei:dataRef[@name]
12514	tei:content/tei:classRef tei:content//tei:sequence/tei:classRef
12539	//tei:elementSpec //tei:classSpec[@type='atts']
12960	tei:classSpec/tei:attList//tei:attDef/tei:datatype/rng:ref
12961	tei:classSpec/tei:attList//tei:attDef/tei:datatype/tei:dataRef
12962	tei:macroSpec/tei:content//rng:ref
12964	tei:dataSpec/tei:content//tei:dataRef
13632	self::placeName self::persName self::district self::settlement self::region self::country self::bloc
13640	//equiv[@filter]
13710	persName orgName addName nameLink roleName forename surname genName country placeName geogName
13804	//GAP/@DISP
14183	content//rng:ref
14340	//*
15461	h:table[descendant::h:span[@data-type='footnote']]
15462	descendant::h:span[@data-type='footnote']
15484	h:pre[@data-type='programlisting']//text()
15524	//h:section[@data-type='titlepage']
15539	//h:figure[@data-type='cover']//h:img[@src != ""]
15766	//h:body/h:section[@data-type='titlepage']
15809	//h:html[@lang != ""] //h:body[@lang != ""]
15848	descendant::*[@class='refname']
17914	/ descendant-or-self::node()/child::tei:back/descendant-or- self::node()/child::tei:interpGrp
18330	/ descendant-or-self::node()/child::parts-of-speech

6.3 A Schema for XML Documents

The most frequent type of XPATH queries select nodes of XML documents. For referring to selected nodes, we fix a single selection variable x . We call an XML document or subdocument, in which a single node is annotated by x , an *x-annotated example*. An x -annotated example is called *positive* for a query if the query selects the x -annotated node in the XML document, and *negative* otherwise.

The dSHA *xml-seq&one-x* is a schema for x -annotated XML documents (See Fig. 4.5) that recognizes the set of all x -annotated examples. These must satisfy the XML data model and contain exactly one occurrence of x .

The automaton starts in hedge state 0 where it expects to read a nested word $\langle w \rangle$, that can be evaluated to tree state 28, in order to go to the final state 29, where it accepts. The sequence of children w of the tree must be evaluated from the tree initial state, which is equally the hedge state 0. If w starts with letter `doc` indicating an XML document node at the root, the automaton moves from state 0 to state 5. There it may either read the variable x and go to state 5, where it expects a subtree in state 21, i.e. an XML element of which no node is annotated by x . Or it may read the symbol $\neg x$ and move to state 6, where it expects a subtree in state 19, i.e. an XML element of which exactly one node is annotated by x . In both cases it can go to the hedge state 26 and from there to the tree state 28. The automaton also states the relationships of elements, attributes, text and comment nodes according to the XML data model.

The alphabets of names and namespaces of XML documents are infinite. In order to represent infinite sets of transition rule symbolically in a finite manner, the automaton use typed else rules. The typed else rule in state 3, for instance, is labeled by `-namespace`, permitting to read any namespace and to go to state 9. State 9 in turn has an else rule labeled by `-name` which permits to read any (local) name and move to state 13.

6.4 Compiler to Automata

We extended on the compilation chain for regular XPATH queries to automata from [Niehren & Sakho 2021]. As a running example, we consider the following query:

$$Q_{15809} : \quad h:body[@lang \neq ""]$$

Query $Q_{15809'}$ selects a node if it has a child named `body` in namespace `h`, that has the attribute node named `lang` containing a nonempty text.

6.4.1 Parser

Our parser for XPATH expressions computes a parse tree following the grammar of XPATH 3.1 from the W3C. In addition, it returns for any forward regular XPATH expression a logical formula in the language FXP [Debarbieux *et al.* 2015]. For the XPATH example $Q_{15809'}$, we obtain the following FXP formula:

$$\begin{aligned} &child(lab_{elem:type} \wedge lab_{h:namespace} \wedge lab_{body:name} \wedge lab_{x:var} \wedge \\ &\quad child(lab_{att:type} \wedge cand_{default:namespace} \wedge lab_{lang:name} \wedge string \neq "")) \end{aligned}$$

Our previous parser needed considerable improvement in order to be able to cover the large variety of queries from the corpus of Lick and Schmitz [Lick & Schmitz 2022].

6.4.2 Nested Regular Expressions

We next compile FXP formulas to nested regular expressions, which extend on standard regular expressions from words to nested words. Again, considerable work was needed to enable a sufficiently large coverage. For the query $Q_{15809'}$ our compiler yields the nested regular expression:

$$\begin{aligned} &\langle (elem:type._) + doc:type)._.\top. \\ &\quad \langle elem:type.h:namespace.body:name.x:var. \\ &\quad \langle att:type.default:namespace.lang:name._._char._char)^*).\top).\top).\top \end{aligned}$$

Note that the test for a nonempty string got translated by the regular expression $_char._char)^*$. It should also be noticed that this expression matches some x -annotated nested words, that are not x -annotated examples, i.e. not belonging to the language $\mathcal{L}(xml-seq\&one-x)$ of the schema. This is since the nested subwords matching universal expression \top are completely unconstrained.

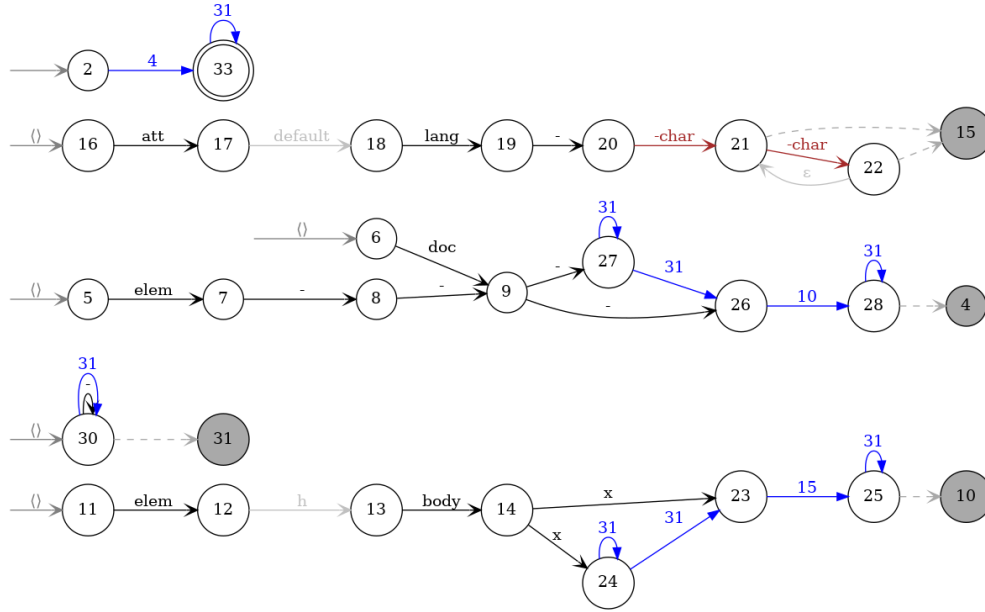


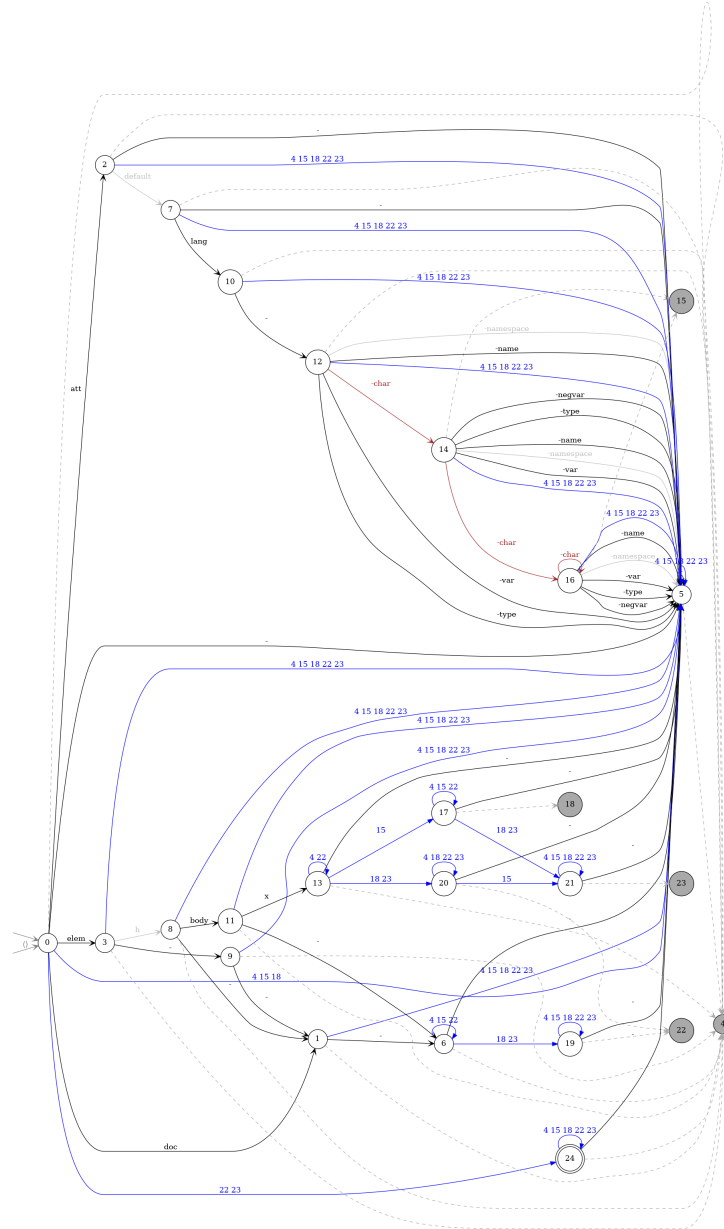
Figure 6.2: The nondeterministic 2-sorted SHA $A_{15809'} = sha(Q_{15809'})$.

6.4.3 Compiler to SHAs

The compiler then converts nested regular expressions into 2-sorted SHAs. This is done by extending a usual compiler from regular expressions to NFAs. The interaction of recursion and nesting leads to some nasty issues, that are discussed and resolved in [Niehren & Sakho 2021]. For developing the present benchmark, we needed to add a treatment of typed wildcards such as `-char`. This is done by introducing typed else rules. For the query $Q_{15809'}$ we obtain The nondeterministic 2-sorted SHA $A_{15809'} = sha(Q_{15809'})$ in Fig. 6.2. Similarly to the nested regular expression, this 2-sorted SHA may recognize some annotated nested words, that are not x -annotated examples, i.e., that do not belong to the language of the schema $L(xml-seq\&one-x)$.

6.4.3.1 Accessible Determinization

The accessible determinization $det(A_{15809'})$ of the SHA of the query $Q_{15809'}$ yields an dSHA with 25 states and 183 transition rules given in Figure 6.3. This is much much larger than one might expect. Even worse, in some cases, the accessible determinization does not finish after some hours.

Figure 6.3: The accessible determinization $\det(A_{15809})$.

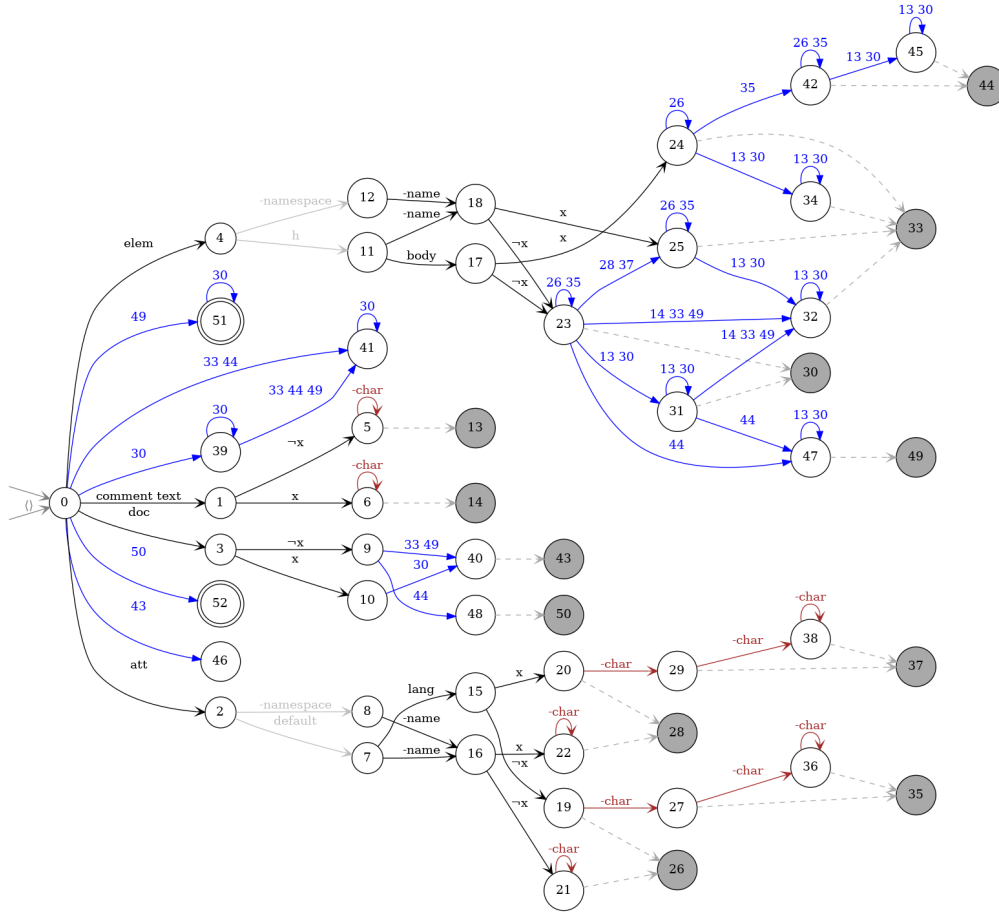


Figure 6.4: The determinization of the schema product $\det(A_{15809'} \times \text{xml-seq\&one-x})$.

6.4.3.2 Determinizing the Schema Product

Accessible determinization applied to the product of the queries' automaton and the schema xml-seq\&one-x permits to compute deterministic SHAs for all queries of our benchmark within a timeout of 100 seconds. The result for $Q_{15809'}$ is a dSHA with 53 states and 110 transition rules, see automaton $\det(A_{15809'} \times \text{xml-seq\&one-x})$ in Figure 6.4. The overall size is smaller, and the automaton is much easier to understand, but the number of states increased.

6.4.4 Schema-Based Determinization

Schema-based determinization as proposed in [Niehren *et al.* 2022a] improves the situation further. For query $Q_{15809'}$ it yields the 2-sorted SHA in the schema-based determinization $\det_S(A_{15809'})$ where $S = \text{xml-seq\&one-x}$ in Fig. 6.5, which has only

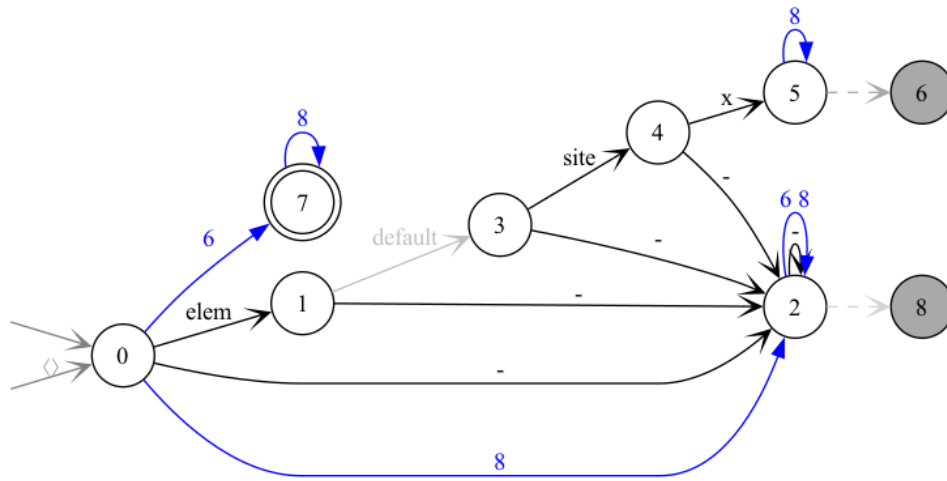


Figure 6.6: The minimization of the schema-based determinization of the SHA for the XPATH query `self :: site`.

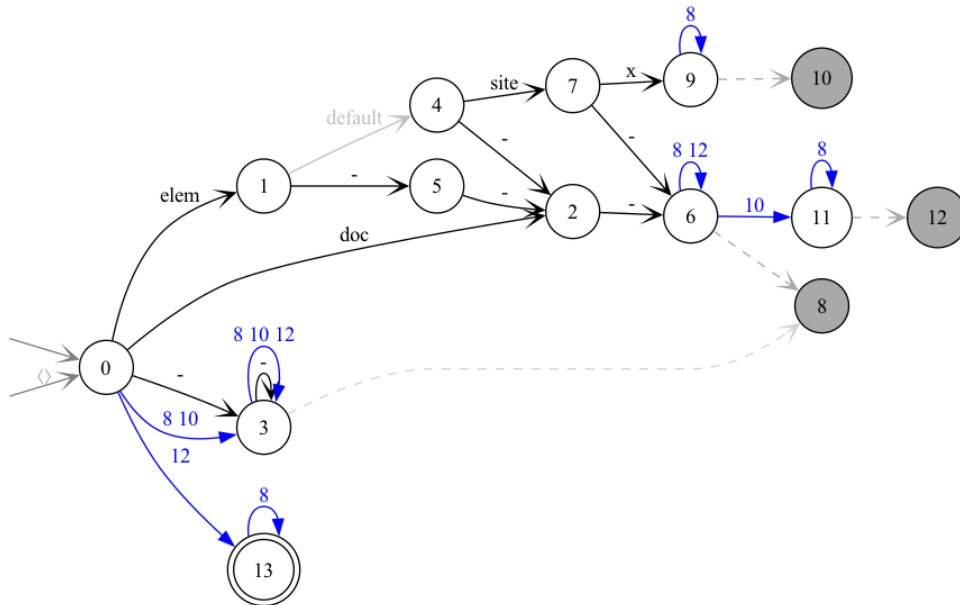


Figure 6.7: The minimization of the schema-based determinization of the SHA for the XPATH query `A0=child::site`.

6.5 Testing Automata

The natural manner to test an automaton for some $XPATH$ query, is to apply some query evaluator to the automaton with some document and to compare the result to an $XPATH$ evaluator. When we started, however, we were lacking any sufficiently efficient query evaluator for $dSHAS$. Therefore, we started testing $dSHAS$ for queries differently, by testing language membership for x -annotated documents. Only when *ASTREAM* 1.0 became available, we could move to testing $dSHAS$ for queries by using query evaluation.

We note in particular, that Sebastian’s *QUIXPath* [Sebastian 2016, Debarbieux *et al.* 2015] could not be used, even though any SHA A can be compiled to the NWA A^{nwa} while preserving determinism and the size of the automaton. The problem is that *QUIXPath* requires an early NWA as input, i.e., an NWA and subset of early selection and rejection states. Without these sets, the NWA query cannot be evaluated with reasonable efficiency. Sebastian computed these sets during the compilation of $XPATH$ queries to early NWAs. This cannot be done though, when compiling $XPATH$ queries to $SHAS$, since these cannot pass information top-down, so that appropriate sets of selection and rejection states may not be available. This is in contrast to earliest downward $SHAS$ (see Chapter 7), that can be compiled to early NWAs with optimal selection and rejection states. But the earliest $dSHA^\downarrow$ s may be of exponential size, so require their on-the-fly generation during query evaluation for a given document, which is not supported by *QUIXPath*.

6.5.1 Membership for Samples

For each regular $XPATH$ query and each XML document, we created a sample of x -annotated examples. An x -annotated example is marked positively, if and only if it belongs to the language of the query, and negatively otherwise.

The first problem was that Lick and Schmitz’ corpus does not come with XML documents, corresponding to the queries harvested in the *XSLT* and *XQUERY* programs *docbook*, *htmlbook*, *teixsl*, *treedown*, and *histei*. So we had to create XML documents ourselves that satisfy the schemas of these applications, ensuring that the queries select some nodes there.

We computed the Boolean markings for the x -annotated example using *SAXON* *XSLT* evaluator. For this, we compiled each $XPATH$ query Q into some *XSLT* program so that when *SAXON* applied to some XML document d , it returns $Q(d)$. We exported

query answers in Dewey notation, similarly to how nodes are returned by Schematron: The Dewey notation of a node is its relative address from the root, i.e., by the list of child steps leading to the node. Such lists can be easily encoded in XML format.

We can then test the correctness of automata A for a given query Q on the sample by using dSHA membership tester. Suppose that h is the hedge encoding the XML document d under consideration. It is sufficient to verify for all x -annotated examples $e = h*[x/\pi]$ of the sample, that the query evaluator $e \in \mathcal{L}(A)$ if and only if, e was marked positively. The latter is equivalent to SAXON confirming $\pi \in Q(d)$.

By testing dSHAs on samples, we could fix various problems that arose on the way to our final collection. Currently, no test failures are remaining, except for the query 13896 below:

```
//HEADER//IDNO[@TYPE='evans citation']
```

The problem here is raised by the blank symbol in the attribute value 'evans citation'. For now, we simply removed this example from our tests.

We omit the details of how we created the samples for each query. We just note that we had to limit the number of marked x -annotated examples, since otherwise, the samples would contain one x -annotated example for each node of the XML document, and thus the overall sample size would have been of quadratic in $O(|h|^2)$.

When having large samples, the testing method becomes time consuming, while when limiting the sample sizes, the testing method becomes even more incomplete. At the end, we considered no more than 10 x -annotated examples per query and document, while choosing positively marked x -annotated examples with priority.

6.5.2 Query Evaluation

Once ASTREAM 1.0 was developed we could apply it to evaluate XPATH queries defined by dSHAs on XML streams, and compare its answer set to the answer set computed by SAXON (after conversion of the XPATH query to some XSLT program).

This testing method again remains incomplete, since restricted to a finite number of XML documents. But when some dSHA A was considered as convincing for defining a query with schema S , then any other dSHA B can be tested for schema-equivalence:

$$\mathcal{L}(A) \cap S = \mathcal{L}(B) \cap S$$

Under the assumption that A defines the target query correctly with respect to the

schema, B does so too if and only if schema-equivalence holds.

6.6 Automata Statistics

We present statistics for the sizes of dSHAs for the regular XPATH queries in our benchmarks.

6.6.1 XPATHMARK Benchmark

Minimal dSHAs for the regular XPATH queries in the XPATHMARK benchmark were presented already in Section 4.4.1. Statistics on the sizes of the automata obtained with and without schema-based determinization are given in column C of Table 6.3.

Table 6.3: Statistics of the SHAs for the regular forward XPATH queries in the XPATHMARK benchmark in Table 4.1. For each SHA we present its size and the number of states in the format size(#states).

query ID	A	$det(A)$	$B = det(A \times S)$	$C = det_S(A)$	$C' = mini(C)$	$B' = mini(B)$
site	50 (19)	114 (16)	121 (38)	51 (14)	51 (14)	121 (38)
A1	152 (63)	399 (37)	393 (79)	172 (37)	172 (37)	348 (66)
A2	137 (57)	597 (53)	210 (59)	122 (34)	71 (20)	138 (40)
A3	133 (55)	484 (46)	279 (69)	138 (36)	111 (28)	213 (51)
A4	181 (74)	486 (42)	724 (95)	253 (42)	259 (42)	484 (72)
A5	201 (80)	690 (57)	860 (116)	381 (57)	180 (37)	314 (61)
A6	260 (99)	547 (45)	812 (98)	297 (45)	304 (45)	529 (74)
A7	178 (71)	468 (41)	523 (85)	222 (41)	138 (30)	276 (57)
A8	923 (279)	2464 (124)	2302 (202)	1261 (118)	294 (48)	492 (77)

One can see, that schema-based determinization does not behave much better than accessible determinization on this benchmark. Still, the overall size of the dSHAs in column $C = det_S(A)$ compared to accessible determinization in column $det(A)$ is reduced by a factor of around 2. The number of states, however, does not change much.

6.6.2 Lick and Schmitz' Benchmark

We compiled all of our 78 XPATH queries to deterministic automata using the compilation chain described in Section 6.4. Here we present the statistics of the benchmark automata that we obtained. The summary is given in Table 6.4. We show for each automaton two numbers $\text{size}(\#states)$ where size is the overall size of the automaton and $\#states$ the number of its states.

The nondeterministic SHAs compiled from the nested regular expressions was cleaned using the schema *xml-seq&one-x*: The dSHA *xml-seq&one-x*: a schema for x -annotated XML documents in Figure 4.5. The result is called $A = sha(Q)$ leading to the statistics in the second column of in Table 6.4.

We note that 37% of the 2-sorted SHAs original stepwise hedge automata for the queries $A = sha(Q)$ have more than 100 states, so they are sometimes bigger than one might expect. The biggest is for query 06176 with 630 states and an overall size of 1391. The reason is that this query is selecting a union of 20 subqueries, all with descendant-or-self axis. For each subquery, we have 4 constructs of respective state sizes: 2, 6, 10 and 13, making a subtotal of $31 * 20 = 620$. With an additional 8 states for one subquery that select all descendants with an attribute named *id* and another 2 for reading any tree, we end up with our total 630 states.

Table 6.4: Experiment results on the XPATH subcorpus from Lick and Schmitz in Table 6.2. For each automaton we present: $\text{size}(\#states)$.

query Q of id	$A =$ $SHA(Q)$	$det(A)$	$B =$ $det(A \times S)$	$C =$ $det_S(A)$	$B' =$ $mini(B)$	$C' =$ $mini(C)$
00744	109 (46)	335 (37)	169 (54)	80 (24)	128 (41)	54 (15)
01705	772 (350)		1308 (279)	746 (172)	221 (48)	113 (19)
01847	559 (252)		1013 (223)	543 (137)	194 (48)	92 (19)
02000	642 (291)		723 (201)	387 (110)	163 (41)	83 (14)
02086	809 (367)		1366 (291)	781 (180)	223 (48)	115 (19)
02091	100 (42)	555 (45)	182 (57)	81 (24)	146 (44)	61 (17)
02194	81 (33)	253 (31)	135 (42)	66 (20)	119 (38)	53 (16)
02697	383 (172)		464 (131)	240 (68)	149 (41)	69 (14)
02762	121 (50)	564 (53)	222 (63)	97 (28)	123 (39)	46 (12)
02909	96 (38)	311 (36)	213 (62)	100 (27)	167 (49)	91 (24)
03257	130 (53)	1310 (92)	445 (85)	224 (46)	210 (49)	87 (20)
03325	753 (342)		834 (231)	450 (128)	169 (41)	89 (14)
03407	716 (325)		797 (221)	429 (122)	167 (41)	87 (14)

03410	938 (427)		1019 (281)	555 (158)	179 (41)	99 (14)
03864	272 (121)		353 (101)	177 (50)	143 (41)	63 (14)
04245	901 (410)		982 (271)	534 (152)	177 (41)	97 (14)
04267	87 (34)	146 (20)	137 (43)	54 (15)	131 (41)	51 (14)
04338	470 (206)		1066 (207)	563 (135)	213 (51)	95 (22)
04358	1006 (444)		3580 (559)	2021 (433)	757 (99)	345 (58)
04953	938 (427)		1019 (281)	555 (158)	179 (41)	99 (14)
05122	83 (33)	292 (33)	221 (55)	92 (23)	161 (44)	63 (16)
05219	698 (315)		1192 (260)	651 (164)	196 (48)	94 (19)
05226	920 (417)		1558 (338)	867 (218)	208 (48)	106 (19)
05460	232 (98)	3468 (174)	509 (127)	269 (77)	156 (44)	62 (16)
05463	204 (86)	1180 (70)	332 (77)	152 (38)	180 (48)	78 (20)
05684	1348 (616)		1068 (284)	719 (226)	193 (39)	124 (16)
05735	111 (45)	412 (44)	201 (58)	106 (30)	161 (47)	96 (27)
05824	62 (25)	150 (21)	130 (42)	50 (15)	112 (37)	38 (11)
06027	115 (48)	1101 (79)	184 (57)	82 (24)	123 (39)	46 (12)
06169	346 (155)		427 (121)	219 (62)	147 (41)	67 (14)
06176	1391 (630)		1661 (448)	1203 (386)	176 (43)	113 (23)
06415	139 (58)	1793 (93)	300 (74)	135 (36)	229 (55)	101 (25)
06458	827 (376)		908 (251)	492 (140)	173 (41)	93 (14)
06512	135 (56)	583 (58)	218 (60)	117 (35)	152 (43)	77 (23)
06639	123 (50)	516 (49)	237 (65)	106 (30)	154 (44)	60 (16)
06726	149 (64)	2806 (149)	176 (53)	97 (30)	121 (38)	55 (16)
06794	270 (121)		354 (102)	178 (51)	144 (42)	64 (15)
06808	525 (240)		609 (172)	321 (93)	145 (41)	65 (14)
06856	306 (138)		390 (112)	198 (57)	139 (41)	59 (14)
06924	598 (274)		682 (192)	362 (105)	147 (41)	67 (14)
06947	744 (342)		828 (232)	444 (129)	151 (41)	71 (14)
07106	457 (206)		538 (151)	282 (80)	153 (41)	73 (14)
07113	695 (296)		2409 (456)	1527 (302)	311 (73)	229 (48)
08632	128 (52)	629 (51)	277 (67)	120 (31)	219 (53)	96 (24)
09123	164 (64)	705 (59)	358 (90)	175 (43)	265 (66)	164 (40)
09138	269 (117)		323 (97)	164 (49)	133 (40)	56 (13)
10337	92 (36)	291 (34)	197 (58)	92 (25)	159 (47)	83 (22)
10745	187 (76)	939 (68)	275 (72)	141 (38)	218 (57)	130 (34)
11160	309 (138)		390 (111)	198 (56)	145 (41)	65 (14)

11227	153 (62)	583 (53)	334 (81)	163 (42)	244 (59)	144 (37)
11368	102 (41)	458 (44)	247 (62)	104 (28)	191 (49)	78 (20)
11478	101 (41)	365 (40)	166 (50)	87 (26)	141 (43)	77 (23)
11780	205 (88)	3832 (190)	254 (71)	143 (41)	164 (47)	99 (27)
11958	109 (44)	348 (35)	213 (57)	90 (24)	178 (48)	76 (20)
12060	64 (25)	162 (22)	139 (44)	56 (16)	121 (39)	44 (12)
12404	84 (33)	258 (31)	170 (52)	77 (22)	143 (44)	68 (19)
12514	182 (77)	2734 (112)	320 (77)	146 (38)	247 (57)	114 (28)
12539	179 (76)	3479 (174)	243 (69)	138 (40)	166 (48)	101 (28)
12960	167 (68)	1340 (81)	421 (88)	190 (46)	317 (64)	146 (33)
12961	166 (68)	1318 (80)	417 (87)	186 (45)	313 (63)	142 (32)
12962	129 (52)	576 (49)	281 (68)	124 (32)	223 (54)	100 (25)
12964	128 (52)	560 (48)	277 (67)	120 (31)	219 (53)	96 (24)
13632	132 (58)	339 (33)	248 (66)	113 (33)	128 (36)	47 (9)
13640	100 (41)	364 (40)	165 (50)	86 (26)	140 (43)	76 (23)
13710	420 (189)		501 (141)	261 (74)	151 (41)	71 (14)
13804	70 (29)	155 (21)	128 (41)	63 (20)	124 (40)	60 (19)
14183	110 (44)	362 (36)	217 (58)	94 (25)	182 (49)	80 (21)
14340	79 (33)	231 (29)	126 (40)	58 (18)	110 (36)	45 (14)
15461	146 (58)	628 (54)	651 (109)	283 (51)	241 (59)	140 (33)
15462	127 (50)	325 (37)	237 (67)	112 (29)	191 (54)	109 (28)
15484	181 (71)	657 (62)	394 (96)	189 (47)	277 (68)	174 (42)
15524	125 (50)	471 (49)	245 (68)	130 (35)	185 (52)	120 (32)
15539	217 (88)	1709 (121)	402 (98)	213 (58)	228 (57)	144 (38)
15766	144 (58)	669 (60)	300 (77)	155 (41)	219 (57)	135 (35)
15809	197 (84)	3795 (188)	230 (67)	129 (39)	145 (43)	82 (24)
15848	124 (49)	303 (35)	221 (63)	103 (27)	179 (51)	100 (26)
17914	179 (75)	2740 (141)	265 (69)	150 (44)	152 (43)	82 (24)
18330	99 (41)	465 (43)	145 (44)	74 (22)	128 (39)	61 (18)

The column for $\text{det}(A)$ contains the statistics for the determinization of A . No schema is used there. We use a timeout of 100 seconds. Whenever this is not enough, the cell in the table is left blank. Indeed, the determinization fails with this timeout for 37% of the queries of our corpus. Roughly, the determinization fails for all 2-sorted SHAs with more than 100 states. For instance, for query 11780 the 2-sorted SHA A has size 205 (88), while the dSHA $\text{det}(A)$ has size 3832 (190).

The column for $B = \text{det}(A \times S)$ contains the determinization of the product of A and the schema $S = \text{xml-seq\&one-x}$. Even though $A \times S$ is always larger than A , we were able to always determinize $A \times S$ within the timeout, in contrast to A . The largest dSHA B obtained is for query 04358: it has size 3580 (559). This shows that B may still be quite big, but often a big improvement in size over $\text{det}(A)$.

The next column reports on $C = \text{det}_S(A)$ obtained by schema-based determinization with schema $S = \text{xml-seq\&one-x}$. Again, the computation succeeds in all cases within the timeout of 100 seconds. The size of C for query 04358 is 2021 (433), which improves in size over B .

In the next two columns, we respectively minimize the determinized SHAs B and C , using a naive minimization algorithm. All automata can be minimized within the timeout of 100 seconds. We note that $C' = \text{mini}(C)$ is always smaller than $B' = \text{mini}(B)$, showing that schema-based determinization yields smaller minimal automata than determinizing the schema-product. The maximal number of states of the minimal dSHAs $C' = \text{mini}(C)$ is 58 for query 04358. In average the number of states decreases by 55%.

In the last column, we compiled the minimized dSHAs of C' to the dNWA $\text{nwa}(C')$. It has the same number of states as C' for all queries and a minor increase is the number of transitions. All these results, including the automata of the intermediate steps, generated during the whole compilation chain are available at in the software heritage archive at the following url: https://archive.softwareheritage.org/browse/origin/?origin_url=https://gitlab.inria.fr/aalserha/xpath-benchmark.

6.7 Example Automata for Lick and Schmitz' Benchmark

Given that the minimized dSHA for the 78 regular XPATH queries selected from Lick and Schmitz' benchmark are all small, we present some. We have chosen somehow arbitrarily queries Q_{01705} , Q_{17914} , Q_{10745} , Q_{15809} , Q_{02762} , Q_{06027} , Q_{02909} , Q_{06415} , and Q_{15339} in the Figures 6.1, 6.9, 6.10, 6.11, 6.12, 6.13, 6.14, and 6.15.

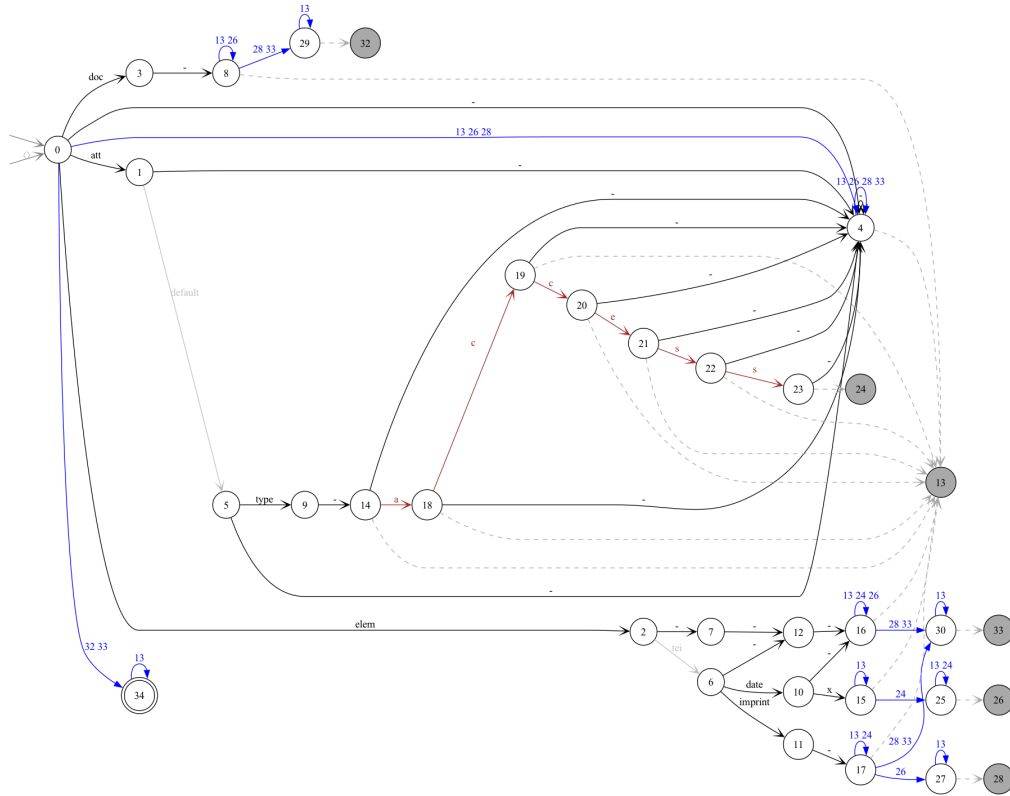


Figure 6.8: The minimization of the schema-based determinization of the SHA compiled from the XPATH query of the Lick and Schmitz' benchmark:

$$Q_{10745} = *//tei:imprint/tei:date[@type='access']$$

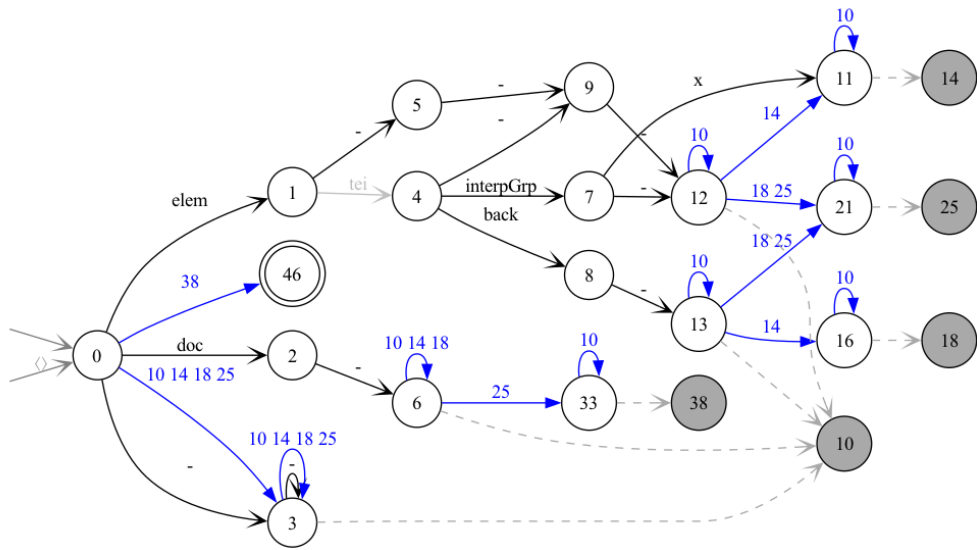


Figure 6.9: The minimization of the schema-based determinization of the SHA compiled from the XPath query of the Lick and Schmitz' benchmark:

$Q_{17914} = /descendant-or-self::node()/child::tei:back/descendant-or-self::node()/child::tei:interpGrp$

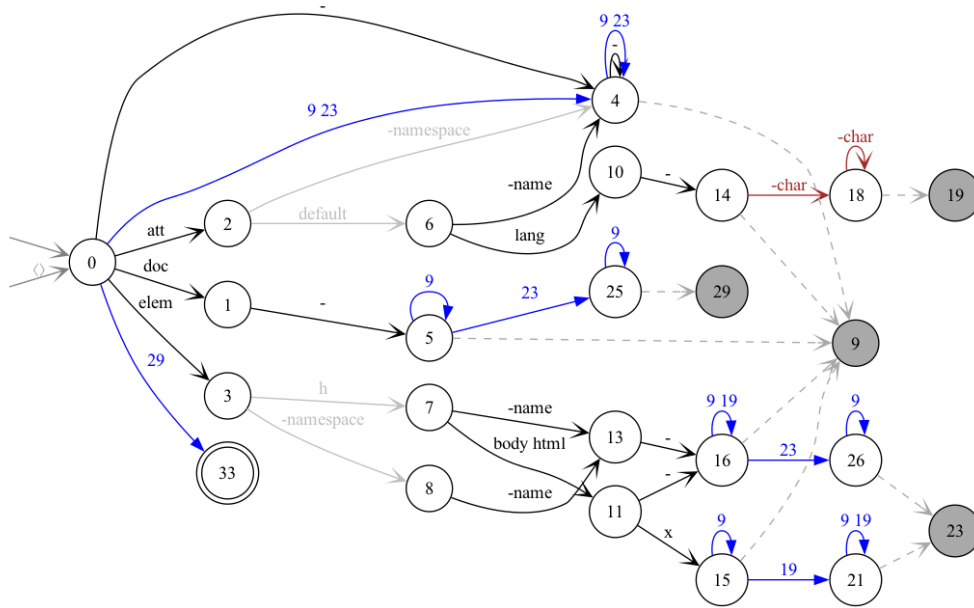


Figure 6.10: The minimization of the schema-based determinization of the SHA compiled from the XPATH query of the Lick and Schmitz' benchmark:

$$Q_{15809} = //h:html[@lang] \mid //h:body[@lang]$$

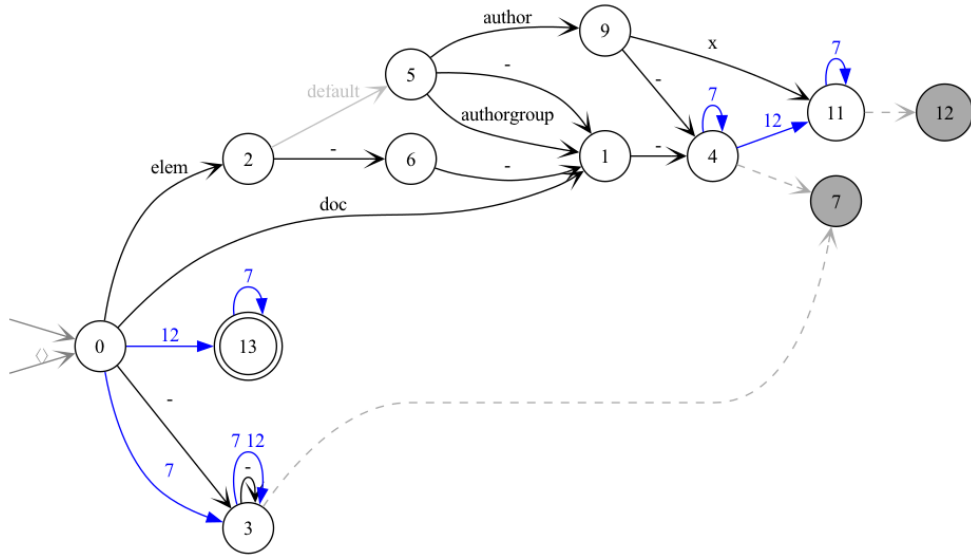


Figure 6.11: The minimization of the schema-based determinization of the SHA compiled from the XPath query of the Lick and Schmitz' benchmark:

$$Q_{02762} = \text{.//authorgroup/author} \mid \text{.//author}$$

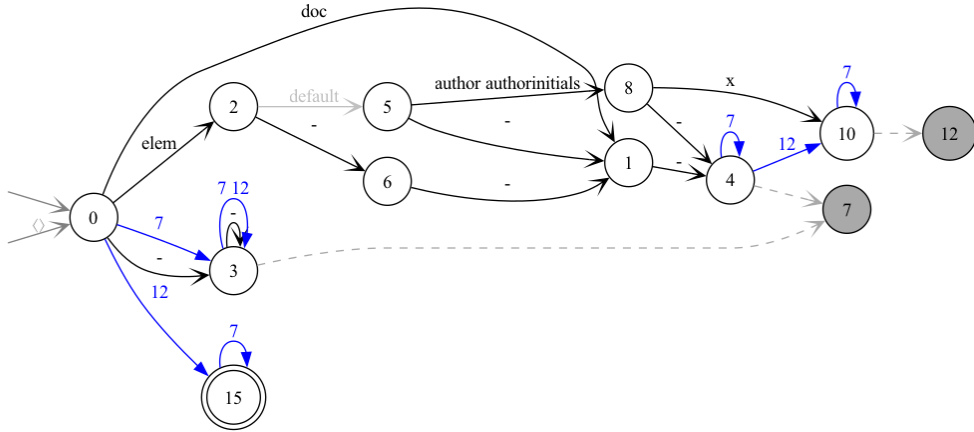


Figure 6.12: The minimization of the schema-based determinization of the SHA compiled from the XPath query of the Lick and Schmitz' benchmark:

$$Q_{06027} = \text{.//authorinitials} \mid \text{.//author}$$

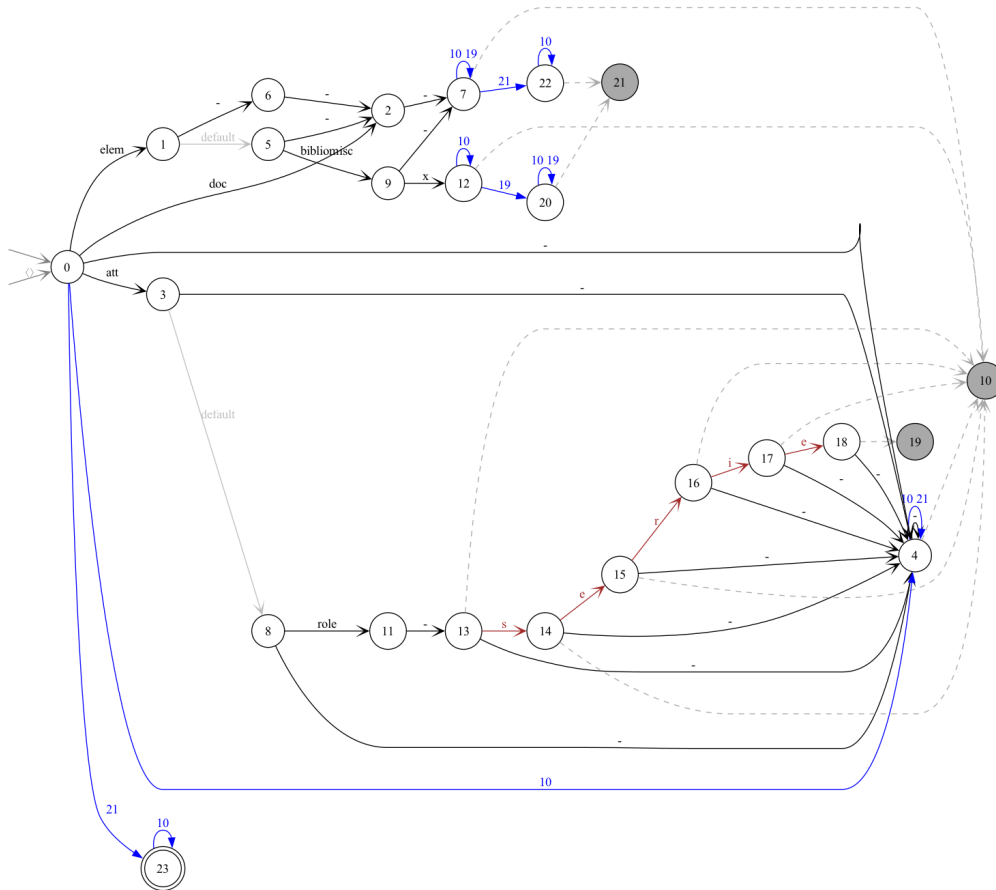


Figure 6.13: The minimization of the schema-based determinization of the SHA compiled from the XPATH query of the Lick and Schmitz' benchmark:

$$Q_{02909} = .//bibliomisc[@role='serie']$$

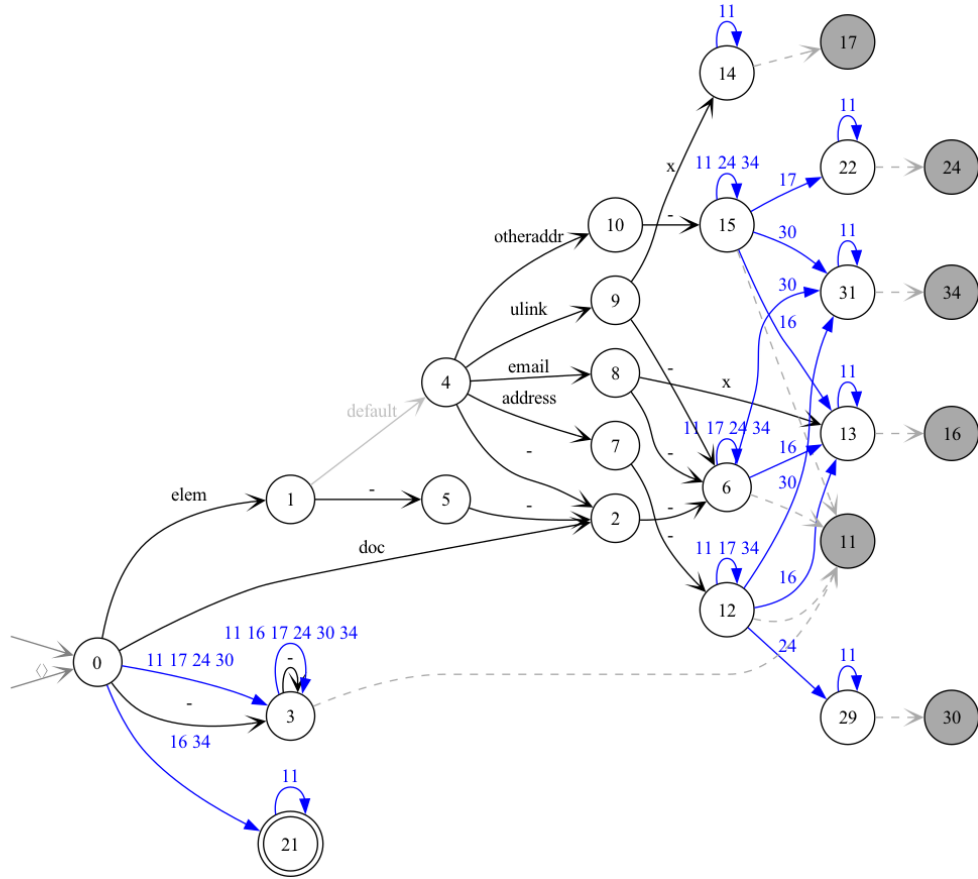


Figure 6.14: The minimization of the schema-based determinization of the SHA compiled from the XPath query of the Lick and Schmitz' benchmark:

$$Q_{06415} = \text{//email} \mid \text{address/otheraddr/ulink}$$

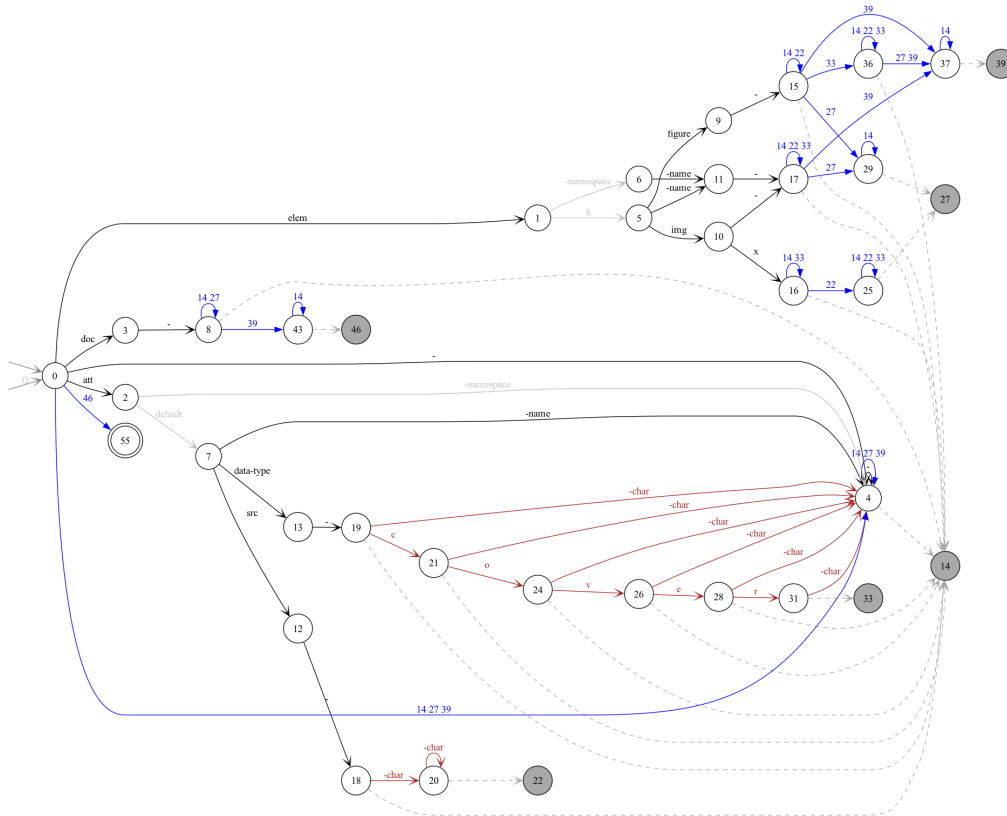


Figure 6.15: The minimization of the schema-based determinization of the SHA compiled from the XPATH query of the Lick and Schmitz benchmark:

$$Q_{15539} = //h:figure[@data-type='cover']//h:img[@src != '']$$

Part III

Projection

Abstract

We show how to evaluate stepwise hedge automata in a top-down manner with complete subhedge and suffix projection. This means that irrelevant subhedge and irrelevant suffixes are projected in a maximal manner. Furthermore, language membership is decided at the earliest prefix when it becomes certain. In order to pass projection information top-down, we base our projection algorithms on compilers from deterministic stepwise hedge automata to $d\text{SHA}^\downarrow$ s with subhedge projection and selection states. The top-down evaluator of this $d\text{SHA}^\downarrow$ can then be run with subhedge projection, either in-memory or in streaming mode.

Contents

7	Complete Subhedge Projection	171
8	Complete Suffix Projection	223
9	Combining Subhedge and Suffix Projection	237
10	Projecting Evaluators for Earliest Membership	245

Introduction

Projection needs to be added to various algorithms on words, trees, hedges, or nested words in order to make them efficient. Intuitively, an algorithm is projecting if it visits only a fragment of the input structure, in the best case, the part

that is relevant to the problem under consideration. The relevance of projection for XML processing was already noticed by [Marian & Siméon 2003, Frisch 2004, Maneth & Nguyen 2010, Sebastian & Niehren 2016]. SAXON’s in-memory evaluator, for instance, projects input XML document relative to an XSLT program, which contains a collection of XPATH queries to be answered simultaneously [Kay 2004]. The QUIXPATH tool [Sebastian & Niehren 2016] evaluates XPATH queries in streaming mode with subtree and descendant projection. Projection during the evaluation JSONPath queries on JSON documents in streaming mode is called fast-forwarding [Gienieczko *et al.* 2024].

Projecting for in-memory algorithms assumes that the full graph of the input hedge has to be constructed. Nevertheless, projection may still save time if one has to match several patterns on the same input hedge, or, if the graph was constructed for different reasons anyway. In streaming mode, the situation is similar: the whole input hedge on the stream needs to be parsed, but the evaluators need to inspect only nodes that are not projected away. Given that pure parsing is by two or three orders of magnitude faster than pattern evaluation, the time gain of projection may be considerable.

In this part, we study subhedge projection and suffix projection for regular hedge languages, as well as their combination. Complete suffix projection is the main objective of earliest membership testing for top-down evaluators, possibly in streaming mode. However, it must be combined with subhedge projection in order to obtain reasonable efficiency.

We also provide a *projecting top-down evaluator* for $d\text{SHA}^\downarrow$. It does the same as the top-down evaluator for SHA, except that it accepts in selection states and projects the current subhedge in subhedge projection states. We provide two versions of the projecting top-down evaluator, of which the first runs in-memory and the second in streaming mode.

Finally, we present earliest top-down membership tester for $d\text{SHAS}$ with complete subhedge and suffix projection. For obtaining this algorithm, we run projecting top-down evaluator on the combination of the earliest $d\text{SHA}$ and the $d\text{SHA}$ with complete subhedge projection for the input $d\text{SHA}$ yields. Again, we have two modes for this evaluator in-memory and streaming.

Outline. In Chapter 7 we start with an algorithm for complete subhedge projection for $d\text{SHAS}$ by compilation to $d\text{SHA}^\downarrow$ s. In the following Chapter 8, we continue with an algorithm for complete suffix projection for $d\text{SHAS}$, again by compilation to

$d\text{SHA}^\downarrow$ s. And then in Chapter 9 we show how to combine both compilers in order to produce an earliest $d\text{SHA}^\downarrow$ with complete subhedge and suffix projection for any $d\text{SHA}$. Projecting evaluators for arbitrary SHA^\downarrow s are given in Chapter 10 and applied too earliest $d\text{SHA}^\downarrow$ s with projection for $d\text{SHAS}$. This leads to an earliest membership tester with complete subhedge and suffix projection for $d\text{SHAS}$.

Complete Subhedge Projection

Abstract

We define top-down evaluators with subhedge projection for stepwise hedge automata. For this, we compile any stepwise hedge automata to some $d\text{SHA}^\downarrow$ s recognizing the same language. The top-down evaluator of the $d\text{SHA}^\downarrow$ passes information top-down that is needed to detect irrelevant subhedges. We show how to obtain complete subhedge projection in this manner, i.e., to project as many irrelevant subhedges as possible during top-down evaluation.

Contents

7.1	Introduction	172
7.2	Definitions and Properties	174
7.2.1	Irrelevant Subhedges	174
7.2.2	Basic Properties	176
7.2.3	Completeness for Subhedge Projection	178
7.3	Safe-No-Change Projection	183
7.3.1	Algorithm	183
7.3.2	Soundness	186
7.3.3	Incompleteness	196
7.4	Congruence Projection	197
7.4.1	Motivation	198
7.4.2	Approach	198
7.4.3	Algorithm	200
7.4.4	Soundness	207

7.4.5	Completeness	217
7.4.6	Automata Sizes	222

7.1 Introduction

We study subhedge projection for the membership problem to a regular hedge language, i.e., whether a given hedge h belongs to a regular hedge language L . A hedge language is sometimes also called a hedge pattern. The membership problem for regular hedge languages is called regular hedge pattern matching.

The starting point of the present chapter is the notion of irrelevant subhedges that we introduce. It states that the positions of the subhedge in a hedge h are irrelevant for the membership to some regular hedge language L with respect to top-down traversal on h . In other words, a position of h is subhedge irrelevant for L , if for all possible continuations beyond the position, the insertion of some subhedge at the position does not affect membership to L .

We contribute an algorithm for hedge pattern matching with complete subhedge projection. Our algorithm decides language membership while maximally projecting irrelevant subhedges away. In other words, it decides whether a hedge h matches a pattern L without visiting any subhedge of h that is located at some position that is subhedge irrelevant with respect to L .

Reconsider the example of the nested regular expression `[self-list-child-item]` mimicking the XPATH filter `[self::list/child::item]`. The nested regular expression is satisfied by a nested list if its root is a `list` element that has some `item` child. The satisfying hedges have the form $\langle \text{list} \cdot h_1 \cdot \langle \text{item} \cdot h_2 \rangle \cdot h_3 \rangle \cdot h_4$ for some hedges h_1, h_2, h_3, h_4 . When evaluating this filter on some hedge, it is sufficient to inspect the first subtree for having the root label `list` and then all its children until one with root label `item` is found. The subhedges of these children, i.e., the hedge h_2 and the subhedges of the top-level trees in h_1 and h_3 , can be projected away, as well as the hedge h_4 . They don't have to be inspected for evaluating this filter. However, one must memoize whether the level of the current node is 0, 1, or greater. This level information can be naturally updated in a top-down manner. The evaluators of SHAS, however, operate bottom-up. Therefore, projecting evaluators for SHAS need to be based on more general machines. We propose *downward step-wise hedge automata* (SHA^\downarrow s), a variant of SHAS that supports top-down processing in addition. They are basically Neumann and Seidl's pushdown forest automata [Neumann & Seidl 1998], except that they apply to unlabeled hedges instead of

labeled forests. Furthermore, Nwas are known to operate in basically the same manner on nested words [Gauwin *et al.* 2008], while allowing for more slightly more general visible pushdowns. We then distinguish subhedge projection states for SHA^\downarrow s, and show how to use them to evaluate SHAs with subhedge projection both in-memory and in streaming mode.

As a first contribution, we present the safe-no-change compiler from SHAs to SHA^\downarrow s that can distinguish appropriate subhedge projection states. The idea is that the safe-no-change SHA^\downarrow can distinguish contexts in which the states of the SHA will safely not change. For instance, the nested regular expression [self-list-child-item] already defined by the deterministic SHA in Figure 3.10, which our compiler maps to the SHA^\downarrow in Figure 3.15. The context information made explicit is about the levels of the states. This permits us to distinguish projection states from level 2 on, and in which subhedges can be ignored. We prove the soundness of our compiler based on a nontrivial invariant that we establish¹. We also present a counter example showing that the safe-no-change projection is not complete for subhedge projection. It shows that a subhedge may be irrelevant even though its state may still be changing.

As a second contribution, we propose the congruence projection algorithm. It again compiles SHAs to SHA^\downarrow s but relies on the congruence relations on automata states. We then prove that congruence projection yields not only a sound algorithm, but that this algorithm is also complete for subhedge projection, i.e., all strongly irrelevant subhedges (see Definition 7.14) are evaluated into some looping state. Congruence projection starts on top-level of hedges with the Myhill-Nerode congruence that is well-known from automata minimization. For languages on words L , this congruence identifies prefixes v that have the same residual language $v^{-1}(L) = \{w \mid v \cdot w \in L\}$. A prefix v is then suffix irrelevant, if its residual language $v^{-1}(L)$ is either universal or empty. In the case of hedges – which extend on words by adding hierarchical nesting via a tree constructor – the congruence must be adapted when moving down into subtrees.

We show that both compilers may increase the size of the automata exponentially, since they must be able to convert bottom-up deterministic automata on monadic trees into top-down deterministic automata. That is the same as inverting deterministic automata on words, which is well known to raise an exponential blow up in the worst case². The exponential explosion can be avoided when interested in the

¹We note that the proof required an adaptation of the original compiler from FCT [Al Serhali & Niehren 2023b].

²For the family of languages $(a+b)^* \cdot a \cdot (a+b)^n$ where $n \in \mathbb{N}$ the minimal deterministic left-to-right

evaluation of hedges by automaton with subhedge projection, by not constructing the complete SHA^\downarrow s statically. Instead, only the needed part of the SHA^\downarrow s may be constructed on the fly when evaluating some hedge.

Our third contribution is a refinement of the two previous compilers so that they can also distinguish safe states for selection at the earliest position. For this, we combine these compilers with a variant of the earliest membership tester of [Al Serhali & Niehren 2023a] that operates in-memory by compiling to SHA^\downarrow s instead of NWAS . Furthermore, membership failure is detected at the earliest position too. In this way, we obtain an earliest in-memory membership tester for deterministic SHAS .

7.2 Definitions and Properties

7.2.1 Irrelevant Subhedges

We define the concept of irrelevant occurrences of subhedges with respect to a given hedge pattern. What this means depends on the kind of algorithm that we will use for pattern matching. We will use algorithms that operate on the input hedge top-down, left-to-right, and bottom-up. This holds for streaming algorithms in particular.

Intuitively, when the pattern matching algorithm reaches a node top-down or left-to-right whose subsequent subhedge is irrelevant, then it can jump over it without inspecting its structure. What jumping means should be clear if the hedge is given by a graph that is stored in memory. Notice, that the full graph is to be constructed beforehand even though some parts of it may turn out irrelevant. Still one may win lots of time by jumping over irrelevant parts if either the graph was already constructed for other reasons, or if many pattern matching problems are to be solved on the same hedge.

In the case of streams, the irrelevant subhedge still needs to be parsed, but it will not be analyzed otherwise. Most typically, the possible analysis is done by automata, which may take 2 orders of magnitudes more time than needed for parsing. Therefore not having to do any analysis may considerably speed up a streaming algorithm.

Definition 7.1. Let $S \subseteq \mathcal{H}_\Sigma$ be a schema and $L \subseteq S$ a language satisfying this schema. We define diff_S^L as the least symmetric relation on prefixes of nested words $u, u' \in \text{pref}(\mathcal{N}_\Sigma)$

automaton has 2^{n+1} states while the minimal deterministic right-to-left automaton has $n + 1$ states.

such that:

$$\text{diff}_S^L(u, u') \Leftrightarrow \exists w \in \text{suffs}(\mathcal{N}_\Sigma). u \cdot w \in \text{nw}(L) \wedge u' \cdot w \in \text{nw}(S \setminus L)$$

A nested word prefix u is called *subhedge relevant* for L with schema S if there exists nested words $v, v' \in \mathcal{N}_\Sigma$ such that $\text{diff}_S^L(u \cdot v, u \cdot v')$. Otherwise, the prefix u is called *subhedge irrelevant* for L with schema S .

So $\text{diff}_S^L(u, u')$ states that u and u' have continuations that behave differently with respect to L and S . Furthermore, a prefix u is subhedge irrelevant if language membership does not depend on hedge insertion at u under the condition that schema membership is guaranteed. The case without schema restrictions is subsumed by the above definition by choosing $S = \mathcal{H}_\Sigma$. In this case, the complement $\overline{\text{diff}_{\mathcal{H}_\Sigma}^L}$ is the Myhill-Nerode congruence of L , which is well-known in formal language theory from the minimization of DFAs (see any standard textbook on formal language theory or Wikipedia³).

This congruence also serves as the basis for Gold-style learning of regular languages from positive and negative examples (see [Gold 1967] or Wikipedia⁴).

Schema restrictions are needed for our application to regular XPATH patterns.

Example 7.2 (Irrelevant subhedges for nested regular expression [self-list-child-item]). We reconsider the nested regular expression [self-list-child-item] from Example 3.24, which is a simplified version of the XPATH [self::list/child::item].

$$[\text{self-list-child-item}] = \langle \text{list} \cdot \text{N-List} \cdot \langle \text{item} \cdot \text{N-List} \rangle \cdot \text{N-List} \rangle \cdot \text{N-List}$$

In the signature $\Sigma = \{\text{list}, \text{item}\}$ and schema $S = \llbracket \text{N-List} \rrbracket$. Let $L = \llbracket [\text{self-list-child-item}] \rrbracket$ be the hedge language of this nested regular expression.

The nested word prefix $u = \langle \text{list} \langle \text{item} \rangle$ is subhedge irrelevant for the language L with schema S . Note that its hedge continuation $\langle \text{list} \langle \text{item} \rangle \rangle$ with the suffix $w = \rangle \rangle$ belongs to L . Hence, for any $h \in \mathcal{H}_\Sigma$ if the hedge continuation $\langle \text{list} \cdot \langle \text{item} \cdot h \rangle \rangle$ belongs to S then it also belongs to L . Nevertheless, for the hedge $h_1 = \langle \rangle$ the hedge continuation $\langle \text{list} \cdot \langle \text{item} \cdot h_1 \rangle \rangle$ does not belong to L , since it does not satisfy the schema S .

The prefix $\langle \text{item} \rangle$ is also irrelevant for the language L even independently of the schema. This is because L does not contain any continuation of this prefix to some hedge. The prefix $\langle \text{list} \rangle$ is not irrelevant for the language L wrt S . This can be seen with suffix

³Myhill-Nerode theorem: https://en.wikipedia.org/wiki/Myhill-Nerode_theorem

⁴Gold style learning in the limit https://en.wikipedia.org/wiki/Language_identification_in_the_limit

$w = \rangle$, since $\langle \text{list} \rangle$ does not belong to L while the hedge continuation $\langle \text{list} \cdot h \rangle$ with $h = \langle \text{item} \rangle$ does belong to L and both continuations satisfy the schema $S = \llbracket \text{N-List} \rrbracket$.

While needed in our main application, schema-restrictions may also have somehow surprising but logical consequences that make the problem of identifying irrelevant subhedges more tedious.

Example 7.3 (Surprising consequences of schema restrictions). *Consider the signature $\Sigma = \{a\}$, the pattern $L = \{\langle a \rangle\}$ and the schema $S = L \cup \{\langle \rangle \cdot a\}$. The prefix \langle is indeed subhedge irrelevant for L and S . In order to see this, we consider all possible closing suffixes one by one. The smallest closing prefix is \rangle . Note that a hedge $\langle h \rangle \in L$ if and only if $h = a$. So membership to L seems to depend on the subhedge h . But note that $\langle h \rangle \in S$ if and only if $h = a$. So when assuming that the hedge $\langle h \rangle$ belongs to the schema S , it must also belong to L . So the pattern must match in any case when assuming schema membership. The next closing suffix is $\rangle \cdot a$. Note that a hedge $\langle h \rangle \cdot a \in S$ if and only if $h = \epsilon$. However, $\langle h \rangle \cdot a \notin L$, so the pattern will not match for any subhedge h with this prefix when assuming the input hedge satisfies the schema. The situation is similar for larger suffixes, so in no case, language membership does depend on the subhedge at prefix \langle , when assuming that the full hedge satisfies the schema S .*

7.2.2 Basic Properties

We first show that subhedge irrelevant prefixes remain irrelevant when extended by any nested word, i.e. by the nested word of any hedge. This property is expected for any reasonable notion of subhedge irrelevance.

Lemma 7.4. *For any nested word prefix u , and language $L, S \subseteq \mathcal{H}_\Sigma$, and nested word $v \in \mathcal{N}_\Sigma$, if the prefix u is subhedge irrelevant for L with schema S and then the prefix $u \cdot v$ is so too.*

Proof. Let u be subhedge irrelevant for L with schema S and $v \in \mathcal{N}_\Sigma$ a nested word. We fix arbitrary nested words $v', v'' \in \mathcal{N}_\Sigma$ and a nested word suffix $w \in \text{suffs}(\mathcal{N}_\Sigma)$ such that $u \cdot v \cdot v' \cdot w \in \text{nw}(S)$ and $u \cdot v \cdot v'' \cdot w \in \text{nw}(S)$. Since $v \cdot v'$ and $v \cdot v''$ are nested words too, the subhedge irrelevance of u yields:

$$u \cdot v \cdot v' \cdot w \in \text{nw}(L) \Leftrightarrow u \cdot v \cdot v'' \cdot w \in \text{nw}(L)$$

Hence, the nested word prefix $u \cdot v$ is subhedge irrelevant for L with schema S too. \square

Definition 7.5. We call a binary relation D on prefixes of nested words a *difference relation* on prefixes if for all prefixes of nested words $u, u' \in \text{prefs}(\mathcal{N}_\Sigma)$ and nested words $v \in \mathcal{N}_\Sigma$:

$$(u \cdot v, u' \cdot v) \in D \Rightarrow (u, u') \in D$$

Lemma 7.6. diff_S^L is a difference relation.

Proof. We have to show for all prefixes $u, u' \in \text{prefs}(\mathcal{N}_\Sigma)$ and nested words $v \in \mathcal{N}_\Sigma$ that $(u \cdot v, u' \cdot v) \in \text{diff}_S^L$ implies $(u, u') \in \text{diff}_S^L$. So let u, u' be prefixes of nested words and v a nested word such that $(u \cdot v, u' \cdot v) \in \text{diff}_S^L$. Then there exists a nested suffix w such that

$$u \cdot v \cdot w \in \text{nw}(L) \wedge u' \cdot v \cdot w \in \text{nw}(\mathbf{S} \setminus L)$$

The suffix $\tilde{w} = v \cdot w$ then satisfies $u \cdot \tilde{w} \in \text{nw}(L) \wedge u' \cdot \tilde{w} \in \text{nw}(\mathbf{S} \setminus L)$. Hence $(u, u') \in \text{diff}_S^L$ as required. \square

In the schema-free case on words, the complement $\overline{\text{diff}_{\Sigma}^L}$ is an equivalence relation that is known as the Myhill-Nerode congruence. In the case of schemas, however, the complement $\overline{\text{diff}_S^L}$ may not be transitive, and thus, it may even not be an equivalence relation. This might be surprising at first sight.

Example 7.7. Let $\Sigma = \{a, b\}$, $L = \{\epsilon, b, aa\}$ and $\mathbf{S} = L \cup \{aab\}$. Then $(\epsilon, a) \in \overline{\text{diff}_S^L}$ since there is no continuation, that extends both ϵ and a into the schema. For the same reason, we have $(a, aa) \in \overline{\text{diff}_S^L}$. However, $\text{diff}_S^L(\epsilon, aa)$ since $\epsilon \cdot b \in L$ and $aa \cdot b \in \mathbf{S} \setminus L$. Thus, $(\epsilon, aa) \notin \overline{\text{diff}_S^L}$, so the complement $\overline{\text{diff}_S^L}$ is not transitive.

It should be noted for all languages L that $\overline{\text{diff}_{\mathcal{H}_\Sigma}^L}$ is reflexive and transitive, so it is an equivalence relation and thus a congruence. For general schemas \mathbf{S} , however, $\overline{\text{diff}_S^L}$ may not be transitive as shown by Example 7.7, and thus not be a congruence.

This indicates that schemas may make it more difficult to decide subhedge irrelevance. And indeed, the natural manner that one might expect to eliminate schemas based on complements does not work, as confirmed by following lemma (in particular the counter example in the proof 1. \Rightarrow 2.):

Lemma 7.8. For any nested word prefix $u \in \text{suffs}(\mathcal{N}_\Sigma)$, and languages $L, \mathbf{S} \subseteq \mathcal{H}_\Sigma$ consider the following two properties:

1. u is irrelevant for L with schema \mathbf{S}
2. u is irrelevant for $L \cup \overline{\mathbf{S}}$ with schema \mathcal{H}_Σ .

Then property 2. implies property 1., but not always vice versa.

Proof. 1. $\not\Rightarrow$ 2.: Here comes a counter-example. Let $L = \{\epsilon\}$ and $\mathbf{S} = \{\epsilon, a\}$. Then prefix a is irrelevant for L with schema \mathbf{S} . However, prefix a is relevant for $L \cup \bar{\mathbf{S}} = \mathcal{H}_\Sigma \setminus \{a\}$ with schema \mathcal{H}_Σ , since for the nested words $v = \epsilon$ and $v' = a$ we have $a \cdot v \notin L \cup \bar{\mathbf{S}}$ and $a \cdot v' \in L \cup \bar{\mathbf{S}}$.

1. \Leftarrow 2.: We assume property 2. Consider arbitrary nested words $v, v' \in \mathcal{N}_\Sigma$ and a nested word suffix $w \in \text{suffs}(\mathcal{N}_\Sigma)$ such that $u \cdot v \cdot w, u \cdot v' \cdot w \in \mathbf{S}$. It is sufficient to show that $u \cdot v \cdot w \in \mathbf{S}$ and $u \cdot v' \cdot w \in \mathbf{S}$ implies $u \cdot v \cdot w \in L \Leftrightarrow u \cdot v' \cdot w \in L$. This implication holds trivially if $u \cdot v \cdot w \in \bar{\mathbf{S}}$ or $u \cdot v' \cdot w \in \bar{\mathbf{S}}$. Otherwise, we have $u \cdot v \cdot w \in \mathbf{S}$ and $u \cdot v' \cdot w \in \mathbf{S}$. Property 2. as assumed then implies $u \cdot v \cdot w \in L \Leftrightarrow u \cdot v' \cdot w \in L$. So property 1. holds. □

7.2.3 Completeness for Subhedge Projection

We define for dSHAS what it means to be complete for subhedge projection. For this we start with the concept of subhedge projection states, a syntactic criterion to identify states of irrelevant subhedges.

7.2.3.1 Subhedge Projection States

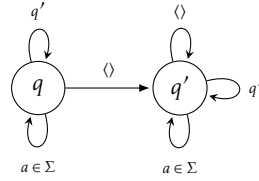
We next introduce the concept of subhedge projection states for SHA^\downarrow s in order to distinguish prefixes of hedges that are subhedge irrelevant, and to evaluate SHA^\downarrow s with subhedge projection.

Intuitively, a subhedge projection state can only loop in itself until a hedge closes (and then it is applied). When going down into a subtree, the subhedge projection state may still be changed into some other looping state. When leaving the subtree, however, one must come back to the original subhedge projection state. Clearly, any sink is a subhedge projection state. The more interesting cases, however, are subhedge projection states that are not sinks.

We start with SHA^\downarrow s without any restrictions but will impose schema-completeness and determinism later on.

Definition 7.9. We call a state $q \in \mathcal{Q}$ a subhedge projection state of Δ if there exists $q' \in \mathcal{Q}$ called the witness of q such that the set of transition rules of Δ containing q' or q

on the leftmost position is included in the following set:

$$\begin{aligned} & \{q \xrightarrow{\langle \rangle} q', q@q' \rightarrow q, q' \xrightarrow{\langle \rangle} q', q'@q' \rightarrow q'\} \\ & \cup \{q' \xrightarrow{a} q', q \xrightarrow{a} q \mid a \in \Sigma\} \end{aligned}$$


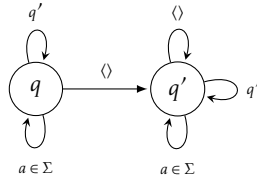
The set of all subhedged projection states of Δ is denoted by \mathcal{Q}_{shp}^Δ .

For any complete SHA^\downarrow , the above set of transition rules must be equal to the set of all transition rules of Δ with q or q' on the leftmost position. But if the SHA^\downarrow is only schema-complete for some schema \mathbf{S} then not all these transitions must be present. Note also that a subhedged projection state q may be equal to its witness q' . Therefore any witness q' of some subhedged projection state is itself a subhedged projection state with witness q' .

In the example $d\text{SHA}^\downarrow A = (\Sigma, \mathcal{Q}, \Delta, I, F)$ in Figure 3.15, we have $\mathcal{Q}_{shp}^\Delta = \{\Pi, 4, 2', 3', 1'', 2''\}$. The witness of all these subhedged projection states is Π . Note that not all possible transitions are present for the states in $\mathcal{Q}_{shp}^\Delta \setminus \{\Pi\}$, given that this automaton is not complete (but still schema-complete for schema $\llbracket \text{N-List} \rrbracket$).

Lemma 7.10. *If $q \in \mathcal{Q}_{shp}^\Delta$ is a subhedged projection state and $q \xrightarrow{h} p$ wrt Δ then $q = p$.*

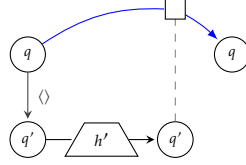
Proof. Suppose that $q \xrightarrow{h} p$ wrt Δ and that q is a subhedged projection state of Δ with witness q' . So the only possible transitions with q or q' on the left-hand side are the following:



Suppose that $h = t_1 \cdot \dots \cdot t_n$ is a sequence of trees or letters $t_i \in \langle \mathcal{H}_\Sigma \rangle \cup \Sigma$ where $1 \leq i \leq n$. Then there exists runs $R_i \in \text{run}^\Delta(t_i)$ and a run $R \in \text{run}^\Delta(h)$ that has the form $h = q_0 \cdot R_1 \cdot q_1 \cdot \dots \cdot R_n \cdot q_n$ where $q_0 = q$ and $q_n = p$. We prove for all $0 \leq i \leq n$ that $q_i = q$ by induction on i . For $i = 0$, this is trivial. For $i > 0$, the induction hypothesis shows that $q_{i-1} = q$

Case $t_i = a \in \Sigma$. Then $q_{i-1} \xrightarrow{a} q_i \in \Delta$. Since $q_{i-1} = q$ is a subhedge projection state of Δ it follows that $q_i = q$ too.

Case $t_i = \langle h' \rangle$ for some $h' \in \mathcal{H}_\Sigma$. Since $q_{i-1} = q$ is a subhedge projection state of Δ with witness q' , the subrun of R recognizing t_i must be justified by the following diagram:



So the subrun of R of h' may only contain the witness q' and furthermore, $q_i = q$.

□

7.2.3.2 Soundness

We show that subhedge projection states can be used to soundly identify subhedges that are irrelevant for subhedge projection. For this, we show that subhedge projection states in runs of deterministic SHA^\downarrow s occur only at irrelevant prefixes.

Proposition 7.11. *Let $A = (\Sigma, \mathcal{Q}, \Delta, I, F)$ be a $d\text{SHA}^\downarrow$ with schema \mathbf{S} , $R \in \text{run}^\Delta(h)$ the run of $\text{compl}(A)$ on some hedge $h \in \mathbf{S}$, and q a subhedge projection state for Δ . Then for any prefix $r \cdot q$ of $\text{nw}(R)$, the prefix $\text{proj}_\Sigma(r)$ of $\text{nw}(h)$ is subhedge irrelevant for $\mathcal{L}(A)$ for schema \mathbf{S} .*

Proof. Let $R \in \text{run}^\Delta(h)$ be the unique run of $\text{compl}(A)$ on some hedge $h \in \mathbf{S}$. Suppose that $r \cdot q \cdot s = \text{nw}(R)$ for some subhedge projection state q of Δ and let $v = \text{proj}_\Sigma(r)$ and $\wp = \text{proj}_\Sigma(s)$. Since $v \cdot \wp = \text{nw}(h)$ and $h \in \mathbf{S}$ it follows that $v \cdot \wp \in \text{nw}(\mathbf{S})$. We fix some hedge $h' \in \mathcal{H}_\Sigma$ such that $v \cdot \text{nw}(h') \cdot \wp \in \text{nw}(\mathbf{S})$ arbitrarily. Let $\tilde{h} \in \mathcal{H}_\Sigma$ such that $\text{nw}(\tilde{h}) = v \cdot \text{nw}(h') \cdot \wp$. For the subhedge irrelevance of v , we have to show that:

$$h \in \mathcal{L}(A) \Leftrightarrow \tilde{h} \in \mathcal{L}(A)$$

“ \Rightarrow ” We suppose that $h \in \mathcal{L}(A)$ and have to show that $\tilde{h} \in \mathcal{L}(A)$. For the partial run $r \cdot q$ of A on \tilde{h} there must exist R' and p such that $q \cdot R' \cdot p \in \text{run}^\Delta(h')$. Since q is a subhedge projection state of Δ , it follows by Lemma 7.10 that $q = p$. Hence $r \cdot q \cdot \text{nw}(R') \cdot q \cdot s$ is a run of A on \tilde{h} . Since R was successful for A , s must end in F ,

and thus the above run on \tilde{h} is successful for A too. Hence $\tilde{h} \in \mathcal{L}(A)$ as we had to show.

“ \Leftarrow ” We suppose that $\tilde{h} \in \mathcal{L}(A)$ and have to show that $h \in \mathcal{L}(A)$. Let \tilde{R} be the unique run of $\text{compl}(A)$ on \tilde{h} . By determinism, \tilde{R} must have the prefix $r \cdot q$. So there exist R', p', s such that $p \cdot R' \cdot q \in \text{run}^\Delta(h')$ and $r \cdot q \cdot R' \cdot p \cdot s = \tilde{R}$. Lemma 7.10 shows that $p = q$. Hence, $r \cdot q \cdot s$ is the unique run of $\text{compl}(A)$ on h and this run is accepting. So $h \in \mathcal{L}(A)$.

□

We note that Proposition 7.11 would not hold without assuming determinism. To see this, we can add some sink to any dSHA as a second initial state. One can then always go to this sink, which is a subhedge projection state. Nevertheless, no prefix going to the sink may be subhedge irrelevant.

7.2.3.3 Completeness

Proposition 7.11 shows that subhedge projection states permit to soundly distinguish prefixes that are subhedge irrelevant for deterministic SHA^\downarrow s. An interesting question is whether all subhedge irrelevant prefixes can be found in this way. A closely related question is whether there for all regular patterns there exist dSHAs that project all irrelevant subhedges.

Example 7.12. *Reconsider Example 7.3 with $\Sigma = \{a\}$, regular pattern $L = \{\langle a \rangle\}$ and regular schema $S = L \cup \{\langle \rangle \cdot a\}$. The prefix $\langle \rangle$ is irrelevant for L and S , since all its continuations ending in the schema are rejected: there is only one such continuation which is $\langle \rangle \cdot a$. However, a dSHA for L and S with maximal subhedge projection – as given in Figure 7.8 – will not go into a subhedge projection state at this prefix. The reason is that it will go into the same state $\{3, 4\}D0$ for any prefix in $\langle \mathcal{N}_\Sigma \rangle$, since ignoring the nested word of the irrelevant subhedge. So this state must accept $\langle a \rangle$. But when reading an a in this state, the dSHA goes into the rejection state $\{5\}D0$, showing that the hedge from the schema $\langle \rangle \cdot a \in S$ is rejected.*

The example shows that we have eventually to reason with sets of prefixes. We first lift the notion of subhedge irrelevance to prefix sets.

Definition 7.13. *Let $S \subseteq \mathcal{H}_\Sigma$ be a schema and $L \subseteq S$ a language satisfying this schema. A set of nested word prefixes $U \subseteq \text{prefs}(\mathcal{N}_\Sigma)$ is called subhedge relevant for L with schema S if there exist prefixes $u', u'' \in U$ and nested words $v', v'' \in \mathcal{N}_\Sigma$ such that $\text{diff}_S^L(u' \cdot v', u'' \cdot v'')$. Otherwise, the set U is called subhedge irrelevant for L wrt S .*

In order to explain the problem of Example 7.12, we define for each nested word prefix $w \in \text{prefs}(\mathcal{N}_\Sigma)$ a class $[w]_S^L$ of similar prefixes up replacing the nested word of any irrelevant subhedge by \mathcal{N}_Σ from the left to the right. For all prefixes $w \in \text{prefs}(\mathcal{N}_\Sigma)$, nested words $v \in \mathcal{N}_\Sigma$, letters $a \in \Sigma$, subsets of prefixes $U \subseteq \text{prefs}(\mathcal{N}_\Sigma)$, and $l \in \hat{\Sigma}$ letters or parenthesis:

$$\begin{aligned}
[w]_S^L &= \text{class_pr}_{\{\epsilon\}}(w) \\
\text{class_pr}_U(\epsilon) &= \begin{cases} U \cdot \mathcal{N}_\Sigma & \text{if } U \text{ irrelevant for } L \text{ wrt } S \\ U & \text{else} \end{cases} \\
\text{class_pr}_U(w \cdot \langle \cdot v \rangle) &= \text{class_ins}_{U'}(\langle \cdot v \rangle) \text{ where } U' = \text{class_pr}_U(w) \\
\text{class_pr}_U(a \cdot v) &= \text{class_ins}_{U'}(a, v) \text{ where } U' = U \cdot a \\
\text{class_pr}_U(\langle v' \rangle \cdot v) &= \text{class_ins}_{U'}(\langle \cdot v' \rangle, v) \text{ where } U' = \text{class_pr}_U(\langle \cdot v' \rangle) \\
\text{class_ins}_U(l, v) &= \begin{cases} U \cdot \mathcal{N}_\Sigma & \text{if } U \text{ irrelevant for } L \text{ wrt } S \\ \text{class_pr}_{U \cdot l}(v) & \text{else} \end{cases}
\end{aligned}$$

Definition 7.14. A prefix $u \in \text{prefs}(\mathcal{N}_\Sigma)$ is called strongly subhedge irrelevant for L wrt S if the set $[u]_S^L$ is subhedge irrelevant for L wrt S .

Example 7.15. In our running Example 7.12 where $L = \{\langle a \rangle\}$, $S = L \cup \{\langle \rangle \cdot a\}$ and $\Sigma = \{a\}$ we have $[\langle \rangle]_S^L = \langle \cdot \mathcal{N}_\Sigma$, so the prefix $\langle \rangle$ is strongly subhedge irrelevant. The set $[\langle a \rangle]_S^L = \langle \mathcal{N}_\Sigma \rangle$ is subhedge relevant, so the prefix $\langle a \rangle$ is not strongly subhedge irrelevant. The set $[\langle \rangle \cdot a]_S^L = \langle \mathcal{N}_\Sigma \rangle \cdot a \cdot \mathcal{N}_\Sigma$ is subhedge irrelevant, so the prefix $\langle \rangle \cdot a$ is strongly subhedge irrelevant.

Definition 7.16. A $d\text{SHA}^\downarrow A = (\Sigma, \mathcal{Q}, \Delta, I, F)$ is called complete for subhedge projection for schema S if A is schema-complete for S and for all hedges $h \in S$ and all prefixes v of $\text{nw}(h)$ that are strongly subhedge irrelevant for Δ , the unique run of A on h goes to some subhedge projection state of Δ at v .

Example 7.17. The $d\text{SHA}$ in Figure 7.8 is complete for subhedge projection for our running Example 7.12 where $L = \{\langle a \rangle\}$ and $S = L \cup \{\langle \rangle \cdot a\}$. It was obtained by congruence projection. On all prefixes of the subhedge irrelevant class $[\langle \rangle]_S^L$ it goes to the subhedge projection state $\{0\}D1$. On all prefixes of the class $[\langle a \rangle]_S^L$ which is not subhedge irrelevant, it goes to the state $\{3, 4\}D0$ which is not a subhedge projection state. And on all prefixes of the subhedge irrelevant class $[\langle \rangle \cdot a]_S^L$ it goes to the subhedge projection state $\{5\}D0$.

Example 7.18. The $d\text{SHA}^\downarrow$ from Figure 3.15 is complete for subhedge projection with schema $\llbracket \text{N-List} \rrbracket$. It can be obtained by safe-no-change projection. In general, however, $d\text{SHA}^\downarrow$ s obtained by safe-no-change projection may not be complete for subhedge projection, as we will discuss in Section 7.3.3.

7.3 Safe-No-Change Projection

We want to solve the membership problem for hedges – also called regular hedge pattern matching – with subhedge projection. We assume that the regular language (pattern) is given by a $d\text{SHA}$ while the schema may be arbitrary.

We present the safe-no-change projection algorithm which compiles the $d\text{SHA}$ into some $d\text{SHA}^\downarrow$ with the same language while introducing subhedge projection states.

7.3.1 Algorithm

The idea of the safe-no-change projection is to push information top-down by which to distinguish looping states that will safely no more be changed. Thereby, subhedge projection states are produced as we will illustrate by examples. The soundness proof of safe-no-change projection is nontrivial and instructive for the soundness proof of congruence projection in Section 7.4.4. Completeness for subhedge projection cannot be expected, since the safe-no-change projection does only take simple loops into account. More general loops cannot be treated and it is not clear how that could be done. The congruence projection algorithm in Section 7.4.3 will eventually give an answer to this.

We keep the safe-no-change compiler independent of the schema and therefore have not even to assume its regularity. But we have to assume the schema-completeness (see Section 3.5) of the input $d\text{SHA}$, in order to show that the output $d\text{SHA}^\downarrow$ recognizes the same language within the schema. The regular pattern matching problem can then be solved in memory and with subhedge projection, by evaluating the $d\text{SHA}^\downarrow$ on the input hedge, in memory and with subhedge projection as discussed in Section 10.1.1.

We now describe the safe-no-change projection algorithm. Let $A = (\Sigma, \mathcal{Q}, \Delta, I, F)$ be a SHA with schema \mathbf{S} . The algorithm takes the $d\text{SHA}$ A as input and compiles it to some $d\text{SHA}^\downarrow A^{\text{snc}}$. We will state why A^{snc} is sound for subhedge projection under the condition that A is schema-complete for schema \mathbf{S} . We do *not* even have to assume that the schema \mathbf{S} is regular.

For any set $P \subseteq \mathcal{Q}$ we define the set of states that safely lead to P :

$$\text{safe}^\Delta(P) = \{q \in \mathcal{Q} \mid \text{acc}^\Delta(\{q\}) \subseteq P\}$$

So a state q is safe for P if any hedge read from q on which the run with Δ does not

block must reach some state in P . We note that $\text{safe}^\Delta(P)$ can be computed in linear time in the size of Δ , by using inverse hedge accessibility from P . We define the set of states that will no longer change:

$$\text{no-change}^\Delta = \{q \mid q \in \text{safe}^\Delta(\{q\})\}$$

So $q \in \text{no-change}^\Delta$ if and only if $\text{acc}^\Delta(\{q\}) \subseteq \{q\}$. This means that all transition rules starting in q must loop in q . In the example automaton from Figure 3.10, the self-looping states are those in $\text{no-change}^\Delta = \{2, 3, 4, 5\}$.

Lemma 7.19. *Let $A = (\Sigma, \mathcal{Q}, \Delta, I, F)$ be a SHA that is schema-complete for schema \mathbf{S} . For any hedge $h \in \mathbf{S}$ and state $q \in I \cap \text{no-change}^\Delta$ it then holds that $q \xrightarrow{h} q$ wrt Δ .*

Proof. The schema-completeness of A for \mathbf{S} applied to $h \in \mathbf{S}$ and $q \in I$ yields the existence of some state $q' \in \mathcal{Q}$ such that $q \xrightarrow{h} q'$ wrt Δ . Let q' be any such state. Note that $q' \in \text{acc}^\Delta(q)$. Since $q \in \text{no-change}^\Delta$, we have $q \in \text{safe}^\Delta(\{q\})$ so that $q' \in \text{acc}^\Delta(\{q\})$ implies $q' = q$. This proves $q \xrightarrow{h} q$ wrt Δ . \square

For any state $q \in \mathcal{Q}$ and subset of states $Q \subseteq \mathcal{Q}$ we define:

$$\begin{aligned} s\text{-down}^\Delta(q, Q) &= \text{safe}^\Delta(\{p \in \mathcal{Q} \mid q @^\Delta p \subseteq Q\}) \\ s\text{-no-change}^\Delta(q) &= s\text{-down}^\Delta(q, \{q\}) \end{aligned}$$

A state belongs to $s\text{-down}^\Delta(q, Q)$ if all states $p \in \text{acc}^\Delta(q)$ satisfy $q @^\Delta p \subseteq Q$. So p is a state down that will safely go up to some state in Q . A state p belongs to $s\text{-no-change}^\Delta(q)$ if it safely does not change q , i.e., if $\{q\} @^\Delta \text{acc}^\Delta(\{p\}) \subseteq \{q\}$.

We next compile the SHA A to a $\text{SHA}^\downarrow A^{\text{snc}} = (\Sigma, \mathcal{Q}^{\text{snc}}, \Delta^{\text{snc}}, I^{\text{snc}}, F^{\text{snc}})$. For this let Π be a fresh symbol and consider the state set:

$$\mathcal{Q}^{\text{snc}} = \{\Pi\} \uplus (\mathcal{Q} \times 2^\mathcal{Q})$$

In practice we restrict the state space to the states that are accessible, or clean the dSHAs keeping only those states that are used in some successful run. But in the worst case, the construction may indeed be exponential.

A pair (q, P) means the state q may will safely no more change in the current subhedge if $q \in P \cup \text{no-change}^\Delta$. In this case, the subhedge can be projected. The sets of initial and final states are defined as follows:

$$I^{\text{snc}} = I \times \{\emptyset\} \quad F^{\text{snc}} = F \times \{\emptyset\}$$

$$\begin{array}{c}
\frac{q \xrightarrow{a} q' \in \Delta \quad q \notin P \cup \text{no-change}^\Delta}{(q, P) \xrightarrow{a} (q', P) \in \Delta^{snc}} \quad \frac{a \in \Sigma \quad q \in P \cup \text{no-change}^\Delta}{(q, P) \xrightarrow{a} (q, P) \in \Delta^{snc}} \\
\\
\frac{\langle \rangle \xrightarrow{} q' \in \Delta \quad q \notin P \cup \text{no-change}^\Delta}{(q, P) \xrightarrow{\langle \rangle} (q', s\text{-no-change}^\Delta(q)) \in \Delta^{snc}} \quad \frac{q \in P \cup \text{no-change}^\Delta}{(q, P) \xrightarrow{\langle \rangle} \Pi \in \Delta^{snc}} \\
\\
\frac{q@p \rightarrow q' \in \Delta \quad q \notin P \cup \text{no-change}^\Delta}{(q, P)@(p, s\text{-no-change}^\Delta(q)) \rightarrow (q', P) \in \Delta^{snc}} \quad \frac{q \in P \cup \text{no-change}^\Delta}{(q, P)@\Pi \rightarrow (q, P) \in \Delta^{snc}} \\
\\
\frac{a \in \Sigma}{\Pi \xrightarrow{a} \Pi \in \Delta^{snc}} \quad \frac{\text{true}}{\Pi@\Pi \rightarrow \Pi \in \Delta^{snc}} \quad \frac{\text{true}}{\Pi \xrightarrow{\langle \rangle} \Pi \in \Delta^{snc}}
\end{array}$$

Figure 7.1: The transition rules of the $\text{SHA}^\downarrow A^{snc}$ inferred from those of the $\text{SHA} A$.

How to generate the transition rules of A^{snc} from those of A is described in Figure 7.1. On states assigned on top-level (q, P) , the set P is empty, so that only the states in no-change^Δ are safe for no change. This is why the definition of I^{snc} and F^{snc} use $P = \emptyset$. When going down from some state (q, P) for which q is safe to not change, i.e., $q \in P \cup \text{no-change}^\Delta$, then the following rule is applied:

$$\frac{q \in P \cup \text{no-change}^\Delta}{(q, P) \xrightarrow{\langle \rangle} \Pi \in \Delta^{snc}}$$

The evaluation on the lower level goes to the extra state Π , where it then loops until going back to q on the upper level. When going down from some state (q, P) such that $q \notin P \cup \text{no-change}^\Delta$, then the following rule is applied:

$$\frac{\langle \rangle \xrightarrow{} q' \in \Delta \quad q \notin P \cup \text{no-change}^\Delta}{(q, P) \xrightarrow{\langle \rangle} (q', s\text{-no-change}^\Delta(q)) \in \Delta^{snc}}$$

The states in the set $s\text{-no-change}^\Delta(q)$ on the lower level safely will not make q change on the upper level for any subhedger to come⁵.

When applied to the SHA in Figure 3.10 for [self-list-child-item], the construction yields the SHA^\downarrow in Figure 7.2 which is indeed equal to the SHA^\downarrow from Figure 3.15 up

⁵We could detect more irrelevant subhedgers by allowing q to change but only in a unique manner. This can be obtained with the alternative definition: $s\text{-no-change}'^\Delta(q) = \cup_{r \in Q} s\text{-down}^\Delta(q, \{r\})$. Computing this set would require quadratic time $O(n |A|)$, while computing $s\text{-no-change}^\Delta(q)$ can be done in linear time $O(|A|)$.

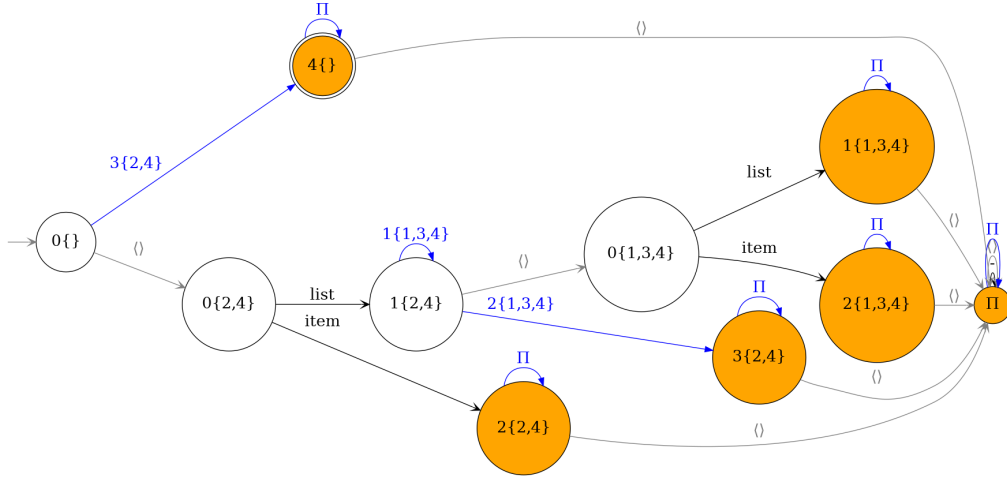


Figure 7.2: The safe-no-change projection $d\text{SHA}^\downarrow A^{\text{snC}}$ constructed from the $d\text{SHA}$ A for query [self-list-child-item] in Figure 3.10. Subhedge projection states are colored in orange. Useless states and transitions leading out of schema $\llbracket \text{N-List} \rrbracket$ are omitted. State Π has an else transition labeled by wildcard letter “–”, which stands for either letter *list* or *item*. This $d\text{SHA}^\downarrow$ is equal to the $d\text{SHA}^\downarrow$ in Figure 3.15 up to the state renaming $0 = (0, \{\})$, $0' = (0, \{2, 4\})$, $0'' = (0, \{1, 3, 4\})$, $1' = (1, \{2, 4\})$, $1'' = (1, \{1, 3, 4\})$, $2' = (2, \{2, 4\})$, $2'' = (2, \{1, 3, 4\})$, $3' = (3, \{2, 4\})$, $4 = (4, \{\})$. Recall that $\text{no-change}^\Delta = \{2, 3, 4, 5\}$.

to state renaming. When run on the hedge $\langle \text{list} \cdot \langle \text{list} \cdot h_1 \rangle \cdot \langle \text{item} \cdot h_2 \rangle \rangle$ as shown in Figure 3.16, it does not have to visit the subhedges h_1 nor h_2 , since all of them will be reached starting from the projection state Π .

7.3.2 Soundness

We next state and prove a soundness result for safe-no-change projection.

Theorem 2 (Soundness of Safe-No-Change Projection). *If a SHA A is schema-complete for some schema S , then safe-no-change projection for A preserves the language within this schema: $\mathcal{L}(A^{\text{snC}}) \cap S = \mathcal{L}(A)$.*

Proof. We have to prove that no more changing states $q \in P \cup \text{no-change}^\Delta$ is sound. If $q \in \text{no-change}^\Delta$, this follows from the schema-completeness of Δ , so that one can neither block on any hedge from the schema nor change the state. In the case $q \in P$, the intuition is that the state on level above – say r – can neither change, since $P = s\text{-no-change}^\Delta(r)$, nor can the automaton block on any hedge from the schema due to schema-completeness.

We first prove the inclusion $\mathcal{L}(A) \subseteq \mathcal{L}(A^{snc})$. Since $\mathcal{L}(A) \subseteq \mathbf{S}$ by definition of schemas, this implies $\mathcal{L}(A) \subseteq \mathcal{L}(A^{snc}) \cap \mathbf{S}$. The proof will be based on the following three Claims 2.1a, 2.2a, and 2.3a. Note that schema-completeness is not needed for this direction.

Claim 2.1a. $\Pi \xrightarrow{h} \Pi$ wrt Δ^{snc} for all hedges $h \in \mathcal{H}_\Sigma$.

The proof is straightforward by induction on the structure of h . It uses the last three transition rules of Δ^{snc} in Figure 7.1 permitting to always stay in Π for whatever hedge follows.

Claim 2.2a. For all $h \in \mathcal{H}_\Sigma$, $q \in \mathcal{Q}$, and $P \subseteq \mathcal{Q}$ such that $q \in P \cup no\text{-}change^\Delta$:

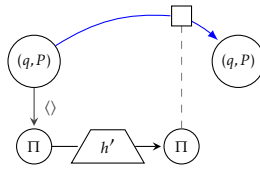
$$(q, P) \xrightarrow{h} (q, P) \text{ wrt } \Delta^{snc}$$

We prove Claim 2.2a by induction on the structure of h .

Case $h = \langle h' \rangle$. In this case, we can use Claim 2.1a to show $\Pi \xrightarrow{h'} \Pi$ wrt Δ^{snc} and the inference rules

$$\frac{q \in P \cup no\text{-}change^\Delta}{(q, P) \xrightarrow{\diamond} \Pi \in \Delta^{snc}} \quad \frac{q \in P \cup no\text{-}change^\Delta}{(q, P) @ \Pi \rightarrow (q, P) \in \Delta^{snc}}$$

in order to close the following diagram with respect to Δ^{snc} :



This proves $(q, P) \xrightarrow{h} (q, P)$ wrt Δ^{snc} as required by the claim.

Case $h = a$. Since $q \in P \cup no\text{-}change^\Delta$ we can apply the inference rule:

$$\frac{a \in \Sigma \quad q \in P \cup no\text{-}change^\Delta}{(q, P) \xrightarrow{a} (q, P) \in \Delta^{snc}}$$

This proves this case of the claim.

Case $h = \epsilon$. We trivially have $(q, P) \xrightarrow{\epsilon} (q, P)$ wrt Δ^{snc} .

Case $h = h' \cdot h''$. By induction hypothesis applied to h' and h'' , we have: $(q, P) \xrightarrow{h'} (q, P)$ and $(q, P) \xrightarrow{h''} (q, P)$ wrt Δ^{snc} . Hence $(q, P) \xrightarrow{h' \cdot h''} (q, P)$ wrt Δ^{snc} .

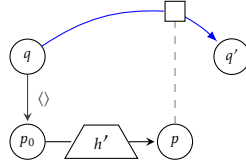
This ends the proof of Claim 2.2a. The next claim, in which the induction step is a little more tedious to prove, is the key of the soundness proof. We define the following predicate for all $q', q'' \in \mathcal{Q}$ and $P \subseteq \mathcal{Q}$:

$$q' \sim_P q'' \text{ iff } (q' = q'' \vee q', q'' \in P)$$

Claim 2.3a. Let $h \in \mathcal{H}_\Sigma$ a hedge, $q, q' \in \mathcal{Q}$ states and $P \subseteq \mathcal{Q}$ a subset of states such that $\text{acc}^\Delta(P) \subseteq P$ and $q \notin P \cup \text{no-change}^\Delta$. If $q \xrightarrow{h} q'$ wrt Δ then there exists q'' such that $(q, P) \xrightarrow{h} (q'', P)$ wrt Δ^{snc} and $q' \sim_P q''$.

Proof. By induction on the structure of h .

Case $h = \langle h' \rangle$. The assumption $q \xrightarrow{h} q'$ wrt Δ shows that there exists states $p_0 \in \langle \rangle^\Delta$ and $p \in \mathcal{Q}$ closing the following diagram:

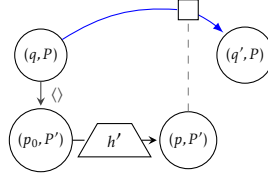


Let $P' = s\text{-no-change}^\Delta(q)$ and note that $\text{acc}^\Delta(P') \subseteq P'$. Since $q \notin P \cup \text{no-change}^\Delta$ we can infer:

$$\frac{\xrightarrow{\langle \rangle} p_0 \in \Delta \quad q \notin P \cup \text{no-change}^\Delta}{(q, P) \xrightarrow{\langle \rangle} (p_0, P') \in \Delta^{snc}} \quad \frac{q @ p \rightarrow q' \in \Delta \quad q \notin P \cup \text{no-change}^\Delta}{(q, P) @ (p, P') \rightarrow (q', P) \in \Delta^{snc}}$$

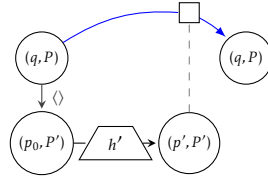
Subcase $p_0 \notin P' \cup \text{no-change}^\Delta$. The induction hypothesis applies to h' shows that there exists p' such that $(p_0, P') \xrightarrow{h'} (p', P')$ wrt Δ^{snc} and $p \sim_P p'$. We distinguish the two cases justifying the latter predicate:

Subsubcase $p' = p$. Hence $(p_0, P') \xrightarrow{h'} (p, P')$ wrt Δ^{snc} , so we can close the diagram as follows:



This shows that $(q, P) \xrightarrow{h} (q', P)$ wrt Δ^{snc} , and thus the claim since $q' \sim_P q'$.

Subsubcase $p, p' \in P'$. Since $p' \in P'$ and $P' = s\text{-no-change}^\Delta(q)$ we have $q@p' \rightarrow q$ in Δ . Hence we can close the diagram as follows:

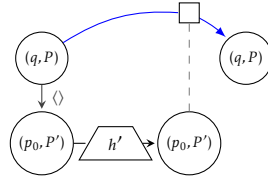


Since $p \in P'$ and $q@p \rightarrow q'$ in Δ we have $q = q'$ by definition of $P' = s\text{-no-change}^\Delta(q)$.

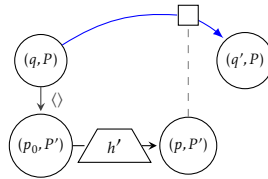
This shows that $(q, P) \xrightarrow{h} (q, P)$ wrt Δ^{snc} . Since $q \sim_P q$ the claim follows.

Subcase $p_0 \in P' \cup \text{no-change}^\Delta$. Claim 2.2a then shows that $(p_0, P') \xrightarrow{h'} (p_0, P')$ wrt Δ^{snc} .

Subsubcase $p_0 \in P'$. Since $p \in \text{acc}^\Delta(p_0)$ and $\text{acc}^\Delta(P') \subseteq P'$ it follows that $p \in P'$ too. By definition $P' = s\text{-no-change}^\Delta(q)$ and the completeness of Δ , the memberships $p_0 \in P'$ and $p \in P'$ imply that $q@^\Delta p_0 = \{q\} = q@^\Delta p$. We can now close the diagram below as follows:



Subsubcase $p_0 \in \text{no-change}^\Delta$. In this case $p_0 = p$ so that $q' \in q@^\Delta p_0$. Hence:



Case $h = a$. Since $q \notin P \cup no-change^\Delta$ we can apply the inference rule:

$$\frac{q \xrightarrow{a} q' \in \Delta \quad q \notin P \cup no-change^\Delta}{(q, P) \xrightarrow{a} (q', P) \in \Delta^{snc}}$$

This shows that $(q, P) \xrightarrow{h} (q', P)$ validating the claim since $q' \sim_P q$.

Case $h = \epsilon$. In this case we have $q = q'$ and $(q, P) \xrightarrow{\epsilon} (q, P)$, so the claim holds.

Case $h = h_1 \cdot h_2$. Since $q \xrightarrow{h} q'$ wrt Δ , there exists $q_1 \in \mathcal{Q}$ such that $q \xrightarrow{h_1} q_1$ wrt Δ and $q_1 \xrightarrow{h_2} q'$ wrt Δ . Since $q \notin P \cup no-change^\Delta$, we apply the induction hypothesis on h_1 . This implies that there exists q'_1 such that:

$$(q, P) \xrightarrow{h_1} (q'_1, P) \text{ wrt } \Delta^{snc} \text{ and } q_1 \sim_P q'_1$$

We distinguish the two cases of $q_1 \sim_P q'_1$:

Subcase $q_1 = q'_1$. We also distinguish two subcases here:

Subsubcase $q_1 \notin P$. The induction hypothesis applied to h_2 yields:

$$\exists q''. (q_1, P) \xrightarrow{h_2} (q'', P) \text{ wrt } \Delta^{snc} \wedge q' \sim_P q''$$

Hence

$$\exists q''. (q, P) \xrightarrow{h} (q'', P) \text{ wrt } \Delta^{snc} \wedge q' \sim_P q''$$

Subsubcase $q_1 \in P$. By Claim 2.2a, we have $(q_1, P) \xrightarrow{h_2} (q_1, P)$. We also have $q' \in acc(\{q_1\})$ and since we assume $acc(P) \subseteq P$, this implies $q' \in P$. Hence $(q, P) \xrightarrow{h} (q_1, P)$ and $q', q_1 \in P$ implying the claim with $q' \sim_P q_1$.

Subcase $q_1, q'_1 \in P$. Since $q'_1 \in P$, Claim 2.2a, implies $(q'_1, P) \xrightarrow{h_2} (q'_1, P)$ wrt Δ^{snc} . Thus $(q, P) \xrightarrow{h} (q'_1, P)$ wrt Δ^{snc} . Since $q' \in acc^\Delta(\{q_1\})$ and $q_1 \in P$, it follows that $q' \in acc^\Delta(P) \subseteq P$. Here, we used as in the previous subsubcase that $acc(P) \subseteq P$ is assumed by the claim. Let $q'' = q'_1$. Then we have $(q, P) \xrightarrow{h} (q'', P)$ wrt Δ^{snc} and $q', q'' \in P$ showing the claim.

This ends the proof of Claim 2.3a.

Proof of inclusion $\mathcal{L}(A) \subseteq \mathcal{L}(A^{snc})$. Let $h \in \mathcal{L}(A)$. Then there exists $q_0 \in I$ and $q \in F$ such that $q_0 \xrightarrow{h} q$. We distinguish two cases:

Case $q_0 \in no-change^\Delta$. By definition of $no-change^\Delta$ and since $q \in acc^\Delta(q_0)$ we have $q_0 = q$. Claim 2.2a shows that $(q_0, \emptyset) \xrightarrow{h} (q_0, \emptyset)$ wrt Δ^{snc} and thus $(q_0, \emptyset) \xrightarrow{h} (q, \emptyset)$ so that $h \in \mathcal{L}(A^{snc})$.

Case $q_0 \notin no-change^\Delta$. Claim 2.3a with $P = \emptyset$ shows that $(q_0, \emptyset) \xrightarrow{h} (q, \emptyset)$ wrt Δ^{snc} and hence $h \in \mathcal{L}(A^{snc})$.

This ends the proof of the first inclusion.

We next want to show the inverse inclusion $\mathcal{L}(A^{snc}) \cap \mathbf{S} \subseteq \mathcal{L}(A)$. It will eventually follow from the following three Claims 2.1b, 2.2b, and 2.3b.

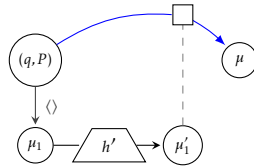
Claim 2.1b. For any hedge h and state $\mu \in \mathcal{Q}^{snc}$, if $\Pi \xrightarrow{h} \mu$ wrt Δ^{snc} then $\mu = \Pi$.

The proof is straightforward by induction on the structure of h : the only transition rules of Δ^{snc} with Π on the left hand side are inferred by the last three rules in Figure 7.1. These require to stay in Π whatever hedge follows.

Claim 2.2b. For any hedge h , set $P \subseteq \mathcal{Q}$, state $q \in P \cup no-change^\Delta$, and state $\mu \in \mathcal{Q}^{snc}$: if $(q, P) \xrightarrow{h} \mu$ wrt Δ^{snc} then $\mu = (q, P)$.

Proof. By induction on the structure of h . Suppose that $(q, P) \xrightarrow{h} \mu$ wrt Δ^{snc} .

Case $h = \langle h' \rangle$. There must exist states $\mu_1, \mu'_1 \in \mathcal{Q}^{snc}$ closing the following diagram:



Since $q \in P \cup no-change^\Delta$, the following rule must have been applied to infer $(q, P) \xrightarrow{\langle \rangle} \mu_1$ wrt Δ^{snc} :

$$\frac{q \in P \cup no-change^\Delta}{(q, P) \xrightarrow{\langle \rangle} \Pi \in \Delta^{snc}}$$

Therefore $\mu_1 = \Pi$. Claim 2.1b shows that $\mu'_1 = \Pi$ too. So μ must have been inferred by applying the rule:

$$\frac{q \in P \cup \text{no-change}^\Delta}{(q, P)@ \Pi \rightarrow (q, P) \in \Delta^{snc}}$$

So $\mu = (q, P)$ as required.

Case $h = a$. The following rule must have been applied:

$$\frac{q \in P \cup \text{no-change}^\Delta}{(q, P) \xrightarrow{a} (q, P) \in \Delta^{snc}}$$

Hence, $\mu = (q, P)$.

Case $h = \epsilon$. Obvious.

Case $h = h_1 \cdot h_2$. There must exist μ_1 such that $(q, P) \xrightarrow{h_1} \mu_1 \xrightarrow{h_2} \mu$ wrt Δ^{snc} . By induction hypothesis applied to h_1 , we have $\mu_1 = (q, P)$. We can thus apply the induction hypothesis to h_2 to obtain $\mu_2 = (q, P)$.

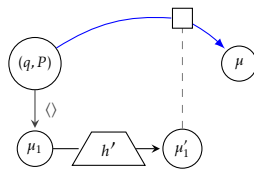
This ends the proof of Claim 2.2b. We next need an inverse of Claim 2.3a.

Claim 2.3b. Let $q \in \mathcal{Q}$, $P \subseteq \mathcal{Q}$ such that $q \notin P \cup \text{no-change}^\Delta$ and $\text{acc}^\Delta(P) \subseteq P$. For any $h \in \mathcal{H}_\Sigma$ such that $(q, P) \xrightarrow{h} \mu$ wrt Δ^{snc} for some $\mu \in \mathcal{Q}^{snc}$ and such that Δ does not have any blocking partial run on h starting from q , there exists q', q'' such that:

$$\mu = (q', P), \quad q \xrightarrow{h} q'' \text{ wrt } \Delta, \text{ and } q' \sim_P q''.$$

Proof. By induction on the structure of $h \in \mathcal{H}_\Sigma$. We distinguish cases for all possible forms h .

Case $h = \langle h' \rangle$. By definition of $(q, P) \xrightarrow{h} \mu$ wrt Δ^{snc} there must exist $\mu_1, \mu'_1 \in \mathcal{Q}^{snc}$ such that the following diagram can be closed:



Since $q \notin P \cup \text{no-change}^\Delta$, the following rule got applied to infer $(q, P) \xrightarrow{\langle \rangle} \mu_1$ where $P' = s\text{-no-change}^\Delta(q)$:

$$\frac{\xrightarrow{\langle \rangle} p \in \Delta \quad q \notin P \cup \text{no-change}^\Delta}{(q, P) \xrightarrow{\langle \rangle} (p, P') \in \Delta^{snc}}$$

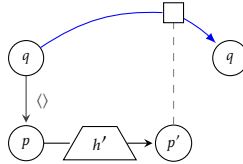
Hence, there exists $p \in \langle \rangle^\Delta$ such that $\mu_1 = (p, P')$. We fix such a state p arbitrarily.

Since Δ does not have any blocking runs on h from q there exists $p', q'' \in \mathcal{Q}$ such that $p \xrightarrow{h'} p'$ and $q@p' \rightarrow q''$ wrt Δ . Furthermore, Δ does not have any blocking partial run on h' starting from p .

Subcase $p \in P'$. In this case, we can apply Claim 2.2b to $(p, P') \xrightarrow{h'} \mu'_1$ wrt Δ in order to show that $\mu'_1 = (p, P')$. Since $p \in \text{acc}^\Delta(\{p\})$ and $p \in P' = s\text{-no-change}^\Delta(q)$ it follows that $q@^\Delta p = \{q\}$. Hence the following rule got applied to infer $(q, P) \xrightarrow{h} \mu$:

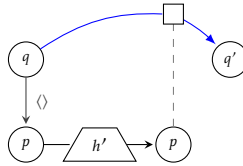
$$\frac{q@p \rightarrow q \in \Delta \quad q \notin P \cup \text{no-change}^\Delta}{(q, P)@ (p, P') \rightarrow (q, P) \in \Delta^{snc}}$$

This shows that $\mu = (q, P)$. Let $q' = q$ so that $\mu = (q', P)$. Since $p \xrightarrow{h'} p'$ wrt Δ we have $p' \in \text{acc}^\Delta(\{p\})$ so that $q@p' \rightarrow q$ wrt Δ by definition of $s\text{-no-change}^\Delta(\{q\})$. Hence, we can close the following diagram:



Let $q'' = q$, so that $q' = q''$. It then holds that $q' \sim_P q''$ and $q \xrightarrow{\langle h' \rangle} q'$ wrt Δ , as required by the claim.

Subcase $p \in \text{no-change}^\Delta$. In this case $p \xrightarrow{h'} p'$ wrt Δ implies that $p' = p$. Hence, we can close the following diagram:



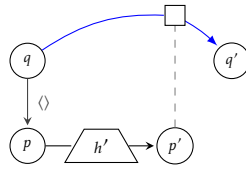
This shows that $q \xrightarrow{\langle h' \rangle} q'$ wrt Δ .

Subcase $p \notin P' \cup \text{no-change}^\Delta$. By induction hypothesis applied to $(p, P') \xrightarrow{h'} \mu'_1$ wrt Δ there exists p'' such that:

$$\mu'_1 = (p', P'), p \xrightarrow{h'} p'' \text{ wrt } \Delta, \text{ and } p' \sim_{P'} p''.$$

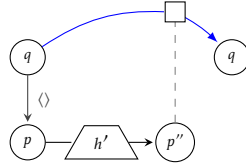
Since Δ does not have blocking runs on h starting with q there exist q'' such that $q @ p'' \rightarrow q''$ wrt Δ . There are two ways to satisfy $p' \sim_{P'} p''$:

Subsubcase $p' = p''$. We then have:



This shows $q \xrightarrow{\langle h' \rangle} q'$ wrt Δ .

Subsubcase $p', p'' \in P$. By definition of P' it follows that $q' = q = q''$. Hence:



This shows $q \xrightarrow{\langle h' \rangle} q$ wrt Δ .

Case $h = a$. Since $q \notin P \cup \text{no-change}^\Delta$, the following inference rule must be used:

$$\frac{q \xrightarrow{a} q' \in \Delta \quad q \notin P \cup \text{no-change}^\Delta}{(q, P) \xrightarrow{a} (q', P) \in \Delta^{snc}}$$

So $\mu = (q', P)$ and $q \xrightarrow{a} q'$ wrt Δ .

Case $h = \epsilon$. Obvious.

Case $h = h_1 \cdot h_2$. The judgment $(q, P) \xrightarrow{h} \mu$ wrt Δ^{snc} shows that there exists μ_1 such that $(q, P) \xrightarrow{h_1} \mu_1 \xrightarrow{h_2} \mu$ wrt Δ^{snc} . Since $q \notin P \cup \text{no-change}^\Delta$, we can apply the induction hypothesis to h_1 . It shows that there exists q'_1 and q''_1 such that $\mu_1 = (q'_1, P)$, $q \xrightarrow{h_1} q'_1$ wrt Δ and $q'_1 \sim_P q''_1$.

Subcase $q'_1 = q''_1$. Hence $(q_1, P) \xrightarrow{h_2} \mu$ wrt Δ^{snc} .

Subsubcase $q'_1 \notin P \cup \text{no-change}^\Delta$. In this case, we can apply the induction hypothesis to $(q'_1, P) \xrightarrow{h_2} \mu$ wrt Δ^{snc} showing the existence of q' such that $\mu = (q', P)$ and a state q'' such that $q'_1 \xrightarrow{h_2} q''$ and $q' \sim_P q''$. Hence $\exists q''$. $q \xrightarrow{h} q''$ wrt Δ and $q' \sim_P q''$, so the claim holds.

Subsubcase $q'_1 \in P \cup \text{no-change}^\Delta$. Claim 2.2b applied to $(q'_1, P) \xrightarrow{h_2} \mu$ wrt Δ^{snc} shows that $\mu = (q'_1, P)$.

Subcase $q'_1, q''_1 \in P$. Recall that $(q''_1, P) \xrightarrow{h_2} \mu$ wrt Δ^{snc} and $q''_1 \in P$. Claim 2.2b shows that $\mu = (q''_1, P)$ wrt Δ^{snc} . We also have $q \xrightarrow{h_1} q''_1$ wrt Δ . Since there are no blocking partial runs on h starting from q there exist a state q'' such that $q''_1 \xrightarrow{h_2} q''$ wrt Δ . Since $q'_1 \in P$ and P is closed by accessibility, we have $q'' \in \text{acc}(\{q'_1\}) \subseteq \text{acc}(P) \subseteq P$. From $q \xrightarrow{h_1} q''_1$ wrt Δ , we get $q \xrightarrow{h} q''$ wrt Δ . Since $q''_1, q'' \in P$ it follows that $q''_1 \sim_P q''$ and thus the claim holds.

This ends the proof of Claim 2.3b.

Proof of inclusion $\mathcal{L}(A^{snc}) \cap \mathbf{S} \subseteq \mathcal{L}(A)$. Let $h \in \mathcal{L}(A^{snc}) \cap \mathbf{S}$. Since $h \in \mathcal{L}(A^{snc})$ then there exists $q_0 \in I$ and $q \in F$ such that $(q_0, \emptyset) \xrightarrow{h} (q, \emptyset)$ wrt Δ^{snc} . By Lemma 7.19 we have that $q_0 \xrightarrow{h} q$ wrt Δ .

We distinguish two cases:

Case $q_0 \in \text{no-change}^\Delta$. Claim 2.2b shows that $q = q_0$. Since A is schema-complete for \mathbf{S} , $h \in \mathbf{S}$, and $q_0 \in I \cap \text{no-change}^\Delta$, Lemma 7.19 shows that $q_0 \xrightarrow{h} q_0$ wrt Δ . Since $q = q_0$ this yields $h \in \mathcal{L}(A)$.

Case $q_0 \notin \text{no-change}^\Delta$. Since A is schema-complete for \mathbf{S} and $h \in \mathbf{S}$ there exist no blocking runs on h that start in q_0 . Therefore, we can apply Claim 2.3b with $P = \emptyset$ to $(q_0, \emptyset) \xrightarrow{h} (q, \emptyset)$ wrt Δ^{snc} . This shows that $q_0 \xrightarrow{h} q$ wrt Δ and hence $h \in \mathcal{L}(A)$.

This ends the proof of the inverse inclusion, and thus of $\mathcal{L}(A) = \mathcal{L}(A^{snc})$. \square

The projecting in-memory evaluator of A^{snc} will be more efficient than that non-projecting evaluator of A . Note, however, that the size of A^{snc} may be exponentially bigger than that of A . Therefore, for evaluating a dSHA A with subhedge projection on a given hedge h , we may prefer to only compute the needed part of A^{snc} on the fly. This part has size $O(|h|)$ and can be computed in time $O(|A| |h|)$, so the exponential preprocessing time is avoided.

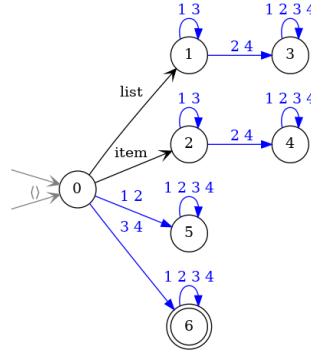


Figure 7.3: A schema-complete dSHA for the XPath filter [child-item]. It is a counter example for the completeness of safe-no-change projection, see Figure 7.4.

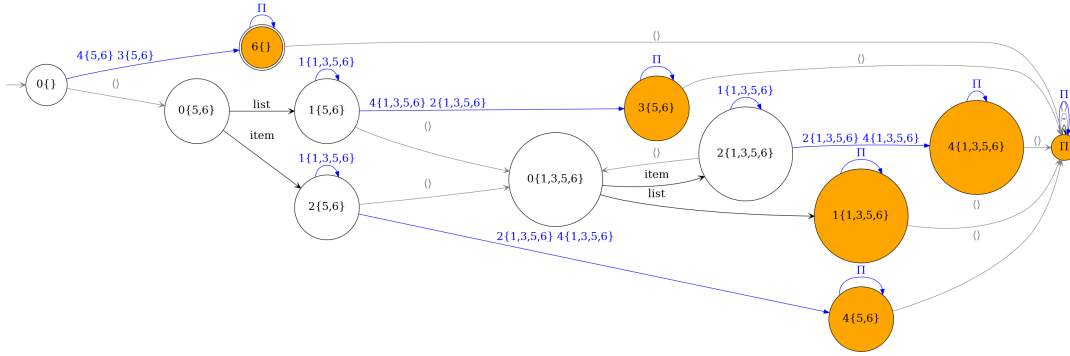


Figure 7.4: The safe-no-change projection A^{smc} of the dSHA A in Figure 7.3. It is incomplete for subhedge projection with schema $\llbracket N\text{-List} \rrbracket$ at the state $(2, \{1, 3, 5, 6\})$: this state is not a subhedge projection state, even though the prefixes $\langle \text{list} \cdot \langle \text{item}$ and $\langle \text{item} \cdot \langle \text{item}$ leading to it are subhedge irrelevant.

7.3.3 Incompleteness

Safe-no-change projection may be incomplete for subhedge projection, so that not all prefixes that are strongly subhedge irrelevant are mapped to subhedge projection states. This is shown by the counter example dSHA A for the XPath filter [child-item] in Figure 7.3 with $no\text{-}change^\Delta = \{3, 4, 5, 6\}$. More precisely, the counter example is for the following nested regular expression:

$$\text{child-item} =_{\text{def}} \langle (\text{list} + \text{item}) \cdot N\text{-List} \cdot \langle \text{item} \cdot N\text{-List} \rangle \cdot N\text{-List} \rangle \cdot N\text{-List}$$

Now the first tree of the hedge is labeled by *list* or *item*. The label of the first tree is eventually followed by some subtree that is labeled by *item*.

The safe-no-change projection A^{smc} is given in Figure 7.4. Note that the prefix $u = \langle \text{item} \langle \text{item}$ has the class $[u]_{\mathcal{S}}^{\mathcal{L}(A)} = \langle \text{item} \langle \text{item} \cdot \mathcal{N}_{\Sigma}$, which is subhedge irrelevant,

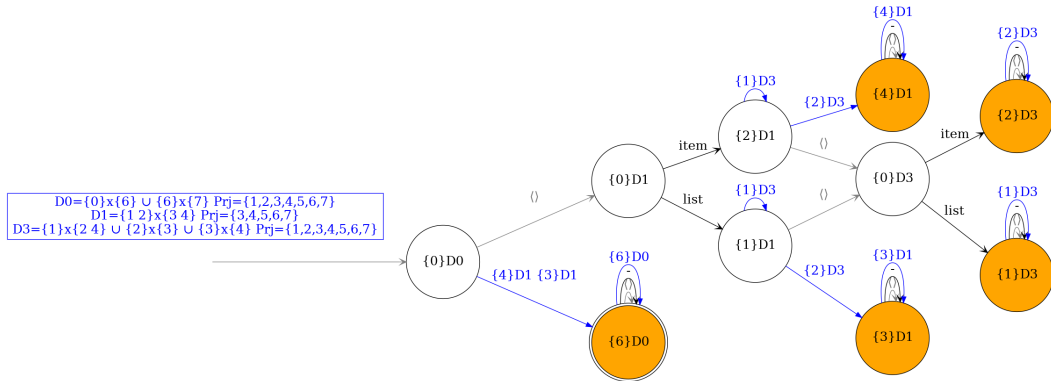


Figure 7.5: The congruence projection $d\text{SHA} \downarrow A^{cgr}(\llbracket \text{N-List} \rrbracket)$ for the counter example of the completeness of safe-no-change projection, i.e., the dSHA A for the filter $[\text{child-item}]$ in Figure 7.3 with the schema final states $F_S = \{0, 5, 6\}$. Note that state $\{1\}D3$ that is reached over the prefix $\langle \text{list} \cdot \langle \text{item}$ and the state $\{2\}D3$ that is reached over the prefix $\langle \text{item} \cdot \langle \text{item}$ are subhedge irrelevant and thus subhedge projection states.

so u is strongly subhedge irrelevant for the language $\llbracket \text{child-item} \rrbracket$ wrt schema N-List . Nevertheless, it leads to the state $(2, \{1, 3, 5, 6\})$ which is not a subhedge projection state since $2 \notin \{1, 3, 4, 5, 6\}$. The problem is that this state can still change to $(4, \{1, 3, 5, 6\})$. This state is somehow equivalent with respect to the filter but not equal to $(2, \{1, 3, 5, 6\})$.

Another incompleteness problem should be mentioned: Safe-no-change projection is sensitive to automata completion as noticed already earlier in Example 3.44. This is because a state may belong to no-change^Δ before completion but not any more after adding a sink. Nevertheless, such states never change on any tree satisfying the schema. This problem applies, for instance, to example dSHA in Figure 3.10 for XPath query $[\text{self-list-child-item}]$. Therefore, it was important to assume only schema-completeness for safe-no-change projection and not to impose full completeness.

7.4 Congruence Projection

We present the congruence projection algorithm for detecting irrelevant subhedges for regular hedge patterns with regular schema restrictions. We prove that congruence projection is complete for subhedge projection, so resolving the incompleteness of safe-no-change projection. For this, congruence projection may no more ignore the schema, so we have to assume that the input schema is regular too.

7.4.1 Motivation

Our starting motivation for congruence projection was to resolve the counter example for the completeness of safe-no-change projection in Figure 7.4. Meanwhile, we noticed that the incompleteness of safe-no-change projection does equally matter for efficiency in many practical examples.

In the counter example for the completeness of safe-no-change projection in Figure 7.4, some state is changed when processing an irrelevant subhedge. This state change, however, moves to a somehow “equivalent” state. So what one would like to detect is whether a state always remains equivalent – rather than unchanged – when processing some irrelevant subhedge. The obvious question is then which equivalence relation to choose?

7.4.2 Approach

Suppose that we want to test whether some hedge satisfying a regular schema S belongs to a language L . In the restricted case of words without schema restrictions, the idea is to use Myhill-Nerode’s congruence $\overline{\text{diff}}_{\Sigma^*}^L$, except that prefixes need to be mapped to states as usual.

In the general case with schemas, however, the situation becomes more complex, given that $\overline{\text{diff}}_S^L$ may fail to be an equivalence relation, as already illustrated in Example 7.7. So it may not be a congruence. In order to deal with that, the congruence projection work with difference relations such as diff_S^L directly, rather than with their complements.

Furthermore, the treatment of the subtree nesting will require to update the considered difference relation whenever moving down into some subtree of the hedge.

7.4.2.1 Difference Relations on States

We next introduce a notion of difference relations on state of a dSHA.

Definition 7.20. Let $(\Sigma, Q, \Delta, _, _)$ be a dSHA. A difference relation for Δ is a symmetric relation on states $D \subseteq Q \times Q$ such that for all $h \in \mathcal{H}_\Sigma$:

$$(q \xrightarrow{h} q' \text{ wrt } \Delta \wedge p \xrightarrow{h} p' \text{ wrt } \Delta \wedge (q', p') \in D) \Rightarrow (q, p) \in D$$

The set of all difference relations for Δ is denoted by \mathcal{D}^Δ .

We call a subset $Q \subseteq \mathcal{Q}$ *compatible* with a difference relation $D \in \mathcal{D}^\Delta$ if $Q^2 \cap D = \emptyset$. This means that no two states of Q may be different with respect to D .

Definition 7.21. Let $(\Sigma, \mathcal{Q}, \Delta, _, _)$ be a SHA and D a difference relation for Δ . A subset of states $Q \subseteq \mathcal{Q}$ is called *subhedge irrelevant* for D wrt Δ if $\text{acc}^\Delta(Q)$ is compatible with D . The set of all subhedge irrelevant subsets of states for D wrt. Δ thus is:

$$dPrj^\Delta(D) = \{Q \subseteq \mathcal{Q} \mid \text{acc}^\Delta(Q)^2 \cap D = \emptyset\}$$

We consider subhedge irrelevance for subsets of states since the congruence projection algorithm has to eliminate the nondeterminism that it introduces by a kind of SHA determinization. Determinization is necessary in order to recognize subhedge irrelevant prefixes properly: all single states in a subset may be subhedge irrelevant while the whole subset is not.

A single state $q \in \mathcal{Q}$ is called *subhedge irrelevant* for D if the singleton $\{q\}$ is subhedge irrelevant for D . The set of all subhedge irrelevant single states of \mathcal{Q} is denoted by $sPrj^\Delta(D)$. In examples where no determinization is needed, it will be sufficient to consider subhedge irrelevant single states.

7.4.2.2 Least Difference Relations

For any binary relation $R \subseteq \mathcal{Q} \times \mathcal{Q}$ let $ldr^\Delta(R)$ be the least difference relation on states that contains R .

Lemma 7.22. $(p_1, p_2) \in ldr^\Delta(R) \Leftrightarrow \exists (q_1, q_2) \in R. \exists h \in \mathcal{H}_\Sigma. p_1 \xrightarrow{h} q_1 \text{ wrt } \Delta \wedge p_2 \xrightarrow{h} q_2 \text{ wrt } \Delta$.

Proof. The set $\{(p_1, p_2) \in \mathcal{Q}^2 \mid \exists (q_1, q_2) \in R. \exists h \in \mathcal{H}_\Sigma. p_1 \xrightarrow{h} q_1 \text{ wrt } \Delta \wedge p_2 \xrightarrow{h} q_2 \text{ wrt } \Delta\}$ clearly is a difference relation that contains R and thus contains the least such difference relation $ldr^\Delta(R)$. Conversely, each pair in the above set must be contained in any difference relation containing R and thus in $ldr^\Delta(R)$. \square

Lemma 7.23. For any $R \subseteq \mathcal{Q} \times \mathcal{Q}$, the difference relation $ldr^\Delta(R)$ is the value of predicate D in the least fixed point of the ground Datalog program generated by the following inference rules:

$$\frac{p, q \in \mathcal{Q}}{D(p, q) :- R(p, q).} \quad \frac{p_1 \xrightarrow{a} p_2 \text{ wrt } \Delta \quad q_1 \xrightarrow{a} q_2 \text{ wrt } \Delta}{D(p_1, q_1) :- D(p_2, q_2).}$$

$$\frac{p_1 @ p \rightarrow p_2 \text{ wrt } \Delta \quad q_1 @ q \rightarrow q_2 \text{ wrt } \Delta}{D(p_1, q_1) :- D(p_2, q_2).} \quad \frac{p, q \in \mathcal{Q}}{D(q, p) :- D(p, q).}$$

Proof. The first inference rule guarantees that $R \subseteq D$. The three later inference rules characterize difference relations $D \in \mathcal{D}^\Delta$: The second and third rules state that differences in D are propagated backwards over hedges h . This is done recursively by treating letter hedges $h = a$ by the second rule and tree hedges $h = \langle h' \rangle$ by the third rule. The fourth rule expresses the symmetry of difference relations D . So the least fixed point of the Datalog program generated by the above rules contains R and is the least relation that satisfies all axioms of difference relations, so it is $ldr^\Delta(R)$. \square

Proposition 7.24. *For any $R \subseteq \mathcal{Q} \times \mathcal{Q}$ the least difference relation $ldr^\Delta(R)$ can be computed in time $O(|A|^2)$.*

Proof. The size of the ground Datalog program computing $ldr^\Delta(R)$ from Lemma 7.23 is at most quadratic in the size of A , so its least fixed point can be computed in time $O(|A|^2)$. \square

7.4.3 Algorithm

We now define the congruence projection algorithm by a compiler from $d\text{SHA}$ and a set of schema final states to SHA^\downarrow_S , which when run on a hedge can detect all subhedge irrelevant prefixes.

7.4.3.1 Inputs and Outputs

As inputs, the congruence projection algorithm is given a $d\text{SHA}$ $A = (\Sigma, \mathcal{Q}, \Delta, I, F)$ and a set F_S with $F \subseteq F_S \subseteq \mathcal{Q}$. The $d\text{SHA}$ defines the regular language (or pattern) $L = \mathcal{L}(A)$, while the regular schema S is recognized by the $d\text{SHA}$ $A[F/F_S] = (\Sigma, \mathcal{Q}, \Delta, I, F_S)$. So the same automaton A up to a change of final states – F versus F_S – defines both the language and the schema.

Example 7.25. *In the example $d\text{SHA}$ in Figure 3.10 for the XPath filter [self-list-child-item] with schema $\llbracket \text{N-List} \rrbracket$, we have $F = \{4\}$ and $F_S = \{4, 5\}$. In the automaton for the counter example [child-item] for safe-no-change projection in Figure 7.3, we have $F = \{6\}$ and $F_S = \{5, 6\}$.*

We note that if L and \mathbf{S} were given by independent SHAs, then we can obtain a common dSHA A and a set $F_{\mathbf{S}}$ as above by computing the product of the dSHAs for L and \mathbf{S} and completing it. As noticed in [Niehren *et al.* 2022a], it may be more efficient to determinize the SHA for \mathbf{S} in a first step, build the product of the SHA for L with the dSHA for \mathbf{S} in a second, and determinize this product in the third step.

The congruence projection of A wrt. $F_{\mathbf{S}}$ will maintain in its current configuration a subset of states $Q \subseteq \mathcal{Q}$ and a difference relation on states in $D \in \mathcal{D}^{\Delta}$. Thereby the congruence projection $d\text{SHA}^{\downarrow}$ can always detect whether the current prefix is subhedge irrelevant for L wrt. schema \mathbf{S} , by testing whether the current set of states Q is subhedge irrelevant for the current difference relation D .

7.4.3.2 Initial Difference Relation

The initial difference relation on states $D_{init}^{\Delta, F, F_{\mathbf{S}}} \in \mathcal{D}^{\Delta}$ is induced from the difference relation on prefixes $\text{diff}_{\mathbf{S}}^L$ as follows:

$$D_{init}^{\Delta, F, F_{\mathbf{S}}} = \{(q', q'') \mid \exists (v', v'') \in \text{diff}_{\mathbf{S}}^L. \exists q_0 \in I. q_0 \xrightarrow{\text{hdg}(v')} q' \text{ wrt } \Delta \wedge q_0 \xrightarrow{\text{hdg}(v'')} q'' \text{ wrt } \Delta\}$$

The next lemma indicates how $D_{init}^{\Delta, F, F_{\mathbf{S}}}$ can be defined from A and $F_{\mathbf{S}}$ without reference to the languages $L = \mathcal{L}(A)$ and $\mathbf{S} = \mathcal{L}(A[F/F_{\mathbf{S}}])$.

Lemma 7.26. $D_{init}^{\Delta, F, F_{\mathbf{S}}} = \text{ldr}^{\Delta}(F \times (F_{\mathbf{S}} \setminus F)) \cap \text{acc}^{\Delta}(I)^2$.

Proof. For the one direction let $(p', p'') \in D_{init}^{\Delta, F, F_{\mathbf{S}}}$. Then there exist nested words $(v', v'') \in \text{diff}_{\mathbf{S}}^L$ and an initial state $q_0 \in I$ such that $q_0 \xrightarrow{\text{hdg}(v')} p' \text{ wrt } \Delta$ and $q_0 \xrightarrow{\text{hdg}(v'')} p'' \text{ wrt } \Delta$. Since v', v'' are nested words, hedge accessibility $(p', p'') \in \text{acc}^{\Delta}(I)^2$ follows. Furthermore, $(v', v'') \in \text{diff}_{\mathbf{S}}^L$ requires the existence of a hedge $h \in \mathcal{H}_{\Sigma}$ such that $\text{hdg}(v') \cdot h \in L$ and $\text{hdg}(v'') \cdot h \in \mathbf{S} \setminus L$. Hence there are states $q' \in F$ and $q'' \in F_{\mathbf{S}} \setminus F$ such that $p' \xrightarrow{h} q'$ and $p'' \xrightarrow{h} q''$. By Lemma 7.22 this implies that $(p', p'') \in \text{ldr}^{\Delta}(F \times (F_{\mathbf{S}} \setminus F))$.

For the other direction let $(p', p'') \in \text{ldr}^{\Delta}(F \times (F_{\mathbf{S}} \setminus F)) \cap \text{acc}^{\Delta}(I)^2$. By Lemma 7.22, property $(p', p'') \in \text{ldr}^{\Delta}(F \times (F_{\mathbf{S}} \setminus F))$ shows that there exist states $q' \in F$ and $q'' \in F_{\mathbf{S}} \setminus F$ and $h \in \mathcal{H}_{\Sigma}$ such that $p' \xrightarrow{h} q' \text{ wrt } \Delta$ and $p'' \xrightarrow{h} q'' \text{ wrt } \Delta$. From $(p', p'') \in \text{acc}^{\Delta}(I)^2$ it follows that there are nested words $v', v'' \in \mathcal{N}_{\Sigma}$ and an initial state $q_0 \in I$ such that $q_0 \xrightarrow{\text{hdg}(v')} p' \text{ wrt } \Delta$ and $q_0 \xrightarrow{\text{hdg}(v'')} p'' \text{ wrt } \Delta$. Hence $(v', v'') \in \text{diff}_{\mathbf{S}}^L$, so that $(q', q'') \in D_{init}^{\Delta, F, F_{\mathbf{S}}}$. \square

As a consequence of Lemma 7.26 and Proposition 7.24, the initial difference relation $D_{init}^{\Delta, F, F_{\mathbf{S}}}$ can be computed in time $O(|A|^2)$ from A and $F_{\mathbf{S}}$.

Proposition 7.27 (Soundness of the initial difference relation). *Let $A = (\mathcal{Q}, \Sigma, \Delta, I, F)$ be a complete d_{SHA} , $F \subseteq F_S \subseteq \mathcal{Q}$, $L = \mathcal{L}(A)$ and $S = \mathcal{L}(A[F/F_S])$. For any hedge $h \in \mathcal{H}_\Sigma$ and state $q \in \mathcal{Q}$ such that $q_0 \xrightarrow{h} q$ wrt Δ for some $q_0 \in I$, the nested word $nw(h)$ is subhedge irrelevant for L and S if and only if q is subhedge irrelevant for $D_{init}^{\Delta, F, F_S}$ wrt. Δ .*

Proof. We show that $nw(h)$ is subhedge relevant for L and S if and only if q is subhedge relevant for $D_{init}^{\Delta, F, F_S}$ wrt. Δ .

“ \Rightarrow ” Let $nw(h)$ be subhedge relevant for L and S . Then there exist hedges $h', h'' \in \mathcal{H}_\Sigma$ and a nested word $w \in \mathcal{N}_\Sigma$ such that:

$$nw(h) \cdot nw(h') \cdot w \in L \wedge nw(h) \cdot nw(h'') \cdot w \in S \setminus L$$

Since A is deterministic and $q_0 \xrightarrow{h} q$ wrt Δ for the unique $q_0 \in I$ it follows that there exist states $p', p'' \in \mathcal{Q}$, $q' \in F$ and $q'' \in F_S \setminus F$ such that:

$$q \xrightarrow{h'} p' \xrightarrow{hdg(w)} q' \text{ wrt } \Delta \wedge q \xrightarrow{h''} p'' \xrightarrow{hdg(w)} q'' \text{ wrt } \Delta$$

From $q' \in F$ and $q'' \in F_S \setminus F$ it follows that $(q', q'') \in D_{init}^{\Delta, F, F_S}$. Since $D_{init}^{\Delta, F, F_S}$ is a difference relation, $p' \xrightarrow{hdg(w)} q' \text{ wrt } \Delta$ and $p'' \xrightarrow{hdg(w)} q'' \text{ wrt } \Delta$, this implies that $(p', p'') \in D_{init}^{\Delta, F, F_S}$ too. Hence, $(p', p'') \in acc^\Delta(\{q\})^2 \cap D_{init}^{\Delta, F, F_S}$, i.e., q is relevant for $D_{init}^{\Delta, F, F_S}$ wrt. Δ .

“ \Leftarrow ” Let q be subhedge relevant for $D_{init}^{\Delta, F, F_S}$ wrt. Δ . Then there exist some pair $(p', p'') \in acc^\Delta(\{q\})^2 \cap D_{init}^{\Delta, F, F_S}$. So there are hedges h' and h'' such that:

$$q \xrightarrow{h'} p' \text{ wrt } \Delta \wedge q \xrightarrow{h''} p'' \text{ wrt } \Delta$$

Since $D_{init}^{\Delta, F, F_S} = ldr^\Delta(F \times (F_S \setminus F))$ by Lemma 7.26, there exist a nested word w and a pair $(q', q'') \in \mathcal{Q}^2$ so that either (q', q'') or (q'', q') belongs to $F \times (F_S \setminus F)$ and:

$$p' \xrightarrow{hdg(w)} q' \text{ wrt } \Delta \wedge p'' \xrightarrow{hdg(w)} q'' \text{ wrt } \Delta$$

Hence $nw(h) \cdot nw(h') \cdot w \in L$ and $nw(h) \cdot nw(h'') \cdot w \in S \setminus L$ or vice versa. This shows that $nw(h)$ is relevant for L and S .

□

7.4.3.3 Updating the Difference Relation

For any difference relation $D \subseteq \mathcal{D}^\Delta$ and subset of states $Q \subseteq \mathcal{Q}$, we define a binary relation $down_Q^\Delta(D) \subseteq \mathcal{Q} \times \mathcal{Q}$ such that for all states $p_1, p_2 \in Q$:

$$(p_1, p_2) \in down_Q^\Delta(D) \text{ iff } \exists q_1, q_2 \in Q. (q_1 @^\Delta p_1, q_2 @^\Delta p_2) \in D$$

For any subset of states $Q \subseteq \mathcal{Q}$ and difference relation $D \in \mathcal{D}^\Delta$ let $D_Q \in \mathcal{D}^\Delta$ be the least difference relation that contains $down_Q^\Delta(D)$:

$$D_Q = ldr^\Delta(down_Q^\Delta(D))$$

Lemma 7.28. *The difference relation D_Q can be computed in time $O(|A|^2)$ from Q , Δ , and D .*

Proof. The binary relation $down_Q^\Delta(D)$ can be computed in time $O(|\Delta|^2)$ from Q , D , and Δ . And the least difference relation $ldr^\Delta(down_Q^\Delta(D))$ can be computed by ground datalog in time $O(|A|^2)$ from $down_Q^\Delta(D)$ by Proposition 7.24. \square

Example 7.29. *Reconsider the $dSHA$ for the $XPATH$ filter [self-list-child-item] in Figure 3.10 with the set of schema-final states $F_S = \{4, 5\}$. Since $F = \{4\}$, we have $F_S \setminus F = \{5\}$. The initial difference relation is $D_0 = D_{init}^{\Delta, F, F_S}$ is the symmetric closure of $(\{0\} \times \{1, 4, 5\}) \cup \{(1, 4), (4, 5)\}$. The subhedge irrelevant single states are $sPrj^\Delta(D_0) = \{1, 2, 3, 4, 5\}$ since only state 0 can access two states in the difference relation D_0 , the final state in $F = \{4\}$ and a non-final state that is schema-final in $F_S \setminus F = \{5\}$ of some hedges. The difference relation $down_{\{0\}}^\Delta(D_{init}^{\Delta, F, F_S})$ is the symmetric closure of $\{1, 2\} \times \{3\}$ which is equal to the difference relation $D_1 = D_{0\{0\}} = ldr^\Delta(down_{\{0\}}^\Delta(D_0))$. Hence, the subhedge irrelevant single states are $sPrj^\Delta(D_1) = \{2, 3, 4, 5\}$ since from states 0 and 1 one can still reach both 1 and 3 while $(1, 3) \in D_1$. The difference relation $D_2 = down_{\{1\}}^\Delta(D_1)$ is the symmetric closure of $\{(1, 2), (2, 3)\}$. Hence, the subhedge irrelevant single states are $sPrj^\Delta(D_2) = \{1, 2, 3, 4, 5\}$. From state 1, in particular one can only reach the states 1 and 3 which are not different for D_2 .*

Projection states for the initial difference relation contain all states that are safe for selection or safe for rejection with respect to the schema:

Lemma 7.30. *All states in $safe^\Delta(F)$ and $safe^\Delta(F_S \setminus F)$ are subhedge irrelevant for $D_{init}^{\Delta, F, F_S}$.*

We omit the proof since the result is not used later on.

$$\begin{array}{c}
\frac{a \in \Sigma \quad Q \in dPrj^\Delta(D)}{(Q, D) \xrightarrow{a} (Q, D) \in \Delta^{cgr}} \quad \frac{Q \in dPrj^\Delta(D)}{(Q, D) \xrightarrow{\langle \rangle} (Q, D) \in \Delta^{cgr}} \\
\\
\frac{Q @ P \rightarrow Q' \in \Delta^{det} \quad Q \in dPrj^\Delta(D)}{(Q, D) @ (Q, D) \rightarrow (Q, D) \in \Delta^{cgr}} \\
\\
\frac{Q \xrightarrow{a} Q' \in \Delta^{det} \quad Q \notin dPrj^\Delta(D)}{(Q, D) \xrightarrow{a} (Q', D) \in \Delta^{cgr}} \quad \frac{Q \notin dPrj^\Delta(D)}{(Q, D) \xrightarrow{\langle \rangle} (\langle \rangle^\Delta, D_Q) \in \Delta^{cgr}} \\
\\
\frac{Q @ acc^\Delta(P) \rightarrow Q' \in \Delta^{det} \quad Q \notin dPrj^\Delta(D) \quad P \in dPrj^\Delta(D_Q)}{(Q, D) @ (P, D_Q) \rightarrow (Q', D) \in \Delta^{cgr}} \\
\\
\frac{Q @ P \rightarrow Q' \in \Delta^{det} \quad Q \notin dPrj^\Delta(D) \quad P \notin dPrj^\Delta(D_Q)}{(Q, D) @ (P, D_Q) \rightarrow (Q', D) \in \Delta^{cgr}}
\end{array}$$

Figure 7.6: The transitions rules Δ^{cgr} of the congruence projection $A^{cgr(\mathbf{S})}$.

Given a $d\text{SHA}$ $A = (\Sigma, \mathcal{Q}, \Delta, I, F)$ and a set $F \subseteq F_{\mathbf{S}} \subseteq \mathcal{Q}$, we now construct the congruence projection $A^{cgr(\mathbf{S})}$ as the following $d\text{SHA}^\downarrow$:

$$A^{cgr(\mathbf{S})} = (\Sigma, \mathcal{Q}^{cgr}, \Delta^{cgr}, I^{cgr(\mathbf{S})}, F^{cgr(\mathbf{S})})$$

where the set of states, initial states, and final states are:

$$\begin{aligned}
\mathcal{Q}^{cgr} &\subseteq 2^{\mathcal{Q}} \times \mathcal{D}^\Delta \\
I^{cgr(\mathbf{S})} &= \{(I, D_{init}^{\Delta, F, F_{\mathbf{S}}})\} \\
F^{cgr(\mathbf{S})} &= \left\{ (Q, D_{init}^{\Delta, F, F_{\mathbf{S}}}) \mid \left(\begin{array}{l} Q \notin dPrj^\Delta(D_{init}^{\Delta, F, F_{\mathbf{S}}}) \Rightarrow Q \cap F \neq \emptyset \quad \wedge \\ Q \in dPrj^\Delta(D_{init}^{\Delta, F, F_{\mathbf{S}}}) \Rightarrow acc^\Delta(Q) \cap F \neq \emptyset \end{array} \right) \right\}
\end{aligned}$$

The transition rules in Δ^{cgr} are given by the inference rules in Figure 7.6.

For illustrating the construction and why determinization is needed, we reconsider Example 7.3, where schema restrictions have consequences that at first sight may be counter intuitive.

Example 7.31 (Determinization during congruence projection is needed). *Reconsider Example 7.3 with signature $\Sigma = \{a\}$, pattern $L = \{\langle a \rangle\}$ and schema $\mathbf{S} = L \cup \{\langle \rangle \cdot a\}$. This language can be defined by the $d\text{SHA}$ A in Figure 7.7. The schema \mathbf{S} can be defined by the same automaton but with the schema-final states $F_{\mathbf{S}} = \{3, 5\}$ and $F = \{3\}$. The $d\text{SHA}$ $A^{cgr(\mathbf{S})}$ produced by congruence projection is shown in Figure 7.8. The unique hedge $\langle a \rangle$*

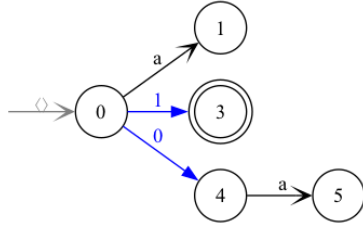


Figure 7.7: A dSHA A for $\mathcal{L}(A) = \{\langle a \rangle\}$ and schema-final states $F_S = \{3, 5\}$ for the schema $S = \mathcal{L}(A) \cup \{\langle \rangle \cdot a\}$.

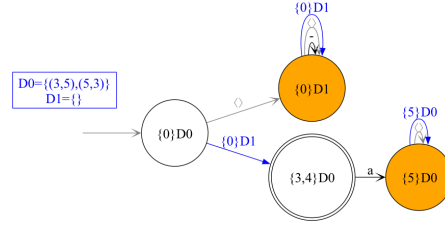


Figure 7.8: The congruence projection $A^{cgr(S)}$ for the dSHA A and the schema-final states F_S in Figure 7.7. It is equal to the $dSHA^\downarrow$ in Example 7.12 up to re-naming of states.

corps/projection//counter-det/project/dot/det-stepwise.png

of the language $L = \mathcal{L}(A)$ is accepted in state $(\{3, 4\}, D0)$, where $D0 = \{(3, 5), (5, 3)\}$. The unique hedge $\langle \rangle \cdot a$ in $S \setminus L$ is rejected: after reading the tree $\langle \rangle$, the automaton goes to state $(\{3, 4\}, D0)$ where it goes to a projecting sink $(\{5\}, D0)$ when reading the subsequent letter a . We note that any hedge in $\langle \Sigma^* \rangle$ is accepted in state $(\{3, 4\}, D0)$ by $A^{cgr(S)}$, even though only the single hedge $\langle a \rangle$ belongs to L . This is sound given that the hedges in $\langle (\epsilon + a \cdot a \cdot a^*) \rangle$ do not belong to schema S anyway. We notice that $(\{3, 4\}, D0)$ cannot be a projection state, since the run on hedge $\langle \rangle \cdot a$ must continue to the sink $(\{5\}, D0)$, so state $(\{3, 4\}, D0)$ is subhedge relevant.

Note also that both individual states 3 and 4 are subhedge irrelevant for $D0$ while the subset with both states $\{3, 4\}$ is subhedge relevant for $D0$, since $(3, 5) \in acc^\Delta(\{3, 4\})^2 \cap D0$. This shows that the determinizing construction is indeed needed to decide subhedge irrelevance for cases such as in this example. Also notice that state 0 is subhedge irrelevant for $D1 = \emptyset$. This is fine, since the acceptance of hedges of the form $\langle h \rangle \cdot h'$ by $A^{cgr(S)}$ does indeed not depend on h . By contrast, it depends on h' , so that the subset of states $\{3, 4\}$ cannot be subhedge irrelevant for $D0$.

Finally, let us discuss how the transition rules of $dSHA^\downarrow A^{cgr(S)}$ are inferred. The most interesting transition rule is $(\{0\}, D0) @ (\{0\}, D1) \rightarrow (\{3, 4\}, D0)$ in Δ^{cgr} . It is produced by the following inference rule where $Q = P = \{0\}$, $acc^\Delta(P) = \{0, 1, 3, 4, 5\}$, $Q' = \{3, 4\}$, $D = D0$, and $D_Q = D1$:

$$\frac{Q @ acc^\Delta(P) \rightarrow Q' \in \Delta^{det} \quad Q \notin dPrj^\Delta(D) \quad P \in dPrj^\Delta(D_Q)}{(Q, D) @ (P, D_Q) \rightarrow (Q', D) \in \Delta^{cgr}}$$

This rules states that if P is subhedge irrelevant for D_Q then all states accessible from P must be tried out, since all of them could be reached by some different subhedge that

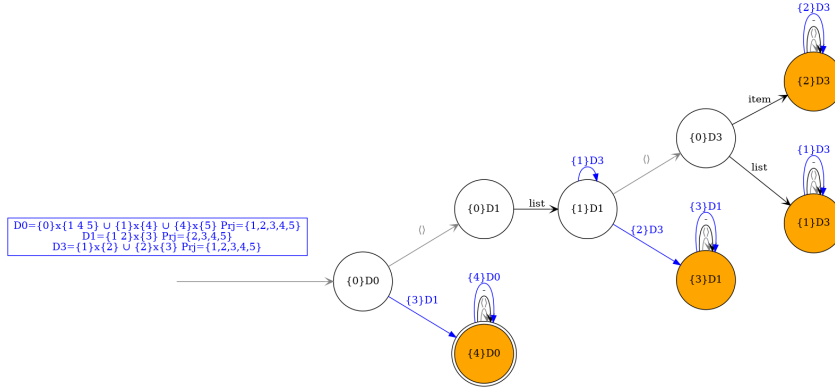


Figure 7.9: The congruence projection $A^{cgr}(\llbracket N\text{-List} \rrbracket)$ of the dSHA A in Figure 3.10 for the XPATH-like filter [self-list-child-item] with schema $\llbracket N\text{-List} \rrbracket$. Subhedge irrelevant states are colored in orange. The underscore stands for any label, either list or item.

got projected away. So one cannot know due to subhedge projection, into which state of Q' one should go. In order to stay deterministic, we thus go into all possible states, i.e., into the subset Q' . In the example, these are all the states that can be reached when reading some hedge in the pattern $\{\langle h \rangle \mid h \in \mathcal{H}_\Sigma\}$, given that the subhedges $h \in \Sigma$ are not inspected with subhedge projection.

Example 7.32 (XPATH filter [self-list-child-item] with schema $\llbracket N\text{-List} \rrbracket$). The congruence projection of the example dSHA in Figure 3.10 is given in Figure 7.9. This dSHA is similar the safe-no-change projection in Figure 3.15, except that the useless state $2'$ is removed and that the four projection states are now looping in themselves, rather than going to a shared looping state Π . It should also be noticed that only singleton state sets are used there, so no determinization is needed there. As we will see, this is typical for experiments on regular XPATH queries where larger subsets do rarely occur.

Example 7.33 (Counter example for completeness of safe-no-change projection). Reconsider the counter example for the completeness of safe-no-change projection, i.e., the dSHA in Figure 7.3 for the XPATH query [child-item] with schema final states $F_S = \{0, 5, 6\}$. Its congruence projection is shown in Figure 7.5. We note that the prefix $\langle \text{item} \rangle$ leads there to the state $\{2\}D3$, which is a subhedge projection state since 2 is subhedge irrelevant for $D3$. In particular, note that $\text{acc}^\Delta(\{2\}) = \{2, 4\}$, so no two states accessible from 2 are different in $D3$. This means that the state 2 may still be changed to 4 but then it does not become different with respect to $D3$. This resolves the incompleteness issue with the safe-no-change projection on this example.

The first property for states (Q, D) assigned by the congruence projection is that

Q is compatible with D . This means that no two states in Q are different with respect to D . Intuitively, each state of Q is as good as each other except for leading out of the schema. So if (Q, D) is assigned to some prefix, and some suffix leads from some state in Q to F , then it cannot lead to $F_S \setminus F$ from some other state in Q .

Lemma 7.34. *If some partial run of $A^{cgr(S)}$ assigns a state (Q, D) to some prefix then Q is compatible with D .*

We omit the proof since this instructive result will not be used directly later on. Still compatibility will play an important role in the soundness proof.

7.4.4 Soundness

We next adapt the soundness result and proof from safe-no-change projection to congruence projection.

Theorem 3 (Soundness of Congruence Projection). *For any $dSHA$ $A = (\mathcal{Q}, \Sigma, \Delta, I, F)$ and set $F \subseteq F_S \subseteq \mathcal{Q}$ the congruence projection of A with respect to F_S preserves the language of A within schema $S = \mathcal{L}(A[F/F_S])$, i.e.:*

$$\mathcal{L}(A^{cgr(S)}) \cap S = \mathcal{L}(A)$$

We note that S is a schema for A since $F \subseteq F_S$. Furthermore, A is schema-complete for S since A is deterministic and $S = \mathcal{L}(A[F/F_S])$.

Proof. The proof of the inclusion $\mathcal{L}(A) \subseteq \mathcal{L}(A^{cgr(S)}) \cap S$ will be based on the following two claims:

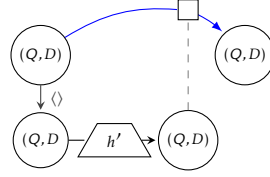
Claim 3.1a. For all $h \in \mathcal{H}_\Sigma$, $D \in \mathcal{D}^\Delta$, and $Q \in dPrj^\Delta(D)$: $(Q, D) \xrightarrow{h} (Q, D)$ wrt Δ^{cgr} .

We prove Claim 3.1a by induction on the structure of h . Suppose that $Q \in dPrj^\Delta(D)$.

Case $h = \langle h' \rangle$. The induction hypothesis applied to h' yields $(Q, D) \xrightarrow{h'} (Q, D)$ wrt Δ^{cgr} . We can thus apply the following two inference rules:

$$\frac{Q \in dPrj^\Delta(D)}{(Q, D) \xrightarrow{\diamond} (Q, D) \in \Delta^{cgr}} \quad \frac{Q \in dPrj^\Delta(D)}{(Q, D) @ (Q, D) \rightarrow (Q, D) \in \Delta^{cgr}}$$

in order to close the following diagram with respect to Δ^{cgr} :



This proves $(Q, D) \xrightarrow{h} (Q, D)$ wrt Δ^{cgr} as required by the claim.

Case $h = a$. Since $q \in dPrj^\Delta(D)$ we can apply the inference rule:

$$\frac{a \in \Sigma \quad Q \in dPrj^\Delta(D)}{(Q, D) \xrightarrow{a} (Q, D) \in \Delta^{cgr}}$$

This proves this case of the claim.

Case $h = \epsilon$. We trivially have $(Q, D) \xrightarrow{\epsilon} (Q, D)$ wrt Δ^{cgr} .

Case $h = h' \cdot h''$. By induction hypothesis applied to h' and h'' we have: $(Q, D) \xrightarrow{h'} (Q, D)$ and $(Q, D) \xrightarrow{h''} (Q, D)$ wrt Δ^{cgr} . Hence $(Q, D) \xrightarrow{h' \cdot h''} (Q, D)$ wrt Δ^{cgr} .

This ends the proof of Claim 3.1a. The next claim is the key of the soundness proof.

For any difference relation D we define a binary relation $dep^a(D) \subseteq 2^Q \times 2^Q$ such that for any two subsets of states $Q', Q'' \subseteq Q$:

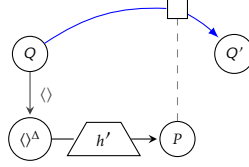
$$(Q', Q'') \in dep^a(D) \Leftrightarrow \begin{cases} Q' \notin dPrj^\Delta(D) \Rightarrow Q' \subseteq Q'' & (1^a) \\ Q' \in dPrj^\Delta(D) \Rightarrow (Q' \subseteq acc^\Delta(Q'') \wedge Q'' \in dPrj^\Delta(D)) & (2^a) \end{cases}$$

Furthermore, note that $Q'' \in dPrj^\Delta(D) \wedge Q' \subseteq acc^\Delta(Q'')$ implies $Q' \in dPrj^\Delta(D)$ and thus $(Q', Q'') \in dep^a(D)$.

Claim 3.2a. Let $h \in \mathcal{H}_\Sigma$ a hedge, $Q, Q' \subseteq Q$ subsets of states, and $D \in \mathcal{D}^\Delta$ a difference relation. If $Q \xrightarrow{h} Q'$ wrt Δ^{det} then there exists $Q'' \subseteq Q$ such that $(Q, D) \xrightarrow{h} (Q'', D)$ wrt Δ^{cgr} and $(Q', Q'') \in dep^a(D)$.

Proof. If $Q \in dPrj^\Delta(D)$ then $(Q, D) \xrightarrow{h} (Q, D)$ wrt Δ^{cgr} by Claim 3.1a. Let $Q'' = Q$. We then have $Q' \subseteq acc^\Delta(Q'')$ and $Q'' \in dPrj^\Delta(D)$ and thus $Q' \in dPrj^\Delta(D)$, so that (1^a) and (2^a) of $(Q', Q'') \in dep^a(D)$. So it is sufficient to prove the claim under the assumption that $Q \notin dPrj^\Delta(D)$. The proof is by induction on the structure of h .

Case $h = \langle h' \rangle$. The assumption $Q \xrightarrow{h} Q'$ wrt Δ^{det} shows that there exists a subset of states $P \subseteq Q$ closing the following diagram:



In particular, we have $Q@P \rightarrow Q'$ in Δ^{det} . Let $D' = D_Q$. Since $Q \notin dPrj^\Delta(D)$ we can infer:

$$\frac{Q \notin dPrj^\Delta(D)}{(Q, D) \xrightarrow{\diamond} (\langle \rangle^\Delta, D') \in \Delta^{cgr}}$$

We have to distinguish two cases depending on whether $\langle \rangle^\Delta$ belongs to $dPrj^\Delta(D')$ or not.

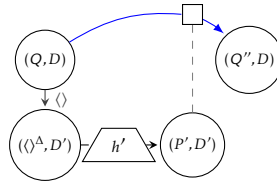
Subcase $\langle \rangle^\Delta \notin dPrj^\Delta(D')$. The induction hypothesis applied to h' yields the existence of $P' \subseteq Q$ such that $(P, P') \in dep^a(D')$ and:

$$(\langle \rangle^\Delta, D') \xrightarrow{h'} (P', D') \text{ wrt } \Delta^{cgr}$$

Subsubcase $P \notin dPrj^\Delta(D')$. Since $(P, P') \in dep^a(D')$ it follows that $P \subseteq P'$. Hence, $P' \notin dPrj^\Delta(D')$. Let $Q'' \subseteq Q$ be such that $Q@P' \rightarrow Q'' \in \Delta^{det}$. We can then apply the following inference rule:

$$\frac{Q@P' \rightarrow Q'' \in \Delta^{det} \quad Q \notin dPrj^\Delta(D) \quad P' \notin dPrj^\Delta(D')}{(Q, D)@(P', D') \rightarrow (Q'', D) \in \Delta^{cgr}}$$

Hence we can close the diagram as follows:

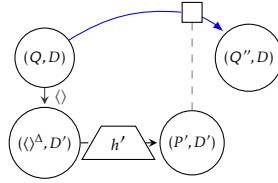


This shows that $(Q, D) \xrightarrow{h} (Q'', D)$ wrt Δ^{cgr} . Since $P \subseteq P'$, $Q@P \rightarrow Q'$ and $Q@P' \rightarrow Q'' \in \Delta^{det}$ it follows that $Q' \subseteq Q''$ and thus $(Q', Q'') \in dep^a(D)$. This shows the claim in this case.

Subsubcase $P \in dPrj^\Delta(D')$. Since $(P, P') \in dep^a(D')$ it follows that $P \subseteq acc^\Delta(P')$ and $P' \in dPrj^\Delta(D')$. Hence, we can apply the following inference rule for some $Q'' \subseteq \mathcal{H}_\Sigma$:

$$\frac{Q @ acc^\Delta(P') \rightarrow Q'' \in \Delta^{det} \quad Q \notin dPrj^\Delta(D) \quad P' \in dPrj^\Delta(D')}{(Q, D) @ (P', D') \rightarrow (Q'', D) \in \Delta^{cgr}}$$

Hence we can close the diagram as follows:

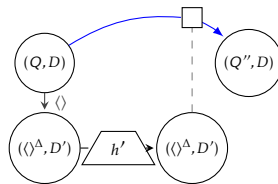


This shows that $(Q, D) \xrightarrow{h} (Q'', D)$ wrt Δ^{cgr} . Since $P \subseteq acc^\Delta(P')$, it follows from $Q @ P \rightarrow Q'$ and $Q @ acc^\Delta(P') \rightarrow Q''$ in Δ^{det} , that $Q' \subseteq Q''$. Thus $(Q', Q'') \in dep^a(D)$, so the claim holds in this case too.

Subcase $\langle \rangle^\Delta \in dPrj^\Delta(D')$. Claim 3.1a shows that $(\langle \rangle^\Delta, D') \xrightarrow{h'} (\langle \rangle^\Delta, D')$ wrt Δ^{cgr} . Let Q'' be such that $Q @ acc^\Delta(\langle \rangle^\Delta) \rightarrow Q'' \in \Delta^{det}$. We can apply the following inference rule:

$$\frac{Q @ acc^\Delta(\langle \rangle^\Delta) \rightarrow Q'' \in \Delta^{det} \quad Q \notin dPrj^\Delta(D) \quad \langle \rangle^\Delta \in dPrj^\Delta(D')}{(Q, D) @ (\langle \rangle^\Delta, D') \rightarrow (Q'', D) \in \Delta^{cgr}}$$

We can thus close the diagram below as follows with respect to Δ^{cgr} :



This shows that $(Q, D) \xrightarrow{h} (Q'', D)$ wrt Δ^{cgr} . Since $P \subseteq acc^\Delta(\langle \rangle^\Delta)$, it follows from $Q @ P \rightarrow Q'$ and $Q @ acc^\Delta(\langle \rangle^\Delta) \rightarrow Q''$ in Δ^{det} that $Q' \subseteq Q''$ and thus $(Q', Q'') \in dep^a(D)$.

Case $h = a$. Since $Q \notin dPrj^\Delta(D)$ we can apply the inference rule:

$$\frac{Q \xrightarrow{a} Q' \in \Delta^{det} \quad Q \notin dPrj^\Delta(D)}{(Q, D) \xrightarrow{a} (Q', D) \in \Delta^{cgr}}$$

Hence $(Q, D) \xrightarrow{h} (Q', D)$ wrt Δ^{cgr} . With $Q'' = Q'$, the claim follows since $(Q', Q'') \in dep^a(D)$.

Case $h = \epsilon$. In this case we have $Q = Q''$ and $(Q, D) \xrightarrow{\epsilon} (Q, D)$ wrt Δ^{cgr} , so the claim holds.

Case $h = h_1 \cdot h_2$. Since $Q \xrightarrow{h} Q'$ wrt Δ^{det} there exists $Q_1 \subseteq Q$ such that $Q \xrightarrow{h_1} Q_1$ and $Q_1 \xrightarrow{h_2} Q'$ wrt Δ^{det} . By induction hypothesis applied to h_1 there exists $Q'_1 \subseteq Q$ such that $(Q, D) \xrightarrow{h_1} (Q'_1, D)$ wrt Δ^{cgr} and $(Q_1, Q'_1) \in dep^a(D)$.

Subcase $Q_1 \in dPrj^\Delta(D)$. Since $(Q_1, Q'_1) \in dep^a(D)$ it follows that $Q_1 \subseteq acc^\Delta(Q'_1)$ and $Q'_1 \in dPrj^\Delta(D)$. Furthermore, Claim 3.1a shows that $(Q'_1, D) \xrightarrow{h_2} (Q', D)$ and hence $(Q, D) \xrightarrow{h} (Q', D)$ wrt Δ^{cgr} . Since $Q' \subseteq acc^\Delta(Q_1)$ and $Q_1 \subseteq acc^\Delta(Q'_1)$ it follows that $Q' \subseteq acc^\Delta(Q'_1)$. Hence, $(Q'_1, Q') \in dep^a(D)$ and $(Q, D) \xrightarrow{h} (Q', D)$.

Subcase $Q_1 \notin dPrj^\Delta(D)$. In this case, we can apply the induction hypothesis to $Q_1 \xrightarrow{h_2} Q'$ wrt Δ^{det} showing that there exists $Q'' \subseteq Q$ such that $(Q'_1, D) \xrightarrow{h_2} (Q'', D)$ wrt Δ^{cgr} and $(Q', Q'') \in dep^a(D)$. Hence, $(Q, D) \xrightarrow{h} (Q'', D)$ wrt Δ^{cgr} and $(Q', Q'') \in dep^a(D)$.

This ends the proof of Claim 3.2a.

Proof of inclusion $\mathcal{L}(A) \subseteq \mathcal{L}(A^{cgr(\mathbf{S})}) \cap \mathbf{S}$. Since \mathbf{S} is a schema for A , we have $\mathcal{L}(A) \subseteq \mathbf{S}$ so that it is sufficient to show $\mathcal{L}(A) \subseteq \mathcal{L}(A^{cgr(\mathbf{S})})$. Let $h \in \mathcal{L}(A)$. Then there exist $q_0 \in I$ and $q \in F$ such that $q_0 \xrightarrow{h} q$ wrt Δ . Let $Q = \{q\}$. Since A is deterministic, it follows that $I \xrightarrow{h} Q$ wrt Δ^{det} . By Claim 3.2a this implies the existence of a subset of states $Q' \in Q$ such that $(Q, Q') \in dep^a(D)$ and $(I, D_{init}^{\Delta, F, Fs}) \xrightarrow{h} (Q', D_{init}^{\Delta, F, Fs})$ wrt Δ^{cgr} . Furthermore, $(I, D_{init}^{\Delta, F, Fs}) \in I^{cgr(\mathbf{S})}$. In order to prove $h \in \mathcal{L}(A^{cgr(\mathbf{S})})$ it is thus sufficient to show that $(Q', D_{init}^{\Delta, F, Fs}) \in F^{cgr(\mathbf{S})}$. We distinguish two cases:

Case $Q' \notin dPrj^\Delta(D_{init}^{\Delta, F, Fs})$. Condition (1^a) of $(Q, Q') \in dep^a(D)$, shows that $Q \subseteq Q'$ so that $q \in Q'$. Since also $q \in F$, it follows that $Q' \cap F \neq \emptyset$. Thus, $(Q', D_{init}^{\Delta, F, Fs}) \in F^{cgr(\mathbf{S})}$, so that $h \in \mathcal{L}(A^{cgr(\mathbf{S})})$.

Case $Q' \in dPrj^\Delta(D_{init}^{\Delta, F, Fs})$. Condition (2^a) of $(Q, Q') \in dep^a(D)$ yields $Q \subseteq acc^\Delta(Q')$. Hence, $q \in acc^\Delta(Q') \cap F$, so that $(Q', D_{init}^{\Delta, F, Fs}) \in F^{cgr(\mathbf{S})}$, and thus $h \in \mathcal{L}(A^{cgr(\mathbf{S})})$.

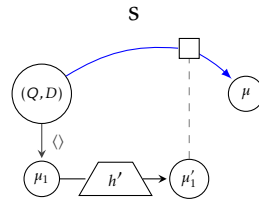
This ends the proof of the first inclusion.

We next want to show the inverse inclusion $\mathcal{L}(A^{cgr(\mathbf{S})}) \cap \mathbf{S} \subseteq \mathcal{L}(A)$. It will eventually follow from the next two claims.

Claim 3.1b. For any hedge $h \in \mathcal{H}_\Sigma$, difference relation $D \subseteq \mathcal{D}^\Delta$, projection state $Q \in dPrj^\Delta(D)$ and state $\mu \in \mathcal{Q}^{cgr}$: if $(Q, D) \xrightarrow{h} \mu$ wrt Δ^{cgr} then $\mu = (Q, D)$.

Proof. By induction on the structure of $h \in \mathcal{H}_\Sigma$. Suppose that $(Q, D) \xrightarrow{h} \mu$ wrt Δ^{cgr} .

Case $h = \langle h' \rangle$. There must exist states $\mu_1, \mu'_1 \in \mathcal{Q}^{cgr}$ closing the following diagram:



Since $Q \in dPrj^\Delta(D)$, the following rule must have been applied to infer $(Q, D) \xrightarrow{\langle \rangle} \mu_1$ wrt Δ^{cgr} :

$$\frac{Q \in dPrj^\Delta(D)}{(Q, D) \xrightarrow{\langle \rangle} (Q, D) \in \Delta^{cgr}}$$

Therefore $\mu_1 = (Q, D)$. The induction hypothesis applied to $(Q, D) \xrightarrow{h'} \mu'_1$ wrt Δ^{cgr} shows that $\mu'_1 = (Q, D)$ too. So μ must have been inferred by applying the rule:

$$\frac{Q \in dPrj^\Delta(D)}{(Q, D) @ (Q, D) \rightarrow (Q, D) \in \Delta^{cgr}}$$

Hence, $\mu = (Q, D)$ as required.

Case $h = a$. The following rule must have been applied:

$$\frac{a \in \Sigma \quad Q \in dPrj^\Delta(D)}{(Q, D) \xrightarrow{a} (Q, D) \in \Delta^{cgr}}$$

Hence, $\mu = (Q, D)$.

Case $h = \epsilon$. Obvious.

Case $h = h_1 \cdot h_2$. There must exist some μ_1 such that $(Q, D) \xrightarrow{h_1} \mu_1 \xrightarrow{h_2} \mu$ wrt Δ^{cgr} . By induction hypothesis applied to h_1 , we have $\mu_1 = (Q, D)$. We can thus apply the

induction hypothesis to h_2 to obtain $\mu_2 = (Q, D)$.

This ends the proof of Claim 3.1b.

We next need an inverse of Claim 3.2a. For relating runs A^{cgr} back to runs of A , we define for any difference relation D another binary relation $dep^b(D) \subseteq 2^Q \times 2^Q$ such that for any two subsets of states $Q', Q'' \subseteq Q$:

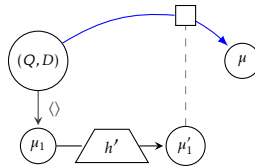
$$(Q', Q'') \in dep^b(D) \Leftrightarrow \begin{cases} (Q' \times Q'') \cap D = \emptyset & \wedge & (0^b \\ Q' \notin dPrj^\Delta(D) \Rightarrow Q'' \subseteq Q' & \wedge & (1^b \\ Q' \in dPrj^\Delta(D) \Rightarrow Q'' \subseteq acc^\Delta(Q') & (2^b \end{cases}$$

Claim 3.2b. Let $Q \in \mathcal{Q}$ be a subset of states that is compatible with a difference relation $D \in \mathcal{D}^\Delta$. For any hedge $h \in \mathcal{H}_\Sigma$ and state $\mu \in \mathcal{Q}^{cgr}$ with $(Q, D) \xrightarrow{h} \mu$ wrt Δ^{cgr} there exist a pair of subsets of states $(Q', Q'') \in dep^b(D)$ such that $\mu = (Q', D)$ and $Q \xrightarrow{h} Q''$ wrt. Δ^{det} .

Proof. If $Q \in dPrj^\Delta(D)$ then Claim 3.1b shows that $\mu = (Q, D)$. Let $Q' = Q$ and Q'' be the unique subset of states such that $Q \xrightarrow{h} Q''$ wrt. Δ^{det} . We have to show that $(Q', Q'') \in dep^b(D)$. Since $Q \xrightarrow{h} Q''$, we have $Q'' \subseteq acc^\Delta(Q')$, so condition $(2^b$ holds. Condition $(1^b$ holds trivially since $Q \in dPrj^\Delta(D)$. For condition $(0^b$, note that $Q \in dPrj^\Delta(D)$ implies $acc^\Delta(Q)^2 \cap D = \emptyset$. Furthermore, $Q' \times Q'' \subseteq acc^\Delta(Q)^2$, so that $(Q' \times Q'') \cap D = \emptyset$ as required.

We can thus assume that $Q \notin dPrj^\Delta(D)$. The proof is by induction on the structure of $h \in \mathcal{H}_\Sigma$. We distinguish all the possible forms of hedges $h \in \mathcal{H}_\Sigma$:

Case $h = \langle h' \rangle$. By definition of $(Q, P) \xrightarrow{h} \mu$ wrt Δ^{cgr} there must exist $\mu_1, \mu'_1 \in \mathcal{Q}^{cgr}$ such that the following diagram can be closed:



Since $Q \notin dPrj^\Delta(D)$, the following inference rule got applied to infer $(Q, D) \xrightarrow{\langle \rangle} \mu_1$

wrt Δ^{cgr} where $D' = D_Q$:

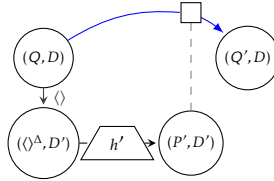
$$\frac{Q \notin dPrj^\Delta(D)}{(Q, D) \xrightarrow{\langle \rangle} (\langle \rangle^\Delta, D') \in \Delta^{cgr}}$$

Hence $\mu_1 = (\langle \rangle^\Delta, D')$. If $\langle \rangle^\Delta \notin dPrj^\Delta(D)$ then the induction hypothesis applied to $(\langle \rangle^\Delta, D') \xrightarrow{h'} \mu'_1$ wrt Δ^{cgr} show that there exists $(P', P'') \in dep^b(D')$ such that $\mu'_1 = (P', D')$ and $\langle \rangle^\Delta \xrightarrow{h'} P''$ wrt Δ^{det} . Otherwise, the same can be concluded as we argued at the beginning.

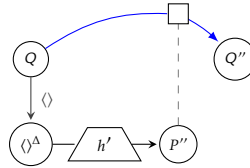
Subcase $P' \in dPrj^\Delta(D')$. The transition rule $(Q, D)@_{\mu_1} \rightarrow \mu$ must thus be inferred as follows for some $Q' \subseteq Q$:

$$\frac{Q@acc^\Delta(P') \rightarrow Q' \in \Delta^{det} \quad Q \notin dPrj^\Delta(D) \quad P' \in dPrj^\Delta(D)}{(Q, D)@_{(P', D')} \rightarrow (Q', D) \in \Delta^{cgr}}$$

This shows that $\mu = (Q', D)$. So we have the following diagram:



Let Q'' be the unique subset of states such that $Q@P'' \rightarrow Q''$ wrt Δ^{det} . We can then close the following diagram:

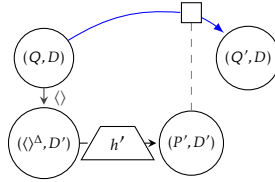


From $(P', P'') \in dep^b(D')$ it follows $(P' \times P'') \cap D' = \emptyset$, and thus $(Q' \times Q'') \cap D = \emptyset$, i.e., condition (0^b) of $(Q', Q'') \in dep^b(D)$. Since $P' \in dPrj^\Delta(D')$, condition (2^b) of $(P', P'') \in dep^b(D')$ yields $P'' \subseteq acc^\Delta(P')$. Since furthermore $Q@P'' \rightarrow Q''$ and $Q@acc^\Delta(P') \rightarrow Q'$ in Δ^{det} , this yields $Q'' \subseteq Q'$, so that conditions (1^b) and (2^b) of $(Q', Q'') \in dep^b(D)$ follow. Hence, $(Q', Q'') \in dep^b(D)$. In summary, we have show that $\mu = (Q', D)$, $Q \xrightarrow{\langle h' \rangle} Q''$ wrt Δ^{det} and $(Q', Q'') \in dep^b(D)$ as required by the claim.

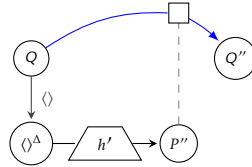
Subcase $P' \notin dPrj^\Delta(D')$. The transition rule $(Q, D)@_{\mu_1} \rightarrow \mu$ must thus be inferred as follows for some $Q' \subseteq Q$:

$$\frac{Q@P' \rightarrow Q' \in \Delta^{det} \quad Q \notin dPrj^\Delta(D) \quad P' \notin dPrj^\Delta(D)}{(Q, D)@(P', D') \rightarrow (Q', D) \in \Delta^{cgr}}$$

This shows that $\mu = (Q', D)$ and that we can close the following diagram:



Condition $(0^b$ of $(P', P'') \in dep^b(D')$ yields $(P' \times P'') \cap D' = \emptyset$. Let Q'' be the unique subset of states such that $Q@P'' \rightarrow Q''$ wrt. Δ^{det} . Since $D' \subseteq down^\Delta_Q(D)$ and $(P' \times P'') \cap D' = \emptyset$, it follows from $Q@P' \rightarrow Q'$ and $Q@P'' \rightarrow Q''$ wrt. Δ^{det} that $(Q' \cap Q'') \cap D = \emptyset$. That is condition $(0^b$ of $(Q', Q'') \in dep^b(D)$. Since $P' \notin dPrj^\Delta(D')$, condition $(1^b$ of $(P', P'') \in dep^b(D')$ is $P'' \subseteq P'$. From $Q@P'' \rightarrow Q''$ and $Q@P' \rightarrow Q'$ it thus follows that $Q'' \subseteq Q'$. Hence, conditions $(1^b$ and $(2^b$ of $(Q', Q'') \in dep^b(D)$ are valid too, so that $(Q', Q'') \in dep^b(D)$ holds. Furthermore:



This shows that $Q \xrightarrow{\langle h' \rangle} Q''$ wrt Δ . The other two requirements of the claim, $\mu = (Q', D)$ and $(Q', Q'') \in dep^b(D)$, were shown earlier.

Case $h = a$. Since $Q \notin dPrj^\Delta(D)$, the following inference rule must be used:

$$\frac{Q \xrightarrow{a} Q' \in \Delta \quad Q \notin dPrj^\Delta(D)}{(Q, D) \xrightarrow{a} (Q', D) \in \Delta^{cgr}}$$

So $\mu = (Q', D)$, $Q \xrightarrow{a} Q'$ wrt Δ^{det} . Furthermore, since Q is compatible with D , and D is a difference relation, Q' is compatible with D too. Hence $(Q' \times Q') \cap D = \emptyset$ showing condition $(0^b$ of $(Q', Q') \in dep^b(D)$. Trivially, $Q' \subseteq Q' \subseteq acc^\Delta(Q')$ so conditions $(1^b$ and $(2^b$ of $(Q', Q') \in dep^b(D)$ hold too. Hence $(Q', Q') \in dep^b(D)$.

Case $h = \epsilon$. Obvious.

Case $h = h_1 \cdot h_2$. Since $(Q, D) \xrightarrow{h} \mu$ wrt. Δ^{cgr} there exists some $\mu_1 \in \mathcal{Q}^{cgr}$ such that $(Q, D) \xrightarrow{h_1} \mu_1$ and $\mu_1 \xrightarrow{h_2} \mu$ wrt Δ^{cgr} . We can apply the induction hypothesis to h_1 . Hence there exist subsets of states $Q_1, Q'_1 \subseteq \mathcal{Q}$ such that $\mu_1 = (Q_1, D)$, $Q \xrightarrow{h_1} Q'_1$ wrt Δ , and $(Q_1, Q'_1) \in dep^b(D)$. We distinguish two cases:

Subcase $Q_1 \in dPrj^\Delta(D)$. Since $\mu_1 = (Q_1, D) \xrightarrow{h_2} \mu$ wrt Δ^{cgr} , Claim 3.1b shows in this case that $\mu = (Q_1, D)$ so that $(Q_1, D) \xrightarrow{h_2} (Q_1, D)$ wrt. Δ^{cgr} . Condition $(2^b$ of $(Q_1, Q'_1) \in dep^b(D)$ and $Q_1 \in dPrj^\Delta(D)$ imply that $Q'_1 \in acc^\Delta(Q_1)$. Let Q_2 be the unique subset of states such that $Q'_1 \xrightarrow{h_2} Q_2$ wrt. Δ^{det} . Then $Q_2 \subseteq acc^\Delta(Q'_1)$ so that $Q_2 \subseteq acc^\Delta(Q_1)$ and thus condition $(2^b$ of $(Q_1, Q_2) \in dep^b(D)$ holds. Condition $(1^b$ of $(Q_1, Q_2) \in dep^b(D)$ is trivial since $Q_1 \in dPrj^\Delta(D)$. Since $Q_1 \in dPrj^\Delta(D)$ and $Q_1 \times Q_2 \in acc^\Delta(Q)^2$ it follows that $Q_1 \times Q_2 \cap D = \emptyset$, so condition $(0^b$ of $(Q_1, Q_2) \in dep^b(D)$ holds too. Hence $Q \xrightarrow{h} Q_2$ wrt Δ^{det} and $(Q_1, Q_2) \in dep^b(D)$.

Subcase $Q_1 \notin dPrj^\Delta(D)$. Since $Q_1 \xrightarrow{h_1} Q'_1$ and Q_1 is compatible with difference relation D , it follows that Q'_1 is compatible with D too. We can thus apply the induction hypothesis to $(Q_1, D) \xrightarrow{h_2} \mu$ showing the existence of subsets of states $(Q_2, Q'_2) \in dep^b(D)$ such that $\mu = (Q_2, D)$ and $Q_1 \xrightarrow{h_2} Q'_2$ wrt Δ . So we have $Q \xrightarrow{h} Q'_2$ wrt. Δ and $(Q_2, Q'_2) \in dep^b(D)$ as required.

This ends the proof of Claim 3.2b.

Proof of inclusion $\mathcal{L}(A^{cgr(S)}) \cap \mathbf{S} \subseteq \mathcal{L}(A)$. Let $h \in \mathcal{L}(A^{cgr(S)}) \cap \mathbf{S}$. Then there exists a final subset of states $Q \in F^{cgr(S)}$ such that $(I, D_{init}^{\Delta, F, F_S}) \xrightarrow{h} (Q, D_{init}^{\Delta, F, F_S})$ wrt Δ^{cgr} . Since I is a singleton or empty, it is compatible with $D_{init}^{\Delta, F, F_S}$. Claim 3.2b thus shows that there exists a subset of states $Q' \subseteq \mathcal{Q}$ such that $I \xrightarrow{h} Q'$ wrt. Δ^{det} and $(Q, Q') \in dep^b(D_{init}^{\Delta, F, F_S})$. Condition $(0^b$ of $(Q, Q') \in dep^b(D_{init}^{\Delta, F, F_S})$ shows that $(Q, Q') \cap D_{init}^{\Delta, F, F_S} = \emptyset$. Since $h \in \mathbf{S}$ it follows that $Q' \cap F_S \neq \emptyset$. The determinism of A shows that Q' is a singleton. So there exists a state $q' \in F_S$ such that $Q' = \{q'\}$.

Case $Q \notin dPrj^\Delta(D_{init}^{\Delta, F, F_S})$. Condition $(1^b$ shows that $Q' \subseteq Q$, so that $q' \in Q$. Since $Q \in F^{cgr}$ we have $Q \cap F \neq \emptyset$. Since Q is compatible with $D_{init}^{\Delta, F, F_S}$, $q' \in Q$ and $Q \cap F \neq \emptyset$, it follows that $q' \notin F_S \setminus F$. Since $q' \in F_S$ this implies $q' \in F$ and thus $h \in \mathcal{L}(A)$.

Case $Q \in dPrj^\Delta(D_{init}^{\Delta, F, F_S})$. Condition $(2^b$ shows that $Q' \subseteq acc^\Delta(Q)$, so that $q' \in acc^\Delta(Q)$. Since $Q \in F^{cgr}$ we have $acc^\Delta(Q) \cap F \neq \emptyset$. Let $q \in acc^\Delta(Q) \cap F$ be arbitrary. Since $Q \in dPrj^\Delta(D_{init}^{\Delta, F, F_S})$, and $(q', q) \in acc^\Delta(Q)^2$ it follows that $(q, q') \notin D_{init}^{\Delta, F, F_S}$. Furthermore, $q \in F$ so that $q' \notin F_S \setminus F$. In combination with $q' \in F_S$ this implies $q' \in F_S$.

so $h \in \mathcal{L}(A)$.

This ends the proof of the inverse inclusion, and thus of $\mathcal{L}(A) = \mathcal{L}(A^{cgr(\mathbf{S})}) \cap \mathbf{S}$. \square

7.4.5 Completeness

We next show the completeness of congruence projection for subhedge projection according to Definition 7.16. Let $A = (\Sigma, \mathcal{Q}, \Delta, I, F)$ be a complete dSHA and $F \subseteq F_{\mathbf{S}} \subseteq \mathcal{Q}$. Automaton A defines the regular pattern $L = \mathcal{L}(A)$ and with the schema-final states in $F_{\mathbf{S}}$ the regular schema $\mathbf{S} = \mathcal{L}(A[F/F_{\mathbf{S}}])$. We first show that all subhedge irrelevant states of difference relations are subhedge projection states $A^{cgr(\mathbf{S})}$ according to Definition 7.9.

Lemma 7.35. *If $Q \in dPrj^{\Delta}(D)$ then (Q, D) is a subhedge projection state of Δ^{cgr} with witness (Q, D) .*

Proof. We assume that $q \in sPrj^{\Delta}(D)$ and have to show that (Q, D) is a subhedge projection state of Δ^{cgr} . We have to show that all transition rules starting with (Q, D) are permitted for a subhedge projection state (Q, D) with witness (Q, D) . They are generated by the following rules, all of which are looping:

$$\frac{a \in \Sigma \quad Q \in dPrj^{\Delta}(D)}{(Q, D) \xrightarrow{a} (Q, D) \in \Delta^{cgr}} \quad \frac{Q \in dPrj^{\Delta}(D)}{(Q, D) @ (Q, D) \rightarrow (Q, D) \in \Delta^{cgr}} \quad \frac{Q \in dPrj^{\Delta}(D)}{(Q, D) \xrightarrow{\langle \rangle} (Q, D) \in \Delta^{cgr}}$$

\square

So if a partial run of $A^{cgr(\mathbf{S})}$ on a prefix u assigns some state (P, D) with $P \in dPrj^{\Delta}(D)$, then (P, D) is a subhedge projection state by Lemma 7.35, and thus subhedge irrelevant by Proposition 7.11.

Lemma 7.36. *Let $A' = A[I/\langle \rangle^{\Delta}]$, $L' = \mathcal{L}(A')$ and $\mathbf{S}' = \mathcal{L}(A'[F/F_{\mathbf{S}}])$. If $(\langle \rangle^{\Delta}, D) \xrightarrow{hdg(u)} (Q, D)$ wrt $A^{cgr(\mathbf{S})}$ for some nested word $u \in \mathcal{N}_{\Sigma}$ and either*

- $Q \notin dPrj^{\Delta}(D)$ and $q \in Q$, or
- $Q \in dPrj^{\Delta}(D)$ and $q \in acc^{\Delta}(Q)$,

then there exists a nested word $u' \in class_{\mathbf{S}'}^{L'}(u)$ and $q_0 \in \langle \rangle^{\Delta}$ such that $q_0 \xrightarrow{hdg(u')} q$ wrt Δ .

Proof. By induction on the length of u . Suppose that $(\langle \rangle^{\Delta}, D) \xrightarrow{hdg(u)} (Q, D)$ wrt $A^{cgr(\mathbf{S})}$. In the base case, we have $u = \epsilon$. Hence $Q = \langle \rangle^{\Delta}$.

Case $Q \notin dPrj^\Delta(D)$ and $q \in Q$. Then $\langle \rangle^\Delta = \{q\}$. The unique run of A' on $u = \epsilon$ starts and ends in the tree initial state $q \in \langle \rangle^\Delta$.

Case $Q \in dPrj^\Delta(D)$ and $q \in acc^\Delta(Q)$. Since $Q \in dPrj^\Delta(D)$, Lemma 7.35 shows that (Q, D) is a subhedge projection state, so that ϵ is subhedge irrelevant for L' and S' by Proposition 7.11. Hence, $class_{S'}^{L'}(\epsilon) = \mathcal{N}_\Sigma$. Furthermore, since $q \in acc^\Delta(Q)$, there exists a hedge h and $q_0 \in I$ such that $q_0 \xrightarrow{h} q$ wrt Δ . Let $u' = nw(h)$. The run of A on u' then ends in q and $u' \in class_{S'}^{L'}(u)$.

For the induction step, we distinguish the possible forms of the nested word u . So there exist $\tilde{u}, v \in \mathcal{N}_\Sigma$ and $a \in \Sigma$ such that either of the following cases holds:

Case $u = \tilde{u} \cdot \langle \cdot v \cdot \rangle$. Let (\tilde{Q}, D) be the state in which the run of $(A')^{cgr(S)}$ on \tilde{u} ends.

Subcase $\tilde{Q} \notin dPrj^\Delta(D)$. Let $D' = D_{\tilde{Q}}$. Then there exist $P \subseteq Q$ such that $(\langle \rangle^\Delta, D') \xrightarrow{v} (P, D')$ wrt $A^{cgr(S)}$. So the run of $(A')^{cgr(S)}$ on v goes to state (P, D') .

Subsubcase $P \in dPrj^\Delta(D')$. By construction of $A^{cgr(S)}$, we then have $Q = \tilde{Q} @^\Delta acc^\Delta(P)$. Since $q \in Q$ there exist $\tilde{q} \in \tilde{Q}$ and $p \in acc^\Delta(P)$ such that $\tilde{q} @ p \rightarrow q$ in Δ . By induction hypothesis, there exists $v' \in class_{S'}^{L'}(v)$ such that the run of A' on v' ends in p . Hence the run of A on $u' = \tilde{u} \cdot \langle \cdot v' \cdot \rangle$ ends in q , and furthermore, $u' \in [u]_S^L$.

Subsubcase $P \notin dPrj^\Delta(D')$. By construction of $A^{cgr(S)}$, we then have $Q = \tilde{Q} @^\Delta P$. Since $q \in Q$ there exist $\tilde{q} \in \tilde{Q}$ and $p \in P$ such that $\tilde{q} @ p \rightarrow q$ in Δ . By induction hypothesis, there exists $v' \in class_{S'}^{L'}(v)$ such that the run of A' on v' ends in p . Hence the run of A on $u' = \tilde{u} \cdot \langle \cdot v' \cdot \rangle$ ends in q , and furthermore, $u' \in [u]_S^L$.

Subcase $\tilde{Q} \in dPrj^\Delta(D)$. Then $Q = \tilde{Q}$ and $q \in acc^\Delta(\tilde{Q})$. By induction hypothesis applied to \tilde{u} there exists $\tilde{u}' \in [\tilde{u}]_S^L$ such that $\langle \rangle^\Delta \xrightarrow{hdg(\tilde{u}')} q$. Since \tilde{u} is irrelevant for L and S' we have that $\tilde{u}' \in [u]_S^L$.

Case $u = \tilde{u} \cdot a$. Easier than the previous cases and thus omitted.

□

Lemma 7.37. Assume a partial run of $A^{cgr(S)}$ assign some state (Q, D) to a prefix $u \in \text{pref}(\mathcal{N}_\Sigma)$. If $Q \notin dPrj^\Delta(D)$ and $q \in Q$ then there exists a prefix $u' \in [u]_S^L$ such that a partial run of A assigns state q to u' .

Proof. By induction on the length of u . In the base case, we have $u = \epsilon$. The partial run of $A^{cgr(S)}$ on u assigns state (Q, D) where $Q = I$ and $D = D_{init}^{\Delta, F, F_S}$. Suppose that $Q \notin dPrj^\Delta(D)$ and $q \in Q$. Then $I = \{q\}$. The unique partial run of A on $u = \epsilon$ ends in the initial state $q \in I$. For the induction step, let $A' = A[I/\langle \rangle^\Delta]$, $L' = \mathcal{L}(A')$ and $S' = \mathcal{L}(A'[F/F_S])$. We distinguish the possible forms of u . So there exist $\tilde{u} \in \text{pref}(\mathcal{N}_\Sigma)$, $v \in \mathcal{N}_\Sigma$, and $a \in \Sigma$ such that either of the following hold:

Case $u = \tilde{u} \cdot \langle \cdot v \cdot \rangle$. Let (\tilde{Q}, \tilde{D}) be the state in which the partial run of $A^{cgr(S)}$ on \tilde{u} ends. From $Q \notin dPrj^\Delta(D)$ it follows that $\tilde{Q} \notin dPrj^\Delta(\tilde{D})$. Furthermore, $(\langle \rangle^\Delta, D) \xrightarrow{v} (Q, D)$ wrt $A^{cgr(S)}$. So the partial run of $(A')^{cgr(S)}$ on v goes to state (Q, D) . By induction hypothesis, there exists $v' \in \text{class}_{S'}^{L'}(v)$ such that the partial run of A' on v' ends in q . Hence the partial run of A on $u' = \tilde{u} \cdot \langle \cdot v' \cdot \rangle$ ends in q , and furthermore, $u' \in [u]_S^L$.

Case $u = \tilde{u} \cdot \langle \cdot v \cdot \rangle$. Let (\tilde{Q}, \tilde{D}) be the state in which the partial run of $A^{cgr(S)}$ on \tilde{u} ends. From $Q \notin dPrj^\Delta(D)$ it follows that $\tilde{Q} \notin dPrj^\Delta(\tilde{D})$. Furthermore, $(\langle \rangle^\Delta, D) \xrightarrow{v} (P, D)$ wrt $A^{cgr(S)}$ for some $P \subseteq Q$.

Subcase $P \in dPrj^\Delta(D)$. Then $Q = \tilde{Q} @^{\Delta} acc^\Delta(P)$. So there exists $\tilde{q} \in Q$ and $p \in acc^\Delta(P)$ such that $\tilde{q} @ p \rightarrow q$ in Δ . By Lemma 7.36 there exists $v' \in \text{class}_{S'}^{L'}(v)$ such that $\langle \rangle^\Delta \xrightarrow{hdg(v')} p$ wrt Δ . By induction hypothesis applied to \tilde{u} there exists an initial state $q_0 \in I$ and a prefix $\tilde{u}' \in [\tilde{u}]_S^L$ such that the partial run of A on \tilde{u}' goes to state \tilde{q} . Hence the partial run of A on $u' = \tilde{u}' \cdot \langle \cdot v' \cdot \rangle$ goes to q and $u' \in [u]_S^L$.

Subcase $P \notin dPrj^\Delta(D)$. Then $Q = \tilde{Q} @^\Delta P$. So there exists $\tilde{q} \in Q$ and $p \in P$ such that $\tilde{q} @ p \rightarrow q$ in Δ . By Lemma 7.36 there exists $v' \in \text{class}_{S'}^{L'}(v)$ such that $\langle \rangle^\Delta \xrightarrow{hdg(v')} p$ wrt Δ . By induction hypothesis applied to \tilde{u} there exists an initial state $q_0 \in I$ and a prefix $\tilde{u}' \in [\tilde{u}]_S^L$ such that the partial run of A on \tilde{u}' goes to state \tilde{q} . Hence the partial run of A on $u' = \tilde{u}' \cdot \langle \cdot v' \cdot \rangle$ goes to q and $u' \in [u]_S^L$.

Case $u = \tilde{u} \cdot a$. Easier than the previous case and thus omitted.

□

The next lemma states the key invariant of congruence projection, which eventually proves it completeness for subhedge projection. For this we define for any $F \subseteq F_S \subseteq F$, the binary relation $\approx_{F_S}^F$ as the symmetric closure of $F \times (F_S \setminus F)$, i.e., for all $(q', q'') \in \mathcal{Q}^2$:

$$q' \approx_{F_S}^F q'' \Leftrightarrow \left\{ \begin{array}{ll} (q' \in F & \wedge \quad q'' \in F_S \setminus F) \quad \vee \\ (q' \in F_S \setminus F & \wedge \quad q'' \in F) \end{array} \right.$$

Lemma 7.38 (Key Invariant). *Let $A = (\mathcal{Q}, \Sigma, \Delta, I, F)$ a $dSHA$, $F \subseteq F_S \subseteq \mathcal{Q}$, $L = \mathcal{L}(A)$ and $S = \mathcal{L}(A[F/F_S])$. If $A^{cgr(S)}$ has a partial run on prefix $u \in \text{pref}(\mathcal{N}_\Sigma)$ to state $(P, D) \in 2^\mathcal{Q} \times D^\Delta$ then for all $(p', p'') \in P^2$, $(r', r'') \in D$, and $(v', v'') \in \mathcal{N}_\Sigma^2$ such that:*

$$p' \xrightarrow{\text{hdg}(v')} r' \text{ wrt } \Delta \wedge p'' \xrightarrow{\text{hdg}(v'')} r'' \text{ wrt } \Delta$$

there exist $(u', u'') \in [u]_S^L$, $w \in \text{suffs}(\mathcal{N}_\Sigma)$, $q' \approx_{F_S}^F q''$, and $q_0 \in I$ such that:

$$q_0 \xrightarrow{\text{hdg}(u' \cdot v' \cdot w)} q' \text{ wrt } \Delta \wedge q_0 \xrightarrow{\text{hdg}(u'' \cdot v'' \cdot w)} q'' \text{ wrt } \Delta$$

Proof. By induction on the number of dangling opening parenthesis of u .

In the base case, u does not have any dangling opening parenthesis, so $u \in \mathcal{N}_\Sigma$ is a nested word. In this case, $D = D_{init}^{\Delta, F, F_S}$. So $A^{cgr(S)}$ has a partial run on nested word $u \in \mathcal{N}_\Sigma$ to $(P, D_{init}^{\Delta, F, F_S})$ where $P \subseteq \mathcal{Q}$. Let $(p', p'') \in P^2$, $(r', r'') \in D_{init}^{\Delta, F, F_S}$, and $(v', v'') \in \mathcal{N}_\Sigma^2$ such that:

$$p' \xrightarrow{\text{hdg}(v')} r' \text{ wrt } \Delta \wedge p'' \xrightarrow{\text{hdg}(v'')} r'' \text{ wrt } \Delta$$

It then follows that $P \notin dPrj^\Delta(D_{init}^{\Delta, F, F_S})$. Therefore we can apply Lemma 7.37. It shows that there exist nested words $(u', u'') \in ([u]_S^L)^2$ and $q_0 \in I$ such that:

$$q_0 \xrightarrow{\text{hdg}(u')} p' \text{ wrt } \Delta \wedge q_0 \xrightarrow{\text{hdg}(u'')} p'' \text{ wrt } \Delta$$

Since $D_{init}^{\Delta, F, F_S} \subseteq \text{ldr}^\Delta(F \times (F_S \setminus F))$ by Lemma 7.26, there exist a nested word $w \in \mathcal{N}_\Sigma$ and states $q' \approx_{F_S}^F q''$ such that:

$$r' \xrightarrow{\text{hdg}(w)} q' \text{ wrt } \Delta \wedge r'' \xrightarrow{\text{hdg}(w)} q'' \text{ wrt } \Delta$$

Then we have:

$$q_0 \xrightarrow{\text{hdg}(u' \cdot v' \cdot w)} q' \text{ wrt } \Delta \wedge q_0 \xrightarrow{\text{hdg}(u'' \cdot v'' \cdot w)} q'' \text{ wrt } \Delta$$

For the induction step let $u = u_1 \cdot \langle \cdot u_2$ for some prefix $u_1 \in \text{pref}(\mathcal{N}_\Sigma)$ and nested word $u_2 \in \mathcal{N}_\Sigma$, so that u_1 has one open dangling bracket less than u . Let $A^{cgr(S)}$ have a partial run on nested word $u \in \mathcal{N}_\Sigma$ to (P, D) . Let (P_1, D_1) be the state that this run assigns to prefix u_1 . Then $D = (D_1)_{P_1}$. Let $(p', p'') \in P^2$, $(r', r'') \in D$, and $(v', v'') \in \mathcal{N}_\Sigma^2$ such that:

$$p' \xrightarrow{\text{hdg}(v')} r' \text{ wrt } \Delta \wedge p'' \xrightarrow{\text{hdg}(v'')} r'' \text{ wrt } \Delta$$

Since $D = (D_1)_{P_1}$ there exist $(p'_1, p''_1) \in (P_1)^2$ and states $(r'_1, r''_1) \in D_1$ such that $p'_1 @ q' \rightarrow r'_1$ and $p''_1 @ q'' \rightarrow r''_1$. In particular, $P_1 \notin dPrj^\Delta(D_1)$ so that we can apply Lemma 7.37. It shows that there exist $(u'_2, u''_2) \in class_{\mathcal{S}}^{L'}(u_2)$ such that $\langle \rangle^\Delta \xrightarrow{u'_2} p'$ and $\langle \rangle^\Delta \xrightarrow{u''_2} p''$. Let $v'_1 = \langle \cdot u'_2 \cdot v' \cdot \rangle$ and $v''_1 = \langle \cdot u'_2 \cdot v'' \cdot \rangle$. Then $p'_1 \xrightarrow{hdg(v'_1)} r'_1$, $p'_1 \xrightarrow{hdg(v''_1)} r''_1$ wrt Δ . By induction hypothesis applied to u_1 – on which $A^{cgr(S)}$ has a partial run to state (P_1, D_1) such that $(p'_1, p''_1) \in (P_1)^2$, $p'_1 \xrightarrow{hdg(v'_1)} r'_1$, $p'_1 \xrightarrow{hdg(v''_1)} r''_1$, and $(r'_1, r''_1) \in D$ – there exist $(u'_1, u''_1) \in [u_1]_{\mathcal{S}}^L$, $w_1 \in suffs(\mathcal{N}_\Sigma)$, $q' \approx_{F_S}^F q''$ and $q_0 \in I$ such that:

$$q_0 \xrightarrow{hdg(u'_1 \cdot v'_1 \cdot w_1)} q' \text{ wrt } \Delta \wedge q_0 \xrightarrow{hdg(u''_1 \cdot v''_1 \cdot w_1)} q'' \text{ wrt } \Delta$$

Let $u' = u'_1 \cdot \langle \cdot u'_2$ and $u'' = u''_1 \cdot \langle \cdot u'_2$ and $w = \rangle \cdot w_1$. The above then yields:

$$q_0 \xrightarrow{hdg(u' \cdot v' \cdot w)} q' \text{ wrt } \Delta \wedge q_0 \xrightarrow{hdg(u'' \cdot v'' \cdot w)} q'' \text{ wrt } \Delta$$

Furthermore, $(u', u'') \in ([u]_{\mathcal{S}}^L)^2$, so this was to be shown. \square

Proposition 7.39. *Let $A = (\mathcal{Q}, \Sigma, \Delta, I, F)$ a $dSHA$, $F \subseteq F_S \subseteq \mathcal{Q}$, $L = \mathcal{L}(A)$ and $S = \mathcal{L}(A[F/F_S])$. If $A^{cgr(S)}$ has a partial run on prefix $u \in pref(\mathcal{N}_\Sigma)$ to some state $(P, D) \in 2^{\mathcal{Q}} \times \mathcal{D}^\Delta$ so that $P \notin dPrj^\Delta(D)$, then the set $[u]_{\mathcal{S}}^L$ is subhedge relevant for L wrt S .*

Proof. Suppose that $A^{cgr(S)}$ has a partial run on prefix $u \in pref(\mathcal{N}_\Sigma)$ to some state $(P, D) \in 2^{\mathcal{Q}} \times \mathcal{D}^\Delta$ so that $P \notin dPrj^\Delta(D)$. Since $P \notin dPrj^\Delta(D)$ there exist $acc^\Delta(P)^2 \cap D \neq \emptyset$. So there exist $(p', p'') \in P^2$, $(r', r'') \in D$, and $(v', v'') \in \mathcal{N}_\Sigma^2$ such that:

$$p' \xrightarrow{hdg(v')} r' \text{ wrt } \Delta \wedge p'' \xrightarrow{hdg(v'')} r'' \text{ wrt } \Delta$$

By Lemma 7.38 there exist $(u', u'') \in [u]_{\mathcal{S}}^L$, $w \in suffs(\mathcal{N}_\Sigma)$, $q' \approx_{F_S}^F q''$ and $q_0 \in I$ such that:

$$q_0 \xrightarrow{hdg(u' \cdot v' \cdot w)} q' \text{ wrt } \Delta \wedge q_0 \xrightarrow{hdg(u'' \cdot v'' \cdot w)} q'' \text{ wrt } \Delta$$

Hence, the set $[u]_{\mathcal{S}}^L$ is relevant for L wrt S . \square

Theorem 4 (Completeness of congruence projection). *For any complete $dSHA$ $A = (\mathcal{Q}, \Sigma, \Delta, I, F)$ and set $F \subseteq F_S \subseteq \mathcal{Q}$, the congruence projection $A^{cgr(S)}$ is sound and complete for subhedge projection for the regular pattern $\mathcal{L}(A)$ wrt the regular schema $S = \mathcal{L}(A[F_S/F])$.*

Proof. The soundness of congruence projection was shown in Theorem 3. For proving the completeness, let $u \in pref(\mathcal{N}_\Sigma)$ be a nested word prefix that is strongly

subhedge irrelevant for L wrt \mathbf{S} . By definition of strong irrelevance, the class $[u]_{\mathbf{S}}^L$ is irrelevant for L and \mathbf{S} . Let (P, D) be the unique state assigned by the partial run of $A^{cgr(\mathbf{S})}$ to prefix u . Since $[u]_{\mathbf{S}}^L$ is irrelevant for L and \mathbf{S} , Proposition 7.39 shows that $P \in dPrj^\Delta(D)$. By Lemma 7.35, (P, D) then is a subhedge projection state of $A^{cgr(\mathbf{S})}$. \square

7.4.6 Automata Sizes

We next discuss the complexity of membership testing with complete subhedge projection based on the in-memory evaluation of the input hedge by the congruence projection of the input automaton.

Lemma 7.40. *The number of states $|Q^{cgr}|$ is in $O(2^{n^2+n})$ where n is the number of states of A .*

Proof. With the deterministic construction the states of $A^{cgr(\mathbf{S})}$ are pairs in $2^Q \times \mathcal{D}^\Delta$. So, the maximal number of states of the congruence projection is $|Q^{cgr(\mathbf{S})}| = 2^{|Q|} |\mathcal{D}^\Delta| \leq 2^n 2^{n^2} = 2^{n^2+n}$. \square

Complete Suffix Projection

Abstract

Earliest $d\text{SHA}^\downarrow$ s detect language membership and non-membership at the earliest prefix where it becomes certain. A SHA^\downarrow enjoys complete suffix projection, so that its evaluator ignores the largest suffix of the input hedge that is irrelevant for language membership, once language membership or non-membership become certain. We present a compiler from deterministic stepwise hedge automata to earliest $d\text{SHA}^\downarrow$ s. We show that earliest $d\text{SHA}^\downarrow$ s for regular monadic queries enable earliest query answering for top-down evaluation, both in-memory and in streaming mode.

Contents

8.1	Introduction	224
8.2	Certainty	226
8.2.1	Σ -Certain Membership	226
8.2.2	Certain Non-membership	227
8.3	Schema-Safety	227
8.4	Certainty Automata	229
8.4.1	Membership	229
8.4.2	Non-membership	232
8.4.3	Combining Both	232
8.5	Earliest $d\text{SHA}^\downarrow$ s with Complete Suffix Projection	233

8.1 Introduction

We want improve the top-down membership testers for dSHAs from Section 3.4, so that they can decide membership as early as possible during the evaluation, rather than waiting until the evaluation has finished.

A membership tester for dSHA inputs a dSHA $A = (\Omega, Q, \Delta, I, F)$ on a hedge $h \in \mathcal{H}_\Omega$. The two membership testers from Section 3.4 were obtained by first computing $\llbracket h \rrbracket^\Delta(I)$ in-memory or $\llbracket h \rrbracket_{str}^\Delta(I, \epsilon)$ in streaming mode, and once this is done, testing whether some final state in F got returned. In this way, the decision is taken late after the evaluation of h by A has finished.

In this chapter, we show how to compile any dSHA to some earliest $d\text{SHA}^\downarrow$, that can detect language membership and non-membership at the earliest prefix where it becomes certain. The top-down evaluator of the earliest $d\text{SHA}^\downarrow$ – in-memory or in streaming mode – can be used for detecting certain membership and certain non-membership at the earliest possible prefix. It guarantees complete suffix projection, meaning that the largest suffix of the nested word of the input hedge is projected away, that is irrelevant membership, i.e., for which language membership or non-membership become certain.

The size of the earliest $d\text{SHA}^\downarrow$ may be exponential larger than the size of the input dSHA. If the size blows up, then one can still generate the part of the earliest $d\text{SHA}^\downarrow$ that is needed for evaluating the hedge of interest on-the-fly. This part is no larger than the input hedge, so of linear size in the inputs of the membership problem.

We start with a running example for a dSHA that will be used throughout the current chapter, and also in Chapter 11.

Example 8.1. In Figure 8.1 we show a dSHA A with signature $\Omega = \Sigma \cup \{x\}$ where $\Sigma = \{a\}$. It defines the monadic query $Q = \text{qry}_S(\mathcal{L}(A))$ with schema $S = \llbracket a^* \cdot \langle \top \rangle \cdot \top \rrbracket$ where \top is the nested regular expression shown previously with $\llbracket \top \rrbracket = \mathcal{H}_\Sigma$. When applied to a hedge of the following form, where $n \in \mathbb{N}$ and $h, h' \in \mathcal{H}_\Sigma$:

$$\underbrace{a \cdot \dots \cdot a}_n \cdot \langle h \rangle \cdot h'$$

the query Q selects the nodes in $1, \dots, n-1$ if h does not start with letter “a” and node n otherwise.

Example 8.2. We continue the running Example 8.1. A successful run of automaton A from Figure 8.1 on the hedge $a^3 \cdot x \cdot \langle a \rangle \cdot a \in \mathcal{H}_\Omega$ is illustrated in top-down manner in Figure 8.6. It shows that the node 3 is selected on the hedge $h' = a^3 \cdot \langle a \rangle \cdot a \in \mathcal{H}_\Sigma$ by the

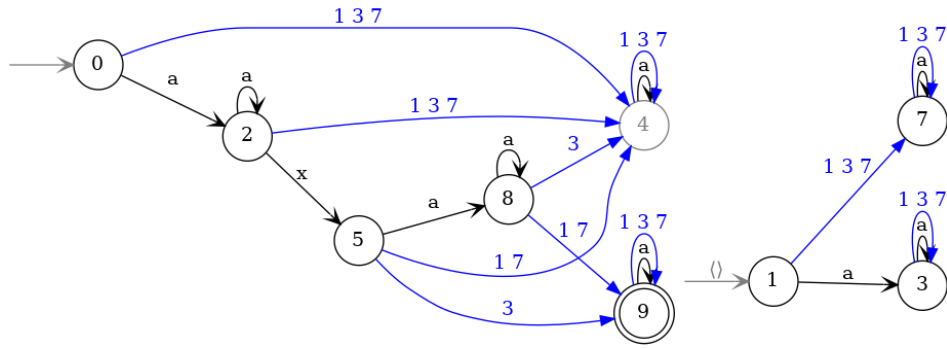


Figure 8.1: A dSHA A for the monadic query $\mathbf{Q} = \text{qry}_{\mathbf{S}}(\mathcal{L}(A))$ on hedges with letters in $\Sigma = \{a\}$ that selects the nodes $1, \dots, n-1$ on hedges of the form $a^n \cdot \langle h \rangle \cdot h'$ if h does not start with letter “ a ” and node n otherwise.

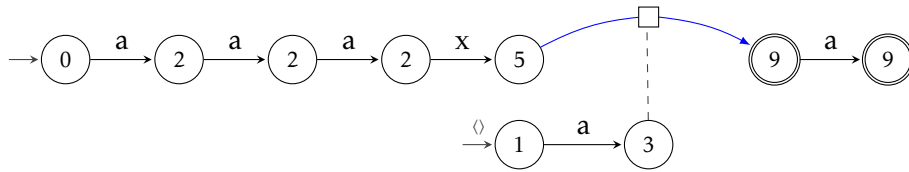


Figure 8.2: A successful run of the dSHA A in Figure 8.1 on the hedge $a^3 \cdot x \cdot \langle a \rangle \cdot a$ showing that $3 \in \mathbf{Q}(a^3 \cdot \langle a \rangle \cdot a)$ where $\mathbf{Q} = \text{qry}_{\mathbf{S}}(\mathcal{L}(A))$.

monadic query defined by A , i.e., that $3 \in \mathbf{Q}(h')$. The earliest prefix where membership $h \in \mathcal{L}(A)$ becomes certain is $aaax\langle a$, as any completion of this prefix to some hedge (in the schema) is accepted by A . How this can be inferred from the run of A will become clearer when compiling the $d\text{SHA } A$ to its earliest $d\text{SHA}^\downarrow A_{e(S)}$ (see Figure 8.5).

8.2 Certainty

We first have to formally define what it means that membership or non-membership become certain at some time point during top-down evaluation. We identify such time points by nested word prefixes. This works for both top-down evaluators, in-memory or in streaming mode.

8.2.1 Σ -Certain Membership

We start with a definition of certain language membership. Intuitively, a nested word prefix is certain for membership to some language, if any hedge completing the nested word prefix is a member of the language. This is basically the same concept as the notion of sufficiency for selection from Definition 1 of [Gauwin *et al.* 2009b], except that for now, we only consider boolean queries (language) rather than monadic queries.

In order to capture monadic queries later on, we introduce a slightly more general certainty notion, that we call Σ -certainty for hedge in \mathcal{H}_Ω where $\Sigma \subseteq \Omega$. Compared to [Al Serhali & Niehren 2023a], we here refine this notion so that it can be made dependent on schemas $L \subseteq S \subseteq \mathcal{H}_\Omega$.

Definition 8.3. Let $\Sigma \subseteq \Omega$ and $S \subseteq \mathcal{H}_\Omega$ be a hedge schema and $L \subseteq S$ a hedge language satisfying this schema. We define that a nested word prefix v is Σ -certain for membership in L with schema S as follows :

$$\Sigma\text{-cert-mem}_S^L(v) \Leftrightarrow_{\text{def}} \forall w \in \text{suffs}(\mathcal{N}_\Sigma). (v \cdot w \in \text{nw}(S) \Rightarrow v \cdot w \in \text{nw}(L))$$

In other words, a nested word prefix v is Σ -certain for membership in $L \subseteq \mathcal{H}_\Omega$, if any completion of v with letters from Σ to a hedge in S belongs to L .

Example 8.4. For instance, if $\Sigma = \{a\}$ then the prefix $v = aaax\langle a$ is Σ -certain for the language of the $d\text{SHA } A$ with signature $\Omega = \{a, x\}$ in Figure 8.1, since any completion of v to some hedge without further x 'es will be accepted by A .

8.2.2 Certain Non-membership

We next define certain non-membership in analogy to certain membership, except that we now restrict ourselves to the full signature Ω . This is basically the notion of sufficiency for rejection from Definition 2 of [Gauwin *et al.* 2009b].

Definition 8.5. We call a prefix v certain for non-membership in $L \subseteq \mathcal{H}_\Omega$ with schema $L \subseteq S$ as follows:

$$\text{cert-nonmem}_S^L(v) \Leftrightarrow_{\text{def}} \forall w \in \text{suffs}(\mathcal{N}_\Omega). (v \cdot w \in \text{nw}(S) \Rightarrow v \cdot w \notin \text{nw}(L))$$

Lemma 8.6. $\text{cert-nonmem}_S^L(v) \Leftrightarrow \Omega\text{-cert-mem}_S^{\bar{L}}(v)$.

Proof. Obvious from the definitions. □

Note that a nested-word prefix is suffix-irrelevant if and only if it is either certain for membership or certain for non-membership.

8.3 Schema-Safety

Certain membership and non-membership to languages of dSHA_S can be reduced to schema safety problems, i.e., whether the current states is safe to reach some final state if not going out of the schema.

Let $A = (\Sigma, \mathcal{Q}, \Delta, I, F)$ be an SHA, $S \subseteq \mathcal{H}_\Sigma$ a schema, and $P \subseteq \mathcal{Q}$. We define the set of safe states leading to P when assuming schema S by:

$$\text{safe}_S^\Delta(P) = \{q \in \mathcal{Q} \mid \forall h \in S. \forall p \in \mathcal{Q}. (q \xrightarrow{h} p \text{ wrt } \Delta \Rightarrow p \in P)\}$$

Now suppose that there exists $F \subseteq F_S \subseteq \mathcal{Q}$ such that $S = \mathcal{L}(A[F/F_S])$. This implies $\mathcal{L}(A) \subseteq S \subseteq \mathcal{H}_\Sigma$ in particular. The set of states that safely reach a subset $P \subseteq \mathcal{Q}$ for all hedges that do not go out of the schema S then is:

$$\text{safe}_{F_S}^\Delta(P) = \text{safe}_S^\Delta(P \cup \overline{F_S})$$

We can now show that both notions of schema-safety coincide.

Lemma 8.7. Let A be a complete dSHA, $F \subseteq F_S \subseteq \mathcal{Q}$, and $S = \mathcal{L}(A[F/F_S])$. For any $P \subseteq \mathcal{Q}$:

$$\text{safe}_{F_S}^\Delta(P) = \text{safe}_S^\Delta(P)$$

Proof. We prove the two inclusions.

“ \subseteq ” Suppose that $q \in \text{safe}_{F_S}^\Delta(P)$. Then $\text{acc}^\Delta(q) \subseteq P \cup \overline{F_S}$. So for any $h \in \mathcal{H}_\Sigma$ it holds that $\llbracket h \rrbracket(q) \in P \cup \overline{F_S}$. So for any $h \in S$, we have $\llbracket h \rrbracket(q) \in P$, i.e. $h \in \text{safe}_S^\Delta(P)$.

“ \supseteq ” Suppose that $q \in \text{safe}_S^\Delta(P)$. Then for any $h \in S$ it holds that $\llbracket h \rrbracket(q) \in P$. Since A is complete and deterministic, for any $h \in \overline{S}$, we have $\llbracket h \rrbracket(q) \in \overline{F_S}$. So for any $h \in \mathcal{H}_\Sigma$ we have $\llbracket h \rrbracket(q) \in P \cup \overline{F_S}$. Hence, $\text{acc}^\Delta(q) \subseteq P \cup \overline{F_S}$, and thus $q \in \text{safe}_{F_S}^\Delta(P)$.

□

If A is deterministic and complete and $S = \mathcal{L}(A[F/F_S])$, then whether a state q of A is safe for F with respect to schema S is a dSHA inclusion problem (and thus in polynomial time).

Lemma 8.8. *For any complete dSHA $A = (\mathcal{Q}, \Sigma, \Delta, I, F)$, subset $F \subseteq F_S \subseteq \mathcal{Q}$, schema $S = \mathcal{L}(A[F/F_S])$, and state $q \in \mathcal{Q}$:*

$$\mathcal{L}(A[I/\{q\}, F/F_S]) \subseteq \mathcal{L}(A[I/\{q\}]) \Leftrightarrow q \in \text{safe}_S^\Delta(F)$$

Proof. By Lemma 8.7 and the completeness of the dSHA A , we have that $\text{safe}_S^\Delta(F) = \text{safe}_{F_S}^\Delta(F)$. Therefore, it is sufficient to show that:

$$\mathcal{L}(A[I/\{q\}, F/F_S]) \subseteq \mathcal{L}(A[I/\{q\}]) \Leftrightarrow q \in \text{safe}_{F_S}^\Delta(F)$$

Let us prove this:

“ \Leftarrow ”. Suppose that $q \notin \text{safe}_{F_S}^\Delta(F)$. The definition of schema-based safety yields $\text{acc}^\Delta(\{q\}) \cap (F_S \setminus F) \neq \emptyset$. So there exists some hedge $h \in \mathcal{H}_\Sigma$ such that $q \xrightarrow{h} F_S \setminus F$ wrt Δ . In particular, $q \in \mathcal{L}(A[I/\{q\}, F/F_S])$. By determinism, there exists no other transition on h from q and thus $q \notin \mathcal{L}(A[I/\{q\}])$. Hence $\mathcal{L}(A[I/\{q\}, F/F_S]) \not\subseteq \mathcal{L}(A[I/\{q\}])$.

“ \Rightarrow ”. Suppose that $q \in \text{safe}_{F_S}^\Delta(F)$. Let $h \in \mathcal{L}(A[I/\{q\}, F/F_S])$. Then there exists $q' \in F_S$ such that $q \xrightarrow{h} q'$ wrt Δ . Hence $q' \in \text{acc}^\Delta(\{q\})$ so that $\text{acc}^\Delta(\{q\}) \subseteq F \cup (\mathcal{Q} \setminus F_S)$ implies $q' \in F$. Thus, $h \in \mathcal{L}(A[I/\{q\}])$.

□

Lemma 8.8 show that schema-safety is an automata inclusion problem. This implies that schema-free safety, where $\mathbf{S} = \mathcal{H}_\Sigma$, is a universality problem:

$$\mathcal{L}(A[I/\{q\}]) = \mathcal{H}_\Omega \Leftrightarrow q \in \text{safe}^\Delta(F)$$

This was stated in Lemma 5 of [Al Serhali & Niehren 2023a], which treated only the schema-less case.

8.4 Certainty Automata

We next show that schema-safety can be used to detect certain language membership and non-membership for languages of dSHAs with respect to some schema.

So let $A = (\Omega, \mathcal{Q}, \Delta, I, F)$ be dSHA. We define for any $Q \subseteq \mathcal{Q}$ and $q \in \mathcal{Q}$, such that $q@^\Delta p$ is well-defined for some $p \in \mathcal{Q}$, the following two sets:

$$\begin{aligned} sdown^\Delta(q, Q) &= \text{safe}^\Delta(\text{down}^\Delta(q, Q)) \\ \text{down}^\Delta(q, Q) &= \{p \in \mathcal{Q} \mid q@^\Delta p \in Q\} \end{aligned}$$

Otherwise, if $q@^\Delta p$ is undefined for all $p \in \mathcal{Q}$, then $sdown^\Delta(q, Q)$ is undefined.

8.4.1 Membership

For deciding Σ -certain membership for schema \mathbf{S} based on schema safety, we define a $d\text{SHA}^\downarrow$ that we call the Σ -certain membership automata $A^{\Sigma\text{-cert-mem}(\mathbf{S})}$ as follows:

$$A^{\Sigma\text{-cert-mem}(\mathbf{S})} = (\Omega, \mathcal{Q}^{\Sigma\text{-cert-mem}}, \Delta^{\Sigma\text{-cert-mem}}, I^{\Sigma\text{-cert-mem}(\mathbf{S})}, F^{\Sigma\text{-cert-mem}})$$

It has the following sets of states:

$$\begin{aligned} \mathcal{Q}^{\Sigma\text{-cert-mem}} &= \mathcal{Q} \times 2^\mathcal{Q} \\ I^{\Sigma\text{-cert-mem}(\mathbf{S})} &= I \times \{\text{safe}_\mathbf{S}^{\Delta_\Sigma}(F)\} \\ F^{\Sigma\text{-cert-mem}} &= F \times 2^\mathcal{Q} \end{aligned}$$

The transition rules in $\Delta^{\Sigma\text{-cert-mem}}$ are such that for all $Q \subseteq \mathcal{Q}$, $q, p \in \mathcal{Q}$, and $q_{init}^{tree} \in \langle \rangle^\Delta$ and $a \in \Omega$:

$$\begin{aligned} (q, Q) &\xrightarrow{\langle \rangle} (q_{init}^{tree}, sdown^{\Delta_\Sigma}(q, Q)) \\ (q, Q)@^\Delta(p, Q') &\rightarrow (q@^\Delta p, Q) \\ (q, Q) &\xrightarrow{a} (a^\Delta(q), Q) \end{aligned}$$

In the first component $A^{\Sigma\text{-cert-mem}(\mathbf{S})}$ behaves like A , while in the second component it computes safety information. Therefore, $L(A) = L(A^{\Sigma\text{-cert-mem}(\mathbf{S})})$. We next show that the streaming evaluator of $A^{\Sigma\text{-cert-mem}(\mathbf{S})}$ detects Σ -certain membership at any prefix.

Proposition 8.9. *Let $A = (\Omega, \mathcal{Q}, \Delta, I, F)$ be a $d\text{SHA}$ and $\mathbf{S} \subseteq \mathcal{H}_\Omega$ a schema. Let $v \in \text{pref}(\mathcal{N}_\Omega)$ be a nested word prefix and $\Sigma \subseteq \Omega$ such that $\Delta|_\Sigma$ is complete. For any $q \in \mathcal{Q}$, $Q \subseteq \mathcal{Q}$, $q_{\text{init}}^{\Sigma\text{-cert-mem}(\mathbf{S})} \in I^{\Sigma\text{-cert-mem}(\mathbf{S})}$, and stack $\sigma \in (\mathcal{Q}^{\Sigma\text{-cert-mem}})^*$, such that:*

$$((q, Q), \sigma) = \llbracket v \rrbracket_{\text{str}}^{\Delta^{\Sigma\text{-cert-mem}}} (q_{\text{init}}^{\Sigma\text{-cert-mem}(\mathbf{S})}, \epsilon)$$

it follows that:

$$\Sigma\text{-cert-mem}_{\mathbf{S}}^{\mathcal{L}(A)}(v) \Leftrightarrow q \in Q$$

Proof. If $I = \emptyset$ the $I^{\Sigma\text{-cert-mem}(\mathbf{S})} = \emptyset$, so there is nothing to show. Otherwise I is a singleton since A is deterministic. Let $q_{\text{init}} \in I$ be the unique element of this singleton. We then have $q_{\text{init}}^{\Sigma\text{-cert-mem}(\mathbf{S})} = (q_{\text{init}}, \text{safe}_{\mathbf{S}}^{\Delta|_\Sigma}(F))$.

Nested word prefixes may contain dangling opening parenthesis but no dangling closing parenthesis. We prove the equivalence for all v and A by induction on the number of dangling opening parenthesis in v . So let $\sigma \in (\mathcal{Q}^{\Sigma\text{-cert-mem}})^*$ be a stack such that:

$$((q, Q), \sigma) = \llbracket v \rrbracket_{\text{str}}^{\Delta^{\Sigma\text{-cert-mem}}} (q_{\text{init}}^{\Sigma\text{-cert-mem}(\mathbf{S})}, \epsilon)$$

In the base case, v does not have dangling opening parenthesis, so v is well-nested. The transition rules of $\Delta^{\Sigma\text{-cert-mem}}$ change the states in the first component in the same manner than Δ . Furthermore, the subset Q of target states in the second component remains unchanged when reading letters in Ω . The same holds when processing nested words of trees $\langle w \rangle$ where w is well-nested: the target set Q is pushed to the stack at the opening parenthesis and popped from the stack at the closing parenthesis for continuation. Therefore, the assumption $\llbracket v \rrbracket_{\text{str}}^{\Delta^{\Sigma\text{-cert-mem}}} (q_{\text{init}}^{\Sigma\text{-cert-mem}(\mathbf{S})}, \epsilon) = ((q, Q), \sigma)$ implies that $Q = \text{safe}_{\mathbf{S}}^{\Delta|_\Sigma}(F)$, $\sigma = \epsilon$ and $\llbracket v \rrbracket^\Delta(q_{\text{init}}) = q$. Hence:

$$\Sigma\text{-cert-mem}_{\mathbf{S}}^{\mathcal{L}(A)}(v) \Leftrightarrow \mathbf{S} \subseteq \mathcal{L}(A[q_{\text{init}}/q])$$

By Lemma 8.8 and the completeness of $\Delta|_\Sigma$, this is equivalent to $q \in \text{safe}_{\mathbf{S}}^{\Delta|_\Sigma}(F)$ and thus to $q \in Q$.

For the induction step, we consider a nested word prefix v with at least one dangling opening parenthesis. We split v at the first dangling parenthesis such that

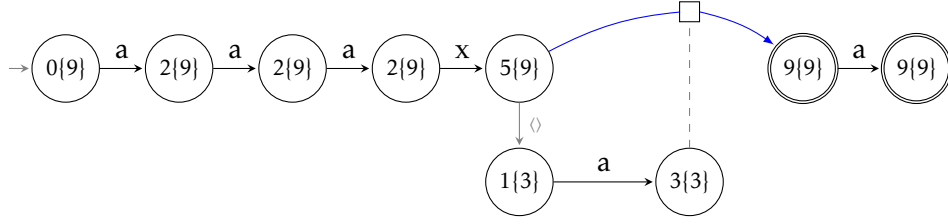


Figure 8.3: A successful run of the $d\text{SHA}^\downarrow A^{\Sigma\text{-cert-mem}}$ on $\text{hdg}(aaax\langle a \rangle a)$.

$v = v_1 \cdot \langle \cdot v_2$ for some v_1 and v_2 . This implies that $v_1 \in n\omega(\mathcal{H}_\Omega)$ in contrast to v_2 . Let:

$$((q_1, Q_1), \sigma_1) = \llbracket v_1 \rrbracket_{\text{str}}^{\Delta^{\Sigma\text{-cert-mem}}} ((q_{\text{init}}^{\Sigma\text{-cert-mem}}(\mathbf{S}), \epsilon)$$

Since the word v_1 is well-nested, we will reach the first dangling parenthesis with the empty stack, so $\sigma_1 = \epsilon$ and $Q_1 = \text{safe}_S^{\Delta_\Sigma}(F)$. Reading the first dangling opening parenthesis in this configuration yields $\llbracket \langle \rrbracket_{\text{str}}^{\Delta^{\Sigma\text{-cert-mem}}} ((q_1, Q_1), \epsilon) = ((t, Q_2), \gamma)$ where $Q_2 = \text{sdown}^{\Delta_\Sigma}(q_1, Q_1)$ and $\gamma = (q_1, Q_1)$. Let $A_2 = A[q_{\text{init}}/q_{\text{init}}^{\text{tree}}, F/F_2]$ where $F_2 = \text{down}^{\Delta}(q_1, Q_1)$. Let $I = (q_{\text{init}}^{\text{tree}}, \text{safe}^{\Delta_\Sigma}(F_2))$ be the initial state of $A_2^{\Sigma\text{-cert-mem}}$. Note that $\text{safe}^{\Delta_\Sigma}(F_2) = \text{sdown}^{\Delta_\Sigma}(q_1, Q_1)$. The assumption $\llbracket v \rrbracket_{\text{str}}^{\Delta^{\Sigma\text{-cert-mem}}} (q_{\text{init}}^{\Sigma\text{-cert-mem}}(\mathbf{S}), \epsilon) = ((q, Q), \sigma)$ implies:

$$\llbracket v_2 \rrbracket_{\text{str}}^{\Delta^{\Sigma\text{-cert-mem}}} (I, \gamma) = ((q, Q), \sigma)$$

Since the first dangling opening parenthesis of v will never be closed, the first stack symbol γ is never popped when reading v_2 , so we have $\sigma = \gamma \cdot \sigma'$ for some stack σ' . Therefore, γ can be canceled out, showing:

$$\llbracket v_2 \rrbracket_{\text{str}}^{\Delta^{\Sigma\text{-cert-mem}}} (I, \epsilon) = ((q, Q), \sigma')$$

Hence, $\Sigma\text{-cert-mem}_S^{\mathcal{L}(A)}(w)$ is equivalent to $\Sigma\text{-cert-mem}_S^{\mathcal{L}(A_2)}(w_2)$. By induction hypothesis applied to v_2 and A_2 , this is equivalent to $q \in Q$. \square

Example 8.10. We illustrate Proposition 8.9 in Figure 8.3 at the $d\text{SHA } A = (\Omega, \mathcal{Q}, \Delta, I, F)$ from Figure 8.1. Recall that it has the signature $\Omega = \Sigma^x$ where $\Sigma = \{a\}$. Given that Δ_Σ is complete, Σ -certain membership of $a^3 \cdot \langle a \rangle \cdot a$ to $\mathcal{L}(A)$ can be detected at the earliest prefix $aaax\langle a$, by running the streaming evaluator of the earliest automaton $A^{\Sigma\text{-cert-mem}}$. Note that the earliest automaton is a $d\text{SHA}^\downarrow$ passing safety information top-down (while $d\text{SHAs}$ cannot pass any information top-down). We have $\text{safe}^{\Delta_\Sigma}(\{9\}) = \{9\}$ and $\text{sdown}^{\Delta_\Sigma}(5, \{9\}) = \{3\}$. Hence, $\llbracket aaax\langle \rrbracket_{\text{str}}^{\Delta^{\Sigma\text{-cert-mem}}} (q_{\text{init}}^{\Sigma\text{-cert-mem}}, \epsilon) = ((1, \{3\}), \sigma)$ where the stack is $\sigma = (5, \{9\})$. Since $1 \notin \{3\}$, membership is not yet Σ -certain. Indeed, the Σ -completion $a^3 \cdot x \cdot \langle \rangle$ is not accepted. After reading the next letter a , we have

$\llbracket aaax\langle a \rangle_{str}^{\Delta^{\Sigma\text{-cert-mem}}} = ((3, \{3\}), \sigma)$. Since the current state 3 belongs to the current set of safe states $\{3\}$, membership is Σ -certain, i.e., membership of all completions without further x 'es.

8.4.2 Non-membership

When interested in non-membership, we can reduce certain non-membership to language $\mathcal{L}(A)$ to Ω -certain membership of $\overline{\mathcal{L}(A)}$ by Lemma 8.6. This can be decided by the Ω -certain membership automaton $(\overline{A})^{\Omega\text{-cert-mem}(\mathbf{S})}$ for the complement automaton $\overline{A} = (\Omega, \mathcal{Q}, \Delta, I, \overline{F})$. It recognizes $\mathcal{L}(\overline{A}) = \overline{\mathcal{L}(A)}$ since A was assumed to be complete.

8.4.3 Combining Both

For earliest query answering in Section 11.2.3 on monadic query answering, we will have to decide Σ -certainty for membership to $\mathcal{L}(A)$ and at the same time certainty of non-membership to $\mathcal{L}(A)$. For doing both at the same time, we can use the product of the Σ -certain membership automata $A^{\Sigma\text{-cert-mem}(\mathbf{S})}$ and the Ω -certain membership automaton $(\overline{A})^{\Omega\text{-cert-mem}(\mathbf{S})}$ where \overline{A} is the complement automaton of A .

In order to achieve this, rather more directly, we define the Σ -certain automaton $A^{\Sigma\text{-cert}(\mathbf{S})}$ as the following $d\text{SHA}^\downarrow$:

$$A^{\Sigma\text{-cert}(\mathbf{S})} = (\Omega, \mathcal{Q}^{\Sigma\text{-cert}}, \Delta^{\Sigma\text{-cert}}, I^{\Sigma\text{-cert}(\mathbf{S})}, F^{\Sigma\text{-cert}})$$

It has the following set of states:

$$\begin{aligned} \mathcal{Q}^{\Sigma\text{-cert}} &= \mathcal{Q} \times 2^{\mathcal{Q}} \times 2^{\mathcal{Q}} \\ I^{\Sigma\text{-cert}(\mathbf{S})} &= I \times \{\text{safe}_{\mathbf{S}}^{\Delta_{\Sigma}}(F)\} \times \{\text{safe}_{\mathbf{S}}^{\Delta}(\overline{F})\} \\ F^{\Sigma\text{-cert}} &= F \times 2^{\mathcal{Q}} \times 2^{\mathcal{Q}} \end{aligned}$$

The transition rules in $\Delta^{\Sigma\text{-cert-mem}}$ are such that for all subsets of states $Q, R, Q', R' \in 2^{\mathcal{Q}}$, $q, p \in \mathcal{Q}$, and $q_{init}^{tree} \in \langle \rangle^{\Delta}$ and $a \in \Omega$:

$$\begin{aligned} (q, Q, R) &\xrightarrow{\langle \rangle} (q_{init}^{tree}, \text{sdown}^{\Delta_{\Sigma}}(q, Q), \text{sdown}^{\Delta}(q, R)) \\ (q, Q, R) @ (p, Q', R') &\rightarrow (q @^{\Delta} p, Q, R) \\ (q, Q, R) &\xrightarrow{a} (a^{\Delta}(q), Q, R) \end{aligned}$$

By construction, the projection of $A^{\Sigma\text{-cert}(\mathbf{S})}$ to components 1 and 2 of the states is

the Σ -certain membership automaton $A^{\Sigma\text{-cert-mem}(\mathbf{S})}$, and the projection of $A^{\Sigma\text{-cert}(\mathbf{S})}$ to components 1 and 3 of the states is the certain non-membership automaton $(\overline{A})^{\Omega\text{-cert-mem}(\mathbf{S})}$.

Proposition 8.11. *Let $A = (\Omega, \mathcal{Q}, \Delta, I, F)$ be a complete $d\text{SHA}$ and $\mathbf{S} \subseteq \mathcal{H}_\Omega$ a schema. Let $v \in \text{prefs}(\mathcal{N}_\Omega)$ be a nested word prefix and $\Sigma \subseteq \Omega$. For any $q \in \mathcal{Q}$, $Q, R \subseteq \mathcal{Q}$, and $q_{\text{init}}^{\Sigma\text{-cert}(\mathbf{S})} \in I^{\Sigma\text{-cert}(\mathbf{S})}$ such that:*

$$((q, Q, R), \sigma) = \llbracket v \rrbracket_{\text{str}}^{\Delta^{\Sigma\text{-cert}}}(q_{\text{init}}^{\Sigma\text{-cert}(\mathbf{S})}, \epsilon)$$

holds for some stack $\sigma \in (\mathcal{Q}^{\Sigma\text{-cert}})^$, then it follows that:*

$$\begin{aligned} \Sigma\text{-cert-mem}_{\mathbf{S}}^{\mathcal{L}(A)}(v) &\Leftrightarrow q \in Q & \text{and} \\ \text{cert-nonmem}_{\mathbf{S}}^{\mathcal{L}(A)}(v) &\Leftrightarrow q \in R \end{aligned}$$

Proof. This is a consequence of Proposition 8.9 applied once for certain membership, and once for certain non-membership via complementation. \square

8.5 Earliest $d\text{SHA}^\downarrow$ s with Complete Suffix Projection

We now want to compile any $d\text{SHA}$ to some earliest $d\text{SHA}^\downarrow$ with suffix projection. This earliest $d\text{SHA}^\downarrow$ is not only able to detect Σ -certain membership and certain non-membership at the earliest possible prefix, but also to stop the computation there. When becoming Σ -certain for membership, it goes into a distinguished selection state, where it stays forever, and when becoming certain for non-membership it blocks the run. We now formalize this notion of earliest $d\text{SHA}^\downarrow$ s.

Definition 8.12. *Let $\Sigma \subseteq \Delta$. A $d\text{SHA}^\downarrow A = (\Omega, \mathcal{Q}, \Delta, I, F)$ with schema $\mathbf{S} \subseteq \mathcal{H}_\Omega$ is called Σ -earliest for \mathbf{S} with complete suffix projection if there exists a state $\mathbf{sel} \in F$ such that for all hedges $h \in \mathbf{S}$ the unique partial run R of A on h satisfies:*

1. *it goes to the state \mathbf{sel} for exactly all those prefixes of $\text{nw}(h)$ that are Σ -certain for membership in $\mathcal{L}(A)$ wrt \mathbf{S} , and*
2. *it blocks for all prefixes of $\text{nw}(h)$ that are certain for non-membership in $\mathcal{L}(A)$ with \mathbf{S} .*

It should be noticed that whenever an earliest automaton goes to state \mathbf{sel} for some prefix, it has to remain there for any larger prefixes in $\text{prefs}(\text{nw}(\mathbf{S}))$, since all these prefixes will also be Σ -certain for membership. Since \mathbf{sel} is final, any input hedge in the schema whose nested word has a prefix leading to \mathbf{sel} will be accepted.

This means that **sel** is a selection state of A with respect to \mathbf{S} in the following sense:

Definition 8.13. Let $A = (\mathcal{Q}, \Sigma, \Delta, I, F)$ be a SHA^\downarrow with schema $\mathbf{S} \subseteq \mathcal{H}_\Sigma$. A state $q \in \mathcal{Q}$ is called a selection state of A wrt. \mathbf{S} if A is schema-complete for \mathbf{S} and all runs of A on hedges in \mathbf{S} that contain q are successful.

Lemma 8.14. If A has a selection state q wrt \mathbf{S} , then any hedge $h \in \mathbf{S}$ that has a partial run with A ending in q is accepted, i.e., $h \in \mathcal{L}(A)$.

Proof. Suppose that a hedge $h \in \mathbf{S}$ has a partial run with A that ends in some selection state q . Since A is schema-complete for \mathbf{S} this partial run can be completed to some run $r \in \text{run}^\Delta(h)$. Since q is a selection state of A wrt. \mathbf{S} , it follows that r is successful, so that $h \in \mathcal{L}(A)$. \square

The certainty automata can detect all suffixes that are certain for projection, but they do neither project them nor return the result of the membership test. We now present a compiler that maps Σ -certain $d\text{SHA}^\downarrow A^{\Sigma\text{-cert}(\mathbf{S})}$ to a Σ -earliest $\text{SHA}^\downarrow A_{e(\mathbf{S})}$ with complete suffix projection. These will report Σ -certain membership and certain non-membership at the earliest prefix that is suffix-irrelevant. In case of certain non-membership they stop the computation, and in case of Σ -certain membership they go into a special selection state in which they stay until the end.

For any $d\text{SHA}$ A and schema $\mathcal{L}(A) \subseteq \mathbf{S}$ we define an $d\text{SHA}^\downarrow A_{e(\mathbf{S})}$ as follows:

$$A_{e(\mathbf{S})} = (\Sigma, \mathcal{Q}_e, \Delta_e, I_{e(\mathbf{S})}, F_e)$$

as follows. Let **sel** be a fresh symbol for the special selection state. We define:

$$\begin{aligned} \mathcal{Q}_e &= \{(q, Q, R) \in \mathcal{Q}^{\Sigma\text{-cert}} \mid q \notin Q \cup R\} \cup \{\mathbf{sel}\} \\ I_{e(\mathbf{S})} &= \{(q, Q, R) \in \mathcal{Q}_e \mid q \in I\} \cup \{\mathbf{sel} \mid (q, Q, R) \in I^{\Sigma\text{-cert}(\mathbf{S})}, q \in Q \setminus R\} \\ F_e &= \{(q, Q, R) \in \mathcal{Q}_e \mid q \in F\} \cup \{\mathbf{sel}\} \end{aligned}$$

The transition rules in Δ_e are given in Figure 8.4.

Theorem 5. For any $d\text{SHA}$ $A = (\Sigma, \mathcal{Q}, \Delta, I, F)$ and schema $\mathbf{S} \subseteq \mathcal{H}_\Sigma$ so that A is schema-complete for schema \mathbf{S} , the $d\text{SHA}^\downarrow A_{e(\mathbf{S})}$ is Σ -earliest for \mathbf{S} .

Proof. This follows from Proposition 8.11 and the construction, which blocks the runs of $A^{\Sigma\text{-cert}(\mathbf{S})}$ once they become certain for non-membership, and goes to the selection state **sel** once becoming Σ -certain for membership. \square

$$\begin{array}{c}
\frac{\mu \xrightarrow{a} \mu' \in \Delta^{\Sigma\text{-cert}} \quad \mu' \in \mathcal{Q}_e}{\mu \xrightarrow{a} \mu' \in \Delta_e} \quad \frac{\mu \xrightarrow{a} (q, Q, R) \in \Delta^{\Sigma\text{-cert}} \quad q \in Q \setminus R}{\mu \xrightarrow{a} \mathbf{sel} \in \Delta_e} \\
\\
\frac{\mu \xrightarrow{\langle \rangle} \mu' \in \Delta^{\Sigma\text{-cert}} \quad \mu' \in \mathcal{Q}_e}{\mu \xrightarrow{\langle \rangle} \mu' \in \Delta_e} \quad \frac{\mu \xrightarrow{\langle \rangle} (q, Q, R) \in \Delta^{\Sigma\text{-cert}} \quad q \in Q \setminus R}{\mu \xrightarrow{\langle \rangle} \mathbf{sel} \in \Delta_e} \\
\\
\frac{\mu @ \mu' \rightarrow \mu'' \in \Delta^{\Sigma\text{-cert}} \quad \mu'' \in \mathcal{Q}_e}{\mu @ \mu' \rightarrow \mu'' \in \Delta_e} \quad \frac{\mu @ \mu' \rightarrow (q, Q, R) \in \Delta^{\Sigma\text{-cert}} \quad q \in Q \setminus R}{\mu @ \mu' \rightarrow \mathbf{sel} \in \Delta_e} \\
\\
\frac{a \in \Sigma \cup \{\langle \rangle\}}{\mathbf{sel} \xrightarrow{a} \mathbf{sel} \in \Delta_e} \quad \frac{\mu \in \mathcal{Q}_e}{\mu @ \mathbf{sel} \rightarrow \mathbf{sel} \in \Delta_e}
\end{array}$$

Figure 8.4: The transition rules Δ_e of the Σ -earliest automaton with complete suffix projection, inferred from transition rules $\Delta^{\Sigma\text{-cert}}$ of the Σ -certain membership and non-membership automaton.

Example 8.15. In Figure 8.5, we present the earliest automaton $A_{e(S)}$ for the $d\text{SHA}$ A in Figure 8.1, up to removing useless **sel** transition rules. A successful run of the earliest automaton $A_{e(S)}$ on the hedge $a^3 \cdot x \cdot \langle a \rangle \cdot a$ is given in Figure 8.6. For the query $Q = \text{qry}_S(\mathcal{L}(A))$. It shows that $3 \in Q(a^3 \cdot \langle a \rangle \cdot a)$ at the earliest prefix $v = a^3 \langle a$ where $3 \in \Sigma\text{-cert-mem}_S^{\mathcal{L}(A)}(v)$, since reaching the selection state **sel** there for the first time. Furthermore, the partial run on $a^2 x a \langle a$ is blocking, showing $2 \notin Q(a^3 \cdot \langle a \rangle \cdot a)$ at the earliest prefix $v = a^3 \langle a$ too where $2 \in \text{cert-nonmem}_S^{\mathcal{L}(A)}(v)$.

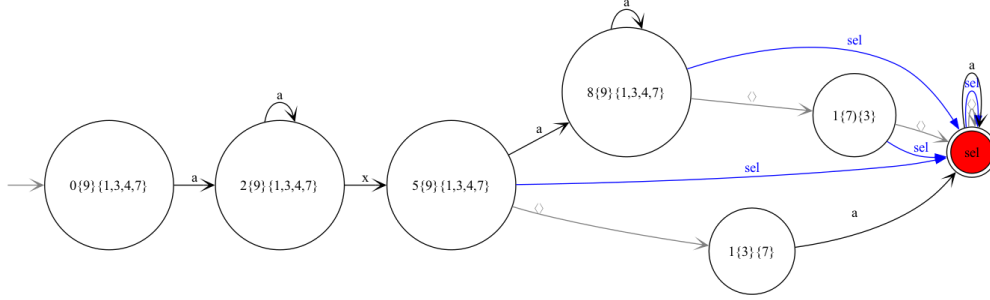


Figure 8.5: The earliest automaton $A_{e(S)}$ for the dSHA A in Figure 8.1, up to removing useless **sel** transition rules.

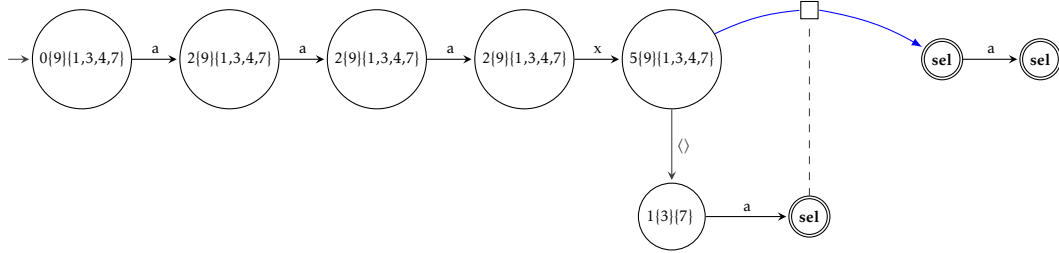


Figure 8.6: A successful run of the earliest dSHA $A_{e(S)}$ in Figure 8.5 on the hedge $a^3 \cdot x \cdot \langle a \rangle \cdot a$ showing that $3 \in \mathbf{Q}(a^3 \cdot \langle a \rangle \cdot a)$ where $\mathbf{Q} = \text{qry}_S(\mathcal{L}(A))$ at the earliest prefix $v = a^3 \langle a \rangle$ where $3 \in \Sigma\text{-cert-mem}_S^{\mathcal{L}(A)}(v)$.

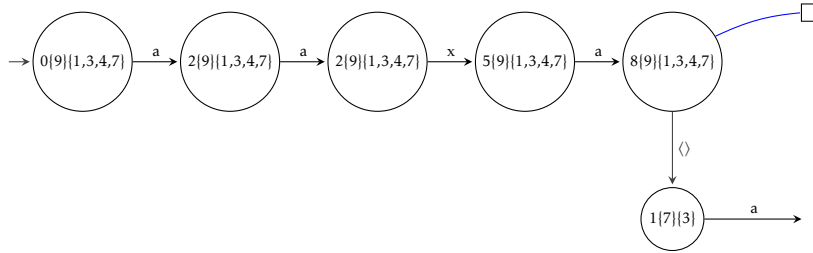


Figure 8.7: A blocking partial run of the earliest dSHA $A_{e(S)}$ in Figure 8.5 on the hedge $a^2 \cdot x \cdot a \cdot \langle a \rangle \cdot a$ showing that $2 \notin \mathbf{Q}(a^3 \cdot \langle a \rangle \cdot a)$ where $\mathbf{Q} = \text{qry}_S(\mathcal{L}(A))$ at the earliest prefix $v = a^3 \langle a \rangle$ where $2 \in \text{cert-nonmem}_S^{\mathcal{L}(A)}(v)$.

Combining Subhedge and Suffix Projection

Abstract

We show how to combine SHA^\downarrow s with selection states and SHA^\downarrow s with subhedge projection states in a generic manner. This permits to add subhedge projection to earliest $d\text{SHA}^\downarrow$ s. We show that the integration of congruence projection into an earliest $d\text{SHA}^\downarrow$ leads to an earliest $d\text{SHA}^\downarrow$ with complete subhedge and suffix projection.

Contents

9.1	Combination Algorithm	237
9.2	Soundness and Completeness	238
9.3	Automaton Size	240
9.4	Benchmark dSHA for XPATH Queries	240

9.1 Combination Algorithm

We show how to enhance earliest automata for dSHAs with subhedge projection. This can be used to yield earliest membership testers with complete subhedge and suffix projection.

Let \mathbf{S} be a schema \mathbf{S} and A^π and A_e two SHA^\downarrow s with schema \mathbf{S} that recognize the

same language up to the schema:

$$\mathcal{L}(A^\pi) \cap \mathcal{L}(\mathbf{S}) = \mathcal{L}(A_e) \cap \mathcal{L}(\mathbf{S})$$

Furthermore, we assume that A_e has a selection state **sel** indicating certain membership at the earliest possible prefix.

In order to obtain earliest membership with subhedge projection, we combine the two SHA^\downarrow s into a single SHA^\downarrow $A_e^\pi = (\Sigma, \mathcal{Q}_e^\pi, \Delta_e^\pi, I_e^\pi, F_e^\pi)$ basically running both automata in parallel, but under a shared control. The state set of A_e^π are as follows:

$$\begin{aligned} \mathcal{Q}_e^\pi &= (\mathcal{Q}^\pi \times (\mathcal{Q}_e \setminus \{\mathbf{sel}\})) \cup \{\mathbf{sel}\} \\ I_e^\pi &= (I^\pi \times (I_e \setminus \{\mathbf{sel}\})) \cup \{\mathbf{sel} \mid \mathbf{sel} \in I_e\} \\ F_e^\pi &= (F^\pi \times (F_e \setminus \{\mathbf{sel}\})) \cup \{\mathbf{sel}\} \end{aligned}$$

The transition rules in Δ_e^π are given by the in Figure 9.1. Any run of A_e^π synchronizes parallel runs of A^π and A_e as follows: whenever A_e goes to **sel**, then A_e^π does so too, and whenever A^π goes into a subhedge projection state then it makes A_e jump over the subsequent subhedge.

9.2 Soundness and Completeness

We next show that the soundness and completeness of a projection algorithm are preserved when combined with some earliest membership tester when assuming determinism.

Proposition 9.1. *Let A^π and A_e be $d\text{SHA}^\downarrow$ s with the same schema \mathbf{S} and the same language up to the schema, i.e. $\mathcal{L}(A^\pi) \cap \mathcal{L}(\mathbf{S}) = \mathcal{L}(A_e) \cap \mathcal{L}(\mathbf{S})$. The combination $d\text{SHA}^\downarrow A_e^\pi$ then has the same language up to schema \mathbf{S} too. Furthermore, if A_e is earliest for \mathbf{S} then A_e^π is earliest for \mathbf{S} too and goes into some subhedge projection state whenever A^π does.*

Proof. If q is a subhedge projection state of A^π and r a state of A_e then (q, r) is a subhedge projection state of A_e^π . The evaluator for automaton A_e^π runs A^π and A_e in parallel on the input hedge, while skipping subhedges starting in subhedge projection states of A^π . These include the subhedges starting in a projection state when evaluating A_e^π on h . By Proposition 7.11 applied to A^π such subhedges are irrelevant for $\mathcal{L}(A^\pi)$ with respect to \mathbf{S} since A^π is assumed to be deterministic. Since $\mathcal{L}(A^\pi) \cap \mathbf{S} = \mathcal{L}(A_e) \cap \mathbf{S}$, skipping such subhedge does not affect acceptance by A_e for

$$\begin{array}{c}
\frac{q \xrightarrow{a} q' \in \Delta^\pi \quad r \xrightarrow{a} r' \in \Delta_e \quad q \notin P \quad r' \neq \mathbf{sel}}{(q, r) \xrightarrow{a} (q', r') \in \Delta_e^\pi} \quad \frac{q \xrightarrow{a} q' \in \Delta^\pi \quad r \xrightarrow{a} \mathbf{sel} \in \Delta_e}{(q, r) \xrightarrow{a} \mathbf{sel} \in \Delta_e^\pi} \\
\\
\frac{q \xrightarrow{a} q' \in \Delta^\pi \quad r \xrightarrow{a} r' \in \Delta_e \quad q \in P \quad r' \neq \mathbf{sel}}{(q, r) \xrightarrow{a} (q', r) \in \Delta_e^\pi} \\
\\
\frac{q@p \rightarrow q' \in \Delta^\pi \quad r@s \rightarrow r' \in \Delta_e \quad r' \neq \mathbf{sel} \quad q \notin P}{(q, r)@(p, s) \rightarrow (q', r') \in \Delta_e^\pi} \quad \frac{q@p \rightarrow q' \in \Delta^\pi \quad r@s \rightarrow \mathbf{sel} \in \Delta_e}{(q, r)@(p, s) \rightarrow \mathbf{sel} \in \Delta_e^\pi} \\
\\
\frac{q@p \rightarrow q' \in \Delta^\pi \quad r@s \rightarrow r' \in \Delta_e \quad q \in P \quad r' \neq \mathbf{sel}}{(q, r)@(p, s) \rightarrow (q', r) \in \Delta_e^\pi} \\
\\
\frac{q \xrightarrow{\langle \rangle} q' \in \Delta^\pi \quad r \xrightarrow{\langle \rangle} r' \in \Delta_e \quad q \notin P \quad r' \neq \mathbf{sel}}{(q, r) \xrightarrow{\langle \rangle} (q', r') \in \Delta_e^\pi} \quad \frac{q \xrightarrow{\langle \rangle} q' \in \Delta^\pi \quad r \xrightarrow{\langle \rangle} \mathbf{sel} \in \Delta_e}{(q, r) \xrightarrow{\langle \rangle} \mathbf{sel} \in \Delta_e^\pi} \\
\\
\frac{q \xrightarrow{\langle \rangle} q' \in \Delta^\pi \quad r \xrightarrow{\langle \rangle} r' \in \Delta_e \quad q \in P \quad r' \neq \mathbf{sel}}{(q, r) \xrightarrow{\langle \rangle} (q', r) \in \Delta_e^\pi} \\
\\
\frac{a \in \Sigma \cup \{\langle \rangle\}}{\mathbf{sel} \xrightarrow{a} \mathbf{sel} \in \Delta_e^\pi} \quad \frac{\mu \in \mathcal{Q}_e^\pi}{\mu@\mathbf{sel} \rightarrow \mathbf{sel} \in \Delta_e^\pi}
\end{array}$$

Figure 9.1: The transition rules Δ_e^π inferred from those of the SHA^\downarrow s A^π with subhedge projection states P and A_e with selection state \mathbf{sel} .

hedges inside schema \mathbf{S} . Therefore the evaluator of A_e^π is earliest with respect to \mathbf{S} too. \square

The above proposition shows that if A^π is complete for subhedge projection then A_e^π is also complete for subhedge projection while being earliest in addition.

Theorem 6. *For any complete dSHA A with schema \mathbf{S} , the dSHA $A_{e(\mathbf{S})}^{cgr(\mathbf{S})}$ is earliest and sound and complete for subhedge projection for schema \mathbf{S} .*

Proof. The congruence projection $A^{cgr(\mathbf{S})}$ is sound by Proposition 3 and complete for subhedge projection wrt \mathbf{S} by Theorem 4. The automaton $A_{e(\mathbf{S})}$ discussed in Section 8.5 is earliest for \mathbf{S} by Theorem 5. So their combination $A_{e(\mathbf{S})}^{cgr(\mathbf{S})}$ yields an earliest automaton with complete subhedge projection wrt \mathbf{S} by Proposition 9.1. \square

9.3 Automaton Size

We next discuss the complexity of earliest membership testing with complete subhedge projection by an in-memory evaluator for $A_{e(\mathbf{S})}^{cgr(\mathbf{S})}$.

Lemma 9.2. *The number of states in $Q_{e(\mathbf{S})}^{cgr(\mathbf{S})}$ is in $O(2^{n^2+2n+\log(n)})$ where $n = |\mathcal{Q}|$.*

Proof. By Lemma 7.40 the number of states of $A^{cgr(\mathbf{S})}$ is in $O(2^{n^2+n})$ where $n = |\mathcal{Q}|$. The number of states of $A_{e(\mathbf{S})}$ is in $O(n2^n)$ and thus in $O(2^{n+\log(n)})$. The number of states of $Q_{e(\mathbf{S})}^{cgr(\mathbf{S})}$ is thus in $O(2^{n^2+n} 2^{n+\log(n)})$ which is in $O(2^{n^2+2n+\log(n)})$ \square

9.4 Benchmark dSHA for XPATH Queries

We start from dSHA defining monadic queries for the regular XPATH queries A1-A8 from the XPATHMARK benchmark [Al Serhali & Niehren 2022] that are given in Table 4.1. These XPATH queries show most of the features of the regular fragment of XPATH. In Table 9.2, we added 14 further XPATH path queries that we found useful for testing too.

Deterministic SHAS for A1-A8 were provided earlier in [Al Serhali & Niehren 2023a]. For the other XPATH queries we compiled them to dSHAS via nested regular expression.

In order to produce the input dSHAS for our evaluators, we intersect these automata with a dSHA for the schema $\llbracket \text{N-List}'_x \rrbracket \cap \llbracket \text{one}_x \rrbracket$ where $\text{N-List}'_x$ is the schema

A0	child::site
A1_0a	/site/*
A1_0b	/site/@*
A1_0c	/site//@*
A1_1a	//bidder/personref[starts-with(@person, 'person0')]
A1_1d	//bidder/personref[@person='person0']
A1_2	//person
A1_3	/site/regions/africa/@*
A1_4	/site/regions/africa/*
A1_5	/site/regions/*
A1_6	//closed_auction/annotation//keyword
A2_1	//closed_auction[descendant::keyword]
A4_0	/site/closed_auctions/closed_auction[annotation]/date
A4_1	/site[open_auctions]/closed_auctions

Figure 9.2: Additional regular XPATH queries for XPATHMARK documents.

we use for hedge encodings of real-world XML documents with x annotations. Thereby we could identify the schema final states F_S . We minimized and completed the result. Figure 9.3 reports the size of the input dSHAs for our evaluators obtained by the above procedure. For each dSHA A , the size is given by two numbers $\text{size}(n)$, the first for the overall size and the second for the number of states n .

For the input dSHA A of each query, we statically computed the whole $d\text{SHA}^\downarrow_s A_{e(S)}^{cgr(S)}$ while using the necessary parts of the determinization algorithm from [Niehren *et al.* 2022a]. The size of the $d\text{SHA}^\downarrow_s$ obtained and the number d of difference relations are reported in Figure 9.3 too. The biggest size is 2091(504) for the input dSHA for A8. The largest number of difference relations $d=24$ is also obtained for this query. So, indeed the size of these automata is much smaller than one might expect from a construction that is highly exponential in the worst case.

The time for computing the earliest congruence projection took between 2.3 and 26 seconds.

We note that we have not yet computed the complete $d\text{SHA}^\downarrow_s$ statically for safe-no-change projection A^{snc} , pure congruence projection $A^{cgr(S)}$ and earliest query answering $A_{e(S)}$. We believe that these automata may become bigger than $A_{e(S)}^{cgr(S)}$ that we constructed in a single shot.

Query ID	dSHA A m(n)	$A_{e(S)}^{cgr(S)}$ size(#states)	#diff- rel. d
A1	482(68)	1296(324)	16
A2	224(42)	316(82)	7
A3	320(53)	662(156)	10
A4	629(74)	1651(404)	18
A5	438(63)	1226(269)	13
A6	675(76)	2090(500)	22
A7	394(59)	728(184)	12
A8	648(79)	2091(504)	24
A0	203(40)	158(44)	5
A1_0a	224(42)	145(44)	6
A1_0b	203(39)	68(23)	5

Query ID	dSHA A m(n)	$A_{e(S)}^{cgr(S)}$ size(#states)	#diff- rel d
A1_0c	230(43)	238(62)	6
A1_1a	305(54)	384(101)	8
A1_1d	305(54)	382(101)	8
A1_2	194(39)	152(42)	5
A1_3	318(53)	672(159)	10
A1_4	312(52)	479(132)	10
A1_5	266(47)	293(84)	8
A1_6	265(47)	588(142)	9
A2_1	232(42)	295(78)	7
A4_0	392(59)	719(184)	12
A4_1	292(49)	287(78)	8

Figure 9.3: Size measures of deterministic automata for regular XPATHMARK queries: $size(\#states)$, where $\#states$ is the number of states and $size$ the overall size of the automaton. For the input dSHA A we have $n = \#states$ and $m = size$. It is obtained by minimizing the accessible product of the query's dSHA with the dSHA of the schema $S = xml\text{-}seq\&one\text{-}x$. We also show the overall size and number of states of the earliest $dSHA^\downarrow$ with complete subhedge and suffix projection $A_{e(S)}^{cgr(S)}$ obtained by earliest congruence projection, and the number d of difference relations used for congruence projection.

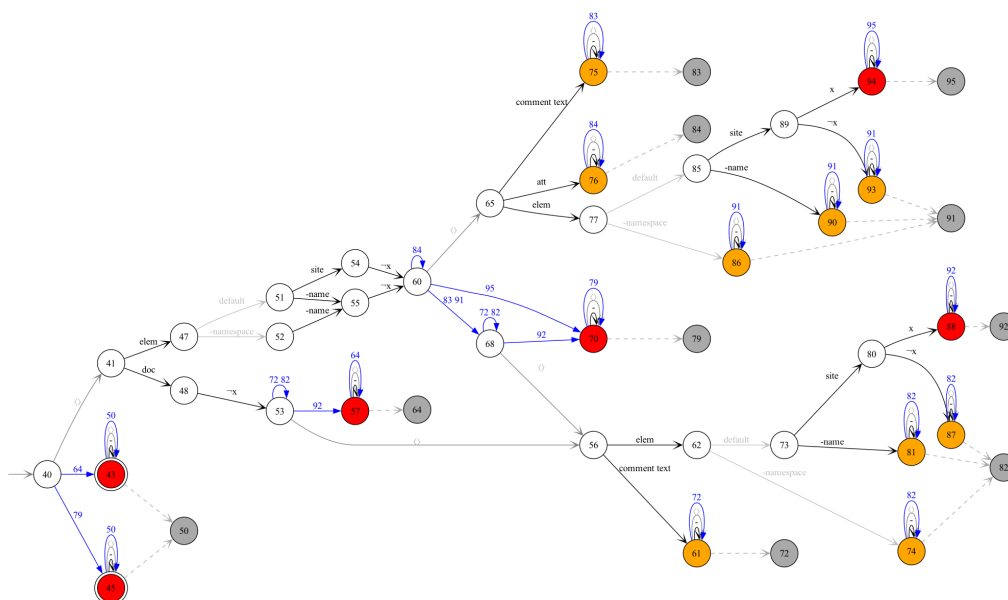


Figure 9.4: The earliest $d_{\text{SHA}}^\downarrow$ with complete subhedge and suffix projection for the XPATH query $A0=\text{child}::\text{site}$.

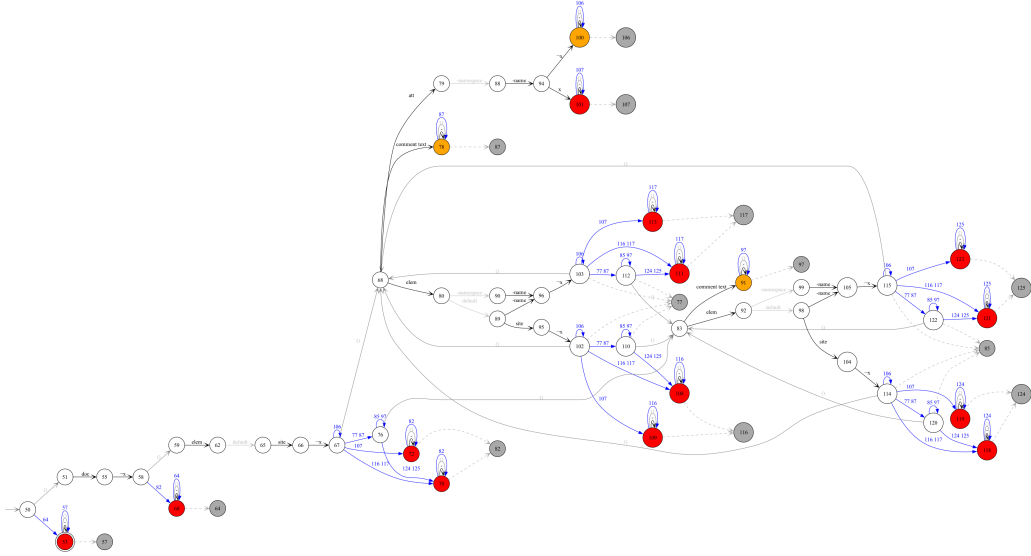


Figure 9.5: The earliest $d\text{SHA}^\downarrow$ with complete subhedge and suffix projection for the XPATH query $A1_0c=/site/@*$.

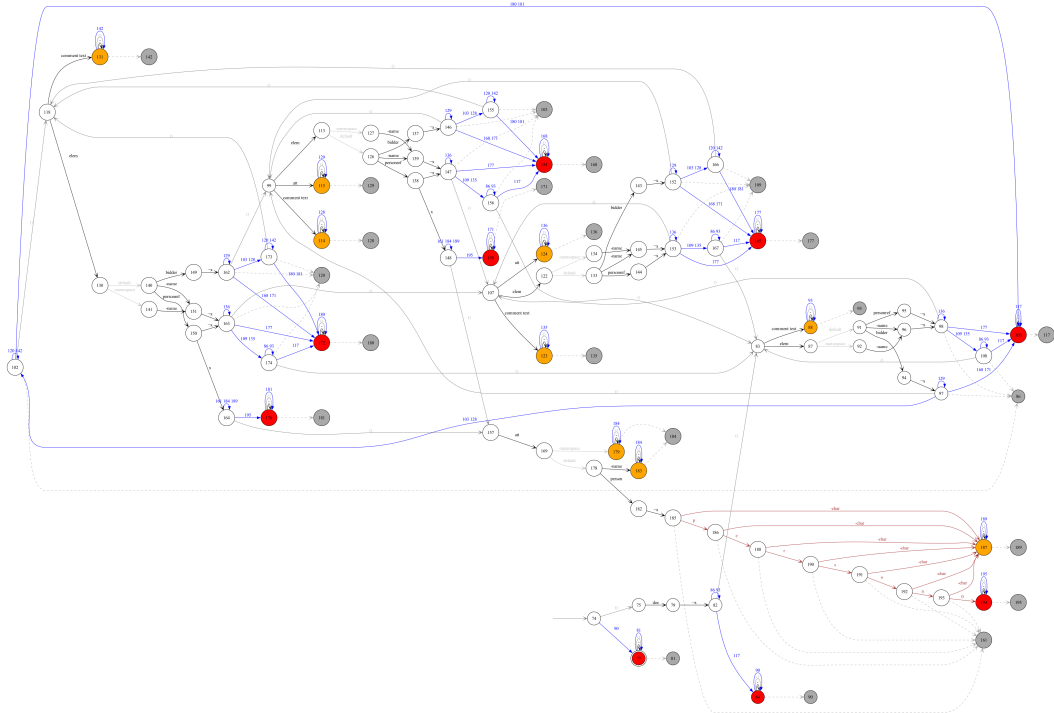


Figure 9.6: The earliest $d\text{SHA}^\downarrow$ with complete subhedge and suffix projection for the XPATH query $A1_1a=/bidder/personref[starts-with(@person,'person0')]$

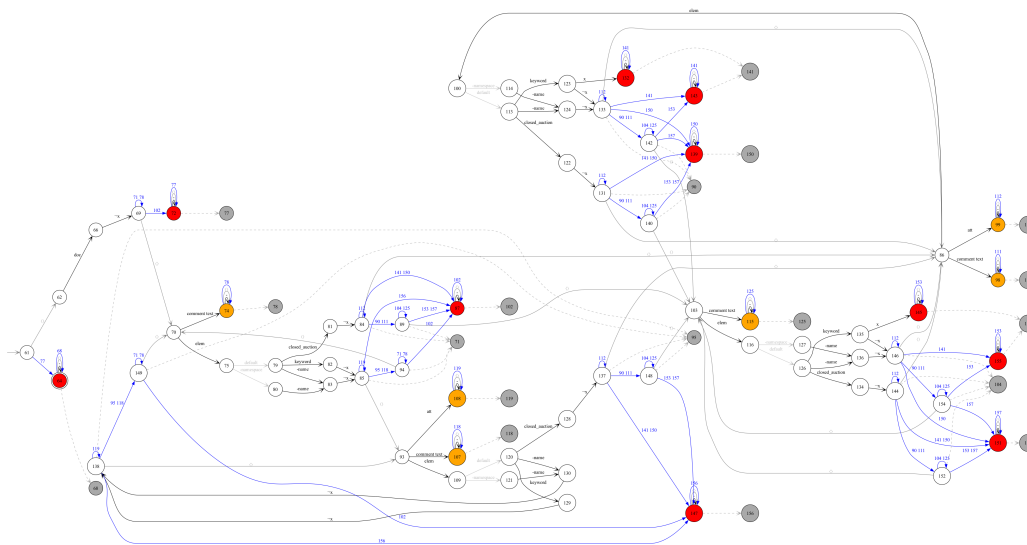


Figure 9.7: The earliest $d\text{SHA}^\downarrow$ with complete subhedge and suffix projection for the XPath query $A2=//closed_auction//keyword$

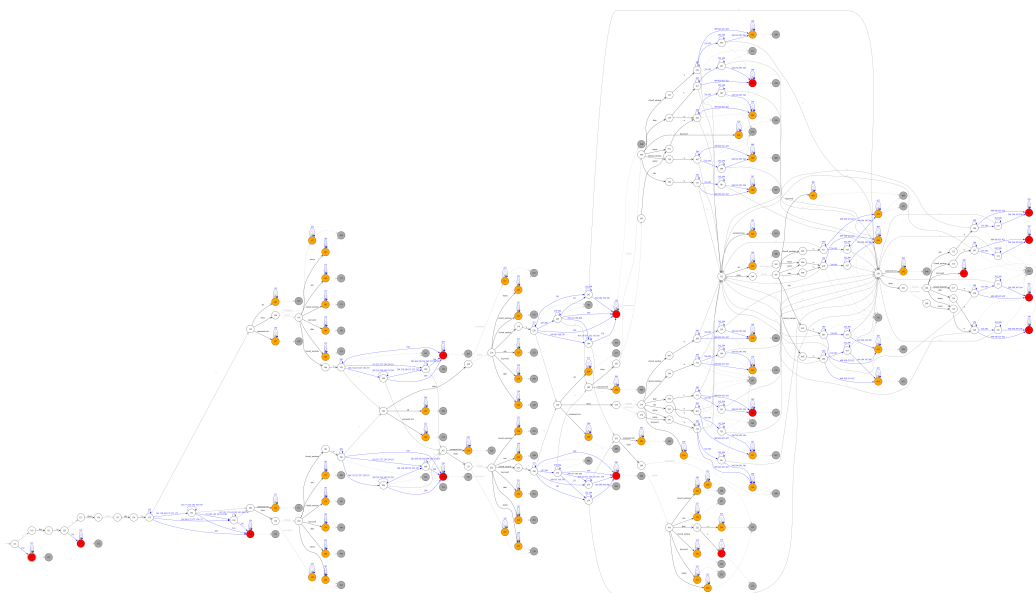


Figure 9.8: The earliest $d\text{SHA}^\downarrow$ with complete subhedge and suffix projection for XPathMark's XPath query:

$A5=/site/closed_auctions/closed_auction \ [descendant::keyword]/date$

Chapter 10

Projecting Evaluators for Earliest Membership

Abstract

We present top-down evaluators with subhedge and suffix projection for $d\text{SHA}s$ that may either run in-memory or in streaming mode. They recognize certain membership and non-membership at the earliest possible event. Both evaluators run in constant time per non-projected event of the input hedge, after a possibly exponential precomputation.

Contents

10.1	Early Evaluators with Projection	245
10.1.1	In-Memory Evaluator	246
10.1.2	Streaming Evaluator	249
10.2	Earliest Membership with Projection	254
10.2.1	In-Memory Complexity	254
10.2.2	Streaming Complexity	255

10.1 Early Evaluators with Projection

We develop early deterministic top-down evaluators with subhedge and suffix projection for $d\text{SHA}^\downarrow$ s in two modes. We start with an in-memory evaluator and then adapt it to the streaming mode.

$$\begin{aligned}
\llbracket h \rrbracket_{shp}(q) &= \begin{cases} q & \text{if } q \in \mathcal{Q}_{shp}^\Delta \\ \llbracket h \rrbracket_{shp'}(q) & \text{else} \end{cases} \\
\llbracket \epsilon \rrbracket_{shp'}(q) &= q \\
\llbracket a \rrbracket_{shp'}(q) &= a^\Delta(q) \\
\llbracket h \cdot h' \rrbracket_{shp'}(q) &= \llbracket h' \rrbracket_{shp}(\llbracket h \rrbracket_{shp}(q)) \\
\llbracket \langle h \rangle \rrbracket_{shp'}(q) &= q @^\Delta(\llbracket h \rrbracket_{shp}(\langle \rangle^\Delta(q)))
\end{aligned}$$

Figure 10.1: The in-memory top-down evaluator with subhedge projection $\llbracket h \rrbracket_{shp}^\Delta = \llbracket h \rrbracket_{shp} : \mathcal{Q} \hookrightarrow \mathcal{Q}$ for a $d\text{SHA}^\downarrow A = (\mathcal{Q}, \Sigma, \Delta, I, F)$ and a hedge $h \in \mathcal{H}_\Sigma$ (See Definition 7.9 for \mathcal{Q}_{shp}^Δ).

10.1.1 In-Memory Evaluator

We next show how to refine the transitions for SHA^\downarrow s with distinguished selection states by early selection and subhedge projection. This yields a in-memory top-down evaluator that reports certain membership once reaching some selection state, while projecting irrelevant subhedges.

10.1.1.1 Adding Subhedge Projection

We start with the deterministic in-memory top-down evaluator for $d\text{SHA}^\downarrow$ s and show how to refine it with subhedge projection. Note that this refinement does neither support suffix projection nor early output.

For any hedge $h \in \mathcal{H}_\Sigma$ and $d\text{SHA}^\downarrow A = (\mathcal{Q}, \Sigma, \Delta, I, F)$, we presented in Section 3.4.1 the deterministic in-memory top-down evaluator:

$$\llbracket h \rrbracket^\Delta = \llbracket h \rrbracket : \mathcal{Q} \rightarrow \mathcal{Q}$$

The next objective is to refine this evaluator such that it jumps over subhedges whenever reaching some subhedge projection state. This leads to an in-memory top-down evaluator with subhedge projection of type

$$\llbracket h \rrbracket_{shp}^\Delta = \llbracket h \rrbracket_{shp} : \mathcal{Q} \hookrightarrow \mathcal{Q}$$

that we define in Figure 10.1. The first rule says that subhedge projecting transitions stay in subhedge projection states until the end of the current subhedge is reached. This is correct by Lemma 7.10 under the condition that there are no blocking runs. The other rules state that the evaluator behaves as without subhedge projection

otherwise.

Lemma 10.1. *For all $dSHA^\downarrow$ s $A = (\mathcal{Q}, \Sigma, \Delta, I, F)$ that is schema-complete for \mathbf{S} , states $q \in \mathcal{Q}$, and hedges $h \in \mathbf{S}$:*

$$\llbracket h \rrbracket_{shp}(q) = \llbracket h \rrbracket(q)$$

Proof. We distinguish two cases:

Case $q \notin \mathcal{Q}_{shp}^\Delta$. Then for any $p \in \mathcal{Q}$; $\llbracket h \rrbracket_{shp}(q) = q = \llbracket h \rrbracket(q)$ by definition of the projecting and non-projecting transitions.

Case $q \in \mathcal{Q}_{shp}^\Delta$. Then $\llbracket h \rrbracket_{shp}(q) = q$. Since $h \in \mathbf{S}$ and A is schema-complete for \mathbf{S} there exists some run of A on h , and there exists q' such that $q' = \llbracket h \rrbracket(q)$ is well-defined. Since the deterministic and nondeterministic evaluators behave the same by Lemma 3.37, it follows that $q \xrightarrow{h} q'$ wrt Δ . Lemma 7.10 shows that $q \xrightarrow{h} q$ wrt Δ . Again by Lemma 3.37, this is equivalent to $q = \llbracket h \rrbracket(q)$ and thus:

$$\llbracket h \rrbracket_{shp}(q) = q = \llbracket h \rrbracket(q)$$

□

10.1.1.2 Adding Early Output

We next refine the in-memory top-down evaluator with subhedge projection by adding early output once reaching some selection state. When this happens, membership is certain. The following suffix is thus irrelevant and can be projected. Suffix projection also applies if the run blocks, since then non-membership is certain.

Let \mathbf{S} be a schema for A , and $Q \subseteq \mathcal{Q}$ a subset of selection states of A wrt. schema \mathbf{S} . Let sel be a fresh symbol. We start with a function that raises the exception sel when applied to some state in Q and otherwise returns the state:

$$raise\text{-}sel_Q(q) = \begin{cases} raise\ sel & \text{if } q \in Q \\ q & \text{otherwise} \end{cases}$$

In Figure 10.2 we define the in-memory top-down evaluator with early output and suffix and subhedge projection, which has the type:

$$\llbracket h \rrbracket_{prj(Q)}^\Delta = \llbracket h \rrbracket_{prj(Q)} : \mathcal{Q} \hookrightarrow \mathcal{Q} \cup \{sel\}$$

$$\begin{aligned}
\llbracket h \rrbracket_{prj(Q)}(q) &= \begin{cases} \text{try } raise\text{-}sel_Q(q) \text{ catch } sel \text{ then } sel & \text{if } q \in \mathcal{Q}_{shp}^\Delta \\ \text{try } \llbracket h \rrbracket_{prj'(Q)}(raise\text{-}sel_Q(q)) \text{ catch } sel \text{ then } sel & \text{else} \end{cases} \\
\llbracket \epsilon \rrbracket_{prj'(Q)}(q) &= q \\
\llbracket h \cdot h' \rrbracket_{prj'(Q)}(q) &= raise\text{-}sel_Q(\llbracket h' \rrbracket_{prj(Q)}(raise\text{-}sel_Q(\llbracket h \rrbracket_{prj(Q)}(q)))) \\
\llbracket a \rrbracket_{prj'(Q)}(q) &= raise\text{-}sel_Q(a^\Delta(q)) \\
\llbracket \langle h \rangle \rrbracket_{prj'(Q)}(q) &= raise\text{-}sel_Q(q @^\Delta (raise\text{-}sel_Q(\llbracket h \rrbracket_{prj(Q)}(raise\text{-}sel_Q(\langle \rangle^\Delta(q)))))
\end{aligned}$$

Figure 10.2: The early top-down in-memory evaluator with subhedge and suffix projection for a $dSHA^\downarrow A = (\Sigma, \mathcal{Q}, \Delta, I, F)$ with schema \mathbf{S} , a subset of selection states Q for A wrt. schema \mathbf{S} and a hedges $h, h' \in \mathcal{H}_\Sigma$.

Whenever it moves to some state q , it applies the function $raise\text{-}sel_Q(q)$, which raises exception sel if $q \in Q$ and continues otherwise with q . So up to detecting whether the current state belongs to Q and raising exception sel in this case, the evaluator $\llbracket h \rrbracket_{prj(Q)}^\Delta$ behaves equal to $\llbracket h \rrbracket_{shp}^\Delta$. Raising exception sel is correct by definition of selection states wrt. \mathbf{S} : it requires that any partial run ending in some selection state can be completed successfully for all hedges belonging to the schema \mathbf{S} . So once the exception is raised, membership can be reported in an early manner. We do so by returning the marker sel .

Lemma 10.2. *Let $A = (\Sigma, \mathcal{Q}, \Delta, I, F)$ be a $dSHA^\downarrow$ that is schema-complete for \mathbf{S} and $Q \subseteq \mathcal{Q}$. For any state $q \in \mathcal{Q}$, if no partial Δ -run on h that starts in q ends in Q then:*

$$\llbracket h \rrbracket_{shp}^\Delta(q) = \llbracket h \rrbracket_{prj(Q)}^\Delta(q)$$

Proof. If no partial Δ -run on h starting in q ends in Q , then the evaluator $\llbracket h \rrbracket_{prj(Q)}^\Delta(q)$ does not raise any exception, so all its calls of $raise\text{-}sel_Q$ can be ignored. When replacing $raise\text{-}sel_Q$ by the identity in the definition of $\llbracket h \rrbracket_{prj(Q)}^\Delta$ then it becomes identical to $\llbracket h \rrbracket_{shp}^\Delta$. Hence:

$$\llbracket h \rrbracket_{shp}^\Delta(q) = \llbracket h \rrbracket_{prj(Q)}^\Delta(q)$$

□

Proposition 10.3. *Let $A = (\Sigma, \mathcal{Q}, \Delta, I, F)$ be a $dSHA^\downarrow$ that is schema-complete for \mathbf{S} and $Q \subseteq \mathcal{Q}$ be a subset of selection states. Then for all hedges $h \in \mathbf{S}$:*

$$h \in \mathcal{L}(A) \Leftrightarrow \llbracket h \rrbracket_{prj(Q)}^\Delta(I) \cap (F \cup \{sel\}) \neq \emptyset$$

Proof. Case 1: Suppose that some partial Δ -run on hedge $h \in \mathbf{S}$ starting with q ends in Q . By Lemma 8.14, and since we assume schema-completeness and that each state in Q is a selection state of A for \mathbf{S} , and $h \in \mathbf{S}$, this implies $h \in \mathcal{L}(A)$. Furthermore, the call of the evaluator $\llbracket Q \rrbracket_{prj'(h)}(I)$ raises the exception *sel*, so $\llbracket Q \rrbracket_{prj(h)}(I) = \text{sel}$. Hence, $\llbracket h \rrbracket_{prj(Q)}^\Delta(I) \cap \{\text{sel}\} \neq \emptyset$.

Case 2: Otherwise, no partial Δ -run on hedge $h \in \mathbf{S}$ starting with q ends in Q . Lemma 10.2 thus show that:

$$\llbracket h \rrbracket_{shp}^\Delta(I) = \llbracket h \rrbracket_{prj(Q)}^\Delta(I)$$

Furthermore, by Lemma 10.1 and schema-completeness we have:

$$\llbracket h \rrbracket_{shp}^\Delta(I) = \llbracket h \rrbracket^\Delta(I)$$

We can therefore conclude as follows:

$$\begin{aligned} h \in \mathcal{L}(A) &\Leftrightarrow \llbracket h \rrbracket^\Delta(I) \cap F \neq \emptyset \\ &\Leftrightarrow \llbracket h \rrbracket_{shp}^\Delta(I) \cap F \neq \emptyset \\ &\Leftrightarrow \llbracket h \rrbracket_{prj(Q)}^\Delta(I) \cap F \neq \emptyset \end{aligned}$$

□

We note that evaluating nondeterministic SHA^\downarrow s deterministically can be done based on determinization. Note, however, that our objective was to avoid determinization of SHA^\downarrow s since it raises additional challenges implying complexity issues compared to the determinization of SHAS .

10.1.2 Streaming Evaluator

We extend the streaming transition $\llbracket v \rrbracket_{str}^\Delta$ for $d\text{SHA}^\downarrow$ s $A = (Q, \Sigma, \Delta, I, F)$ from Section 3.4.2 with subhedge projection to $\llbracket v \rrbracket_{str,shp}^\Delta$, in analogy to how we extended the deterministic in-memory evaluator $\llbracket h \rrbracket^\Delta$ to $\llbracket h \rrbracket_{shp}^\Delta$.

In Figure 10.3 we define the streaming evaluator with subhedge projection with respect to Δ . It has the following type for any nested word factor $v \in \hat{\Sigma}^*$:

$$\llbracket v \rrbracket_{str,shp}^\Delta = \llbracket v \rrbracket_{str,shp} : \mathcal{K} \hookrightarrow \mathcal{K} \text{ where } \mathcal{K} = Q \times Q^*$$

The first equation in Figure 10.3 states that the subhedge projecting evaluator

$$\begin{aligned}
\llbracket v \rrbracket_{str,shp}(q, \sigma) &= \begin{cases} (q, \sigma) & \text{if } v \in \mathcal{N}_\Sigma \text{ \& } q \in \mathcal{Q}_{shp}^\Delta \\ \llbracket v \rrbracket_{str,shp'}(q, \sigma) & \text{if } q \notin \mathcal{Q}_{shp}^\Delta \end{cases} \\
\llbracket \epsilon \rrbracket_{str,shp'}(q, \sigma) &= (q, \sigma) \\
\llbracket v \cdot v' \rrbracket_{str,shp'}(q, \sigma) &= \llbracket v' \rrbracket_{str,shp}(\llbracket v \rrbracket_{str,shp'}(q, \sigma)) \\
\llbracket a \rrbracket_{str,shp'}(q, \sigma) &= (a^\Delta(q), \sigma) \\
\llbracket \langle \rrbracket_{str,shp'}(q, \sigma) &= (\langle \rangle^\Delta(q), \sigma \cdot q) \\
\llbracket \langle \rrbracket_{str,shp'}(p, \sigma \cdot q) &= (q @^\Delta p, \sigma)
\end{aligned}$$

Figure 10.3: The streaming evaluator with subhedge projection for a $d\text{SHA}^\downarrow A = (\mathcal{Q}, \Sigma, \Delta, I, F)$, where $v, v' \in \hat{\Sigma}^*$ are nested word factors, $a \in \Sigma$ a letter, $p, q \in \mathcal{Q}$ states, and $\sigma \in \mathcal{Q}^*$ stacks.

jumps over nested words if the current state is a subhedge projection state. This is correct since a subhedge projection state cannot be changed by any subhedge. Otherwise, it behaves as the previous streaming evaluator.

Proposition 10.4. *For any $\text{SHA}^\downarrow A = (\Sigma, \mathcal{Q}, \Delta, I, F)$, state $q \in \mathcal{Q}$, and nested word $v \in \mathcal{N}_\Sigma$ such that Δ has no blocking partial run on $\text{hdg}(v)$ starting from q :*

$$\llbracket v \rrbracket_{str,shp}(q, \epsilon) = \llbracket v \rrbracket_{str}(q, \epsilon)$$

Proof. The proof is by induction on the structure of $h = \text{hdg}(v)$. Hence $v = nw(h)$, so that Lemma 3.39 yields:

$$\llbracket v \rrbracket_{str}(q, \epsilon) = (\llbracket h \rrbracket(q), \epsilon)$$

Since we assume that A does not have any blocking run on h starting with q it follows that $\llbracket h \rrbracket(q)$ is defined. Let $q' = \llbracket h \rrbracket(q)$. Since the nondeterministic transitions correspond to the deterministic evaluator by Lemma 3.37, it follows that $q \xrightarrow{h} q'$ wrt Δ .

Subcase $q \in \mathcal{Q}_{shp}^\Delta$. Since A has no blocking partial run on h , Lemma 7.10 shows that $q \xrightarrow{h} q'$ wrt Δ implies $q = q'$. By definition of streaming subhedge projecting transitions we have:

$$\llbracket v \rrbracket_{str,shp}(q, \epsilon) = (q, \epsilon)$$

Hence:

$$\llbracket v \rrbracket_{str,shp}(q, \epsilon) = (q, \epsilon) = (\llbracket h \rrbracket(q), \epsilon) = \llbracket h \rrbracket_{str}(q, \epsilon)$$

Subcase $q \notin \mathcal{Q}_{shp}^\Delta$. By definition of the streaming evaluator with subhedge projec-

tion we have:

$$\llbracket v \rrbracket_{str,shp}(q, \epsilon) = \llbracket v \rrbracket_{str,shp'}(q, \epsilon)$$

We distinguish all possible forms of hedge h .

Subsubcase $h = h' \cdot h''$. Let $v' = nw(h')$ and $v'' = nw(h'')$. Thus, $v = v' \cdot v'' = nw(h)$. The definition of the streaming evaluator with subhedge projection shows:

$$\begin{aligned} \llbracket v'' \cdot v' \rrbracket_{str,shp}(q, \epsilon) &= \llbracket v' \rrbracket_{str,shp}(\llbracket v'' \rrbracket_{str,shp'}(q, \epsilon)) \\ &= \llbracket v' \rrbracket_{str,shp}(\llbracket v'' \rrbracket_{str,shp}(q, \epsilon)) \end{aligned}$$

By induction hypothesis applied to the nested words v' and v'' we get:

$$\begin{aligned} \llbracket v'' \rrbracket_{str,shp}(q, \epsilon) &= \llbracket v'' \rrbracket_{str}(q, \epsilon) \\ \llbracket v' \rrbracket_{str,shp}(\llbracket v'' \rrbracket_{str}(q, \epsilon)) &= \llbracket v \rrbracket_{str}(\llbracket v'' \rrbracket_{str}(q, \epsilon)) \end{aligned}$$

Hence, we can close this case as follows:

$$\llbracket v \rrbracket_{str,shp}(q, \epsilon) = \llbracket v \rrbracket_{str}(q, \epsilon)$$

Subsubcases $h = \langle h' \rangle$ or $h = a$ or $h = \epsilon$. Straightforward.

□

A streaming evaluator with subhedge projection for deterministic SHA^\downarrow s A on hedges h can thus be obtained by computing the streaming transition relation with subhedge projection for A of $nw(h)$ starting with the initial configuration. This costs time at most $O(1)$ per letter of $nw(h)$, i.e. constant time per event of the stream.

10.1.2.1 Adding Early Output

We next refine the streaming evaluator with subhedge projection by adding early output once reaching some selection state. This works basically the same way as in the in-memory case.

Let $A = (\mathcal{Q}, \Sigma, \Delta, I, F)$ be a $d\text{SHA}^\downarrow$. For any configuration in $(q, \sigma) \in \mathcal{K} = \mathcal{Q} \times \mathcal{Q}^*$ we define:

$$\text{raise-sel}_Q^K(q, \sigma) = (\text{raise-sel}_Q(q), \sigma)$$

Let \mathbf{S} be a schema for A , and $Q \subseteq \mathcal{Q}$ a subset of selection states of A wrt. schema \mathbf{S} . In Figure 10.4 we define the streaming evaluator with early output and suffix

$$\begin{aligned}
\llbracket v \rrbracket_{str,prj(Q)}(\kappa) &= \begin{cases} \text{try } raise\text{-}sel_Q^K(\kappa) \text{ catch } sel \text{ then } sel & \text{if } v \in \mathcal{N}_\Sigma \text{ \& } \kappa \in \mathcal{Q}_{shp}^\Delta \times \mathcal{Q}^* \\ \text{try } \llbracket Q \rrbracket_{str,prj'(v)}(\kappa) \text{ catch } sel \text{ then } sel & \text{else} \end{cases} \\
\llbracket \epsilon \rrbracket_{str,prj'(Q)}(\kappa) &= \kappa \\
\llbracket v \cdot v' \rrbracket_{str,prj'(Q)}(\kappa) &= raise\text{-}sel_Q^K(\llbracket v' \rrbracket_{str,prj(Q)}(raise\text{-}sel_Q^K(\llbracket v \rrbracket_{str,prj(Q)}(\kappa)))) \\
\llbracket a \rrbracket_{str,prj'(Q)}(\kappa) &= raise\text{-}sel_Q^K(\llbracket a \rrbracket_{str,shp}^\Delta(\kappa)) \\
\llbracket \langle \rangle \rrbracket_{str,prj'(Q)}(\kappa) &= raise\text{-}sel_Q^K(\llbracket \langle \rangle \rrbracket_{str,shp}^\Delta(\kappa)) \\
\llbracket \rangle \rrbracket_{str,prj'(Q)}(\kappa) &= raise\text{-}sel_Q^K(\llbracket \rangle \rrbracket_{str,shp}^\Delta(\kappa))
\end{aligned}$$

Figure 10.4: The early streaming evaluator with subhedge and suffix projection for a $dSHA^\downarrow A = (\mathcal{Q}, \Sigma, \Delta, I, F)$ with schema \mathbf{S} , a subset of selection states Q for A wrt. schema \mathbf{S} , nested word factors $v, v' \in \hat{\Sigma}^*$, and configurations $\kappa \in \mathcal{K} = \mathcal{Q} \times \mathcal{Q}^*$.

and subhedge projection of the following type for any nested word factor $v \in \hat{\Sigma}^*$:

$$\llbracket v \rrbracket_{str,prj(Q)}^\Delta = \llbracket v \rrbracket_{str,prj(Q)} : \mathcal{K} \hookrightarrow \mathcal{K} \cup \{sel\} \text{ where } \mathcal{K} = \mathcal{Q} \times \mathcal{Q}^*$$

Whenever the early streaming evaluator with projection moves to some configuration $\kappa \in \mathcal{K}$, it applies the function $raise\text{-}sel_Q^K(\kappa)$, which raises the exception sel if $\kappa \in \mathcal{Q} \times \mathcal{Q}^*$ and continues with κ otherwise. So up to detecting whether the state of the current configuration belongs to Q and then raising the exception sel , the evaluator $\llbracket h \rrbracket_{str,prj(Q)}^\Delta$ behaves exactly like $\llbracket h \rrbracket_{str,shp}^\Delta$. Raising the exception sel is correct by definition of selection states wrt. \mathbf{S} : it requires that any partial run ending in some selection state can be completed successfully for all hedges belonging to the schema \mathbf{S} . So once the exception is raised, certain membership can be reported in an early manner. We do so by returning the marker sel .

Lemma 10.5. *Let $A = (\Sigma, \mathcal{Q}, \Delta, I, F)$ be a $dSHA^\downarrow$ that is schema-complete for \mathbf{S} , $Q \subseteq \mathcal{Q}$, and $v \in \mathcal{N}_\Sigma$. For any configuration $\kappa = (q, \sigma) \in \mathcal{K} = \mathcal{Q} \times \mathcal{Q}^*$ so that no Δ -run on $hdg(v)$ starting in q does end in Q :*

$$\llbracket v \rrbracket_{str,shp}^\Delta(\kappa) = \llbracket v \rrbracket_{str,prj(Q)}^\Delta(\kappa)$$

Proof. If no partial Δ -run on $hdg(v)$ starting in q does end in Q , then the evaluator $\llbracket h \rrbracket_{str,prj(Q)}^\Delta(\kappa)$ does not raise any exception, so all its calls $raise\text{-}sel_Q^K$ can be safely ignored. When replacing $raise\text{-}sel_Q^K$ by the identity in the definition of $\llbracket h \rrbracket_{str,prj(Q)}^\Delta$ then it becomes identical to $\llbracket h \rrbracket_{str,shp}^\Delta$. Hence:

$$\llbracket h \rrbracket_{str,shp}^\Delta(\kappa) = \llbracket h \rrbracket_{str,prj(Q)}^\Delta(\kappa)$$

□

Proposition 10.6. *Let $A = (\Sigma, \mathcal{Q}, \Delta, I, F)$ be a SHA^\downarrow that is schema-complete for \mathbf{S} and $Q \subseteq \mathcal{Q}$ be a subset of selection states. Then for all hedges $h \in \mathbf{S}$ and states $q \in \mathcal{Q}$:*

$$h \in \mathcal{L}(A) \Leftrightarrow \llbracket nw(h) \rrbracket_{str, prj(Q)}^\Delta (I \times \{\epsilon\}) \cap ((F \times \{\epsilon\}) \cup \{sel\}) \neq \emptyset$$

Proof. If $I = \emptyset$ then the proposition is trivial. Therefore, we can assume without loss of generality that there exists $q_{init} \in \mathcal{Q}$ such that $I = \{q_{init}\}$.

Case 1: some partial Δ -run on hedge $h \in \mathbf{S}$ starting with q does end in Q . By Lemma 8.14, this implies $h \in \mathcal{L}(A)$. In this case, the call of the evaluator $\llbracket nw(h) \rrbracket_{str, prj'(Q)}(q_{init}, \epsilon)$ raises the exception sel , so that:

$$\llbracket nw(h) \rrbracket_{str, prj(Q)}(q_{init}, \epsilon) = sel$$

Hence, in this case $\llbracket nw(h) \rrbracket_{str, prj(Q)}^\Delta (I \times \{\epsilon\}) \cap \{sel\} \neq \emptyset$.

Case 2: no partial Δ -run on hedge $h \in \mathbf{S}$ starting in q ends in Q . Lemma 10.5 then shows that:

$$\llbracket nw(h) \rrbracket_{str, shp}^\Delta (q_{init}, \epsilon) = \llbracket nw(h) \rrbracket_{str, prj(Q)}^\Delta (q_{init}, \epsilon)$$

Furthermore, by Lemma 10.1 we have:

$$\llbracket nw(h) \rrbracket_{str, shp}^\Delta (q_{init}, \epsilon) = (\llbracket nw(h) \rrbracket_{str}^\Delta (q_{init}), \epsilon)$$

We can therefore conclude as follows:

$$\begin{aligned} h \in \mathcal{L}(A) &\Leftrightarrow \llbracket nw(h) \rrbracket_{str}^\Delta (I \times \{\epsilon\}) \cap (F \times \{\epsilon\}) \neq \emptyset \\ &\Leftrightarrow \llbracket nw(h) \rrbracket_{str, shp}^\Delta (I \times \{\epsilon\}) \cap (F \times \{\epsilon\}) \neq \emptyset \\ &\Leftrightarrow \llbracket nw(h) \rrbracket_{str, prj(Q)}^\Delta (I \times \{\epsilon\}) \cap (F \times \{\epsilon\}) \neq \emptyset \end{aligned}$$

□

Finally, note that nondeterministic SHA^\downarrow s can be evaluated on streams based on determinization. However, we prefer to avoid SHA^\downarrow determinization, since it raises additional challenges compared to the determinization of SHAS .

10.2 Earliest Membership with Projection

By applying the early evaluators from Section 10.1 to the earliest SHA^\downarrow s with subhedge projection $A_{e(S)}^{cgr(S)}$ from Chapter 9 we can test membership in an earliest manner and with complete subhedge and prefix projection.

10.2.1 In-Memory Complexity

We start by using the early in-memory early evaluator with projection from Section 10.1.1.

Corollary 10.7. *We fix a signature Σ . For any complete $d\text{SHA}$ $A = (\mathcal{Q}, \Sigma, \Delta, I, F)$ with schema $S \subseteq \mathcal{H}_\Sigma$ and hedge $h \in S$, membership $h \in \mathcal{L}(A)$ can be decided at the earliest time point when it becomes certain. In-memory, this requires time $O(1)$ per non-projected node of h after a preprocessing time of $O(|A_{e(S)}^{cgr(S)}| + n^3 d + m s)$, where d is the number of difference relations, s the number of subsets of safe states in $\mathcal{Q}_{e(S)}^{cgr(S)}$, and $n = |\mathcal{Q}|$, and m the overall size of A . The preprocessing time is also in $O(2^{2n^2+4n+2\log(n)} + n^3 d + m s)$.*

Proof. Since A is complete, Theorem 6 shows that $A_{e(S)}^{cgr(S)}$ is earliest and complete for subhedge projection for schema S . The early in-memory evaluator with projection of $A_{e(S)}^{cgr(S)}$ can be run on any input hedge $h \in S$ to compute $\llbracket h \rrbracket_{prj(\text{sel})}$. This requires time $O(1)$ per non-projected node of h after a preprocessing time of $O(|A_{e(S)}^{cgr(S)}| + n^3 d + m s)$. Since Σ is fixed, the size $|A_{e(S)}^{cgr(S)}|$ is bounded by $O(|\mathcal{Q}_{e(S)}^{cgr(S)}|^2)$. Lemma 9.2 shows that $|\mathcal{Q}_{e(S)}^{cgr(S)}|$ is in $O(2^{n^2+2n+\log(n)})$, so $|A_{e(S)}^{cgr(S)}|$ is in $O(2^{2n^2+4n+2\log(n)})$, and thus also the preprocessing time is also in $O(2^{2n^2+4n+2\log(n)} + n^3 d + m s)$. \square

The upper bound $O(|A_{e(S)}^{cgr(S)}| + n^3 d + m s)$ is often feasible in practice of $d\text{SHAS}$ for regular XPath queries, as we illustrated already in Figure 9.3 for the regular XPathMark queries. For these queries we have $n \leq 2d$. Therefore, the dominating part is $n^3 d$ which in the worst case of A8 for the XPathMark is around 11.8 million.

Should this not be the case, then we can still avoid precomputing $A_{e(S)}^{cgr(S)}$ from the input $d\text{SHA}$ A and F_S statically. As before, we can compute only the part of this automaton needed for evaluating the input hedge h dynamically on-the-fly. If this part is small, then the overall time and space will go down considerably. Preprocessing will no more be needed at all, but the needed part of it has to be done at running time. This part may be way smaller than $O(|A_{e(S)}^{cgr(S)}| + n^3 d + m s)$ though.

10.2.2 Streaming Complexity

We next use the early streaming evaluator with projection from Section 10.1.2.

Corollary 10.8. *Let $A = (\mathcal{Q}, \Sigma, \Delta, I, F)$ be a complete $d\text{SHA}$ with language $L = \mathcal{L}(A)$, and $F \subseteq F_S \subseteq \mathcal{Q}$ define schema $S = \mathcal{L}(A[F/F_S])$. Earliest membership with complete subhedge projection for L wrt S can be tested in streaming mode for any hedge $h \in S$ in time $O(1)$ per non-projected letter of $nw(h)$ after a preprocessing time of $O(|A_{e(S)}^{cgr(S)}| + n^3 d + m s)$, where d is the number of difference relations and s the number of safe subsets of states in $\mathcal{Q}_{e(S)}^{cgr(S)}$. The space required is in $O(\text{depth}(h) + |A_{e(S)}^{cgr(S)}|)$.*

Proof. By Theorem 6 the $d\text{SHA}^\downarrow A_{e(S)}^{cgr(S)}$ is earliest and sound and complete for subhedge projection. By Lemma 3.39 and Proposition 10.4 we can thus obtain a earliest streaming membership tester with complete subhedge and suffix projection for a hedge $h \in S$ by running the early streaming evaluator with projection of $A_{e(S)}^{cgr(S)}$ on the nested word $nw(h)$, i.e., by computing $\llbracket nw(h) \rrbracket_{str, prj(\text{sel})}$. Since this SHA^\downarrow is deterministic, the streaming evaluation can be done in time $O(1)$ per letter of $nw(h)$ after a preprocessing time of $O(|A_{e(S)}^{cgr(S)}| + n^3 d + m s)$. \square

As for the in-memory case, the upper bound $O(|A_{e(S)}^{cgr(S)}| + n^3 d + m s)$ is often feasible in practice of $d\text{SHAs}$ for regular XPath queries, as we will illustrated already in Figure 9.3 for the regular XPathMark queries. The dominating part for precomputing $A_{e(S)}^{cgr(S)}$ is again the term $n^3 d$. Should the static precomputation of $A_{e(S)}^{cgr(S)}$ not be possible, then it is sufficient to create only the needed part of the $\text{SHA}^\downarrow A_{e(S)}^{cgr(S)}$ for the evaluation of the hedge h dynamically on the fly.

Finally notice that the upper bound for the memory consumption $O(\text{depth}(h) + |A_{e(S)}^{cgr(S)}|)$ can be improved. In an dynamic approach where we do not precompute $A_{e(S)}^{cgr(S)}$, we only need to store the needed part of $A_{e(S)}^{cgr(S)}$ for evaluating h , which may be much smaller. More importantly, we do not necessarily have to store stacks of size $O(\text{depth}(h))$, since stacks need not to be changed when projecting subhedges. The sizes of the stacks that we need to store are in $O(\text{depth}(\text{relevant}_S^{\mathcal{L}(A)}(h)))$ where $\text{relevant}_S^{\mathcal{L}(A)}(h)$ is the hedge obtained from h by removing all subhedges that are strongly irrelevant for $\mathcal{L}(A)$ and S .

Example 10.9. *For XPath queries like $/a/b/c$, storing stacks of depth 3 is sufficient for a streaming evaluator, when having a counter available for counting the current level of nodes in subhedges that are parsed but projected otherwise. In our example, we*

have $\text{depth}(\text{relevant}_S^{\mathcal{L}(A)}(h)) \leq 3$ for all $h \in S$. Therefore, our streaming algorithm for this query will run for any hedge in a “stackless” manner.

Streaming algorithms that may use only a finite memory and one counter are called stackless in [Barloy *et al.* 2021]. For XPATH queries with descendant axes, however, our streaming approach will not be stackless, due to the lack of descendant projection (see e.g. [Sebastian & Niehren 2016]). Furthermore, in the few cases with descendant axes, where stackless processing is possible (see [Barloy *et al.* 2021] for a characterization), one needs to use semi-group techniques to figure out how this can be done.

Part IV

Querying

Abstract

We present the first earliest query answering algorithm with complete subhedge and suffix projection for regular monadic queries on hedges. We implemented our algorithm in streaming mode in the *ASTREAM* tool, which permits to answer regular *XPATH* queries on *XML* streams. Our experiments show that complete subhedge and suffix projection make *ASTREAM* competitive in time efficiency with the best existing streaming tools for regular *XPATH* queries, while having lower latency and being more memory efficient in cases where these are not earliest.

Contents

11 Earliest Query Answering for Regular Monadic Queries	261
12 Experiments on Regular <i>XPATH</i> Evaluation on <i>XML</i> Streams	283
13 Conclusion and Future Work	297
Bibliography	301

Introduction

Monadic query answering is the algorithmic task in which we are most interested in this thesis. It is more difficult than pure membership testing. To clarify the relationship, membership testing is sometimes also called Boolean query answering. For monadic query answering it is no longer sufficient to check constraint of the input structure; one must also select all those elements that satisfy some constraints.

Earliest query answering (*EQA*) is the problem to enumerate all certain query answers, each at the earliest event when it becomes certain. While previously studied for stream processing, we propose to explore *EQA* for top-down hedge evaluation, which can operate either in-memory or in streaming mode. Specifically, we investigate *EQA* for regular monadic queries on hedges, defined by deterministic stepwise hedge automata.

We start with an algorithm for earliest monadic query answering with complete subhedge and suffix projection. This algorithm applies to regular monadic queries defined by deterministic stepwise hedge automata. Previously, no earliest query answering algorithm existed for regular monadic queries on hedges with complete subhedge projection. The use of stepwise hedge automata has proven to be crucial for obtaining complete subhedge projection.

To develop our algorithm, we establish a novel link between certain query answering and non-answers, and Σ -certain membership and non-membership of regular languages. This allows us to base our earliest query answering algorithm on the earliest $d\text{SHA}^\downarrow$ s with complete subhedge and suffix projection from Part III.

We implemented our earliest query algorithm in streaming mode in the *ASTREAM* tool, which enables answering regular *XPATH* queries on XML streams. The high efficiency of *ASTREAM* is confirmed by our experimental results for regular *XPATH* queries of *XPATHMARK* benchmark and Lick and Schmitz' benchmark harvested from practical *XSLT* and *XQUERY* programs.

Earliest Query Answering for Regular Monadic Queries

Abstract

We present an earliest query answering (*EQA*) algorithm with complete subhedge and suffix projection that requires time $O(c)$ per non-projected event. The measure c is the concurrency of the query, i.e., the number of alive candidates of the query at the event, independently of which algorithm is chosen. Our *EQA* algorithm can evaluate any regular monadic query either in streaming mode or top-down in-memory. In addition to the time per event, our *EQA* algorithm requires a polynomial preprocessing time in the size of the earliest $d\text{SHA}^\downarrow$ with complete subhedge and suffix projection. This possible exponential preprocessing time can be avoided when admitting time $O(c \cdot m)$ per event, where m is the size of the deterministic stepwise hedge automaton defining the query.

Contents

11.1	Introduction	262
11.2	Certainty for Monadic Queries	263
11.2.1	Certain Answers	264
11.2.2	Certain Non-answers	264
11.2.3	Deciding Certainty	265
11.3	Candidate Automata	267
11.3.1	Construction	267
11.3.2	In-Memory Correctness	269

11.3.3 Streaming Correctness	273
11.4 Earliest Monadic Query Answering	277
11.4.1 Earliest Candidate Automata	277
11.4.2 Adding Subhedge Projection	280

11.1 Introduction

We study the problem of enumerating the answer set of regular monadic node selection queries on hedges. The objective is to produce all certain query answers during a top-down traversal in the earliest manner. The only previously *EQA* algorithm was presented by Gauwin et al. [Gauwin *et al.* 2009b]. It runs in streaming mode on nested words, but cannot support top-down in-memory evaluation. Complete suffix projection is supported under the terms of earliest rejection and selection, but subhedge projection is missing. Furthermore, the worst case time complexity per event is quite high at $O(c n^2)$, where the measure c is the concurrency of the query, i.e., the number of alive candidates for the query at the event, independently of which algorithm chosen, and n is the number of states of an earliest dNwa for the query.

We present an *EQA* algorithm for regular monadic queries on hedges with complete subhedge and suffix projection. It requires time $O(c)$ per non-projected event. Our *EQA* algorithm can run either in streaming mode or top-down in-memory. In addition to the time per event, it requires polynomial preprocessing time in the size of the earliest $d\text{SHA}^\downarrow$ with complete subhedge and suffix projection. This preprocessing is possibly exponential in the size of the input dSHA defining the query, but can be avoided when admitting time $O(c m)$ per event, where m is the number of states of the input dSHA defining the query.

Our approach is to adapt the general ideas of Gauwin et al. from dNwas to dSHAs, while integrating our subhedge projection algorithm. Gauwin's quadratic factor n^2 can be removed when allowing for a polynomial preprocessing time in the size of the earliest $d\text{SHA}^\downarrow$ with complete subhedge and suffix projection. If not permitting it, the quadratic factor n^2 can be reduced to a linear factor m , where m is the overall size of the input dSHA defining the query. Furthermore, Gauwin's cubic preprocessing in time $O(n^3)$ is removed all over. The reduction of the factor n^2 to m per event and the removal for the $O(n^3)$ preprocessing are due to the fact that hedge accessibility is less costly for dSHAs than for dNwas even without any preprocessing.

Rather than compiling to dNwas as done by Gauwin et al. we compile the queries' automata to $d\text{SHA}^\downarrow$. This change simplifies the presentation considerably and is otherwise not relevant. We use the earliest $d\text{SHA}^\downarrow$ s with complete suffix projection from Chapter 8, as a replacement of Gauwin's earliest dNwas for selection and rejection. We combine them with $d\text{SHA}^\downarrow$ s with complete subhedge projection as proposed in Chapter 7. We then show how to run earliest $d\text{SHA}^\downarrow$ s with complete suffix and subhedge projection in order to enumerate all query answers in an earliest manner and with complete suffix and subhedge projection. The earliest $d\text{SHA}^\downarrow$ is created completely at preprocessing time, or partially on need at running time.

It should also be noticed that both *EQA* algorithms use different inputs for defining regular monadic queries. While, the $d\text{SHA}$ that our algorithm inputs can be converted in linear time to a dNwa as required for input by Gauwin's algorithm. The converse conversion, however, may require exponential time, except for single entry dNwas where linear time is enough. Since Nwa determinization is problematic for non single entry Nwas as noticed by [Niehren & Sakho 2021] and discussed in Part II, the only dNwas that could be obtained by determinization in practice are single entry, and can thus be converted to $d\text{SHAs}$ in linear time. Therefore, the difference in the class of the input automata between both algorithm is of low relevance.

We implemented our new *EQA* algorithm in the *ASTREAM* tool and applied it to the regular *XPATH* queries from the *XPATHMARK* collection [Franceschet 2005b] scaling to huge documents, and to the regular *XPATH* queries extracted from practical *XSLT* programs by Lick and Schmitz [Lick & Schmitz 2022] but on smaller documents. It turns out that *ASTREAM* runs efficiently on huge *XML* documents (>100GB) for all queries with low concurrency. Some queries can be answered in streaming mode where the best existing non earliest query answering algorithm failed to be earliest [Debarbieux *et al.* 2015].

11.2 Certainty for Monadic Queries

We move from the problem of language membership, i.e. the answering boolean queries, to the problem to answer monadic queries. Since we want to do it in an earliest manner, we have to lift the notions of certain membership and non-membership to notions of certain answers and non-answers of a monadic query.

11.2.1 Certain Answers

In order to justify early selection, we need the concept of certain answers. Let $\mathbf{S} \subseteq \mathcal{H}_\Sigma$ be a schema, $\mathbf{Q} : \mathbf{S} \rightarrow 2^{\mathbb{N}}$ be a monadic query with schema \mathbf{S} , and $v \in \text{prefs}(\mathcal{N}_\Sigma)$ a nested word prefix.

Definition 11.1. A natural number $\pi \in \mathbb{N}$ is a certain answer of a monadic query $\mathbf{Q} : \mathbf{S} \rightarrow 2^{\mathbb{N}}$ at nested word prefix v if the following holds:

$$\pi \in \text{CA}^{\mathbf{Q}}(v) \Leftrightarrow_{\text{def}} \pi \in \text{pos}(v) \wedge \forall h \in \mathbf{S}. v \in \text{prefs}(\text{nw}(h)) \rightarrow \pi \in \mathbf{Q}(h)$$

A natural number π is thus a certain answer of query \mathbf{Q} , with schema \mathbf{S} , at nested word prefix v , if π is a position of v that answers the query on all completions of v to some hedge $h \in \mathbf{S}$. Certain answers can be safely selected, independently of how the prefix continues to evolve to some hedge in the schema.

Example 11.2. We reconsider $d\text{SHA}$ A from Figure 8.1 that defines the monadic query $\mathbf{Q} = \mathcal{L}(A)$ for the running EQA Example 8.1. It selects in the nodes $1, \dots, n-1$ if h does not start with letter “a” and node n otherwise.

The fact that position 3 is a certain query answer of \mathbf{Q} at the prefix $v = \text{aaa}\langle a$, i.e., $3 \in \text{CA}^{\mathbf{Q}}(v)$ can be seen as follows. The subhedge h'' , that will follow v , is irrelevant since prefix v leads to state 3 in the run, which can no more be changed by any hedge. So when closing the subtree $\langle h \rangle$ where $h = a \cdot h''$, the apply rule $5@3 \rightarrow 9$ of A has to be used. This leads to state 9, which is safe for selection on the upper level, showing that all further continuations $\text{nw}(h')$ of prefix $v \cdot \text{nw}(\langle h \rangle)$ lead to acceptance.

11.2.2 Certain Non-answers

In analogy, we can define that π is certainly a non-answer of \mathbf{Q} with schema \mathbf{S} at nested word prefix v .

Definition 11.3. A natural number π is certainly a non-answer of a monadic query $\mathbf{Q} : \mathbf{S} \rightarrow 2^{\mathbb{N}}$ at nested word prefix v if the following holds:

$$\pi \in \text{CNA}^{\mathbf{Q}}(v) \Leftrightarrow_{\text{def}} \pi \in \text{nod}(v) \wedge \forall h \in \mathbf{S}. v \in \text{prefs}(\text{nw}(h)) \rightarrow \pi \notin \mathbf{Q}(h)$$

Once a node π becomes a certain non-answer then it can be safely rejected.

Example 11.4. The nodes $\pi \in \{1, \dots, n-1\}$, for instance, are certain non-answers on our example query at prefix $v = a^n\langle a$, i.e., $\pi \in \text{CNA}^{\mathbf{Q}}(v)$.

Definition 11.5. We call a node π alive for a monadic query $Q : \mathcal{S} \rightarrow 2^{\mathbb{N}}$ at a nested word prefix v if π is neither a certain answer nor a certain non-answer of Q at v :

$$\pi \in \text{alive}^Q(v) \Leftrightarrow \pi \notin \text{CA}^Q(v) \wedge \pi \notin \text{CNA}^Q(v)$$

The concurrency c of a monadic query Q at v is its number of alive nodes at prefix v .

Example 11.6. For the shorter prefix $v = \text{aaa}\langle$, for instance, all n nodes in $\{1, \dots, n\}$ are alive, so $\pi \in \text{alive}^Q(v)$. This shows that the concurrency of this query at prefix v is equal to n . Therefore, these nodes need to be stored by any top-down query answering algorithm at this event.

11.2.3 Deciding Certainty

We next link certain query answers to certain Σ -membership from Definition 8.3.

Lemma 11.7. For any schema $\mathcal{S} \subseteq \mathcal{H}_\Sigma$, prefix $v \in \text{prefs}(\mathcal{N}_\Sigma)$, language $L \subseteq \mathcal{H}_{\Sigma^x}$, and candidate $\alpha = [x/\pi]$ with $\pi \in \text{pos}(v)$:

$$\Sigma\text{-cert-mem}_{x\text{-str}(\mathcal{S})}^L(v * \alpha) \Leftrightarrow \pi \in \text{CA}^{qrys(L)}(v)$$

Proof. “ \Rightarrow ” For the forward implication, we assume $\Sigma\text{-cert-mem}_{x\text{-str}(\mathcal{S})}^L(v * \alpha)$. We fix $h \in \mathcal{S}$ arbitrarily such that $v \in \text{prefs}(nw(h))$. Then there exists $w \in \hat{\Sigma}^*$ such that $v \cdot w = nw(h)$. Let $h' = h * \alpha$. Note that $h' \in x\text{-str}(\mathcal{S})$ since $\alpha(x) = \pi \in \text{nod}(h)$. Since $h' = nw((v * \alpha) \cdot w)$, the certainty membership $\Sigma\text{-cert-mem}_{x\text{-str}(\mathcal{S})}^L(v * \alpha)$ yields $h' \in L$. Hence, $\alpha \in qrys(L)(h)$. Since $h \in \mathcal{S}$ was arbitrary, this show that $\alpha(x) = \pi \in \text{CA}^{qrys(L)}(v)$

“ \Leftarrow ” For the backward implication, we assume $\pi \in \text{CA}^{qrys(L)}(v)$. We fix $h' \in x\text{-str}(\mathcal{S})$ and $w \in \hat{\Sigma}^*$ arbitrarily such that $(v * \alpha) \cdot w \in nw(h')$. Let $h = nw(v \cdot w)$. Since $h' \in x\text{-str}(\mathcal{S})$ it follows that $h \in \mathcal{S}$. Furthermore $v \in \text{prefs}(nw(h))$, and since π is a certain answer of $qrys(L)$ on v , it follows that $\alpha(x) = \pi \in qrys(L)(h)$. Thus $h' = h * \alpha \in L$.

□

Proposition 11.8. Let $A = (\Sigma^x, \mathcal{Q}, \Delta, I, F)$ be a $d\text{SHA}$ such that $\Delta|_\Sigma$ is complete, and $\mathcal{S} \subseteq \mathcal{H}_\Sigma$ a schema. Let $D_0 \in I^{\Sigma\text{-cert-mem}(\mathcal{S})}$, $v \in \text{prefs}(\mathcal{N}_\Sigma)$, and $\pi \in \text{pos}(v)$. Let:

$$((q, Q), \sigma) = \llbracket v * [x/\pi] \rrbracket_{\text{str}}^{\Delta^{\Sigma\text{-cert-mem}}} (D_0, \epsilon)$$

It then holds that:

$$\pi \in \text{CA}^{qrys(\mathcal{L}(A))}(v) \Leftrightarrow q \in Q$$

Proof. Let $\alpha = [x/\pi]$ and $((q, Q), \sigma) = \llbracket v * \alpha \rrbracket_{str}^{\Delta^{\Sigma\text{-cert-mem}}}(D_0, \epsilon)$.

" \Rightarrow " For the forward direction, we assume $\alpha(x) \in CA^{qry_s(\mathcal{L}(A))}(v)$. Lemma 11.7 yields $\Sigma\text{-cert-mem}_{x\text{-str}(\mathbf{S})}^{\mathcal{L}(A)}(v * \alpha)$. Since $\Delta_{|\Sigma}$ is complete, Proposition 8.9 shows $q \in Q$.

" \Leftarrow " For the backward direction, we assume that $q \in Q$. Since $\Delta_{|\Sigma}$ is complete, Proposition 8.9 shows $\Sigma\text{-cert-mem}_{x\text{-str}(\mathbf{S})}^{\mathcal{L}(A)}(v * \alpha)$. Lemma 11.7 thus proves that $\alpha(x) \in CA^{qry_s(\mathcal{L}(A))}(v)$.

□

We can now link certain query answers and non-answers to the earliest automaton.

Proposition 11.9 (Characterizing certain query answers and non-answers). *Let $A = (\Sigma^x, \mathcal{Q}, \Delta, I, F)$ be a complete dSHA and $S \subseteq \mathcal{H}_\Sigma$ a schema. Let $D_0 \in I_{e(S)}$. For any prefix $v \in \text{pref}(\mathcal{N}_\Sigma)$:*

- $CA^{qry_s(\mathcal{L}(A))}(v) = \{\pi \in \text{pos}(v) \mid \exists \sigma \in (\mathcal{Q}_e)^*. (\mathbf{sel}, \sigma) = \llbracket v * [x/\pi] \rrbracket_{str}^{\Delta_e}(D_0, \epsilon)\}$
- $CNA^{qry_s(\mathcal{L}(A))}(v) = \{\pi \in \text{pos}(v) \mid \llbracket v * [x/\pi] \rrbracket_{str}^{\Delta_e}(D_0, \epsilon) \text{ is undefined}\}$

Proof. Let $(q_0, Q_0, R_0) = D_0$ and $\pi \in \text{pos}(v)$.

- Let $((q, Q), \sigma) = \llbracket v * [x/\pi] \rrbracket_{str}^{\Delta^{\Sigma\text{-cert-mem}}}((q_0, Q_0), \epsilon)$. By Proposition 11.9, we have $\pi \in CNA^{qry_s(\mathcal{L}(A))}(v)$ if and only if $q \in Q$. This is equivalent to that there exists $\sigma \in (\mathcal{Q}_e)^*$ such that $(\mathbf{sel}, \sigma) = \llbracket v * [x/\pi] \rrbracket_{str}^{\Delta_e}(D_0, \epsilon)$.
- Let $((q, R), \sigma) = \llbracket v * [x/\pi] \rrbracket_{str}^{\Delta^{\Sigma\text{-cert-mem}}}((q_0, R_0), \epsilon)$. The complement automaton $\bar{A} = (\Sigma^x, \mathcal{Q}, \Delta, I, \bar{F})$ has the same transition rules than A . By Proposition 11.9, we have $\pi \in CNA^{qry_s(\mathcal{L}(A))}(v)$ if and only if $q \in R$. Since $q \in R$, it follows that $\llbracket v * [x/\pi] \rrbracket_{str}^{\Delta_e}(D_0, \epsilon)$ is undefined.

□

This characterization can be used for deciding whether a node π is a certain query answer or non-answer of $qry_s(\mathcal{L}(A))$ at prefix v . The proposition shows that it is sufficient to run the earliest automaton $A_{e(S)}$ on $v * [x/\pi]$ to compute $\llbracket v * [x/\pi] \rrbracket_{str}^{\Delta_e}(D_0, \epsilon)$. If the result is undefined then $\pi \in CNA^{qry_s(\mathcal{L}(A))}(v)$. Otherwise, if the result exists, $\pi \in CNA^{qry_s(\mathcal{L}(A))}(v)$ if and only if the result contains state **sel**.

11.3 Candidate Automata

We now move to the problem of how to answer monadic queries defined by $d\text{SHA}^\downarrow$ in a top-down manner.

Any monadic query answering algorithm receives as inputs an automaton with signature $\Sigma \cup \{x\}$ defining the query and a hedge $h \in \mathcal{H}_\Sigma$. A naive query evaluation algorithm guesses all possible nodes π of where to insert x into h , and runs A on $h*[x/\pi]$ to see whether the guess was successful. In the worst case, however, the algorithm has to buffer all candidates $[x/\pi]$ where $p \in \text{pos}(h)$. In a less naive manner, the algorithm inserts x only at those nodes for which the current state has an outgoing x -transition rule.

One idea to obtain such a naive query answering algorithm is to compile the input $d\text{SHA}^\downarrow$ to an infinitary $d\text{SHA}^\downarrow$ – also called $d\text{SHA}_\infty^\downarrow$ – which is like a $d\text{SHA}^\downarrow$ except that its alphabet, state space, and rule set may be infinite. The top-down evaluator of this $d\text{SHA}_\infty^\downarrow$ on the input hedge then performs the guessing and testing of the naive algorithm, either in-memory or in streaming mode.

11.3.1 Construction

Let x be the selection variable of the input $d\text{SHA}^\downarrow$ defining the monadic query. Our algorithm will generate a set of answer candidates $[x/\pi]$ binding the selection variable x to positions π of the input hedge. Since the input hedge is not fixed at before hand, we have to consider the infinite set of potential candidates for all possible input hedges:

$$\text{Cands} = \{\alpha \mid \alpha : \{x\} \hookrightarrow \mathbb{N}\}$$

Let $A = (\Sigma^x, \mathcal{Q}, \Delta, I, F)$ be the input $d\text{SHA}^\downarrow$ defining the monadic query. We compile A to the candidate automaton A^{cnd} which is the following $d\text{SHA}_\infty^\downarrow$:

$$\text{cnd}(A) = (\Sigma \cup \mathbb{N}, \mathcal{Q}^{cnd}, \Delta^{cnd}, I^{cnd}, F^{cnd})$$

It has the infinite alphabet $\Sigma \cup \mathbb{N}$, which does not contain the selection variable x any more, and the following set of states:

$$\begin{aligned} \mathcal{Q}^{cnd} &= 2^{\mathcal{Q} \times \text{Cands} \times 2^{\{x\}}} \\ I^{cnd} &= \{D_0\} \quad \text{where} \quad D_0 = \{(q, [], \emptyset) \mid q \in I\} \\ F^{cnd} &= \{D \in \mathcal{Q}^{cnd} \mid (q, [x, \pi], \emptyset) \in D, q \in F, \pi \in \mathbb{N}\} \end{aligned}$$

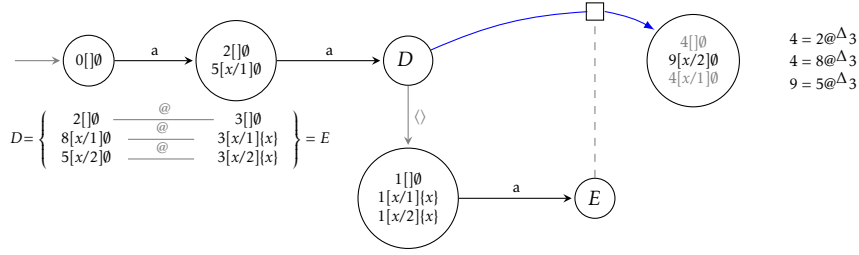


Figure 11.1: The run of the candidate automaton $cnd(A)$ for the $dSHA$ A in Figure 8.1 on the hedge $a \cdot a \cdot \langle a \rangle$.

The candidate automaton reads an input hedge $h \in \mathcal{H}_\Sigma$, guesses all possible node $\pi \in nod(h)$, and runs A on all hedges $h*[x/\pi]$ simultaneously. If the automaton goes to some state $D \in \mathcal{Q}^{cnd}$ and $(q, \alpha, V) \in D$, then the candidate α was created by A , when the node $\alpha(x)$ was visited before the current node by depth-first search. The candidate α was then evaluated to state q . The set $V \subseteq dom(\alpha)$ contains x if x was bound to some node $\alpha(x)$ in the context, i.e., outside the current subtree.

The set of transition rules of the candidate automaton Δ^{cnd} contains the following transition rules for all $D, E \in \mathcal{Q}^{cnd}$, $a \in \Sigma$, $V \subseteq \{x\}$, and $\pi \in \mathbb{N}$:

$$\begin{aligned}
 D &\xrightarrow{a} \{(a^\Delta(q), \alpha, V) \mid (q, \alpha, V) \in D\} \\
 D &\xrightarrow{\pi} \{(x^\Delta(q), [x/\pi], \emptyset) \mid (q, [], \emptyset) \in D\} \cup D \\
 D &\xrightarrow{\langle} \{(p, \alpha, dom(\alpha)) \mid q \xrightarrow{\langle} p \in \Delta, (q, \alpha, V) \in D\} \\
 D @ E &\rightarrow \{(q @^\Delta p, \alpha', V) \mid (q, \alpha, V) \in D, (p, \alpha', dom(\alpha)) \in E, \alpha \in \{[], \alpha'\}\}
 \end{aligned}$$

When reading a position $\pi \in \mathbb{N}$ in a state D that contains a triple with the empty candidate $(q, [], \emptyset)$, a new candidate $[x/\pi]$ is created, and the triple $(x^\Delta(q), [x/\pi], \emptyset)$ is added to D .

When starting a subtree, the current state D of A^{cnd} is changed as follows: for any triple $(q, \alpha, V) \in D$, the next state of $cnd(A)$ contains the triple $(p, \alpha, dom(\alpha))$ such that $q \xrightarrow{\langle} p \in \Delta$. When ending a subtree, the state D of the parent hedge is applied to the current state E as follows: any triple $(q, \alpha, V) \in D$ must be matched with some triple $(p, \alpha', dom(\alpha)) \in E$, so that A^{cnd} can continue in $q @^\Delta p$. Matching here means that either $\alpha = \alpha'$ or, $\alpha' = [x/\pi]$ and $\alpha = []$. This is expressed by the condition $\alpha \in \{[], \alpha'\}$. Note that if $\alpha = []$ matches $\alpha' = [x/\pi]$ then $dom(\alpha) = \emptyset$ so that π was not bound in the context. This is where the knowledge of the context is needed.

Example 11.10. Let A be the $dSHA$ in Figure 8.1. The run of the candidate automaton $cnd(A)$ on the hedge $a \cdot a \cdot \langle a \rangle$ is given in Figure 11.1. There, tuples in the states are

written without commas and parentheses, for instance $(2, [], \emptyset)$ as $2[]\emptyset$ and $(1, [x/1], \{x\})$ as $1[x/1]\{x\}$. The run of $\text{cnd}(A)$ first consumes $a \cdot a$ and goes into the state:

$$D = \{2[]\emptyset, 8[x/1]\emptyset, 5[x/2]\emptyset\}$$

It contains the candidates $[x/1]$ and $[x/2]$ for the two leading a positions, plus the empty candidate $[]$. When opening the subtree $\langle a \rangle$, the run goes into the set $\{1[]\emptyset, 1[x/1]\{x\}, 1[x/2]\{x\}\}$. The state of each of the candidates is set to the unique tree initial state $1 \in \langle \rangle^\Delta$. Furthermore, the set memoizes that the candidates $[x/1]$ and $[x/2]$ were bound in the context. It then consumes the letter a and reaches the state:

$$E = \{3[]\emptyset, 3[x/1]\{x\}, 3[x/2]\{x\}\}$$

When closing the subtree, the tuples of the parent's states D in state q are matched with the tuples of E in state p , as illustrated in the figure, so that one can apply rule $q@p \rightarrow q@^\Delta p$ of A . The tuple in state 5 of D , for instance, matches the tuple in state 3 of E , so $\text{cnd}(A)$ continues the candidate $[x/2]$ in state $9 = 5@^\Delta 3$. Since $9 \in F^A$, position 2 is selected, i.e. $2 \in \text{qry}_S(\mathcal{L}(A))(a \cdot a \cdot \langle a \rangle)$.

11.3.2 In-Memory Correctness

The correctness of the candidate automaton is stated by the following lemma.

Lemma 11.11. *Let $I^{\text{cnd}} = \{D_0\}$. Then:*

$$\llbracket \text{ann-nod}(h) \rrbracket^{\Delta^{\text{cnd}}}(D_0) = \{(q, \alpha, \emptyset) \mid q \in \llbracket h * \alpha \rrbracket^\Delta(I), \alpha : \{x\} \hookrightarrow \text{nod}(h)\}$$

Proof. If $I = \emptyset$ then $D_0 = \emptyset$ and the lemma is trivial. Otherwise there exists a unique state $q_0 \in \mathcal{Q}$ such that $I = \{q_0\}$ and $D_0 = \{(q_0, [], \emptyset)\}$.

We call a state $D \in \mathcal{Q}^{\text{cnd}}$ admissible if any $(q, \alpha, V) \in D$ satisfies $V \subseteq \text{dom}(\alpha)$. Note that all states accessible from D_0 via Δ^{cnd} are admissible.

For any $n \in \mathbb{N}$ and variable assignment $\alpha : \{x\} \hookrightarrow \{n, \dots, n + |h| - 1\}$ we define a hedge $h *^n \alpha \in \mathcal{H}_{\Sigma^x}$ by substituting in $\text{ann-nod}_n(h)$ the position $\alpha(x)$ if defined by x and removing all other positions:

$$h *^n \alpha = \text{ann-nod}_n(h)[\alpha(x)/x][\pi/\epsilon \mid \pi \neq \alpha(x)]$$

Clearly, $h * \alpha = h *^1 \alpha$. We call a state $D \in \mathcal{Q}^{\text{cnd}}$ applicable to h with offset n if for all $(_, [x/\pi], V) \in D$ it holds that $\pi < n$.

Claim 11.12. *We will prove for all $h \in \mathcal{H}_\Sigma$, $n \in \mathbb{N}$, and admissible states $D \in \mathcal{Q}^{cnd}$ that are applicable with offset n :*

$$\begin{aligned} \llbracket ann-nod_n(h) \rrbracket^{\Delta^{cnd}}(D) &= \{ (\llbracket h *^n [x/\pi] \rrbracket^\Delta(q), [x/\pi], \emptyset) \mid \pi \in \{n, \dots, n + |h| - 1\}, (q, [], \emptyset) \in D \} \\ &\cup \{ (\llbracket h \rrbracket^\Delta(q), \alpha, V) \mid (q, \alpha, V) \in D \} \end{aligned}$$

With $D = D_0$ and $n = 0$ this claim yields exactly the lemma. The proof of the claim is by induction on the structure of h , using the facts that admissibility and applicability are maintained while the offsets are adapted.

So let $h \in \mathcal{H}_\Sigma$, $n \in \mathbb{N}$, and $D \in \mathcal{Q}^{cnd}$ an admissible state that is applicable with offset n . We distinguish cases according to all possible forms of h :

Case $h = a$. We have $ann-nod_n(h) = a \cdot n$ and $h *^n [x/n] = a \cdot x$.

$$\begin{aligned} \llbracket ann-nod_n(h) \rrbracket^{\Delta^{cnd}}(D) &= \llbracket a \cdot n \rrbracket^{\Delta^{cnd}}(D) \\ &= \{ (x^\Delta(a^\Delta(q)), [x/n], \emptyset) \mid (q, [], \emptyset) \in D \} \\ &\cup \{ (a^\Delta(q), \alpha, V) \mid (q, \alpha, V) \in D \} \\ &= \{ (\llbracket a \cdot n \rrbracket^\Delta(q), [x/n], \emptyset) \mid (q, [], \emptyset) \in D \} \\ &\cup \{ (\llbracket a \rrbracket^\Delta(q), \alpha, V) \mid (q, \alpha, V) \in D \} \\ &= \{ (\llbracket h *^n [x/n] \rrbracket^\Delta(q), [x/n], \emptyset) \mid (q, [], \emptyset) \in D \} \\ &\cup \{ (\llbracket h \rrbracket^\Delta(q), \alpha, V) \mid (q, \alpha, V) \in D \} \end{aligned}$$

Case $h = \epsilon$. We have $ann-nod_n(h) = \epsilon$ and $h *^n [] = \epsilon$.

$$\begin{aligned} \llbracket ann-nod_n(h) \rrbracket^{\Delta^{cnd}}(D) &= \llbracket \epsilon \rrbracket^{\Delta^{cnd}}(D) \\ &= D \\ &= \{ (\llbracket h *^n [x/\pi] \rrbracket^\Delta(q), [x/\pi], \emptyset) \mid \pi \in \emptyset, (q, [], \emptyset) \in D \} \\ &\cup \{ (\llbracket h \rrbracket^\Delta(q), \alpha, V) \mid (q, \alpha, V) \in D \} \end{aligned}$$

Case $h = h_1 \cdot h_2$. Let $n_1 = n + |h_1|$. Then we have $ann-nod_n(h) = ann-nod_n(h_1) \cdot ann-nod_{n_1}(h_2)$. Hence:

$$\llbracket ann-nod_n(h) \rrbracket^{\Delta^{cnd}}(D) = \llbracket ann-nod_{n_1}(h_2) \rrbracket^{\Delta^{cnd}}(\llbracket ann-nod_n(h_1) \rrbracket^{\Delta^{cnd}}(D))$$

Let $D_1 = \llbracket ann-nod_n(h_1) \rrbracket^{\Delta^{cnd}}(D)$, $D_2 = \llbracket ann-nod_{n_1}(h_2) \rrbracket^{\Delta^{cnd}}(D_1)$ and $n_2 = n_1 + |h_2|$. The

induction hypothesis applied to h_1 , D , and n yields:

$$\begin{aligned} D_1 &= \{(\llbracket h_1 *^n [x/\pi] \rrbracket^\Delta(q), [x/\pi], \emptyset) \mid \pi \in \{n, \dots, n_1 - 1\}, (q, [], \emptyset) \in D\} \\ &\cup \{(\llbracket h_1 \rrbracket^\Delta(q), \alpha, V) \mid (q, \alpha, V) \in D\} \end{aligned}$$

Note that D_1 is applicable with offset n_1 . The induction hypothesis applied to h_2 , D_1 , and n_1 yields:

$$\begin{aligned} D_2 &= \{(\llbracket h_2 *^{n_1} [x/\pi] \rrbracket^\Delta(q_1), [x/\pi], \emptyset) \mid \pi \in \{n_1, \dots, n_2 - 1\}, (q_1, [], \emptyset) \in D_1\} \\ &\cup \{(\llbracket h_2 \rrbracket^\Delta(q_1), \alpha, V) \mid (q_1, \alpha, V) \in D_1\} \end{aligned}$$

Hence,

$$\begin{aligned} &\llbracket ann-nod_n(h) \rrbracket^{\Delta^{cnd}}(D) \\ &= \{(\llbracket h_2 *^{n_1} [x/\pi] \rrbracket^\Delta(\llbracket h_1 \rrbracket^\Delta(q)), [x/\pi], \emptyset) \mid \pi \in \{n_1, \dots, n_2 - 1\}, (q, [], \emptyset) \in D\} \\ &\cup \{(\llbracket h_2 \rrbracket^\Delta(\llbracket h_1 *^n [x/\pi] \rrbracket^\Delta(q)), [x/\pi], \emptyset) \mid \pi \in \{n, \dots, n_1 - 1\}, (q, [], \emptyset) \in D\} \\ &\cup \{(\llbracket h_2 \rrbracket^\Delta(\llbracket h_1 \rrbracket^\Delta(q)), \alpha, V) \mid (q, \alpha, V) \in D\} \end{aligned}$$

Note that $\llbracket h_2 *^{n_1} [x/\pi] \rrbracket^\Delta(\llbracket h_1 \rrbracket^\Delta(q)) = (\llbracket h_1 \cdot h_2 *^n [x/\pi] \rrbracket^\Delta(q))$ for any $\pi \in \{n, \dots, n_2 - 1\}$, and similarly, $\llbracket h_2 \rrbracket^\Delta(\llbracket h_1 *^n [x/\pi] \rrbracket^\Delta(q)) = (\llbracket h_1 \cdot h_2 *^n [x/\pi] \rrbracket^\Delta(q))$ for any $\pi \in \{n, \dots, n_1 - 1\}$.

Thus:

$$\begin{aligned} &\llbracket ann-nod_n(h) \rrbracket^{\Delta^{cnd}}(D) \\ &= \{(\llbracket h_1 \cdot h_2 *^n [x/\pi] \rrbracket^\Delta(q), [x/\pi], \emptyset) \mid \pi \in \{n, \dots, n_2 - 1\}, (q, [], \emptyset) \in D\} \\ &\cup \{(\llbracket h_1 \cdot h_2 \rrbracket^\Delta(q), \alpha, V) \mid (q, \alpha, V) \in D\} \\ &= \{(\llbracket h *^n [x/\pi] \rrbracket^\Delta(q), [x/\pi], \emptyset) \mid \pi \in \{n, \dots, n + |h| - 1\}, (q, [], \emptyset) \in D\} \\ &\cup \{(\llbracket h \rrbracket^\Delta(q), \alpha, V) \mid (q, \alpha, V) \in D\} \end{aligned}$$

Case $h = \langle h_1 \rangle$. Thus $ann-nod_n(h) = \langle n \cdot ann-nod_{n+1}(h_1) \rangle$. Let:

$$D' = \{(\langle \rangle^\Delta(q), \alpha, dom(\alpha)) \mid (q, \alpha, V) \in D\}$$

Thus $D \xrightarrow{\langle \rangle} D'$ wrt Δ^{cnd} and:

$$\begin{aligned} \llbracket ann-nod_n(h) \rrbracket^{\Delta^{cnd}}(D) &= \llbracket \langle n \cdot ann-nod_{n+1}(h_1) \rangle \rrbracket^{\Delta^{cnd}}(D) \\ &= D @^{\Delta^{cnd}} \llbracket n \cdot ann-nod_{n+1}(h_1) \rrbracket^{\Delta^{cnd}}(D') \\ &= D @^{\Delta^{cnd}} \llbracket ann-nod_{n+1}(h_1) \rrbracket^{\Delta^{cnd}}(\llbracket n \rrbracket^{\Delta^{cnd}}(D')) \end{aligned}$$

Let $D_1 = D' \cup \{(x^\Delta(\langle \rangle^\Delta(q)), [x/n], \emptyset) \mid (q, [], \emptyset) \in D\}$ and $D_2 = \llbracket ann-nod_{n+1}(h_1) \rrbracket^{\Delta^{cnd}}(D_1)$.

Then:

$$\llbracket ann-nod(h)^n \rrbracket^{\Delta^{cnd}}(D) = D @^{\Delta^{cnd}} \llbracket ann-nod_{n+1}(h_1) \rrbracket^{\Delta^{cnd}}(D_1) = D @^{\Delta^{cnd}} D_2$$

Let and $n_1 = n + |h_1|$. Note that D_1 is applicable with offset n_1 . The induction hypothesis applied to h_1 , D_1 and $n + 1$ shows:

$$\begin{aligned} D_2 &= \{(\llbracket h_1 \rrbracket^{\Delta}(q_1), \alpha, V) \mid (q_1, \alpha, V) \in D_1\} \\ &\cup \{(\llbracket h_1 *^{n+1} [x/\pi] \rrbracket^{\Delta}(q_1), [x/\pi], \emptyset) \mid \pi \in \{n+1, \dots, n_1\}, (q_1, [], \emptyset) \in D_1\} \end{aligned}$$

Thereby and by the definition of D_1 we obtain:

$$\begin{aligned} D_2 &= \{(\llbracket h_1 \rrbracket^{\Delta}(\langle \rangle^{\Delta}(q)), \alpha, dom(\alpha)) \mid (q, \alpha, V) \in D\} \\ &\cup \{(\llbracket h_1 \rrbracket^{\Delta}(x^{\Delta}(\langle \rangle^{\Delta}(q))), [x/n], \emptyset) \mid (q, [], \emptyset) \in D\} \\ &\cup \{(\llbracket h_1 *^{n+1} [x/\pi] \rrbracket^{\Delta}(\langle \rangle^{\Delta}(q)), [x/\pi], \emptyset) \mid \pi \in \{n+1, \dots, n_1\}, (q, [], \emptyset) \in D\} \end{aligned}$$

With this equation, we can develop the right hand side of the equation $\llbracket ann-nod_n(h) \rrbracket^{\Delta^{cnd}}(D) = D @^{\Delta^{cnd}} D_2$ by applying the definition of the closing rule of Δ^{cnd} :

$$\begin{aligned} D @^{\Delta^{cnd}} D_2 &= \{(q @ \llbracket h_1 \rrbracket^{\Delta}(\langle \rangle^{\Delta}(q)), \alpha, V) \mid (q, \alpha, V) \in D\} \\ &\cup \{(q @ \llbracket n \cdot h_1 \rrbracket^{\Delta}(\langle \rangle^{\Delta}(q)), [x/n], \emptyset) \mid (q, [], \emptyset) \in D\} \\ &\cup \{(q @ \llbracket h_1 *^{n+1} [x/\pi] \rrbracket^{\Delta}(\langle \rangle^{\Delta}(q)), [x/\pi], \emptyset) \mid \pi \in \{n+1, \dots, n_1\}, (q, [], \emptyset) \in D\} \end{aligned}$$

Note that the admissibility of D ensures above that $[x/n]$ cannot match with any $(q, [], \{x\}) \in D$. We next apply the definition of $\llbracket \langle h_1 \rangle \rrbracket^{\Delta}$ to show:

$$\begin{aligned} D @^{\Delta^{cnd}} D_2 &= \{(\llbracket \langle h_1 \rangle \rrbracket^{\Delta}(q), \alpha, V) \mid (q, \alpha, V) \in D\} \\ &\cup \{(\llbracket \langle n \cdot h_1 \rangle \rrbracket^{\Delta}(q), [x/n], \emptyset) \mid (q, [], \emptyset) \in D\} \\ &\cup \{(\llbracket \langle h_1 *^{n+1} [x/\pi] \rangle \rrbracket^{\Delta}(q), [x/\pi], \emptyset) \mid \pi \in \{n+1, \dots, n_1\}, (q, [], \emptyset) \in D\} \end{aligned}$$

Since $h = \langle h_1 \rangle$ we get:

$$\begin{aligned} D @^{\Delta^{cnd}} D_2 &= \{(\llbracket h \rrbracket^{\Delta}(q), \alpha, V) \mid (q, \alpha, V) \in D\} \\ &\cup \{(\llbracket h *^n [x/n] \rrbracket^{\Delta}(q), [x/n], \emptyset) \mid (q, [], \emptyset) \in D\} \\ &\cup \{(\llbracket h *^n [x/\pi] \rrbracket^{\Delta}(q), [x/\pi], \emptyset) \mid \pi \in \{n+1, \dots, n_1\}, (q, [], \emptyset) \in D\} \\ &= \{(\llbracket h \rrbracket^{\Delta}(q), \alpha, V) \mid (q, \alpha, V) \in D\} \\ &\cup \{(\llbracket h *^n [x/\pi] \rrbracket^{\Delta}(q), [x/\pi], \emptyset) \mid \pi \in \{n, \dots, |h| - 1\}, (q, [], \emptyset) \in D\} \end{aligned}$$

Since $\llbracket ann-nod_n(h) \rrbracket^{\Delta^{cnd}}(D) = D @^{\Delta^{cnd}} D_2$ the inductive statement follows.

We finally notice that the restriction of states D to be applicable with offset n is not strictly necessary. The claim remains true when omitting it. \square

The candidate automaton thus computes the answer set of the query as follows:

Proposition 11.13 (Correctness of the candidate automaton). *If $D_0 \in I^{cnd}$ then:*

$$qry_S(\mathcal{L}(A))(h) = \{\pi \mid (q, [x/\pi], \emptyset) \in \llbracket ann-nod(h) \rrbracket^{\Delta^{cnd}}(D_0), q \in F\}$$

Proof. Let $D_0 \in I^{cnd}$. If $I = \emptyset$ then $D_0 = \emptyset$ and the lemma is trivial. Otherwise, $I = \{q_0\}$ for some $q_0 \in \mathcal{Q}$ and $D_0 = \{(q_0, [], \emptyset)\}$. Furthermore, let $\alpha = [x/\pi]$ for some $\pi \in \mathbb{N}$. By Lemma 11.11, we have that $q = \llbracket h * \alpha \rrbracket^\Delta(q_0)$ iff $(q, \alpha, \emptyset) \in \llbracket ann-nod(h) \rrbracket^{\Delta^{cnd}}(D_0)$. Hence:

$$\begin{aligned} \pi \in qry_S(\mathcal{L}(A))(h) & \text{ iff } \pi = \alpha(x) \text{ and } h * \alpha \in \mathcal{L}(A) \\ & \text{ iff } \pi = \alpha(x) \text{ and } \llbracket h * \alpha \rrbracket^\Delta(q_0) \in F \\ & \text{ iff } \pi = \alpha(x) \text{ and } \exists q \in F. (q, \alpha, \emptyset) \in \llbracket ann-nod(h) \rrbracket^{\Delta^{cnd}}(D_0). \end{aligned}$$

\square

Corollary 11.14. $qry_S(\mathcal{L}(A))(h) \neq \emptyset \Leftrightarrow ann-nod(h) \in \mathcal{L}(A^{cnd})$

Proof. Let $I^{cnd} = \{D_0\}$. By Lemma 11.13, $qry_S(\mathcal{L}(A))(h) \neq \emptyset$ if and only if there exists $q \in F$ and $\alpha : \{x\} \rightarrow \mathbb{N}$ such that $(q, \alpha, \emptyset) \in \llbracket ann-nod(h) \rrbracket^{\Delta^{cnd}}(D_0)$. This is equivalent to $\llbracket ann-nod(h) \rrbracket^{\Delta^{cnd}}(D_0) \in F^{cnd}$ and thus to $ann-nod(h) \in L(A^{cnd})$. \square

11.3.3 Streaming Correctness

Lemma 11.15. *Let $A = (\mathcal{Q}, \Sigma, \Delta, I, F)$ be a $dSHA^\downarrow$ with signature Σ^x such that $\Delta|_\Sigma$ is complete. Then there exists q_0 such that $I = \{q_0\}$. Then $I^{cnd} = \{M_0\}$ where $M_0 = \{(q_0, [], \emptyset)\}$. Let $v \in \text{prefs}(\mathcal{N}_\Sigma)$ be a nested word prefix and $M \in \mathcal{Q}^{cnd}$ such that:*

$$\exists \sigma'. (M, \sigma') = \llbracket ann-nod(v) \rrbracket_{str}^{\Delta^{cnd}}(M_0, \epsilon)$$

Then for any variable assignment $\alpha : \{x\} \rightarrow \text{pos}(v)$, and $q \in \mathcal{Q}$:

$$\exists \sigma. (q, \sigma) = \llbracket v * \alpha \rrbracket_{str}^\Delta(q_0, \epsilon) \Leftrightarrow \exists V. (q, \alpha, V) \in M$$

Proof. By induction on the structure of v . Let $v \in \text{prefs}(\mathcal{N}_\Sigma)$ and $M \in \mathcal{Q}^{cnd}$ such that:

$$\exists \sigma. (M, \sigma) = \llbracket ann-nod(v) \rrbracket_{str}^{\Delta^{cnd}}(M_0, \epsilon)$$

Consider a variable assignment $\alpha : \{x\} \rightarrow \text{pos}(v)$, and $q \in \mathcal{Q}$. In the base case, v is the empty word:

Case $v = \epsilon$. Then, $\text{ann-nod}(v) = \epsilon$ and $\llbracket \text{ann-nod}(v) \rrbracket_{\text{str}}^{\text{cnd}}(M_0, \epsilon) = (M_0, \epsilon)$. Hence, $M = M_0$. On the other hand, $\llbracket v * \alpha \rrbracket_{\text{str}}^\Delta(q_0, \epsilon)$ is defined only if $\alpha = []$, and then it is equal to (q_0, ϵ) . So

$$\begin{aligned} \exists V. (q, \alpha, V) \in M &\Leftrightarrow q = q_0 \wedge \alpha = [] \\ &\Leftrightarrow (q, \epsilon) = \llbracket v * \alpha \rrbracket_{\text{str}}^\Delta(q_0, \epsilon) \\ &\Leftrightarrow \exists \sigma. (q, \sigma) = \llbracket v * \alpha \rrbracket_{\text{str}}^\Delta(q_0, \epsilon) \end{aligned}$$

For the inductive step, there are three cases depending on the type of the last letter of v . Then, there are $a \in \Sigma$ or $v', v'' \in \text{prefs}(\mathcal{N}_\Sigma)$ such that:

Case $v = v' \cdot a$. So there exists $\pi \in \mathbb{N}$ such that $\text{ann-nod}(v) = \text{ann-nod}(v') \cdot a \cdot \pi$. Let $(M', _) = \llbracket \text{ann-nod}(v') \rrbracket_{\text{str}}^{\text{cnd}}(M_0, \epsilon)$. We consider two subcases depending on whether α binds x to the last position π or not:

Subcase $\alpha = [x/\pi]$. We have $(q, \alpha, _) \in M$ iff exists q' and q'' such that $(q', [], \emptyset) \in M'$ and $q' \xrightarrow{a} q'' \xrightarrow{x} q \in \Delta$. By induction hypothesis, $(q', [], \emptyset) \in M'$ is equivalent to $(q', _) = \llbracket v' * [] \rrbracket_{\text{str}}^\Delta(q_0, \epsilon)$. Hence, $(q, \alpha, _) \in M$ iff exists q' and q'' such that $(q', _) = \llbracket v' * [] \rrbracket_{\text{str}}^\Delta(q_0, \epsilon)$ and $q' \xrightarrow{a} q'' \xrightarrow{x} q \in \Delta$. And this is equivalent to $(q, _) = \llbracket v * \alpha \rrbracket_{\text{str}}^\Delta(q_0, \epsilon)$.

Subcase $\alpha \neq [x/\pi]$. We have $(q, \alpha, _) \in M$ iff exists q' such that $(q', \alpha, _) \in M'$ and $q' \xrightarrow{a} q \in \Delta$. By induction hypothesis, $(q', \alpha, _) \in M'$ is equivalent to $(q', _) = \llbracket v' * \alpha \rrbracket_{\text{str}}^\Delta(q_0, \epsilon)$. Hence, $(q, \alpha, _) \in M$ iff exists q' such that $(q', _) = \llbracket v' * \alpha \rrbracket_{\text{str}}^\Delta(q_0, \epsilon)$ and $q' \xrightarrow{a} q \in \Delta$. And this is equivalent to $(q, _) = \llbracket v * \alpha \rrbracket_{\text{str}}^\Delta(q_0, \epsilon)$.

Case $v = v' \cdot \langle \cdot \rangle$. Hence, $\text{ann-nod}(v) = \text{ann-nod}(v') \cdot \langle \cdot \rangle$. Let $(M', _) = \llbracket \text{ann-nod}(v') \rrbracket_{\text{str}}^{\text{cnd}}(M_0, \epsilon)$. If α binds some position to x then it must be a position of $\text{ann-nod}(v')$. So, $(q, \alpha, _) \in M$ iff $q = \langle \rangle^\Delta$ and there exist $q' \in \mathcal{Q}$ and $Q' \subseteq \mathcal{Q}$ such that $(q', \alpha, _, Q') \in M'$ and $\text{sdown}^{\Delta_\Sigma}(q', Q') = Q$. By induction hypothesis, $(q', \alpha, _, Q') \in M'$ is equivalent to $((q', Q'), _) = \llbracket v' * \alpha \rrbracket_{\text{str}}^\Delta(q_0, \epsilon)$. Hence, $(q, \alpha, _, Q) \in M$ iff $q = \langle \rangle^\Delta$ and there exist $q' \in \mathcal{Q}$ and $Q' \subseteq \mathcal{Q}$ such that $((q', Q'), _) = \llbracket v' * \alpha \rrbracket_{\text{str}}^\Delta(q_0, \epsilon)$ and $\text{sdown}^{\Delta_\Sigma}(q', Q') = Q$. The latter is equivalent to $((q, Q), _) = \llbracket v * \alpha \rrbracket_{\text{str}}^\Delta(q_0, \epsilon)$.

Case $v = v' \cdot \langle \cdot v'' \cdot \rangle$ and v'' is well-nested. Let $n = \#_\Sigma(v)$. We have $\text{ann-nod}(v) = \text{ann-nod}(v') \cdot \langle \cdot n \cdot \text{ann-nod}(v'')^{n+1} \rangle$. Let σ be the stack in the second component of $(M, _)$. The run of $\text{cnd}(A)$ leading to (M, σ) must have the following form: Let

$(M', \sigma) = \llbracket ann-nod(v') \rrbracket_{str}^{\Delta_{cnd}}(M_0, \epsilon)$, and

$$M_0 = \{(q_0, \alpha', dom(\alpha'), sdown^{\Delta_{\Sigma}}(q', Q')) \mid (q', \alpha', _, Q') \in M'\}.$$

Let $(M'', _) = \llbracket n \cdot ann-nod(v'') \rrbracket_{str}^{\Delta_{cnd}}(M_0, \sigma \cdot M')$. We distinguish two cases, depending on whether α binds x to a position of v' or not.

Case $\alpha = [x/\pi]$ where $\pi \in pos(v')$. We have $v * \alpha = (v' * \alpha) \cdot \langle v'' \rangle$. We consider both implications independently.

" \Rightarrow " We suppose that $(q, \alpha, _, Q) \in M$ and have to show that $((q, Q), _) = \llbracket v * \alpha \rrbracket_{str}^{\Delta}(q_0, \epsilon)$. Since $(q, \alpha, _, Q) \in M$ there is $(q'', \alpha, \{x\}, S'') \in M''$, $(q', \alpha, _, Q) \in M'$ such that $S'' = sdown^{\Delta_{\Sigma}}(q', Q)$ and $q = q' @^{\Delta} q''$. By induction hypothesis, $(M', \sigma) = \llbracket ann-nod(v') \rrbracket_{str}^{\Delta_{cnd}}(M_0, \epsilon)$ and $(q', \alpha, _, Q) \in M'$ imply that there exists σ' such that $((q', Q), \sigma') = \llbracket v' * \alpha \rrbracket_{str}^{\Delta}(q_0, \epsilon)$. The induction hypothesis applied to $(q'', \alpha, \{x\}, Q'') \in M''$ implies that $((q'', Q''), _) = \llbracket v'' \rrbracket_{str}^{\Delta}(\langle \rangle^{\Delta}, \sigma' \cdot (q', Q))$. Since v'' is well nested, the stack component of $((q'', Q''), _)$ must be unchanged, i.e. equal to $\sigma' \cdot (q', Q)$. Hence, we can apply the closing rule of A showing that $((q' @^{\Delta} q'', Q), \sigma') = \llbracket v * \alpha \rrbracket_{str}^{\Delta}(q_0, \sigma)$. Since $q' @^{\Delta} q'' = q$ it follows that $((q, Q), _) = \llbracket v * \alpha \rrbracket_{str}^{\Delta}(q_0, \epsilon)$ as required.

" \Leftarrow " We suppose that $((q, Q), _) = \llbracket v * \alpha \rrbracket_{str}^{\Delta}(q_0, \sigma)$ and have to show that $(q, \alpha, _, Q) \in M$. Let $((q', Q'), _) = \llbracket v' * \alpha \rrbracket_{str}^{\Delta}(q_0, \sigma)$. By induction hypothesis, we have $(q', \alpha, _, Q') \in M'$. Let $((q'', Q''), _) = \llbracket v'' \rrbracket_{str}^{\Delta}(q_0, \sigma \cdot (q', S'))$. So $Q'' = sdown^{\Delta_{\Sigma}}(q', S')$. By induction hypothesis, we have $(q'', _, _, Q'') \in M''$. Hence also $(q'', \alpha, \{x\}, Q'') \in M''$. The closing rule permits to match the tuples with α of M' and M'' yielding $(q' @^{\Delta} q'', \alpha, _, Q') \in M$. Since α can only belong to a single tuple of M it follows that $q' @^{\Delta} q'' = q$ and $Q' = Q$. Hence, $(q, \alpha, _, Q) \in M$.

Case $\alpha = [x/n]$. So we have $v * \alpha = v' \cdot \langle x \cdot v'' \rangle$. We consider both implications independently.

" \Rightarrow " We suppose that $(q, \alpha, _, Q) \in M$ and have to show that $((q, Q), _) = \llbracket v * \alpha \rrbracket_{str}^{\Delta}(q_0, \epsilon)$. Since $(q, \alpha, _, Q)$ there are $(q'', \alpha, \{x\}, S'') \in M''$ and $(q', _, \emptyset, Q) \in M'$ such that $S'' = sdown^{\Delta_{\Sigma}}(q', Q)$ and $q = q' @^{\Delta} q''$. By induction hypothesis, $(M', \sigma) = \llbracket ann-nod(v') \rrbracket_{str}^{\Delta_{cnd}}(M_0, \epsilon)$ and $(q', _, \emptyset, Q) \in M'$ imply that there exists σ' such that $((q', Q), \sigma') = \llbracket v' * [] \rrbracket_{str}^{\Delta}(q_0, \epsilon)$. The induction hypothesis applied to $(q'', \alpha, \{x\}, Q'') \in M''$ implies that $((q'', Q''), _) = \llbracket x \cdot v'' \rrbracket_{str}^{\Delta}(\langle \rangle^{\Delta}, \sigma' \cdot (q', Q))$. Since v'' is well nested, the stack component of $((q'', Q''), _)$ must be unchanged, i.e. equal to $\sigma' \cdot (q', Q)$. Hence, we can apply the closing rule of A showing that $((q' @^{\Delta} q'', Q), \sigma') = \llbracket v * \alpha \rrbracket_{str}^{\Delta}(q_0, \sigma)$. Since $q' @^{\Delta} q'' = q$ it follows that $((q, Q), _) = \llbracket v * \alpha \rrbracket_{str}^{\Delta}(q_0, \epsilon)$ as required.

“ \Leftarrow ” We suppose that $((q, Q), _) = \llbracket v * \alpha \rrbracket_{str}^\Delta(q_0, \sigma)$ and have to show that $(q, \alpha, _, Q) \in M$. Let $((q', Q'), _) = \llbracket v' \rrbracket_{str}^\Delta(q_0, \sigma)$. By induction hypothesis, we have $(q', [], _, Q') \in M'$. Let $((q'', Q''), _) = \llbracket x \cdot v'' \rrbracket_{str}^\Delta(q_0, \sigma \cdot (q', S'))$. So $Q'' = sdown^{\Delta|_{\Sigma}}(q', S')$. By induction hypothesis, we have $(q'', \alpha, _, Q'') \in M''$. The closing rule permits to match the tuple with α in M'' with the tuple with $[]$ of M' . This yields $(q' @^\Delta q'', \alpha, _, Q') \in M$. Since α can only belong to a single tuple of M it follows that $q' @ q'' = q$ and $Q' = Q$. Hence, $(q, \alpha, _, Q) \in M$.

Case else. So now either $\alpha = []$ or $\alpha = [x/\pi]$ with $\pi - n - 1 \in pos(v'')$ and we have $v * \alpha = v' * \cdot \langle v'' * \alpha \rangle$. We consider both implications independently.

“ \Rightarrow ” We suppose that $(q, \alpha, _, Q) \in M$ and have to show that $((q, Q), _) = \llbracket v * \alpha \rrbracket_{str}^\Delta(q_0, \epsilon)$. Since $(q, \alpha, _, Q)$ there are $(q'', \alpha, \emptyset, S'') \in M''$ and $(q', [], \emptyset, Q) \in M'$ such that $S'' = sdown^{\Delta|_{\Sigma}}(q', Q)$ and $q = q' @^\Delta q''$. By induction hypothesis, $(M', _) = \llbracket ann-nod(v') \rrbracket_{str}^{\Delta_{cnd}}(M_0, \epsilon)$ and $(q', [], _, Q) \in M'$ imply that there exists σ' such that $((q', Q), \sigma') = \llbracket v' * [] \rrbracket_{str}^\Delta(q_0, \epsilon)$. The induction hypothesis applied to $(q'', \alpha, _, Q'') \in M''$ implies that $((q'', Q''), _) = \llbracket v'' * \alpha \rrbracket_{str}^\Delta(\langle \rangle^\Delta, \sigma' \cdot (q', Q))$. Since v'' is well nested, the stack component of $((q'', Q''), _)$ must be unchanged, i.e. equal to $\sigma' \cdot (q', Q)$. Hence, we can apply the closing rule of A showing that $((q' @^\Delta q'', Q), \sigma') = \llbracket v * \alpha \rrbracket_{str}^\Delta(init, \sigma)$. Since $q' @^\Delta q'' = q$ it follows that $((q, Q), _) = \llbracket v * \alpha \rrbracket_{str}^\Delta(init, \epsilon)$ as required.

“ \Leftarrow ” We suppose that $((q, Q), _) = \llbracket v * \alpha \rrbracket_{str}^\Delta(init, \sigma)$ and have to show that $(q, \alpha, _, Q) \in M$. Let $((q', Q'), _) = \llbracket v' \rrbracket_{str}^\Delta(q_0, \sigma)$. By induction hypothesis, we have $(q', [], _, Q') \in M'$. Let $((q'', Q''), _) = \llbracket v'' * \alpha \rrbracket_{str}^\Delta(q_0, \sigma \cdot (q', S'))$. So $Q'' = sdown^{\Delta|_{\Sigma}}(q', S')$. By induction hypothesis, we have $(q'', \alpha, _, Q'') \in M''$. The closing rule permits to match the tuple with α in M'' with the tuple with $[]$ of M' . This yields $(q' @^\Delta q'', \alpha, _, Q') \in M$. Since α can only belong to a single tuple of M it follows that $q' @ q'' = q$ and $Q' = Q$. Hence, $(q, \alpha, _, Q) \in M$.

□

We can thus obtain an EQA algorithm by running the streaming evaluator of the earliest automaton $cnd(A)$. Without removing candidates that are certainly non-answers, however, it would maintain and update many candidates that are no more alive, leading to quadratic time in $O(m^2)$ even for bounded concurrency.

Certain non-answers can be detected in analogy to certain answers. For this we can add a set of states that are safe for rejection. At the beginning, this is $safe^\Delta(Q \setminus F)$ later on we have to use the function $sdown^\Delta$ to update it when moving down into subtrees.

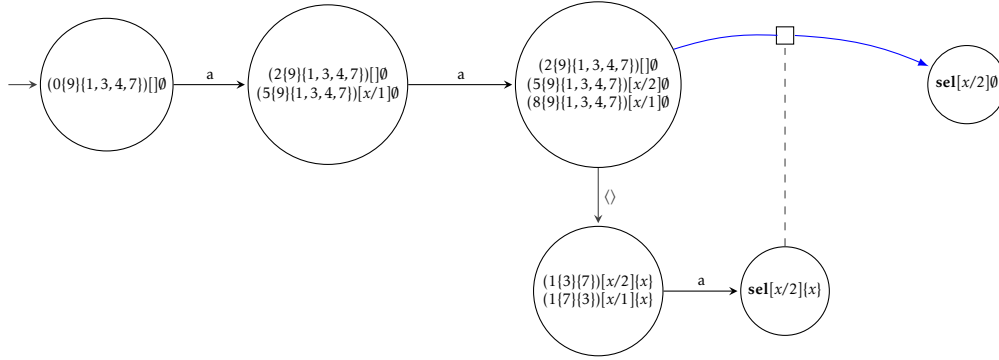


Figure 11.2: A run of the earliest candidate automaton $cnd(A_{e(S)})$ for the dSHA A in Figure 8.1. The underlying earliest automaton $A_{e(S)}$ is given in Figure 8.5.

11.4 Earliest Monadic Query Answering

The naive monadic query answering algorithm obtained by evaluating the candidate automaton can be improved by making the automaton detect certain answers and non-answers earlier. The objective is to no more buffer all answer candidates. Instead, certain answers should be output to some output stream, while all certain answers and non-answers should be removed from the candidate buffer.

We next show how to obtain top-down *EQA* algorithms for monadic queries with complete suffix projection – either in-memory or in streaming mode. They enumerate the certain answers and non-answers of a query in an earliest manner. The enumerated certain answers may be shown to some user, while the enumerated certain non-answers may be removed eagerly from the candidate buffer.

11.4.1 Earliest Candidate Automata

For regular monadic queries defined by dSHA A we can obtain an *EQA* algorithm with complete suffix projection by evaluating the earliest candidate automaton of A that we define next.

For any dSHA A over Σ^x , the earliest candidate automaton with complete suffix projection is the $dSHA_{\infty}^{\downarrow} cnd(A_{e(S)})$. The top-down evaluator of the earliest candidate automaton can decide at each prefix $ann-nod(v)$, whether the node π is certain for selection for x . The automaton $cnd(A_{e(S)})$ is the $dSHA_{\infty}^{\downarrow}$ obtained from the earliest automaton $A_{e(S)}$ with complete suffix projection by constructing its candidate automaton.

Example 11.16. For the $d\text{SHA}$ A in Figure 8.1, the earliest candidate automaton $A_{e(S)}$ constructed for the $d\text{SHA}$ A is given in Figure 8.5. An example run of the earliest candidate automaton $\text{cnd}(A_{e(S)}) = (\Sigma, Q', \Delta', I', F')$ is given in Figure 11.2. If $M_0 \in I'$ then it satisfies:

$$\llbracket a1a2\{3a4\} \rrbracket_{\text{str}}^{\Delta'}(M_0, \epsilon) = (\{\mathbf{sel}[x/2]\{x\}\}, \sigma) \quad \text{where } \sigma = 5\{9\}\{1, 3, 4, 7\}$$

The run assigns the candidate $[x/2]$ to the selection state **sel**, showing that node 2 is a certain answer of $\text{qry}_S(\mathcal{L}(A))$. The fact that the candidate $[x/1]$ is no more present, by contrast, shows that node 1 is a certain non-answer of the query. It's run by $A_{e(S)}$ blocked, so it was removed by the earliest candidate automaton. Also the run of $A_{e(S)}$ on the empty candidate $[]$ blocked, so it got also removed.

Proposition 11.17 (Enumerating certain answers and non-answers). Let A be a complete $d\text{SHA}$ with alphabet Σ^x . Let $\text{cnd}(A_{e(S)}) = (\Sigma, Q', \Delta', I', F')$ and $M_0 \in I'$. If $v \in \text{pref}_S(\mathcal{N}_\Sigma)$ is a prefix and $(M, \sigma) = \llbracket \text{ann-nod}(v) \rrbracket_{\text{str}}^{\Delta'}(M_0, \epsilon)$ then:

- $CA^{\text{qry}_S(\mathcal{L}(A))}(v) = \{\pi \in \text{pos}(v) \mid (\mathbf{sel}, [x/\pi], V) \in M\}$
- $CNA^{\text{qry}_S(\mathcal{L}(A))}(v) = \{\pi \in \text{pos}(v) \mid \nexists D \nexists V. (D, [x/\pi], V) \in M\}$

Proof. Let $v \in \text{pref}_S(\mathcal{N}_\Sigma)$ be a nested word prefix and $(M, \sigma) = \llbracket \text{ann-nod}(v) \rrbracket_{\text{str}}^{\Delta'}(M_0, \epsilon)$. Let $D_0 \in I_{e(S)}$.

- For any $\pi \in \text{pos}(v)$, Corollary 11.9 shows that $\pi \in CA^{\text{qry}_S(\mathcal{L}(A))}(v)$ is equivalent to $\exists \sigma. (\mathbf{sel}, \sigma) = \llbracket v * [x/\pi] \rrbracket_{\text{str}}^{\Delta_{e(S)}}(D_0, \epsilon)$. By Lemma 11.15 this is equivalent to $\exists V. (\mathbf{sel}, \alpha, V) \in M$. Hence:

$$CA^{\text{qry}_S(A)}(v) = \{\pi \in \text{pos}(v) \mid (\mathbf{sel}, [x/\pi], V) \in M\}$$

- For any $\pi \in \text{pos}(v)$, Corollary 11.9 shows that $\pi \in CNA^{\text{qry}_S(\mathcal{L}(A))}(v)$ is equivalent to that $\llbracket v * [x/\pi] \rrbracket_{\text{str}}^{\Delta_{e(S)}}(D_0, \epsilon)$ is undefined. By Lemma 11.15 this is equivalent to $\nexists D \nexists V. (D, \alpha, V) \in M$. Hence:

$$CNA^{\text{qry}_S(A)}(v) = \{\pi \in \text{pos}(v) \mid \nexists D \nexists V. (D, [x/\pi], V) \in M\}$$

□

Theorem 7 (Streaming EQA for Monadic Queries). Let A be a complete $d\text{SHA}$ with alphabet Σ^x and schema S and $Q = \text{qry}_S(\mathcal{L}(A))$. Streaming EQA for query Q with

complete suffix projection on a hedge $h \in \mathcal{H}_\Sigma$ can be done by either computing $A_{e(s)}$ statically or dynamically on the fly:

- in time $O(c)$ per event, where c is the concurrency of query Q at the event, after a precomputation in time $O(|A_{e(s)}|)$, or
- in time $O(c \cdot m)$ per event.

In both case, the memory is bounded by $O(c \cdot \text{depth}(h) + O(|A_{e(s)}|))$.

Proof. Given a dSHA A defining a monadic query, it is sufficient to run $A_{e(s)}$ on $nw(h)$ in streaming mode, and to output the detected certain answers .

Let $cnd(A_{e(s)}) = (\Sigma, \mathcal{Q}', \Delta', I', F')$ and $I' = \{M_0\}$. Let $h \in \mathbf{S}$ be a hedge and $v \in \text{prefs}(nw(h))$ an event. Let:

$$(M, \sigma) = \llbracket ann-nod(v) \rrbracket_{str}^{\Delta'}(M_0, \epsilon)$$

By Proposition 11.17, we have

- $CA^{qry_s(\mathcal{L}(A))}(v) = \{\pi \in pos(v) \mid (\mathbf{sel}, [x/\pi], V) \in M\}$
- $CNA^{qry_s(\mathcal{L}(A))}(v) = \{\pi \in pos(v) \mid \nexists D \nexists V. (D, [x/\pi], V) \in M\}$

It is sufficient to remove all tuples $(\mathbf{sel}, [x/\pi], V) \in M$ from M and output π . Since the partial runs of certain non-answers have all been blocked, only alive candidates remain in the current state. Hence, the cost per event will be in $O(c)$.

Either $A_{e(s)}$ is precomputed statically, and in this case, the time per event stays for doing one transition with $cnd(A_{e(s)})$ is in $O(c)$, but a precomputation time in $O(|A_{e(s)}|)$ is needed.

Or else the transition rules of $A_{e(s)}$ needed for evaluating h by $cnd(A_{e(s)})$ are computed dynamically on-the-fly. For each transition of $A_{e(s)}$ one needs to compute $acc^\Delta(Q)$ for two sets $Q \subseteq \mathcal{Q}$ which can be done in $O(m)$. So for each transition rule of the candidate automaton $cnd(A_{e(s)})$ one needs time in $O(c \cdot m)$. \square

This complexity for dSHAs improves on Gauwin et al. [Gauwin et al. 2009b] for dNwas, which required time $O(c \cdot n^2)$ per event after $O(n^3)$ preprocessing time.

For monadic queries where c is bounded for all events and input hedges, the complexity per event is reduced to $O(m)$.

Corollary 11.18 (Top-Down In-Memory EQA for Monadic Queries). *Let A be a complete $d\text{SHA}$ with alphabet Σ^x and schema S and $Q = \text{qry}_S(\mathcal{L}(A))$. Top-down in-memory EQA for query Q with complete suffix projection on a hedge $h \in \mathcal{H}_\Sigma$ can be done by either computing $A_{e(S)}$ statically or dynamically on-the-fly. Note here that although the hedge h is completely present in memory, the events would refer to the ones of the top-down traversal of the hedge, and thus the per-event complexity:*

- in time $O(c)$ per event, where c is the concurrency of query Q at the event, after a precomputation in time $O(|A_{e(S)}|)$, or
- in time $O(c \ m)$ per event.

Proof. This follows from Theorem 7 and the fact that streaming evaluation and top-down in-memory evaluation are basically doing the same state transitions. \square

We finally notice that running the $d\text{SHA}^\downarrow$ s by using state-and-stack sharing [Debarbieux *et al.* 2015], the running time per event can be reduced to:

- in time $O(n_{e(S)})$ per event, where $n_{e(S)}$ is the set the number of states of $A_{e(S)}$, and a precomputation in time $O(|A_{e(S)}|)$, or
- in time $O(n_{e(S)} \ m)$ per event.

11.4.2 Adding Subhedge Projection

In order to add congruence projection to the EQA algorithm with complete suffix projection, we have to run the automaton $A_{e(S)}^{cgr(S)}$. For adding earliest query answering to safe-no-change projection, we have to run the automaton $A_{e(S)}^{snc}$ instead.

Example 11.19. *The earliest congruence projection for the XPath -like query `child*-list[child-item]` on nested lists is shown in Figure 11.4. It binds x in state 2D1 after having read a top-level `list-element` and then checks for the existence of a child element `item`, going into selection state 1D3 once it is found. All other children of the top-level `list element` are projected in state 2D3.*

However, there is one additional issue with monadic queries: Nested word prefixes cannot be considered as irrelevant for the subsequent subhedge, if the selection variable x can be bound there, even if the acceptance doesn't depend on where it will be bound. This is since the position of the binding is to be returned by any query answering algorithm.

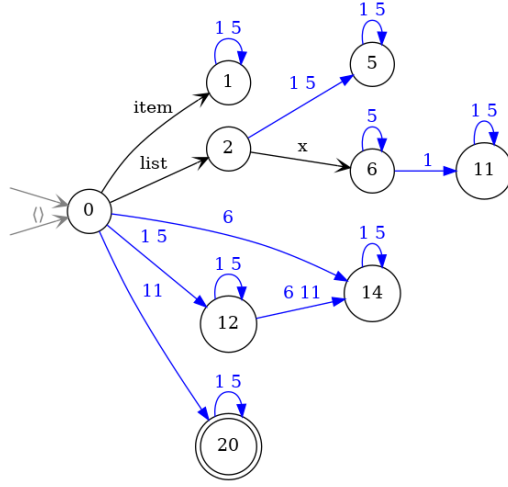


Figure 11.3: The dSHA for the XPATH query `self::list[child::item]`. The selection position is indicated by x .

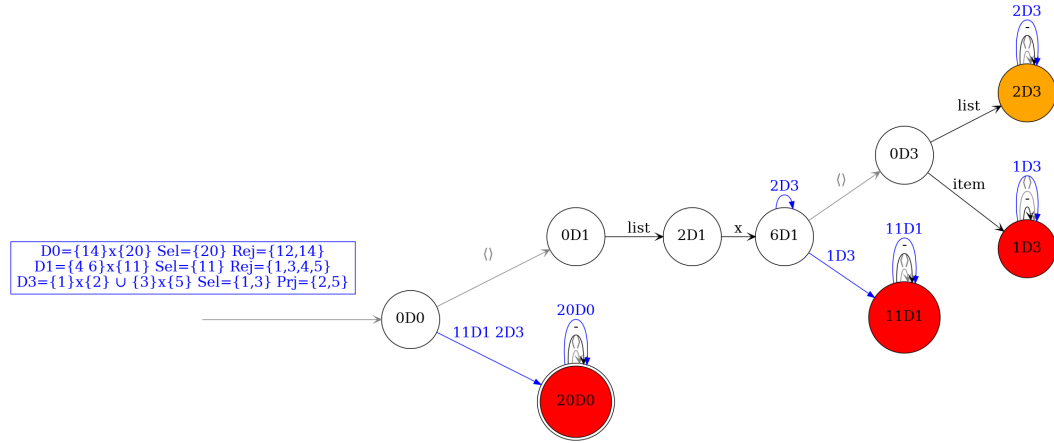


Figure 11.4: The earliest congruence projection $A_{e([N-List] \cap [one_x])}^{cgr([N-List] \cap [one_x])}$ for the dSHA A in Figure 11.3 with $F_S = \{14, 20\}$. It defines the query of the XPATH `self::list[child::item]`.

Definition 11.20. We call a prefix v binding irrelevant for L and S if there does not exist any hedge h containing x and suffix w such that $v \cdot w \in nw(S)$ and $v \cdot nw(h) \cdot w \in nw(L)$.

Our subhedge projecting evaluator can project only at subhedge irrelevant prefixes that are also binding irrelevant. Given an dSHA A and a set of schema final-states F_S , one can decide whether a prefix is subhedge irrelevant if the current state q does accept hedges containing x but can access some other state that is not safe for rejection and permits x -bindings. In ground Datalog, we can distinguish binding irrelevant states q by imposing the following rules for all $q, q' \in Q$:

`binding_irrelev(q) :- state(q), not binding_relev(q).`

`binding_relev(q) :- not bind_x(q), acc(q,p), bind_x(p), not rej(p).`

Note in our example of the XPATH-like filter [self-list-child-item], that state 2D3 of the $d\text{SHA} \downarrow A_{e(\llbracket \text{N-List} \rrbracket \cap \llbracket \text{one}_x \rrbracket)}^{cgr(\llbracket \text{N-List} \rrbracket \cap \llbracket \text{one}_x \rrbracket)}$ in Figure 11.4 is binding irrelevant, since x must be bound on the top-level `list` element in state 2D1, so it cannot be bound on any `list` element below.

Chapter 12

Experiments on Regular XPath Evaluation on XML Streams

Abstract

We describe the *ASTREAM* tool which implements our *EQA* algorithms with projections. It permits to answer regular XPath queries on XML streams. We show experimentally that complete subhedge and suffix projection make *ASTREAM* competitive in time efficiency with the best existing tools for answering XPath queries on XML streams.

Contents

12.1	Introduction	284
12.2	Streaming Evaluation Tool: <i>ASTREAM</i>	284
12.3	Experiments without projection: Linear increase with size	286
12.3.1	Scaling-Up Document Sizes without Subhedge Projection	286
12.3.2	Being Earliest	287
12.3.3	Factorization	287
12.4	Experiments with projection	288
12.4.1	Evaluation Measures	288
12.4.2	Earliest Congruence Projection	290
12.4.3	Earliest Safe-No-Change Projection	291
12.4.4	Comparison to External Tools	292
12.4.5	Experiments with Lick and Schmitz' Benchmark	293

12.1 Introduction

We discuss our implementation of the *EQA* algorithm for dSHAS with complete subhedge and suffix projection from Chapter 11 in the *ASTREAM* tool. Both safe-no-change and congruence projection are supported, though only in streaming mode. The application domain that the *ASTREAM* tool targets is the answering of regular XPATH queries on XML streams.

For the experimentation, we start the from collection of deterministic SHAS collection constructed with the compiler from [Niehren & Sakho 2021] for the forward regular XPATH queries A1-A8 of the XPATHMARK benchmark [Franceschet 2005b] in Figure 4.1 and the additional regular XPATH queries that we used for testing evaluators on XPATHMARK documents in Figure 9.2. We also use the 78 benchmark dSHAS constructed for the subcorpus of regular XPATH queries in Figure 6.2 selected from Lick and Schmitz’ benchmark, as described in Chapter 6, and published in the Software Heritage archive [Al Serhali & Niehren 2022]. These are regular XPATH queries from real-world XSLT and XQUERY programs.

It turns out that congruence projection projects much more strongly than safe-no-change projection for at least half of the benchmark queries. It reduces the running time for all regular XPATH queries considerably since large parts of the input hedges can be projected away. In our benchmark, the projected percentage ranges from 75.7% to 100% of the input stream. For XPATH queries that contain only child axes, the earliest query answering algorithm of *ASTREAM* with congruence projection becomes competitive in efficiency with the best existing streaming tool called QUIXPATH [Debarbieux *et al.* 2015], which however is not always earliest for some queries. Our current implementation of earliest congruence projection in *ASTREAM* is slower by a factor of 1.3 to 2.9 than QUIXPATH on the benchmark queries. The improvement is smaller for XPATH queries with the descendant axis, where less subhedge projection is possible. Instead, some kind of descendant projection would be needed. Still, even in the worst case in our benchmark, earliest congruence projection with *ASTREAM* is currently only by a factor of 13.8 slower than QUIXPATH.

12.2 Streaming Evaluation Tool: *ASTREAM*

We are using and developing the *ASTREAM* tool for answering monadic dSHA queries on XML streams with schema restrictions. Version 1.01 of *ASTREAM* was presented

in [Al Serhali & Niehren 2023a] and supports earliest query answering without projection. Given a $d\text{SHA}$ A defining a monadic query, a set of schema final states F_S and an XML stream w , it constructs on-the-fly the needed part of the $d\text{SHA}^\downarrow A_{e(S)}$ while parsing w and evaluating A on the hedge encoding w .

For the FCT conference version [Al Serhali & Niehren 2023b], we enhanced ASTREAM with safe-no-change projection. This leads to version ASTREAM 2.01. It constructs the earliest safe-no-change projection $d\text{SHA}^\downarrow A_{e(S)}^{snc}$ on the fly while evaluating the monadic query defined by A on the hedge encoding the XML stream w . Subhedge projection can be switched on or off, in order to compare both versions without further implementation differences.

For the journal version [Al Serhali & Niehren 2024], we added earliest congruence projection and integrated it into ASTREAM leading to ASTREAM 3.0.

ASTREAM 3.0 is different from the two previous versions in that the $d\text{SHA}^\downarrow A_{e(S)}^{cgr}$ is constructed statically and entirely, independently of the input hedge. We then use a generic earliest streaming evaluator for SHA^\downarrow s that rejects in rejection states, selects in selection states, and projects in all subhedge projection states. This evaluator could also be run with $A_{e(S)}^{snc}$ or $A_{e(S)}$, as long as these do not grow too big.

It should be noticed that $A_{e(S)}^{cgr}$ turned out to be nicely small for our whole benchmark, while the other $d\text{SHA}^\downarrow A_{e(S)}^{snc}$ and $A_{e(S)}$ risk becoming bigger. As stated earlier, we did not construct these $d\text{SHA}^\downarrow$ s so far for our benchmark queries. This is why we continued to run the earliest streaming with safe-no-change projection with ASTREAM 2.01, while we used ASTREAM 3.0 for earliest streaming with congruence projection.

All ASTREAM versions rely on Java's abc-Datalog for computing the least fixed points of Datalog programs in a bottom-up manner. Datalog programs are needed for all logical reasoning: for computing subsets of states that are safe for rejection, selection, or subhedge projection on all levels. Also, the difference relations are computed based on Datalog. Note that earliest on-the-fly query evaluation requires running Datalog during query evaluation, while with a static approach for constructing $d\text{SHA}^\downarrow$ s entirely, Datalog is only needed at preprocessing time during the automaton construction.

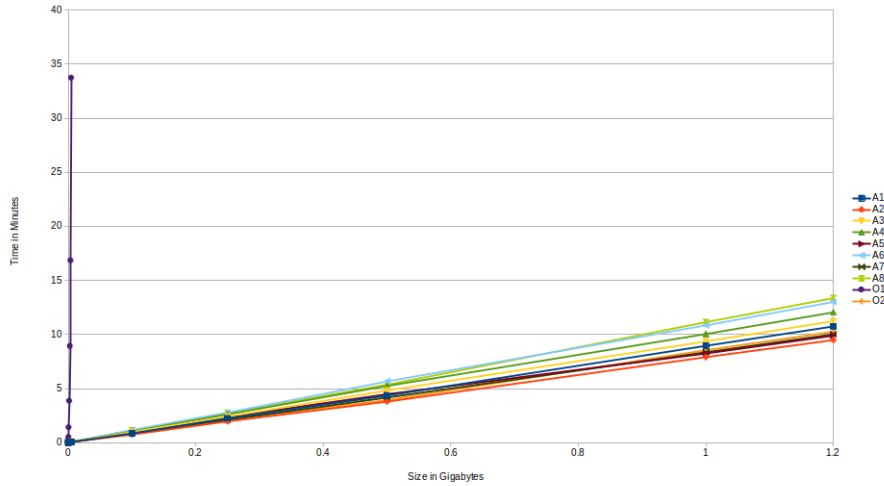


Figure 12.1: Running times of ASTREAM 1.01 for streaming XPATHMARK queries on XML documents whose size scale from 27KB to 1.2GB. Only the running time of query O1 is not scaling up linearly, and thus running out of time for 128KB already. The concurrency of all other queries is bounded on these documents.

12.3 Experiments without projection: Linear increase with size

In this section, we present our experiments with ASTREAM 1.01, i.e, earliest query answering without projection. We first test with a list of regular queries from XPATHMARK on documents of sizes ranging from 27KB to 1.2GB. The correctness was shown by comparison to SAXON evaluator. We then analyze the behavior of different queries with different concurrencies and compare it to the QUIXPATH tool. Finally, we discuss the needed optimization to handle queries with high concurrency.

12.3.1 Scaling-Up Document Sizes without Subhedge Projection

We run ASTREAM 1.01 without subhedge projection on XML documents of increasing size up to 1.2GB, but can also stream much larger documents >100GB. Up to 1GB, we verified the correctness of the answer sets by comparison to SAXON's in-memory evaluator (which is limited to 1GB).

The running times on the scaling documents are reported in Figure 12.1. The times grow linearly for all these queries given that their concurrency is bounded by 2, except for O1 where it grows quadratically, since its concurrency grows linearly

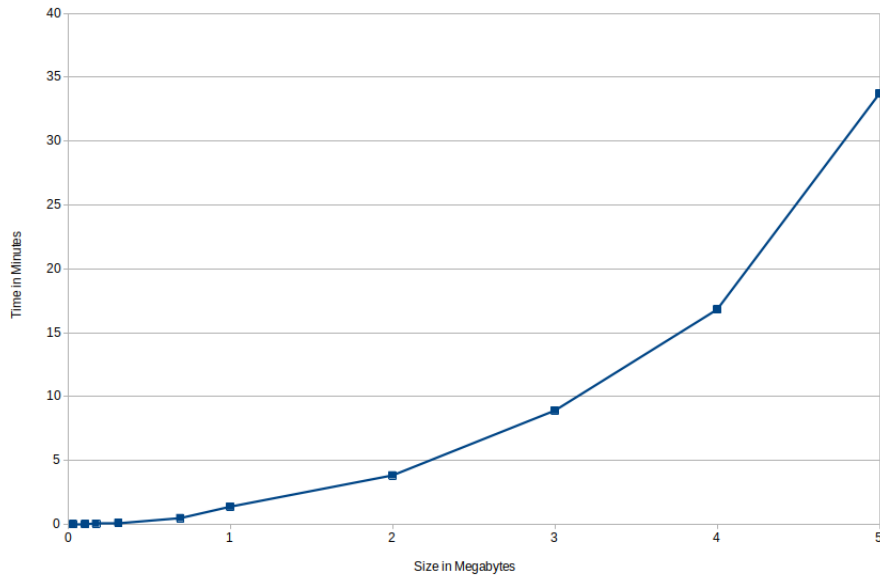


Figure 12.2: The quadratic running time of ASTREAM 1.01 on XPath query O1 with linear concurrency on XML documents ranging from 27 KB to 5 MB.

with the size of the document. The quadratic growth can be observed clearly in Figure 12.2 on smaller documents scaling from 27KB to 5MB.

On average, for A1-A8, ASTREAM 1.01 is by a factor of 60 slower than QUIXPath, so it requires minutes instead of seconds. This high execution time is expected since we are running without any projection algorithm, while QUIXPath support subtree and descendant projection.

12.3.2 Being Earliest

On the one hand side, QUIXPath cannot stream O2 on large documents, since it is not *earliest*. Even though the concurrency of O2 per event is at most 1, linearly many candidates are buffered by QUIXPath, until the buffer overflows for documents larger than 5GB. O2 poses no problem to ASTREAM since it is earliest, so it buffers no more than one candidate per event.

12.3.3 Factorization

On the other hand, QUIXPath can stream queries with high concurrency (O1) due to QUIXPath's state and stack sharing, i.e., the sharing of the computations of all concurrent candidates in the same state. All three ASTREAM versions currently lack

an alternative for factorizing the computations of multiple candidate that share the same configuration. In *ASTREAM* 1.01, the earliest automaton were built on-the-fly while keeping safety sets for selection and rejection states, the configuration was not a simple state and stack but also two additional sets of states, which made it too costly to go for an approach through factorization, like state-and-stack sharing. We believe that this constraint will be removed once the earliest automata is built statically, as discussed in the next section.

Given that the computation time of *ASTREAM* 1.01 per event is $c * n$ where $c = d$ is the documents size for O1, the computation time for O1 is in $O(d * n)$ per event, and thus in $O(d^2 * n)$ for the whole document. The quadratic size increase in d of *ASTREAM* 1.01 can be seen in the experiments on smaller documents in Figure 12.2. In the experiments on larger documents in Figure 12.1, *ASTREAM* 1.01 runs out of time for O1 on documents higher than 5MB.

12.4 Experiments with projection

We now turn to the experiments with projection. The primary tests are conducted on queries from the *XPATHMARK* performance benchmark, given its scalability with larger documents. Additionally, we present results for the Lick and Schmitz' benchmark.

We first define the evaluation metrics used for the *XPATHMARK* tests, followed by experiments with both congruence and safe-no-change projections. Next, we compare the performance of the more efficient congruence projection with that of *XPATH*. Finally, we briefly present the results from the Lick and Schmitz' benchmark.

12.4.1 Evaluation Measures

We want to measure the time for running the three streaming query evaluators for all dSHAS obtained from the *XPATH* expressions in the collection in Figures 4.1 and 9.2.

One advantage of the *XPATHMARK* benchmark is that it comes with a generator of XML documents to which the queries can be applied and that can be scaled in size. We created an XML document of size 1.1 GB, which is sufficiently large to make streaming mandatory. Our experiments show that the efficiency of query evaluation grows linearly with the size of the non-projected part of the XML document. This holds for each of our three evaluators. Therefore, measuring the time of the

evaluator on XML documents of other sizes would not show any new insights, except that memory consumption remains moderate too.

The time gain of projection for a query Q is the percentage of time saved by projection during query evaluation when ignoring the pure document parsing time t_{parse} seconds:

$$\text{time-gain}_Q = 100 * (1 - (t_{\text{noproject}_0} - t_{\text{parse}}) / (t_{\text{project}_0} - t_{\text{parse}}))$$

We can measure the time gain for earliest safe-no-change projection by *ASTREAM* 2.01 and for earliest congruence projection by *ASTREAM* 3.0, since projection can be switched off in both of them. The disadvantage of the time gain is that it depends on the details of the implementation.

The event gain is a better measure for the projection power. The parser sends a stream of needed events to the SHA^\downarrow , while ignoring parts of the input document that can be projected away. For this, it must always be informed by the automaton about what kind of events can be projected in the current state. The event gain of projection event-gain_Q is then the percentage of events that are recognized to be irrelevant and thus no more created by a projecting evaluator for query Q . One might expect that the time gain is always smaller than the event gain.

$$\text{time-gain}_Q \leq \text{event-gain}_Q$$

Indeed, this will be confirmed by our experiments. We believe that these discrepancies between these two measure indicate how much room remains for optimizing the implementation of an evaluator. In this way, we can observe that some room for further optimizations in *ASTREAM* 3.0 still remains.

We use Java's *XMLStreamReader* Interface v1.0 from *javax.xml.stream* package to parse and stream XML files in Scala. Parsing a 1.1 GB XML documents requires $t_{\text{parse}} = 15$ seconds, while querying it without projection took us in average $t_{\text{noproject}_{\text{avg}}} = 752$ seconds, while varying from 600 to 900 seconds. The average processing time for 1 MB is thus 0.72 seconds. Once knowing this, one can predict the expected evaluation time from the size of the non-projected part of the XML document. It is:

$$\text{event-gain}_Q * 1100 * 0.72 \text{ seconds} + t_{\text{parse}} \text{ seconds}$$

Without projection, the parser generates 1,012,018,728 events for the 1.1GB XML document. This is the baseline for computing event-gain_Q for all queries Q .

Query ID	#ans- wer	time in sec	time gain	event gain
A1	38267	29.3	98.1%	98.9%
A2	117389	171.8	77.9%	81.1%
A3	117389	33.4	97.4%	97.8%
A4	24959	33.2	97.8%	98.9%
A5	50186			
A6	30307	45.9	96.9%	98.2%
A7	179889	30.5	98.0%	98.7%
A8	67688	37.8	97.3%	98.7%
A0	1	21.3	99.1%	100.0%
A1_0a	6	15.2	99.9%	100.0%
A1_0b	0	0.0	100.0%	100.0%
A1_0c	3600816	226.6	72.9%	75.7%
A1_1a	2	187.8	78.1%	80.3%
A1_1d	2	217.3	74.8%	80.3%
A1_2	1062965	238.2	68.2%	76.0%
A1_3	25074	16.2	99.1%	99.8%
A1_4	5170	17.5	99.7%	100.0%
A1_5	6	14.3	100.0%	100.0%
A1_6	117389	188.7	75.4%	81.1%
A2_1	50186	199.7	76.5%	81.1%
A4_0	91650	21.8	99.0%	99.3%
A4_1	1	12.0	100.0%	100.0%

Figure 12.3: Time gain and event gain by earliest congruence projection with ASTREAM 3.01 on a 1.1GB XPATHMARK stream.

We ran ASTREAM with Scala v2.13.3, on a machine with the operating system Ubuntu 20.04.06 LTS (64-bit). The machine is Dell Precision-7750 machine, equipped with an Intel® Core™ i7-10875H CPU @ 2.30GHz × 16 and 32 GB of RAM.

12.4.2 Earliest Congruence Projection

Recall that we were able to build statically all the automata for this collection and thus, we are here using the earliest automata with congruence projection presented in Figure 9.3. We present the measures of our evaluator with the earliest congruence projection in Figure 12.3. The event gain is high, varying between 75.7% for A1_0c to 100% for A0, A1_0a, A1_0b, A1_4, A1_5, and A4_1. The event gain for queries without descendant axis is above 98.2%. For these queries, all subtrees that are not

on the main path of the query can be projected away. In particular, only subtrees until a fixed depth have to be inspected. These are the queries A1, A4, A6-A8 and A4_0 and the queries with event gain 100% listed above. The time gain for all these queries is a little smaller than the event gain but still above 98.2%.

We next consider the second type of queries having the descendant axis, specifically A2, A1_0c, A1_1a, A1_1d, A1_2, A1_6, and A2_1. Intuitively, a lower event gain is expected in these cases because subhedge projection with the descendant axis cannot directly exclude an entire subtree under a given element; all descendant elements must be inspected. For instance, the query A1_0c equal to `/site//@*` must select all attributes of all elements under the root element *site*, requiring the inspection of every XML element in the document, for attribute presence, except for the root element *site*.

The lower event gain reported for the aforementioned queries align with the above expectation, showing percentages ranging from 75.7% to 81.1%. Also, these queries exhibit a larger gap between time gain and event gain, which ranges from 2.2% to 5.7% and will be subject to future optimization. It is worth noting that some of these queries combine multiple features at once, like queries A1_1a and A1_1d which have descendant axis, filter with attributes, and string comparison.

The only exception to this trend for the queries having descendant axis are A3 and A1_3, with respective event gains of 97.8% and 99.8%. The reason behind this high gain is that these queries starts from the root, with a child axis path of depth 3 before querying the descendant axis. This initial path allows for the exclusion and projection of most elements under *site* that do not fit the specified path, with their entire subtrees.

12.4.3 Earliest Safe-No-Change Projection

We compare earliest safe-no-change projection with the earliest congruence projection in see Table 12.4. For this we restrict ourselves to the queries A1 – A8 from the XPATHMARK in Table 4.1.

The XPATH queries A1, A4, A6, A7, A8 don't have descendant axes. The time gain for safe-no-change projection on these queries is between 92.3 – 98.9%. For earliest congruence projection, the time gain was better for A1, A4, A6 with at least 96.6%. For A7 and A8, the time gain of safe-no-change projection is close to the event gain of congruence projection, but the time gain of congruence projection is 1.5% lower. So we hope that an optimized version of congruence projection could outperform

Query ID	time (sec) snc	time (sec) congr	time gain snc	time gain congr	event gain congr
A1	72.8	29.3	92.3%	98.1%	98.9%
A2	664.6	171.8	8.5%	77.9%	81.1%
A3	666.1	33.4	10.9%	97.4%	97.8%
A4	78.5	33.2	92.4%	97.8%	98.9%
A5	77.7		92.3%		
A6	65.2	45.9	95.1%	96.9%	98.2%
A7	24.6	30.5	98.8%	98.0%	98.7%
A8	24.8	37.5	98.9%	97.3%	98.7%

Figure 12.4: Time gain for earliest safe-no-change and earliest congruence projection, and the event gain of earliest congruence projection.

safe-no-change projection in all cases.

The XPATH queries with descendant axis are A2, A3, A5. For A2 and A3 in particular, the time gain of safe-no-change projection is very low (not even 10.9%) while congruence projection yields at least 77.9%. We believe that this is a bug in our current implementation of safe-no-change projection, which may be prohibiting the projection of attributes and text nodes for these queries (but not for the others). Congruence projection on A2 gains 77.9% of time and 81.1% of events. This is much better, but still far from the projection power for queries without descendant axis. For A3, congruence projection gains 97.4% of time. This is much better than for A2 since the descendant axis in A3 is at the end, while for A2 it is at the beginning of the path. But even for A2, congruence projection is very helpful.

12.4.4 Comparison to External Tools

QUIXPATH was shown to be the most efficient large coverage external tool for regular XPATH evaluation on XML streams [Sebastian & Niehren 2016]. We therefore compare the efficiency of congruence projection to QUIXPATH in Table 12.5, and thereby by transitivity to the many alternative tools. We note that QUIXPATH performs early query answering by compilation to possibly nondeterministic Nwas with selection states, without always being earliest. Apart from this, it bases its efficiency on both subtree projection and descendant projection.

The experiments with QUIXPATH in [Sebastian & Niehren 2016] use the parse-free time. We therefore do the same for congruence projection by reducing the measured overall time by 15 seconds of parsing time.

Compared to the parsing-free times for the XPATH queries without descendant

Query ID	parse-free time (sec) QUIXPath	parse-free time (sec) congr. prj.	fract.
A1	11.0	14.3	1.3
A4	11.6	18.2	1.6
A6	10.7	30.9	2.9
A7	8.6	15.5	1.8
A8	8.8	22.5	2.6
A2	11.4	156.8	13.8
A3	11.5	18.4	1.6
A5	12.0		

Figure 12.5: Comparison of of parse-free timings in seconds for XPATHMARK queries with QUIXPath and earliest congruence projection.

axis, the earliest congruence projection demonstrates significant improvements.. On average, our projection is only slower by a factor of 1.3 - 2.9 than QUIXPath. This shows that our implementation is already close to be competitive with the existing XML streaming tools, while being the only one guaranteeing earliest query answering.

For query A2 with descendant axis, congruence projection is by a factor of 13.8 slower than QUIXPath. The reason is that QUIXPath supports descendant projection, while congruence projection concerns only subhedge. For query A3 with a descendant axis at the end, congruence projection is only by a factor of 1.6 behind, so descendant projection seems less relevant here.

12.4.5 Experiments with Lick and Schmitz' Benchmark

We reconsider the list of queries presented in Table 6.2. It contains queries from five different projects: docbook, htmlbook, teixsl, treedown, and histei. We also choose a single matching XML document per project of size less than 2MB. As done for the XPATHMARK benchmark, we attempted – and surprisingly succeeded – to statically compute the earliest automaton for these queries. We show the results in Table 12.2. The sizes we remarkably small for automata working top down where size explosion is a common and expected behavior. Beside the below three queries, with their respective measures:

12961 1629 (336)
 12960 1651 (343)
 04358 4223 (810),

the rest of list have a size less than 900. We used these automata to test with congruence projection. We could correctly answer all the queries in the list, yielding the same answer set as with SAXON. The overall time for computing all the answer sets was in 110000 ms on a Macbook pro Apple M1 laptop with 16GB of RAM. With SAXON in-memory evaluation it required around 45000 ms. The low running time of ASTREAM reflects the low concurrency of all the queries on all these documents according to Theorem 7. There are 12 queries with concurrency 1, 47 with concurrency 2, 6 with concurrency 3, and 12 with concurrency 4. Our efficiency results for ASTREAM thus show for the first time, that EQA is indeed feasible in practical scenarios with queries of low concurrency.

Table 12.2: Size measures of earliest $d\text{SHA}^\downarrow$ s for regular XPath queries of Lick and Schmitz' benchmark from Table 6.2.

Query ID	$d\text{SHA A}$ m(n)	$A_{e(S)}^{cgr(S)}$ size(#states)
00744	119 (38)	191 (51)
01705	207 (45)	197 (40)
01847	178 (45)	180 (43)
02000	153 (38)	151 (27)
02086	209 (45)	201 (40)
02091	134 (41)	305 (85)
02194	110 (35)	136 (38)
02697	139 (38)	194 (44)
02762	114 (36)	182 (49)
02909	157 (46)	253 (69)
03257	194 (46)	601 (141)
03325	159 (38)	234 (44)
03407	157 (38)	230 (44)
03410	169 (38)	254 (44)
03864	133 (38)	182 (44)
04245	167 (38)	250 (44)
04267	121 (38)	181 (50)
04338	193 (48)	571 (138)
04358	683 (96)	4223 (810)
04953	169 (38)	254 (44)
05122	149 (41)	343 (87)
05219	180 (45)	426 (94)

05226	192 (45)	462 (94)
05460	142 (41)	374 (95)
05463	162 (45)	437 (107)
05684	183 (36)	130 (19)
05735	151 (44)	201 (56)
05824	103 (34)	158 (43)
06027	114 (36)	182 (49)
06169	137 (38)	190 (44)
06176	166 (40)	226 (48)
06415	207 (52)	861 (216)
06458	163 (38)	242 (44)
06512	135 (40)	348 (88)
06639	140 (41)	368 (95)
06726	112 (35)	139 (38)
06794	134 (39)	189 (47)
06808	135 (38)	186 (44)
06924	137 (38)	190 (44)
06947	141 (38)	198 (44)
07106	143 (38)	202 (44)
07113	317 (74)	350 (79)
08632	198 (50)	758 (174)
09123	248 (63)	552 (136)
09138	124 (37)	204 (52)
10337	149 (44)	245 (67)
10745	202 (54)	471 (121)
11160	135 (38)	186 (44)
11227	230 (57)	492 (122)
11368	172 (46)	608 (152)
11478	131 (40)	171 (48)
11780	154 (44)	205 (56)
11958	162 (45)	437 (107)
12060	112 (36)	178 (49)
12404	133 (41)	217 (60)
12514	224 (54)	788 (180)
12539	156 (45)	210 (58)
12960	286 (61)	1651 (343)

12961	282 (60)	1629 (336)
12962	202 (51)	695 (163)
12964	198 (50)	685 (160)
13632	118 (33)	39 (9)
13640	130 (40)	170 (48)
13710	141 (38)	198 (44)
13804	131 (40)	242 (65)
14183	166 (46)	447 (110)
14340	101 (33)	122 (34)
15461	226 (56)	474 (123)
15462	180 (51)	264 (72)
15484	254 (64)	643 (147)
15524	175 (49)	225 (61)
15539	212 (55)	543 (136)
15766	204 (54)	391 (103)
15809	136 (41)	182 (51)
15848	168 (48)	241 (65)
17914	149 (42)	418 (106)
18330	119 (37)	186 (53)

Conclusion and Future Work

In this dissertation, we introduced, implemented, and tested multiple algorithms to provide a practical solution for the earliest query answering problem for regular queries on hedges.

First, we developed a schema-based determinization algorithm for both finite state automata (NFAS) and stepwise hedge automata (SHAS). We proved its equivalence to accessible determinization followed by schema-based cleaning and demonstrated its better efficiency. This efficiency is supported by complexity bounds and experimental results on a scalable set of queries. These deterministic automata provided a strong foundation for finding a practical solution for query answering algorithms with an automata-based approach.

We tested our newly introduced determinization algorithm on a corpus of forward navigational XPATH queries collected from practical online programs. The results showed that schema-based determinization is successful where schema-less methods fail, with 100% of the automata of the benchmark being determinized efficiently and later minimized. These findings suggest that schema-based determinization provides a practical solution for representing XPATH queries with small deterministic automata and brought us one step closer to answering our main problem.

Next, we tackled the algorithmic problem of projection. We introduced subhedge and suffix projection and demonstrated how to evaluate stepwise hedge automata in a top-down manner using both techniques. To achieve this, we defined the notions of irrelevant subhedges and suffixes. We then combined these techniques to obtain earliest $d\text{SHA}^\downarrow$ s with complete subhedge and suffix projection, providing related

evaluators that detect certain membership and non-membership in an earliest manner. These evaluators operate efficiently in both in-memory and streaming modes.

Finally, we extended our algorithms and evaluators to handle monadic queries, providing an earliest query answering (*EQA*) algorithm with complete subhedge and suffix projection for regular monadic queries represented by deterministic step-wise hedge automata. Our *EQA* algorithm operates in both streaming and top-down in-memory modes, with a complexity of $O(c)$ per non-projected event, where c is the concurrency of the query, assuming the earliest $d\text{SHA}^\downarrow$ with complete subhedge projection is constructed statically. If the automaton is built dynamically on the fly, the complexity becomes $O(c\ m)$ per event, where m is the size of the automaton defining the query. We updated our tool *ASTREAM* with our *EQA* algorithms and tested it on benchmarks where it proved competitiveness compared to the best existing tools.

Future Work

As future work, it would be interesting to explore both algorithmic and practical aspects further. First, investigating descendant projection for stepwise hedge automata to cover related path queries that rely on it, and finding a way to combine it with both suffix and subhedge projections; although we have initiated this work, time constraints have left it unfinished and not included in this dissertation. Second, examining the feasibility of *EQA* beyond regular queries is essential, as it would allow addressing a broader class, particularly queries with data joins that are common in real-world applications. One potential approach is leveraging networks of automata, as previously explored by Sebastian[Sebastian 2016]. Finally, investigating *EQA* for hyperstreams presents another intriguing research direction, allowing for parallel query evaluation on streams rather than sequential processing, the latter being the setting in our current work.

On the practical side, updating our compilers from nested regular expressions to support the backward axis would enable handling broader fragments of *XPATH* queries. Additionally, we have yet to implement all evaluators, particularly the in-memory ones. Exploring this direction and comparing our results with similar tools, such as *SAXON*, would be beneficial. Applying our algorithms to query other types of nested documents, such as *JSON*, could also be an interesting extension. Finally, improving and optimizing the current *EQA* implementation in streaming

mode would help reduce the execution time gap with other tools.

Bibliography

- [Abiteboul *et al.* 1995] S. Abiteboul, R. Hull and V. Vianu. Foundations of databases. Addison-Wesley, 1995.
- [Abiteboul *et al.* 2000] S. Abiteboul, P. Buneman and D. Suciu. Data on the web: from relations to semistructured data and XML. Morgan Kaufmann, 2000.
- [Al Serhali & Niehren 2022] Antonio Al Serhali and Joachim Niehren. *A Benchmark Collection of Deterministic Automata for XPath Queries*. In XML Prague 2022, Prague, Czech Republic, 2022.
- [Al Serhali & Niehren 2023a] Antonio Al Serhali and Joachim Niehren. *Earliest Query Answering for Deterministic Stepwise Hedge Automata*. In Benedek Nagy, editor, Implementation and Application of Automata, pages 53–65, Cham, 2023. Springer Nature Switzerland.
- [Al Serhali & Niehren 2023b] Antonio Al Serhali and Joachim Niehren. *Subhedge Projection for Stepwise Hedge Automata*. In Henning Fernau and Klaus Jansen, editors, Fundamentals of Computation Theory - 24th International Symposium, FCT 2023, Trier, Germany, September 18-21, 2023, Proceedings, volume 14292 of *Lecture Notes in Computer Science*, pages 16–31. Springer, 2023.
- [Al Serhali & Niehren 2024] Antonio Al Serhali and Joachim Niehren. *Complete Subhedge Projection for Stepwise Hedge Automata*. Algorithms, vol. 17, no. 8, 2024.
- [Alur & Madhusudan 2004] Rajeev Alur and P. Madhusudan. *Visibly pushdown languages*. In László Babai, editor, Proceedings of the 36th Annual ACM

- Symposium on Theory of Computing, Chicago, IL, USA, June 13-16, 2004, pages 202–211. ACM, 2004.
- [Alur 2007] Rajeev Alur. *Marrying Words and Trees*. In 26th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, pages 233–242. ACM-Press, 2007.
- [Angles *et al.* 2017] Renzo Angles, Marcelo Arenas, Pablo Barceló, Aidan Hogan, Juan Reutter and Domagoj Vrgoč. *Foundations of Modern Query Languages for Graph Databases*. ACM Comput. Surv., vol. 50, no. 5, September 2017.
- [Arenas *et al.* 2014] Marcelo Arenas, Pablo Barceló, Leonid Libkin and Filip Murlak. *Foundations of data exchange*. Cambridge University Press, 2014.
- [Arenas *et al.* 2022] Marcelo Arenas, Pablo Barceló, Leonid Libkin, Wim Martens and Andreas Pieris. *Database theory*. Open source at <https://github.com/pdm-book/community>, 2022.
- [Barcelo & Libkin 2005] P. Barcelo and L. Libkin. *Temporal logics over unranked trees*. In 20th Annual IEEE Symposium on Logic in Computer Science (LICS'05), pages 31–40, 2005.
- [Barloy *et al.* 2021] Corentin Barloy, Filip Murlak and Charles Paperman. *Stackless Processing of Streamed Trees*. In Proceedings of the 40th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS'21, page 109–125, New York, NY, USA, 2021. Association for Computing Machinery.
- [Benedikt & Jeffrey 2007] Michael Benedikt and Alan Jeffrey. *Efficient and Expressive Tree Filters*. In Foundations of Software Technology and Theoretical Computer Science, volume 4855 of *Lecture Notes in Computer Science*, pages 461–472. Springer Verlag, 2007.
- [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.
- [Bergougnoux *et al.* 2019] Benjamin Bergougnoux, Florent Capelli and Mamadou Moustapha Kanté. *Counting minimal transversals of β -acyclic hypergraphs*. J. Comput. Syst. Sci., vol. 101, pages 21–30, 2019.

- [Blackburn *et al.* 2001] Patrick Blackburn, Maarten de Rijke and Yde Venema. *Modal logic*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2001.
- [Bojanczyk & Walukiewicz 2008] Mikolaj Bojanczyk and Igor Walukiewicz. *Forest algebras*. In Jörg Flum, Erich Grädel and Thomas Wilke, editors, *Logic and Automata: History and Perspectives* [in Honor of Wolfgang Thomas], volume 2 of *Texts in Logic and Games*, pages 107–132. Amsterdam University Press, 2008.
- [Carme *et al.* 2004] Julien Carme, Joachim Niehren and Marc Tommasi. *Querying Unranked Trees with Stepwise Tree Automata*. In Vincent van Oostrom, editor, *Rewriting Techniques and Applications*, 15th International Conference, RTA 2004, Aachen, Germany, June 3-5, 2004, *Proceedings*, volume 3091 of *Lecture Notes in Computer Science*, pages 105–118. Springer, 2004.
- [Carney *et al.* 2002] Don Carney, Uunefinedur 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.
- [Ceri *et al.* 2012] S. Ceri, G. Gottlob and L. Tanca. *Logic programming and databases*. *Surveys in Computer Science*. Springer Berlin Heidelberg, 2012.
- [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.
- [Cleaveland & Steffen 1991] Rance Cleaveland and Bernhard Steffen. *A Linear-Time Model-Checking Algorithm for the Alternation-Free Modal Mu-Calculus*. In Kim Guldstrand Larsen and Arne Skou, editors, *Computer Aided Verification*, 3rd International Workshop, CAV '91, Aalborg, Denmark, July, 1-4, 1991, *Proceedings*, volume 575 of *Lecture Notes in Computer Science*, pages 48–58. Springer, 1991.
- [Comon *et al.* 2002] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison and M. Tommasi. *Tree Automata Techniques and Applications*. Avail-

able on:

verb!http://www.grappa.univ-lille3.fr/tata!

- [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>.
- [Cook 1971] Stephen A. Cook. *The complexity of theorem-proving procedures*. In Proceedings of the Third Annual ACM Symposium on Theory of Computing, STOC '71, page 151–158, New York, NY, USA, 1971. Association for Computing Machinery.
- [Cugola & Margara 2012] Gianpaolo Cugola and Alessandro Margara. *Processing flows of information: From data stream to complex event processing*. ACM Comput. Surv., vol. 44, no. 3, June 2012.
- [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.
- [Dietzfelbinger *et al.* 1994] Martin Dietzfelbinger, Anna R. Karlin, Kurt Mehlhorn, Friedhelm Meyer auf der Heide, Hans Rohnert and Robert Endre Tarjan. *Dynamic Perfect Hashing: Upper and Lower Bounds*. SIAM J. Comput., vol. 23, no. 4, pages 738–761, 1994.
- [Durand & Grandjean 2004] Arnaud Durand and Etienne Grandjean. *The Complexity of Acyclic Conjunctive Queries Revisited*. Technical report, Université de Paris 7, 2004.
- [Fagin *et al.* 2003] Ronald Fagin, Joseph Halpern, Yoram Moses and Moshe Vardi. *Reasoning about knowledge*. MIT Press, 01 2003.
- [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.
- [Florescu *et al.* 1998] Daniela Florescu, Alon Y. Levy and Dan Suciu. *Query Containment for Conjunctive Queries with Regular Expressions*. In Alberto O. Mendelzon and Jan Paredaens, editors, Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database

- Systems, June 1-3, 1998, Seattle, Washington, USA, pages 139–148. ACM Press, 1998.
- [Franceschet 2005a] Massimo Franceschet. *XPathMark: An XPath Benchmark for the XMark Generated Data*. In Stéphane Bressan, Stefano Ceri, Ela Hunt, Zachary G. Ives, Zohra Bellahsene, Michael Rys and Rainer Unland, editors, Database and XML Technologies, pages 129–143, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.
- [Franceschet 2005b] Massimo Franceschet. *XPathMark Performance Test*. <https://users.dimi.uniud.it/~massimo.franceschet/xpathmark/PTbench.html>. Accessed: 2024-10-15.
- [Frisch 2004] Alain Frisch. *Regular Tree Language Recognition with Static Information*. In Exploring New Frontiers of Theoretical Informatics, IFIP 18th World Computer Congress, TCS 3rd International Conference on Theoretical Computer Science, pages 661–674, 2004.
- [Gauwin & Niehren 2011] Olivier Gauwin and Joachim Niehren. *Streamable Fragments of Forward XPath*. In 16th International Conference on Implementation and Application of Automata, pages 3–15, Blois, France, 2011. Long version: <http://www.grappa.univ-lille3.fr/niehren/Papers/streamability/0.pdf>.
- [Gauwin et al. 2008] Olivier Gauwin, Joachim Niehren and Yves Roos. *Streaming Tree Automata*. Information Processing Letters, vol. 109, no. 1, pages 13–17, 2008.
- [Gauwin et al. 2009a] Olivier Gauwin, Joachim Niehren and Sophie Tison. *Bounded Delay and Concurrency for Earliest Query Answering*. In 3rd International Conference on Language and Automata Theory and Applications, volume 5457 of *Lecture Notes in Computer Science*, pages 350–361. Springer Verlag, 2009.
- [Gauwin et al. 2009b] 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.
- [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.

- [Gauwin 2009] Olivier Gauwin. *Streaming Tree Automata and XPath*. PhD thesis, Université Lille 1, 2009.
- [Gécseg & Steinby 1984] F. Gécseg and M. Steinby. *Tree automata*. Akadémiai Kiadó, Budapest, 1984.
- [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, 2006.
- [Gienieczko *et al.* 2024] Mateusz Gienieczko, Filip Murlak and Charles Paperman. *Supporting Descendants in SIMD-Accelerated JSONPath*. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4, ASPLOS '23*, page 338–361, New York, NY, USA, 2024. Association for Computing Machinery.
- [Gold 1967] E. Mark Gold. *Language Identification in the Limit*. *Inf. Control.*, vol. 10, no. 5, pages 447–474, 1967.
- [Gottlob *et al.* 2003] Georg Gottlob, Christoph Koch and Reinhard Pichler. *The complexity of XPath query evaluation*. In *Proceedings of the Twenty-Second ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS '03*, page 179–190, New York, NY, USA, 2003. Association for Computing Machinery.
- [Green *et al.* 2004] Todd J. Green, Ashish Gupta, Gerome Miklau, Makoto Onizuka and Dan Suciu. *Processing XML streams with deterministic automata and stream indexes*. *ACM Trans. Database Syst.*, vol. 29, no. 4, pages 752–788, 2004.
- [Grež *et al.* 2019] Alejandro Grež, Cristian Riveros and Martín Ugarte. *A Formal Framework for Complex Event Processing*. In Pablo Barcelo and Marco Calautti, editors, *22nd International Conference on Database Theory (ICDT 2019)*, volume 127 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 5:1–5:18, Dagstuhl, Germany, 2019. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.
- [Hosoya & Pierce 2003] Haruo Hosoya and Benjamin C. Pierce. *XDuce: A Statically Typed XML Processing Language*. *ACM Transactions on Internet Technology*, vol. 3, no. 2, pages 117–148, 2003.
- [Hustadt 2001] Ullrich Hustadt. *Temporal Logic: Mathematical Foundations and Computational Aspects, Volume 2*, Dov M. Gabbay, Mark A. Reynolds, and

- Marcelo Finger. *Journal of Logic, Language and Information*, vol. 10, pages 406–410, 06 2001.
- [Immerman & Kozen 1989] Neil Immerman and Dexter Kozen. *Definability with Bounded Number of Bound Variables*. *Inf. Comput.*, vol. 83, no. 2, pages 121–139, 1989.
- [Kay 2004] Michael Kay. *The saxon XSLT and XQuery processor*. <https://www.saxonica.com>.
- [Kay 2010] Michael Kay. *A Streaming XSLT Processor*. In *Balisage: The Markup Conference 2010*. Balisage Series on Markup Technologies, volume 5, 2010.
- [Kay 2011] M. Kay. *XSLT 2.0 and XPath 2.0 programmer’s reference*. Wiley, 2011.
- [Koch *et al.* 2004] Christoph Koch, Stefanie Scherzinger, Nicole Schweikardt and Bernhard Stegmaier. *FluXQuery: An Optimizing XQuery Processor for Streaming XML Data*. In *Twenty-first ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 1309–1312, 2004.
- [Kripke 1959] Saul Kripke. *A Completeness Theorem in Modal Logic*. *J. Symb. Log.*, vol. 24, no. 1, pages 1–14, 1959.
- [Kumar *et al.* 2007] Viraj Kumar, P. Madhusudan and Mahesh Viswanathan. *Visibly pushdown automata for streaming XML*. In *16th international conference on World Wide Web*, pages 1053–1062. ACM-Press, 2007.
- [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.
- [Lick & Schmitz 2022] Anthony Lick and Sylvain Schmitz. *XPath Benchmark*. <https://archive.softwareheritage.org/browse/directory/1ea68cf5bb3f9f3f2fe8c7995f1802ebadf17fb5>. Accessed: 2024-10-15.
- [Lick 2019] Anthony Lick. *Logique de requêtes à la XPath : systèmes de preuve et pertinence pratique*. Theses, Université Paris-Saclay, 2019.
- [Maneth & Nguyen 2010] Sebastian Maneth and Kim Nguyen. *XPath Whole Query Optimization*. *VLPB Journal*, vol. 3, no. 1, pages 882–893, 2010.

- [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.
- [Marian & Siméon 2003] Amélie Marian and Jérôme Siméon. *Projecting XML Documents*. In VLDB, pages 213–224, 2003.
- [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.
- [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.
- [McNaughton & Papert 1971] Robert McNaughton and Seymour A. Papert. Counter-free automata (mit research monograph no. 65). The MIT Press, 1971.
- [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 and Ariel Fuxman, editors, Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012, pages 253–264. ACM, 2012.
- [Muñoz & Riveros 2022a] Martin Muñoz and Cristian Riveros. *Streaming Enumeration on Nested Documents*. In Dan Olteanu and Nils Vortmeier, editors, 25th International Conference on Database Theory, ICDT 2022, Marchs 29 to April 1, 2022, Edinburgh, UK (Virtual Conference), volume 220 of *LIPIcs*, pages 19:1–19:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022.
- [Muñoz & Riveros 2022b] Martin Muñoz and Cristian Riveros. *Streaming Enumeration on Nested Documents*. In Dan Olteanu and Nils Vortmeier, editors, 25th International Conference on Database Theory, ICDT 2022, March 29 to April 1, 2022, Edinburgh, UK (Virtual Conference), volume 220 of *LIPIcs*, pages 19:1–19:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022.

- [Murata 2000] M. Murata. *Hedge Automata: a Formal Model for XML Schemata*. Web page.
- [Murlak *et al.* 2016] Filip Murlak, Charles Paperman and Michal Pilipczuk. *Schema Validation via Streaming Circuits*. In Tova Milo and Wang-Chiew Tan, editors, Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2016, San Francisco, CA, USA, June 26 - July 01, 2016, pages 237–249. ACM, 2016.
- [Neumann & Seidl 1998] Andreas Neumann and Helmut Seidl. *Locating Matches of Tree Patterns in Forests*. In 18-th Conference on Foundations of Software Technology and Theoretical Computer Science, 1998.
- [Niehren & Sakho 2021] Joachim Niehren and Momar Sakho. *Determinization and Minimization of Automata for Nested Words Revisited*. Algorithms, vol. 14, no. 3, page 68, 2021.
- [Niehren *et al.* 2022a] Joachim Niehren, Momar Sakho and Antonio Al Serhali. *Schema-Based Automata Determinization*. In Pierre Ganty and Dario Della Monica, editors, Proceedings of the 13th International Symposium on Games, Automata, Logics and Formal Verification, GandALF 2022, Madrid, Spain, September 21-23, 2022, volume 370 of EPTCS, pages 49–65, 2022.
- [Niehren *et al.* 2022b] Joachim Niehren, Sylvain Salvati and Rustam Azimov. *Jumping Evaluation of Nested Regular Path Queries*. In ICLP 2022 - 38th International Conference on Logic Programming, Haifa, Israel, July 2022.
- [Niehren 2024] Joachim Niehren. *Stepwise Hedge Automata are exponentially more succinct than Forest Automata*. in preperation.
- [Okhotin & Salomaa 2014] Alexander Okhotin and Kai Salomaa. *Complexity of input-driven pushdown automata*. SIGACT News, vol. 45, no. 2, pages 47–67, 2014.
- [Olteanu 2007] Dan Olteanu. *SPEX: Streamed and Progressive Evaluation of XPath*. IEEE Trans. on Know. Data Eng., vol. 19, no. 7, pages 934–949, 2007.
- [Pair & Quéré 1968] Claude Pair and Alain Quéré. *Définition et étude des bilangages réguliers*. Information and Control, vol. 13, no. 6, pages 565–593, 1968.
- [Pitcher 2005] Corin Pitcher. *Visibly Pushdown Expression Effects for XML Stream Processing*. In PlanX, 2005.

- [Sakho *et al.* 2017] Momar Sakho, Iovka Boneva and Joachim Niehren. *Complexity of Certain Query Answering on Hyperstreams*. In BDA 2017 - 33ème conférence sur la “ Gestion de Données - Principes, Technologies et Applications ”, Nancy, France, 2017.
- [Sakho 2020] Momar Sakho. *Certain Query Answering on Hyperstreams*. Theses, Université de Lille ; Inria, 2020.
- [Schmidt *et al.* 2002] Albrecht Schmidt, Florian Waas, Martin Kersten, Michael J. Carey, Ioana Manolescu and Ralph Busse. *XMark: a benchmark for XML data management*. In Proceedings of the 28th International Conference on Very Large Data Bases, VLDB '02, page 974–985. VLDB Endowment, 2002.
- [Schwentick & Zeume 2012] Thomas Schwentick and Thomas Zeume. *Two-Variable Logic with Two Order Relations*. Logical Methods in Computer Science, vol. Volume 8, Issue 1, March 2012.
- [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, 2016.
- [Sebastian 2016] Tom Sebastian. *Evaluation of XPath Queries on XML Streams with Networks of Early Nested Word Automata*. Theses, Université Lille 1, 2016.
- [Smullyan 2012] R.R. Smullyan. *First-order logic*. Ergebnisse der Mathematik und ihrer Grenzgebiete. 2. Folge. Springer Berlin Heidelberg, 2012.
- [Stockmeyer 1974] Larry J. Stockmeyer. *The complexity of decision problems in automata theory and logic*. PhD thesis, Massachusetts Institute of Technology, USA, 1974.
- [Straubing 1994] H. Straubing. *Finite automata, formal logic, and circuit complexity*. Progress in Computer Science and Applied Series. Birkhäuser, 1994.
- [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.
- [Vardi 1982] Moshe Vardi. *The Complexity of Relational Query Languages (Extended Abstract)*. In STOC, pages 137–146, 01 1982.

- [Veanes *et al.* 2012] Margus Veanes, Pieter Hooimeijer, Benjamin Livshits, David Molnar and Nikolaj Bjorner. *Symbolic finite state transducers: algorithms and applications*. In Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '12, page 137–150, New York, NY, USA, 2012. Association for Computing Machinery.
- [von Braunmühl & Verbeek 1985] Burchard von Braunmühl and Rutger Verbeek. *Input Driven Languages are Recognized in $\log n$ Space*. In Marek Karplinski and Jan van Leeuwen, editors, Topics in the Theory of Computation, volume 102 of *North-Holland Mathematics Studies*, pages 1 – 19. North-Holland, 1985.
- [Yannakakis 1981] Mihalis Yannakakis. *Algorithms for Acyclic Database Schemes*. In Proceeding of VLDB, pages 82–94, 1981.
- [Yu 1997] Sheng Yu. *Regular Languages*. In Grzegorz Rozenberg and Arto Salomaa, editors, Handbook of Formal Languages, Volume 1: Word, Language, Grammar, pages 41–110. Springer, 1997.