On Tarski's Relation Algebra

QUERYING TREES AND CHAINS

and

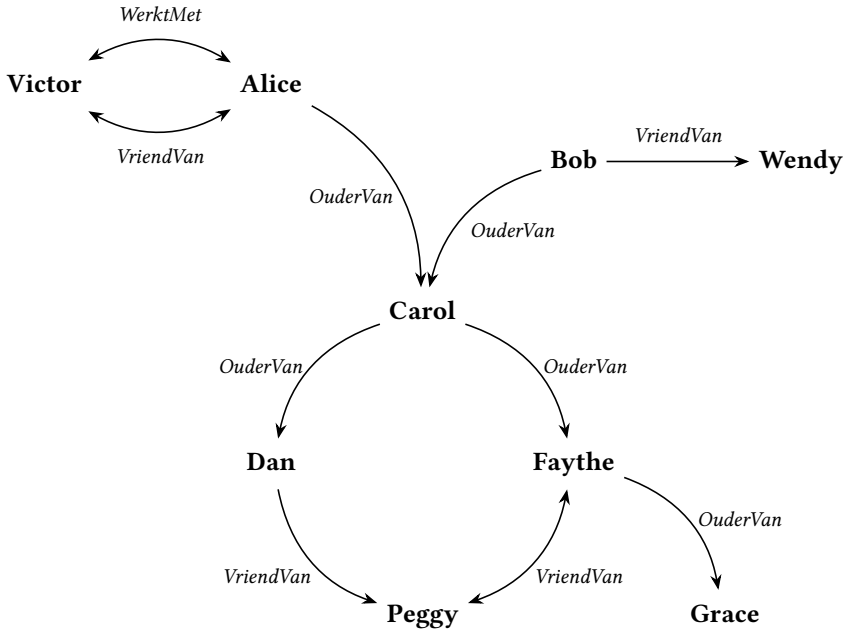
THE SEMI-JOIN ALGEBRA

Jelle Hellings

http://jhellings.nl

Samenvatting

In het graaf-gestructureerde gegevensmodel staan data-elementen (de knopen) en de onderlinge relaties tussen deze data-elementen (de bogen) centraal. Door zijn eenvoud is het graaf-gestructureerde gegevensmodel breed toepasbaar, en toepassingen zijn onder meer te vinden in XML-data, RDF-data, sociale netwerken en gen- en proteïnenetwerken. Figuur 1 geeft een voorbeeld van een klein sociaal netwerk gemodelleerd als een graaf.



Figuur 1: Een typisch voorbeeld van graaf-gestructureerde gegevens: een sociaal netwerk.

De graaf in Figuur 1 representeert de relaties *OuderVan*, *VriendVan* en *WerktMet*, maar dit zijn niet de enige aanwezige relaties. Zo kunnen we *OuderVan* gebruiken om de *GrootouderVan*-relatie samen te stellen en kunnen we *VriendVan* en *WerktMet* gebruiken om de *WerkVriendVan*-relatie samen te stellen. Deze indirecte relaties, en vele andere, kunnen eenvoudig uitgedrukt worden door middel van een ondervragingstaal die gespecialiseerd is in het uitdrukken van graafvragen.

Vele praktische ondervragingstalen voor graaf-gestructureerde gegevens zijn gebaseerd

op (delen van) Tarski's relatiealgebra uitgebreid met de Kleene-ster operatie. Voorbeelden zijn onder andere XPath, SPARQL, de RPQs en GXPath. Vanwege deze centrale rol van (fragmenten van) de relatiealgebra, hebben we in ons werk twee aspecten van de relatiealgebra diepgaand bestudeerd:

1. Vele natuurlijke en artificiële bronnen van graaf-gestructureerde gegevens bevatten hiërarchische relaties die via een boomstructuur beschreven kunnen worden. Voorbeelden zijn taxonomieën, bedrijfsstructuren, bestand- en directorystructuren, XML-data, en JSON-data. Het is dan ook verwonderlijk dat er nog geen systematische studie is geweest naar de uitdrukkingskracht van (fragmenten van) de relatiealgebra—welke graafvragen wel of niet uitgedrukt kunnen worden—bij het ondervragen van boomstructuren.

   We hebben zo'n systematische studie naar de uitdrukkingskracht van de relatiealgebra op boom- en ketenstructuren ondernomen. Hierbij hebben we vooral de uitdrukkingskracht vergeleken van verschillende fragmenten van de relatiealgebra, dit enerzijds om een dieper inzicht te krijgen in de rol die iedere operatie in de relatiealgebra vervult, en anderzijds om een dieper inzicht te krijgen in het bevragen van boom- en ketenstructuren. In onze studie hebben we vijf eigenschappen gevonden waarmee we de uitdrukkingskracht van elk fragment van de relatiealgebra in grote lijnen kunnen karakteriseren. Die zijn: (1) of het fragment graafvragen kan uitdrukken waarmee het ketens, binaire bomen en ternaire bomen van elkaar kan onderscheiden; (2) of het fragment alleen neergaande uitdrukkingen bevat (die boomstructuren neergaand bevragen via ouder-kind relaties); (3) of het fragment alleen lokale uitdrukkingen bevat (die boomstructuren neergaand bevragen via een vast aantal boog-stappen); (4) of het fragment negatie kan uitdrukken; en (5) of het fragment de Kleene-ster operatie kan uitdrukken.

   Voor fragmenten die voldoen aan (2) of (3), die respectievelijk de neergaande of de lokale fragmenten worden genoemd, hebben we bovendien ook alle onderlinge verbanden tussen de uitdrukkingskracht van elk van deze fragmenten kunnen vaststellen en bewijzen. De belangrijkste bevinding daarbij is dat twee operaties, de doorsnede en het verschil, of overbodig zijn en geen uitdrukkingskracht toevoegen of maar zeer beperkt uitdrukkingskracht toevoegen. Tevens hebben we in veel andere gevallen kunnen aantonen dat fragmenten verschillende uitdrukkingskracht hebben. Om deze gevallen te bestuderen hebben we verscheidende nieuwe bewijstechnieken ontwikkeld.

2. Centraal in de relatiealgebra staat de compositie, die gebruikt wordt om indirecte relaties in termen van paden in graafstructuren uit te drukken. Zo kunnen we compositie gebruiken om alle grootouders te vinden in de graaf van Figuur 1. De standaard manier om dit te doen is eerst de *GrootouderVan*-relatie te berekenen: dit doen we door compositie te gebruiken om de *OuderVan*-relatie met zichzelf te koppelen. Hiermee worden alle grootouders aan hun kleinkinderen gekoppeld en dit kunnen al snel zeer veel grootouder-kleindkind-paren worden. Vervolgens gooien we alle informatie over de gevonden kleinkinderen weg. Dit is echter zeer inefficiënt: als we alleen willen weten wie grootouder is, dan is het berekenen van alle kleinkinderen geheel overbodig (we hoeven alleen te berekenen of er een kleinkind is, niet wie de kleinkinderen zijn). Met het oog op efficiënte beantwoording van graafvragen die uitgedrukt zijn in de relatiealgebra wensen we dan ook het gebruik van compositie te elimineren in de gevallen waarin de volledige kracht van compositie overbodig is.

Om overbodig gebruik van compositie te kunnen elimineren, beschouwen we een alternatieve ondervragingstaal: de semi-join-algebra. Deze verkrijgen we door de dure compositie en Kleene-ster operaties te vervangen door semi-join operaties en een simpele vorm van fixpunt-herhaling. Waar we compositie gebruiken om relaties aan elkaar te koppelen, controleren we met de semi-join alleen of we zo'n koppeling kunnen maken. Daarmee is het resultaat van de semi-join veel eenvoudiger te berekenen. Alhoewel resultaten voor graafvragen uitgedrukt in de semi-join-algebra eenvoudiger te berekenen zijn dan de resultaten voor graafvragen uitgedrukt in de relatiealgebra, is de semi-join-algebra in de praktijk minder gebruiksvriendelijk (omdat het moeilijker is graafvragen in de semi-join-algebra uit te drukken).

Daarom bestuderen we hoe we automatisch graafvragen, uitgedrukt in de relatiealgebra, geheel of gedeeltelijk kunnen omzetten in eenvoudiger-te-beantwoorden graafvragen in de semi-join-algebra. Ons belangrijkste resultaat is dat voor elk fragment van de relatiealgebra waarin het gebruik van doorsnede en verschil beperkt is tot boogrelaties er een fragment van de semi-join-algebra bestaat dat exact dezelfde graafvragen kan uitdrukken (met betrekking tot graafvragen die beantwoord kunnen worden met het teruggeven van een verzameling knopen). Om dit resultaat praktisch relevant te maken, geven we tevens een constructieve methode om (delen van) graafvragen in de relatiealgebra te herschrijven naar graafvragen in de semi-join-algebra. Niet alleen kunnen we garanderen dat deze constructieve methode enkel een beperkte toename veroorzaakt in het aantal berekenstappen nodig voor het beantwoorden van een graafvraag, maar bovendien kan de methode ook meerdere van deze berekenstappen significant eenvoudiger maken. Verder bestuderen we hoe de kosten van het gebruik van andere operaties in de relatiealgebra verlaagd kunnen worden.

Tezamen geven deze twee studies een zeer gedetailleerd beeld van de uitdrukkingskracht van fragmenten van de relatiealgebra. Tevens geven onze resultaten verscheidene aanknopingspunten voor het ontwikkelen van nieuwe technieken om efficiënt complexe graafvragen te beantwoorden.

Abstract

Many practical query languages for graph data are based on fragments of Tarski's relation algebra which, optionally, is augmented with the Kleene-star operator. Examples include XPath, SPARQL, the RPQs, and GXPath. Because of this central role of (fragments of) the relation algebra, we study two aspects in more detail.

1. Many natural and artificial sources of graph data contain hierarchical relations that can be modeled using tree structures, and these tree structures are frequently queried using a query language based on a fragment of the relation algebra. Surprisingly, a systematic study of the relative expressive power of relation algebra fragments on trees has not yet been undertaken. To address this, we start with a basic relation algebra fragment which only allows composition and union. We then study how the expressive power of the query language changes if we add diversity, converse, projections, coprojections, intersections, difference, and/or Kleene-star, this both for path queries and Boolean queries. We do this both on labeled and unlabeled structures.

Our main contribution is the identification of properties that can be used to categorize relation algebra fragments according to their expressive power. These are: (1) whether the fragment can distinguish *branching* structures; (2) whether the fragment is *downward*; (3) whether the fragment is *local*; (4) whether the fragment has *negation*; and (5) whether the fragment can express the *Kleene-star*. For the downward and local fragments, we did not only introduced the above characterization, but we also established and proved all relationships in the relative expressive power of these fragments. Central in this are the roles of intersection and difference, which are either redundant or add only limited expressive power to each downward and local fragment we consider.

2. Many graph query languages rely on composition to navigate graphs and select nodes of interest, even though evaluating compositions of relations can be very costly. Often, this need for composition can be reduced by rewriting such queries towards queries that use semi-joins instead. In this way, the evaluation cost can be significantly reduced. We study techniques to recognize and apply such rewritings automatically. Concretely, we study the relationship between the expressive power of the relation algebra, that relies heavily on composition and Kleene-star for graph navigation, and the semi-join algebras, that replaces the expensive composition and Kleene-star operators in favor of the semi-join operators and a simple form of fixpoint iteration.

Our main result is that each fragment of the relation algebra where intersection and/or difference is used only on edges (and not on complex compositions) is equivalent to a fragment of the semi-join algebra for queries that evaluate to a set of nodes. For practical relevance, we exhibit a constructive method for rewriting relation algebra queries to semi-join algebra

queries. We prove that this method leads to only a well-bounded increase in the number of steps needed to evaluate the rewritten queries, will never increase the cost of evaluating individual steps, and can significantly reduce the cost of evaluating individual steps. We also study how the cost of other expensive operators in the relation algebra can be reduced.

Combined, these two studies give a detailed picture of the expressive power of the fragments of the relation algebra. Moreover, our results provide several opportunities for the development of new techniques for the efficient evaluation of graph queries.

Preface

This doctoral dissertation titled "On Tarski's Relation Algebra: Querying trees and chains, and the semi-join algebra" is the culmination of my research at Hasselt University on the topic of the expressive power of graph query languages based on fragments of Tarski's relation algebra.

Organization and scope of this dissertation

The presentation of my results is in four distinct parts. First, Part I serves as an introduction to the graph data model, to graph query languages based on Tarski's relation algebra, and to related terminology and notations used throughout this work. Next, Parts II and III present the results of our study on two distinct aspects of the relation algebra. Finally, Part IV provides a short unifying conclusion on our research.

More specifically, we study in Part II the relative expressive power of fragments of the relation algebra with respect to querying tree structures and chain structures. This Part is based on the following three papers:

1. Jelle Hellings, Yuqing Wu, Marc Gyssens, and Dirk Van Gucht. The power of Tarski's relation algebra on trees. In *Proceedings of the 10th International Symposium on Foundations of Information and Knowledge Systems*, 2018.

2. Jelle Hellings, Marc Gyssens, Yuqing Wu, Dirk Van Gucht, Jan Van den Bussche, Stijn Vansummeren, and George H. L. Fletcher. Relative expressive power of downward fragments of navigational query languages on trees and chains. In *Proceedings of the 15th Symposium on Database Programming Languages*, pages 59–68, 2015.

3. Jelle Hellings, Marc Gyssens, Yuqing Wu, Dirk Van Gucht, Jan Van den Bussche, Stijn Vansummeren, and George H. L. Fletcher. Comparing downward fragments of the relational calculus with transitive closure on trees. Technical report, Hasselt University, 2018. URL: `https://arxiv.org/abs/1803.01390`.

The latter paper is being prepared for submission to a journal.

In Part III, we study the relationships between the expressive power of the relation algebra and the semi-join algebra. The latter is obtained by replacing composition and Kleene-star in the relation algebra by semi-joins and a form of fixpoint iteration. We also study how these relationships can be used to optimize graph query evaluation. This Part is based on the following paper:

4. Jelle Hellings, Catherine L. Pilachowski, Dirk Van Gucht, Marc Gyssens, and Yuqing Wu. From relation algebra to semi-join algebra: An approach for graph query optimization.

In *Proceedings of the 16th International Symposium on Database Programming Languages*, pages 5:1–5:10, 2017.

Scope of my research

In this dissertation, I only present my research on Tarski's relation algebra. My research focus was not only on the relation algebra, however, but also on various other aspects of data management. Below is an overview of my published research results that have not been integrated in this dissertation.

For my Master thesis project I developed efficient *external-memory bisimulation partitioning algorithms*. These algorithms can be used for the construction of bisimulation-based indices of very big directed acyclic graphs. I have also developed specializations of these algorithms that can deal with large XML documents [45].[1] During my first year at Hasselt University, we worked on publishing these results, resulting in the following paper:

5. Jelle Hellings, George H.L. Fletcher, and Herman Haverkort. Efficient external-memory bisimulation on dags. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 553–564, 2012.

We have introduced the graph query language *walk logic* [54]. This novel graph query language makes it easy to query for path-based structural graph properties. Examples of such queries include 'Does the graph have a Hamiltonian cycle?' and 'Does the graph have an Eulerian tour?'. We not only introduced and formalized walk logic; we also studied the expressive power of walk logic, and the relationships between walk logic and other graph query languages. This work resulted in the following paper:

6. Jelle Hellings, Bart Kuijpers, Jan Van den Bussche, and Xiaowang Zhang. Walk logic as a framework for path query languages on graph databases. In *Proceedings of the 16th International Conference on Database Theory*, pages 117–128, 2013.

We have worked on data dependencies in the setting of semi-structured data. Specifically, we have provided an *axiomatization of functional constraints* and *constant constraints* defined on patterns in semi-structured data. This work resolved an open problem stated in Akhtar et al. [2] and resulted in the following two papers:

7. Jelle Hellings, Marc Gyssens, Jan Paredaens, and Yuqing Wu. Implication and axiomatization of functional constraints on patterns with an application to the RDF data model. In *Proceedings of the 8th International Symposium on Foundations of Information and Knowledge Systems*, pages 250–269. Springer International Publishing, 2014.

8. Jelle Hellings, Marc Gyssens, Jan Paredaens, and Yuqing Wu. Implication and axiomatization of functional and constant constraints. *Annals of Mathematics and Artificial Intelligence*, 76(3):251–279, 2016.

We studied the notion of *counting-only queries* on collections of sets, queries which can be answered by only looking at the number of sets groups of objects occur in. As it turns out, many practical queries are counting-only and can be expressed in simple well-behaved counting-only query languages. This work resulted in the following two papers:

[1] We refer to `http://jhellings.nl/projects/exbisim/` for more information on the external memory bisimulation partitioning algorithms we developed.

9. Jelle Hellings, Marc Gyssens, Dirk Van Gucht, and Yuqing Wu. First-order definable counting-only queries. In *Proceedings of the 10th International Symposium on Foundations of Information and Knowledge Systems*, 2018.

10. Marc Gyssens, Jelle Hellings, Jan Paredaens, Dirk Van Gucht, Jef Wijsen, and Yuqing Wu. Calculi for symmetric queries.

The latter paper is being prepared for submission to a journal.

We introduced the *context-free graph queries* as a generalization of the regular path queries, and studied this graph query language in detail. This work resulted in a single paper:

11. Jelle Hellings. Conjunctive context-free path queries. In *Proceedings of the 17th International Conference on Database Theory*, pages 119–130, 2014.

We have also worked on answering context-free graph queries with *short paths* instead of node-pairs. These short paths give insight in the derivation of the traditional node-pair query answer. We not only developed an efficient algorithm to compute the shortest paths, but we have also analyzes the worst-case length of these paths (in terms of the size of the graph and the query). This work has not yet been published, but preliminary notes have been made available:

12. Jelle Hellings. Path results for context-free grammar queries on graphs. Technical report, Hasselt University, 2016. URL: `https://arxiv.org/abs/1502.02242`.

Finally, we worked on high-performance cache-friendly *temporal join algorithms* that can deal with skewed data (in which only few records in the joined relations are part in the join result). To support the operations needed in such temporal join algorithms, we developed novel append-only temporal index structures. We are currently preparing our results for submission to a peer-reviewed conference:

13. Jelle Hellings and Yuqing Wu. Stab-forests: Dynamic data structures for efficient temporal query processing.

Acknowledgments

I would not have become a researcher working on database management in general and database theory in particular without the guidance of George H. L. Fletcher, Jan Van den Bussche, and Marc Gyssens. I thank George H. L. Fletcher for supervising my Master thesis project, as this work eventually pushed me into pursuing doctoral research. Next, I thank Jan Van den Bussche for welcoming me at Hasselt University, for introducing me to database theory research, and for supporting all my research endeavors. Finally, I thank Marc Gyssens, my principal doctoral research adviser. The support of my adviser has been invaluable for my growth as a person and as a researcher. Not only did Marc Gyssens support my research endeavors, but I could also rely on him for his patience and general support the last six years.

I would also like to thank my adviser for setting up and supporting my research collaborations with Dirk Van Gucht and Yuqing Wu, collaborations that have played significant roles in my research over the years, especially with respect to the relation algebra work presented in this dissertation. I would like to thank Dirk Van Gucht and Yuqing Wu for hosting my research visits to Indiana University, Bloomington. Each of these visits has been inspiring

and these visits have led to many research results and to surprising new directions in my work. I would also like to thank Yuqing Wu for inviting me for several research visits to Pomona College in California during the last summers. Each of these visits felt like vacation, despite the sheer amount of work we did.

I would like to thank the members of my doctoral committee, my adviser Marc Gyssens, my co-advisers Jan Van den Bussche and Bart Kuijpers, and Frank Neven for their general support over the years and for the helpful remarks on earlier drafts of this dissertation. I would also like to thank the other members of the Jury for their time and their helpful remarks on this dissertation. Finally, I would like to thank all other people with whom I had fruitful research collaborations over the years, namely Herman Haverkort, Jan Paredaens, Catherine L. Pilachowski, Stijn Vansummeren, Jef Wijsen, and Xiaowang Zhang.

Overall, I have enjoyed my time at Hasselt University, and this was mostly due to the great atmosphere in the Databases and Theoretical Computer Science Research Group. The professors should be extremely proud for having created such an open, friendly, and supportive environment. I would not only like to thank the professors of the research group, but also my other direct coworkers. In particular, I would like to thank Jonny Daenen and Dimitri Surinx, with whom I taught many courses, for providing so much fun during teaching. I would also like to thank all other coworkers, in particular, Robert Brijder, Bas Ketsman, and Brecht Vandevoort, with whom I enjoyed my time during lunches, had energizing walks around the campus, attended conferences, and with whom I shared many interesting ideas.

Contents

List of Figures

List of Tables

Part I

On Tarski's Relation Algebra

A GENERAL INTRODUCTION

CHAPTER 1

Graph querying and Tarski's relation algebra

The graph data model, in which data is represented by labeled binary relations, is a versatile and natural data model for representing RDF data, social networks, gene and protein networks, and other types of data. Figure 1.1 shows a graph representing a fragment of a social network as an example of such data. In this graph, the nodes represent objects corresponding to persons and the labeled edges represent various semantic relationships between these persons. The graph represents the binary relationships *ParentOf*, *FriendOf*, and *WorksWith*, shown in Table 1.1. Indirectly, this graph also express many other binary relationships, including *GrandparentOf*, *AncestorOf*, *ChildOf*, and *WorkFriend* shown in Table 1.2.

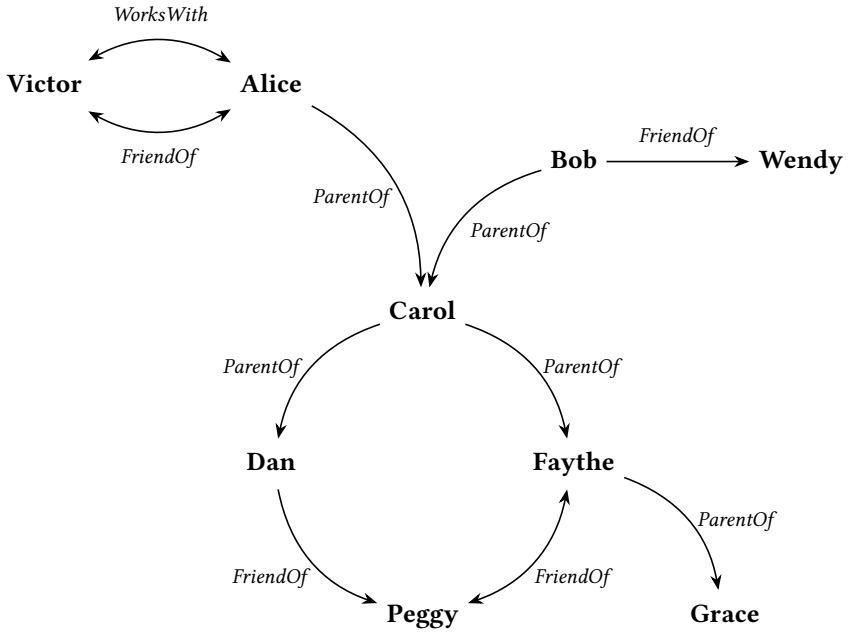


Figure 1.1: An example of social network graph data.

Table 1.1: Binary relations representing the graph of Figure 1.1.

ParentOf	
Alice	Carol
Bob	Carol
Carol	Dan
Carol	Faythe
Faythe	Grace

FriendOf	
Alice	Victor
Bob	Wendy
Dan	Peggy
Faythe	Peggy
Peggy	Faythe
Victor	Alice

WorksWith	
Alice	Victor
Victor	Alice

Table 1.2: Examples of derived relationships represented by the graph of Figure 1.1.

AncestorOf	
Alice	Carol
Alice	Dan
Alice	Faythe
Alice	Grace
Bob	Carol
Bob	Dan
Bob	Faythe
Bob	Grace
Carol	Dan
Carol	Faythe
Carol	Grace
Faythe	Grace

GrandparentOf	
Alice	Dan
Alice	Faythe
Bob	Dan
Bob	Faythe
Carol	Grace

ChildOf	
Carol	Alice
Carol	Bob
Dan	Carol
Faythe	Carol
Grace	Faythe

WorkFriend	
Alice	Victor
Victor	Alice

1.1 Formalizing the graph data model

Before we introduce query languages to help express indirect relationships, we formalize the graph data model used in this work. We use an *edge-labeled graph data model*:[2]

Definition 1.1. A *graph* is a triple $\mathcal{G} = (\mathcal{V}, \Sigma, \mathbf{E})$, with $\mathcal{V}$ a finite set of nodes, Σ a finite set of edge labels, and $\mathbf{E} : \Sigma \to 2^{\mathcal{V}\times\mathcal{V}}$ a function mapping edge labels to edge relations. A graph is *unlabeled* if $|\Sigma| = 1$. We use $\mathcal{E}$ to refer to the union of all edge relations, and, when the graph is unlabeled, to the single edge relation. It is said that edge $(m, n) \in \mathcal{E}$ is an *outgoing edge* of m and an *incoming edge* of n.

A *path* of length k is a sequence $n_1\, \ell_1\, n_2 \cdots \ell_k\, n_{k+1}$ with $n_1, \ldots, n_{k+1} \in \mathcal{V}$, $\ell_1, \ldots, \ell_k \in \Sigma$, and, for all $1 \leq i \leq k$, $(n_i, n_{i+1}) \in \mathbf{E}\left(\ell_j\right)$. A path $n_1 \ldots n_{k+1}$ forms a *cycle* if $n_0 = n_{k+1}$ and the path contains at least one edge. A graph is *acyclic* if there are no cycles. A *tree* $\mathcal{T} = (\mathcal{V}, \Sigma, \mathbf{E})$ is an acyclic graph in which exactly one node, the *root*, has no incoming edges, and all other nodes have exactly one incoming edge. If (m, n) is an edge in $\mathcal{T}$, then node m is the *parent* of node n and node n is a *child* of node m. A graph is a *forest* if it is the union of a set of disjoint trees. A *chain* is a tree in which all nodes have at most one child.

[2]Various node-labeled graph data models are frequently used in the setting of XML data (see, e.g., [9, 97]). The results presented in this work can easily be adapted to these node-labeled graph data models.

Given a tree, and disregarding the direction of its edges, the *distance* between two nodes is the number of edges on the unique shortest path between them. We write $\|m \leftrightarrow n\|_{\mathcal{T}}$ to denote the distance between nodes m and n in tree $\mathcal{T}$. In general, the unique shortest path between m and n consists of a sequence of upward child-to-parent edges followed by a sequence of downward parent-to-child edges. If one of the nodes is an ancestor of the other, then the unique shortest path between m and n is directed. We define the *directed distance*, denoted by $\|m \rightarrow n\|_{\mathcal{T}}$, as $\|m \rightarrow n\|_{\mathcal{T}} = \|m \leftrightarrow n\|_{\mathcal{T}}$ if m is an ancestor of n and as $\|m \rightarrow n\|_{\mathcal{T}} = -\|m \leftrightarrow n\|_{\mathcal{T}}$ if n is an ancestor of m. Finally, the *depth* of tree $\mathcal{T}$, denoted by $\text{depth}(\mathcal{T})$, is the maximum distance of the root node to any leaf node.

Example 1.1. Consider the graphs of Figure 1.2. The unlabeled graph on the *left* is not acyclic, as there is a path from node u to itself via nodes v and w. The graph in the *middle* is acyclic, but not a tree, as node m has two parents. The graph $\mathcal{T}$ on the *right* is a tree with root node r and three leaf nodes. This tree is not a chain as node r has more than one child. We have $\text{depth}(\mathcal{T}) = \|r \rightarrow l_2\|_{\mathcal{T}} = \|r \rightarrow l_3\|_{\mathcal{T}} = 2$, $\|l_1 \leftrightarrow l_2\|_{\mathcal{T}} = \|l_2 \leftrightarrow l_1\|_{\mathcal{T}} = 3$, and $\|l_2 \rightarrow r\|_{\mathcal{T}} = \|l_3 \rightarrow r\|_{\mathcal{T}} = -2$.

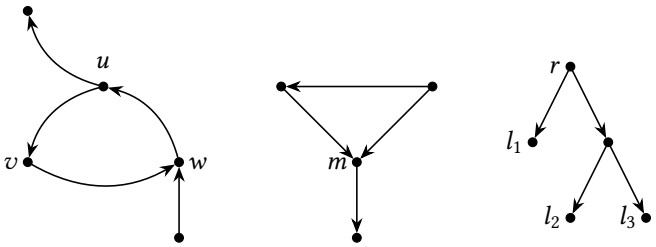


Figure 1.2: Three directed unlabeled graphs: one cyclic, one acyclic, and one a tree.

1.2 Tarski's relation algebra

The relationships that are part of a graph are referred to as *extensional relationships*, whereas the indirect relationships such as *GrandparentOf*, *AncestorOf*, *FamilyOf*, and *WorkFriend* are referred to as *intentional relationships*.[3] Examples of these intentional relationships can be found in Table 1.2. An important aspect of graph query languages is to provide the necessary tools to define these intentional relationships. For this purpose, many graph query languages rely at their core on a fragment of the relation algebra of Tarski [36, 89] augmented with the Kleene-star operator (reflexive transitive closure).

Central in the relation algebra is graph query navigation, which is primarily supported by *composition* ($\circ$). To see this, consider the query

$$\mathit{GrandparentOf} = \mathit{ParentOf} \circ \mathit{ParentOf}\,.$$

This query defines the *GrandparentOf* relationship by using composition to navigate from a grandparent to a grandchild via the *ParentOf* relationship. Where a single composition step

[3] The terminology of extensional data (facts) and intentional data (derived rules) is commonly used in the setting of Datalog [1].

allows one to take a single step through the graph, the *Kleene-plus operator* ($^+$) allows one to take one-or-more steps through the graph. As an example, the query

$$AncestorOf = [ParentOf]^+$$

defines the *AncestorOf* relationship. Alongside the Kleene-plus operator, one usually also uses the *Kleene-star operator* (*), which allows one to take zero-or-more steps. The query $[ParentOf]^*$, for example, defines an *AncestorOfAndSelf* relationship.

The relation algebra not only allows defining intentional relationships along directed paths in the graph, but also along undirected paths. As an example, the relationship *FamilyOf* is defined by the query

$$FamilyOf = [ParentOf \cup ParentOf^\frown]^*,$$

in which $\cup$ denotes the *union* and $^\frown$ denotes the *converse*, the latter of which is used to invert the *ParentOf* relationship to obtain the *ChildOf* relationship.

Not all relationships are definable via directed or undirected paths. Using *intersection* ($\cap$), the relation algebra allows to express relationships based on more complicated graph structures. A simple example is

$$WorkFriend = FriendOf \cap WorksWith,$$

which defines the *WorkFriend* relationship. The relationships defined using intersection can themselves also be indirect intentional relationships. A clear example is the query $(FriendOf \circ FriendOf) \cap FriendOf$ that defines the relationship of pairs (m, n) such that m is both a friend of n and a friend-of-a-friend of n.

Using only edge labels, composition, and intersection one can inductively describe *series-parallel graph structures*, which are frequently used to model simple electric networks [28]. The most basic series-parallel graphs are edges. To build bigger series-parallel graphs out of smaller ones, we use composition and intersection. Let e_1 and e_2 be expressions that describe series-parallel graph structures connecting nodes m_1 and n_1 and nodes m_2 and n_2, respectively. Intuitively speaking, the composition $e_1 \circ e_2$ expresses *serial composition* of these series-parallel graph structures by requiring that nodes $n_1 = m_2$. Likewise, the intersection $e_1 \cap e_2$ expresses *parallel composition* of these series-parallel graph structures by requiring that $n_1 = n_2$ and $m_1 = m_2$.

Example 1.2. Consider the three unlabeled graphs in Figure 1.3. The graph on the *right* is constructed by taking the parallel composition of the paths of length two and three (*left* and *middle*). The paths themselves are constructed by the serial composition of $\mathcal{E}$-edges. Consequently, the pair (p, q) is returned by the query $\mathcal{E} \circ \mathcal{E} \cap \mathcal{E} \circ \mathcal{E} \circ \mathcal{E}$.

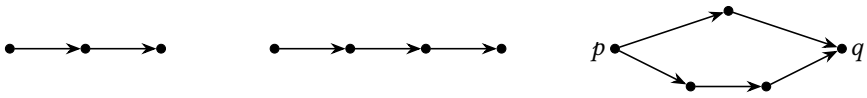


Figure 1.3: Three simple unlabeled graphs that can be constructed using serial and parallel composition. These three graphs are all series-parallel graphs.

Table 1.3: More examples of intentional relationships represented by the graph of Figure 1.1. The shorthand *Ggp* stands for great-grandparents.

Ggp		*GgpAndFriends*		*NonGgpAndFriends*	
Alice	Alice	Alice	Victor	Dan	Peggy
Bob	Bob	Bob	Wendy	Faythe	Peggy
				Peggy	Faythe
				Victor	Alice

Besides union and intersection, the relation algebra also provides *difference*. This operator can be used to eliminate certain structures from relationship definitions. Related to the intersection-queries, we have the query $\mathit{FriendOf} - \mathit{WorksWith}$ which defines a *NonWorkFriend* relationship and the query $(\mathit{FriendOf} \circ \mathit{FriendOf}) - \mathit{FriendOf}$ which defines the relationship of friend-of-friends that are not friends.

Using the *projection* operators (π_1 and π_2), relationships can be limited to either their begin-node or end-node. The query

$$\mathit{Ggp} = \pi_1[\mathit{ParentOf} \circ \mathit{ParentOf} \circ \mathit{ParentOf}]$$

defines the great-grandparents relationship of Table 1.3, *left*, and the query $\pi_2[\mathit{ParentOf} \circ \mathit{ParentOf} \circ \mathit{ParentOf}]$ defines a great-grandchildren relationship.

In the relation algebra, projection-expressions result in binary relations and can be used as a building block in combination with any other expression. In this manner, projections can be used to represent conditions that should hold on individual nodes that are part of the graph structure used to define a relationship.[4] To illustrate this usage of projections, consider the query

$$\mathit{GgpAndFriends} = \pi_1[\mathit{ParentOf} \circ \mathit{ParentOf} \circ \mathit{ParentOf}] \circ \mathit{FriendOf}\,.$$

In this query, the projection $\pi_1[\mathit{ParentOf} \circ \mathit{ParentOf} \circ \mathit{ParentOf}]$ is used to select great-grandparents, and the full query defines the *GgpAndFriends* relationship relating great-grandparents and their friends. This query yields the relationship of Table 1.3, *middle*. Intuitively speaking, the projections add the ability to define relationships in terms of tree-like branching graph patterns.

Example 1.3. Consider the labeled tree in Figure 1.4. The query $\pi_1[\ell_1] \circ \ell_2 \circ \pi_1[\ell_3] \circ \ell_4$ matches this tree structure, and will return the node pair (m, n).

We have seen that the projections express node conditions. The final operator of the relation algebra, the *coprojections* ($\overline{\pi}_1$ and $\overline{\pi}_2$), express node conditions which represent the complement of projections. For example, the query $\overline{\pi}_1[\mathit{ParentOf}]$ defines the relationship of all persons that do not have children, and the query

$$\mathit{NonGgpAndFriends} = \overline{\pi}_1[\mathit{ParentOf} \circ \mathit{ParentOf} \circ \mathit{ParentOf}] \circ \mathit{FriendOf}$$

defines the *NonGgpAndFriends* relationship of Table 1.3, *right*. This query relates non-great-grandparents and their friends.

[4] Several graph query languages opt for a two-sorted design in which node conditions are expressed by unary relations, see, e.g., [71, 74, 75, 90, 91]. We have chosen for a more uniform approach in which every operator yields binary relations, this to simplify presentation.

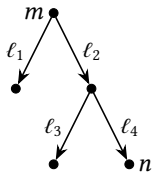


Figure 1.4: A labeled tree that matches $\pi_1[\ell_1] \circ \ell_2 \circ \pi_1[\ell_3] \circ \ell_4$ exactly.

Besides the above operators, Tarski's relation algebra also uses the atoms $\emptyset$, id, and di. The *empty-set* atom $\emptyset$ can be used to express queries that return an empty result, the *identity* atom id represents the set of all nodes (paired with themselves), and the *diversity* atom di represents the set of all distinct node pairs. As examples, consider the queries $(\mathit{FriendOf} \circ \mathit{FriendOf}) \cap \mathrm{di}$ and $(\mathit{FriendOf} \circ \mathit{FriendOf}) - \mathrm{id}$. Both queries return pairs (m, n) if n is a friend-of-a-friend of m, excluding the pairs that represent people that are friend-of-friends with themselves. As another example, consider the queries $(\mathit{ParentOf}^\frown \circ \mathit{ParentOf}) \cap \mathrm{di}$ and $(\mathit{ParentOf}^\frown \circ \mathit{ParentOf}) - \mathrm{id}$, which both return pairs (m, n) of siblings.

1.3 Formalizing the relation algebra

Next, we shall formalize the relation algebra of Tarski as used throughout this work.[5] In our setting, a query q maps a graph to a set of node tuples. We write $[\![q]\!]_{\mathcal{G}}$ to denote the *evaluation* of q on graph $\mathcal{G}$. We can interpret a query q as a *Boolean query*, in which case $[\![q]\!]_{\mathcal{G}} \neq \emptyset$ represents `True`. For simplicity, we assume that queries always yield binary relations (sets of node-pairs, $[\![q]\!]_{\mathcal{G}} \subseteq \mathcal{V} \times \mathcal{V}$). If R is a binary relation, then $\mathrm{R}|_1 = \{m \mid \exists n\ (m, n) \in \mathrm{R}\}$ and $\mathrm{R}|_2 = \{n \mid \exists m\ (m, n) \in \mathrm{R}\}$ denote the first and second column, respectively, of R.

Definition 1.2. The *relation algebra* with the Kleene-star is defined by the grammar

$$e := \emptyset \mid \mathrm{id} \mid \mathrm{di} \mid \ell \mid \ell^\frown \mid \pi_j[e] \mid \overline{\pi}_j[e] \mid e \circ e \mid e \cup e \mid e \cap e \mid e - e \mid [e]^*,$$

in which $\ell \in \Sigma$ and $j \in \{1, 2\}$. Let $\mathcal{G} = (\mathcal{V}, \Sigma, \mathbf{E})$ be a graph and let e be an expression. The semantics of evaluation is defined as follows:

$$\begin{aligned}
[\![\emptyset]\!]_{\mathcal{G}} &= \emptyset; \\
[\![\mathrm{id}]\!]_{\mathcal{G}} &= \{(m, m) \mid m \in \mathcal{V}\}; \\
[\![\mathrm{di}]\!]_{\mathcal{G}} &= \{(m, n) \mid m, n \in \mathcal{V} \wedge m \neq n\}; \\
[\![\ell]\!]_{\mathcal{G}} &= \mathbf{E}(\ell); \\
[\![\ell^\frown]\!]_{\mathcal{G}} &= \{(n, m) \mid (m, n) \in \mathbf{E}(\ell)\}; \\
[\![\pi_j[e]]\!]_{\mathcal{G}} &= \{(m, m) \mid m \in [\![e]\!]_{\mathcal{G}}|_j\}; \\
[\![\overline{\pi}_j[e]]\!]_{\mathcal{G}} &= [\![\mathrm{id}]\!]_{\mathcal{G}} - [\![\pi_j[e]]\!]_{\mathcal{G}}; \\
[\![e_1 \circ e_2]\!]_{\mathcal{G}} &= \{(m, n) \mid \exists z\ (m, z) \in [\![e_1]\!]_{\mathcal{G}} \wedge (z, n) \in [\![e_2]\!]_{\mathcal{G}}\};
\end{aligned}$$

[5] Our formalization of the relation algebra is adapted from the formalization used by Fletcher et al. [31–34, 87]. This also extends to the formalization of relation algebra fragments and the notions of path equivalence and Boolean equivalence introduced in Section 1.4.

$$\begin{aligned}
[\![e_1 \cup e_2]\!]_{\mathcal{G}} &= [\![e_1]\!]_{\mathcal{G}} \cup [\![e_2]\!]_{\mathcal{G}};\\
[\![e_1 \cap e_2]\!]_{\mathcal{G}} &= [\![e_1]\!]_{\mathcal{G}} \cap [\![e_2]\!]_{\mathcal{G}};\\
[\![e_1 - e_2]\!]_{\mathcal{G}} &= [\![e_1]\!]_{\mathcal{G}} - [\![e_2]\!]_{\mathcal{G}};\\
[\![[e]^*]\!]_{\mathcal{G}} &= \textstyle\bigcup_{i\geq 0} [\![e^i]\!]_{\mathcal{G}},
\end{aligned}$$

with $e^0 = \mathrm{id}$ and $e^k = e \circ e^{k-1}$. We denote the relation algebra, augmented with the Kleene-star, by $\mathcal{N}$. If an expression always evaluates to a subset of id, as is the case for projections and coprojections, then it is called a *node expression*.

We write $\mathcal{F} \subseteq \{\mathrm{di}, ^\frown, \pi, \overline{\pi}, \cap, -, ^*\}$ to denote a set of operators in which π represents both π_1 and π_2 and, likewise, $\overline{\pi}$ represents both $\overline{\pi}_1$ and $\overline{\pi}_2$. By $\mathcal{N}(\mathcal{F})$ we denote the fragment of $\mathcal{N}$ that only allows $\emptyset$, id, $\ell \in \Sigma$, $\circ$, $\cup$, and all operators in $\mathcal{F}$.

We sometimes use the shorthand $\mathrm{all} = (\mathrm{id} \cup \mathrm{di})$ and $[e]^+ = e \circ [e]^*$. Usually, the relation algebra also has a general *converse* operator $^{-1}$ with

$$[\![[e]^{-1}]\!]_{\mathcal{G}} = \{(n, m) \mid (m, n) \in [\![e]\!]_{\mathcal{G}}\}.$$

One can easily show that the general converse operator does not provide additional expressive power over the label-converse operator $^\frown$ that we already use, as one can always push the converse operator down to the level of edge labels by using the following rewrite rules:

$$\begin{aligned}
[\ell]^{-1} &= \ell^\frown;\\
[\ell^\frown]^{-1} &= \ell;\\
[f_j[e]]^{-1} &= f_j[e];\\
[e_1 \circ e_2]^{-1} &= [e_2]^{-1} \circ [e_1]^{-1};\\
[e_1 \oplus e_2]^{-1} &= [e_1]^{-1} \oplus [e_2]^{-1};\\
[[e]^*]^{-1} &= [[e]^{-1}]^*,
\end{aligned}$$

where $f \in \{\pi, \overline{\pi}\}$, $j \in \{1, 2\}$, and $\oplus \in \{\cup, \cap, -\}$. To simplify notation, we usually only allow converse at the edge-label level unless stated otherwise, but we will use the shorthand notation $[e]^{-1}$ to express the converse of e obtained by pushing the converse operator down to edge labels. Similarly, we write e^{-k}, $k \geq 0$, as a shorthand for $[e]^{-1} \circ e^{-(k-1)}$.

As an alternative to the relation algebra, we can also consider arbitrary first-order logic formulae to specify graph queries. Let $\mathcal{G} = (\mathcal{V}, \Sigma, \mathbf{E})$ be a graph with $\Sigma = \{\ell_1, \ldots, \ell_{|\Sigma|}\}$. We write FO[k] to denote the first-order logic over the structure $(\mathcal{V}; \ell_1, \ldots, \ell_{|\Sigma|})$ in which ℓ_i, $1 \leq i \leq |\Sigma|$, represents the edge relation $\mathbf{E}(\ell_i)$, and in which variables are restricted to a set of at most k variables. By $\mathrm{L}^k_{\infty\omega}$, we denote the standard infinitary first-order logic extension of FO[k] that allows conjunctions and disjunctions over arbitrary sets. It is well-known that the relation algebra (excluding the Kleene-star) has the same expressive power as FO[3] formulae with two free variables [36, 89] and that the Kleene-star operator can be expressed in $\mathrm{L}^3_{\infty\omega}$ [39].

1.4 Equivalence notions and language fragments

In this work, we primarily study the expressive power of the relation algebra and its fragments. To do so, we introduce the appropriate equivalence notions. We consider two such notions: path equivalence and Boolean equivalence.

Definition 1.3. Let q_1 and q_2 be queries. We say that q_1 and q_2 are *path-equivalent*, denoted by $q_1 \equiv_{\text{path}} q_2$, if, for every graph $\mathcal{G}$, $[\![q_1]\!]_{\mathcal{G}} = [\![q_2]\!]_{\mathcal{G}}$ and are *Boolean-equivalent*, denoted by $q_1 \equiv_{\text{bool}} q_2$, if, for every graph $\mathcal{G}$, $[\![q_1]\!]_{\mathcal{G}} = \emptyset$ if and only if $[\![q_2]\!]_{\mathcal{G}} = \emptyset$.

In a straightforward way, we can also consider path equivalence and Boolean equivalence with respect to a subset of all graphs (e.g., labeled chains or unlabeled trees).

Expressions that are path-equivalent are also Boolean-equivalent, but the reverse is generally not true.

Example 1.4. The equivalence $e_1 \cap e_2 \equiv_{\text{path}} e_1 - (e_1 - e_2)$ is well-known. Hence, also $e_1 \cap e_2 \equiv_{\text{bool}} e_1 - (e_1 - e_2)$. We also have $\pi_1[\ell] \equiv_{\text{bool}} \ell \equiv_{\text{bool}} \pi_2[\ell]$, but $\pi_1[\ell] \not\equiv_{\text{path}} \ell$ and $\ell \not\equiv_{\text{path}} \pi_2[\ell]$.

The equivalence notions introduced extend naturally to subsumption and equivalence notions between classes of expressions.

Definition 1.4. Let $z \in \{\text{path}, \text{bool}\}$. We say that the class of queries $\mathcal{L}_1$ is *z-subsumed* by the class of queries $\mathcal{L}_2$, denoted by $\mathcal{L}_1 \preceq_z \mathcal{L}_2$, if every query in $\mathcal{L}_1$ is z-equivalent to a query in $\mathcal{L}_2$. We say that the classes of queries $\mathcal{L}_1$ and $\mathcal{L}_2$ are *z-equivalent*, denoted by $\mathcal{L}_1 \equiv_z \mathcal{L}_2$, if $\mathcal{L}_1 \preceq_z \mathcal{L}_2$ and $\mathcal{L}_2 \preceq_z \mathcal{L}_1$.

In a straightforward way, we can also consider z-subsumption and z-equivalence with respect to a subset of all graphs (e.g., labeled chains or unlabeled trees).

Example 1.5. We have $\mathcal{N}(\mathcal{F}) \equiv_{\text{path}} \mathcal{N}(\mathcal{F} - \{\cap\})$ whenever $\{-\} \subseteq \mathcal{F}$.

Examples 1.4 and 1.5 illustrate that several fragments of the relation algebra can express exactly the same set of queries: if, for example, intersection is missing from a fragment that has difference, then difference can be used instead to express intersection. We observe that the following rewrite rules can be used to express operators using other operators:

$$\begin{aligned}
\pi_1[e] &= \pi_2[[e]^{-1}] = \overline{\pi}_j[\overline{\pi}_1[e]] = (e \circ [e]^{-1}) \cap \text{id} = (e \circ \text{all}) \cap \text{id};\\
\pi_2[e] &= \pi_1[[e]^{-1}] = \overline{\pi}_j[\overline{\pi}_2[e]] = ([e]^{-1} \circ e) \cap \text{id} = (\text{all} \circ e) \cap \text{id};\\
\overline{\pi}_1[e] &= \overline{\pi}_2[[e]^{-1}] = \text{id} - \pi_1[e];\\
\overline{\pi}_2[e] &= \overline{\pi}_1[[e]^{-1}] = \text{id} - \pi_2[e];\\
e_1 \cap e_2 &= e_1 - (e_1 - e_2),
\end{aligned}$$

with $j \in \{1, 2\}$. Let $\mathcal{F} \subseteq \{\text{di}, ^\frown, \pi, \overline{\pi}, \cap, -, ^*\}$. By $\underline{\mathcal{F}}$ we denote the set of operators obtained from $\mathcal{F}$ by adding operators to $\mathcal{F}$ that can be expressed using the operators already in $\mathcal{F}$ using the above rules.

1.5 Relation algebra and graph querying

The roots of Tarski's relation algebra can be traced back to the calculus of relation, of which the underlying theory was developed during the period 1864–1895 by De Morgan, Pierce, and Schröder. Starting in 1941, interest in the calculus of relations was revitalized by the work of Alfred Tarski and his students and colleagues, as they showed that the fundamentals of mathematical theory can be developed within the calculus of relations [36, 89].

The relation algebra is not only of theoretical interest. As we have already demonstrated in this chapter, the relation algebra can also be used as a simple yet powerful graph query

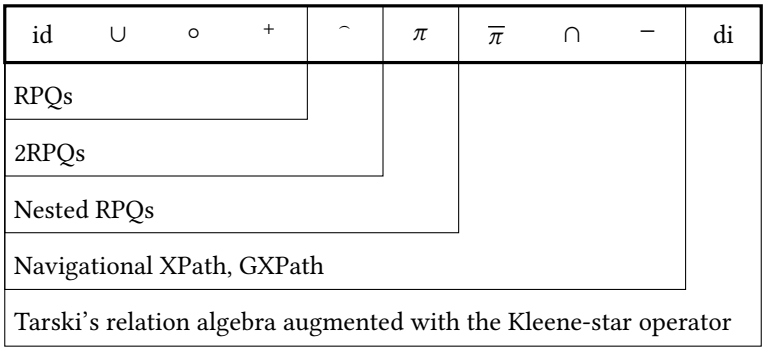


Figure 1.5: An overview of the relationships between graph query languages and fragments of the relation algebra.

language. Indeed, many graph query languages are directly related to fragments of the relation algebra. An overview of these relationships is visualized in Figure 1.5. The first example are the regular path queries (RPQs) that use *regular expressions* to define the labeling of paths in the graph that are of interest [27]. The result of a regular path query is a set of node pairs that are connected by a path of interest. The RPQs are equivalent to the relation algebra fragment $\mathcal{N}(^*)$ and the first-order fragment of the RPQs is equivalent to $\mathcal{N}()$. This is no coincidence: the RPQs with and without the Kleene-star operator are often studied as a formalization of the "greatest common divisor" of navigation in many practical graph and tree query languages, which also motivated our choice for the most basic language, $\mathcal{N}()$.

The RPQs can only be used to define very simple path-based intentional relationships. To strengthen the expressive power of the RPQs, several more expressive variants have been proposed and studied in the literature. The 2RPQs are obtained by adding converse ($^\frown$) [7] and the nested RPQs are obtained by also adding projections (π_1 and π_2) [8]. Usually, expressions in these languages serve as binary predicates in a conjunctive query framework such as the CRPQs and C2RPQs [15, 25]. The Regular Queries are a particular powerful CRPQ-based language that can express all queries in $\mathcal{N}(^\frown, \pi, \cap, ^*)$ [81]. The RPQ-based query languages have been studied in detail with respect to graph querying. To the best of our knowledge, the expressive power of these languages have not been studied in-depth on the tree data model and/or with respect to FO[2], however.

The relation algebra and its fragments are also at the core of many other graph query languages. Examples include the XPath query language (for querying XML data) [9, 10, 13, 20, 74, 75, 90, 91, 97], the SPARQL query language (for querying RDF data) [44, 83], the graph query language GXPath [71], and languages used for program verification such as PDL and KAT [30, 65]. Via the connections between the relation algebra and FO[2] we explore in Part III, there is also a close relation between the relation algebra and logics used in formal verification such as CTL, LTL, and the μ-calculus [21].

1.6 Overview of this work

Part I served as an introduction to the graph data model, to graph query languages based on Tarski's relation algebra, and to related terminology and notations used throughout this work. Next, in Parts II and III we present the results of our study on two distinct aspects

concerning the expressive power of fragments of Tarski's relation algebra in a non-traditional restricted setting.

More specifically, we study in Part II the relative expressive power of fragments of the relation algebra with respect to querying tree structures and chain structures. The main contribution presented in Part II is the identification of properties that we use to categorize relation algebra fragments according to their expressive power. We also prove that the expressive power of intersection and difference are limited: in many fragments, they are either redundant or add only limited expressive power. To prove these redundancies, we introduce condition automata and condition tree queries.

In Part III, we study the relationships in the expressive power of the relation algebra and the semi-join algebra, the latter which is obtained by replacing the expensive composition and Kleene-star operators in the relation algebra by semi-joins and a form of fixpoint iteration. The main contribution presented in Part II is that each fragment of the relation algebra where intersection and/or difference is used only on edges, is equivalent to a fragment of the semi-join algebra for queries that evaluate to a set of nodes. For practical relevance, we show how to utilize these relationships for optimizing graph query evaluation. We also study how the cost of other expensive operators in the relation algebra can be reduced.

Finally, in Part IV, we provide a short unifying conclusion on the research presented in Part II and III.

Part II

On Tarski's Relation Algebra

QUERYING TREES AND CHAINS

CHAPTER 2

Introduction[6]

The versatile tree data model can be used to model many natural and artificial hierarchical relations, including taxonomies, corporate hierarchies, and file and directory structures. It is therefore not surprising that tree data models have been continuously studied since the 1960s [13, 29, 93]. Modern query languages for querying tree data put heavy reliance on navigating the tree structure. Prime examples of this are XPath [10, 20, 74, 90] and the various JSON query languages [58]. Indeed, XPath queries primarily navigate XML documents via the parent-child and the ancestor-descendant axis. In the JSON data model most data retrieval is done by explicit top-down single-edge traversal steps of a data structure representation of the JSON data [58]. Even in more declarative settings, such as within the PostgreSQL relational database management system, the JSON query facilities primary aim at navigation [92]. We also find this focus on navigation outside the setting of tree data. As an example, we mention the nested relational database model that uses navigation of the nested structure as an important tool to query the data (see, e.g., [24]).

In the previous chapter, we argued that fragments of Tarski's relation algebra form the basis of many navigation oriented versatile graph and tree query languages including the regular path queries, XPath, and SPARQL. As such, the relation algebra and its fragments have already been studied in great detail, including studies that provide a complete characterization of the relative expressive power of these languages for querying graphs [31–34, 36, 87, 89] and partial characterizations of the relative expressive power of variants of these languages for querying unordered trees [9, 97] and sibling-ordered trees [74, 75, 90, 91]. The latter studies only covered a few fragments of the relation algebra, however, and a comprehensive study has not yet been undertaken before.

In this part, we undertake a more comprehensive study. More specifically, we investigate path equivalence and Boolean equivalence of fragments of the relation algebra when they are used to query unlabeled trees and labeled trees. We also present several results for querying unlabeled chains and labeled chains, as in many cases separations involving diversity are easier to obtain on chains than on trees. On their own, these results on chains provide additional insight into the expressive power of relation algebra fragments on strings.

Unfortunately, the relative simplicity of the tree data model and chain data model turns out to be a curse rather than a blessing: compared to the graph data model [31–34, 36, 87, 89], this simplicity makes it much more difficult to establish separation results using strong brute-force methods. Consequently, the study on trees and chains forces us to search for

[6]The results in this chapter are partly based on the papers "Relative expressive power of downward fragments of navigational query languages on trees and chains" [52, 53] and "The power of Tarski's relation algebra on trees" [57].

deeper methods to reach our goals. Therefore, we believe that our study not only gives insight in the expressive power of the relation algebra and its fragments, but also contributes to a better understanding of the fundamental differences between graph data models, tree data models, and chain data models.

The main contribution presented in this part is the introduction of several properties that can be used to categorize relation algebra fragments according to their expressive power. This in turn yields several separation results on trees and chains:

1. *Recognizing branches and siblings.* The language $\mathcal{N}()$ can only query trees by navigating alongside a single path from ancestor to descendant. Consequently, no query in $\mathcal{N}()$ can distinguish between chains and trees. Other query languages support recognizing branching up to a certain degree, and we can classify these languages accordingly.

To do so, we introduce *k-subtree-reductions.* Languages that are closed under k-subtree-reduction steps allow the removal of a child of a node that is structurally equivalent to at least k other children of that node without changing the outcome of Boolean queries. First, the query language $\mathcal{N}(^\frown, \pi, \overline{\pi}, \cap)$ is 1-*subtree-reducible* and, consequently, can only recognize siblings if they are not structurally equivalent. Next, query languages $\mathcal{N}(\mathcal{F})$ with $\mathrm{di} \in \mathcal{F}$ and $\cap \notin \underline{\mathcal{F}}$ are 2-*subtree-reducible* and can, in very limited circumstances, distinguish up to two structurally equivalent children of a node. Finally, the full relation algebra is 3-*subtree-reducible*, and query languages $\mathcal{N}(\mathcal{F})$ with $\{^\frown, -\} \subseteq \mathcal{F}$ or $\{\mathrm{di}, \cap\} \subseteq \mathcal{F}$ can always distinguish between nodes that have one, two, or at least three structurally equivalent children.

2. *Downward queries versus non-downward queries.* Queries in $\mathcal{N}(\mathcal{F})$ with $\mathcal{F} \subseteq \{\pi, \overline{\pi}, \cap, -, ^*\}$ yield node pairs (m, n) such that one can navigate from m to n by traversing along a sequence of parent-child axes. Hence, we call these query languages *downward.* We observe that these downward query languages are all 1-subtree-reducible, which puts an upper bound on their expressive power. Diversity and the converse operator are *non-downward* in nature. Based on this observation, it follows that languages with diversity or converse are not path-equivalent to downward query languages. This can usually be strengthened towards Boolean inequivalence, as diversity and converse play a significant role in 2-subtree-reducible languages and 3-subtree-reducible languages.

3. *Local queries versus non-local queries.* Queries in $\mathcal{N}(\mathcal{F})$ with $\mathcal{F} \subseteq \{^\frown, \pi, \overline{\pi}, \cap, -\}$ yield node pairs (m, n) such that one can navigate between m and n by traversing a number of edges, with the number depending only on the length of the query. Hence, we call these query languages *local.* Diversity and the Kleene-star operator are *non-local* in nature. Based on this observation, it follows that languages with diversity or Kleene-star are not path-equivalent to local query languages. This can be strengthened towards Boolean inequivalence, as diversity and Kleene-star can, in many cases, be used to express non-local properties on which unlabeled trees and labeled chains can be distinguished. We do so by exploiting the fact that many properties on which simple tree and chains can be distinguished are non-local and rely on a limited form of counting. A simple example of this are chain queries of the form "are there k edges in the chain labeled with edge-label ℓ".

4. *Monotonicity and homomorphisms.* A query language is *monotone* if, for every query q, every graph $\mathcal{G}$, and every graph $\mathcal{G}'$ obtained by adding nodes and edges to $\mathcal{G}$, we have $[\![q]\!]_{\mathcal{G}} \subseteq [\![q]\!]_{\mathcal{G}'}$. One the one hand, one can show that the query language $\mathcal{N}(\mathrm{di}, ^\frown, \pi, \cap, ^*)$ is monotone. To prove this, one can show that $\mathcal{N}(^\frown, \pi, \cap, ^*)$ is closed under *homomorphisms*

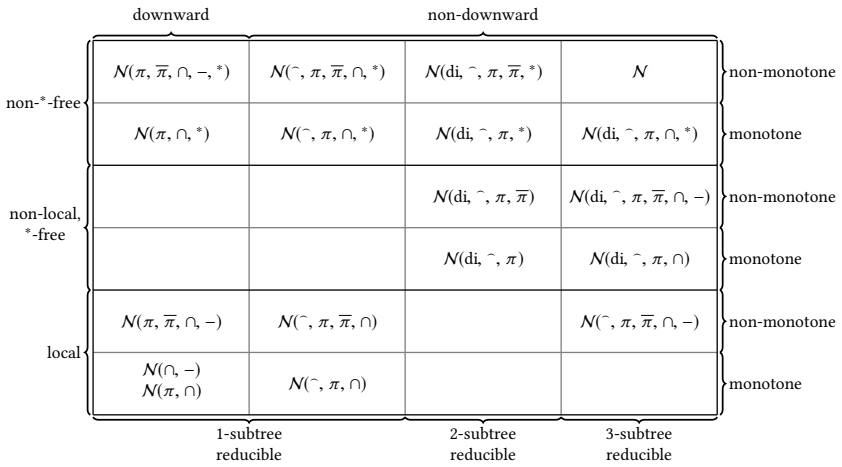


Figure 2.1: Initial classification of the relative expressive power of fragments of the relation algebra with respect to path queries on labeled trees. In each box, the largest fragment(s) that satisfy the classification of that particular box are included. The more to the right and to the top a certain box is situated, the stronger the expressiveness of the corresponding fragment(s) become.

and that $\mathcal{N}(\mathrm{di}, ^\frown, \pi, \cap, ^*)$ is closed under *injective homomorphisms*. Consequently, these languages have very limited expressive power, especially with respect to Boolean queries on unlabeled structures. We show that every query in $\mathcal{N}(^\frown, \pi, \cap, ^*)$ is, on unlabeled trees, Boolean-equivalent to either $\emptyset$ or a query of the form $\mathcal{E}^k$, a query that only puts a lower bound on the length of the longest path from the root to a leaf node.

On the other hand, the query languages $\mathcal{N}(\mathcal{F})$ with $\overline{\pi} \in \underline{\mathcal{F}}$ are *non-monotone*. As an example, one needs only coprojections to construct a Boolean query that puts an upper bound on the length of a chain. Such queries are not monotone and consequently not expressible in $\mathcal{N}(\mathrm{di}, ^\frown, \pi, \cap, ^*)$.

5. *The Kleene-star.* It is well-known that the transitive closure of a relation cannot be expressed in first-order logic and that the query language $\mathcal{N}(\mathrm{di}, ^\frown, \pi, \overline{\pi}, \cap, -)$ is path-equivalent to FO[3] [36, 89]. Consequently, even basic queries using the Kleene-star cannot be expressed by any expression in $\mathcal{N}(\mathrm{di}, ^\frown, \pi, \overline{\pi}, \cap, -)$. For Boolean queries on labeled structures, similar results can be shown. For unlabeled structures, Kleene-star does not always add expressive power, however. More specifically, when querying unlabeled chains, Kleene-star does not add expressive power to the monotone query languages, and, when querying unlabeled trees, Kleene-star does not add expressive power to the query languages that are both monotone and 2-subtree-reducible.

In Figure 2.1, we visualize the above categorization, which yields an initial classification of the expressive power of the query languages we study on trees. It does not provide all details, however, which we will start to unravel in this part. For a full index on how specific results are proven, we refer to the reader to Appendix A.

Our results on the relative expressive power are complete with respect to the downward fragments and the local fragments of the relation algebra and we have proven that intersection and difference are redundant in many of these fragments. These redundancies are presented in Chapter 4 for the downward language fragments and in Chapter 5 for the local language fragments. With respect to querying trees using the non-downward and non-local first-order fragments of the relation algebra, one challenging problem remains open, however: does adding difference yields more expressive power to fragments containing at least diversity, coprojections, and intersection?

In Chapter 3, the above classification is further formalized and most separations are proven. In Chapter 4, the redundancies of intersection and difference in downward query fragments are proven. In Chapter 5, the redundancies of intersection and difference in local query fragments are proven. Finally, in Chapter 6, we conclude on our findings, discuss the remaining open problems, and propose avenues for future work.

CHAPTER 3

Characterizations for querying trees and chains[7]

In this chapter, we formalize the classification of fragments of the relation algebra presented in Figure 2.1.

3.1 Organization

In Section 3.2, we introduce basic expression simplifications that are used throughout this chapter. Then, in Section 3.3, we introduce the k-subtree-reductions and present the separation results obtained using these reductions. In Section 3.4, we formalize what downward queries and local queries are and derive basic path separations based on these definitions. In Section 3.5, we add diversity to local query languages, and show that the resulting languages can express non-local counting-based structural properties on chains and trees. In Section 3.6, we compare various non-local query languages by exploiting the simple nature of diversity on chains. In Section 3.7, we look at queries that are closed under homomorphisms and injective homomorphisms. In Section 3.8, we look at the expressive power of the Kleene-star operator by using the relationship between the relation algebra and first-order logic. In Section 3.9, we explorer other remaining cases that can be solved using the relationship between the relation algebra and first-order logic. In Section 3.10, we look at cases that are solved using brute-force techniques. Finally, in Section 3.11, we look at related work.

3.2 Basic rewriting and expression simplification

In Section 1.4, we have provided equivalences showing that certain operators can be expressed using other operators, proving redundancies for these operators on graphs. Next, we provide additional equivalences that only hold on trees or chains, which imply additional redundancies for certain operators.

Proposition 3.1. *Let* $\mathcal{F}_1, \mathcal{F}_2 \subseteq \{\mathrm{di}, ^\frown, \pi, \overline{\pi}, \cap, -, ^*\}$ *with* $\{^\frown, ^*\} \subseteq \mathcal{F}_1$ *and* $\{\mathrm{di}, -, ^*\} \subseteq \mathcal{F}_2$.

(i) On labeled trees, we have $\mathcal{N}(\mathcal{F}_1 \cup \{\mathrm{di}, -\}) \preceq_{\mathrm{path}} \mathcal{N}(\mathcal{F}_1 \cup \{-\})$.

(ii) On labeled chains, we have $\mathcal{N}(\mathcal{F}_1 \cup \{\mathrm{di}\}) \preceq_{\mathrm{path}} \mathcal{N}(\mathcal{F}_1)$.

(iii) On labeled chains, we have $\mathcal{N}(\mathcal{F}_2 \cup \{^\frown\}) \preceq_{\mathrm{path}} \mathcal{N}(\mathcal{F}_2)$.

[7] The results in this chapter are partly based on the papers "Relative expressive power of downward fragments of navigational query languages on trees and chains" [52, 53] and "The power of Tarski's relation algebra on trees" [57].

Proof. On trees, we have the following equivalence

$$\mathrm{di} \equiv_{\mathrm{path}} ([\mathcal{E}^\frown]^* \circ [\mathcal{E}]^*) - \mathrm{id}.$$

On chains, the above can be simplified to the following equivalence:

$$\mathrm{di} \equiv_{\mathrm{path}} [\mathcal{E}]^+ \cup [\mathcal{E}^\frown]^+.$$

Hence, on chains $[\mathcal{E}^\frown]^+ \equiv_{\mathrm{path}} \mathrm{di} - [\mathcal{E}]^+$. We have

$$\mathcal{E}^\frown \equiv_{\mathrm{path}} [\mathcal{E}^\frown]^+ - \left([\mathcal{E}^\frown]^+ \circ [\mathcal{E}^\frown]^+\right).$$

Using $\mathcal{E}^\frown$ we can also express $\ell^\frown$ on chains, $\ell \in \Sigma$. We have $\ell^\frown \equiv_{\mathrm{path}} \mathcal{E}^\frown \circ \ell \circ \mathcal{E}^\frown$. □

Besides the above rewriting results that allows for the elimination of certain operators altogether, we can also use query rewriting to simplify the structure of queries. For example, we can simplify usage of the empty-set and identity atoms and the union operator:

Lemma 3.2. *Let $\mathcal{F} \subseteq \{\mathrm{di}, {}^\frown, \pi, \overline{\pi}, \cap, -, {}^*\}$ and let e be an expression in $\mathcal{N}(\mathcal{F})$.*

(i) Expression e is path-equivalent to a union of $\cup$-free expressions in $\mathcal{N}(\mathcal{F})$.

(ii) If $e \not\equiv_{\mathrm{path}} \emptyset$, then e is path-equivalent to an $\emptyset$-free expression in $\mathcal{N}(\mathcal{F})$.

(iii) If $\mathcal{F} \subseteq \{\mathrm{di}, {}^\frown, \pi, \overline{\pi}, {}^*\}$, e is $\cup$-free, and $e \not\equiv_{\mathrm{path}} \mathrm{id}$, then e is path-equivalent to an $\{\mathrm{id}, \cup\}$-free expression in $\mathcal{N}(\mathcal{F})$.

Proof. These three statements can be proven with straightforward rewrite rules, which we include for completeness. Let e, e_1, and e_2 be expressions and let $j \in \{1, 2\}$. For Statement (i), we have

$$\begin{aligned}
\pi_j[e_1 \cup e_2] &\equiv_{\mathrm{path}} \pi_j[e_1] \cup \pi_j[e_2];\\
\overline{\pi}_j[e_1 \cup e_2] &\equiv_{\mathrm{path}} \overline{\pi}_j[e_1] \circ \overline{\pi}_j[e_2];\\
(e_1 \cup e_2) \circ e &\equiv_{\mathrm{path}} (e_1 \circ e) \cup (e_2 \circ e);\\
e \circ (e_1 \cup e_2) &\equiv_{\mathrm{path}} (e \circ e_1) \cup (e \circ e_2);\\
(e_1 \cup e_2) \cap e &\equiv_{\mathrm{path}} (e_1 \cap e) \cup (e_2 \cap e);\\
e \cap (e_1 \cup e_2) &\equiv_{\mathrm{path}} (e \cap e_1) \cup (e \cap e_2);\\
(e_1 \cup e_2) - e &\equiv_{\mathrm{path}} (e_1 - e) \cup (e_2 - e);\\
e - (e_1 \cup e_2) &\equiv_{\mathrm{path}} (e - e_1) - e_2;\\
[e_1 \cup e_2]^* &\equiv_{\mathrm{path}} [[e_1]^* \circ [e_2]^*]^*.
\end{aligned}$$

For Statement (ii), we have

$$\begin{aligned}
\pi_j[\emptyset] &\equiv_{\mathrm{path}} \emptyset;\\
\overline{\pi}_j[\emptyset] &\equiv_{\mathrm{path}} \mathrm{id};\\
\emptyset \circ e &\equiv_{\mathrm{path}} e \circ \emptyset \equiv_{\mathrm{path}} \emptyset;\\
\emptyset \cup e &\equiv_{\mathrm{path}} e \cup \emptyset \equiv_{\mathrm{path}} e;
\end{aligned}$$

$$\begin{aligned} \emptyset \cap e &\equiv_{\text{path}} e \cap \emptyset \equiv_{\text{path}} \emptyset; \\ \emptyset - e &\equiv_{\text{path}} \emptyset; \\ e - \emptyset &\equiv_{\text{path}} e; \\ [\emptyset]^* &\equiv_{\text{path}} \text{id}. \end{aligned}$$

Finally, for Statement (iii), we have

$$\begin{aligned} \pi_j[\text{id}] &\equiv_{\text{path}} \text{id}; \\ \overline{\pi}_j[\text{id}] &\equiv_{\text{path}} \emptyset; \\ e \circ \text{id} &\equiv_{\text{path}} \text{id} \circ e \equiv_{\text{path}} e; \\ [\text{id}]^* &\equiv_{\text{path}} \text{id}. \end{aligned}$$

□

As a final simplification, we observe that the Kleene-star operator never adds expressive power when querying a given graph (with k nodes).

Lemma 3.3. *Let $\mathcal{G}$ be a graph with k nodes and let e be an expression. Let e' be the expression obtained from e by replacing every subexpressions of the form $[g]^*$ by $\bigcup_{0 \le i \le k} g^i$. We have $[\![e]\!]_{\mathcal{G}} = [\![e']\!]_{\mathcal{G}}$.*

We can use the above elimination trick to carry over results from language fragments without the Kleene-star to language fragments with the Kleene-star.

3.3 Detecting branches and subtree-reductions

The obvious way to detect whether or not a tree is a chain is by detecting whether some node has several children. This is particularly easy when these children are connected to their parent using distinct edge-label. Next, we show that the most basic language is not capable of detecting labeled branching.

Lemma 3.4. *Let e be an expression in $\mathcal{N}(^*)$. If there exists a tree $\mathcal{T}$ such that $[\![e]\!]_{\mathcal{T}} \neq \emptyset$, then there exists a chain C such that $[\![e]\!]_C \neq \emptyset$.*

Proof. Let $\mathcal{T} = (\mathcal{V}, \Sigma, \mathbf{E})$. As $\mathcal{T}$ is given, we can use Lemma 3.3, and we assume that e is *-free. Using Lemma 3.2 (i), we write e as a union of $\cup$-free expressions. Due to $[\![e]\!]_{\mathcal{T}} \neq \emptyset$, the union must contain a $\cup$-free expression e' for which $[\![e']\!]_{\mathcal{T}} \neq \emptyset$. If $e' \equiv_{\text{path}} \text{id}$, then a chain C with a single node suffices. Else, we use Lemma 3.2 (iii) and conclude that e' can be written as a composition of edge labels $\ell_1 \circ \ldots \circ \ell_k$. In this case, a chain C representing the path $n_1\ \ell_1\ n_2 \ldots n_i\ \ell_k\ n_{k+1}$ suffices. □

In contrast, most other languages are able to detect labeled branching, as shown next.

Proposition 3.5. *Let $\mathcal{F} \subseteq \{\text{di}, \hat{}, \pi, \overline{\pi}, \cap, -, ^*\}$. If $\pi \in \underline{\mathcal{F}}$ or $\hat{} \in \underline{\mathcal{F}}$, then, already on labeled trees, we have $\mathcal{N}(\mathcal{F}) \not\leq_{\text{bool}} \mathcal{N}(^*)$.*

Proof. Consider the expressions $e_1 = \ell_1\hat{} \circ \ell_2$ and $e_2 = \pi_1[\ell_1] \circ \pi_1[\ell_2]$, which are Boolean-equivalent. Clearly, for any tree $\mathcal{T}'$, we have $[\![e_i]\!]_{\mathcal{T}'} \neq \emptyset$, $i \in \{1, 2\}$, only if $\mathcal{T}'$ has a node with at least two children, one reachable via an edge labeled ℓ_1 and another via an edge labeled ℓ_2. Hence, for the tree $\mathcal{T}$ of Figure 3.1, we have $[\![e_i]\!]_{\mathcal{T}} \neq \emptyset$ and for all chains C, we have $[\![e_i]\!]_C = \emptyset$. Let e be any expression in $\mathcal{N}(\mathcal{F})$ with $[\![e_i]\!]_{\mathcal{T}} \neq \emptyset$. Due to Lemma 3.4, there exists a chain C with $[\![e]\!]_C \neq \emptyset$. Hence, e is not Boolean-equivalent to e_i, and we conclude that $\mathcal{N}(^*)$ cannot express e_i. □

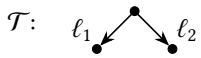


Figure 3.1: A tree with two labeled branches.

Figure 3.2: Tree $\mathcal{T}_1$ (a chain) on the *left*, $\mathcal{T}_2$ in the *middle*, and tree $\mathcal{T}_3$ on the *right*. These trees can be distinguished by counting the number of children of the root.

Detecting branches in the situation above, where a single node has several structurally distinct branches, is relatively simple. Next, we look at which language fragments are able to detect branching when all branches are structurally identical. As a first step towards this goal, we take advantage of the simple structure of trees to exhibit limitations on the expressive power of relation algebra fragments. Thereto, we introduce *subtree-reductions*.

Let $k > 0$. A *k-subtree-reduction step* on tree $\mathcal{T} = (\mathcal{V}, \Sigma, \mathbf{E})$ consists of first finding different nodes $m, n_1, \ldots, n_{k+1} \in \mathcal{V}$ and an edge label $\ell \in \Sigma$ such that the subtrees rooted at $n_1, \ldots, n_{k+1}$ are isomorphic and $(m, n_1), \ldots, (m, n_{k+1}) \in \mathbf{E}(\ell)$, and then picking a node n_i, $1 \leq i \leq k+1$, and removing the subtree rooted at n_i.

Definition 3.1. We say that a tree is *k-subtree-reducible* if we can apply a k-subtree-reduction step.[8]

Example 3.1. Consider the unlabeled trees $\mathcal{T}_1$, $\mathcal{T}_2$, and $\mathcal{T}_3$ in Figure 3.2. The tree $\mathcal{T}_1$ can be obtained by a 1-subtree-reduction step on $\mathcal{T}_2$ and $\mathcal{T}_2$ can be obtained by a 2-subtree-reduction step on $\mathcal{T}_3$. Consequently, $\mathcal{T}_1$ can also be obtained by two 1-subtree-reduction steps on $\mathcal{T}_3$. Hence, $\mathcal{T}_2$ is 1-subtree-reducible and $\mathcal{T}_3$ is 1-subtree-reducible and 2-subtree-reducible.

We now exhibit conditions under which the result of a relation algebra expression is invariant under subtree-reductions at the Boolean level. Using standard k-pebble-games [37, 39, 70], we conclude the following:

Lemma 3.6. *Let $\mathcal{T}$ be a tree and let $\mathcal{T}'$ be obtained from $\mathcal{T}$ by a k-subtree-reduction step. For every query q in $\mathrm{L}^k_{\infty\omega}$, we have $[\![q]\!]_{\mathcal{T}} \neq \emptyset$ if and only if $[\![q]\!]_{\mathcal{T}'} \neq \emptyset$.*

Observe that $\mathcal{N} \preceq_{\text{path}} \mathrm{L}^3_{\infty\omega}$. In Part III we study the relationship between fragments of the relation algebra and $\mathrm{L}^2_{\infty\omega}$. In particular, Theorem 8.6 will show that the language $\mathcal{N}(\mathrm{di}, ^\frown, \pi, \overline{\pi})$ and its fragments are already Boolean-subsumed by $\mathrm{L}^2_{\infty\omega}$ on graphs. Combining this with Lemma 3.6 yields the following:

Proposition 3.7. *Let $\mathcal{F} \subseteq \{\mathrm{di}, ^\frown, \pi, \overline{\pi}, \cap, -, ^*\}$, let e be an expression in $\mathcal{N}(\mathcal{F})$, let $\mathcal{T}$ be a tree, and let $\mathcal{T}'$ be obtained from $\mathcal{T}$ by a k-subtree-reduction step.*

(i) If $k \geq 3$, then $[\![e]\!]_{\mathcal{T}} \neq \emptyset$ if and only if $[\![e]\!]_{\mathcal{T}'} \neq \emptyset$.

(ii) If $k \geq 2$ and $\cap \notin \mathcal{F}$, then $[\![e]\!]_{\mathcal{T}} \neq \emptyset$ if and only if $[\![e]\!]_{\mathcal{T}'} \neq \emptyset$.

[8] The 1-subtree-reduction step bears a close relationship to the F+B-index and the F&B-index used for indexing the structure of tree data [63]. As with the F+B-index, the 1-subtree-reduction steps will only merge child nodes when both the forward structure (subtrees rooted at the children) and backward structure (children of a single parent) "behave equivalently". The k-subtree-reduction step generalizes their underlying principles by also taking into account how often (up-to-k) these structures occur.

For 1-subtree-reduction steps, we can further strengthen Proposition 3.7:

Proposition 3.8. *Let $\mathcal{F} \subseteq \{^\frown, \pi, \overline{\pi}, \cap, ^*\}$, let e be an expression in $\mathcal{N}(\mathcal{F})$, let $\mathcal{T}$ be a tree, and let $\mathcal{T}'$ be obtained from $\mathcal{T}$ by a 1-subtree-reduction step. We have $[\![e]\!]_{\mathcal{T}} \neq \emptyset$ if and only if $[\![e]\!]_{\mathcal{T}'} \neq \emptyset$.*

Proof. Let $\mathcal{T} = (\mathcal{V}, \Sigma, \mathbf{E})$ and let $\mathcal{T}' = (\mathcal{V}', \Sigma', \mathbf{E}')$. We observe that the number of nodes in $\mathcal{T}'$ is bounded above by the number of nodes in $\mathcal{T}$. Hence, by Lemma 3.3, we can assume that e is Kleene-star free.

Let $n_1, n_2 \in \mathcal{V}$ be siblings in $\mathcal{T}$ such that the subtrees rooted at n_1 and n_2 are isomorphic and $\mathcal{T}'$ is obtained from $\mathcal{T}$ by eliminating the subtree rooted at n_2. Let $\mathcal{V}_1$ and $\mathcal{V}_2$ be the nodes in the subtrees of $\mathcal{T}$ rooted at n_1 and n_2, respectively, and let $b : \mathcal{V}_1 \to \mathcal{V}_2$ be the bijection establishing that these subtrees are isomorphic. Let g be the identity on $\mathcal{V} - (\mathcal{V}_1 \cup \mathcal{V}_2)$, and let $f = b \cup b^{-1} \cup g$. Since f is an automorphism of $\mathcal{T}$, we have

(i) if $m, n \in \mathcal{V}$, then $(m, n) \in [\![e]\!]_{\mathcal{T}}$ if and only if $(f(m), f(n)) \in [\![e]\!]_{\mathcal{T}}$.

By induction on the length of e, one can prove that we have in addition

(ii) if $m \in \mathcal{V}_1, n \in \mathcal{V}_2$ or $m \in \mathcal{V}_2, n \in \mathcal{V}_1$, then $(m, n) \in [\![e]\!]_{\mathcal{T}}$ implies $(f(m), n) \in [\![e]\!]_{\mathcal{T}}$;

(iii) if $m, n \in \mathcal{V}'$, then $(m, n) \in [\![e]\!]_{\mathcal{T}}$ if and only if $(m, n) \in [\![e]\!]_{\mathcal{T}'}$.

Observe that $f = f^{-1}$. Hence, Property (ii) also yields $(m, n) \in [\![e]\!]_{\mathcal{T}}$ implies $(m, f(n)) \in [\![e]\!]_{\mathcal{T}}$. The base cases are expressions of the form $\emptyset$, id, ℓ, and $\ell^\frown$, with ℓ an edge label, for which it is straightforward to verify that the Properties (ii) and (iii) hold. Now assume that the Properties hold for all expressions e of length at most i. Let e be an expression of length $i + 1$. We distinguish the following cases:

1. $e = \overline{\pi}_1[e']$.

(ii) We have $(m, n) \in [\![e]\!]_{\mathcal{T}}$ only if $m = n$. Hence, the premise of Property (ii) is not applicable to e in this case, and we conclude that the Property holds voidly.

(iii) We assume $m, n \in \mathcal{V}'$. If $(m, n) \in [\![e]\!]_{\mathcal{T}}$, then $m = n$ and, for all z, $(m, z) \notin [\![e']\!]_{\mathcal{T}}$. In this case, we also have, for all $z \in \mathcal{V}'$, $(m, z) \notin [\![e']\!]_{\mathcal{T}}$. By applying Property (iii) on e', we obtain $(m, z) \notin [\![e']\!]_{\mathcal{T}'}$ for all z. Hence, we conclude $(m, n) \in [\![e]\!]_{\mathcal{T}'}$.

If $(m, n) \in [\![e]\!]_{\mathcal{T}'}$, then $m = n$ and, for all z', $(m, z') \notin [\![e']\!]_{\mathcal{T}'}$. By applying Property (iii) on e', we obtain $(m, z') \notin [\![e']\!]_{\mathcal{T}}$ for all $z' \in \mathcal{V}'$. Let $z'' \notin \mathcal{V}'$. We have $z'' \in \mathcal{V}_2$, $f(z'') \in \mathcal{V}'$, and $(m, f(z'')) \notin [\![e']\!]_{\mathcal{T}}$. If $m \in \mathcal{V}_1$, then we apply Property (ii) on e' to obtain $(m, z'') \in [\![e']\!]_{\mathcal{T}}$ implies $(m, f(z'')) \in [\![e']\!]_{\mathcal{T}}$. Else, if $m \notin \mathcal{V}_1$, then $f(m) = m$ and we apply Property (i) on e' to obtain $(m, z'') \in [\![e']\!]_{\mathcal{T}}$ implies $(m, f(z'')) \in [\![e']\!]_{\mathcal{T}}$. Hence, in both cases, $(m, z'') \notin [\![e']\!]_{\mathcal{T}}$ must hold. We conclude $(m, n) \in [\![e]\!]_{\mathcal{T}}$.

2. $e = e_1 \circ e_2$.

(ii) We assume $m \in \mathcal{V}_1, n \in \mathcal{V}_2$. The case for $m \in \mathcal{V}_2, n \in \mathcal{V}_1$ is analogous. We have $(m, n) \in [\![e]\!]_{\mathcal{T}}$ if there exists z such that $(m, z) \in [\![e_1]\!]_{\mathcal{T}}$ and $(z, n) \in [\![e_2]\!]_{\mathcal{T}}$. We distinguish three cases. (a) $z \in \mathcal{V}_1$. We apply Property (i) on e_1 and Property (ii) on e_2 to obtain $(f(m), f(z)) \in [\![e_1]\!]_{\mathcal{T}}$ and $(f(z), n) \in [\![e_2]\!]_{\mathcal{T}}$. Hence, $(f(m), n) \in [\![e]\!]_{\mathcal{T}}$. (b) $z \in \mathcal{V}_2$. We apply Property (ii) on e_1 to obtain $(f(m), z) \in [\![e_1]\!]_{\mathcal{T}}$. Hence, $(f(m), n) \in [\![e]\!]_{\mathcal{T}}$. (c) $z \notin \mathcal{V}_1$ *and* $z \notin \mathcal{V}_2$. We have $f(z) = z$ and we apply Property (i) on e_1 to obtain $(f(m), z) \in [\![e_1]\!]_{\mathcal{T}}$. Hence, $(f(m), n) \in [\![e]\!]_{\mathcal{T}}$.

(iii) We assume $m, n \in \mathcal{V}'$. If $(m,n) \in [\![e]\!]_{\mathcal{T}}$, then there exists z such that $(m,z) \in [\![e_1]\!]_{\mathcal{T}}$ and $(z,n) \in [\![e_2]\!]_{\mathcal{T}}$. If $z \in \mathcal{V}'$, then we apply Property (iii) on e_1 and e_2 to obtain $(m,z) \in [\![e_1]\!]_{\mathcal{T}'}$ and $(z,n) \in [\![e_2]\!]_{\mathcal{T}'}$. Hence, $(m,n) \in [\![e]\!]_{\mathcal{T}'}$. If $z \notin \mathcal{V}'$, then $z \in \mathcal{V}_2$ and $f(z) \in \mathcal{V}'$. If, additionally, $m \notin \mathcal{V}_1$, then $f(m) = m$ and we apply Property (i) on e' to obtain $(m, f(z)) \in [\![e_1]\!]_{\mathcal{T}}$. Else, if $m \in \mathcal{V}_1$, then we apply Property (ii) on e_1 to obtain $(m, f(z)) \in [\![e_1]\!]_{\mathcal{T}}$. Likewise, we can also obtain $(f(z), n) \in [\![e_2]\!]_{\mathcal{T}}$. We apply Property (iii) on e_1 and e_2 to obtain $(m, f(z)) \in [\![e_1]\!]_{\mathcal{T}'}$ and $(f(z), n) \in [\![e_2]\!]_{\mathcal{T}'}$. Hence, $(m,n) \in [\![e]\!]_{\mathcal{T}'}$.

If $(m,n) \in [\![e]\!]_{\mathcal{T}'}$, then there exists z' such that $(m,z') \in [\![e_1]\!]_{\mathcal{T}'}$ and $(z',n) \in [\![e_2]\!]_{\mathcal{T}'}$. By applying Property (iii) on e_1 and e_2, we obtain $(m,z') \in [\![e_1]\!]_{\mathcal{T}}$, $(z',n) \in [\![e_2]\!]_{\mathcal{T}}$, and we conclude $(m,n) \in [\![e]\!]_{\mathcal{T}}$.

3. $e = e_1 \cup e_2$.

(ii) We assume $m \in \mathcal{V}_1, n \in \mathcal{V}_2$. The case for $m \in \mathcal{V}_2, n \in \mathcal{V}_1$ is analogous. We have $(m,n) \in [\![e]\!]_{\mathcal{T}}$ if $(m,n) \in [\![e_1]\!]_{\mathcal{T}}$ or $(m,n) \in [\![e_2]\!]_{\mathcal{T}}$. We apply Property (ii) on e_1 and e_2 to obtain $(f(m),n) \in [\![e_1]\!]_{\mathcal{T}}$ or $(f(m),n) \in [\![e_2]\!]_{\mathcal{T}}$. Hence, we conclude $(f(m),n) \in [\![e]\!]_{\mathcal{T}}$.

(iii) We assume $m, n \in \mathcal{V}'$. We have $(m,n) \in [\![e]\!]_{\mathcal{T}}$ if and only if $(m,n) \in [\![e_1]\!]_{\mathcal{T}}$ or $(m,n) \in [\![e_2]\!]_{\mathcal{T}}$. By applying Property (iii) on e_1 and e_2, we have $(m,n) \in [\![e_1]\!]_{\mathcal{T}}$ or $(m,n) \in [\![e_2]\!]_{\mathcal{T}}$ if and only if $(m,n) \in [\![e_1]\!]_{\mathcal{T}'}$ or $(m,n) \in [\![e_2]\!]_{\mathcal{T}'}$. Hence, we conclude $(m,n) \in [\![e]\!]_{\mathcal{T}}$ if and only if $(m,n) \in [\![e]\!]_{\mathcal{T}'}$.

4. $e = e_1 \cap e_2$.

(ii) We assume $m \in \mathcal{V}_1, n \in \mathcal{V}_2$. The case for $m \in \mathcal{V}_2, n \in \mathcal{V}_1$ is analogous. We have $(m,n) \in [\![e]\!]_{\mathcal{T}}$ if $(m,n) \in [\![e_1]\!]_{\mathcal{T}}$ and $(m,n) \in [\![e_2]\!]_{\mathcal{T}}$. We apply Property (ii) on e_1 and e_2 to obtain $(f(m),n) \in [\![e_1]\!]_{\mathcal{T}}$ and $(f(m),n) \in [\![e_2]\!]_{\mathcal{T}}$. Hence, we conclude $(f(m),n) \in [\![e]\!]_{\mathcal{T}}$.

(iii) We assume $m, n \in \mathcal{V}'$. We have $(m,n) \in [\![e]\!]_{\mathcal{T}}$ if and only if $(m,n) \in [\![e_1]\!]_{\mathcal{T}}$ and $(m,n) \in [\![e_2]\!]_{\mathcal{T}}$. By applying Property (iii) on e_1 and e_2, we have $(m,n) \in [\![e_1]\!]_{\mathcal{T}}$ and $(m,n) \in [\![e_2]\!]_{\mathcal{T}}$ if and only if $(m,n) \in [\![e_1]\!]_{\mathcal{T}'}$ and $(m,n) \in [\![e_2]\!]_{\mathcal{T}'}$. Hence, we conclude $(m,n) \in [\![e]\!]_{\mathcal{T}}$ if and only if $(m,n) \in [\![e]\!]_{\mathcal{T}'}$.

The case for $e = \pi_1[e']$ can be obtained by rewriting $\pi_1[e']$ to $\overline{\pi}_1[\overline{\pi}_1[e']]$. The cases for $e = \pi_2[e']$ and $e = \overline{\pi}_2[e']$ are analogous to $e = \pi_1[e']$ and $e = \overline{\pi}_1[e']$, respectively, completing our proof. □

From the limitations imposed by Proposition 3.7 and Proposition 3.8 on the expressive power of the fragments considered, we deduce the following separation results:

Proposition 3.9. *Let* $\mathcal{F}_1, \mathcal{F}_2 \subseteq \{\mathrm{di}, ^\frown, \pi, \overline{\pi}, \cap, ^*\}$ *with* $\mathrm{di} \notin \mathcal{F}_1$ *and* $\cap \notin \underline{\mathcal{F}_2}$. *Already on unlabeled trees, we have*

(i) $\mathcal{N}(\mathrm{di}) \not\leq_{\mathrm{bool}} \mathcal{N}(\mathcal{F}_1)$ *and* $\mathcal{N}(^\frown, -) \not\leq_{\mathrm{bool}} \mathcal{N}(\mathcal{F}_1)$ *and*

(ii) $\mathcal{N}(\mathrm{di}, \cap) \not\leq_{\mathrm{bool}} \mathcal{N}(\mathcal{F}_2)$ *and* $\mathcal{N}(^\frown, -) \not\leq_{\mathrm{bool}} \mathcal{N}(\mathcal{F}_2)$.

Proof. Consider the unlabeled trees $\mathcal{T}_1$, $\mathcal{T}_2$, and $\mathcal{T}_3$ in Example 3.1. Since $\mathcal{T}_1$ can be obtained by a 1-subtree-reduction on $\mathcal{T}_2$, we have, by Proposition 3.8, that, for every e in $\mathcal{N}(\mathcal{F}_1)$, $[\![e]\!]_{\mathcal{T}_2} \neq \emptyset \iff [\![e]\!]_{\mathcal{T}_1} \neq \emptyset$. Now consider the expressions $e_1 = \ell \circ \mathrm{di} \circ \mathrm{di} \circ \ell$ in $\mathcal{N}(\mathrm{di})$ and $e_2 = (\ell^\frown \circ \ell) - \mathrm{id}$ in $\mathcal{N}(^\frown, -)$. We have $[\![e_1]\!]_{\mathcal{T}_2} \neq \emptyset$ and $[\![e_2]\!]_{\mathcal{T}_2} \neq \emptyset$, while $[\![e_1]\!]_{\mathcal{T}_1} = [\![e_2]\!]_{\mathcal{T}_1} = \emptyset$.

Since $\mathcal{T}_2$ can be obtained by a 2-subtree-reduction on $\mathcal{T}_3$, we have, by Proposition 3.7 (ii), that, for every e in $\mathcal{N}(\mathcal{F}_2)$, $[\![e]\!]_{\mathcal{T}_3} \neq \emptyset \iff [\![e]\!]_{\mathcal{T}_2} \neq \emptyset$. Now consider the expressions $e_3 = (((\mathrm{di} \circ \ell) \cap \mathrm{di}) \circ ((\mathrm{di} \circ \ell) \cap \mathrm{di})) \cap \mathrm{di}$ in $\mathcal{N}(\mathrm{di}, \cap)$ and $e_4 = (((\ell^\frown \circ \ell) - \mathrm{id}) \circ ((\ell^\frown \circ \ell) - \mathrm{id})) - \mathrm{id}$ in $\mathcal{N}(^\frown, -)$. We have $[\![e_3]\!]_{\mathcal{T}_3} \neq \emptyset$ and $[\![e_4]\!]_{\mathcal{T}_3} \neq \emptyset$, while $[\![e_3]\!]_{\mathcal{T}_2} = [\![e_4]\!]_{\mathcal{T}_2} = \emptyset$. □

The proof of Proposition 3.9 relies on languages being able to distinguish at least one, two, or three structurally equivalent children of a node. To do so, the proof uses minimal languages that satisfy the conditions of Propositions 3.7 and 3.8. Hence, the classification provided by k-subtree-reductions is strict.

3.4 Downward queries and local queries

Relation algebra expressions without diversity or converse can only inspect a tree downwards from ancestor to descendant. Likewise, relation algebra expressions without diversity or Kleene-star can only inspect a local neighborhood around a given node. To study this in more detail, we first define the notions of downward querying and locality:

Definition 3.2. A query q is *downward* if, for every tree $\mathcal{T}$, and for all nodes m and n, $(m, n) \in [\![q]\!]_{\mathcal{T}}$ only if m is an ancestor of n (i.e., there is a directed path from m to n).[9] A query q is *local* if there exists $k \geq 0$ such that, for every tree $\mathcal{T}$, and for all nodes m and n, $(m, n) \in [\![q]\!]_{\mathcal{T}}$ if and only if $(m, n) \in [\![q]\!]_{\mathcal{T}'}$, with $\mathcal{T}'$ the smallest subtree of $\mathcal{T}$ containing all nodes at distance at most k from the nearest common ancestor of m and n.

A straightforward induction on the length of expressions yields the following:

Proposition 3.10. *Let $\mathcal{F}_1 \subseteq \{\pi, \overline{\pi}, \cap, -, ^*\}$ and $\mathcal{F}_2 \subseteq \{^\frown, \pi, \overline{\pi}, \cap, -\}$.*

(i) Every expression in $\mathcal{N}(\mathcal{F}_1)$ is downward.

(ii) Every expression in $\mathcal{N}(\mathcal{F}_2)$ is local.

The queries di and $\mathcal{E}^\frown$ are not downward and the queries di and $[\mathcal{E}]^*$ are not local. Combined with Proposition 3.10, we derive

Corollary 3.11. *Let $\mathcal{F}_1 \subseteq \{\pi, \overline{\pi}, \cap, -, ^*\}$ and $\mathcal{F}_2 \subseteq \{^\frown, \pi, \overline{\pi}, \cap, -\}$. Already on unlabeled chains, we have*

(i) $\mathcal{N}(\mathrm{di}) \not\leq_{\mathrm{path}} \mathcal{N}(\mathcal{F}_1)$ and $\mathcal{N}(^\frown) \not\leq_{\mathrm{path}} \mathcal{N}(\mathcal{F}_1)$

(ii) $\mathcal{N}(\mathrm{di}) \not\leq_{\mathrm{path}} \mathcal{N}(\mathcal{F}_2)$ and $\mathcal{N}(^*) \not\leq_{\mathrm{path}} \mathcal{N}(\mathcal{F}_2)$.

The behavior of local queries is straightforward to study, especially on chains. To do so, we use the following technical Lemma.

Lemma 3.12. *Let $\mathcal{F} \subseteq \{^\frown, \pi, \overline{\pi}, \cap, -\}$ and let e be an $\cup$-free expression in $\mathcal{N}(\mathcal{F})$. There exists an integer k such that, for every chain C and every $(m, n) \in [\![e]\!]_C$, we have $\|m \to n\|_C = k$.*

[9] We do not require m to be a strict ancestor of n, as there is always a directed path of length 0 from a node to itself. Hence, m and n can be the same node.

Proof. The value k can be derived directly from the structure of the $\cup$-free expression e. We define the function $\delta(e)$ inductively by:

$$\begin{aligned}\delta(\emptyset) = \delta(\mathrm{id}) &= 0;\\ \delta(\ell) &= 1;\\ \delta(\ell^\frown) &= -1;\\ \delta(f_j[e']) &= 0;\\ \delta(e_1 \circ e_2) &= \delta(e_1) + \delta(e_2);\\ \delta(e_1 \cap e_2) = \delta(e_1 - e_2) &= \delta(e_1),\end{aligned}$$

with $f \in \{\pi, \overline{\pi}\}$ and $j \in \{1, 2\}$. A straightforward induction on the length of expressions yields $k = \delta(e)$. □

Using Lemma 3.12, we prove the following collapse for projection:

Proposition 3.13. *On labeled chains, we have* $\mathcal{N}(^\frown) \equiv_{\mathrm{path}} \mathcal{N}(^\frown, \pi)$.

Proof. Let e be a $\cup$-free expression in $\mathcal{N}(^\frown)$. We show that $e_1 = e \circ [e]^{-1}$ and $e_2 = [e]^{-1} \circ e$ are path-equivalent to $\pi_1[e]$ and $\pi_2[e]$, respectively. We only prove $e_1 \equiv_{\mathrm{path}} \pi_1[e]$; the proof for $e_2 \equiv_{\mathrm{path}} \pi_2[e]$ is analogous.

Observe that $\delta(e_1) = \delta(e) + \delta([e]^{-1}) = \delta(e) - \delta(e) = 0$. Hence, by Lemma 3.12, $(m, n) \in [\![e_1]\!]_C$ only if $m = n$. We have $(m, m) \in [\![e_1]\!]_C$ if and only if there exists a node z such that $(m, z) \in [\![e]\!]_C$ and $(z, m) \in [\![[e]^{-1}]\!]_C$. By definition of $[e]^{-1}$, we have $(m, z) \in [\![e]\!]_C$ if and only if $(z, m) \in [\![[e]^{-1}]\!]_C$. By definition of $\pi_1[e]$, there exists a node z' such that $(m, z') \in [\![e]\!]_C$ if and only if $(m, m) \in [\![\pi_1[e]]\!]_C$. Hence, we conclude $e_1 \equiv_{\mathrm{path}} \pi_1[e]$.

To rewrite arbitrary $\cup$-free expressions, we apply the above rewrite from projection to converse in a recursive fashion. To deal with expressions that are not $\cup$-free, we first apply Lemma 3.2 (i) and then rewrite each of the resulting $\cup$-free expressions. □

3.5 Adding diversity to local query languages

As already noticed, diversity always adds power to a local relation algebra fragment at the path level, as it can construct non-local relation algebra expressions. We can also use this property to our advantage to prove that diversity often adds expressive power at the Boolean level, too. To illustrate this, we strengthen Corollary 3.11 (ii):

Proposition 3.14. *Let* $\mathcal{F} \subseteq \{^\frown, \pi, \overline{\pi}, \cap, -\}$. *Already on labeled chains, we have* $\mathcal{N}(\mathrm{di}) \not\equiv_{\mathrm{bool}} \mathcal{N}(\mathcal{F})$.

Proof. Consider the expression $e = \ell' \circ \ell \circ \ell' \circ \mathrm{di} \circ \ell' \circ \ell \circ \ell \circ \ell'$ and the chains C, C_1, and C_2 of Figure 3.3. We have $[\![e]\!]_C \neq \emptyset$, $[\![e]\!]_{C_1} = \emptyset$, and $[\![e]\!]_{C_2} = \emptyset$, this independent of x. Now, assume there exists an expression e' in $\mathcal{N}(\mathcal{F})$ with $e \equiv_{\mathrm{bool}} e'$. By Proposition 3.10 (ii), e' is local. Hence, we know there exists $k \geq 0$ such that e' satisfies Definition 3.2. Consider the chains C, C_1, and C_2 of Figure 3.3 with $x > k$. Notice that every subchain of C containing all nodes at distance at most k from some given node is isomorphic to some subchain of either C_1 or C_2 of length at most $2k$. Hence, $(m, n) \in [\![e']\!]_C$ implies either $[\![e']\!]_{C_1} \neq \emptyset$, $[\![e']\!]_{C_2} \neq \emptyset$, or both, contradicting $e \equiv_{\mathrm{bool}} e'$. Hence, no expression in $\mathcal{N}(\mathcal{F})$ is Boolean-equivalent to e. □

Next, we use the above techniques to prove a similar result for unlabeled trees:

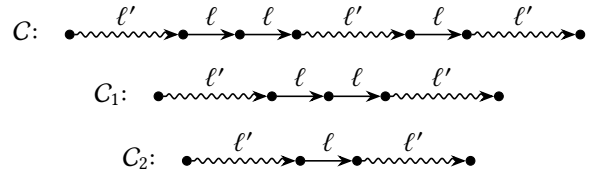


Figure 3.3: Chain C at the *top*, C_1 at the *middle*, and C_2 at the *bottom*. Chain C can be distinguished from C_1 and C_2 by detecting the presence of the patterns $\ell' \circ \ell \circ \ell'$ and $\ell' \circ \ell \circ \ell \circ \ell'$. The symbol $\rightsquigarrow$ represents a path of x edges, with x as in the proof of Proposition 3.14.

Figure 3.4: Tree $\mathcal{T}$ on the *left* and tree $\mathcal{T}'$ on the *right*. These trees can be distinguished by counting the number of non-root nodes that have two children. The symbol $\rightsquigarrow$ represents a path of x edges, with x as in the proof of Proposition 3.15.

Proposition 3.15. *Let $\mathcal{F} \subseteq \{^\frown, \pi, \overline{\pi}, \cap, -\}$. Already on unlabeled trees, we have $\mathcal{N}(\mathrm{di}, \cap) \not\equiv_{\mathrm{bool}} \mathcal{N}(\mathcal{F})$.*

Proof. Observe that $\underline{\{\mathrm{di}, \cap\}} = \{\mathrm{di}, \pi, \cap\}$ and consider the expression $e = P_{2,\neg r} \circ \mathrm{di} \circ P_{2,\neg r}$ with

$$\begin{aligned} P_{2,\neg r} &= \pi_2[\mathcal{E}] \circ P_2; \\ P_2 &= \pi_1[S_2]; \\ S_2 &= (\mathcal{E} \circ \mathrm{di}) \cap \mathcal{E}. \end{aligned}$$

The expression e is in $\mathcal{N}(\mathrm{di}, \pi, \cap)$ and selects node pairs among distinct non-root nodes such that each node in the pair has at least two distinct children. Now, assume there exists an expression e' in $\mathcal{N}(\mathcal{F})$ such that $e \equiv_{\mathrm{bool}} e'$. Since e' is local, we know there exists $k \geq 0$ such that e' satisfies Definition 3.2. Consider the trees $\mathcal{T}$ and $\mathcal{T}'$ of Figure 3.4 with $x = 2k$. Clearly, $[\![e]\!]_{\mathcal{T}} \neq \emptyset$ and $[\![e]\!]_{\mathcal{T}'} = \emptyset$. By $e \equiv_{\mathrm{bool}} e'$, we must have $[\![e']\!]_{\mathcal{T}} \neq \emptyset$. Let $(m, n) \in [\![e']\!]_{\mathcal{T}}$. Since every subtree of $\mathcal{T}$ containing all nodes at distance at most k from some given node is contained in a subtree of $\mathcal{T}$ that is isomorphic to $\mathcal{T}'$, we may conclude that $[\![e']\!]_{\mathcal{T}'} \neq \emptyset$. However, $[\![e]\!]_{\mathcal{T}'} = \emptyset$, contradicting $e \equiv_{\mathrm{bool}} e'$. Hence, no expression in $\mathcal{N}(\mathcal{F})$ is Boolean-equivalent to e. □

3.6 Non-local query languages on chains

The erratic behavior of diversity in the non-local relation algebra fragments (allowing one to jump from any node to any other node in a tree) makes studying the expressive power of these fragments inherently difficult. Luckily, we can obtain several separation results by studying these fragments on chains. The first set of separation results we prove use

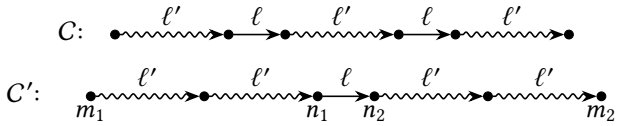


Figure 3.5: Chain C at the *top* and chain C' at the *bottom*. These chains can be distinguished by counting the number of edges labeled with ℓ. The symbol $\rightsquigarrow$ represents a path of x edges, with x as in the proof of Proposition 3.16.

locality-based arguments on local subexpressions of non-local expressions, which allows us to prove limits on the expressive power of query languages that utilize diversity:

Proposition 3.16. *Let $\mathcal{F} \subseteq \{\mathrm{di}, ^\frown, \pi, \overline{\pi}, \cap, -, ^*\}$ with $\{\mathrm{di}, ^\frown\} \subseteq \mathcal{F}$ or $\{\mathrm{di}, \pi\} \subseteq \mathcal{F}$. Already on labeled chains, we have $\mathcal{N}(\mathcal{F}) \not\equiv_{\mathrm{bool}} \mathcal{N}(\mathrm{di})$.*

Proof. Consider the path-equivalent expressions

$$e_1 = (\ell \circ \ell^\frown) \circ \mathrm{di} \circ (\ell \circ \ell^\frown);$$
$$e_2 = \pi_1[\ell] \circ \mathrm{di} \circ \pi_1[\ell].$$

The expression e_1 is in $\mathcal{N}(\mathrm{di}, ^\frown)$ and e_2 is in $\mathcal{N}(\mathrm{di}, \pi)$. Choose $e \in \{e_1, e_2\}$. On the chains C and C' of Figure 3.5 we have $[\![e]\!]_C \neq \emptyset$ and $[\![e]\!]_{C'} = \emptyset$, independent of x, which will be determined next.

Now, assume there exists an expression e' in $\mathcal{N}(\mathrm{di})$ such that $e \equiv_{\mathrm{bool}} e'$. By Lemma 3.2 (i) and Lemma 3.2 (iii), we can assume that e' is a union of $\{\mathrm{id}, \cup\}$-free expressions. Let S be the set of these $\{\mathrm{id}, \cup\}$-free expressions in e'. Choose x such that, for all $s \in S$, x is larger than the number of compositions in s.

Let $s \in S$ be an expression with $[\![s]\!]_C \neq \emptyset$. Split s into a sequence of di-free expressions $t_1, \ldots, t_k$ such that $s = t_1 \circ \mathrm{di} \circ \ldots \circ \mathrm{di} \circ t_k$. First, we show that every term t_i, $1 \leq j \leq k$, is of the form ℓ'^y, with $0 \leq y < x$, or $\ell'^{z_1} \circ \ell \circ \ell'^{z_2}$, with $0 \leq z_1 + z_2 < x$ and $0 \leq \min(z_1, z_2)$. We do so by contradiction. We only have to consider the existence of a subexpressions of the form $\ell \circ \ell'^z \circ \ell$. By construction, $z < x$. Hence, $[\![\ell \circ \ell'^z \circ \ell]\!]_C = \emptyset$ and $[\![s]\!]_C = \emptyset$, a contradiction.

By induction on the number of terms $t_1, \ldots, t_i$, $1 \leq i \leq k$, we shall prove that $[\![s]\!]_{C'} \neq \emptyset$. The base case is t_1. If t_1 is of the form ℓ'^y, then there exists a node m' in C' with $0 \leq \|m_1 \rightarrow m'\|_C = y < x$. Hence, $(m_1, m') \in [\![t_1]\!]_{C'}$. Else, if t_1 is of the form $\ell'^{z_1} \circ \ell \circ \ell'^{z_2}$, then there exists nodes n'_1 and n'_2 with $0 \leq \|n'_1 \rightarrow n_1\|_C = z_1 < x$ and $0 \leq \|n_2 \rightarrow n'_2\|_C = z_2 < x$. Hence, $(n'_1, n'_2) \in [\![t_1]\!]_{C'}$.

Now assume that, for all j, $1 \leq j < i$, there exists a pair of nodes v, w with $(v, w) \in [\![t_1 \circ \mathrm{di} \circ \ldots \circ \mathrm{di} \circ t_j]\!]_{C'}$ and either $0 \leq \|m_1 \rightarrow w\|_C < x$, $0 \leq \|n_2 \rightarrow w\|_C < x$, or $0 \leq \|w \rightarrow m_2\|_C < x$.

Next, we consider $t_1 \circ \mathrm{di} \circ \ldots \circ \mathrm{di} \circ t_i$. Let $(v, w) \in [\![t_1 \circ \mathrm{di} \circ \ldots \circ \mathrm{di} \circ t_{i-1}]\!]_{C'}$ such that v, w satisfy the statement of the induction hypothesis. Based on the form of t_i, we distinguish the following two cases:

1. If t_i is of the form ℓ'^y, then there exists nodes m'_1 and m'_2 with $0 \leq \|m_1 \rightarrow m'_1\|_C = y < x$ and $0 \leq \|m'_2 \rightarrow m_2\|_C = y < x$. Hence, $(m_1, m'_1) \in [\![t_1]\!]_{C'}$ and $(m'_2, m_2) \in [\![t_1]\!]_{C'}$. Due to the length of C, $m'_1 \neq m'_2$. Hence, $w \neq m'_1$, $w \neq m'_2$, or both. Pick $m' \in \{m'_1, m'_2\}$ such that

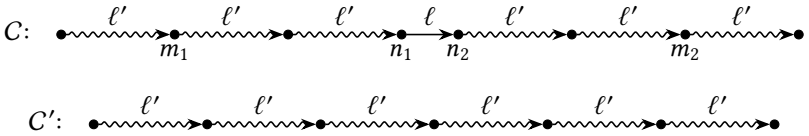


Figure 3.6: Chain C at the *top* and chain C' at the *bottom*; only one has an edge labeled ℓ. The symbol $\rightsquigarrow$ represents a path of x edges, with x as in the proof of Proposition 3.17.

$w \neq m'$. We conclude $(v, m') \in [\![t_1 \circ \mathrm{di} \circ \ldots \circ \mathrm{di} \circ t_i]\!]_{C'}$ and either $0 \leq \|m_1 \to m'_1\|_C < x$ or $0 \leq \|m'_2 \to m_2\|_C < x$.

2. Else, if t_i is of the form $\ell'^{z_1} \circ \ell \circ \ell'^{z_2}$, then there exists nodes n'_1 and n'_2 with $0 \leq \|n'_1 \to n_1\|_C = z_1 < x$ and $0 \leq \|n_2 \to n'_2\|_C = z_2 < x$. Hence, $(n'_1, n'_2) \in [\![t_i]\!]_{C'}$. Due to the length of C', we must have $n'_1 \neq w$ and we conclude $(v, n'_1) \in [\![t_1 \circ \mathrm{di} \circ \ldots \circ \mathrm{di} \circ t_i]\!]_{C'}$ and $0 \leq \|n_2 \to n'_2\|_C < x$.

We conclude $[\![e']\!]_{C'} \neq \emptyset$, a contradiction. Hence, no expression in $\mathcal{N}(\mathcal{F})$ is Boolean-equivalent to e. □

The above techniques can also be used with path queries, as shown next.

Proposition 3.17. *Let $\mathcal{F} \subseteq \{\mathrm{di}, ^\frown\}$. Already on labeled chains, we have $\mathcal{N}(\mathrm{di}, ^\frown, \pi) \not\equiv_{\mathrm{path}} \mathcal{N}(\mathcal{F})$.*

Proof. Consider the expression $e = \pi_1[\mathrm{all} \circ \ell]$ and chains C and C' of Figure 3.6. We have $[\![e]\!]_C = [\![\mathrm{id}]\!]_C$ and we have $[\![e]\!]_{C'} = \emptyset$.

Now, assume there exists an expression e' in $\mathcal{N}(\mathcal{F})$ such that $e \equiv_{\mathrm{path}} e'$. By Lemma 3.2 (i) and Lemma 3.2 (iii), we may assume that e' is a union of $\{\mathrm{id}, \cup\}$-free expressions. Let S be the set of these $\{\mathrm{id}, \cup\}$-free expressions in e'. Choose x such that, for all $s \in S$, x is larger than the number of compositions in s.

Let $s \in S$ be an expression with $(m_1, m_1) \in [\![s]\!]_C$. First, we prove by contradiction that s is not ℓ-free. Therefore, assume that s is ℓ-free and let s' be the expression obtained from s by replacing di by ℓ'. By construction, we have $[\![s']\!]_{C'} \subseteq [\![s]\!]_{C'}$. Observe that s' is a composition of ℓ' and $\ell'^\frown$ terms. Hence s' is local and, by construction, we have $-x < |\delta(s')| < x$. We conclude $[\![s']\!]_{C'} \neq \emptyset$, a contradiction. Hence, s is not ℓ-free.

Let $s = t_1 \circ \ldots \circ t_k$, with every t_i, $1 \leq i \leq k$, a term of the form ℓ, $\ell^\frown$, ℓ', $\ell'^\frown$, or di. Let t_j, $1 \leq j \leq k$, be the last term of the form ℓ or $\ell^\frown$. Observe that $\|m_1 \to n_1\|_C = x$ and $\|m_1 \to n_2\|_C = x + 1$. Hence, at least one of the terms t_i, $j < i \leq k$, must be di. Choose t_i, $j < i \leq k$, to be the first such term. Let $s'' = t_1 \circ \ldots \circ t_i \circ t'_{i+1} \circ t'_k$ be the expression obtained from s by replacing all occurrences of di in $t_{i+1} \circ \ldots \circ t_k$ by ℓ'. By construction, we have $[\![t'_{i+1} \circ \ldots \circ t'_k]\!]_C \subseteq [\![t_{i+1} \circ \ldots \circ t_k]\!]_C$. Since $(m_1, m_1) \in [\![s]\!]_C$, there exist nodes v and w such that $-x < \|n_1 \to v\|_C < x$, $-x < \|n_2 \to v\|_C < x$, $(m_1, v) \in [\![t_1 \circ \ldots \circ t_{i-1}]\!]_C$, $v \neq w$, and $(w, m_1) \in [\![t_{i+1} \circ \ldots \circ t_k]\!]_C$. Observe that $t'_{i+1} \circ \ldots \circ t'_k$ is a composition of ℓ' and $\ell'^\frown$ terms. Hence $t'_{i+1} \circ \ldots \circ t'_k$ is local and, by construction, we have $-x < |\delta(t'_{i+1} \circ \ldots \circ t'_k)| < x$. We conclude that there must exist a node m' with $-x < \|m' \to m_2\|_C = \delta(t'_{i+1} \circ \ldots \circ t'_k) < x$ and, hence, $(m', m_2) \in [\![s']\!]_{C'} \neq \emptyset$. Because of the length of C', we must have $v \neq m'$. We conclude $(m_1, m_2) \in [\![s]\!]_C$, a contradiction. Hence, no expression in $\mathcal{N}(\mathcal{F})$ is path-equivalent to e. □

Proposition 3.1 (ii) shows that diversity on chains can be expressed using the *ancestor axis*, expressed by $[\mathcal{E}^\frown]^+$, and the *descendant axis*, expressed by $[\mathcal{E}]^+$, which in turn allows us to simplify projection terms which contain diversity, as we show next.

Lemma 3.18. *Let $\mathcal{F} \subseteq \{\mathrm{di}, ^\frown, \pi\}$ and let $\pi_j[e], j \in \{1,2\}$, be an expression in $\mathcal{N}(\mathcal{F})$. There exists a finite set S of expressions of the form $\pi_1[\mathcal{E}^v] \circ \pi_2[\mathcal{E}^w]$, $v, w \geq 0$, such that $\pi_j[e] \equiv_{\mathrm{path}} \bigcup S$.*

Proof. By Proposition 3.1 (ii), we have $\mathrm{di} \equiv_{\mathrm{path}} [\mathcal{E}]^+ \cup [\mathcal{E}^\frown]^+$. We also have $\pi_2[e] \equiv_{\mathrm{path}} \pi_1[[e]^{-1}]$. Hence, every projection expression $\pi_j[e]$, $j \in \{1,2\}$, can be written as a union of expressions of the form $\pi_1[e']$ in which e' is build over the atoms id, $\mathcal{E}$, $\mathcal{E}^\frown$, $[\mathcal{E}]^+$, and $[\mathcal{E}^\frown]^+$, using the operators $\circ$ and π_1. We shall call such expressions e' *normal* in the remainder of this proof. So, it remains to show that Lemma 3.18 holds for expressions $\pi_1[e']$, with e' normal. We do this by structural induction on e'. We have the following base cases:

$$\pi_1[\mathrm{id}] \equiv_{\mathrm{path}} \mathrm{id} \equiv_{\mathrm{path}} \pi_1[\mathcal{E}^0] \circ \pi_2[\mathcal{E}^0];$$
$$\pi_1[\mathcal{E}] \equiv_{\mathrm{path}} \pi_1[[\mathcal{E}]^+] \equiv_{\mathrm{path}} \pi_1[\mathcal{E}^1] \circ \pi_2[\mathcal{E}^0];$$
$$\pi_1[\mathcal{E}^\frown] \equiv_{\mathrm{path}} \pi_1[[\mathcal{E}^\frown]^+] \equiv_{\mathrm{path}} \pi_1[\mathcal{E}^0] \circ \pi_2[\mathcal{E}^1].$$

Now, assume that Lemma 3.18 already holds for expressions $\pi_1[e'']$, with e'' a normal expression containing at most i operators, $i \geq 1$, and let $e = \pi_1[e']$ with e' a normal expression containing $i+1$ operators. Then either $e' = \pi_1[e'_1]$ or $e' = e'_1 \circ e'_2$, with e'_1 and e'_2 normal expressions containing at most i operators. In the first case, $e = \pi_1[\pi_1[e'_1]] \equiv_{\mathrm{path}} \pi_1[e'_1]$, and Lemma 3.18 holds for e by the induction hypothesis. In the second case, we have that $e = \pi_1[e'_1 \circ e'_2] \equiv_{\mathrm{path}} \pi_1[e'_1 \circ \pi_1[e'_2]]$. By the induction hypothesis, e'_2 is path-equivalent to a finite union of expressions of the form $\pi_1[\mathcal{E}^{v_2}] \circ \pi_2[\mathcal{E}^{w_2}]$, $v_2, w_2 \geq 0$. For e'_1, we distinguish again two cases:

1. Expression $e'_1 = \pi_1[e''_1]$, with e''_1 again a normal expression containing at most i operators. Hence, by the induction hypothesis, e'_1 is path-equivalent to a finite union of expressions of the form $\pi_1[\mathcal{E}^{v_1}] \circ \pi_2[\mathcal{E}^{w_1}]$, $v_1, w_1 \geq 0$. It now suffices to observe that

$$\pi_1[\mathcal{E}^{v_1}] \circ \pi_2[\mathcal{E}^{w_1}] \circ \pi_1[\mathcal{E}^{v_1}] \circ \pi_2[\mathcal{E}^{w_1}] \equiv_{\mathrm{path}} \pi_1[\mathcal{E}^{\max(v_1, v_2)}] \circ \pi_2[\mathcal{E}^{\max(w_1, w_2)}]$$

to see that Lemma 3.18 holds for e.

2. In the other case, we can assume without loss of generality that e'_1 is an atom. Hence, it suffices to observe that, for $v, w \geq 0$,

$$\pi_1[\mathrm{id} \circ (\pi_1[\mathcal{E}^v] \circ \pi_2[\mathcal{E}^w])] \equiv_{\mathrm{path}} \pi_1[\mathcal{E}^v] \circ \pi_2[\mathcal{E}^w];$$
$$\pi_1[\mathcal{E} \circ (\pi_1[\mathcal{E}^v] \circ \pi_2[\mathcal{E}^w])] \equiv_{\mathrm{path}} \pi_1[\mathcal{E}^{v+1}] \circ \pi_2[\mathcal{E}^{\max(0, w-1)}];$$
$$\pi_1[\mathcal{E}^\frown \circ (\pi_1[\mathcal{E}^v] \circ \pi_2[\mathcal{E}^w])] \equiv_{\mathrm{path}} \pi_1[\mathcal{E}^{\max(0, v-1)}] \circ \pi_2[\mathcal{E}^{w+1}];$$
$$\pi_1[[\mathcal{E}]^+ \circ (\pi_1[\mathcal{E}^v] \circ \pi_2[\mathcal{E}^w])] \equiv_{\mathrm{path}} \textstyle\bigcup_{1 \leq i \leq \max(1, w)} \pi_1[\mathcal{E}^{v+i}] \circ \pi_2[\mathcal{E}^{\max(0, w-i)}];$$
$$\pi_1[[\mathcal{E}^\frown]^+ \circ (\pi_1[\mathcal{E}^v] \circ \pi_2[\mathcal{E}^w])] \equiv_{\mathrm{path}} \textstyle\bigcup_{1 \leq i \leq \max(1, v)} \pi_1[\mathcal{E}^{\max(0, v-i)}] \circ \pi_2[\mathcal{E}^{w+i}],$$

to see that Lemma 3.18 holds for e in this case, too. □

Example 3.2. Consider the expression $e = \pi_1[\mathrm{di} \circ \mathcal{E} \circ \mathcal{E}]$. We have

$$\begin{aligned} e &\equiv_{\mathrm{path}} \pi_1[[\mathcal{E}]^+ \circ \pi_1[\mathcal{E}^2] \circ \pi_2[\mathcal{E}^0]] \cup \pi_1[[[\mathcal{E}]^{-1}]^+ \circ \pi_1[\mathcal{E}^2] \circ \pi_2[\mathcal{E}^0]] \\ &\equiv_{\mathrm{path}} \pi_1[\pi_1[\mathcal{E}^3] \circ \pi_2[\mathcal{E}^0]] \cup \pi_1[\pi_1[\mathcal{E}^1] \circ \pi_2[\mathcal{E}^1]] \cup \pi_1[\pi_1[\mathcal{E}^0] \circ \pi_2[\mathcal{E}^2]] \\ &\equiv_{\mathrm{path}} \pi_1[\mathcal{E}^3] \circ \pi_2[\mathcal{E}^0] \cup \pi_1[\mathcal{E}^1] \circ \pi_2[\mathcal{E}^1] \cup \pi_1[\mathcal{E}^0] \circ \pi_2[\mathcal{E}^2]. \end{aligned}$$

As Example 3.2 shows, we can use Lemma 3.18 to partially eliminate diversity from non-local expressions on unlabeled chains. Combining Lemma 3.18 and Proposition 3.13 yields the following collapse:

Proposition 3.19. *On unlabeled chains, we have* $\mathcal{N}(\mathrm{di}, ^\frown, \pi) \preceq_{\mathrm{path}} \mathcal{N}(\mathrm{di}, ^\frown)$.

Proof. We use Lemma 3.18 to rewrite all expressions of the form $\pi_j[e]$, $j \in \{1, 2\}$, to a path-equivalent finite union of expressions of the form $\pi_1[\mathcal{E}^v] \circ \pi_2[\mathcal{E}^w]$. We use Proposition 3.13 to rewrite $\pi_1[\mathcal{E}^v] \circ \pi_2[\mathcal{E}^w]$ to the path-equivalent expression $\mathcal{E}^v \circ \mathcal{E}^{-v} \circ \mathcal{E}^w \circ \mathcal{E}^{-w}$. □

Using Lemma 3.18, Proposition 3.1, and Lemma 3.3, we can also partially eliminate diversity from non-local expressions on chains, after which we can use additional distance-based arguments on subexpressions, as shown next.

Proposition 3.20. *Already on unlabeled chains, we have*

(i) $\mathcal{N}(\mathrm{di}, \cap) \not\preceq_{\mathrm{path}} \mathcal{N}(\mathrm{di}, ^\frown, \pi)$,

(ii) $\mathcal{N}(^\frown) \not\preceq_{\mathrm{path}} \mathcal{N}(\mathrm{di}, \pi, ^*)$.

Proof. First, we prove Statement (i). Consider the expression $e = (\mathrm{di} \circ \mathcal{E}) \cap \mathrm{di}$ in $\mathcal{N}(\mathrm{di}, \cap)$. On a chain, this expression yields all pairs of distinct non-root nodes that are not edges. Now, assume there exists an expression e' in $\mathcal{N}(\mathrm{di}, ^\frown, \pi)$ such that, on unlabeled chains, $e \equiv_{\mathrm{path}} e'$. Since e is non-local, e' must be non-local, too, hence, it must contain diversity. Using Lemma 3.18, we can rewrite e' into a union of expressions each of which is a composition of terms of the form id, di, $\mathcal{E}$, $\mathcal{E}^\frown$, $\pi_1[\mathcal{E}^v]$, or $\pi_2[\mathcal{E}^w]$, $v, w \geq 0$. Let $s = t_1 \circ \cdots \circ t_n$ be such an expression in which at least one term is diversity. By Proposition 3.1 (ii), s is path-equivalent to the infinite union

$$\bigcup_{k_1,\ldots,k_n \neq 0} t_{1,k_1} \circ \cdots \circ t_{n,k_n},$$

in which $t_{i,k_i} = t_i$ if $t_i \neq \mathrm{di}$ and $t_{i,k_i} = \mathcal{E}^{k_i}$ if $t_i = \mathrm{di}$, $1 \leq i \leq n$. Since s contains at least one diversity term, the set

$$\{\delta(t_{1,k_1} \circ \cdots \circ t_{n,k_n}) \mid k_1, \ldots, k_n \neq 0\}$$

covers all integer numbers with at most one exception (if s contains exactly one di term). We can therefore choose an expression $s' = t_{1,k_1} \circ \cdots \circ t_{n,k_n}$ for which $\delta(s') = 0$ or $\delta(s') = 1$. Observe that s' is a local expression in $\mathcal{N}(^\frown, \pi)$. Now, choose an unlabeled chain C which is sufficiently long to ensure that $[\![s']\!]_C \neq \emptyset$. Then, $[\![s']\!]_C$ contains either an identical node pair (if $\delta(s') = 0$) or an edge (if $\delta(s') = 1$), contradicting $e \equiv_{\mathrm{path}} e'$. Hence, no expression in $\mathcal{N}(\mathrm{di}, ^\frown, \pi)$ is path-equivalent to e.

Next, we prove Statement (ii) using a similar argument. Let $e = \mathcal{E}^\frown$. Assume there exists an expression e' in $\mathcal{N}(\mathrm{di}, \pi)$ such that, on unlabeled chains, $e \equiv_{\mathrm{path}} e'$. Using the analysis from the previous case, we rewrite e' into an infinite union of expressions of the form $t_{1,k_1} \circ \cdots \circ t_{n,k_n}$ with $k_1, \ldots, k_n \neq 0$. Again, choose an expression $s' = t_{1,k_1} \circ \cdots \circ t_{n,k_n}$ for which $\delta(s') = 0$ or $\delta(s') = 1$. Hence, we conclude that e' is not path-equivalent to e. To complete the proof for $\mathcal{N}(\mathrm{di}, ^\frown, \pi, ^*)$, we observe that every expression of the form $[g]^*$ is equivalent to an infinite union of expressions of the form g^k, $k \geq 0$. Hence, we simply replace every subexpression of the form $[g]^*$ by g^k, for some k, $k \geq 0$, and continue the above proof. □

3.7 Monotone queries and homomorphisms

Only coprojection and difference provide a form of negation. As a consequence, all fragments of $\mathcal{N}$ that do not include these two operators are *monotone*.

Definition 3.3. A query is *monotone* if, for every graph $\mathcal{G}$ and every graph $\mathcal{G}'$ obtained by adding nodes or edges to $\mathcal{G}$, we have $[\![e]\!]_{\mathcal{G}} \subseteq [\![e]\!]_{\mathcal{G}'}$.

A query language being monotone puts obvious limits on the expressive power. We can, however, show more powerful limitations by considering closure results under *homomorphisms*:

Definition 3.4. Let $\mathcal{G}_1 = (\mathcal{V}_1, \Sigma, \mathbf{E}_1)$ and $\mathcal{G}_2 = (\mathcal{V}_2, \Sigma, \mathbf{E}_2)$ be graphs, let F be a class of functions mapping $\mathcal{V}_1$ to $\mathcal{V}_2$, and let $\mathcal{L}$ be a query language. We say that $\mathcal{L}$ is *closed* under F if, for every query q in $\mathcal{L}$ and every $f \in F$, we have, $(m, n) \in [\![e]\!]_{\mathcal{G}_1}$ implies $(f(m), f(n)) \in [\![e]\!]_{\mathcal{G}_2}$. We say that a mapping $h : \mathcal{V}_1 \to \mathcal{V}_2$ is a *homomorphism* from $\mathcal{G}_1$ to $\mathcal{G}_2$ if, for every pair of nodes $m, n \in \mathcal{V}_1$ and every edge label $\ell \in \Sigma$, we have that $(m, n) \in \mathbf{E}_1(\ell)$ implies $(h(m), h(n)) \in \mathbf{E}_2(\ell)$. A homomorphism $h : \mathcal{V}_1 \to \mathcal{V}_2$ is called *injective* if, for all $m, n \in \mathcal{V}_1$, $n \neq m$ implies $h(n) \neq h(m)$.

Observe that diversity is a form of inequality. Hence, via a straightforward induction on the structure of expressions, we can show that the fragments of $\mathcal{N}$ that cannot express coprojection are closed under injective homomorphisms, whereas the languages that can express coprojections are not closed under injective homomorphisms. Additionally, we can show that the fragments of $\mathcal{N}$ that do not use diversity and cannot express coprojection are closed under homomorphisms.

Lemma 3.21. *Let $\mathcal{F} \subseteq \{\mathrm{di}, ^\frown, \pi, \cap, ^*\}$.*

(i) $\mathcal{N}(\mathcal{F})$ is closed under injective homomorphisms.

(ii) If $\mathrm{di} \notin \mathcal{F}$, then $\mathcal{N}(\mathcal{F})$ is closed under homomorphisms.

We use the above to show that the Boolean expressive power of the monotone query languages we consider is very limited.

Lemma 3.22. *Let $\mathcal{F} \subseteq \{\mathrm{di}, ^\frown, \pi, \cap, ^*\}$.*

(i) Let e be an expression in $\mathcal{N}(\mathcal{F})$. If $e \not\equiv_{\mathrm{path}} \emptyset$, then there exists a $k \geq 0$ such that $e \equiv_{\mathrm{bool}} \mathcal{E}^k$ on unlabeled chains.

(ii) Let e be an expression in $\mathcal{N}(\mathcal{F} - \{\mathrm{di}\})$. If $e \not\equiv_{\mathrm{path}} \emptyset$, then there exists a $k \geq 0$ such that $e \equiv_{\mathrm{bool}} \mathcal{E}^k$ on unlabeled trees.

Proof (sketch). (i) Let e be an expression in $\mathcal{N}(\mathcal{F})$, let C be the unlabeled chain with minimum depth for which $[\![e]\!]_C \neq \emptyset$, let C' be any unlabeled chain with $\mathrm{depth}(C') \geq \mathrm{depth}(C)$, and let r and r' be the root nodes of C and C', respectively. The function h that maps node z in C to the node z' in C' with $\|r \to z\|_C = \|r' \to z'\|_{C'}$ is an injective homomorphism. Hence, by Lemma 3.21, $[\![e]\!]_C \neq \emptyset$ implies $[\![e]\!]_{C'} \neq \emptyset$. Observe that we have $[\![\mathcal{E}^{\mathrm{depth}(C)}]\!]_C \neq \emptyset$ and chain C is the smallest chain for which this holds. As $\mathrm{depth}(C') \geq \mathrm{depth}(C)$, we also have $[\![\mathcal{E}^{\mathrm{depth}(C)}]\!]_{C'} \neq \emptyset$.

(ii) Let e be an expression in $\mathcal{N}(\mathcal{F} - \{\text{di}\})$ and let $\mathcal{T}$ be an unlabeled tree with $[\![e]\!]_{\mathcal{T}} \neq \emptyset$. Let C' be the unlabeled chain with $\text{depth}(\mathcal{T}) = \text{depth}(C)$ and let r and r' be the root nodes of $\mathcal{T}$ and C', respectively. The function h that maps every node z in $\mathcal{T}$ to the node z' in C' with $\|r \to z\|_{\mathcal{T}} = \|r' \to z'\|_C$ is a homomorphism. Hence, by Lemma 3.21, $[\![e]\!]_{\mathcal{T}} \neq \emptyset$ implies $[\![e]\!]_{C'} \neq \emptyset$. As e satisfies all conditions of Statement (i), we use Statement (i) to find the value k, $0 \leq k \leq \text{depth}(\mathcal{T})$, for which $e \equiv_{\text{bool}} \mathcal{E}^k$. □

Lemma 3.22 has significant consequences for the expressive power of Boolean queries in $\mathcal{N}(\text{di}, \frown, \pi, \cap, ^*)$ and $\mathcal{N}(\frown, \pi, \cap, ^*)$. In this light, we remark that the ability to count up-to-2 siblings in $\mathcal{N}(\text{di})$ and up-to-3 siblings in $\mathcal{N}(\text{di}, \cap)$, as demonstrated by Proposition 3.9, seem highly limited, e.g., for every expression in $\mathcal{N}(\text{di})$, we can construct a chain on which the expression evaluates to non-empty.

Corollary 3.23.

(i) On unlabeled chains, we have $\mathcal{N}(\text{di}, \frown, \pi, \cap, ^*) \leq_{\text{bool}} \mathcal{N}()$.

(ii) On unlabeled trees, we have $\mathcal{N}(\frown, \pi, \cap, ^*) \leq_{\text{bool}} \mathcal{N}()$.

Finally, we use Lemma 3.22 to conclude the following:

Theorem 3.24. *Already on unlabeled chains, we have* $\mathcal{N}(\overline{\pi}) \not\leq_{\text{bool}} \mathcal{N}(\text{di}, \frown, \pi, \cap, ^*)$.

Proof. Consider the expression $e = \overline{\pi}_2[\mathcal{E}] \circ \mathcal{E} \circ \overline{\pi}_1[\mathcal{E}]$ and the unlabeled chains C_1 and C_2 of depth 1 and 2, respectively. The expression e only evaluates to `True` if the queried chain has exactly a single edge. Hence, $[\![e]\!]_{C_1} \neq \emptyset$ and $[\![e]\!]_{C_2} = \emptyset$. Observe that for every expression of the form $\mathcal{E}^k$ with $[\![\mathcal{E}^k]\!]_{C_1} \neq \emptyset$, we also have $[\![\mathcal{E}^k]\!]_{C_2} \neq \emptyset$. Hence, no expression in $\mathcal{N}(\text{di}, \frown, \pi, \cap, ^*)$ is Boolean-equivalent to e. □

3.8 Adding the Kleene-star

It is well-known that Tarski's relation algebra ($\mathcal{N}(\text{di}, \frown, \pi, \overline{\pi}, \cap, -)$) is path-equivalent to FO[3] [36, 89]. For first-order logic, well-known bounds on their expressive power are known [70]. We shall use these results to prove that the Kleene-star usually adds expressive power when added to a relation algebra fragment. For path queries, we have the following:

Proposition 3.25. *Already on unlabeled chains, we have* $\mathcal{N}(^*) \not\leq_{\text{path}} \mathcal{N}(\text{di}, \frown, \pi, \overline{\pi}, \cap, -)$.

With respect to Boolean queries on unlabeled trees and chains, Theorem 3.23 already showed that Kleene-star does, in many cases, not add expressive power. Next, we show the cases in which Kleene-star does add expressive power to Boolean queries.

Proposition 3.26. *Let* $\mathcal{F} \subseteq \{\text{di}, \frown, \pi, \overline{\pi}, \cap, -\}$.

(i) Already on labeled chains, we have $\mathcal{N}(^*) \not\leq_{\text{bool}} \mathcal{N}(\mathcal{F})$.

(ii) Already on unlabeled chains, we have $\mathcal{N}(\overline{\pi}, ^*) \not\leq_{\text{bool}} \mathcal{N}(\mathcal{F})$.

(iii) Already on unlabeled trees, we have $\mathcal{N}(\text{di}, \cap, ^*) \not\leq_{\text{bool}} \mathcal{N}(\mathcal{F})$.

Proof. Using well-known results on the expressive power of first-order logic, we conclude that no expression in $\mathcal{N}(\mathcal{F})$ is Boolean-equivalent to $\ell_1 \circ [\ell_2 \circ \ell_2]^* \circ \ell_1$, to $\overline{\pi}_2[\mathcal{E}] \circ [\mathcal{E} \circ \mathcal{E}]^* \circ \overline{\pi}_1[\mathcal{E}]$, or to $X \circ [\mathcal{E} \circ \mathcal{E}]^* \circ X$, in which $X = (\mathcal{E} \circ \text{di}) \cap \mathcal{E}$ yields those edges from parent nodes to child nodes in which the parent also has other children. □

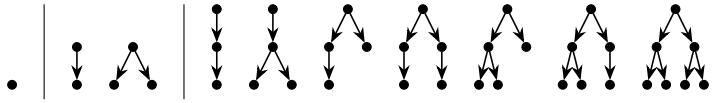


Figure 3.7: All non-2-subtree-reducible unlabeled trees of depth up-to-2.

The above does not cover Boolean queries on unlabeled trees expressed in $\mathcal{N}(\mathcal{F})$ with $\mathcal{F} \subseteq \{\text{di}, ^\frown, \pi, ^*\}$. For this case, we will show that we have $\mathcal{N}(\mathcal{F} \cup \{^*\}) \preceq_{\text{bool}} \mathcal{N}(\mathcal{F})$. To do so, we first use subtree-reductions to obtain limits on the the size of trees of a given depth:

Lemma 3.27. *Given a finite set of edge labels Σ and constants $d, k \geq 0$.*

(i) There exists only a finite number of trees $\mathcal{T}$ with $\text{depth}(\mathcal{T}) \leq d$ that are not k-subtree-reducible.

(ii) There exists a constant $w \geq 0$ such that, for every tree $\mathcal{T} = (\mathcal{V}, \Sigma, \mathbf{E})$ that is not k-subtree-reducible and with $\text{depth}(\mathcal{T}) \leq d$, we have $|\mathcal{V}| \leq w$.

Proof. We can enumerate all trees of depth at-most-d that are not k-subtree-reducible inductively, which proves the Lemma. The base case is $d = 0$, in which case we only have the tree with a single node. Now assume we have a set S_i of all trees of depth up-to-i, $i \geq 0$, and that these trees satisfy the Lemma. The biggest tree of depth $i + 1$ we can construct consists of a root node that has, per edge label $\ell \in \Sigma$ and per tree $\mathcal{T}$ in S_i, k outgoing edges labeled ℓ to children that are roots of copies of $\mathcal{T}$. All other trees of depth $i + 1$ that are not k-subtree-reducible are equivalent to a tree that can be constructed by taking a subset of the nodes and edges in this tree. □

Lemma 3.27 does not provide explicit upper bounds on the number of trees and the size of these trees of a given depth d. It is clear that these upper bounds grow at least exponentially fast with d, however.

Example 3.3. Figure 3.7 shows all unlabeled trees of depth up-to-2 that are not 2-subtree-reducible.

Next, we use Lemma 3.22 (i) and Lemma 3.27 to prove the claimed collapse:

Theorem 3.28. *Let $\mathcal{F} \subseteq \{\text{di}, ^\frown, \pi\}$. On unlabeled trees, we have $\mathcal{N}(\mathcal{F} \cup \{^*\}) \preceq_{\text{bool}} \mathcal{N}(\mathcal{F})$.*

Proof. Let e be an expression in $\mathcal{N}(\mathcal{F} \cup \{^*\})$ such that $e \not\equiv_{\text{path}} \emptyset$. By Lemma 3.22 (i), there exists a $k \geq 0$ such that $e \equiv_{\text{bool}} \mathcal{E}^k$ on unlabeled chains. Let $\mathcal{T}$ be a tree with $\text{depth}(\mathcal{T}) \geq k$ and let C be a chain with $\text{depth}(C) = k$. There exists an injective homomorphism from C to $\mathcal{T}$. Hence, we have $[\![\mathcal{E}^k]\!]_{\mathcal{T}} \neq \emptyset$ if and only if $[\![e]\!]_{\mathcal{T}} \neq \emptyset$. It remains to find an expression e' in $\mathcal{N}(\mathcal{F})$ such that, for every tree $\mathcal{T}'$ with $\text{depth}(\mathcal{T}') < k$, we have $[\![e]\!]_{\mathcal{T}'} \neq \emptyset$ if and only if $[\![e']\!]_{\mathcal{T}'} \neq \emptyset$. By Lemma 3.27 (ii), there exists a constant $w \geq 0$ that upper bounds the size of any tree of depth up-to-k that is not 2-subtree-reducible. Construct e' from e by replacing every subexpression of the form $[g]^*$ by $\bigcup_{0 \leq i \leq w} g^i$. Let $\mathcal{T}'$ be a tree with $\text{depth}(\mathcal{T}') < k$. We distinguish two cases:

1. *$\mathcal{T}'$ has at most w nodes.* In this case, $[\![e]\!]_{\mathcal{T}} = [\![e']\!]_{\mathcal{T}}$ by Lemma 3.3.

2. *$\mathcal{T}'$ has more than w nodes.* By Lemma 3.27 (ii), we can perform a sequence of 2-subtree-reduce steps on $\mathcal{T}'$ until we end up with a tree $\mathcal{T}''$ with at most w nodes. By Lemma 3.3, we have $[\![e]\!]_{\mathcal{T}''} = [\![e']\!]_{\mathcal{T}''}$. By Proposition 3.7 (ii), we have $[\![e']\!]_{\mathcal{T}''} \neq \emptyset$ if and only if $[\![e']\!]_{\mathcal{T}'} \neq \emptyset$. We conclude $[\![e']\!]_{\mathcal{T}'} \neq \emptyset$ if and only if $[\![e]\!]_{\mathcal{T}'} \neq \emptyset$.

We conclude $\mathcal{E}^k \cup e' \equiv_{\text{bool}} e$ and, by construction, $\mathcal{E}^k \cup e'$ is in $\mathcal{N}(\mathcal{F})$. □

3.9 First-order logic and the relation algebra

On simple structures such as chains, we can show that first-order logic (and, hence, also FO[3]) can only distinguish chains in very limited ways, which we show next.

Theorem 3.29. *On unlabeled chains, every first-order logic query is Boolean-equivalent to an expression in $\mathcal{N}(\overline{\pi})$.*

Proof. Let q be a first-order logic formula with quantifier rank r and let C_d be the chain with $\text{depth}(C) = d$. We construct e in $\mathcal{N}(\overline{\pi})$ as the union of the following terms:

(i) if $[\![q]\!]_{C_{2^r}} \neq \emptyset$, then include the term $\mathcal{E}^{2^r}$; and

(ii) for every $1 \leq i < 2^r$, if $[\![q]\!]_{C_i} \neq \emptyset$, then include the term $\overline{\pi}_2[\mathcal{E}] \circ \mathcal{E}^i \circ \overline{\pi}_1[\mathcal{E}]$.

To show that $q \equiv_{\text{bool}} e$, we show that, for every unlabeled chain C, we have $[\![q]\!]_C \neq \emptyset$ if and only if $[\![e]\!]_C \neq \emptyset$. We distinguish two cases:

1. $\text{depth}(C) \geq 2^r$. By construction, we have $[\![e]\!]_C \neq \emptyset$ if and only if the term $\mathcal{E}^{2^r}$ is included. By standard results on the expressive power of first-order logic [70, Theorem 3.6], we have $[\![q]\!]_C \neq \emptyset$ if and only if $[\![q]\!]_{C_{2^r}} \neq \emptyset$. Hence, in this case, we have $[\![e]\!]_C \neq \emptyset$ if and only if $[\![q]\!]_C \neq \emptyset$.

2. $\text{depth}(C) < i$. By construction, we have $[\![e]\!]_C \neq \emptyset$ if and only if the term $\overline{\pi}_2[\mathcal{E}] \circ \mathcal{E}^{\text{depth}(C)} \circ \overline{\pi}_1[\mathcal{E}]$ is included, which is the case if and only if $[\![q]\!]_C \neq \emptyset$. Hence, also in this case, we have $[\![e]\!]_C \neq \emptyset$ if and only if $[\![q]\!]_C \neq \emptyset$. □

As a consequence of the above, we have

Corollary 3.30. *Let $\mathcal{F} \subseteq \{\text{di}, ^\frown, \pi, \overline{\pi}, \cap, -\}$. Already on unlabeled chains, we have $\mathcal{N}(\mathcal{F}) \preceq_{\text{bool}} \mathcal{N}(\overline{\pi})$.*

As already mentioned, Theorem 8.6 of Part III will show that the language $\mathcal{N}(\text{di}, ^\frown, \pi, \overline{\pi})$ and its fragments are already Boolean-subsumed by FO[2] on graphs. Hence, for these languages, we can use well-known pebble games to reason about their expressive power [37, 39, 70]. We can use this to establish the following:

Proposition 3.31. *Let $\mathcal{F} \subseteq \{\text{di}, ^\frown, \pi, \overline{\pi}\}$. Already on labeled chains, we have $\mathcal{N}(\text{di}, \cap) \not\preceq_{\text{bool}} \mathcal{N}(\mathcal{F})$.*

Proof. Observe that $\pi \in \underline{\{\text{di}, \cap\}}$. Consider the expression

$$e = (\pi_1[\ell] \circ \text{di} \circ \pi_1[\ell] \circ \text{di} \circ \pi_1[\ell]) \cap \text{di}$$

in $\mathcal{N}(\text{di}, \cap)$. This query returns `True` on chains that have at least three edges labeled ℓ. Hence, on the chains C and C' of Figure 3.8, we have $[\![e]\!]_C \neq \emptyset$ and $[\![e]\!]_{C'} = \emptyset$, and this independent

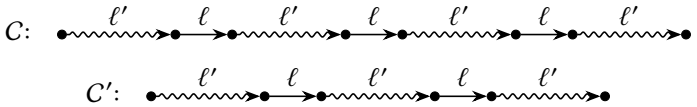


Figure 3.8: Chain C at the *top* and chain C' at the *bottom*. These chains can be distinguished by counting the number of edges labeled with ℓ. The symbol $\rightsquigarrow$ represents a path of x edges, with x as in the proof of Proposition 3.31.

of x. To show that no expression in $\mathcal{N}(\mathcal{F})$ can distinguish C from C', we show that no FO[2] formula of quantifier rank k can do so. To do this, we use k-round 2-pebble games [37, 39, 70]. Choose $x = 2k + 1$ and consider k-round 2-pebble games on C and C'. With two pebbles in play, the possible subgraph represented by the placement of two pebbles is either a graph with a single node (when the first pebble is placed), a graph with two disjoint nodes, a graph with a single edge labeled ℓ', or a graph with a single edge labeled ℓ.

Consider the i-th round, $1 \leq i \leq k$, in a two-pebble game of k rounds. We assume that the Spoiler played a pebble p at node m in chain $S \in \{C, C'\}$ that already holds a pebble q at node n, and the Duplicator needs to respond with a pebble p' at node m' in the other chain S' that already holds a pebble q' at node n'. The strategy for the Duplicator during this round is as follows:

1. If m is a parent of n, then choose node m' in S' such that m' is the parent of n'.
2. Else, if m is a child of n, then choose node m' in S' such that m' is the child of n'.
3. Else choose a node m' that is not the parent of n', not the child of n', and such that the subchain of S consisting of all nodes at distance at most $k - i$ from m is isomorphic to the subchain of S' consisting of all nodes at distance at most $k - i$ from m'.

For the placement of the first pebble, we use the third rule. It is straightforward to verify that the rules above always apply, proving that the Duplicator has a winning strategy. □

3.10 Brute-force results

Using a brute-force approach in the style of Fletcher et al. [32], we establish several separations, both at the path and Boolean levels. At the core of these brute-force results is the observation that one can effectively compute the set of query results obtainable by queries in some relation algebra fragment $\mathcal{N}(\mathcal{F})$, $\mathcal{F} \subseteq \{\mathrm{di}, \frown, \pi, \overline{\pi}, \cap, -\}$, on a given graph.[10] For path separations between languages $\mathcal{L}_1$ and $\mathcal{L}_2$, we may conclude that $\mathcal{L}_1 \not\leq_{\mathrm{path}} \mathcal{L}_2$ if there exists a query q in $\mathcal{L}_1$ and a tree $\mathcal{T}$ such that no query in $\mathcal{L}_2$ evaluates to $[\![q]\!]_{\mathcal{T}}$.

Proposition 3.32. *Let* $\mathcal{F}_1 \subseteq \{\mathrm{di}, \pi, \overline{\pi}, \cap, -, ^*\}$, $\mathcal{F}_2 \subseteq \{\mathrm{di}, \frown, ^*\}$, *and let* $\mathcal{F}_3 \subseteq \{\mathrm{di}, ^*\}$.

(i) Already on unlabeled trees, we have $\mathcal{N}(\frown) \not\leq_{\mathrm{path}} \mathcal{N}(\mathcal{F}_1)$.

[10] Originally, we used the brute-force implementation developed by Jan Van den Bussche, which was used to prove several results in Fletcher et al. [32]. During the course of our research, we implemented a new highly-optimized brute-force program from scratch. This new brute-force program was tested and adapted by Dimitri Surinx for his work on containment-based Boolean graph queries [86, 88] and Catherine L. Pilachowski for her work on the semi-join algebra. At a later moment, we developed a new prototype of the brute-force program that uses SIMD instructions to further improve performance. We refer to `http://jhellings.nl/projects/bruteforce/` for more information on the brute-force programs.

Figure 3.9: The unlabeled tree $\mathcal{T}$ and chain C used by the brute-force procedure to determine path separations.

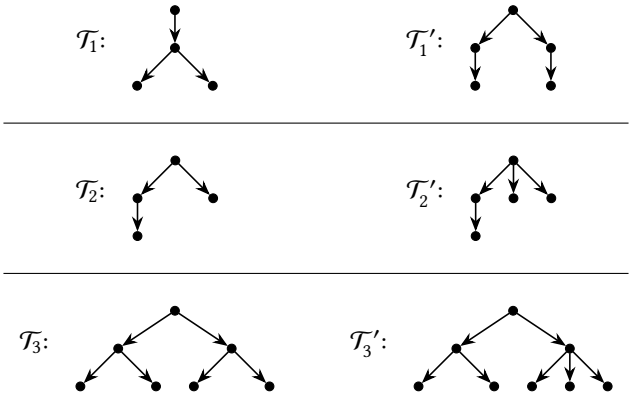


Figure 3.10: The pairs of unlabeled trees $(\mathcal{T}_1, \mathcal{T}_1')$, $(\mathcal{T}_2, \mathcal{T}_2')$, and $(\mathcal{T}_3, \mathcal{T}_3')$ used by the brute-force procedure to determine Boolean separations.

(ii) Already on unlabeled trees, we have $\mathcal{N}(\pi) \not\leq_{\text{path}} \mathcal{N}(\mathcal{F}_2)$.

(iii) Already on unlabeled chains, we have $\mathcal{N}(\pi) \not\leq_{\text{path}} \mathcal{N}(\mathcal{F}_3)$.

Proof. Let $\mathcal{T}$ and C be the tree and chain of Figure 3.9. Consider the expressions $e_1 = \mathcal{E}^\frown$, $e_2 = \pi_1[\mathcal{E}] \circ \pi_2[\mathcal{E}]$, and $e_3 = \pi_1[\mathcal{E} \circ \mathcal{E}]$. An exhaustive search shows that no expression in $\mathcal{N}(\mathcal{F}_1)$ evaluates to $[\![e_1]\!]_{\mathcal{T}}$, no expression in $\mathcal{N}(\mathcal{F}_2)$ evaluates to $[\![e_2]\!]_{\mathcal{T}}$, and no expression in $\mathcal{N}(\mathcal{F}_3)$ evaluates to $[\![e_3]\!]_C$. □

At the Boolean level, the key notion in the brute-force approach is the ability to *distinguish* a pair of trees. We say that a query q distinguishes a pair of trees $\mathcal{T}_1$ and $\mathcal{T}_2$ if $[\![q]\!]_{\mathcal{T}_1} = \emptyset$ and $[\![q]\!]_{\mathcal{T}_2} \neq \emptyset$, or vice versa. Given two languages $\mathcal{L}_1$ and $\mathcal{L}_2$, we may conclude that $\mathcal{L}_1 \not\leq_{\text{bool}} \mathcal{L}_2$ if we can find a query q in $\mathcal{L}_1$ and a pair of trees $\mathcal{T}_1$ and $\mathcal{T}_2$, indistinguishable by any query in $\mathcal{L}_2$, but distinguishable by q. Using this approach, we prove the following.

Proposition 3.33. *Let* $\mathcal{F}_1 \subseteq \{\text{di}, ^*\}$, $\mathcal{F}_2 \subseteq \{\text{di}, \frown, ^*\}$, *and* $\mathcal{F}_3 \subseteq \{\text{di}, \pi, \overline{\pi}, \cap, -, ^*\}$. *Already on unlabeled trees, we have*

(i) $\mathcal{N}(\text{di}, \frown) \not\leq_{\text{bool}} \mathcal{N}(\mathcal{F}_1)$;

(ii) $\mathcal{N}(\text{di}, \pi) \not\leq_{\text{bool}} \mathcal{N}(\mathcal{F}_2)$;

(iii) $\mathcal{N}(\text{di}, \frown, \cap) \not\leq_{\text{bool}} \mathcal{N}(\mathcal{F}_3)$.

Proof. Let $\mathcal{T}_1, \mathcal{T}_1', \mathcal{T}_2, \mathcal{T}_2', \mathcal{T}_3$, and $\mathcal{T}_3'$ be the trees of Figure 3.10. Consider the following three expressions:

$$e_1 = (\mathcal{E}^\frown \circ \mathcal{E}^\frown \circ \mathcal{E}) \circ \text{di} \circ (\mathcal{E}^\frown \circ \mathcal{E} \circ \mathcal{E});$$
$$e_2 = (\mathcal{E} \circ \mathcal{E} \circ \mathcal{E}^\frown) \circ \text{di} \circ \pi_1[\mathcal{E}^\frown \circ \mathcal{E} \circ \mathcal{E}] \circ \text{di} \circ \pi_1[\mathcal{E}^\frown \circ \mathcal{E} \circ \mathcal{E}] \circ \text{di} \circ (\mathcal{E} \circ \mathcal{E}^\frown \circ \mathcal{E}^\frown);$$

Table 3.1: Path equivalence relationships between language fragments of $\mathcal{X}_{r,[]}{}^{\uparrow}$ and relation algebra fragments.

Benedikt et al.	Fragment of $\mathcal{N}$
$\mathcal{X}_{r,[]}$	$\mathcal{N}(\hat{}, \pi_1)$
$\mathcal{X}_r$	$\mathcal{N}(\hat{})$
$\mathcal{X}_{[]}$	$\mathcal{N}(\pi_1)$
$\mathcal{X}$	$\mathcal{N}()$

$e_3 = (((\mathcal{E}^{\hat{}} \circ \mathcal{E}) \cap \mathrm{di}) \circ ((\mathcal{E}^{\hat{}} \circ \mathcal{E}) \cap \mathrm{di})) \cap \mathrm{di}$.

We have $[\![e_1]\!]_{\mathcal{T}_1} = \emptyset$ and $[\![e_1]\!]_{\mathcal{T}_1'} \neq \emptyset$. An exhaustive search shows that no expression in $\mathcal{N}(\mathcal{F}_1)$ can distinguish $\mathcal{T}_1$ from $\mathcal{T}_1'$. Observe that e_2 is in $\mathcal{N}(\mathrm{di}, \hat{}, \pi)$. By Proposition 3.34, e_2 is Boolean-equivalent to an expression in $\mathcal{N}(\mathrm{di}, \pi)$. An exhaustive search shows that no expression in $\mathcal{N}(\mathcal{F}_2)$ can distinguish $\mathcal{T}_2$ from $\mathcal{T}_2'$. We have $[\![e_3]\!]_{\mathcal{T}_2} = \emptyset$ and $[\![e_3]\!]_{\mathcal{T}_2'} \neq \emptyset$. Finally, an exhaustive search shows that no expression in $\mathcal{N}(\mathcal{F}_3)$ can distinguish $\mathcal{T}_3$ from $\mathcal{T}_3'$. □

3.11 Related work and results from the literature

On graphs, the relative expressive power of fragments of the relation algebra has been studied in great detail by Fletcher et al. [31–34, 87]. These results include path queries and Boolean queries on both labeled and unlabeled graphs, and can be summarized by

Proposition 3.34 (Fletcher et al.)**.** *Let* $\mathcal{F}, \mathcal{F}_1, \mathcal{F}_2 \subseteq \{\mathrm{di}, \hat{}, \pi, \overline{\pi}, \cap, -, {}^*\}$.

(i) On labeled and unlabeled graphs, we have $\mathcal{N}(\mathcal{F}_1) \preceq_{\mathrm{path}} \mathcal{N}(\mathcal{F}_2)$ *if and only if* $\underline{\mathcal{F}_1} \subseteq \underline{\mathcal{F}_2}$.

(ii) On labeled and unlabeled graphs, we have $\mathcal{N}(\mathcal{F} \cup \{\hat{}\}) \preceq_{\mathrm{bool}} \mathcal{N}(\mathcal{F} \cup \{\pi\})$ *if* $\mathcal{F} \subseteq \{\mathrm{di}, \hat{}, \pi, \overline{\pi}\}$.

(iii) On unlabeled graphs, we have $\mathcal{N}(\mathcal{F} \cup \{{}^*\}) \preceq_{\mathrm{bool}} \mathcal{N}(\mathcal{F})$ *if* $\mathcal{F} \subseteq \{\mathrm{di}, \pi\}$.

Of these results, only the collapse results (Proposition 3.34 (ii) and Proposition 3.34 (iii)) immediately apply to the tree data model. The collapse result of Proposition 3.34 (iii) can, on trees, be extended to also include the converse operator, as proven in Theorem 3.28.

A close inspection of the individual separation results of Fletcher et al. reveals that none of separation results directly apply to trees or chains, as the proofs of these separation results rely on non-tree graph structures. Consequently, our work improves on these results significantly by showing that many separations already hold on the much simpler tree and chain data models.

The XPath query language for XML data has several formalizations. The formalization of Benedikt et al. [9] includes a study of the relative expressive power of fragments of $\mathcal{X}_{[]}{}^{\uparrow}$ on node-labeled trees. The fragments of $\mathcal{X}_{[]}{}^{\uparrow}$ are closely related to the fragments of $\mathcal{N}(\hat{}, \pi)$. In Table 3.1, we provide the mapping between the query languages studied by Benedikt et al. and the query languages we study.

Additionally, the study of Benedikt et al. also considers fragments obtained by adding a weak form of the Kleene-star operator to the languages: the descendant axis $[\mathcal{E}]^*$ (and the ancestor axis $[\mathcal{E}^{\hat{}}]^*$ if $\hat{}$ is included in the fragment). Without much effort, we can show that

Table 3.2: Path-subsumption relationship between language fragments of Path^+ and relation algebra fragments.

Wu et al.	Fragment of $\mathcal{N}$
Path^+	$\mathcal{N}(\frown, \pi, \cap)$
$\text{Path}^+(\cap)$	$\mathcal{N}(\frown, \cap)$
$\text{Path}^+(\pi_1, \pi_2)$	$\mathcal{N}(\frown, \pi)$
$\text{DPath}^+(\pi_1)$	$\mathcal{N}(\pi_1)$

the results of Benedikt et al. on node-labeled trees also apply to the edge-labeled trees we study.

Proposition 3.35 (Benedikt et al. [9, Proposition 2.1])**.** *Let $\mathcal{F}_1, \mathcal{F}_2 \subseteq \{\frown, \pi, ^*\}$. Already on labeled trees, we have $\mathcal{N}(\mathcal{F}_1) \preceq_{\text{path}} \mathcal{N}(\mathcal{F}_2)$ if and only if $\mathcal{F}_1 \subseteq \mathcal{F}_2$.*

Wu et al. [97] studies the language Path^+ and its fragments on node-labeled trees. These languages are all path-equivalent to $\cup$-free fragments of $\mathcal{N}$. In Table 3.2, we provide the mapping between the query languages studied in Wu et al. and the query languages we study.

Among other things, Wu et al. showed that the query languages Path^+, $\text{Path}^+(\cap)$, and $\text{Path}^+(\pi_1, \pi_2)$ are path-equivalent on labeled trees. These results translate to our setting.

Proposition 3.36 (Wu et al. [97, Theorem 4.1])**.** *On labeled trees, every $\cup$-free expression in $\mathcal{N}(\frown, \pi, \cap)$ is path-equivalent to a $\cup$-free expression in $\mathcal{N}(\frown, \pi)$ and in $\mathcal{N}(\frown, \cap)$.*

Benedikt et al. and Wu et al. only studied fragments that are monotone and 1-subtree-reducible. From our results, we conclude that these fragments have only very limited expressive power, especially with respect to detecting branching structures in trees. Our work extends the results for these limited fragments to the more expressive full relation algebra, to Boolean queries, to unlabeled trees, and to chains.

Other XPath formalizations such as Conditional XPath, Regular XPath, and Regular $\text{XPath}^{\approx}$ [74, 75, 90, 91] study relationships between first-order logic and the relation algebra on sibling-ordered node-labeled trees. The choice of a sibling-ordered tree data model makes these studies incomparable with our work: on sibling-ordered trees, FO[3] is equivalent to general first-order logic and Conditional XPath is equivalent to FO[3] [74]. In our tree data model, the relation algebra, which is equivalent to FO[3], is *not* equivalent to first-order logic, and cannot express simple first-order counting queries such as

$$\exists n \exists c_1 \exists c_2 \exists c_3 \exists c_4 \; \mathcal{E}(n, c_1) \wedge \mathcal{E}(n, c_2) \wedge \mathcal{E}(n, c_3) \wedge \mathcal{E}(n, c_4) \wedge (c_1 \neq c_2) \wedge (c_1 \neq c_3) \wedge (c_1 \neq c_4) \wedge (c_2 \neq c_3) \wedge (c_2 \neq c_4) \wedge (c_3 \neq c_4),$$

which evaluates to `True` on all trees that have a node with at least four distinct children. With an ordered *sibling* axis, as present in the sibling-ordered tree data model, the above counting query is Boolean-equivalent to

$$\mathit{Right} \circ \mathit{Right} \circ \mathit{Right},$$

in which *Right* is the succeeding-sibling relation.

CHAPTER 4

Downward queries and condition automata[11]

In this chapter, we complete the study on the relative expressive power among the downward relation algebra fragments, the fragments $\mathcal{N}(\mathcal{F})$, $\mathcal{F} \subseteq \{\pi, \overline{\pi}, \cap, -, ^*\}$. In Figure 4.1, we visualize these relationships. In Chapter 3, we have already proven the cases in which we have separations. Hence, the only remaining results to prove are the cases in which we have redundancy of either intersection or difference. To prove these redundancies, we introduce *condition automata*, which are a generalization of the well-known finite automata [72].

4.1 Organization

First, in Section 4.2 we introduce the condition automata. Then, in Section 4.3, we show that specific fragments of the condition automata are path-equivalent to the fragments of the downward relation algebra that we study in this chapter. In Section 4.4, we develop techniques to make condition automata id-free (which resembles removing empty-string-transitions from finite automata) and use these id-free condition automata to show that, on labeled trees, condition automata are closed under intersection. In Section 4.5, we develop techniques to make condition automata deterministic (which resembles the translation from non-deterministic to deterministic finite automata) and use these deterministic condition automata to show that, on labeled trees, condition automata are closed under difference. In Section 4.6, we use these closure results to prove the cases in which either intersection or difference are redundant. Finally, in Section 4.7, we use the condition automata to show the redundancy of projection in Boolean queries on chains.

4.2 Condition automata

Observe that queries in $\mathcal{N}(^*)$ select pairs of nodes m, n such that there exists a directed path from m to n whose labeling satisfies some regular expression. In the case of trees, this directed path is unique, which yields a strong connection between $\mathcal{N}(^*)$ and the closure results under intersection and difference for regular languages [72]. As a consequence, we can show, in a relatively straightforward way, that $\mathcal{N}(\cap, -, ^*) \preceq_{\text{path}} \mathcal{N}(^*)$.

Example 4.1. We can rewrite the expressions $[\ell^3]^+ \cap [\ell^7]^+$ and $[\ell^3]^+ - [\ell^7]^+$ to path-equivalent

[11] The results in this chapter are based on the paper "Relative expressive power of downward fragments of navigational query languages on trees and chains" [52, 53].

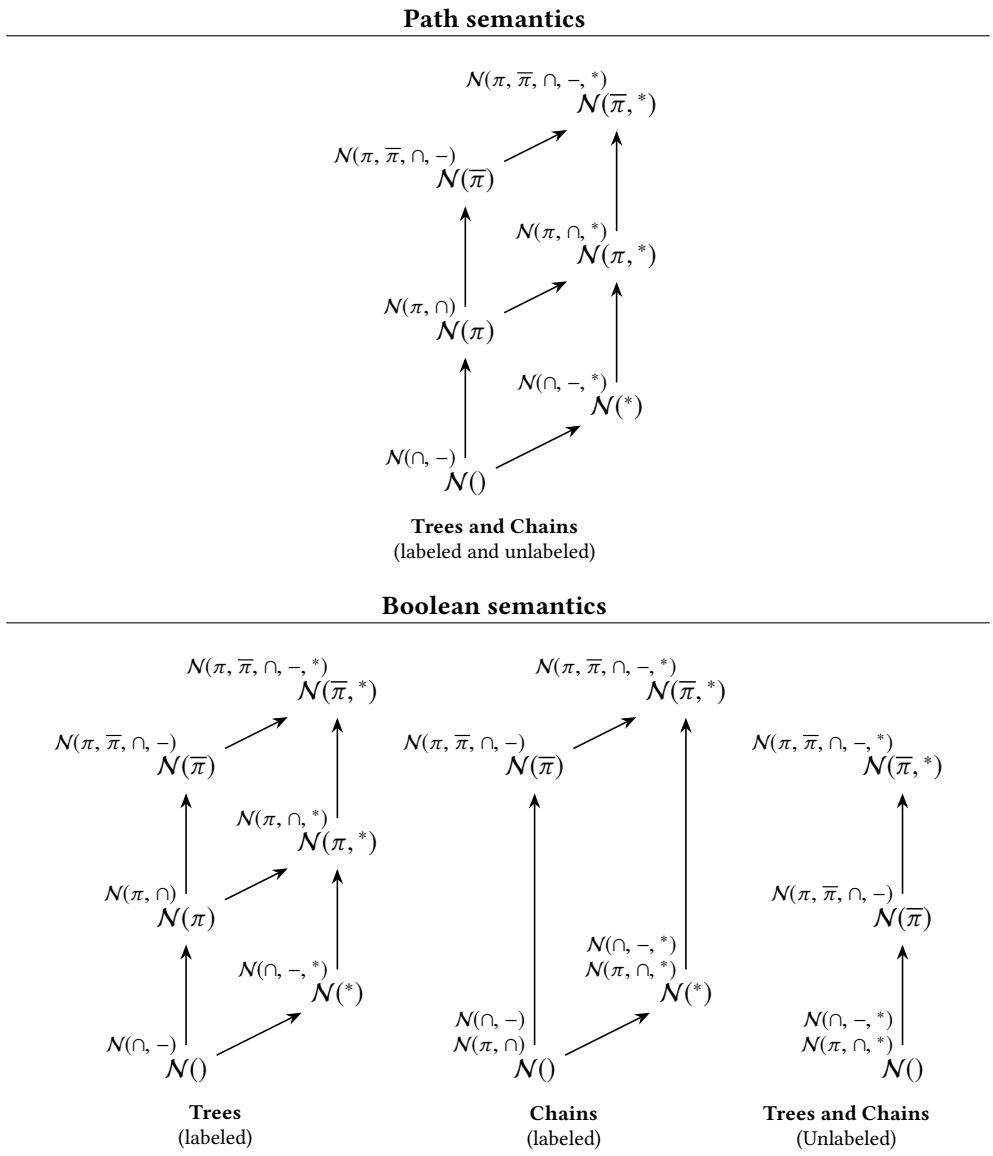


Figure 4.1: Hasse diagrams that visualize the relationships between the downward fragments of the relation algebra ($\mathcal{N}(\pi, \overline{\pi}, \cap, -.^*)$). Each node represents a minimally sized fragment and the superscripts on the left-hand side represent all maximally sized fragments that are equivalent to the fragment represented by the node. Arrows represent strict subsumption relations.

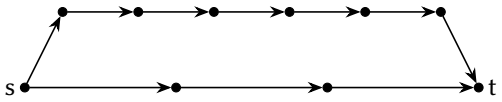


Figure 4.2: An acyclic directed graph, which is not a chain, a tree, or a forest. Observe that $\ell^3 \cap \ell^7$ returns the node pair (s, t).

expressions that use neither intersection nor difference:

$$[\ell^3]^+ \cap [\ell^7]^+ = [\ell^{21}]^+;$$
$$[\ell^3]^+ - [\ell^7]^+ = \left(\ell^3 \cup \ell^6 \cup \ell^9 \cup \ell^{12} \cup \ell^{15} \cup \ell^{18}\right) \circ [\ell^{21}]^*.$$

Notice that this rewriting does not work on arbitrary graphs. Indeed, on the graph $\mathcal{G}$ in Figure 4.2, we have $[\![[\ell^3]^+ \cap [\ell^7]^+]\!]_{\mathcal{G}} \neq \emptyset$, whereas $[\![[\ell^{21}]^+]\!]_{\mathcal{G}} = \emptyset$.

For regular expressions, the closure results under intersection and difference are usually obtained by first proving that regular expressions have the same expressive power as finite automata, and then proving that finite automata are closed under intersection and difference. We extend these automata-based techniques to the languages $\mathcal{N}(\mathcal{F})$ with $\mathcal{F} \subseteq \{\pi, \overline{\pi}, ^*\}$ by introducing *conditions* on automaton states. We use these extended automata to prove that, on trees, $\mathcal{N}(\mathcal{F})$ is closed under intersection and $\mathcal{N}(\underline{\mathcal{F}})$ is closed under difference. The *conditions* we consider are node expressions of the form $\emptyset$, id, $\pi_1[e]$, $\pi_2[e]$, $\overline{\pi}_1[e]$, or $\overline{\pi}_2[e]$.

Definition 4.1. A *condition automaton* is a 7-tuple $\mathcal{A} = (S, \Sigma, C, I, F, \delta, \gamma)$, where S is a set of states, Σ a set of transition labels, C a set of node expressions, $I \subseteq S$ a set of initial states, $F \subseteq S$ a set of final states, $\delta \subseteq S \times (\Sigma \cup \{\mathrm{id}\}) \times S$ the transition relation, and $\gamma \subseteq S \times C$ the state-condition relation. For a state $q \in S$, we denote $\gamma(q) = \{c \mid (q, c) \in \gamma\}$.

Let $\mathcal{F} \subseteq \{^*, \pi, \overline{\pi}\}$. We say that $\mathcal{A}$ is $\mathcal{F}$*-free* if every condition in C is an expression in $\mathcal{N}(\{\pi, \overline{\pi}, ^*\} - \mathcal{F})$, we say that $\mathcal{A}$ is *acyclic* if the transition relation δ of $\mathcal{A}$ is acyclic (viewed as a labeled graph relation), and we say that $\mathcal{A}$ is id-*transition-free* if $\delta \subseteq S \times \Sigma \times S$.

Example 4.2. Consider the condition automaton $\mathcal{A} = (S, \Sigma, C, I, F, \delta, \gamma)$ with

$$S = \{q_1, q_2, q_3, q_4\};$$
$$\Sigma = \{\ell_1, \ell_2, \ell_3\};$$
$$C = \{\mathrm{id}, \pi_2[{\ell_1}^2], \pi_1[{\ell_2}^3]\};$$
$$I = \{q_1, q_4\};$$
$$F = \{q_3, q_4\};$$
$$\delta = \{(q_1, \ell_1, q_2), (q_1, \ell_3, q_4), (q_2, \ell_1, q_2), (q_2, \ell_2, q_3)\};$$
$$\gamma = \{(q_1, \mathrm{id}), (q_2, \pi_1[{\ell_1}^2]), (q_2, \pi_2[{\ell_2}^3])\}.$$

This automaton is visualized in Figure 4.3. Using this visualization, it is easy to verify that the condition automaton is not acyclic (due to the ℓ_1 labeled self-loop), is $\{\overline{\pi}, ^*\}$-free, and is id-transition-free.

Observe that condition automata are strongly related to finite automata, the main difference being that states in the automata have a set of conditions. In the evaluation of condition automata on trees, this set of conditions determines in which tree nodes a state can hold, which we define next.

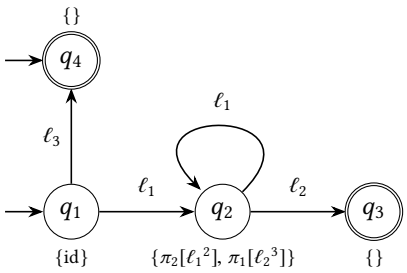


Figure 4.3: An example of a condition automaton.

Definition 4.2. Let $\mathcal{G} = (\mathcal{V}, \Sigma, \mathbf{E})$ be a graph, let $\mathcal{A} = (S, \Sigma, C, I, F, \delta, \gamma)$ be a condition automaton, and let $q \in S$. We define the *condition expression* $\mathsf{conditions}(q)$ of q by

$$\mathsf{conditions}(q) = \bigcap_{e \in \gamma(q)} e$$

when $\gamma(q) \neq \emptyset$ and $\mathsf{conditions}(q) = \emptyset$ otherwise. As every expression in $\gamma(q)$ is a condition, the expression $\mathsf{conditions}(q)$ is a node expression. Observe that if $\gamma(q) = \{e_1, \ldots, e_k\}$, then $\mathsf{conditions}(q) \equiv_{\text{path}} e_1 \circ \ldots \circ e_k$. We usually assume that $\mathsf{conditions}(q)$ is written as a composition of terms instead of an intersection of terms.

We say that a node $n \in \mathcal{V}$ *satisfies* state $q \in S$ if $(n, n) \in [\![\mathsf{conditions}(q)]\!]_{\mathcal{G}}$. A *run* of $\mathcal{A}$ on $\mathcal{G}$ is a sequence $(q_0, n_0)\ \ell_0\ (q_1, n_1)\ \ell_1 \ldots (q_{i-1}, n_{i-1})\ \ell_{i-1}\ (q_i, n_i)$, where $q_0, \ldots, q_i \in S$, $n_0, \ldots, n_i \in \mathcal{V}$, $\ell_0, \ldots, \ell_{i-1} \in \Sigma \cup \{\mathrm{id}\}$, and the following conditions hold:

1. for all $0 \leq j \leq i$, n_j satisfies q_j;
2. for all $0 \leq j < i$, $(q_j, \ell_j, q_{j+1}) \in \delta$; and
3. for all $0 \leq j < i$, $(n_j, n_{j+1}) \in [\![\ell_j]\!]_{\mathcal{G}}$.

We say that $\mathcal{A}$ *accepts* node pair $(m, n) \in \mathcal{V} \times \mathcal{V}$ if there exists a run $(q_0, m)\ \ell_0 \ldots (q_i, n)$ of $\mathcal{A}$ on $\mathcal{G}$ with $q_0 \in I$ and $q_i \in F$. We define the *evaluation* of $\mathcal{A}$ on $\mathcal{G}$, denoted by $[\![\mathcal{A}]\!]_{\mathcal{G}}$, as $[\![\mathcal{A}]\!]_{\mathcal{G}} = \{(m, n) \mid \mathcal{A} \text{ accepts } (m, n)\}$.

Example 4.3. Consider the condition automaton of Example 4.2, shown in Figure 4.3, and the tree $\mathcal{T}$ shown in Figure 4.4. For this combination of a condition automaton and a tree, we can construct several accepting runs. Examples are the run $(q_1, r)\ \ell_3\ (q_4, m)$, which semantically implies

$$(r, m) \in [\![\mathsf{conditions}(q_1) \circ \ell_3 \circ \mathsf{conditions}(q_4)]\!]_{\mathcal{T}} = [\![\mathrm{id} \circ \ell_3 \circ \mathrm{id}]\!]_{\mathcal{T}},$$

and $(q_1, n_1)\ \ell_1\ (q_2, n_2)\ \ell_1\ (q_2, n_3)\ \ell_2\ (q_3, n_4)$, which semantically implies

$$\begin{aligned}(n_1, n_4) \in [\![&\mathsf{conditions}(q_1) \circ \ell_1 \circ \mathsf{conditions}(q_2) \circ \ell_1 \circ \\ &\mathsf{conditions}(q_2) \circ \ell_2 \circ \mathsf{conditions}(q_3)]\!]_{\mathcal{T}} = \\ &[\![\mathrm{id} \circ \ell_1 \circ \pi_2[{\ell_1}^2] \circ \pi_1[{\ell_2}^3] \circ \ell_1 \circ \pi_2[{\ell_1}^2] \circ \pi_1[{\ell_2}^3] \circ \ell_2 \circ \mathrm{id}]\!]_{\mathcal{T}}.\end{aligned}$$

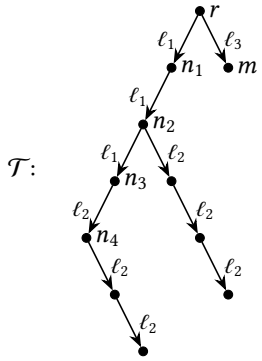


Figure 4.4: A labeled tree in which six distinct nodes are named.

Table 4.1: Relation algebra fragments and the corresponding classes of condition automata.

Relation algebra	Class of condition automata
$\mathcal{N}()$	$\{^*, \pi, \overline{\pi}\}$-free and acyclic.
$\mathcal{N}(\pi)$	$\{^*, \overline{\pi}\}$-free and acyclic.
$\mathcal{N}(\pi, \overline{\pi})$	$\{^*\}$-free and acyclic.
$\mathcal{N}(^*)$	$\{\pi, \overline{\pi}\}$-free.
$\mathcal{N}(\pi, ^*)$	$\{\overline{\pi}\}$-free.
$\mathcal{N}(\pi, \overline{\pi}, ^*)$	no restrictions.

4.3 Condition automata and downward queries

Our first goal is to show the path equivalence of $\mathcal{N}(\mathcal{F})$, $\mathcal{F} \subseteq \{\pi, \overline{\pi}, ^*\}$, with a restricted class of condition automata, as proposed in Table 4.1.

Example 4.4. Consider the condition automaton of Example 4.2, shown in Figure 4.3. By carefully examining the automaton, one can conclude that it is path-equivalent to the expression $\ell_1 \circ \pi_2[{\ell_1}^2] \circ \pi_1[{\ell_2}^3] \circ [\ell_1 \circ \pi_2[{\ell_1}^2] \circ \pi_1[{\ell_2}^3]]^* \circ \ell_2 \cup \ell_3 \cup \mathrm{id}$.

To show the path equivalence proposed in Table 4.1, we first adapt standard closure properties for finite automata under composition, union, and Kleene-plus to the setting of condition automata:

Proposition 4.1. *Let $\mathcal{F} \subseteq \{\pi, \overline{\pi}, ^*\}$ and let $\mathcal{A}_1$ and $\mathcal{A}_2$ be $\mathcal{F}$-free condition automata. There exists $\mathcal{F}$-free condition automata $\mathcal{A}_\circ$, $\mathcal{A}_\cup$, and $\mathcal{A}^+$ such that, for every graph $\mathcal{G}$, $[\![\mathcal{A}_\circ]\!]_\mathcal{G} = [\![\mathcal{A}_1]\!]_\mathcal{G} \circ [\![\mathcal{A}_2]\!]_\mathcal{G}$, $[\![\mathcal{A}_\cup]\!]_\mathcal{G} = [\![\mathcal{A}_1]\!]_\mathcal{G} \cup [\![\mathcal{A}_2]\!]_\mathcal{G}$, and $[\![\mathcal{A}^+]\!]_\mathcal{G} = [[\![\mathcal{A}_1]\!]_\mathcal{G}]^+$. The condition automata $\mathcal{A}_\circ$ and $\mathcal{A}_\cup$ are acyclic whenever $\mathcal{A}_1$ and $\mathcal{A}_2$ are acyclic.*

Proof. Let $\mathcal{A}_1 = (S_1, \Sigma_1, C_1, I_1, F_1, \delta_1, \gamma_1)$ and $\mathcal{A}_2 = (S_2, \Sigma_2, C_2, I_2, F_2, \delta_2, \gamma_2)$ be $\mathcal{F}$-free condition automata. Without loss of generality, we may assume that $S_1 \cap S_2 = \emptyset$. We define $\mathcal{A}_\circ$, $\mathcal{A}_\cup$, and $\mathcal{A}^+$ as follows:

1. $\mathcal{A}_\circ = (S_1 \cup S_2, \Sigma_1 \cup \Sigma_2, C_1 \cup C_2, I_1, F_2, \delta_1 \cup \delta_2 \cup \delta_\circ, \gamma_1 \cup \gamma_2)$, in which $\delta_\circ = \{(q_1, \mathrm{id}, q_2) \mid (q_1 \in F_1) \wedge (q_2 \in I_2)\}$.

2. $\mathcal{A}_\cup = (S_1 \cup S_2, \Sigma_1 \cup \Sigma_2, C_1 \cup C_2, I_1 \cup I_2, F_1 \cup F_2, \delta_1 \cup \delta_2, \gamma_1 \cup \gamma_2)$.

Table 4.2: Basic building blocks used by the translation from expressions to condition automata. In the table, ℓ is an edge label.

e	Condition automaton
$\emptyset$	$\mathcal{A} = (\{v, w\}, \Sigma, \emptyset, \{v\}, \{w\}, \emptyset, \emptyset)$
id	$\mathcal{A} = (\{v, w\}, \Sigma, \emptyset, \{v\}, \{w\}, \{(v, \mathrm{id}, w)\}, \emptyset)$
ℓ	$\mathcal{A} = (\{v, w\}, \Sigma, \emptyset, \{v\}, \{w\}, \{(v, \ell, w)\}, \emptyset)$
$\pi_1[e']$	$\mathcal{A} = (\{v\}, \Sigma, \{\pi_1[e']\}, \{v\}, \{v\}, \emptyset, \{(v, \pi_1[e'])\})$
$\pi_2[e']$	$\mathcal{A} = (\{v\}, \Sigma, \{\pi_2[e']\}, \{v\}, \{v\}, \emptyset, \{(v, \pi_2[e'])\})$
$\overline{\pi}_1[e']$	$\mathcal{A} = (\{v\}, \Sigma, \{\overline{\pi}_1[e']\}, \{v\}, \{v\}, \emptyset, \{(v, \overline{\pi}_1[e'])\})$
$\overline{\pi}_2[e']$	$\mathcal{A} = (\{v\}, \Sigma, \{\overline{\pi}_2[e']\}, \{v\}, \{v\}, \emptyset, \{(v, \overline{\pi}_2[e'])\})$

3. $\mathcal{A}^+ = (S_1 \cup \{v, w\}, \Sigma_1, C_1, \{v\}, \{w\}, \delta_1 \cup \delta^+, \gamma_1)$, in which $v, w \notin S_1$ are two distinct fresh states and $\delta^+ = \{(v, \mathrm{id}, q) \mid q \in I_1\} \cup \{(q, \mathrm{id}, w) \mid q \in F_1\} \cup \{(w, \mathrm{id}, v)\}$.

Observe that we did not add new condition expressions to the set of condition expressions in the proposed constructions. Hence, we conclude that $\mathcal{A}_\circ$, $\mathcal{A}_\cup$, and $\mathcal{A}^+$ are $\mathcal{F}$-free whenever $\mathcal{A}_1$ and $\mathcal{A}_2$ are $\mathcal{F}$-free. In $\mathcal{A}_\circ$ and $\mathcal{A}_\cup$, no new loops have been introduced, and, hence, $\mathcal{A}_\circ$ and $\mathcal{A}_\cup$ are acyclic whenever $\mathcal{A}_1$ and $\mathcal{A}_2$ are acyclic. It is straightforward to see that $\mathcal{A}_\circ$, $\mathcal{A}_\cup$, and $\mathcal{A}^+$ satisfy the other requirements of the Proposition. □

We can translate relation algebra expressions to condition automata in a recursive manner. The base cases are expressions that are atoms or node expressions, and are described in Table 4.2. The recursive cases are compositions, unions, and transitive closures, for which we use the closure results of Proposition 4.1 in a straightforward manner. We conclude

Proposition 4.2. *Let $\mathcal{F} \subseteq \{\pi, \overline{\pi}, ^*\}$. On labeled graphs, every expression in $\mathcal{N}(\mathcal{F})$ is path-equivalent to some condition automaton in the class specified for $\mathcal{N}(\mathcal{F})$ in Table 4.1.*

Finally, to show the other direction, we adapt the translation of finite automata to regular expressions to the setting of condition automata.

Proposition 4.3. *Let $\mathcal{F} \subseteq \{\pi, \overline{\pi}, ^*\}$. On labeled graphs, every condition automaton in the class specified for $\mathcal{N}(\mathcal{F})$ in Table 4.1 is path-equivalent to some expression in $\mathcal{N}(\mathcal{F})$.*

Proof. Let $\mathcal{A} = (S, \Sigma, C, I, F, \delta, \gamma)$ be a condition automaton. Let $v, w \notin S$ be two distinct fresh states. Let $\mathcal{A}' = (S \cup \{v, w\}, \Sigma, C, \{v\}, \{w\}, \delta \cup \delta_{v,w}, \gamma)$ with $\delta_{v,w} = \{(v, \mathrm{id}, q) \mid q \in I\} \cup \{(q, \mathrm{id}, w) \mid q \in F\}$ be a condition automaton path-equivalent to $\mathcal{A}$ which has only one initial state and one final state. We translate $\mathcal{A}'$ into an expression using the Algorithm ToExpression presented in Figure 4.5.

Let $\mathcal{G} = (\mathcal{V}, \Sigma, \mathbf{E})$ be a graph. We prove that the expression $e_{v,w}$ returned by Algorithm ToExpression is path-equivalent to $\mathcal{A}'$. We do so by proving the following invariants of the Algorithm:

1. *Let $q_1, q_2 \in S \cup \{v, w\}$. If no path exists from state q_1 to state q_2 with at least a single transition, then $e_{q_1,q_2} \equiv_{\mathrm{path}} \emptyset$.*

If there exists no path from q_1 to q_2, then also no transition exists from q_1 to q_2. Hence, we initialize $e_{q_1,q_2} = \emptyset$. After initialization, the value of e_{q_1,q_2} only changes at Line 7. As there exists no path from q_1 to q_2, we have one of the following three cases:

Algorithm ToExpression($\mathcal{A} = (S, \Sigma, C, I, F, \delta, \gamma)$)**:**

1: $M[q] \coloneqq \texttt{True}$, $q \in S \cup \{v, w\}$
2: **for all** $q \in S \cup \{v, w\}$ and $r \in S \cup \{v, w\}$ **do**
3: If there are no transitions between q and r, then $e_{q,r} \coloneqq \emptyset$. Otherwise

$$e_{q,r} \coloneqq \bigcup_{(q,\ell,r)\in\delta\cup\delta_{v,w}} \text{conditions}(q) \circ \ell \circ \text{conditions}(r)$$

4: **while** $\exists q\,(q \in S) \wedge (M[q] = \texttt{True})$ **do**
5: Choose q with $(q \in S) \wedge (M[q] = \texttt{True})$
6: **for** $p_1, p_2 \in S \cup \{v, w\}$ with $q \notin \{p_1, p_2\}$ **do**
7: $e_{p_1,p_2} \coloneqq e_{p_1,p_2} \cup e_{p_1,q} \circ [e_{q,q}]^* \circ e_{q,p_2}$
8: If applicable, remove $\emptyset$ from e_{p_1,p_2} or reduce e_{p_1,p_2} to $\emptyset$
9: $M[q] \coloneqq \texttt{False}$
10: **return** $e_{v,w}$

Figure 4.5: Algorithm ToExpression that translates a condition automaton $\mathcal{A}$ to a path-equivalent relation algebra expression.

(a) No path from q_1 to q exists and there exists a path from q to q_2. In this case, we have $e_{q_1,q_2} = \emptyset \cup \emptyset \circ [e_{q,q}]^* \circ e_{q,q_2}$ after Line 7.

(b) There exists a path from q_1 to q and no path from q to q_2 exists. In this case, we have $e_{q_1,q_2} = \emptyset \cup e_{q_1,q} \circ [e_{q,q}]^* \circ \emptyset$ after Line 7.

(c) No path from q_1 to q exists and no path from q to q_2 exists. In this case, we have $e_{q_1,q_2} = \emptyset \cup \emptyset \circ [e_{q,q}]^* \circ \emptyset$ after Line 7.

In all three cases, e_{q_1,q_2} can be simplified to $\emptyset$ using Lemma 3.2 (ii).

2. *Let $q \in S \cup \{v, w\}$. If $\mathcal{A}'$ is acyclic, then $e_{q,q} \equiv_{\text{path}} \emptyset$.*

Observe that $\mathcal{A}'$ is acyclic if there exists no path from a state to itself with at least a single transition. Hence, by Invariant 1, we have $e_{q,q} = \emptyset$.

3. *Let $q_1, q_2 \in S \cup \{v, w\}$. The expression e_{q_1,q_2} is in $\mathcal{N}(\mathcal{F})$.*

We initialize e_{q_1,q_2} as either $\emptyset$ or a union of expressions of the form conditions$(q_1) \circ \ell \circ$ conditions(q_2), with ℓ an edge label. Clearly, these expressions are in $\mathcal{N}(\mathcal{F})$ if all condition expressions in C are in $\mathcal{N}(\mathcal{F})$. After initialization, the value of e_{q_1,q_2} only changes at Line 7. Line 7 does not introduce the operators π and $\overline{\pi}$.

Line 7 introduces the operator *. This is only the case for subexpressions of the form $[e_{q,q}]^*$. If $\mathcal{A}'$ is acyclic, which must be the case when $^* \notin \mathcal{F}$, then, by Invariant 2, we have $e_{q,q} \equiv_{\text{path}} \emptyset$ and, using Lemma 3.2 (i), we have $[e_{q,q}]^* \equiv_{\text{path}} [\emptyset]^*$, which is path-equivalent to id.

4. *Let $q_1, q_2 \in S \cup \{v, w\}$. If $(m, n) \in [\![e_{q_1,q_2}]\!]_{\mathcal{G}}$, then there exists a run $(t_1, m) \ldots (t_i, n)$ of $\mathcal{A}'$ on $\mathcal{G}$ with $t_1 = q_1$ and $t_i = q_2$ that performs at least one transition.*

If at Line 3 we have $(m, n) \in [\![e_{q_1,q_2}]\!]_{\mathcal{G}}$, then, by the initial construction of e_{q_1,q_2} at Line 3, and, by the semantics of $\cup$, there exists a subexpression conditions$(q_1) \circ \ell \circ$ conditions(q_2)

in e_{q_1,q_2} such that $(m,n) \in [\![\text{conditions}(q_1) \circ \ell \circ \text{conditions}(q_2)]\!]_{\mathcal{G}}$ and $(q_1,\ell,q_2) \in \delta \cup \delta_{v,w}$. By the semantics of $\circ$, this implies $(m,m) \in [\![\text{conditions}(q_1)]\!]_{\mathcal{G}}$, $(n,n) \in [\![\text{conditions}(q_2)]\!]_{\mathcal{G}}$, and $(m,n) \in [\![\ell]\!]_{\mathcal{G}}$. Hence, we construct the run $(q_1,m)\ \ell\ (q_2,n)$ of $\mathcal{A}'$ on $\mathcal{G}$.

Assume the Invariant holds before execution of Line 7. Now consider the change made to e_{q_1,q_2} when executing Line 7. For distinction, we denote the new value of e_{q_1,q_2} by e'_{q_1,q_2}. If $(m,n) \in [\![e'_{q_1,q_2}]\!]_{\mathcal{G}}$, then, by the construction of e'_{q_1,q_2}, there are two possible cases:

(a) $(m,n) \in [\![e_{q_1,q_2}]\!]_{\mathcal{G}}$, in which case the Invariant can be applied to e_{q_1,q_2} to provide the required run.

(b) $(m,n) \notin [\![e_{q_1,q_2}]\!]_{\mathcal{G}}$ and $(m,n) \in [\![e_{q_1,q} \circ [e_{q,q}]^* \circ e_{q,q_2}]\!]_{\mathcal{G}}$. By the semantics of $\circ$, there exists nodes $m', n' \in \mathcal{V}$ such that $(m,m') \in [\![e_{q_1,q}]\!]_{\mathcal{G}}$, $(m',n') \in [\![[e_{q,q}]^*]\!]_{\mathcal{G}}$, and $(n',n) \in [\![e_{q,q_2}]\!]_{\mathcal{G}}$. By applying the Invariant on $(m,m') \in [\![e_{q_1,q}]\!]_{\mathcal{G}}$ and $(n',n) \in [\![e_{q,q_2}]\!]_{\mathcal{G}}$, we conclude that there exists runs $(q_1,m)\ldots(q,m')$ and $(q,n')\ldots(q_2,n)$ of $\mathcal{A}$ on $\mathcal{G}$. Since $[e_{q,q}]^* = [e_{q,q}]^+ \cup \text{id}$, there are two possible cases:

   i. $(m',n') \in [\![[e_{q,q}]^+]\!]_{\mathcal{G}}$. By the semantics of *, there exists $k \geq 1$ such that $(m',n') \in [\![e^k_{q,q}]\!]_{\mathcal{G}}$. By the semantics of $\circ$, there exists nodes $n_1,\ldots,n_{k+1}$ with $m' = n_1$ and $n' = n_{k+1}$ such that, for $1 \leq i \leq k$, $(n_i,n_{i+1}) \in [\![e_{q,q}]\!]_{\mathcal{G}}$. By applying the Invariant on every $(n_i,n_{i+1}) \in [\![e_{q,q}]\!]_{\mathcal{G}}$, we conclude that there exists runs $(q,n_i)\ldots(q,n_{i+1})$ of $\mathcal{A}$ on $\mathcal{G}$. To construct the required run, we concatenate the runs $(q_1,m)\ldots(q,m')$, $(q,n_1)\ldots(q,n_2)$, $\ldots$, $(q,n_k)\ldots(q,n_{k+1})$, $(q,n')\ldots(q_2,n)$.

   ii. $(m',n') \in [\![\text{id}]\!]_{\mathcal{G}}$. Hence, $m' = n'$. To construct the required run, we concatenate the runs $(q_1,m)\ldots(q,m')$ and $(q,n')\ldots(q_2,n)$.

5. *Let $q_1,q_2 \in S \cup \{v,w\}$ If, at some point during the execution of the algorithm, $(m,n) \in [\![e_{q_1,q_2}]\!]_{\mathcal{G}}$, then $(m,n) \in [\![e_{q_1,q_2}]\!]_{\mathcal{G}}$ at all later steps.*

Follows immediately from inspecting Line 7 of the algorithm.

6. *Let $(q_0,n_0)\ \ell_0\ (q_1,n_1)\ell_1\ldots(q_{i-1},n_{i-1})\ \ell_{i-1}\ (q_i,n_i)$ be a run of $\mathcal{A}'$ on $\mathcal{G}$ that performs at least one transition. If $M[q_j] = \texttt{False}$, $1 \leq j \leq i-1$, then $(n_0,n_i) \in [\![e_{q_0,q_i}]\!]_{\mathcal{G}}$.*

First, we consider the runs that meet the conditions of the Invariant before the execution of the while-loop starting at Line 4. Initially, at Line 3, all states are marked, and, hence, only runs of the form $(q_0,n_0)\ell(q_1,n_1)$ of $\mathcal{A}'$ on $\mathcal{G}$ satisfy the restrictions. By the definition of a run, we have $(q_0,\ell,q_1) \in \delta \cup \delta_{v,w}$. By the initial construction of e_{q_0,q_1} at Line 3, e_{q_0,q_1} is a union that includes the subexpression $\text{conditions}(q_0) \circ \ell \circ \text{conditions}(q_1)$. By the definition of a run, we also have $(n_0,n_0) \in [\![\text{conditions}(q_0)]\!]_{\mathcal{G}}$, $(n_0,n_1) \in [\![\ell]\!]_{\mathcal{G}}$, and $(n_1,n_1) \in [\![\text{conditions}(q_1)]\!]_{\mathcal{G}}$. Hence, $(n_0,n_1) \in [\![\text{conditions}(q_0) \circ \ell \circ \text{conditions}(q_1)]\!]_{\mathcal{G}}$, yielding $(n_0,n_1) \in [\![e_{q_0,q_1}]\!]_{\mathcal{G}}$.

Next we consider the runs that meet the conditions of the Invariant due to changes made during the execution of the while-loop starting at Line 4. Assume the Invariant holds before execution of Line 5. Now consider that we unmark state q by executing Line 9. Let

$$(q_0,n_0)\ \ell_0\ (q_1,n_1)\ \ell_1\ldots(q_{i-1},n_{i-1})\ \ell_{i-1}\ (q_i,n_i)$$

be a run of $\mathcal{A}'$ on $\mathcal{G}$ that performs at least one transition with $M[q_1] = M[q_2] = \cdots = M[q_{i-1}] = \texttt{False}$. We have two cases.

(a) $q \notin \{q_1,\ldots,q_{i-1}\}$. Hence, by the Invariant, we had $(n_0,n_i) \in [\![e_{q_0,q_i}]\!]_{\mathcal{G}}$ at Line 5. By Invariant 5, we conclude that, when executing Line 9, we have $(n_0,n_i) \in [\![e_{q_0,q_i}]\!]_{\mathcal{G}}$.

(b) $q \in \{q_1, \ldots, q_{i-1}\}$. During an iteration of the while loop at Line 4, the value of e_{q_0,q_i} is changed by the execution of Line 7. We denote the new value of e_{q_0,q_i} by e'_{q_0,q_i} for distinction. We have

$$\begin{aligned} e'_{q_0,q_i} &= e_{q_0,q_i} \cup e_{q_0,q} \circ [e_{q,q}]^* \circ e_{q,q_i} \\ &= e_{q_0,q_i} \cup e_{q_0,q} \circ \left([e_{q,q}]^+ \cup \mathrm{id}\right) \circ e_{q,q_i} \\ &= e_{q_0,q_i} \cup e_{q_0,q} \circ [e_{q,q}]^+ \circ e_{q,q_i} \cup e_{q_0,q} \circ \mathrm{id} \circ e_{q,q_i} \end{aligned}$$

Based on the number of occurrences of q in $\{q_1, \ldots, q_{i-1}\}$, we distinguish two cases:

i. There exists exactly one j, $1 \leq j \leq i-1$, with $q_j = q$. We split the run into $(q_0, n_0) \ldots (q_j, n_j)$ and $(q_j, n_j) \ldots (q_i, n_i)$. We had $M[q_1] = \cdots = M[q_{j-1}] = \texttt{False}$ and $M[q_{j+1}] = \cdots = M[q_{i-1}] = \texttt{False}$ at Line 5. Hence, by the Invariant, we had $(n_0, n_j) \in [\![e_{q_0,q}]\!]_{\mathcal{G}}$ and $(n_j, n_i) \in [\![e_{q,q_i}]\!]_{\mathcal{G}}$ at Line 5. It follows that $(n_0, n_i) \in [\![e_{q_0,q} \circ \mathrm{id} \circ e_{q,q_i}]\!]_{\mathcal{G}}$. Hence, we have $(n_0, n_i) \in [\![e'_{q_0,q_i}]\!]_{\mathcal{G}}$.

ii. There exists several j, $1 \leq j \leq i-1$ with $q_j = q$. We split the run

$$(q_0, n_0)\ \ell_0\ (q_1, n_1)\ \ell_1 \ldots (q_{i-1}, n_{i-1})\ \ell_{i-1}\ (q_i, n_i)$$

at every (q_j, n_j), $1 \leq j \leq i$, with $q_j = q$ resulting in runs $(q_1, n_1) \ldots (q, n'_1)$, $(q, n'_1) \ldots (q, n'_2)$, $\ldots$, $(q, n'_{k-1}) \ldots (q, n'_k)$, $(q, n'_k) \ldots (q_i, n_i)$, with $1 \leq k$. At Line 5 we had $M[q'] = \texttt{False}$ for all $q' \in \{q_1, \ldots, q_{i-1}\} - \{q\}$. Hence, by the Invariant, we had $(n_1, n'_1) \in [\![e_{q_1,q}]\!]_{\mathcal{G}}$, $(n'_l, n'_{l+1}) \in [\![e_{q,q}]\!]_{\mathcal{G}}$, for $1 \leq l < k$, and $(n'_k, n_i) \in [\![e_{q,q_i}]\!]_{\mathcal{G}}$. It follows that $(n'_1, n'_k) \in [\![[e_{q,q}]^*]\!]_{\mathcal{G}}$ and $(n_0, n_i) \in [\![e_{q_0,q} \circ [e_{q,q}]^* \circ e_{q,q_i}]\!]_{\mathcal{G}}$. Hence, we have $(n_0, n_i) \in [\![e'_{q_0,q_i}]\!]_{\mathcal{G}}$.

In both cases we use Invariant 5 to conclude that, after the execution of Line 9, we have $(n_0, n_i) \in [\![e_{q_0,q_i}]\!]_{\mathcal{G}}$.

As $v \neq w$, v is the only initial state, and w is the only final state, every accepting run of $\mathcal{A}'$ performs at least one transition. Hence Invariants 4 and 6 imply that $\mathcal{A}'$ and the resulting expression $e_{v,w}$ are path-equivalent. Invariant 3 implies that the resulting expression is, as required, in $\mathcal{N}(\mathcal{F})$. □

Notice that we did not only prove path equivalence between classes of condition automata and classes of the expressions on general labeled graphs, but also provided constructive algorithms to translate between these classes.

4.4 Closure under intersection

In the following, we work towards showing that condition automata, when used as queries on *trees*, are not only closed under $\circ$, $\cup$, and *, as Proposition 4.1 shows, but also under $\cap$ and $-$. We then use this closure result to remove $\cap$ and $-$ from expressions that are used to query trees. The standard approach to constructing the intersection of two finite automata is by making their cross-product. In a fairly straightforward manner, we can apply a similar cross-product construction to condition automata, given that they are id-transition-free. Observe that the id-labeled transitions fulfill a similar role as empty-string-transitions in finite automata and, as such, can be removed, which we show next.

Definition 4.3. Let $\mathcal{A} = (S, \Sigma, C, I, F, \delta, \gamma)$ be a condition automaton and $q \text{ id } q_1 \cdots \text{id } q_j$ be a path in $\mathcal{A}$. We say that the pair $(q, \{q, q_1, \ldots, q_j\})$ is an *identity pair* of $\mathcal{A}$.

Lemma 4.4. *Let $\mathcal{F} \subseteq \{\pi, \overline{\pi}, {}^*\}$ and let $\mathcal{A}$ be an $\mathcal{F}$-free condition automaton. On labeled graphs, there exists an* id*-transition-free and $\mathcal{F}$-free condition automaton $\mathcal{A}_{\not\text{id}}$ that is path-equivalent to $\mathcal{A}$. The condition automaton $\mathcal{A}_{\not\text{id}}$ is acyclic whenever $\mathcal{A}$ is acyclic.*

Proof. Let $\mathcal{A} = (S, \Sigma, C, I, F, \delta, \gamma)$ be an $\mathcal{F}$-free condition automaton. We construct $\mathcal{A}_{\not\text{id}} = (S_{\not\text{id}}, \Sigma, C, I_{\not\text{id}}, F_{\not\text{id}}, \delta_{\not\text{id}}, \gamma_{\not\text{id}})$ with

$$
\begin{aligned}
S_{\not\text{id}} &= \{(q, Q) \mid (q, Q) \text{ is an identity pair of } \mathcal{A}\};\\
I_{\not\text{id}} &= \{(q, Q) \mid ((q, Q) \in S_{\not\text{id}}) \wedge (q \in I)\};\\
F_{\not\text{id}} &= \{(q, Q) \mid ((q, Q) \in S_{\not\text{id}}) \wedge (Q \cap F \neq \emptyset)\};\\
\delta_{\not\text{id}} &= \{((p, P), \ell, (q, Q)) \mid ((p, P) \in S_{\not\text{id}}) \wedge (\ell \in \Sigma) \wedge\\
&\qquad ((q, Q) \in S_{\not\text{id}}) \wedge (\exists p' \, (p' \in P) \wedge (p', \ell, q) \in \delta)\};\\
\gamma_{\not\text{id}} &= \{((q, Q), c) \mid ((q, Q) \in S_{\not\text{id}}) \wedge (\exists q' \, (q' \in Q) \wedge (c \in \gamma(q')))\}.
\end{aligned}
$$

Let $\mathcal{G} = (\mathcal{V}, \Sigma, \mathbf{E})$ be an arbitrary graph. To prove that $\mathcal{A}_{\not\text{id}}$ is path-equivalent to $\mathcal{A}$, we prove $[\![\mathcal{A}_{\not\text{id}}]\!]_{\mathcal{G}} = [\![\mathcal{A}]\!]_{\mathcal{G}}$.

1. *First, we prove $[\![\mathcal{A}_{\not\text{id}}]\!]_{\mathcal{G}} \subseteq [\![\mathcal{A}]\!]_{\mathcal{G}}$.* Assume $(n_0, n_i) \in [\![\mathcal{A}_{\not\text{id}}]\!]_{\mathcal{G}}$. Hence, there exists a run $((q_0, Q_0), n_0) \ldots ((q_i, Q_i), n_i)$ of $\mathcal{A}_{\not\text{id}}$ on $\mathcal{G}$ with $(q_0, Q_0) \in I_{\not\text{id}}$ and $(q_i, Q_i) \in F_{\not\text{id}}$. By the definition of $F_{\not\text{id}}$, we have $Q_i \cap F \neq \emptyset$. Let $p \in Q_i \cap F$. For every j, $0 \leq j \leq i$, we prove that there exists a run $(q_j, n_j) \ldots (p, n_i)$ of $\mathcal{A}$ on $\mathcal{G}$ using induction on $i - j$. The base case is $j = i$. Consider $((q_i, Q_i), n_i)$. For every $q \in Q_i$, n_i satisfies q by the definition of $S_{\not\text{id}}$ and $\gamma_{\not\text{id}}$. Moreover, there exists a path $q_i \text{ id } s_1 \ldots s_k \text{ id } p$ in $\mathcal{A}$ from q_i to p with $q_i, s_1, \ldots, s_k \in Q_i$. Hence, we conclude that $(q_i, n_i) \text{ id } (s_1, n_i) \ldots (s_k, n_i) \text{ id } (p, n_i)$ is a run of $\mathcal{A}$ on $\mathcal{G}$.

Now assume as induction hypothesis that, for all j, $0 < k \leq j \leq i$, there exists a run $(q_j, n_j) \ldots (p, n_i)$ of $\mathcal{A}$ on $\mathcal{G}$. Consider $((q_{k-1}, Q_{k-1}), n_{k-1}) \, \ell_{k-1} \, ((q_k, Q_k), n_k)$, the k-th step in the run. By the definition of a run, we have $((q_{k-1}, Q_{k-1}), \ell_{k-1}, (q_k, Q_k)) \in \delta_{\not\text{id}}$, n_{k-1} satisfies (q_{k-1}, Q_{k-1}), and n_k satisfies (q_k, Q_k). By the construction of $S_{\not\text{id}}$ and $\gamma_{\not\text{id}}$, we have, for every $q \in Q_{k-1}$, n_{k-1} satisfies q, and, for every $q \in Q_k$, n_k satisfies q. By the construction of $\delta_{\not\text{id}}$, there exists a state $p_{k-1} \in Q_{k-1}$ such that $(p_{k-1}, \ell, q_k) \in \delta$ and, by the construction of $S_{\not\text{id}}$, there exists a path $q_{k-1} \text{ id } s_1 \ldots s_{i'} \text{ id } p_{k-1}$ in $\mathcal{A}$ from q_{k-1} to p_{k-1} with $s_1, \ldots, s_{i'} \in Q_{k-1}$. Hence, we conclude that $(q_{k-1}, n_{k-1}) \text{ id } (s_1, n_j) \ldots (s_{i'}, n_j) \text{ id } (p_{k-1}, n_j) \, \ell_j \, (q_k, n_k)$ is a run of $\mathcal{A}$ on $\mathcal{G}$. Using the induction hypothesis on k, we also conclude that there exists a run $(q_k, n_k) \ldots (p, n_i)$ of $\mathcal{A}$ on $\mathcal{G}$. We concatenate these runs to conclude that there exists a run $(q_{k-1}, n_{k-1}) \ldots (p, n_i)$ of $\mathcal{A}$ on $\mathcal{G}$.

By the definition of $I_{\not\text{id}}$, we have $q_0 \in I$. Hence, we conclude that a run $(q_0, n_0) \ldots (p, n_i)$ of $\mathcal{A}$ on $\mathcal{G}$ with $q_0 \in I$ and $p \in F$ exists, and, as a consequence, $(n_0, n_i) \in [\![\mathcal{A}]\!]_{\mathcal{G}}$.

2. *Next, we prove $[\![\mathcal{A}_{\not\text{id}}]\!]_{\mathcal{G}} \supseteq [\![\mathcal{A}]\!]_{\mathcal{G}}$.* Assume $(n_0, n_i) \in [\![\mathcal{A}]\!]_{\mathcal{G}}$. Hence, there exists a run $(q_0, n_0) \ldots (q_i, n_i)$ of $\mathcal{A}$ on $\mathcal{G}$ with $q_0 \in I$ and $q_i \in F$. For every j, $0 \leq j \leq i$, we prove that there exists a run $((q_j, Q), n_j) \ldots ((p, P), n_i)$ of $\mathcal{A}_{\not\text{id}}$ on $\mathcal{G}$ with $q_i \in P$ using induction on $i - j$. The base case is $j = i$. Observe that $(q_i, \{q_i\})$ is an identity pair. Hence, $(q_i, \{q_i\}) \in S_{\not\text{id}}$. By the construction of $\gamma_{\not\text{id}}$, n_i satisfies $(q_i, \{q_i\})$. We conclude that $(q_i, \{q_i\})$ is a run of $\mathcal{A}_{\not\text{id}}$ on $\mathcal{G}$. Now assume as induction hypothesis that, for all k, $0 < k \leq j \leq i$,

there exists a run $((q_j, Q), n_j) \dots ((p, P), n_i)$ of $\mathcal{A}_{\not\text{id}}$ on $\mathcal{G}$ with $q_i \in P$. Consider the k-th step in the run, $(q_{k-1}, n_{k-1})\ \ell_{k-1}\ (q_k, n_k)$. By the induction hypothesis, there exists a run $r = ((q_k, Q), n_k) \dots ((p, P), n_i)$ of $\mathcal{A}_{\not\text{id}}$ on $\mathcal{G}$ with $q_i \in P$. We distinguish two cases:

(a) $\ell_{k-1} \neq \text{id}$. By the definition of a run, we have $(n_{k-1}, n_k) \in \ell_{k-1}$. By the construction of $S_{\not\text{id}}$, we have $(q_{k-1}, \{q_{k-1}\}) \in S_{\not\text{id}}$, by the construction of $\gamma_{\not\text{id}}$, we have n_{k-1} satisfies $(q_{k-1}, \{q_{k-1}\})$, and, by the construction of $\delta_{\not\text{id}}$, we have $((q_{k-1}, \{q_{k-1}\}), \ell_{k-1}, (q_k, Q)) \in \delta_{\not\text{id}}$. Hence, we conclude that $((q_{k-1}, \{q_{k-1}\}), n_{k-1})\ \ell_{k-1}\ ((q_k, Q), n_k) \dots ((p, P), n_i)$ is a run of $\mathcal{A}_{\not\text{id}}$ on $\mathcal{G}$.

(b) $\ell_{k-1} = \text{id}$. By the semantics of id, we have $n_{k-1} = n_k$. Let $Q = \{q_k, p_1, \dots, p_{i'}\}$. By the construction of $S_{\not\text{id}}$, there exists a path $q_k\ \text{id}\ p_1 \dots p_{i'}$ in $\mathcal{A}$ and, by the definition of a run, we have $(q_{k-1}, \text{id}, q_k) \in \delta$. Hence, we conclude that $q_{k-1}\ \text{id}\ q_k\ \text{id}\ p_1 \dots p_{i'}$ is a path in $\mathcal{A}$, $(q_{k-1}, Q \cup \{q_{k-1}\})$ is an identity pair, and, by the construction of $S_{\not\text{id}}$, we have $(q_{k-1}, Q \cup \{q_{k-1}\}) \in S_{\not\text{id}}$. We again distinguish two cases:

i. $r = ((q_k, Q), n_k)$. In this case, we have $q_i \in Q$, and, hence, $q_i \in Q \cup \{q_{k-1}\}$. We conclude that $((q_{k-1}, Q \cup \{q_{k-1}\}), n_k)$ is a run of $\mathcal{A}_{\not\text{id}}$ with $q_i \in Q \cup \{q_{k-1}\}$.
ii. $r = ((q_k, Q), n_k)\, \ell'\, ((q', Q'), n') \dots ((p, P), n_i)$. In this case, we have, by the definition of a run, $((q_k, Q), \ell', (q', Q')) \in \delta_{\not\text{id}}$. By the definition of $\delta_{\not\text{id}}$, we have $((q_k, Q), \ell', (q', Q')) \in \delta_{\not\text{id}}$ if and only if there exists $q'' \in Q$ such that $(q'', \ell', q') \in \delta$. It follows that $q'' \in Q \cup \{q_{k-1}\}$ and $((q_{k-1}, Q \cup \{q_{k-1}\}), \ell', (q', Q')) \in \delta_{\not\text{id}}$. Hence, we conclude that $((q_{k-1}, Q \cup \{q_{k-1}\}), n_k)\ \ell'\ ((q', Q'), n') \dots ((p, P), n_i)$ is a run of $\mathcal{A}_{\not\text{id}}$.

By the definition of $I_{\not\text{id}}$, we have $(q_0, Q) \in I_{\not\text{id}}$ and by the definition of $F_{\not\text{id}}$ and $q_i \in P$, we have $(p, P) \in F_{\not\text{id}}$. Hence, the run $((q_0, Q), n_0) \dots ((p, P), n_i)$ is a run of $\mathcal{A}_{\not\text{id}}$ on $\mathcal{G}$ with $(q_0, Q) \in I_{\not\text{id}}$ and $(p, P) \in F_{\not\text{id}}$, and, as a consequence, $(n_0, n_i) \in [\![\mathcal{A}_{\not\text{id}}]\!]_{\mathcal{G}}$.

We conclude $[\![\mathcal{A}_{\not\text{id}}]\!]_{\mathcal{G}} = [\![\mathcal{A}]\!]_{\mathcal{G}}$. As we did not add new condition expressions to the set of condition expressions in the above constructions, it follows that $\mathcal{A}_{\not\text{id}}$ is $\mathcal{F}$-free whenever $\mathcal{A}$ is $\mathcal{F}$-free. As every path in $\mathcal{A}_{\not\text{id}}$ can be translated to a path of at least equal length in $\mathcal{A}$ using the same reasoning as in the proof of $[\![\mathcal{A}_{\not\text{id}}]\!]_{\mathcal{G}} \subseteq [\![\mathcal{A}]\!]_{\mathcal{G}}$, it finally follows that $\mathcal{A}_{\not\text{id}}$ is acyclic whenever $\mathcal{A}$ is acyclic. □

Hence, we may assume that condition automata are id-transition-free.

Example 4.5. In Figure 4.6 two condition automata are shown. The condition automaton on the *left* is a simple automaton with id-transitions. The id-transition-free condition automaton on the *right* is obtained by applying the construction of Lemma 4.4. The main step in constructing the condition automaton on the *right* is constructing the identity pairs (as these are the states of the constructed condition automaton). Observe that the condition automaton on the *left* has the following paths consisting of identity-transitions only:

$$u, \quad u\ \text{id}\ v, \quad u\ \text{id}\ v\ \text{id}\ w, \quad v, \quad v\ \text{id}\ w, \quad w,$$

resulting in the identity pairs $(u, \{u\})$, $(v, \{v\})$, and $(w, \{w\})$, the identity pairs $(u, \{u, v\})$ and $(v, \{v, w\})$, and the identity pair $(u, \{u, v, w\})$, which are the states in the condition automaton on the *right*.

We now proceed with showing that condition automata, when used to query trees as opposed to general labeled graphs, are closed under intersection. We already know from Example 4.1 that, on general graphs standard, automata-techniques cannot be adapted to

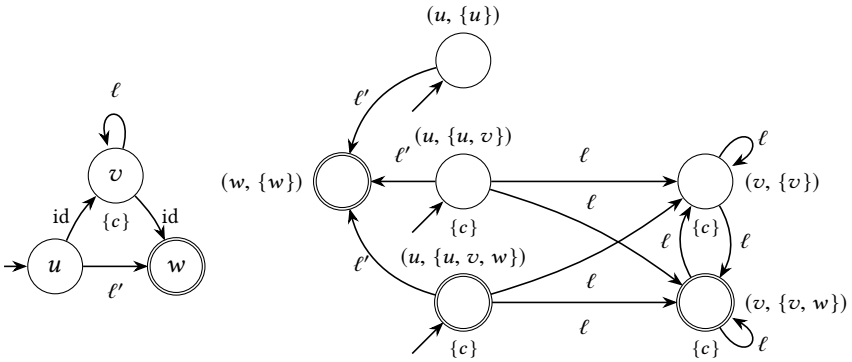


Figure 4.6: Two path-equivalent condition automata. Only the one on the *right* is id-transition-free.

obtain these closure results. On trees, however, the situation of Example 4.1 cannot occur, as a directed path between two nodes in a tree is always unique. This observation is crucial in showing that the cross-product construction on condition automata works for querying trees. The lemma below formalizes this observation:

Lemma 4.5. *Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be* id*-transition-free condition automata and let $\mathcal{T} = (\mathcal{V}, \Sigma, \mathbf{E})$ be a tree. If there exists a run $r_1 = (p_1, n_1)\, \ell_1^1 \cdots \ell_{i_1}^1\, (q_1, n_{i_1+1})$ of $\mathcal{A}_1$ on $\mathcal{T}$ and there exists a run $r_2 = (p_2, m_1)\, \ell_1^2 \cdots \ell_{i_2}^2\, (q_2, m_{i_2+1})$ of $\mathcal{A}_2$ on $\mathcal{T}$ with $n_1 = m_1$ and $n_{i_1+1} = m_{i_2+1}$, then $i_1 = i_2 = i$ and, for all $1 \leq j \leq i$, $\ell_j^1 = \ell_j^2$ and $n_j = m_j$.*

Proof. Let $m = m_1 = m_2$ and $n = n_1 = n_2$. By the semantics of condition automata, the existence of run r_1 implies that $(m, n) \in [\![\ell_1^1 \circ \ldots \circ \ell_{i_1}^1]\!]_{\mathcal{T}}$ and the existence of run r_2 implies that $(m, n) \in [\![\ell_1^2 \circ \ldots \circ \ell_{i_2}^2]\!]_{\mathcal{T}}$. As $\mathcal{A}_1$ and $\mathcal{A}_2$ are id-transition-free, every $\ell_{j_1}^1$ and every $\ell_{j_2}^2$, with $1 \leq j_1 \leq i_1$ and $1 \leq j_2 \leq i_2$, is an edge label. As there is only a single downward path from node m to node n in $\mathcal{T}$, the two runs must traverse the same path, and, hence, follow the same edge labels. We conclude that $i_1 = i_2 = i$ and, for all $1 \leq j \leq i$, $\ell_j^1 = \ell_j^2$. □

This allows us to prove the following:

Proposition 4.6. *Let $\mathcal{F} \subseteq \{\pi, \overline{\pi}, ^*\}$ and let $\mathcal{A}_1$ and $\mathcal{A}_2$ be $\mathcal{F}$-free condition automata. There exists an $\mathcal{F}$-free condition automaton $\mathcal{A}_\cap$ such that, for every tree $\mathcal{T}$, we have $[\![\mathcal{A}_\cap]\!]_{\mathcal{T}} = [\![\mathcal{A}_1]\!]_{\mathcal{T}} \cap [\![\mathcal{A}_2]\!]_{\mathcal{T}}$. The condition automaton $\mathcal{A}_\cap$ is acyclic whenever $\mathcal{A}_1$ or $\mathcal{A}_2$ is acyclic.*

Proof. Let $\mathcal{A}_1 = (S_1, \Sigma_1, C_1, I_1, F_1, \delta_1, \gamma_1)$ and $\mathcal{A}_2 = (S_2, \Sigma_2, C_2, I_2, F_2, \delta_2, \gamma_2)$ be condition automata. By Lemma 4.4, we assume that $\mathcal{A}_1$ and $\mathcal{A}_2$ are id-transition-free. We construct $\mathcal{A}_\cap = (S_1 \times S_2, \Sigma_1 \cup \Sigma_2, C_1 \cup C_2, I_1 \times I_2, F_1 \times F_2, \delta_\cap, \gamma_\cap)$ where

$$\delta_\cap = \{((p_1, q_1), \ell, (p_2, q_2)) \mid (p_1, \ell, p_2) \in \delta_1 \wedge (q_1, \ell, q_2) \in \delta_2\};$$
$$\gamma_\cap = \{((p, q), c) \mid (p, c) \in \gamma_1 \vee (q, c) \in \gamma_2\}.$$

Let $\mathcal{T} = (\mathcal{V}, \Sigma, \mathbf{E})$ be a tree and let $m, n \in \mathcal{V}$ be a pair of nodes. We have $(m, n) \in [\![\mathcal{A}_1]\!]_{\mathcal{T}} \cap [\![\mathcal{A}_2]\!]_{\mathcal{T}}$ if and only if there exists a run $(p_1, m)\, \ell_1^1 \cdots \ell_{i_1}^1\, (q_1, n)$ of $\mathcal{A}_1$ on $\mathcal{T}$ with $p_1 \in I_1$ and $q_1 \in F_1$ and a run $(p_2, m)\, \ell_1^2 \cdots \ell_{i_2}^2\, (q_2, n)$ of $\mathcal{A}_2$ on $\mathcal{T}$ with $p_2 \in I_2$ and $q_2 \in F_2$. Since $\mathcal{A}_1$ and

$\mathcal{A}_2$ are id-transition-free, by Lemma 4.5, and by the construction of $\mathcal{A}_\cap$, these runs exist if and only if there exists a run $((p_1, p_2), m)\,\ell_1 \cdots \ell_i\,((q_1, q_2), n)$ of $\mathcal{A}_\cap$ on $\mathcal{T}$ with $(p_1, p_2) \in I_1 \times I_2$ and $(q_1, q_2) \in F_1 \times F_2$. Hence, we conclude $(m, n) \in [\![\mathcal{A}_\cap]\!]_\mathcal{T}$. Observe that we did not add new condition expressions to the set of condition expressions in the proposed constructions. Hence, we conclude that $\mathcal{A}_\cap$ is $\mathcal{F}$-free whenever $\mathcal{A}_1$ and $\mathcal{A}_2$ are $\mathcal{F}$-free. We also observe that every run r of $\mathcal{A}_\cap$ on $\mathcal{T}$ can be split into runs of $\mathcal{A}_1$ and $\mathcal{A}_2$ on tree $\mathcal{T}$ of the same length as r. Hence, there can only be loops in $\mathcal{A}_\cap$ if both $\mathcal{A}_1$ and $\mathcal{A}_2$ have loops and we conclude that $\mathcal{A}_\cap$ is acyclic whenever $\mathcal{A}_1$ or $\mathcal{A}_2$ is acyclic. □

4.5 Closure under difference

Next, we show that condition automata, when used as tree queries, are also closed under difference. Usually, the difference of two finite automata $\mathcal{A}_1$ and $\mathcal{A}_2$ is constructed by first constructing the complement of $\mathcal{A}_2$, and then constructing the intersection of $\mathcal{A}_1$ with the resulting automaton. We cannot use such a complement construction for condition automata: the complement of a downward binary relation (represented by a condition automaton when evaluated on a tree) is not a downward binary relation. Observe, however, that it is not necessary to consider the full complement for this purpose: as the difference of two downward binary relations is itself a downward relation, we can restrict ourselves to the *downward complement* of a binary relation.

Definition 4.4. Let $\mathcal{T} = (\mathcal{V}, \Sigma, \mathbf{E})$ be a tree. We define the *downward complement* of a binary relation $\mathrm{R} \subseteq \mathcal{V} \times \mathcal{V}$, denoted by $\overline{\mathrm{R}}_\downarrow$, as

$$\overline{\mathrm{R}}_\downarrow = \{(m, n) \mid (m, n) \notin \mathrm{R} \wedge (m, n) \in [\![[\mathcal{E}]^*]\!]_\mathcal{T}\}.$$

If $\mathcal{A}_1$ and $\mathcal{A}_2$ are condition automata and $\mathcal{T}$ is a tree, then we have $[\![\mathcal{A}_1]\!]_\mathcal{T} - [\![\mathcal{A}_2]\!]_\mathcal{T} \equiv [\![\mathcal{A}_1]\!]_\mathcal{T} \cap \overline{[\![\mathcal{A}_2]\!]_\mathcal{T}}_\downarrow$. Hence, we only need to show that condition automata are closed under downward complement. Recall that finite automata are closed under complement and the complement of an automaton can easily be constructed if the automaton is deterministic [72]. For the construction of the downward complement of a condition automaton, we introduce notion similar to deterministic finite automata.

Definition 4.5. The condition automaton $\mathcal{A} = (S, \Sigma, C, I, F, \delta, \gamma)$ is *deterministic* if it is id-transition-free and if it satisfies the following condition: for every tree $\mathcal{T} = (\mathcal{V}, \Sigma, \mathbf{E})$ and for every pair of nodes m, n with m an ancestor of n, there exists exactly one run $(q, m)\,\ell \ldots (p, n)$ of $\mathcal{A}$ on $\mathcal{T}$ with $q \in I$.

We observe that if the condition automaton does not specify any conditions, then Definition 4.5 reduces to the classical definition of a deterministic finite automaton. Moreover, the definition of a deterministic condition automaton relies on the automaton being evaluated on trees, as more general graphs can have several identically-labeled paths between pairs of nodes.

Example 4.6. The condition automaton in Figure 4.3 is clearly not deterministic: there are already two different possible runs of length one starting at an initial state. In Figure 4.7 we exhibit a conditional automaton over $\Sigma = \{\ell_1, \ell_2\}$ that is deterministic. This deterministic condition automaton accepts node pairs (m, n), $m \neq n$, if m satisfies $\pi_2[{\ell_1}^3]$ and if there exists a path from m to n whose labeling matches the regular expression $\ell_1[\ell_2]^*\ell_1$. It also accepts node pairs (n, n) if n does not satisfy $\pi_2[{\ell_1}^3]$.

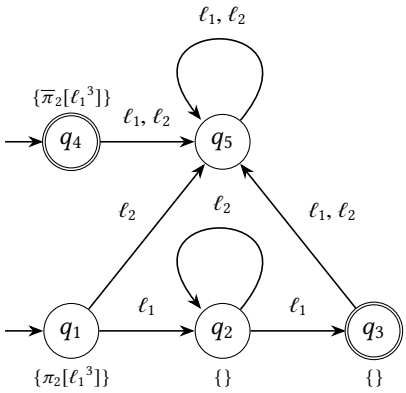


Figure 4.7: An example of a deterministic condition automaton.

In the construction of deterministic condition automata we use the *condition complement* of a condition e, denoted by $\mathrm{ccompl}(e)$, and defined as follows:

$$\mathrm{ccompl}(e) = \begin{cases} \emptyset & \text{if } e = \mathrm{id}; \\ \mathrm{id} & \text{if } e = \emptyset; \\ \overline{\pi}_j[e'] & \text{if } e = \pi_j[e'], j \in \{1,2\}; \\ \pi_j[e'] & \text{if } e = \overline{\pi}_j[e'], j \in \{1,2\}. \end{cases}$$

Observe that the condition complement of a projection expression is a coprojection expression, and vice-versa. If S is a set of conditions, then we use the notation $\mathrm{ccompl}(S)$ to denote the set $\{\mathrm{ccompl}(c) \mid c \in S\}$.

Lemma 4.7. *Let $\mathcal{F} \subseteq \{\pi, \overline{\pi}, {}^*\}$ and let $\mathcal{A}$ be an $\mathcal{F}$-free condition automaton. There exists a deterministic condition automaton $\mathcal{A}_D$ that is path-equivalent to $\mathcal{A}$ with respect to labeled trees. The condition automaton $\mathcal{A}_D$ is $\{{}^*\}$-free if ${}^* \notin \mathcal{F}$ and $\{\pi, \overline{\pi}\}$-free if $\pi, \overline{\pi} \notin \mathcal{F}$.*

Proof. Let $\mathcal{A} = (S, \Sigma, C, I, F, \delta, \gamma)$ be a condition automaton. By Lemma 4.4, we assume that $\mathcal{A}$ is id-transition-free. We construct $\mathcal{A}_D = (S_D, \Sigma, C_D, I_D, F_D, \delta_D, \gamma_D)$, where S_D, I_D, and δ_D are constructed by Algorithm MakeDeterministic, presented in Figure 4.8, and

$$\begin{aligned} C_D &= C \cup \mathrm{ccompl}(C); \\ F_D &= \{(Q,V) \mid (Q,V) \in S_D \wedge Q \cap F \neq \emptyset\}; \\ \gamma_D &= \{((Q,V), c) \mid (Q,V) \in S_D \wedge (c \in V \vee c \in \mathrm{ccompl}(C - V))\}. \end{aligned}$$

Let $\mathcal{T} = (\mathcal{V}, \Sigma, \mathbf{E})$ be a tree and let $m, n \in \mathcal{V}$ be nodes. If $n' \in \mathcal{V}$, then $\zeta(n')$ denotes the set $\{c \mid c \in C \wedge (n', n') \in [\![c]\!]_{\mathcal{T}}\}$. Both determinism of $\mathcal{A}_D$ and path equivalence of $\mathcal{A}_D$ and $\mathcal{A}$ are guaranteed, as this construction satisfies the following properties:

1. *There exists exactly one $V \subseteq C$ with $(n,n) \in [\![\mathrm{conditions}(V \cup \mathrm{ccompl}(C - V))]\!]_{\mathcal{T}}$. Moreover, $V = \zeta(n)$.*

Algorithm MakeDeterministic($\mathcal{A} = (S, \Sigma, C, I, F, \delta, \gamma)$)**:**

```
1:  S_D, I_D, new, δ_D := ∅, ∅, ∅, ∅
2:  for V ⊆ C do
3:      Q := {q | q ∈ I ∧ γ(q) ⊆ V}
4:      S_D, I_D, new := S_D ∪ {(Q, V)}, I_D ∪ {(Q, V)}, new ∪ {(Q, V)}
5:  while new ≠ ∅ do
6:      Remove state (Q, V) from new
7:      for ℓ ∈ Σ do
8:          P := {p | ∃q q ∈ Q ∧ (q, ℓ, p) ∈ δ}
9:          for W ⊆ C do
10:             P' := {p | p ∈ P ∧ γ(p) ⊆ W}
11:             if (P', W) ∉ S_D then
12:                 S_D, new := S_D ∪ {(P', W)}, new ∪ {(P', W)}
13:             δ_D := δ_D ∪ {((Q, V), ℓ, (P', W))}
```

Figure 4.8: Algorithm MakeDeterministic that translates a condition automaton $\mathcal{A}$ to a path-equivalent deterministic condition automaton.

By definition, $\zeta(n)$ is the set of all conditions satisfied by n, hence, we can choose $V = \zeta(n)$ and we have $(n, n) \in [\![\text{conditions}(V \cup \text{ccompl}(C - V))]\!]_{\mathcal{T}}$. To show that no other choice for V is possible, we consider any $V' \subseteq C$ with $V \neq V'$. We show that $(n, n) \notin [\![\text{conditions}(V' \cup \text{ccompl}(C - V'))]\!]_{\mathcal{T}}$. As $V \neq V'$, there exists $c \in C$ such that $c \in V - V'$ or $c \in V' - V$:

(a) $c \in V - V'$. By $c \in V$ and $(n, n) \in [\![\text{conditions}(V \cup \text{ccompl}(C - V))]\!]_{\mathcal{T}}$, we have $(n, n) \in [\![c]\!]_{\mathcal{T}}$ and $(n, n) \notin [\![\text{ccompl}(c)]\!]_{\mathcal{T}}$. As $c \notin V'$, we have $\text{ccompl}(c) \in V' \cup \text{ccompl}(C - V')$. Hence, $(n, n) \notin [\![\text{conditions}(V' \cup \text{ccompl}(C - V'))]\!]_{\mathcal{T}}$.

(b) $c \in V' - V$. Since $c \notin V$, we have $(n, n) \notin [\![c]\!]_{\mathcal{T}}$. As $c \in V'$, we have $c \in V' \cup \text{ccompl}(C - V')$. Hence, $(n, n) \notin [\![\text{conditions}(V' \cup \text{ccompl}(C - V'))]\!]_{\mathcal{T}}$.

2. *There exists exactly one state $(P, V) \in I_D$ such that m satisfies (P, V).*

By Property 1, we have $V = \zeta(m)$. By the construction of S_D, there exists exactly one set of states $Q \subseteq I$ such that $(Q, V) \in I_D$ and $\gamma((Q, V)) = V \cup \text{ccompl}(C - V)$.

3. *Let $(P, V) \in S_D$ be a state such that m satisfies (P, V). If there exists an edge label $\ell \in \Sigma$ such that $(m, n) \in \ell$, then there exists exactly one transition $((P, V), \ell, (Q, W)) \in \delta$ such that n satisfies (Q, W).*

By Property 1, we have $W = \zeta(n)$. By the construction of S_D and δ_D, there is exactly one set of states $Q \subseteq S$ such that $(Q, W) \in S_D$ and $((P, V), \ell, (Q, W)) \in \delta_D$. By the choice of W, m must satisfy (Q, W).

4. *Let $(P, V) \in S_D$ be a state such that m satisfies (P, V). If there exists a directed path from m to n, then there exists exactly one run $((P, V), m) \ldots ((Q, W), n)$ of $\mathcal{A}_D$ on $\mathcal{T}$.*

Repeated application of Property 3.

5. *If $(p, n)\, \ell\, (p', n')$ is a run of $\mathcal{A}$ on $\mathcal{T}$ then, for every (P, V) with $p \in P$, there exists exactly one transition $((P, V), \ell, (P', V'))$ such that n' satisfies (P', V'). For this transition, we have $p' \in P'$.*

By Property 1, we have $V = \zeta(n)$ and $V' = \zeta(n')$. By the construction of δ_D, there is exactly one set of states $P' \subseteq S$ such that $((P, V), \ell, (P', V')) \in \delta_D$. Observe that we must have $\gamma(p') \subseteq V'$, hence $p' \in P'$.

6. *If there exists a run* $(q_1, m) \dots (q_i, n)$ *of* $\mathcal{A}$ *on* $\mathcal{T}$ *with* $q_1 \in I$*, then there exists a run* $((Q_1, V_1), m) \dots ((Q_i, V_i), n)$ *of* $\mathcal{A}_D$ *on* $\mathcal{T}$ *with* $(Q_1, V_1) \in I_D$*, and, for all* j*,* $1 \leq j \leq i$*,* $q_j \in Q_j$*.*

By induction on the length of the run. The base cases are runs of length 0, which are covered by Property 2. The inductive cases are runs of length $i \geq 1$, for which Property 5 can be used in a straightforward manner to extend runs of length $i - 1$ to length i.

7. *If there exists a run* $((P, V), m)\ \ell\ ((Q, W), n)$ *of* $\mathcal{A}_D$ *on* $\mathcal{T}$ *with* $Q \neq \emptyset$*, then, for all* $q \in Q$*, there exists a run* $(p, m)\ \ell\ (q, n)$*, with* $p \in P$*, of* $\mathcal{A}$ *on* $\mathcal{T}$*.*

By the construction of δ_D, there must be a $p \in P$ such that $(p, \ell, q) \in \delta$. By $p \in P$ and $q \in Q$ and by the definition of S_D and γ_D, we have $\gamma(p) \subseteq V \subseteq \gamma_D(P)$ and $\gamma(q) \subseteq W \subseteq \gamma_D(Q)$. Hence m satisfies p and n satisfies q. Thus $(p, m)\ \ell\ (q, n)$ is a run of $\mathcal{A}$ on $\mathcal{T}$.

8. *If there exists a run* $((Q_1, V_1), m) \dots ((Q_i, V_i), n)$ *of* $\mathcal{A}_D$ *on* $\mathcal{T}$ *with* $Q_i \neq \emptyset$*, then, for all* $q_i \in Q_i$*, there exists a run* $(q_1, m) \dots (q_i, n)$ *of* $\mathcal{A}$ *on* $\mathcal{T}$ *with, for all* j*,* $1 \leq j < i$*,* $q_j \in Q_j$*.*

By induction on the length of the run. The base cases involve runs of the form $((Q_i, V_i), n)$ of $\mathcal{A}_D$ on $\mathcal{T}$ with $Q_i \neq \emptyset$. We can choose any $q \in Q_i$ and, by the definition of S_D and γ_D, we have $\gamma(q) \subseteq V_i \subseteq \gamma_D((Q_i, V_i))$. Hence, n satisfies q_i and we conclude that (q_i, n) is a run of $\mathcal{A}$ on $\mathcal{T}$. The inductive cases are runs of length $i \geq 1$, for which Property 7 can be used in a straightforward manner to extend runs of length $i - 1$ to length i.

By Property 2 and Property 4 we conclude that $\mathcal{A}_D$ is a deterministic condition automaton. By Property 6 and the construction of I_D and F_D, $[\![\mathcal{A}]\!]_{\mathcal{T}} \subseteq [\![\mathcal{A}_D]\!]_{\mathcal{T}}$. By Property 8 and the construction of I_D and F_D, $[\![\mathcal{A}_D]\!]_{\mathcal{T}} \subseteq [\![\mathcal{A}]\!]_{\mathcal{T}}$. Hence, we conclude that $\mathcal{A}$ and $\mathcal{A}_D$ are path-equivalent. The construction of C did not add any usage of *, and introduced $\overline{\pi}$ only when π was present. Hence, the condition automata $\mathcal{A}_D$ is $\{^*\}$-free if $^* \notin \mathcal{F}$ and $\{\pi, \overline{\pi}\}$-free if $\pi, \overline{\pi} \notin \mathcal{F}$. □

Using Lemma 4.7, we can construct the downward complement of a condition automaton.

Proposition 4.8. *Let* $\mathcal{F} \subseteq \{\pi, \overline{\pi}, ^*\}$ *and let* $\mathcal{A}$ *be an* $\mathcal{F}$*-free condition automaton. There exists a condition automaton* $\mathcal{A}'$ *such that, for every tree* $\mathcal{T}$*, we have* $[\![\mathcal{A}']\!]_{\mathcal{T}} = \overline{[\![\mathcal{A}]\!]_{\mathcal{T}}}_{\downarrow}$*. The condition automaton* $\mathcal{A}'$ *is* $\{^*\}$*-free if* $^* \notin \mathcal{F}$ *and* $\{\pi, \overline{\pi}\}$*-free if* $\pi, \overline{\pi} \notin \mathcal{F}$*.*

Proof. Let $\mathcal{A}_D = (S_D, \Sigma_D, C_D, I_D, F_D, \delta_D, \gamma_D)$ be a deterministic condition automaton equivalent to $\mathcal{A}$. We construct $\mathcal{A}' = (S_D, \Sigma_D, C_D, I_D, S_D - F_D, \delta_D, \gamma_D)$. Besides those changes made by constructing a deterministic condition automaton, we did not add new condition expressions to the set of condition expressions in the proposed constructions. Hence, the condition automaton $\mathcal{A}'$ is $\{^*\}$-free if $^* \notin \mathcal{F}$ and $\{\pi, \overline{\pi}\}$-free if $\pi, \overline{\pi} \notin \mathcal{F}$. □

We can now conclude the following:

Corollary 4.9. *Let* $\mathcal{F} \subseteq \{\pi, \overline{\pi}, ^*\}$ *and let* $\mathcal{A}_1$ *and* $\mathcal{A}_2$ *be* $\mathcal{F}$*-free condition automata. There exists a condition automaton* $\mathcal{A}_-$ *such that, for every tree* $\mathcal{T}$*, we have* $[\![\mathcal{A}_-]\!]_{\mathcal{T}} = [\![\mathcal{A}_1]\!]_{\mathcal{T}} - [\![\mathcal{A}_2]\!]_{\mathcal{T}}$*. The condition automaton* $\mathcal{A}_-$ *is* $\{^*\}$*-free if* $^* \notin \mathcal{F}$*,* $\{\pi, \overline{\pi}\}$*-free if* $\pi, \overline{\pi} \notin \mathcal{F}$*, and acyclic whenever* $\mathcal{A}_1$ *is acyclic.*

Proof. Since $[\![\mathcal{A}_1]\!]_{\mathcal{T}} - [\![\mathcal{A}_2]\!]_{\mathcal{T}} = [\![\mathcal{A}_1]\!]_{\mathcal{T}} \cap \overline{[\![\mathcal{A}_2]\!]_{\mathcal{T}}}_{\downarrow}$, we can apply Propositions 4.6 and 4.8 to construct $\mathcal{A}_-$. □

4.6 The collapse of ∩ and − in downward queries

Proposition 4.6 and Corollary 4.9 can be used to remove intersection and difference from an expression at the highest level, but these results ignore expressions inside projections and coprojections. To fully remove intersection and difference, we use a bottom-up construction:

Theorem 4.10. *Let $\mathcal{F} \subseteq \{\pi, \overline{\pi}, \cap, -, ^*\}$. On labeled trees, we have $\mathcal{N}(\mathcal{F}) \preceq_{\text{path}} \mathcal{N}(\underline{\mathcal{F}} - \{\cap, -\})$.*

Proof. Given an expression e in $\mathcal{N}(\mathcal{F})$, we construct the path-equivalent expression in $\mathcal{N}(\underline{\mathcal{F}} - \{\cap, -\})$ in a bottom-up fashion by constructing a condition automaton $\mathcal{A}$ that is path-equivalent to e and is appropriate, according to Table 4.1, for the class $\mathcal{N}(\underline{\mathcal{F}} - \{\cap, -\})$. Using Proposition 4.3, the constructed condition automaton $\mathcal{A}$ can be translated to an expression in $\mathcal{N}(\underline{\mathcal{F}} - \{\cap, -\})$.

The base cases are expressions of the form $\emptyset$, id, and ℓ (for ℓ an edge label), for which we directly construct condition automata using Proposition 4.2. We use Proposition 4.1 to deal with the operators $\circ$, $\cup$, and *. We deal with expressions of the form $f_j[e]$, $f \in \{\pi, \overline{\pi}\}$, $j \in \{1, 2\}$ by translating the condition automaton path-equivalent to e to an expression e', which is in $\mathcal{N}(\underline{\mathcal{F}} - \{\cap, -\})$, and then use Proposition 4.2 to construct the condition automaton path-equivalent to $f_j[e]$. Finally, we use Proposition 4.6 and Corollary 4.9 to deal with the operators $\cap$ and $-$. □

Observe that Theorem 4.10 does not strictly depend on the graph being a tree: indirectly, Theorem 4.10 depends on Lemma 4.5, which holds for all graphs in which every pair of nodes is connected by at most one directed path. Hence, the results of Theorem 4.10 can be generalized to, for example, forests.

4.7 Condition automata on chains

Condition automata as a tool to represent and manipulate expressions can also be used to simplify Boolean queries. As a step in this direction, we use condition automata to simplify expressions in $\mathcal{N}(\mathcal{F})$, $\{\pi\} \subseteq \mathcal{F} \subseteq \{\pi, ^*\}$, that are used to query chains. We do this by providing manipulation steps that reduce the total *weight* of the projections in an expression:

Definition 4.6. Let e be an expression in $\mathcal{N}(\pi, ^*)$. We define the *condition depth* of e, denoted by $\text{cdepth}(e)$, as

$$\text{cdepth}(e) = \begin{cases} 0 & \text{if } e \in \{\emptyset, \text{id}\}; \\ 0 & \text{if } e = \ell, \text{ with } \ell \text{ an edge label}; \\ \text{cdepth}(e') & \text{if } e = [e']^*; \\ \text{cdepth}(e') + 1 & \text{if } e = \pi_j[e'], j \in \{1, 2\}; \\ \max(\text{cdepth}(e_1), \text{cdepth}(e_2)) & \text{if } e = e_1 \oplus e_2, \oplus \in \{\circ, \cup\}. \end{cases}$$

We define the *condition depth* of a $\{\overline{\pi}\}$-free condition automaton $\mathcal{A} = (S, \Sigma, C, I, F, \delta, \gamma)$, denoted by $\text{cdepth}(\mathcal{A})$, as $\text{cdepth}(\mathcal{A}) = \max\{\text{cdepth}(c) \mid c \in C\}$. We define the *condition weight* of $\mathcal{A}$, denoted by $\text{cweight}(\mathcal{A})$, as

$$\text{cweight}(\mathcal{A}) = |\{c \mid c \in C \wedge \text{cdepth}(c) = \text{cdepth}(\mathcal{A})\}|.$$

We now prove the following technical lemma:

Lemma 4.11. *Let $\mathcal{A}$ be a $\{\overline{\pi}\}$-free and* id*-transition-free condition automaton. If* $\mathrm{cdepth}(\mathcal{A}) > 0$*, then there exists a $\{\overline{\pi}\}$-free and* id*-transition-free condition automaton $\mathcal{A}_\pi$ such that*

(i) for every labeled chain C, we have $[\![\mathcal{A}]\!]_C = \emptyset$ if and only if $[\![\mathcal{A}_\pi]\!]_C = \emptyset$;

(ii) either $\mathrm{cdepth}(\mathcal{A}) > \mathrm{cdepth}(\mathcal{A}_\pi)$ *or both* $\mathrm{cdepth}(\mathcal{A}) = \mathrm{cdepth}(\mathcal{A}_\pi)$ *and* $\mathrm{cweight}(\mathcal{A}) > \mathrm{cweight}(\mathcal{A}_\pi)$.

The condition automaton $\mathcal{A}_\pi$ is acyclic and $\{^*\}$-free whenever $\mathcal{A}$ is acyclic and $\{^*\}$-free.

Proof. Let $\mathcal{A} = (S, \Sigma, C, I, F, \delta, \gamma)$ be a $\{\overline{\pi}\}$-free and id-transition-free condition automaton. Choose a condition $c \in C$ with $\mathrm{cdepth}(\mathcal{A}) = \mathrm{cdepth}(c)$. If c is either $\emptyset$ or id, then we can eliminate it in a straightforward manner. Hence, without loss of generality, we assume that c is of the form $\pi_j[e']$, $j \in \{1, 2\}$. Let $\mathcal{A}' = (S', \Sigma', C', I', F', \delta', \gamma')$ be a $\{\overline{\pi}\}$-free and id-transition-free condition automaton equivalent to e' constructed using Proposition 4.2 and Lemma 4.4. It is straightforward to verify that $\mathrm{cdepth}(\mathcal{A}') = \mathrm{cdepth}(e') = \mathrm{cdepth}(e) - 1$. We define $\mathcal{A}_\pi = (S_\pi, \Sigma_\pi, C_\pi, I_\pi, F_\pi, \delta_\pi, \gamma_\pi)$ as follows:

$$S_c = \{q \mid c \in \gamma(q)\}; \tag{4.1}$$

$$S_{\neg c} = S - S_c; \tag{4.2}$$

$$S_{\neg 1} = \{(q, Q) \mid q \in S_c \wedge Q \subseteq S' \wedge Q \cap I' = \emptyset\}; \tag{4.3}$$

$$S_{\neg 2} = \{(q, Q) \mid q \in S_c \wedge Q \subseteq S' \wedge Q \cap F' = \emptyset\}; \tag{4.4}$$

$$S_\pi = (S \times \mathcal{P}(S')) - S_{\neg j} \cup \{\rho\} \times (\mathcal{P}(S') - \emptyset)\,; \tag{4.5}$$

$$\Sigma_\pi = \Sigma; \tag{4.6}$$

$$C_\pi = (C - \{c\}) \cup C'; \tag{4.7}$$

$$I_1 = \{(q, \{q'\}) \mid q \in S_c \cap I \wedge q' \in I'\}; \tag{4.8}$$

$$\begin{aligned} I_2 = {} & \{(q, Q) \mid q \in S_{\neg c} \cap I \wedge \emptyset \subset Q \subseteq I'\} \\ & \cup \{(q, Q) \mid q \in S_c \cap I \wedge \emptyset \subset Q \subseteq I' \cap F'\} \\ & \cup \{(\rho, Q) \mid \emptyset \subset Q \subseteq I'\}; \end{aligned} \tag{4.9}$$

$$I_\pi = \{(q, \emptyset) \mid q \in S_{\neg c} \cap I\} \cup I_j; \tag{4.10}$$

$$\begin{aligned} F_1 = {} & \{(q, Q) \mid q \in S_{\neg c} \cap F \wedge \emptyset \subset Q \subseteq F'\} \\ & \cup \{(q, Q) \mid q \in S_c \cap F \wedge \emptyset \subset Q \subseteq F' \cap I'\} \\ & \cup \{(\rho, Q) \mid \emptyset \subset Q \subseteq F'\}; \end{aligned} \tag{4.11}$$

$$F_2 = \{(q, \{q'\}) \mid q \in S_c \cap F \wedge q' \in F'\}; \tag{4.12}$$

$$F_\pi = \{(q, \emptyset) \mid q \in S_{\neg c} \cap F\} \cup F_j; \tag{4.13}$$

$$\begin{aligned} \delta_{\mathcal{P}(S')} = \{(P, \ell, Q) \mid {} & P \subseteq S' \wedge \ell \in \Sigma_\pi \wedge Q \subseteq S' \wedge \\ & (\forall p\ p \notin P \vee (\exists q\ q \in Q \wedge (p, \ell, q) \in \delta')) \wedge \\ & (\forall q\ q \notin Q \vee (\exists p\ p \in P \wedge (p, \ell, q) \in \delta'))\}; \end{aligned} \tag{4.14}$$

$$\begin{aligned} \delta_{1,\mathrm{b}} = \{((p, P \cup P'), \ell, (q, Q)) \mid {} & \\ & (p, P \cup P') \in S_\pi \wedge P' \subseteq F' \wedge (q, Q) \in S_\pi \wedge \\ & ((p, \ell, q) \in \delta \vee ((p = \rho \vee p \in F) \wedge q = \rho)) \wedge \\ & (P, \ell, Q) \in \delta_{\mathcal{P}(S')}\}; \end{aligned} \tag{4.15}$$

$$\begin{aligned}
\delta_{2,\mathrm{b}} = \{((p,P),\ell,(q,Q\cup Q')) \mid\ & \\
& (p,P)\in S_\pi \wedge Q'\subseteq I' \wedge (q,Q\cup Q')\in S_\pi \wedge \\
& ((p,\ell,q)\in\delta \vee (p=\rho\wedge(q=\rho\vee q\in I)))\wedge \\
& (P,\ell,Q)\in\delta_{\mathcal{P}(S')}\}; && (4.16)\\
\delta_{1,c} = \{((p,P\cup P'),\ell,(q,Q\cup\{q'\})) \mid\ & \\
& (p,P\cup P')\in S_\pi \wedge (q,Q\cup\{q'\})\in S_\pi \wedge \\
& (p,\ell,q)\in\delta \wedge (P,\ell,Q)\in\delta_{\mathcal{P}(S')} \wedge \\
& q\in S_c \wedge q'\in I' \wedge P'\subseteq F'\}; && (4.17)\\
\delta_{2,c} = \{((p,P\cup\{p'\}),\ell,(q,Q\cup Q')) \mid\ & \\
& (p,P\cup\{p'\})\in S_\pi \wedge (q,Q\cup Q')\in S_\pi \wedge \\
& (p,\ell,q)\in\delta \wedge (P,\ell,Q)\in\delta_{\mathcal{P}(S')} \wedge \\
& p\in S_c \wedge p'\in F' \wedge Q'\subseteq I'\}; && (4.18)\\
\delta_\pi = \delta_{j,\mathrm{b}}\cup\delta_{j,c}; && (4.19)\\
\gamma_\pi = \{((q,Q),c') \mid (q,Q)\in S_\pi \wedge\ & \\
& (c'\in\gamma(q)\vee(\exists q'\ q'\in Q\wedge c'\in\gamma'(q')))\}, && (4.20)
\end{aligned}$$

in which $\rho\notin S\cup S'$ is a fresh state, $\mathcal{P}(S)=\{S' \mid S'\subseteq S\}$ is the power set of set S, and we use the values 1 and 2 and the variable $j, j\in\{1,2\}$, to indicate that definitions depend on the type j of the condition $c=\pi_j[e']$.

We shall prove that $\mathcal{A}_\pi$ satisfies the necessary properties.

1. *Either* $d>\mathrm{cdepth}(\mathcal{A}_\pi)$ *or both* $d=\mathrm{cdepth}(\mathcal{A}_\pi)$ *and* $\mathrm{cweight}(\mathcal{A})>\mathrm{cweight}(\mathcal{A}_\pi)$.

Observe that $\mathrm{cdepth}(\mathcal{A}')<\mathrm{cdepth}(\mathcal{A})$. Hence, the property follows directly from (4.7), the definition of $\mathrm{cdepth}(\cdot)$, and the definition of $\mathrm{cweight}(\cdot)$.

2. *Let* $C=(\mathcal{V},\Sigma,\mathbf{E})$ *be a labeled chain and let* $c=\pi_1[e']$. *If* $(m,n)\in[\![\mathcal{A}]\!]_C$, *then there exists* $v\in\mathcal{V}$ *such that* $(m,v)\in[\![\mathcal{A}_\pi]\!]_C$.

3. *Let* $C=(\mathcal{V},\Sigma,\mathbf{E})$ *be a labeled chain and let* $c=\pi_2[e']$. *If* $(m,n)\in[\![\mathcal{A}]\!]_C$, *then there exists* $v\in\mathcal{V}$ *such that* $(v,n)\in[\![\mathcal{A}_\pi]\!]_C$.

We only prove Property 2; Property 3 can be proven in an analogous way. We show that a single run of $\mathcal{A}$ is simulated by a single run of $\mathcal{A}_\pi$ that, at the same time, also simulates the runs of $\mathcal{A}'$ starting at every state $q\in S$ with $c\in\gamma(q)$.

Let $(q_1,n_1)\dots(q_z,n_z)$ be an id-transition-free run with $q_1\in I$ and $q_z\in F$ proving $(n_1,n_z)\in[\![\mathcal{A}]\!]_C$. Now consider a state q_i, $1\le i\le z$, such that $c\in\gamma(q_i)$. Observe that, by (4.1), we have $c\in\gamma(q_i)$ if and only if $q_i\in S_c$. As n_i satisfies q_i, we must have $(n_i,n_i)\in[\![c]\!]_C$. Hence, by the semantics of $\pi_1[\cdot]$, there must exist an id-transition-free run $(p_i,n_i)\dots(p_i',m_i)$ of $\mathcal{A}'$ on C with $p_i\in I'$ and $p_i'\in F'$ proving that a node m_i exists such that $(n_i,m_i)\in[\![\mathcal{A}']\!]_C$.

For every state q_i with $q_i\in S_c$ we choose such a run $(p_i,n_i)\dots(p_i',m_i)$ of $\mathcal{A}'$ on C with $p_i\in I'$ and $p_i'\in F$. Let d be the maximum distance between, on the one hand, node n_1, and, on the other hand, node n_z and the nodes m_i in these runs. For every $1\le k\le d$, we define

$$R_k=\{s \mid (s,n_k) \text{ is part of a run } (p_i,n_i)\dots(p_i',m_i) \text{ with } 1\le i\le z \text{ and } q_i\in S_c\},$$

we define $r_k = q_k$ if $k \leq z$ and $r_k = \rho$ if $k > z$. Let $n_1 \ell_1 \dots n_d$ be the directed path from node n_1 to the node at distance d in chain C. We prove that $((r_1, R_1), n_1)\ \ell_1\ \dots\ ((r_d, R_d), n_d)$ is a run of $\mathcal{A}_\pi$ on C, with $(r_1, R_1) \in I_\pi$ and $(r_d, R_d) \in F_\pi$, in the following steps:

(a) *For every k, $1 \leq k \leq d$, we have* $(r_k, R_k) \in S_\pi$. If $1 \leq k \leq z$, then $(r_k, R_k) \in S \times \mathcal{P}(S')$. If $r_k \in S_c$, we have $(p_k, n_k) \in I'$ and, hence, $p_k \in R_k$. By (4.3), we have $(r_k, R_k) \notin S_{\neg 1}$, and, hence, $(r_k, R_k) \in S_\pi$. For $k > z$, we have $(r_k, R_k) \in \{\rho\} \times \mathcal{P}(S')$. By the definition of R_k, we must have $R_k \neq \emptyset$. Hence, we conclude, $(r_k, R_k) \in S_\pi$.

(b) *We have* $(r_1, R_1) \in I_\pi$. By construction, we have $r_1 = q_1$ and $q_1 \in I$. If $r_1 \notin S_c$, then $R_1 = \emptyset$ and, by (4.10), $(r_1, R_1) \in I_\pi$. If $r_1 \in S_c$, then $R_1 = \{p_1\}$, with $p_1 \in I'$, and, by (4.8), $(r_1, R_1) \in I_\pi$.

(c) *For every k, $1 \leq k \leq d$, n_k satisfies* (r_k, R_k). If $1 \leq k \leq z$, then $r_k = q_k$ and n_k satisfies q_k. Hence, we also have $(n_k, n_k) \in [\![\mathsf{conditions}(\gamma(q_k) \setminus \{c\})]\!]_C$. By construction of R_k, we have, for every $s \in R_k$, that n_k satisfies s. Hence, by (4.20), n_k satisfies (r_k, R_k).

(d) *For every k, $1 \leq k < d$,* $((r_k, R_k), \ell_k, (r_{k+1}, R_{k+1})) \in \delta_\pi$. Construct sets P and Q in the following way:

$$P = \{p \mid p \in R_k \wedge (\exists q\ q \in R_{k+1} \wedge (p, \ell_k, q) \in \delta')\};$$
$$Q = \{q \mid q \in R_{k+1} \wedge (\exists p\ p \in R_k \wedge (p, \ell_k, q) \in \delta')\}.$$

Let $P' = R_k - P$. The set of states P contains those states of R_k with a *successor state* in R_{k+1}: for every $p \in P$, there exists a $q \in R_{k+1}$ such that $(p, \ell_k, q) \in \delta_\pi$. Hence, P' contains all states from R_k for which there is no successor state in R_{k+1}. By (4.15) and (4.17), the states in P' must all be final states. We now prove that this restriction on P' holds. Let $s \in P'$ be a state. By the construction of P, there does not exist a state $s' \in R_{k+1}$ such that $(s, \ell_k, s') \in \delta'$. Hence, s can only be a state used at the end of a run $(p_i, n_i) \dots (p'_i, m_i)$, $1 \leq i \leq k$, with $p'_i \in F'$ and $s = p'_i$. We conclude that $P' \subseteq F'$.

Let $Q' = R_{k+1} - Q$. The set of states Q contains those states of R_{k+1} with a *predecessor state* in R_k: for every $q \in Q$, there exists a $p \in R_k$ such that $(p, \ell_k, q) \in \delta_\pi$. Hence, Q' contains all states from R_{k+1} for which there is no predecessor state in R_k. By (4.15) and (4.17), there can only be at most a single state in Q', which must be an initial state. We now prove that these restrictions on Q' hold. Let $s_1, s_2 \in Q'$ be states. By the construction of Q, there does not exist a state $s' \in R_k$ such that $(s', \ell_k, s_1) \in \delta'$ or $(s', \ell_k, s_2) \in \delta'$. Hence, both s_1 and s_2 can only be states used at the begin of the run $(p_k, n_i) \dots (p'_k, m_i)$ with $p_k \in I'$ and we have $s_1 = p_{k+1} = s_2 = p_{k+1}$. We conclude that Q' contains at most a single state, which must be an initial state.

If $r_{k+1} = \rho$, then, by construction of R_{k+1}, we have, for every $s \in R_{k+1}$ and every $1 \leq i \leq z$ with $q_i \in S_c$, $(s, n_{k+1}) \neq (p_i, n_i)$. Hence, s is not at the begin of any run $(p_i, n_i) \dots (p'_i, m_i)$. It follows that there exists a state $s' \in S'$ and a run $(p_i, n_i) \dots (p'_i, m_i)$ containing $(s', n_k)\ \ell_k\ (s, n_{k+1})$. Hence, $s' \in P$, $s \notin Q'$, and $Q' = \emptyset$. If $r_k \in S_c$, then, by construction, we have $p_k \in R_k$ and $p_k \in I'$. If $1 \leq k < z$, then $r_k = q_k$, $r_{k+1} = q_{k+1}$, and $(q_k, \ell, q_{k+1}) \in \delta$. We use (4.15) if $Q' = \emptyset$ and (4.17) if $Q' \neq \emptyset$ to conclude $((r_k, R_k), \ell_k, (r_{k+1}, R_{k+1})) \in \delta_\pi$. Else, if $k = z$, then $r_k = q_k$, $r_{k+1} = \rho$, and $q_k \in F$. Due to $r_{k+1} = \rho$, we have $Q' = \emptyset$. We use (4.15) to conclude $((r_k, R_k), \ell_k, (r_{k+1}, R_{k+1})) \in \delta_\pi$. Finally, if $k > z$, then $r_k = \rho = r_{k+1}$. Due to $r_{k+1} = \rho$, we again have $Q' = \emptyset$. We use (4.15) to conclude $((r_k, R_k), \ell_k, (r_{k+1}, R_{k+1})) \in \delta_\pi$.

(e) *We have* $(r_d, R_d) \in F_\pi$. If $r_d \neq \rho$ then $d = z$ and $q_z \in F$. If $R_d = \emptyset$ then, by (4.13), we have $(r_d, R_d) \in F_\pi$. If $R_d \neq \emptyset$, then, as $n_z = n_d$ is the node with maximum node distance d to n_1 used in any of the runs $(p_i, n_i) \ldots (p'_i, m_i)$ with $1 \leq i \leq z$ and $q_i \in S_c$, we must also have, for every $s \in R_d$, $s \in F'$. If $q_z \in S_c$, then, by construction, $p_z \in R_d$ with $p_z \in I'$. By (4.11) we conclude that, in all these cases, we have $(r_d, R_d) \in F_\pi$.

We conclude that $((r_1, R_1), n_1)\ \ell_1 \ldots ((r_d, R_d), n_d)$ is a run of $\mathcal{A}_\pi$ on C with $(r_1, R_1) \in I_\pi$ and $(r_d, R_d) \in F_\pi$, and, hence $(n_1, n_d) \in [\![\mathcal{A}_\pi]\!]_C$.

4. *Let* $C = (\mathcal{V}, \Sigma, \mathbf{E})$ *be a labeled chain and let* $c = \pi_1[e']$. *If* $(m, n) \in [\![\mathcal{A}_\pi]\!]_C$, *then there exists a* $v \in \mathcal{V}$ *such that* $(m, v) \in [\![\mathcal{A}]\!]_C$.

5. *Let* $C = (\mathcal{V}, \Sigma, \mathbf{E})$ *be a labeled chain and let* $c = \pi_2[e']$. *If* $(m, n) \in [\![\mathcal{A}_\pi]\!]_C$, *then there exists a* $v \in \mathcal{V}$ *such that* $(v, n) \in [\![\mathcal{A}]\!]_C$.

We only prove Property 4; Property 5 can be proven in an analogous way. We show that a single run of $\mathcal{A}_\pi$ simulates a single run of $\mathcal{A}$ and, at the same time, also simulates the runs of $\mathcal{A}'$ starting at every state $q \in S$ with $c \in \gamma(q)$.

Let $((q_1, Q_1), n_1)\ \ell_1 \ldots ((q_z, Q_z), n_z)$ be an id-transition-free run of $\mathcal{A}_\pi$ on C proving $(n_1, n_z) \in [\![C]\!]_{\mathcal{A}_\pi}$. Hence, we have $(q_1, Q_1) \in I_\pi$ and $(q_z, Q_z) \in F_\pi$. Choose i such that $1 \leq i \leq z$, $q_i \neq \rho$, and $i = z$ or $q_{i+1} = \rho$. We prove that $(q_1, n_1)\ \ell_1 \ldots (q_i, n_i)$ is a run of $\mathcal{A}$ on C with $q_1 \in I$ and $q_i \in F$, in the following steps:

(a) *We have* $q_1 \in I$. By (4.8) and (4.10), we have $(q_1, Q_1) \in I_\pi$ only if $q_1 \in I$.

(b) *For all* k, $1 \leq k \leq i$, n_k *satisfies* q_k. We have n_k satisfies (q_k, Q_k). By (4.20), node n_k satisfies all conditions in $\gamma(q_k) \setminus \{c\}$. Hence, if $q_k \notin S_c$, then n_k satisfies q_k. If $q_k \in S_c$, then, by (4.3), there exists a state $p_1 \in Q_k$ such that $p_1 \in I'$. We prove that there are states $p_1 \in Q_k, \ldots, p_d \in Q_{k+d}$ such that $(p_1, n_k)\ \ell_k \ldots (p_d, n_{k+d})$ is a run of $\mathcal{A}'$ on C with $p_1 \in I'$ and $p_d \in F'$. We do so by induction on the length of the run. The base case is (p_1, n_k) and, as $p_1 \in Q_k$, this case is already proven. Thus assume we have a run $(p_1, n_k)\ \ell_k \ldots (p_e, n_{k+e})$ of $\mathcal{A}'$ on C with $1 \leq e < d$ and $p_e \notin F'$. We show that we can extend this run to a run of length $e + 1$. Observe that (4.19) depends on (4.14), via (4.15) and (4.17). If $q_{e+1} \neq \rho$ and $p_e \notin F$, then (4.15) or (4.17) applies, and, hence, by (4.14), there must be a state $p_{e+1} \in Q_{k+e+1}$ such that $(p_e, \ell_{k+e}, p_{e+1}) \in \delta'$. Else, if $q_{e+1} = \rho$ and $p_e \notin F$, then (4.15) applies, and, hence, by (4.14), there must be a state p_{e+1} such that $(p_e, \ell_{k+e}, p_{e+1}) \in \delta'$. We observe that this construction will terminate, as the original run has a finite length z. Hence, at some point we encounter a state $p_d \in F'$. We conclude $(n_k, n_{k+d}) \in [\![\mathcal{A}']\!]_C$, and, by the semantics of $\pi_1[\cdot]$, we also conclude $(n_k, n_k) \in [\![c]\!]_C$. Hence, n_k satisfies all conditions in $\gamma(q_k)$ and, in particular, n_k satisfies q_k.

(c) *For all* $1 \leq k < i$, $(q_k, \ell_k, q_{k+1}) \in \delta$. We have $q_k \neq \rho$ and $q_{k+1} \neq \rho$, and, by the definition of a run, $((q_k, Q_k), \ell_k, (q_{k+1}, Q_{k+1})) \in \delta_\pi$. When $q_k \neq \rho$ and $q_{k+1} \neq \rho$, then each of (4.15) and (4.17) guarantees that $(q_k, \ell_k, q_{k+1}) \in \delta$.

(d) *We have* $q_i \in F$. We distinguish three cases. If $z = i$ and $Q_i = \emptyset$, then, by (4.13), $(q_i, Q_i) \in F_\pi$ implies $q_i \in F$. If $z = i$ and $Q_i \neq \emptyset$, then, by (4.11), $(q_i, Q_i) \in F_\pi$ implies $q_i \in F$ and $Q_i \subseteq F'$. If $z \neq i$, then we must have $q_{i+1} = \rho$. Hence, by (4.15), $((q_i, Q_i), \ell_i, (q_{i+1}, Q_{i+1})) \in \delta_\pi$ implies $q_i \in F$.

Hence, we conclude $(n_1, n_i) \in [\![\mathcal{A}]\!]_C$.

6. *$\mathcal{A}_\pi$ is a $\{\overline{\pi}\}$-free and* id*-transition-free condition automaton. The condition automata $\mathcal{A}_\pi$ is acyclic whenever $\mathcal{A}$ is acyclic and $\{^*\}$-free.*

By (4.7) and by (4.19) we immediately conclude that $\mathcal{A}_\pi$ is a $\{\overline{\pi}\}$-free and id-transition-free condition automaton and $\mathcal{A}_\pi$ is $\{^*\}$-free whenever $\mathcal{A}$ is $\{^*\}$-free. If $\mathcal{A}$ is acyclic and $\{^*\}$-free, then we can use the proofs of Property 4 or 5 to translate every run of $\mathcal{A}_\pi$ into runs of $\mathcal{A}$ and $\mathcal{A}'$. Using this translation, a run of $\mathcal{A}_\pi$ can only be unbounded in length if runs of $\mathcal{A}$ or of $\mathcal{A}'$ can be unbounded in length. Hence, $\mathcal{A}_\pi$ must be acyclic whenever $\mathcal{A}$ and $\mathcal{A}'$ are acyclic. □

Lemma 4.11 only removes a single projection operator. To fully remove projections, we repeat these removal steps until no projections are left. This leads to the following result:

Theorem 4.12. *Let $\mathcal{F} \subseteq \{\pi, ^*\}$. On labeled chains, we have $\mathcal{N}(\mathcal{F}) \preceq_{\text{bool}} \mathcal{N}(\mathcal{F} - \{\pi\})$.*

Proof. We use Proposition 4.2 to translate an expression to a condition automaton $\mathcal{A}$, then we repeatedly apply Lemma 4.11 to remove conditions, and, finally, we use Proposition 4.3 to translate the resultant $\{\pi\}$-free automaton back to an expression in $\mathcal{N}(\mathcal{F} - \{\pi\})$. Observe that only a finite number of condition removal steps on $\mathcal{A}$ can be made, as Lemma 4.11 guarantees that either $\text{cdepth}(\mathcal{A})$ strictly decreases or else $\text{cdepth}(\mathcal{A})$ does not change and $\text{cweight}(\mathcal{A})$ strictly decreases. □

We observed that Theorem 4.10 does not strictly depend on the graph being a tree. A similar observation holds for Theorem 4.12: for Boolean queries, we can remove a π-condition whenever the condition checks a part of the graph that does not branch. This is the case for π_2-conditions on trees, as trees do not have branching in the direction from a node to its ancestors. For π_1, this observation does not hold, as is illustrated by the proof of Proposition 3.5.

Proposition 4.13. *Let $\mathcal{F} \subseteq \{^*\}$. On labeled trees, we have $\mathcal{N}(\mathcal{F} \cup \{\pi_2\}) \preceq_{\text{bool}} \mathcal{N}(\mathcal{F})$, but $\mathcal{N}(\mathcal{F} \cup \{\pi_1\}) \not\preceq_{\text{bool}} \mathcal{N}(\mathcal{F})$.*

CHAPTER 5

Local queries and condition tree queries[12]

In this chapter, we complete the study on the relative expressive power among the local relation algebra fragments, the fragments $\mathcal{N}(\mathcal{F})$, $\mathcal{F} \subseteq \{^\frown, \pi, \overline{\pi}, \cap, -\}$. In Figure 5.1, we visualize these relationships. In Chapter 3, we have already proven the cases in which we have separations. In Chapter 4, we have already proven the cases in which either intersection or difference are redundant for the downward local fragments. Hence, the only remaining results to prove are the redundancy of either intersection or difference in various non-downward local fragments. To prove these redundancies we introduce *condition tree queries*, a generalization of the tree queries of Wu et al. [97] (see also Proposition 3.36). The step from the tree queries of Wu et al. to condition tree queries is similar to the step from, on the one hand, finite automata that cannot deal with node expression such as projections and coprojections, and, on the other hand, the condition automata of Chapter 4 that can deal with these node expression.

5.1 Organization

First, in Section 5.2 we introduce the condition tree queries and we also prove the close relationships between these condition tree queries and fragments of $\mathcal{N}(^\frown, \pi, \overline{\pi})$. In Sections 5.3–5.5, we use the condition tree queries to prove several redundancies of intersection and difference for path queries on trees, Boolean queries on trees, and, finally, path queries on chains. The results for Boolean queries on chains follow from combining the Boolean results on trees with the path results on chains.

5.2 Condition tree queries

As outlined, we first define condition tree queries syntactically and semantically:

Definition 5.1. A *condition tree query* Q is a tuple $Q = (\mathcal{T}, C, \mathrm{s}, \mathrm{t}, \gamma)$, where $\mathcal{T} = (\mathcal{V}, \Sigma, \mathbf{E})$ is a labeled tree, C is a set of node expressions that represent *node conditions*, $\mathrm{s} \in \mathcal{V}$ is the *source node*, $\mathrm{t} \in \mathcal{V}$ is the *target node*, and $\gamma \subseteq \mathcal{V} \times C$ is the *node-condition relation*. We write $\gamma(n)$ to denote the set $\{e \mid (n, e) \in \gamma\}$.[13] Let $\mathcal{T}' = (\mathcal{V}', \Sigma, \mathbf{E}')$ be a tree. Then $(m, n) \in [\![Q]\!]_{\mathcal{T}'}$ consists of all the node pairs $(m, n) \in \mathcal{V}' \times \mathcal{V}'$ for which there exists a mapping $f : \mathcal{V} \to \mathcal{V}'$ satisfying the following conditions:

[12] The results in this chapter are partly based on the paper "The power of Tarski's relation algebra on trees" [57].

[13] We observe that the condition tree queries without node conditions are equivalent to the tree queries as defined in Wu et al. [97].

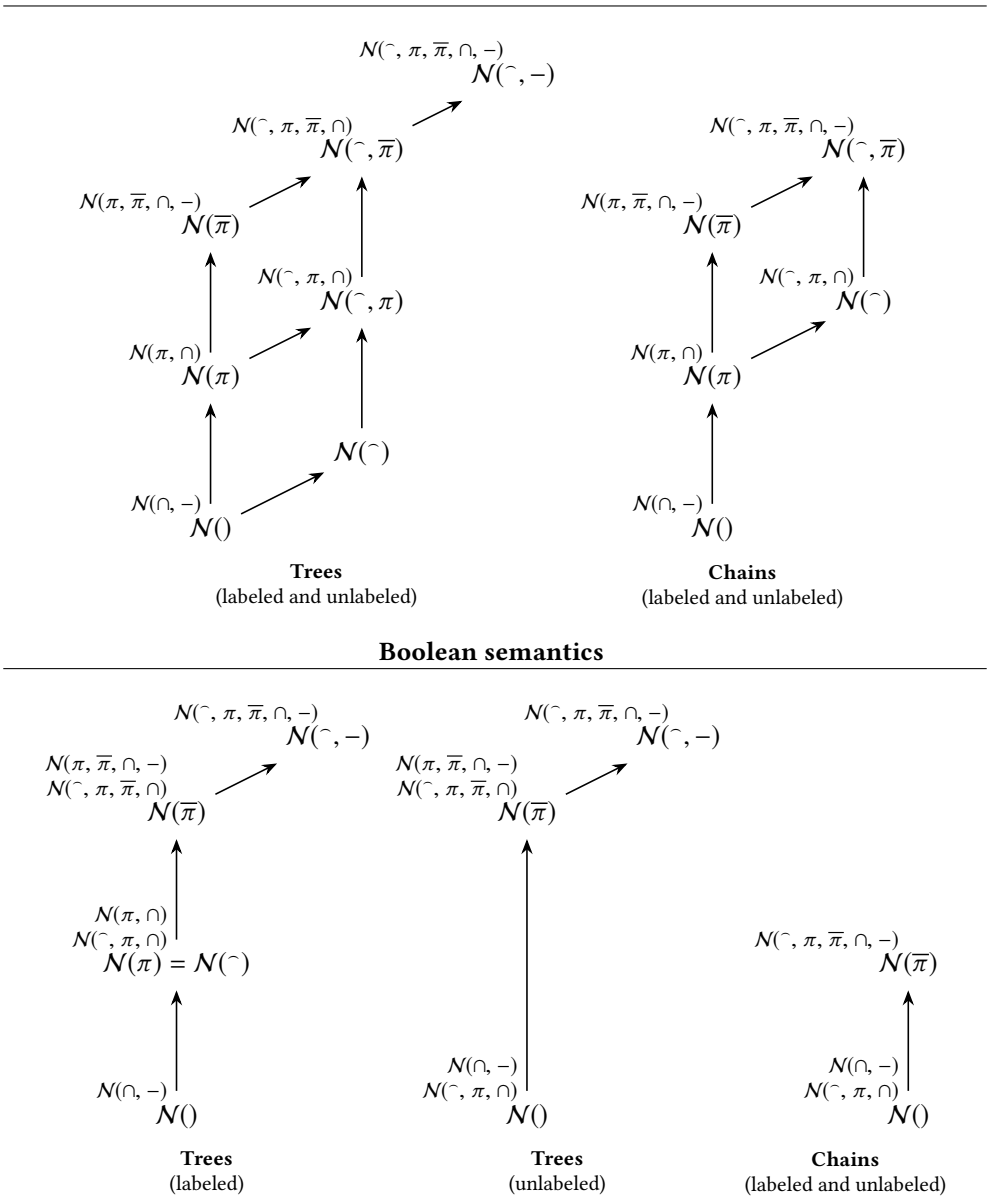


Figure 5.1: Hasse diagrams that visualize the relationships between the local fragments of the relation algebra ($\mathcal{N}(\frown, \pi, \overline{\pi}, \cap, -)$). Each node represents a minimally sized fragment and the superscripts on the left-hand side represent all maximally sized fragments that are equivalent to the fragment represented by the node. Arrows represent strict subsumption relations.

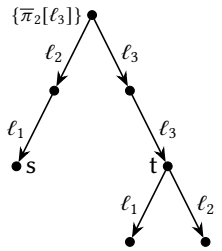


Figure 5.2: A condition tree query with a single condition.

(i) $f(\mathrm{s}) = m$ and $f(\mathrm{t}) = n$;

(ii) for all $v \in \mathcal{V}$ and $e \in \gamma(n)$, $(f(v), f(v)) \in [\![e]\!]_{\mathcal{T}'}$; and

(iii) for all $\ell \in \Sigma$ and $(v, w) \in \mathbf{E}(\ell)$, $(f(v), f(w)) \in \mathbf{E}'(\ell)$.

Intuitively speaking, condition tree queries define intentional relationships between pairs of nodes (m, n) by specifying a tree pattern that connects source node m and target node n.

Example 5.1. The condition tree query Q in Figure 5.2 selects a node pair (s, t) if the following tree traversal steps are all successful: (1) from s, go up via two edges labeled ℓ_1 and ℓ_2; (2) check if the node where we have arrived satisfies the node expression $\overline{\pi}_2[\ell_3]$; (3) from there, go down via two edges labeled ℓ_3, arriving at t; (4) check if t has outgoing edges labeled ℓ_1 and ℓ_2. The condition tree query Q is path-equivalent to the $\mathcal{N}(^\frown, \overline{\pi})$ expression $\ell_1{}^\frown \circ \ell_2{}^\frown \circ \overline{\pi}_2[\ell_3] \circ \ell_3 \circ \ell_3 \circ \pi_1[\ell_1] \circ \pi_1[\ell_2]$.

In the remainder of this section, we formalize the relationship between condition tree queries and relation algebra expressions exhibited in Example 5.1. Thereto, let $\mathcal{F} \subseteq \{^\frown, \pi, \overline{\pi}\}$, and let $\mathbf{Q}_{\mathtt{tree}}(\mathcal{F})$ be the class of all condition tree queries in which node conditions are restricted to $\cup$-free expressions in $\mathcal{N}(\mathcal{F})$. We claim that, for $\underline{\mathcal{F}} = \{^\frown, \pi\}$ and $\underline{\mathcal{F}} = \{^\frown, \pi, \overline{\pi}\}$, $\mathbf{Q}_{\mathtt{tree}}(\mathcal{F})$ and the class of all $\cup$-free expressions in $\mathcal{N}(\mathcal{F})$ path-subsume each other.

First, we show the rewrite from $\mathcal{N}(\mathcal{F})$ to a condition tree query:

Proposition 5.1. *Let $\mathcal{F} \subseteq \{^\frown, \pi, \overline{\pi}\}$ and let e be a $\cup$-free expression in $\mathcal{N}(\mathcal{F})$. There exists a condition tree query Q in $\mathbf{Q}_{\mathtt{tree}}(\mathcal{F})$ such that $e \equiv_{\mathrm{path}} Q$.*

Proof. Due to e being $\cup$-free and in $\mathcal{N}(^\frown, \pi, \overline{\pi})$, it is of the form $t_1 \circ t_2 \circ \cdots \circ t_k$, in which every t_i, $1 \leq i \leq k$, is an expression of the form $\emptyset$, id, ℓ, $\ell^\frown$, or $f_j[e']$ with ℓ an edge label, $f \in \{\pi, \overline{\pi}\}$, and $j \in \{1, 2\}$. Algorithm PRODUCETQ, presented in Figure 5.3, will construct the path-equivalent condition tree query from $t_1 \circ \cdots \circ t_n$ that satisfies this Proposition. It is straightforward to prove that the main for-loop satisfies the loop invariant "the structure $((\mathcal{V}, \Sigma, \mathbf{E}), C, \mathrm{s}, \mathit{current}, \gamma)$ is a condition tree query in $\mathbf{Q}_{\mathtt{tree}}(\mathcal{F})$ and is path-equivalent to $\mathrm{id} \circ t_1 \circ \cdots \circ t_{i-1}$." Hence, termination of the Algorithm yields the desired query Q. □

For the translation from condition tree queries to $\cup$-free expressions, we shall assume that the condition tree queries are restricted to *up-down queries*.

Definition 5.2. A condition tree query $Q = (\mathcal{T}, C, \mathrm{s}, \mathrm{t}, \gamma)$ is an *up-down query* if all edges of $\mathcal{T}$ are on the unique path from s to t not taking into account the direction of the edges.

Algorithm PRODUCETQ($t_1 \circ \cdots \circ t_k, \Sigma$)**:**

1: s := a fresh tree node
2: $\mathcal{V}, \mathbf{E}, C, \gamma(\mathrm{s}), current := \{\mathrm{s}\}, \{\ell \mapsto \emptyset \mid \ell \in \Sigma\}, \emptyset, \emptyset, \mathrm{s}$
3: **for** t_i in $[t_1, \ldots, t_k]$ **do**
4: **if** t_i is of the form ℓ **then**
5: n := a fresh tree node
6: $\mathcal{V}, \ell(\mathbf{E}), current := \mathcal{V} \cup \{n\}, \ell(\mathbf{E}) \cup \{(current, n)\}, n$
7: **else if** t_i is of the form $\ell^\frown$ **then**
8: **if not** $\exists n \exists \ell'$ with $(n, current) \in \mathbf{E}(\ell')$ **then**
9: n := a fresh tree node
10: $\mathcal{V}, \ell(\mathbf{E}), current := \mathcal{V} \cup \{n\}, \ell(\mathbf{E}) \cup \{(n, current)\}, n$
11: **else if** $\exists n \exists \ell'$ with $\ell = \ell'$ and $(n, current) \in \mathbf{E}(\ell')$ **then**
12: $current := n$
13: **else** #($\exists n \exists \ell'$ with $\ell \neq \ell'$ and $(n, current) \in \mathbf{E}(\ell')$).
14: **return** PRODUCETQ($\emptyset, \Sigma$)
15: **else** #(t is a node expression).
16: $C, \gamma(current) := C \cup \{t_i\}, \gamma(current) \cup \{t_i\}$
17: **return** $((\mathcal{V}, \Sigma, \mathbf{E}), C, \mathrm{s}, current, \gamma)$

Figure 5.3: Algorithm PRODUCETQ that translates $\cup$-free expressions in $\mathcal{N}(^\frown, \pi, \overline{\pi})$ to path-equivalent condition tree query (provided a set Σ of possible edge labels).

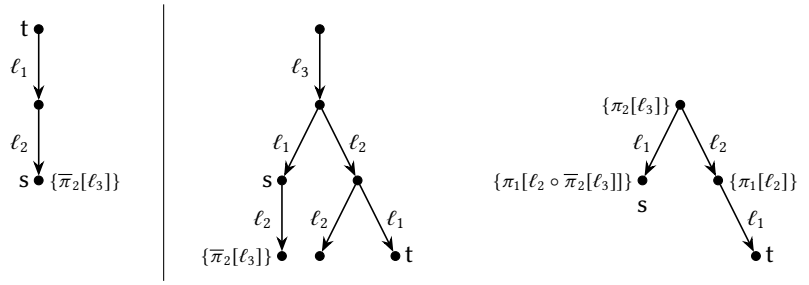


Figure 5.4: The condition tree query on the *left* is up-down. The condition tree query in the *middle* is not, but this query is path-equivalent to the up-down query on the right.

Example 5.2. An up-down query can look like a chain if the target node is an ancestor of the source node, or vice versa, as illustrated by Figure 5.4, *left*. This up-down query is path-equivalent to $\overline{\pi}_2[\ell_3] \circ \ell_2{}^\frown \circ \ell_1{}^\frown$.

The condition tree query in the *middle* is not up-down, but is path-equivalent to the up-down tree query on the *right*. Observe that the *right* query is obtained by pushing parts of the tree traversal described by the *middle* query into node conditions. The *middle* and *right* queries are path-equivalent to $\pi_1[\ell_2 \circ \overline{\pi}_2[\ell_3]] \circ \ell_1{}^\frown \circ \pi_2[\ell_3] \circ \ell_2 \circ \pi_1[\ell_2] \circ \ell_1$.

As illustrated in Example 5.2, we can rewrite a condition tree query to an up-down query by pushing into node conditions those parts of the condition tree query not on the path from source to target. The first step in this is to push all node conditions of a node into a single expression, for which we define the *condition expression*.

Definition 5.3. Let $((\mathcal{V}, \Sigma, \mathbf{E}), C, \mathrm{s}, \mathrm{t}, \gamma)$ be a condition tree query and let $n \in \mathcal{V}$. We define the *condition expression* $\mathsf{conditions}(n)$ of n by

$$\mathsf{conditions}(n) = \bigcap_{e \in \gamma(n)} e$$

when $\gamma(n) \neq \emptyset$ and $\mathsf{conditions}(n) = \mathrm{id}$ otherwise. As every expression in $\gamma(n)$ is a node expression, the expression $\mathsf{conditions}(n)$ is also a node expression. Observe that if $\gamma(n) = \{e_1, \ldots, e_k\}$, then $\mathsf{conditions}(n) \equiv_{\mathrm{path}} e_1 \circ \cdots \circ e_k$. We usually assume that $\mathsf{conditions}(n)$ is written as such a composition of terms instead of an intersection of terms.

We notice that the condition expressions for tree queries are conceptually strongly related to the condition expressions for condition automata (see Definition 4.2).

Lemma 5.2. *Let $\{\pi\} \subseteq \underline{\mathcal{F}} \subseteq \{^\frown, \overline{\pi}, \pi\}$ and let Q be a condition tree query in $\mathbf{Q}_{\mathtt{tree}}(\mathcal{F})$. There exists an up-down query Q' in $\mathbf{Q}_{\mathtt{tree}}(\mathcal{F})$ such that $Q \equiv_{\mathrm{path}} Q'$.*

Proof. Let $Q = (\mathcal{T}, C, \mathrm{s}, \mathrm{t}, \gamma)$ with $\mathcal{T} = (\mathcal{V}, \Sigma, \mathbf{E})$. If Q is not up-down, then there must exist a node $n \in \mathcal{V} - \{\mathrm{s}, \mathrm{t}\}$ such that n is a leaf node or such that n is the root node and has only a single child. In both cases, we are able to remove n from Q. First, we consider the *leaf-prune* of leaf node n. Let $m\ \ell\ n$ be the edge in $\mathcal{T}$ that connects n to its parent m. Remove n from the tree and add the node expression $\pi_1[\ell \circ \mathsf{conditions}(n)]$ to C and $\gamma(m)$. Let Q'' be the result of the leaf-prune. By construction, we have Q'' in $\mathbf{Q}_{\mathtt{tree}}(\mathcal{F})$ and $Q \equiv_{\mathrm{path}} Q''$. Next, we consider the *root-prune* of root node n. Let $n\ \ell\ m$ be the edge in $\mathcal{T}$ that connects n to its single child m. Remove n from the tree and add the node expression $\pi_2[\mathsf{conditions}(n) \circ \ell]$ to C and $\gamma(m)$. Let Q'' be the result of the root-prune. By construction, we have Q'' in $\mathbf{Q}_{\mathtt{tree}}(\mathcal{F})$ and $Q \equiv_{\mathrm{path}} Q''$. By repeatedly applying leaf-pruning and root-pruning until both are no longer possible, we end up with an up-down query that satisfies the Lemma. □

As illustrated in Example 5.2, an up-down query can be translated straightforwardly into a path-equivalent relation algebra expression, provided we have the converse operator ($^\frown$) at our disposal:

Proposition 5.3. *Let $\{^\frown\} \subseteq \mathcal{F} \subseteq \{^\frown, \pi, \overline{\pi}\}$ and let Q be a condition tree query in $\mathbf{Q}_{\mathtt{tree}}(\mathcal{F})$. There exists a $\cup$-free expression e in $\mathcal{N}(\mathcal{F})$ such that $e \equiv_{\mathrm{path}} Q$.*

Proof. Due to Lemma 5.2, we may assume, without loss of generality, that $Q = (\mathcal{T}, C, \mathrm{s}, \mathrm{t}, \gamma)$ is an up-down query. Hence, we can represent Q by two paths $r\ \ell_{\mathrm{s},1}\ m_1 \cdots \ell_{\mathrm{s},u}\ \mathrm{s}$ and $r\ \ell_{\mathrm{t},1}\ n_1 \cdots \ell_{\mathrm{t},d}\ \mathrm{t}$. Observe that if $r = \mathrm{s}$, then the first path reduces to just r and if $r = \mathrm{t}$, then the second path reduces to just r. The expression

$$\mathsf{conditions}(\mathrm{s}) \circ \ell_{\mathrm{s},u}{}^\frown \circ \cdots \circ \mathsf{conditions}(m_1) \circ \ell_{\mathrm{s},1}{}^\frown \circ \mathsf{conditions}(r) \circ \ell_{\mathrm{t},1} \circ \mathsf{conditions}(n_1) \circ \cdots \circ \ell_{\mathrm{t},d} \circ \mathsf{conditions}(\mathrm{t})$$

is in $\mathcal{N}(\mathcal{F})$ and is path-equivalent to Q. □

5.3 Path queries on trees

First, we show the redundancy of intersection in the presence of projection. Consider the following example:

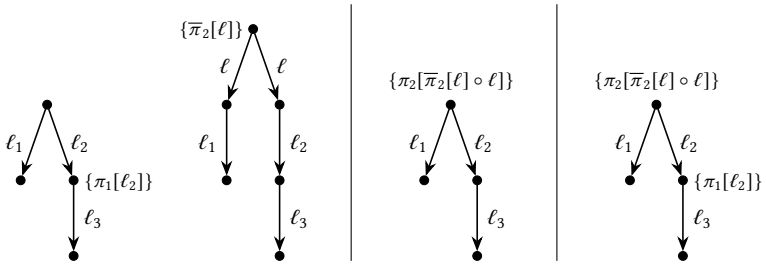


Figure 5.5: The step-by-step intersection of the two up-down queries on the *left*, resulting in the up-down query on the *right*.

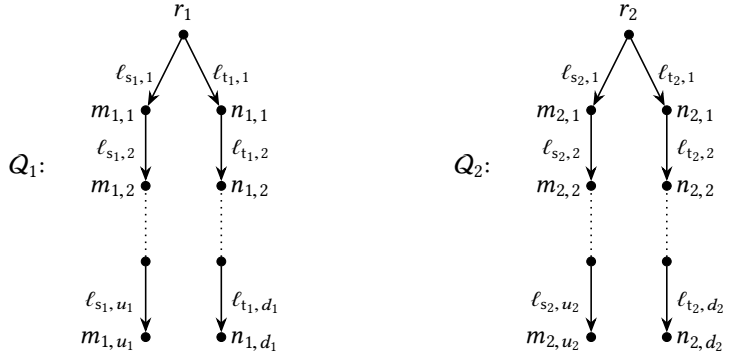


Figure 5.6: Up-down queries Q_1 and Q_2, as used in the proof of Proposition 5.4.

Example 5.3. Suppose we want to compute the intersection of the two up-down queries in Figure 5.5, *left*. Since the two up-down queries have different depths, a pair of nodes of a tree can only be in the result of the intersection of the two queries on that tree if the children of the root of the second query are mapped to the same node. Hence, we can replace the second up-down query by the one shown in the *middle*. Since both queries now have the same shape and corresponding edges have the same label, the intersection is easily obtained by merging the node conditions, resulting in the up-down query on the *right*.

Next, we show that the steps taken in Example 5.3 are sound:

Proposition 5.4. *Let* $\{\pi\} \subseteq \mathcal{F} \subseteq \{\frown, \pi, \overline{\pi}\}$ *and let* Q_1 *and* Q_2 *be condition tree queries in* $\mathbf{Q}_{\mathtt{tree}}(\mathcal{F})$. *There exists a condition tree query* Q *such that, for every tree* $\mathcal{T}$, $[\![Q]\!]_{\mathcal{T}} = [\![Q_1]\!]_{\mathcal{T}} \cap [\![Q_2]\!]_{\mathcal{T}}$.

Proof. Let $Q_1 = (\mathcal{T}_1, C_1, \mathrm{s}_1, \mathrm{t}_1, \gamma_1)$ and $Q_2 = (\mathcal{T}_2, C_2, \mathrm{s}_2, \mathrm{t}_2, \gamma_2)$ be condition tree queries. By Lemma 5.2, we can assume, without loss of generality, that Q_1 and Q_2 are up-down queries. As Q_1 and Q_2 are up-down queries, we can represent Q_1 by two paths $r_1\, \ell_{\mathrm{s}_1,1}\, m_{1,1} \cdots \ell_{\mathrm{s}_1,u_1}\, \mathrm{s}_1$ and $r_1\, \ell_{\mathrm{t}_1,1}\, n_{1,1} \cdots \ell_{\mathrm{t}_1,d_1}\, \mathrm{t}_1$ and Q_2 by two paths $r_2\, \ell_{\mathrm{s}_2,1}\, m_{2,1} \cdots \ell_{\mathrm{s}_2,u_2}\, \mathrm{s}_2$ and $r_2\, \ell_{\mathrm{t}_2,1}\, n_{2,1} \cdots \ell_{\mathrm{t}_2,d_2}\, \mathrm{t}_2$, as visualized by the two tree queries in Figure 5.6.

Let $\mathcal{T}'$ be an arbitrary tree. If $u_1 - d_1 \neq u_2 - d_2$, then $[\![Q_1]\!]_{\mathcal{T}'} \cap [\![Q_2]\!]_{\mathcal{T}'} = \emptyset$. If $u_1 - d_1 = u_2 - d_2$, then we distinguish two cases:

1. $u_1 \neq u_2$ *and* $d_1 \neq d_2$. By symmetry, assume $u_2 > u_1$. Let $\Delta = d_2 - d_1 = u_2 - u_1$.

Now assume that $(v, w) \in [\![Q_1]\!]_{\mathcal{T}'} \cap [\![Q_2]\!]_{\mathcal{T}'}$. Consider the mapping f from Q_2 to $\mathcal{T}'$ that shows $(v, w) \in [\![Q_2]\!]_{\mathcal{T}'}$. Due to $(v, w) \in [\![Q_1]\!]_{\mathcal{T}'}$, this mapping f must map the first Δ nodes on the paths $r_2 \dots s_2$ and $r_2 \dots t_2$ to the same Δ nodes in $\mathcal{T}'$. Hence, if for some i, $1 \leq i \leq \Delta$, $\ell_{s_2,i} \neq \ell_{t_2,i}$, then $[\![Q_1]\!]_{\mathcal{T}'} \cap [\![Q_2]\!]_{\mathcal{T}'} = \emptyset$. Thus, assume, for all i, $1 \leq i \leq \Delta$, that $\ell_{s_2,i} = \ell_{t_2,i}$. We can embed this restriction on the nodes $m_{2,1}, \dots, m_{2,\Delta}$ and $n_{2,1}, \dots, n_{2,\Delta}$ by replacing Q_2 by the up-down query Q_2' represented by two paths $r\ \ell_{s_2,\Delta+1}\ m_{2,\Delta+1} \cdots \ell_{2,u_2}\ s_2$ and $r\ \ell_{t_2,\Delta+1}\ n_{2,\Delta+1} \cdots \ell_{t_2,d_2}\ t_2$ with

$$\gamma(r) = \pi_2[\mathsf{conditions}(r_2) \circ \ell_{s_2,1} \circ \mathsf{conditions}(m_{2,1}) \circ \mathsf{conditions}(n_{2,1}) \circ \cdots \circ \ell_{s_2,\Delta} \circ \mathsf{conditions}(m_{2,\Delta}) \circ \mathsf{conditions}(n_{2,\Delta})].$$

On Q_1 and Q_2' the conditions of the next case apply.

2. *$u_1 = u_2$ and $d_1 = d_2$* In this case, $[\![Q_1]\!]_{\mathcal{T}'} \cap [\![Q_2]\!]_{\mathcal{T}'} = \emptyset$ whenever $\ell_{s_1,i} \neq \ell_{s_2,i}$, $1 \leq i \leq u_1 = u_2$, or $\ell_{t_1,i} \neq \ell_{t_2,i}$, $1 \leq i \leq d_1 = d_2$. In all other cases, we define $Q = (\mathcal{T}_1, C_1 \cup C_2, s_1, t_1, \gamma')$, in which

$$\begin{aligned}
\gamma'(s_1) &= \gamma(s_1) \cup \gamma(s_2); \\
\gamma'(t_1) &= \gamma(t_1) \cup \gamma(t_2); \\
\gamma'(r_1) &= \gamma(r_1) \cup \gamma(r_2); \\
\gamma'(m_{1,i}) &= \gamma_1(m_{1,i}) \cup \gamma_2(m_{2,i}) && \text{with } 1 \leq i \leq u_1 = u_2; \\
\gamma'(n_{1,i}) &= \gamma_1(n_{1,i}) \cup \gamma_2(n_{2,i}) && \text{with } 1 \leq i \leq d_1 = d_2.
\end{aligned}$$

□

We can use Proposition 5.4 directly to remove intersection at the top level in ∪-free expressions, this via a translation to condition tree queries. To remove other occurrences of intersection, a straightforward induction argument on the structure of expressions suffices.

Theorem 5.5. *Let $\{^\frown, \pi\} \subseteq \mathcal{F} \subseteq \{^\frown, \pi, \overline{\pi}, \cap\}$. On labeled trees, every ∪-free expression in $\mathcal{N}(\mathcal{F})$ is path-equivalent to a ∪-free expression in $\mathcal{N}(\mathcal{F} - \{\cap\})$.*

Proof. The proof is by induction on the length of e. The base cases are the atoms, which are all already in $\mathcal{N}(\mathcal{F} - \{\cap\})$. We assume that, for all ∪-free expressions e' in $\mathcal{N}(\mathcal{F})$ of length at most i, there exists a ∪-free expression e'' in $\mathcal{N}(\mathcal{F} - \{\cap\})$ such that $e' \equiv_{\text{path}} e''$.

Let e be a ∪-free expression in $\mathcal{N}(\mathcal{F})$ and not in $\mathcal{N}(\mathcal{F} - \{\cap\})$ of length $i+1$. We distinguish the following three cases:

1. $e = e_1 \circ e_2$. Use the inductive hypothesis on e_1 and e_2 to obtain ∪-free expressions e_1' and e_2' in $\mathcal{N}(\mathcal{F} - \{\cap\})$ with $e_1 \equiv_{\text{path}} e_1'$ and e_2', $e_1 \equiv_{\text{path}} e_2'$. Hence, we have $e \equiv_{\text{path}} e_1' \circ e_2'$.

2. *$e = f_j[e']$ with $f \in \{\pi, \overline{\pi}\}$ and $j \in \{1, 2\}$.* Use the induction hypothesis on e' to obtain a ∪-free expression e'' in $\mathcal{N}(\mathcal{F} - \{\cap\})$ with $e' \equiv_{\text{path}} e''$. Hence, we have $e \equiv_{\text{path}} f_j[e'']$.

3. $e = e_1 \cap e_2$. Use the inductive hypothesis on e_1 and e_2 to obtain ∪-free expressions e_1' and e_2' in $\mathcal{N}(\mathcal{F} - \{\cap\})$ with $e_1 \equiv_{\text{path}} e_1'$ and $e_2 \equiv_{\text{path}} e_2'$. By Proposition 5.1 and Lemma 5.2, we can construct up-down queries Q_1 and Q_2 in $\mathbf{Q}_{\mathtt{tree}}(\mathcal{F} - \{\cap\})$ with $e_1' \equiv_{\text{path}} Q_1$ and $e_2' \equiv_{\text{path}} Q_2$. By Proposition 5.4, we can construct up-down query Q in $\mathbf{Q}_{\mathtt{tree}}(\mathcal{F} - \{\cap\})$ such that, for every tree $\mathcal{T}$, $[\![Q]\!]_{\mathcal{T}} = [\![Q_1]\!]_{\mathcal{T}} \cap [\![Q_2]\!]_{\mathcal{T}}$. Finally, by Proposition 5.3, we can construct a ∪-free expression e' in $\mathcal{N}(\mathcal{F} - \{\cap\})$ such that $e' \equiv_{\text{path}} Q$. By the above construction, we conclude $e \equiv_{\text{path}} e'$. □

Combining Theorem 5.5 with Lemma 3.2 (i) yields the redundancy of intersection for local queries, as presented in Figure 5.1.

Corollary 5.6. *Let* $\{^\frown, \pi\} \subseteq \mathcal{F} \subseteq \{^\frown, \pi, \overline{\pi}, \cap\}$. *On labeled trees, we have* $\mathcal{N}(\mathcal{F}) \equiv_{\text{path}} \mathcal{N}(\mathcal{F} - \{\cap\})$.

5.4 Boolean queries on trees

Fletcher et al. [31–34, 87] already showed that, on graphs, the converse operator $^\frown$ is in many cases subsumed by the projection operators (Proposition 3.34). On labeled trees, this result can be strengthened by showing that projections and the converse operator have equivalent expressive power in Boolean queries. To prove this, we need an intermediate result on $\mathbf{Q}_{\mathtt{tree}}()$:

Proposition 5.7. *For every condition tree query* Q *in* $\mathbf{Q}_{\mathtt{tree}}()$, *there exists a* $\cup$*-free expression in* $\mathcal{N}(^\frown)$ *such that* $Q \equiv_{\text{bool}} e$.

Proof. Let $Q = ((\mathcal{V}, \Sigma, \mathbf{E}), C, \mathrm{s}, \mathrm{t}, \gamma)$ and let r be the root node of tree $(\mathcal{V}, \Sigma, \mathbf{E})$. Construct $Q_r = ((\mathcal{V}, \Sigma, \mathbf{E}), C, r, r, \gamma)$. Observe that Q_r in $\mathbf{Q}_{\mathtt{tree}}()$ and we have, for every tree $\mathcal{T}$, $[\![Q]\!]_{\mathcal{T}} \neq \emptyset$ if and only if $[\![Q_r]\!]_{\mathcal{T}} \neq \emptyset$. Hence, we have $Q \equiv_{\text{bool}} Q_r$.

Before we translate Q_r to an expression in $\mathcal{N}(^\frown)$, we reduce Q_r to a single node. We do so by a leaf-prune construction based on the construction of Lemma 5.2. Let $n \in \mathcal{V} - \{r\}$ be a leaf node and let $m\,\ell\,n$ be the edge in $\mathcal{T}$ that connects n to its parent m. As conditions(n) is a node expression, we have $\pi_1[\ell \circ \text{conditions}(n)] \equiv_{\text{path}} \ell \circ \text{conditions}(n) \circ \ell^\frown$. If conditions$(n)$ in $\mathcal{N}(^\frown)$, then also $\ell \circ \text{conditions}(n) \circ \ell^\frown$ in $\mathcal{N}(^\frown)$. Remove n from the tree and add the node expression $\ell \circ \text{conditions}(n) \circ \ell^\frown$ to C, $\gamma(m)$, and let Q'' be the result. By construction, we have Q'' in $\mathbf{Q}_{\mathtt{tree}}(^\frown)$ and $Q_r \equiv_{\text{path}} Q''$.

Let Q' be a condition tree query in $\mathbf{Q}_{\mathtt{tree}}(^\frown)$ obtained by repeating the above leaf-prune operations on Q_r until only the root node r' remains. By construction, we have $Q_r \equiv_{\text{path}} Q' \equiv_{\text{path}} \text{conditions}(r')$. As Q' in $\mathbf{Q}_{\mathtt{tree}}(^\frown)$, we have conditions$(r')$ in $\mathcal{N}(^\frown)$ and we conclude $Q \equiv_{\text{bool}} \text{conditions}(r')$. □

Next, we observe that the class of condition tree queries $\mathbf{Q}_{\mathtt{tree}}()$, which consists of all condition-free condition tree queries, is equivalent to the tree queries of Wu et al. [97, Theorem 4.1]. Hence, we have

Lemma 5.8 (Wu et al.)**.** *For every* $\cup$*-free expression in* $\mathcal{N}(\pi)$, *there exists a condition tree query* Q *in* $\mathbf{Q}_{\mathtt{tree}}()$ *with* $e \equiv_{\text{path}} Q$.

Combining Proposition 5.7, Lemma 5.8, and Lemma 3.2 (i) yields the equivalence of projection and converse for local queries, as presented in Figure 5.1.

Corollary 5.9. *On labeled trees, we have* $\mathcal{N}(\pi) \equiv_{\text{bool}} \mathcal{N}(^\frown)$.

5.5 Path queries on chains

Figure 5.1 shows two main redundancies on chains that are not present on trees: the equivalence of projection to converse, which we already proved in Proposition 3.13, and the redundancy of difference in the presence of coprojection. For proving the redundancy of difference, we once again use condition tree queries. When querying trees, we have already

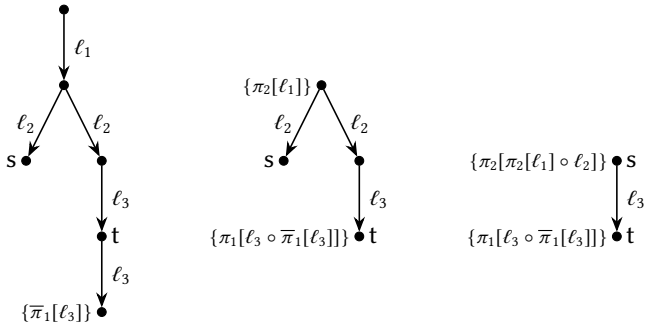


Figure 5.7: A condition tree query on the *left*, a path-equivalent up-down query in the *middle*, and a chain query on the *right*. On chains, the chain query is path-equivalent to the other two queries.

seen that condition tree queries can be normalized into the up-down queries. On chains, we can normalize up-down queries further to *chain queries*:

Definition 5.4. A condition tree query $Q = (\mathcal{T}, C, \mathrm{s}, \mathrm{t}, \gamma)$ is a *chain query* if $\mathcal{T}$ consists of a single path from the root node r to the leaf node l such that $\{r, l\} = \{\mathrm{s}, \mathrm{t}\}$.

Example 5.4. Of course, every chain query is an up-down query, but not every up-down query is a chain query. The up-down query in Figure 5.4, *left*, is a chain query, whereas the up-down query in Figure 5.4, *right*, is not a chain query. The query in Figure 5.7, *left*, is a condition tree query which is neither an up-down query nor a chain query. The query in the *middle* is the up-down query path-equivalent to the query on the *left*. On chains, the queries on the *left* and *middle* are path-equivalent to the chain query on the *right*. This path equivalence does not hold on general trees, however. The chain query on the *right* is obtained by merging the two paths in the up-down query in the *middle*, after which root-pruning is applied.

As observed in Example 5.4, we can rewrite up-down queries to chain queries by merging the two paths from the root to s and t. Next, we prove soundness of this algorithm.

Lemma 5.10. *Let $\{\pi\} \subseteq \mathcal{F} \subseteq \{^\frown, \overline{\pi}, \pi\}$ and let Q be an up-down query in $\mathbf{Q}_{\mathtt{tree}}(\mathcal{F})$. There exists a chain query Q' in $\mathbf{Q}_{\mathtt{tree}}(\mathcal{F})$ such that $Q \equiv_{\mathrm{path}} Q'$ on labeled chains.*

Proof. Without loss of generality, we assume that $Q = (\mathcal{T}, C, \mathrm{s}, \mathrm{t}, \gamma)$ is not yet a chain query, but is up-down. We represent Q by two paths $r\ \ell_{\mathrm{s},1}\ n_{\mathrm{s},1} \cdots \ell_{\mathrm{s},u}\ n_{\mathrm{s},u}$ with $\mathrm{s} = n_{\mathrm{s},u}$ and $r\ \ell_{\mathrm{t},1}\ n_{\mathrm{t},1} \cdots \ell_{\mathrm{t},d}\ n_{\mathrm{t},d}$ with $\mathrm{t} = n_{\mathrm{t},d}$. As Q is not a chain query, we have $r \neq \mathrm{s}$ and $r \neq \mathrm{t}$. Let C' be a chain. We have $[\![Q]\!]_{C'} = \emptyset$ whenever $\ell_{\mathrm{s},i} \neq \ell_{\mathrm{t},i}$, $1 \leq i \leq \min(u, d)$.

Let $k_\mathrm{s} = u$ and $k_\mathrm{t} = d$ and choose z and z' out of s, t such that the path from r to z is at least as long as the path from r to z'. We have $k_z = \max(u, d)$ and $k_{z'} = \min(u, d)$. We define $Q'' = (C'', C, n_{z,u}, n_{z,d}, \gamma')$, in which C'' is the chain $r\ \ell_{z,1}\ n_{z,1} \cdots \ell_{z,k_z}\ n_{z,k_z}$ and

$$\begin{aligned}
\gamma'(r) &= \gamma(r); \\
\gamma'(n_{z,i}) &= \gamma(n_{z,i}) \cup \gamma(n_{z',i}) && \text{with } 1 \leq i \leq \min(u, d); \\
\gamma'(n_{z,i}) &= \gamma(n_{z,i}) && \text{with } \min(u, d) < i \leq \max(u, d).
\end{aligned}$$

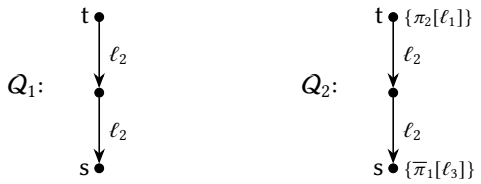


Figure 5.8: Two simple chain queries. The query on the *left* represents the query $\ell_2{}^\frown \circ \ell_2{}^\frown$ and the query on the *right* represents the query $\overline{\pi}_1[\ell_3] \circ \ell_2{}^\frown \circ \ell_2{}^\frown \circ \pi_2[\ell_1]$.

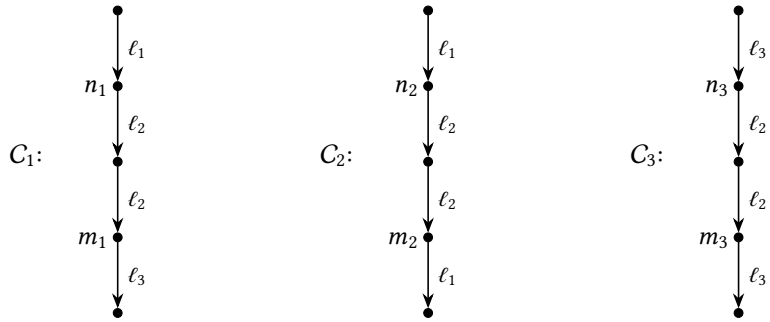


Figure 5.9: Three labeled chains, each satisfying slightly different conditions.

It is straightforward to verify that $Q'' \equiv_{\text{path}} Q$. The condition tree query Q'' is no longer an up-down query, as the root is neither s nor t. To construct a chain query out of Q'', we apply the root-prune constructions of Lemma 5.2 until the root is either s or t, resulting in the query Q'. By construction, Q' in $\mathbf{Q}_{\mathtt{tree}}(\mathcal{F})$, $Q' \equiv_{\text{path}} Q$, and Q' is a chain query. □

We introduce chain queries to help remove differences from expression in $\mathcal{N}(^\frown, \pi, \overline{\pi}, -)$. Unfortunately, chain queries are not closed under difference, which we illustrate in the following example.

Example 5.5. Let Q_1 and Q_2 be the chain queries in Figure 5.8. We can easily construct a chain C such that $[\![Q_1]\!]_C - [\![Q_2]\!]_C \neq \emptyset$. Take, for example, the chains C_1, C_2, and C_3 in Figure 5.9.

We have $(m_1, n_1) \notin [\![Q_1]\!]_{C_1} - [\![Q_2]\!]_{C_1}$ as m_1 satisfies the node condition $\overline{\pi}_1[\ell_3]$ and n_1 satisfies the node condition $\pi_2[\ell_1]$. We have $(m_2, n_2) \in [\![Q_1]\!]_{C_2} - [\![Q_2]\!]_{C_2}$ as m_2 does not satisfy the node condition $\overline{\pi}_1[\ell_3]$ and we have $(m_3, n_3) \in [\![Q_1]\!]_{C_3} - [\![Q_2]\!]_{C_3}$ as n_3 does not satisfy the node condition $\pi_2[\ell_1]$. Combining these observations leads us to conclude that $(m, n) \in [\![Q_1]\!]_C - [\![Q_2]\!]_C$ if the following two conditions are met: (1) $(m, n) \in [\![Q_1]\!]_C$, which is the case when a path $n\ \ell_2\ v\ \ell_2\ m$ exists in C; and (2) m does not satisfy the node condition $\overline{\pi}_1[\ell_3]$, n does not satisfy the node condition $\pi_2[\ell_1]$, or both.

Observe that the second condition is a disjunction of conditions. The standard way to express a disjunction of conditions is by a union of queries. Indeed, the simplest expression in $\mathcal{N}(^\frown, \pi, \overline{\pi})$ path-equivalent to the difference of Q_1 and Q_2 is $(\pi_1[\ell_3] \circ \ell_2{}^\frown \circ \ell_2{}^\frown) \cup (\ell_2{}^\frown \circ \ell_2{}^\frown \circ \overline{\pi}_2[\ell_1])$. Unfortunately, the chain queries are, by definition, not closed under union.

In Example 5.5, we observed that the difference of two specific chain queries can be expressed by a finite union of expressions (or chain queries). Next, we show that this

observation holds in general.

Proposition 5.11. *Let $\{\overline{\pi}\} \subseteq \mathcal{F} \subseteq \{^\frown, \pi, \overline{\pi}\}$ and let Q_1 and Q_2 be condition tree queries in $\mathbf{Q}_{\mathtt{tree}}(\mathcal{F})$. There exists a finite set of chain queries S such that, for every chain C, $\bigcup_{Q\in S} [\![Q]\!]_C = [\![Q_1]\!]_C - [\![Q_2]\!]_C$.*

Proof. Observe that $\overline{\pi} \in \mathcal{F}$ implies that we can express projections and we assume that $\pi \in \mathcal{F}$. Due to Lemma 5.10, we can assume, without loss of generality, that $Q_1 = (\mathcal{T}_1, C_1, \mathrm{s}_1, \mathrm{t}_1, \gamma_1)$ and $Q_2 = (\mathcal{T}_2, C_2, \mathrm{s}_2, \mathrm{t}_2, \gamma_2)$ are chain queries. As Q_1 and Q_2 are chain queries, we can represent Q_1 by the path $n_{1,0}\, \ell_{1,1}\, n_{1,1} \cdots \ell_{1,k_1}\, n_{1,k_1}$ and Q_2 by the path $n_{2,0}\, \ell_{2,1}\, n_{2,1} \cdots \ell_{2,k_2}\, n_{2,k_2}$. We distinguish the following three cases:

1. $k_1 \neq k_2$. Observe that $(m,n) \in [\![Q_1]\!]_C$ only if $\|m \leftrightarrow n\|_C = k_1$ and $(m,n) \in [\![Q_2]\!]_C$ only if $\|m \leftrightarrow n\|_C = k_2$. Hence, $[\![Q_1]\!]_C$ and $[\![Q_2]\!]_C$ do not overlap for any chain C and we have $S = \{Q_1\}$.

2. $k_1 = k_2 > 0$ *and either* $n_{1,0} = \mathrm{s}_1$, $n_{2,0} = \mathrm{t}_2$ *or* $n_{1,0} = \mathrm{t}_1$, $n_{2,0} = \mathrm{s}_2$. We assume $n_{1,0} = \mathrm{s}_1$, $n_{2,0} = \mathrm{t}_2$, the other case is analogous. We have $(m,n) \in [\![Q_1]\!]_C$ only if $\|m \to n\|_C > 0$ and $(m,n) \in [\![Q_2]\!]_C$ only if $\|m \to n\|_C < 0$. Hence, $[\![Q_1]\!]_C$ and $[\![Q_2]\!]_C$ do not overlap for any chain C and we have $S = \{Q_1\}$.

3. $k_1 = k_2$, *and either* $n_{1,0} = \mathrm{s}_1$, $n_{2,0} = \mathrm{s}_2$ *or* $n_{1,0} = \mathrm{t}_1$, $n_{2,0} = \mathrm{t}_2$. In this case, $[\![Q_1]\!]_C - [\![Q_2]\!]_C = [\![Q_1]\!]_C$ for every chain C whenever $\ell_{1,i} \neq \ell_{2,i}$, $1 \leq i \leq k_1 = k_2$ and we have $S = \{Q_1\}$. In all other cases, we define the condition tree queries $Q_{j,e} = (\mathcal{T}_1, C \cup \{\overline{\pi}_1[e]\}, \mathrm{s}_1, \mathrm{t}_1, \gamma_{j,e})$ with $1 \leq j \leq k_1$, $e \in \gamma_2(n_{2,j})$, and $\gamma_{j,e} = \gamma$ except that $\gamma_{j,e}(n_{1,j}) = \gamma(n_{1,j}) \cup \{\overline{\pi}_1[e]\}$. Let $S = \{Q_{j,e} \mid (1 \leq j \leq k_1) \wedge (e \in \gamma_2(n_{2,j}))\}$. Let $C = (\mathcal{V}, \Sigma, \mathbf{E})$ be a chain and $m, n \in \mathcal{V}$. From a straightforward analysis on the cases for which $(m,n) \in [\![Q_1]\!]_C$ holds while $(m,n) \in [\![Q_2]\!]_C$ not holds, it follows that $\bigcup_{Q\in S} [\![Q]\!]_C = [\![Q_1]\!]_C - [\![Q_2]\!]_C$. □

We can use Proposition 5.11 directly to remove difference at the top level in $\cup$-free expressions, this via a translation to condition tree queries. To remove other occurrences of difference, a straightforward induction argument on the structure of expressions suffices.

Theorem 5.12. *Let $\{^\frown, \overline{\pi}\} \subseteq \mathcal{F} \subseteq \{^\frown, \pi, \overline{\pi}, \cap, -\}$. On labeled chains, every $\cup$-free expression in $\mathcal{N}(\mathcal{F})$ is path-equivalent to a finite union of $\cup$-free expressions in $\mathcal{N}(^\frown, \overline{\pi})$.*

Proof. We have $\mathcal{F} \subseteq \{^\frown, \overline{\pi}, -\}$. Hence, we assume $\mathcal{F} = \{^\frown, \overline{\pi}, -\}$. The proof is by induction on the length of e. The base cases are the atoms, which are all already in $\mathcal{N}(^\frown, \overline{\pi})$. We assume that, for all $\cup$-free expressions e' in $\mathcal{N}(\mathcal{F})$ of length at most i, there exists a finite set of $\cup$-free expressions S' such that $\bigcup_{e''\in S'} e''$ in $\mathcal{N}(^\frown, \overline{\pi})$ and $e' \equiv_{\mathrm{path}} \bigcup_{e''\in S'} e''$.

Let e be a $\cup$-free expression of length $i+1$, in $\mathcal{N}(\mathcal{F})$, and not in $\mathcal{N}(^\frown, \overline{\pi})$. We distinguish the following three cases:

1. $e = e_1 \circ e_2$. Use the inductive hypothesis on e_1 and e_2 to obtain sets of $\cup$-free expressions S_1 and S_2 in $\mathcal{N}(^\frown, \overline{\pi})$ such that $e_1 \equiv_{\mathrm{path}} \bigcup_{e'_1\in S_1} e'_1$ and $e_2 \equiv_{\mathrm{path}} \bigcup_{e'_2\in S_2} e'_2$. Let $S = \{e'_1 \circ e'_2 \mid e'_1 \in S_1 \wedge e'_2 \in S_2\}$. The set S is a finite set of $\cup$-free expressions in $\mathcal{N}(^\frown, \overline{\pi})$ and we have $e \equiv_{\mathrm{path}} \bigcup_{e'\in S} e'$.

2. $e = \overline{\pi}_j[e']$ *with* $j \in \{1,2\}$. Use the induction hypothesis on e' to obtain the set of $\cup$-free expressions $S' = \{e'_1, \ldots, e'_k\}$ in $\mathcal{N}(^\frown, \overline{\pi})$ such that $e' \equiv_{\mathrm{path}} \bigcup_{1\leq i\leq k} e'_i$. Let $S =$

$\{\overline{\pi}_j[e'_1] \circ \cdots \circ \overline{\pi}_j[e'_k]\}$.[14] The set S is a finite set of $\cup$-free expressions in $\mathcal{N}(\frown, \overline{\pi})$ and we have $e \equiv_{\text{path}} \bigcup_{e' \in S} e'$.

3. $e = e_1 - e_2$. Use the inductive hypothesis on e_1 and e_2 to obtain sets of $\cup$-free expressions S_1 and S_2 in $\mathcal{N}(\frown, \overline{\pi})$ such that $e_1 \equiv_{\text{path}} \bigcup_{e'_1 \in S_1} e'_1$ and $e_2 \equiv_{\text{path}} \bigcup_{e'_2 \in S_2} e'_2$. Observe that

$$e \equiv_{\text{path}} \Big(\bigcup_{e'_1 \in S_1} e'_1\Big) - \Big(\bigcup_{e'_2 \in S_2} e'_2\Big) \equiv_{\text{path}} \bigcup_{e'_1 \in S_1} \Big(e'_1 - \Big(\bigcup_{e'_2 \in S_2} e'_2\Big)\Big) \equiv_{\text{path}} \bigcup_{e'_1 \in S_1} \Big(\bigcap_{e'_2 \in S_2} e'_1 - e'_2\Big).$$

Let $e'_1 \in S'_1$ and $e'_2 \in S'_2$. Use Proposition 5.1 and Lemma 5.10 to translate e'_1 and e'_2 to chain queries Q'_1 and Q'_2 and use Proposition 5.11 to construct a finite set of condition tree queries S' in $\mathbf{Q}_{\texttt{tree}}(\frown, \overline{\pi})$ such that, for every chain C, $\bigcup_{Q \in S} [\![Q]\!]_C = [\![Q'_1]\!]_C - [\![Q'_2]\!]_C$. Use Proposition 5.3 to translate every condition tree query in S' to a $\cup$-free expression in $\mathcal{N}(\frown, \overline{\pi})$. By $d(e'_1, e'_2)$, we denote the finite set of expressions resulting from translating S' to $\cup$-free expressions in $\mathcal{N}(\frown, \overline{\pi})$. We have

$$e \equiv_{\text{path}} \bigcup_{e'_1 \in S_1} \Big(\bigcap_{e'_2 \in S_2} \Big(\bigcup_{e' \in d(e'_1, e'_2)} e'\Big)\Big),$$

and this expression is in $\mathcal{N}(\frown, \overline{\pi}, \cap)$. Using Corollary 5.6, we rewrite the above expression into a path-equivalent $\cap$-free expression e' in $\mathcal{N}(\frown, \overline{\pi})$. Finally, using Lemma 3.2 (i), we can rewrite e' into a set S of $\cup$-free expressions in $\mathcal{N}(\frown, \overline{\pi})$ such that $e \equiv_{\text{path}} \bigcup_{e'' \in S} e''$. □

Finally, combining Theorem 5.12 with Lemma 3.2 (i) yields the following:

Corollary 5.13. *Let $\{\frown, \overline{\pi}\} \subseteq \mathcal{F} \subseteq \{\frown, \pi, \overline{\pi}, \cap, -\}$. On labeled chains, we have $\mathcal{N}(\mathcal{F}) \equiv_{\text{path}} \mathcal{N}(\frown, \overline{\pi})$.*

[14] We observe that $\overline{\pi}_j[e'_i]$, $1 \leq i \leq k$, is a node expression, hence the composition $\overline{\pi}_j[e'_1] \circ \cdots \circ \overline{\pi}_j[e'_k]$ is path-equivalent to $\overline{\pi}_j[e'_1] \cap \cdots \cap \overline{\pi}_j[e'_k]$. Observe that this expression is a basic complement-step on a union of projection-terms.

CHAPTER 6

Conclusion, discussion, and future work

In Part II, we set out to improve our understanding of querying tree data by studying the relative expressive power of queries in fragments of the relation algebra when used to query trees. Along the way, we also studied chains to gain additional insights in the hard-to-prove cases. We settled the relative expressive power of the downward fragments, i.e. fragments of $\mathcal{N}(\pi, \overline{\pi}, \cap, -, {}^*)$, and the local fragments, i.e. fragments of $\mathcal{N}(\hat{}, \pi, \overline{\pi}, \cap, -)$. For these language fragments, we provided full Hasse diagrams of the relative expressive power (see Figure 4.1 and Figure 5.1, respectively). For both the downward and the local fragments, we were able to establish several cases in which intersection and difference are redundant and do not add expressive power at all. With respect to Boolean queries, several other redundancies have been established, especially on unlabeled structures.

While the relative simplicity of the tree data model in comparison with the general graph data model seems to suggest that the study on trees ought to be a less challenging endeavor, it turned out that the opposite is the case. Indeed, we can show that the techniques used to prove relative expressive power results on graphs (see, e.g., [32]), are only applicable in a few cases when proving results on trees and chains. For queries outside the downward and local fragments, our effort was primarily focused on the tree data model, for which only a handful of cases remain unsolved. The main open cases involve adding difference to already rich non-downward, non-local fragments of the relation algebra:

Problem 6.1. *Let* $\{\mathrm{di}, \overline{\pi}, \cap\} \subseteq \mathcal{F} \subseteq \{\mathrm{di}, \hat{}, \pi, \overline{\pi}, \cap, {}^*\}$ *and let* $z \in \{\mathrm{bool}, \mathrm{path}\}$. *With respect to either labeled trees, unlabeled trees, labeled chains, or unlabeled chains, do we have a collapse* $\mathcal{N}(\mathcal{F} \cup \{-\}) \preceq_z \mathcal{N}(\mathcal{F})$ *or not?*

Observe that Problem 6.1 describes 32 individual cases. Of these cases, only two cases have been fully solved: by Theorem 3.29, we have $\mathcal{N}(\mathrm{di}, \hat{}, \pi, \overline{\pi}, \cap, -) \preceq_{\mathrm{bool}} \mathcal{N}(\mathrm{di}, \hat{}, \pi, \overline{\pi}, \cap)$ and $\mathcal{N}(\mathrm{di}, \pi, \overline{\pi}, \cap, -) \preceq_{\mathrm{bool}} \mathcal{N}(\mathrm{di}, \pi, \overline{\pi}, \cap)$ on unlabeled chains. Unfortunately, the proof of Theorem 3.29 does not generalize to any of the other cases.

Besides the above open problem, all other open problems on trees involve the relative expressive power of fragments $\mathcal{N}(\mathcal{F}_1 \cup \{^*\})$ and $\mathcal{N}(\mathcal{F}_2 \cup \{^*\})$ for cases in which we have already proven either $\mathcal{N}(\mathcal{F}_1) \preceq_{\mathrm{bool}} \mathcal{N}(\mathcal{F}_2)$, $\mathcal{N}(\mathcal{F}_1) \preceq_{\mathrm{path}} \mathcal{N}(\mathcal{F}_2)$, or both.

Problem 6.2. *Let* $\mathcal{F}_1, \mathcal{F}_2 \subseteq \{\mathrm{di}, \hat{}, \pi, \overline{\pi}, \cap, -\}$ *and let* $z \in \{\mathrm{path}, \mathrm{bool}\}$. *With respect to either labeled trees, unlabeled trees, labeled chains, or unlabeled chains, in which cases does* $\mathcal{N}(\mathcal{F}_1) \preceq_z \mathcal{N}(\mathcal{F}_2)$ *imply* $\mathcal{N}(\mathcal{F}_1 \cup \{^*\}) \preceq_z \mathcal{N}(\mathcal{F}_2 \cup \{^*\})$*?*

Problem 6.2 has been answered in the positive for all downward fragments. With respect

Figure 6.1: The 3-clique graph $\mathcal{G}_3$ and the bow-tie graph $\mathcal{G}_{\bowtie}$. These graphs can be distinguished using difference, but not without difference.

to the non-downward local fragments on trees, four cases remain open for path queries and eleven cases remain open for Boolean queries, however.

To better understand the difficulty in the remaining open cases, we compare our study on trees and chains with the study on graphs by Fletcher et al. [31–34, 87]. The main difference is that all Boolean separations on graphs can be proven by *strong separations*.

6.1 Organization

In Section 6.2, we take a look at the limited applicability of strong separations in our study. In Section 6.3, we discuss two approaches that can aid in solving the remaining open problems. Finally, in Section 6.4, we discuss other directions for future research.

6.2 Strong separations on trees and chains

When compared to the study of the relative expressive power of the relation algebra on graphs [31–34, 87], we see major differences. First, we observe that most separation results for the graph data model are *strong Boolean separations*: to prove that $\mathcal{N}(\mathcal{F}_1) \not\leq_{\text{bool,strong}} \mathcal{N}(\mathcal{F}_2)$, one provides two finite graphs $\mathcal{G}_1$ and $\mathcal{G}_2$, and then show that only the language $\mathcal{N}(\mathcal{F}_1)$ has an expression e with $[\![e]\!]_{\mathcal{G}_1} = \emptyset$ and $[\![e]\!]_{\mathcal{G}_2} \neq \emptyset$. Likewise, we can also consider *strong path separations*: to prove that $\mathcal{N}(\mathcal{F}_1) \not\leq_{\text{path,strong}} \mathcal{N}(\mathcal{F}_2)$, one provides a finite graph $\mathcal{G}$ and query q in $\mathcal{N}(\mathcal{F}_1)$ and then show that no expression in $\mathcal{N}(\mathcal{F}_2)$ evaluates to $[\![q]\!]_{\mathcal{G}}$.

Example 6.1 (Fletcher et al. [32, Proposition 11]). Consider the expression $e = (\mathcal{E} \circ \mathcal{E}) - (\mathcal{E} \cup \text{id})$, and graphs $\mathcal{G}_3$ and $\mathcal{G}_{\bowtie}$ of Figure 6.1. We have $[\![e]\!]_{\mathcal{G}_3} = \emptyset$ and $[\![e]\!]_{\mathcal{G}_{\bowtie}} \neq \emptyset$. Let $\mathcal{F} \subseteq \{\text{di}, ^\frown, \pi, \overline{\pi}, \cap, ^*\}$. By an exhaustive search, one can show that there does not exist an expression e' in $\mathcal{N}(\mathcal{F})$ with $[\![e']\!]_{\mathcal{G}_3} = \emptyset$ and $[\![e']\!]_{\mathcal{G}_{\bowtie}} \neq \emptyset$. Hence, $\mathcal{N}(-) \not\leq_{\text{bool,strong}} \mathcal{N}(\mathcal{F})$.

From the definition of strong separations, it immediately follows that they can be proven using brute-force methods. Furthermore, using Lemma 3.3, strong separations carry over from fragments without the Kleene-star to fragments with the Kleene-star. Unfortunately, in the setting of querying trees or chains, we can prove that in several cases no strong separations exist, even when separations exist. First, we shall prove this for chains. To do so, we use the following properties about querying chains:

Lemma 6.3. *Let C, C_1, and C_2 be labeled chains.*

(i) For every expression e in $\mathcal{N}(\mathcal{F})$, $\mathcal{F} \subseteq \{\text{di}, ^\frown, \pi, \cap, ^*\}$, with $[\![e]\!]_{C_1} \neq \emptyset$ and $[\![e]\!]_{C_2} = \emptyset$, there exists an expression e' in $\mathcal{N}()$ such that $[\![e']\!]_{C_1} \neq \emptyset$ and $[\![e']\!]_{C_2} = \emptyset$.

(ii) There exists an expression e_C in $\mathcal{N}(\overline{\pi})$ such that, for every chain C', we have $[\![e_C]\!]_{C'} \neq \emptyset$ if and only if C' is isomorphic to C.

(iii) For every expression e in $\mathcal{N}(\mathcal{F})$, $\mathcal{F} \subseteq \{\pi, \overline{\pi}, \cap, -, ^*\}$, there exists an expression e' in $\mathcal{N}(\pi)$ such that $[\![e]\!]_C = [\![e']\!]_C$.

(iv) For every expression e in $\mathcal{N}$, there exists an expression e_1 in $\mathcal{N}(\frown)$ and an expression e_2 in $\mathcal{N}(\mathrm{di}, \pi)$ such that $[\![e]\!]_C = [\![e_1]\!]_C = [\![e_2]\!]_C$.

Proof. First, we prove Statement (i). We represent C_1 by the path $n_1\, \ell_1\, n_2 \ldots n_k\, \ell_k\, n_{k+1}$ with n_1 the root of C_1 and n_{k+1} the leaf. Now consider the expression $e' = \ell_1 \circ \ldots \circ \ell_k$. We have $[\![e']\!]_{C_1} \neq \emptyset$. As $[\![e]\!]_{C_1} \neq \emptyset$ and $[\![e]\!]_{C_2} = \emptyset$, Lemma 3.21 guarantees that there does not exists a homomorphism from C_1 to C_2. Hence, we conclude $[\![e']\!]_{C_2} = \emptyset$. For Statement (ii), we use the expression $e_C = \overline{\pi}_2[\mathcal{E}] \circ e' \circ \overline{\pi}_1[\mathcal{E}]$. Next, we prove Statement (iii). Let r be the root of chain C, l be the leaf of chain C, let $(m, n) \in [\![e]\!]_C$, let $d_r = \|r \to m\|_C$ and $d_l = \|n \to l\|_C$. To prove Statement (iii), we observe that e is downward, which implies that $\mathrm{depth}(C) > d_r + d_l$. We construct $t_{(m,n)} = \pi_2[\mathcal{E}^{d_r}] \circ \mathcal{E}^{\mathrm{depth}(C)-d_r-d_l} \circ \pi_1[\mathcal{E}^{d_l}]$ and we have $[\![t_{(m,n)}]\!]_C = \{(m, n)\}$. We construct $e' = \bigcup_{(m,n)\in[\![e]\!]_C} t_{(m,n)}$, which is in $\mathcal{N}(\pi)$, and we conclude $[\![e]\!]_C = [\![e']\!]_C$. To prove Statement (iv), we construct e_1 by replacing every term $t_{(m,n)}$ in the above by $\mathcal{E}^{-d_r} \circ \mathcal{E}^{\mathrm{depth}(C)} \circ \mathcal{E}^{-d_l}$ and we construct e_2 by replacing every term $t_{(m,n)}$ in the above by $\pi_2[\mathcal{E}^{d_r}] \circ \pi_1[\mathcal{E}^{\mathrm{depth}(C)-d_r}] \circ \mathrm{all} \circ \pi_2[\mathcal{E}^{\mathrm{depth}(C)-d_l}] \circ \pi_1[\mathcal{E}^{d_l}]$. □

From Lemma 6.3, the strong Boolean separation of Theorem 3.24, and the strong path separations of Corollary 3.11 and Proposition 3.32 (iii), the characterization of strong separations for chains follows:

Theorem 6.4. *Let $\mathcal{F}_1, \mathcal{F}_2 \subseteq \{\mathrm{di}, \frown, \pi, \overline{\pi}, \cap, -\}$. Already on labeled and unlabeled chains, we have*

(i) $\mathcal{N}(\mathcal{F}_1) \not\leq_{\mathrm{bool,strong}} \mathcal{N}(\mathcal{F}_2)$ if and only if $\overline{\pi} \in \underline{\mathcal{F}_1}$ and $\overline{\pi} \notin \underline{\mathcal{F}_2}$.

(ii) $\mathcal{N}(\mathcal{F}_1) \not\leq_{\mathrm{path,strong}} \mathcal{N}(\mathcal{F}_2)$ if and only if

(ii).1. $\pi \in \mathcal{F}_1$ and $\mathcal{F}_2 \subseteq \{\cap, -\}$ or $\mathcal{F}_2 \subseteq \{\mathrm{di}\}$;

(ii).2. $\mathrm{di} \in \mathcal{F}_1$ and $\mathcal{F}_2 \subseteq \{\pi, \overline{\pi}, \cap, -\}$;

(ii).3. $\frown \in \mathcal{F}_1$ and $\frown \notin \mathcal{F}_2, \{\mathrm{di}, \pi\} \subsetneq \underline{\mathcal{F}_2}$; or

(ii).4. $\mathrm{di}, \pi \in \mathcal{F}_1$ and $\frown \notin \mathcal{F}_2, \{\mathrm{di}, \pi\} \subsetneq \underline{\mathcal{F}_2}$.

From Theorem 6.4, it immediately follows that none of the open cases on chains can be answered using strong separations.

With respect to the tree data model, we have several additional strong Boolean separations beyond the strong Boolean separations for chains (Theorem 6.4 (i)). The Boolean separations of Propositions 3.5, 3.9, and 3.33 and of Theorem 3.24 are all strong Boolean separations; only the separation of Proposition 3.15 is not a strong Boolean separation. Unfortunately, also on the tree data model, we can prove that several remaining open cases cannot be answered with strong separations. To show this, we use the following properties on querying trees:

Lemma 6.5. *Let $\mathcal{F} \subseteq \{\mathrm{di}, \frown, \pi, \overline{\pi}, \cap, -, {}^*\}$ and let $\mathcal{T}$ be a tree.*

(i) If $\mathcal{T}$ is not 2-subtree-reducible, then there exists an expression e in $\mathcal{N}(\mathrm{di}, \overline{\pi}, \cap)$ such that, for every $\mathcal{T}'$ that is not 2-subtree-reducible, we have $[\![e]\!]_{\mathcal{T}'} \neq \emptyset$ if and only if $\mathcal{T}'$ is isomorphic to $\mathcal{T}$.

(ii) If $\mathcal{T}$ is not 3-subtree-reducible, then there exists an expression e in $\mathcal{N}(\mathrm{di}, \frown, \overline{\pi}, \cap)$ such that, for every $\mathcal{T}'$ that is not 3-subtree-reducible, we have $[\![e]\!]_{\mathcal{T}'} \neq \emptyset$ if and only if $\mathcal{T}'$ is isomorphic to $\mathcal{T}$.

(iii) If $\mathcal{T}$ is not 3-subtree-reducible, then there exists an expression e in $\mathcal{N}(^\frown, -)$ such that, for every $\mathcal{T}'$ that is not 3-subtree-reducible, we have $[\![e]\!]_{\mathcal{T}'} \neq \emptyset$ if and only if $\mathcal{T}'$ is isomorphic to $\mathcal{T}'$.

Proof. We construct expressions $e = \overline{\pi}_2[\mathcal{E}] \circ s(r)$ with r the root node of $\mathcal{T}$ and $s(n)$ a node expression such that $(m, m) \in [\![s(n)]\!]_{\mathcal{T}'}$, with $\mathcal{T}'$ a tree that is not k-subtree-reducible, $k \in \{2, 3\}$, if and only if the subtree rooted at n and m are isomorphic.

The base cases for the construction of $s(n)$ are leaf nodes. Observe that $\overline{\pi} \in \underline{\{^\frown, -\}}$, hence, for all three Statements of the Lemma, expressions $\overline{\pi}_1[\mathcal{E}]$ suffice. Next, we consider non-leaf nodes n with children $m_1, \ldots, m_j$ reachable via labels $\ell_1, \ldots, \ell_j$. As no k-subtree-reduction step is possible on $\mathcal{T}$, this implies that one can find at most two (Statement (i)) or three children (Statement (ii) and Statement (iii)) reachable via label ℓ such that the subtrees rooted at these children are isomorphic. To distinguish between one, two, or three such children, we will define expressions $s(m)^i = \pi_1[\ell \circ s(m) \circ c(m)^i]$. For Statement (i), we define $c(m)^i$, $i \in \{1, 2\}$, by:

$$c(m)^1 = \overline{\pi}_1[c(m)^2];$$
$$c(m)^2 = \pi_2[(\ell \circ s(m) \circ \mathrm{di}) \cap (\ell \circ s(m))].$$

For Statement (ii), we define $c(m)^i$, $i \in \{1, 2, 3\}$, by:

$$c(m)^1 = \overline{\pi}_1[\mathit{Sibling}(m)];$$
$$c(m)^2 = \overline{\pi}_1[c(m)^{=1} \cup c(m)^{=3}];$$
$$c(m)^3 = \pi_1[(\mathit{Sibling}(m) \circ \mathit{Sibling}(m)) \cap \mathrm{di}],$$

with $\mathit{Sibling}(m) = (s(m) \circ \ell^\frown \circ \ell \circ s(m)) \cap \mathrm{di}$. Finally, for Statement (iii), we only need to apply the rewrite $e \cap \mathrm{di} \equiv_{\mathrm{path}} e - \mathrm{id}$ to eliminate usage of di in the expressions constructed for Statement (ii). □

From Lemma 6.5, Proposition 3.1 (i), and Proposition 3.7 (i) we derive a partial characterization of strong separations for trees:

Proposition 6.6. *Let $\mathcal{F} \subseteq \{\mathrm{di}, ^\frown, \pi, \overline{\pi}, \cap, -\}$.*

(i) If $\mathrm{di} \notin \mathcal{F}$ and $\{^\frown, -\} \subseteq \mathcal{F}$, then no strong Boolean separation and no strong path separation exists between $\mathcal{N}(\mathcal{F})$ and $\mathcal{N}(\mathcal{F} \cup \{\mathrm{di}\})$ on labeled or unlabeled trees.

(ii) If $\{\mathrm{di}, ^\frown, \overline{\pi}, \cap\} \subseteq \underline{\mathcal{F}}$, then no strong Boolean separation exists between $\mathcal{N}(\mathcal{F})$ and $\mathcal{N}$ on labeled or unlabeled trees.

(iii) If $\{^\frown, -\} \subseteq \underline{\mathcal{F}}$, then no strong Boolean separation exists between $\mathcal{N}(\mathcal{F})$ and $\mathcal{N}$ on labeled or unlabeled trees.

Proof. First, we prove Statement (i). By Proposition 3.1 (ii), we have $\mathrm{di} \equiv_{\mathrm{path}} ([\mathcal{E}^\frown]^* \circ [\mathcal{E}]^*) - \mathrm{id}$. As strong Boolean separations are obtained on finite graphs, we can use Lemma 3.3 to eliminate this usage of Kleene-star. Consequently, any expression in $\mathcal{N}(\mathcal{F} \cup \{\mathrm{di}\})$ can be rewritten into an instance-equivalent expression in $\mathcal{N}(\mathcal{F})$. Statement (ii) and (iii) are obtained by combining Proposition 3.7 (i), Lemma 6.5 (ii), and Lemma 6.5 (iii). □

Proposition 6.6 only provides a partial characterization of the cases in which strong Boolean separations or strong path separations exists on labeled and unlabeled trees. We recognize finding a full characterization as an interesting avenue for future work:

Problem 6.7. *Let* $\mathcal{F}_1, \mathcal{F}_2 \subseteq \{\mathrm{di}, ^\frown, \pi, \overline{\pi}, \cap, -\}$ *and let* $z \in \{\mathrm{path}, \mathrm{bool}\}$. *With respect to labeled and unlabeled trees, in which cases do we have* $\mathcal{N}(\mathcal{F}_1) \preceq_{z,\mathrm{strong}} \mathcal{N}(\mathcal{F}_2)$*?*

We observe that the study of strong path separations is closely related to the study of the expressive power of the relation algebra at the instance level. For the tree data model, this already received some attention in the literature, e.g. [35, 43], although these studies do not include non-local operators such as diversity.

6.3 On solving the remaining open problems

Theorem 6.4 and Proposition 6.6 unfortunately rule out strong separations for most of the open cases stated in Problem 6.1 and Problem 6.2. With respect to solving Problem 6.1, we recognize two other promising avenues.

First, we combine Problem 6.1 with Proposition 3.1 (iii), yielding:

Lemma 6.8. *We have* $\mathcal{N}(\mathrm{di}, \pi, \overline{\pi}, \cap, -, ^*) \preceq_{\mathrm{path}} \mathcal{N}(\mathrm{di}, \pi, \overline{\pi}, \cap, ^*)$ *on labeled and unlabeled chains only if we also have* $\mathcal{N}(^\frown) \preceq_{\mathrm{path}} \mathcal{N}(\mathrm{di}, \pi, \overline{\pi}, \cap, ^*)$.

Hence, by proving $\mathcal{N}(^\frown) \not\preceq_{\mathrm{path}} \mathcal{N}(\mathrm{di}, \pi, \overline{\pi}, \cap, ^*)$ on unlabeled chains, which is still an open case, we also solve four of the open cases of Problem 6.1.

Another approach towards solving Problem 6.1 involves the binary *complement* of expressions. Let $\mathcal{G} = (\mathcal{V}, \Sigma, \mathbf{E})$ be a graph and let e be an expression. We define the semantics of the *complement operator* $\overline{\ }$ by

$$[\![\overline{e}]\!]_{\mathcal{G}} = \{(m, n) \mid (m, n \in \mathcal{V}) \wedge ((m, n) \notin [\![e]\!]_{\mathcal{G}})\}.$$

If we consider relation algebra fragments extended with the complement operator, then we observe the following:

Lemma 6.9. *Let* $\{\mathrm{di}, \cap\} \subseteq \mathcal{F} \subseteq \{\mathrm{di}, ^\frown, \pi, \overline{\pi}, \cap, ^*\}$. *We have* $\mathcal{N}(\mathcal{F} \cup \{-\}) \equiv_{\mathrm{path}} \mathcal{N}(\mathcal{F} \cup \{\overline{\ }\})$.

Proof. We have $\overline{e} \equiv_{\mathrm{path}} \mathrm{all} - e$ and we have $e_1 - e_2 \equiv_{\mathrm{path}} e_1 \cap \overline{e_2}$. □

As a consequence of Lemma 6.9, we have

Proposition 6.10. *Let* $\{\mathrm{di}, \overline{\pi}, \cap\} \subseteq \mathcal{F} \subseteq \{\mathrm{di}, ^\frown, \pi, \overline{\pi}, \cap, ^*\}$ *and let* $z \in \{\mathrm{bool}, \mathrm{path}\}$. *With respect to either labeled trees, unlabeled trees, labeled chains, or unlabeled chains, we have* $\mathcal{N}(\mathcal{F} \cup \{-\}) \not\preceq_z \mathcal{N}(\mathcal{F})$ *if and only if* $\mathcal{N}(\mathcal{F} \cup \{\overline{\ }\}) \not\preceq_z \mathcal{N}(\mathcal{F})$.

At first sight, Proposition 6.10 seems a bit useless, as it equates two problems without hinting at a solution for any of these problems. Fortunately, with respect to the complement, we do already have several useful results. First, we observe the following equivalences:

$$\overline{\emptyset} \equiv_{\mathrm{path}} \mathrm{all};$$
$$\overline{\mathrm{id}} \equiv_{\mathrm{path}} \mathrm{di};$$
$$\overline{\mathrm{di}} \equiv_{\mathrm{path}} \mathrm{id};$$

$$\begin{aligned}
\overline{\mathcal{E}} &\equiv_{\text{path}} \text{di} \circ \mathcal{E} \cup (\text{all} \circ \overline{\pi}_2[\mathcal{E}]);\\
\overline{\mathcal{E}^\frown} &\equiv_{\text{path}} \mathcal{E}^\frown \circ \text{di} \cup (\overline{\pi}_1[\mathcal{E}^\frown] \circ \text{all});\\
\overline{\pi_j[e]} &\equiv_{\text{path}} \overline{\pi}_j[e] \cup \text{di};\\
\overline{\overline{\pi}_j[e]} &\equiv_{\text{path}} \pi_j[e] \cup \text{di};\\
\overline{e_1 \cup e_2} &\equiv_{\text{path}} \overline{e_1} \cap \overline{e_2};\\
\overline{e_1 \cap e_2} &\equiv_{\text{path}} \overline{e_1} \cup \overline{e_2}.
\end{aligned}$$

with e, e_1, and e_2 expressions, and $j \in \{1, 2\}$. In the above, only the complement of composition is missing. For many cases, we have been able to manually express complements of such compositions, e.g., for $k > 0$:

$$\overline{\mathcal{E}^{-k}} \equiv_{\text{path}} \mathcal{E}^{-k} \circ \text{di} \cup \overline{\pi}_1[\mathcal{E}^{-k}] \circ \text{all},$$

Unfortunately, we have not been able to find a way to express general complements. At the same time, we have also been unsuccessful in proving that the complement of compositions on either trees or chains is not expressible.

6.4 Other directions for future work

The obvious next step in this line of research is to complete our systematic study of the relative expressive power of relation algebra fragments on trees and chains, this by solving Problems 6.1, 6.2, and 6.7. Beyond these obvious next steps, we recognize several avenues of future work.

It remains open if the collapse results we obtained can be of use for query optimization. On the one hand, the condition automata of Chapter 4 and the condition tree queries of Chapter 5 are constructive techniques that can be used to eliminate particular operators, but, on the other hand, their application will usually result in a significant blow up in the query size. Hence, it is yet unclear whether elimination of certain operators can always offset the increase in query size, and in which cases such optimization strategy would be beneficial. Part III briefly touches upon this topic: Corollary 8.11 shows that elimination of intersection and difference enables the use of query optimization techniques not otherwise available.

We are also interested in extending our study to richer graph query languages. One of the main limitations of the relation algebra is its inability to count beyond 3. Proposition 3.7 already proved this inability and even simple queries such as "does there exist nodes with exactly k outgoing edges" can only be expressed in $\mathcal{N}$ if $k = 1$ or $k = 2$. Unfortunately, there are plenty real-world graph and tree queries that involve counting. As an example, consider a social network graph $\mathcal{G}$ and the query

$$\mathit{PopularPerson} = \{m \mid |\{n \mid (n, m) \in [\![\mathit{FriendOf}]\!]_{\mathcal{G}}\}| \geq 1\,000\},$$

which returns persons with at least a thousand friends. A straightforward way to support such queries, without changing the language too much, is by introducing counting operators. As an example, we could add the operators $\pi_1^{\geq k}$ and $\pi_2^{\geq k}$ with

$$\begin{aligned}
[\![\pi_1^{\geq k}[e]]\!]_{\mathcal{G}} &= \{(m, m) \mid |\{n \mid (m, n) \in [\![e]\!]_{\mathcal{G}}\}| \geq k\};\\
[\![\pi_2^{\geq k}[e]]\!]_{\mathcal{G}} &= \{(m, m) \mid |\{n \mid (n, m) \in [\![e]\!]_{\mathcal{G}}\}| \geq k\}.
\end{aligned}$$

From a first-order logic point of view, these additions are related to adding counting quantifiers of the form $\exists^{\geq k} x\ e(x)$. Such counting logics have been studied extensively in the literature (e.g., [37, 39, 79, 80]).

We can consider the counting operators as a form of aggregation, and, therefore, it is only natural to also consider using a multiset semantics instead of the set semantics usually used for the relation algebra and its fragments. As an example, we take a look at the graph query

$$\mathit{FriendOf} \circ \mathit{FriendOf},$$

which returns pairs (n, m) such that m is a friend-of-friend of n. These friend-of-friends retrieved by this query are at the basis of computing friend suggestions in many social networks. To limit the number of suggested friends for each person n, we probably want to select only those friends-of-friends m that have at least k friends in common with n. This information is readily available when using multiset semantics, as the pair (n, m) will be in the output exactly k' times if and only if n and m have exactly k' friends in common. Using set semantics, this information is missing entirely. We believe that the study of counting operators and, based on that, aggregation and multiset semantics is a logical next step in graph querying, for which not much is known yet.

A last direction of future work we consider is related to the Kleene-star operator. While this operator does add expressive power beyond FO[3], it remains relatively limited in its capabilities. In logics, one often considers more general fixpoint iteration instead of the limited Kleene-star operator, e.g., the logic $\mathrm{L}^3_{\infty\omega}$ which is obtained by adding the more powerful fixpoint iteration to FO[3] [39, 79, 80]. Whereas we believe that the relation algebra is relatively intuitive and user-friendly to use, we do not believe that this remains the case when we add unrestricted fixpoint iteration. Alternatively, we may therefore consider replacing the Kleene-star by a form of recursion inspired by context-free grammars:

Example 6.2 (Hellings [46, 47]). Consider the following context-free production rule for a non-terminal R

$$R \longrightarrow \mathit{ParentOf}^{\frown}\ \mathit{ParentOf} \mid \mathit{ParentOf}^{\frown}\ R\ \mathit{ParentOf}.$$

Similar to how the 2RPQs use regular expressions, this context-free grammar will produce the set of strings $S = \{\mathit{ParentOf}^{-i}\mathit{ParentOf}^{i} \mid i \geq 1\}$, and every string in S would represent a valid labeling of a path between a node pair (m, n) in a tree (or graph in general). In this case, the query R will evaluate to node pairs (m, n) such that m and n share an i-th degree ancestor, $i \geq 1$, a query that is not expressible in the relation algebra.

We can easily combine such context-free style production rules with the monotone operators of the relation algebra. For the non-monotone operators, we can adopt the usual approach taken by Datalog with negation [1]. In this sense, the resulting language is a restricted binary-relation-only variant of Datalog aimed at intuitive but powerful graph querying and, due to its restricted nature, the resulting query language allows for a simple variable-free notation. We are interested to see what the exact expressive power of such language is, what the exact complexity of query evaluation is, and if the restricted nature, compared to Datalog, allows for more efficient (distributed) query evaluation on graphs and trees than usual Datalog queries.

Part III

On Tarski's Relation Algebra

THE SEMI-JOIN ALGEBRA

CHAPTER 7

Introduction[15]

The relation algebra is, as discussed in Chapter 1, a versatile graph query language in which it is relatively easy to express complex intentional relationships in terms of graph navigation. To do so, the relation algebra relies heavily on *composition*, a relatively expensive operator to evaluate. This is problematic when the full expressive power of composition is not required. To illustrate this, we reconsider the query

$$GgpAndFriends = \pi_1[ParentOf \circ ParentOf \circ ParentOf] \circ FriendOf,$$

from Chapter 1. This query yields pairs (m, n) such that m is a great-grandparent and a friend of n. A naive evaluation of this query involves using composition to navigate the graph via the *ParentOf* edge to check if m is a great-grandparent. Conceptually, this usage of composition does not neatly fit the intended purpose, however: the subquery $ParentOf \circ ParentOf \circ ParentOf$ yields pairs (m, z) such that m is the great-grandparent of great-grandchild z and, after computing these pairs, the projection-step will discard all computed information on the great-grandchildren. Consequently, evaluating the query by evaluating each of the operators involved is a relative wasteful process, and a more efficient approach would be to use a simple breath-first-search algorithm to find all great-grandparents without also computing all great-grandchildren.

For relatively simple queries such as *GgpAndFriends*, we can add operators to the relation algebra to enable the direct expression of more efficient query evaluation approaches at a high level. One way to do so, is by adding the semi-join operators $\ltimes$ and $\rtimes$. Rather than computing the composition of relations, semi-joins instead only determine the pairs that are involved in such compositions. In particular, if R and S are binary relations, then the left semi-join $R \ltimes S$ determines the pairs in R that can be composed with pairs in S, i.e., $\{(m, n) \in R \mid \exists z\ (n, z) \in S\}$ and the right semi-join $R \rtimes S$ determines the pairs in S that can be composed with pairs in R, i.e., $\{(m, n) \in S \mid \exists z\ (z, m) \in R\}$. By using these semi-joins instead of compositions, we can rewrite the query *GgpAndFriend* into the path-equivalent query

$$\pi_1[ParentOf \ltimes (ParentOf \ltimes ParentOf)] \rtimes FriendOf.$$

The main advantage of replacing compositions by semi-joins in the above query is easy to see: evaluation of the resulting query by evaluating each operation involved is possible in linear time with respect to the size of the edge relations, whereas evaluation of the

[15] The results in this chapter are based on the paper "From relation algebra to semi-join algebra: an approach for graph query optimization". [55].

original query takes quadratic time. As social networks and other graph datasets tend to be extremely large, the original composition-based approach is unacceptably expensive, whereas the semi-join-based approach might be acceptable. This also holds in general, as it is well-known that evaluating semi-joins is more efficient than evaluating compositions, even in the worst-case [69].

To achieve these improvements in practice, we can add the semi-join operators to appropriate query languages. This does, however, put the burden of efficient query evaluation on the users: in the above rewriting, we needed both the left and right semi-join operators. With respect to the former, we additionally had to insert parenthesis to control the order of evaluation of this non-associative operator. So, even in this simple example, the resulting expression becomes less intuitive and harder to write. Therefore, we believe that in modern graph database systems, which use declarative high-level graph query languages, such rewritings should be performed *for* the users, rather than *by* the users.

Here, we study ways to apply these semi-join optimizations automatically. More concretely, we study how fragments of the relation algebra relate to fragments of the semi-join algebra, the latter obtained by replacing composition by semi-joins and the Kleene-star by appropriate less-costly forms of fixpoint iteration.

To the best of our knowledge, we are the first to study the relationships between the expressive power of the relation algebra and the semi-join algebra comprehensively. We should point out that the study of semi-joins has already received attention in the setting of Codd's relational algebra [23, 64, 66–68]. In this setting, the semi-join version of the relational algebra is studied as a query language that has limited expressive power, cheap query evaluation, and for which many decision problems are decidable.

In the design and implementation of relational database systems, *basic* semi-join rewrite rules are well-known and the automatic usage of semi-join steps plays an important role in the efficient evaluation of distributed joins [11] and in Yannakakis algorithm for evaluating acyclic joins [94, 98]. In both cases, these semi-join steps are used as *reducers* that provide a preprocessing step aimed at reducing the size of intermediate relations before joining them. A similar reducer-based role for the semi-join has also been studied in the context of the multiset relational algebra [84]. This focus on using the semi-join as a reducer sharply contrasts with our usage, as we aim at eliminating compositions altogether in favor of semi-joins.

In Chapter 8, we formalize the semi-join algebra and establish relationships between the expressive power of the relation algebra and the semi-join algebra. Next, in Chapter 9, we focus on how these relationships can be utilized practically, by investigating how these semi-join rewrites contribute to optimizing graph query evaluation of relation algebra expressions. In Chapter 10, we take a first look at Tarski's relation algebra from a practical-query point of view. We identify an obvious shortcoming in the relation algebra, and address this shortcoming by adding node selections. Then, we investigate the impact this addition has on graph query optimization. Finally, in Chapter 11, we conclude on our findings, discuss efficient graph query evaluation, and identify directions for future work.

CHAPTER 8

On the expressive power of the semi-join algebra[16]

In this chapter, we formalize the semi-join algebra and study the relationships between the expressive power of fragments of the relation algebra and fragments of the semi-join algebra. Our main results are as follows:

1. We will show that the semi-join algebra has the same expressive power as FO[2], first-order logic in which formulae are restricted to having two variables. It is well-known that the relation algebra has the same expressive power as FO[3] [36, 89], which already puts limitations on the expressive power of the semi-join algebra when compared to the relation algebra [37, 39, 70, 79, 80].

2. To further establish the relationships in the expressive power of the relation algebra and the semi-join algebra, we investigate how the relative expressive power of fragments of the relation algebra compares to fragments of the semi-join algebra. We do so by showing that expressions of the form $\pi_j[e]$, $j \in \{1, 2\}$, can be rewritten into path-equivalent expressions in the semi-join algebra whenever the expression is in $\mathcal{N}(\mathcal{F})$, $\mathcal{F} \subseteq \{\text{di}, \frown, \pi, \overline{\pi}\}$. We call this path equivalence of projection-expressions *projection equivalence*.

3. We extend the above results towards relation algebra augmented with the Kleene-star operator. We show that expressions in $\mathcal{N}(\mathcal{F})$, $\mathcal{F} \subseteq \{\text{di}, \frown, \pi, \overline{\pi}, ^*\}$, can be rewritten into projection-equivalent expressions in the semi-join algebra augmented with a simple form of fixpoint iteration.

4. The above mentioned rewrites only place restrictions on the usage of intersection and difference. To show that these restrictions are not too severe, we prove that not all expressions using both composition and intersection are expressible in FO[2]. We identify syntactical restrictions on the usage of intersection and difference in the relation algebra, and show that the resulting language fragments have exactly the same expressive power, with respect to projection equivalence, as the semi-join algebra. From these results, it follows that intersection and difference only provide limited expressive power in the semi-join algebra.

5. In the setting of the tree data model used in Part II and the setting of finite sibling-ordered trees [74], we can use known redundancies involving the intersection and

[16] The results in this chapter are based on the paper "From relation algebra to semi-join algebra: an approach for graph query optimization". [55].

difference operators to our advantage. We strengthen the well-known collapse of first-order logic on sibling ordered trees (FO^{tree}) to Conditional XPath (a fragment of the relation algebra), by proving a collapse at the level of Boolean queries of FO^{tree} to the semi-join algebra.

6. Finally, we take a look at how the newly introduced notion of projection equivalence compares to the standard notions of path equivalence and Boolean equivalence. As a consequence, we also strengthen the known Boolean equivalences between fragments of the relation algebra when querying graphs [32] to projection equivalences.

8.1 Organization

In Section 8.2, we formalize the semi-join algebra. In Section 8.3, we show that the semi-join algebra is path-equivalent to FO[2]. In Section 8.4, we present sound rewrite rules for relation algebra expressions that aim at substituting composition and Kleene-star operators by semi-join and fixpoint operators. In Section 8.5, we analyze these rewrite rules in more detail and formalize the exact relationships between fragments of the relation algebra and corresponding fragments of the semi-join algebra. In Section 8.6, we analyze cases in which intersection and difference can be properly rewritten. In Section 8.7, we apply the relationships between FO[3] and FO[2] established in the previous sections and combine them with well-known results on the expressive power of FO[3] on trees and chains. Finally, in Section 8.8, we compare the newly introduced notion of projection equivalence to the standard notions of path equivalence and Boolean equivalence.

8.2 The semi-join algebra

In this section, we formally introduce the semi-join algebra.

Definition 8.1. The *semi-join algebra* is defined by the grammar

$$e := \emptyset \mid \text{id} \mid \text{di} \mid \ell \mid \ell^{\frown} \mid \pi_j[e] \mid \overline{\pi}_j[e] \mid e \ltimes e \mid e \rtimes e \mid e \cup e \mid e \cap e \mid e - e,$$

in which $\ell \in \Sigma$ and $j \in \{1, 2\}$. Let $\mathcal{G} = (\mathcal{V}, \Sigma, \mathbf{E})$ be a graph and let e be an expression. The semantics of evaluation for $\emptyset$, id, di, ℓ, $\ell^{\frown}$, $\pi_j[e]$, $\overline{\pi}_j[e]$, $e_1 \cup e_2$, $e_1 \cap e_2$, and $e_1 - e_2$ is defined in the same way as for the relation algebra (Definition 1.2). The semantics of evaluation for $e_1 \ltimes e_2$ and $e_1 \rtimes e_2$ is defined as follows:

$$[\![e_1 \ltimes e_2]\!]_{\mathcal{G}} = \{(m, n) \mid (m, n) \in [\![e_1]\!]_{\mathcal{G}} \wedge \exists z\, (n, z) \in [\![e_2]\!]_{\mathcal{G}}\};$$
$$[\![e_1 \rtimes e_2]\!]_{\mathcal{G}} = \{(m, n) \mid (m, n) \in [\![e_2]\!]_{\mathcal{G}} \wedge \exists z\, (z, m) \in [\![e_1]\!]_{\mathcal{G}}\}.$$

We denote the semi-join algebra by $\mathcal{M}$.

Example 8.1. The expressions

$$e_1 = \pi_1[\mathit{ParentOf} \circ \overline{\pi}_1[\mathit{OwnsPet}] \circ \mathit{ResearcherAt}];$$
$$e_2 = \pi_1[\mathit{ParentOf} \ltimes (\overline{\pi}_1[\mathit{OwnsPet}] \ltimes \mathit{ResearcherAt})]$$

both return people that are parents of researchers that do not own any pets. The expression e_1 is in $\mathcal{N}$ and the expression e_2 is in $\mathcal{M}$. Both expressions are node expressions. We have $e_1 \equiv_{\text{path}} e_2$.

As an FO[2]-like counterpart of the Kleene-star, which itself is an iterated version of composition, we introduce a form of fixpoint iteration. We add the operator $\mathsf{fp}_{j,\mathfrak{R}}[e \text{ union } b]$ with $j \in \{1,2\}$, b an expression, e an expression, and $\mathfrak{R}$ the single free variable of e. We do not allow $\mathfrak{R}$ to occur elsewhere. The semantics of evaluating $\mathsf{fp}_{j,\mathfrak{R}}[e \text{ union } b]$ on graph $\mathcal{G}$ is defined next. Let $s_0 := [\![b]\!]_{\mathcal{G}}|_j$ and define $s_i := s_{i-1} \cup [\![e]\!]_{\mathcal{G}+s_{i-1}}|_j$ in which $\mathcal{G} + s_{i-1}$ is the graph $\mathcal{G}$ augmented with the edge relation $\{(n,n) \mid n \in s_{i-1}\}$ labeled with $\mathfrak{R}$. Due to monotonicity of $\cup$, there exists a k, $k \leq |\mathcal{V}|$, such that $s_k = s_{k+1}$. We define $[\![\mathsf{fp}_{j,\mathfrak{R}}[e \text{ union } b]]\!]_{\mathcal{G}} = \{(n,n) \mid n \in s_k\}$.

Example 8.2. The expression

$$e_1 = \pi_1[[\mathit{ParentOf} \circ \overline{\pi}_1[\mathit{OwnsPet}]]^* \circ \mathit{ResearcherAt}]$$

is in $\mathcal{N}$ and returns people that are ancestors of a chain of descendants that do not own pets and where the youngest descendant is also a researcher. Let $e = \mathit{ParentOf} \ltimes (\overline{\pi}_1[\mathit{OwnsPet}] \ltimes \mathfrak{R})$. This expression has a single free variable $\mathfrak{R}$. Consider the expression

$$e_2 = \mathsf{fp}_{1,\mathfrak{R}}[e \text{ union } \mathit{ResearcherAt}].$$

We have $e_1 \equiv_{\text{path}} e_2$. We observe that e_1 and e_2 do not have free variables.

We only introduce fixpoint iteration here as a less costly alternative to the Kleene-star. For this purpose, general fixpoints are too strong, however. Therefore, we put additional restrictions on the expression e used in $\mathsf{fp}_{j,\mathfrak{R}}[e \text{ union } b]$: if $j = 1$, then e must be *right-recursive* in $\mathfrak{R}$ and, if $j = 2$, then e must be *left-recursive* in $\mathfrak{R}$, which we inductively define next.

Let $x \in \{\text{left}, \text{right}\}$. If $\mathfrak{R}$ is a variable, then the expression $\mathfrak{R}$ is x-recursive in $\mathfrak{R}$. Expressions of the form $e = e_1 \cup e_2$ are x-recursive in $\mathfrak{R}$ if e_1 and e_2 are x-recursive in $\mathfrak{R}$. Expressions of the form $e = e_1 \ltimes e_2$ are right-recursive in $\mathfrak{R}$ if e_1 does not have free variables and e_2 is right-recursive in $\mathfrak{R}$. Expressions of the form $e = e_1 \rtimes e_2$ are left-recursive in $\mathfrak{R}$ if e_2 does not have free variables and e_1 is left-recursive in $\mathfrak{R}$. Expressions of the form $e = \mathsf{fp}_{1,\mathfrak{R}'}[e' \text{ union } b']$ are right-recursive in $\mathfrak{R}$ if b' is right-recursive in $\mathfrak{R}$. Finally, expressions of the form $e = \mathsf{fp}_{2,\mathfrak{R}'}[e' \text{ union } b']$ are left-recursive in $\mathfrak{R}$ if b' is left-recursive in $\mathfrak{R}$.[17]

Example 8.3. The expression $e = \mathit{ParentOf} \ltimes (\overline{\pi}_1[\mathit{OwnsPet}] \ltimes \mathfrak{R})$, as used in Example 8.2, is right-recursive. The expression $e' = \mathfrak{R} \rtimes \mathit{FamilyOf} \cup \mathfrak{R} \rtimes \mathit{FriendOf}$ is left-recursive. The expression

$$e'' = \mathsf{fp}_{2,\mathfrak{R}}[e' \text{ union } \mathit{OwnsPet}^\frown]$$

yields pet owners and people that are related to pet owners via friend and family relations.

We denote the semi-join algebra, augmented with this restricted form of fixpoint iteration, by $\mathcal{M}^{\mathsf{fp}}$.

[17] The concepts of left-recursive and right-recursive expressions are closely related to concepts in formal languages [72]. Indeed, all expressions we allow can be mapped to concepts in *context-free grammars*: node variables map to non-terminals, unions map to individual grammar rules for a non-terminal, and semi-joins map to the compositions within a single grammar rule. This is no coincidence: it is well known that a context-free grammar that is left-recursive or right-recursive can always be rewritten into a regular expression (using Kleene-star instead of recursion) [72].

8.3 The relationship between FO[2] and $\mathcal{M}^{\text{fp}}$

Let $\mathcal{G} = (\mathcal{V}, \Sigma, \mathbf{E})$ be a graph with $\Sigma = \{\ell_1, \ldots, \ell_{|\Sigma|}\}$. We consider FO[2] as the first-order logic over the structure $(\mathcal{V}; \ell_1, \ldots, \ell_{|\Sigma|})$ and where variables are restricted to v and w. We have the following:

Theorem 8.1. *$\mathcal{M}$ is path-equivalent to* FO[2].

Proof (sketch). Our proof is based on the proof of the equivalence of FO[2] and the multi-dimensional modal logic MLR_2 [76, Section 2.3.1]. For the translation from FO[2] queries of the form $\{(v, w) \mid \varphi(v, w)\}$, with φ a FO[2] formula with free variables v and w, to expressions in $\mathcal{M}$, we use the following rewriting κ:

$$\begin{aligned}
\kappa(v \triangleq w) &= \text{id}; & \kappa(w \triangleq v) &= \text{id};\\
\kappa(v \triangleq v) &= \text{all}; & \kappa(w \triangleq w) &= \text{all};\\
\kappa(\ell(v, w)) &= \ell; & \kappa(\ell(w, v)) &= \ell^{\frown};\\
\kappa(\ell(v, v)) &= (\ell \cap \text{id}) \rtimes \text{all}; & \kappa(\ell(w, w)) &= \text{all} \ltimes (\ell \cap \text{id});\\
\kappa(\neg\varphi) &= \text{all} - \kappa(\varphi); & \kappa(\varphi \vee \psi) &= \kappa(\varphi) \cup \kappa(\psi);\\
\kappa(\exists v\ \varphi) &= \text{all} \ltimes \kappa(\varphi[v, w/w, v]); & \kappa(\exists w\ \varphi) &= \kappa(\varphi[v, w/w, v]) \rtimes \text{all}.
\end{aligned}$$

In the above, $\triangleq$ denotes the equality operator as used in first-order logic, and the notation $\varphi[v, w/w, v]$, for FO[2] formula φ, denotes the formula based on φ in which we have simultaneously substituted v for w and w for v (i.e., swapped v and w).

For the translation from expressions in $\mathcal{M}$ to FO[2] queries of the form $\{(v, w) \mid \varphi(v, w)\}$, with φ a FO[2] formula with free variables v and w, we use the following rewriting λ:

$$\begin{aligned}
\lambda(\text{id}) &= v \triangleq w; & \lambda(\text{di}) &= \neg(v \triangleq w);\\
\lambda(\ell) &= \ell(v, w); & \lambda(\ell^{\frown}) &= \ell(w, v);\\
\lambda(\pi_1[e]) &= v \triangleq w \wedge \exists w\ \lambda(e); & \lambda(\pi_2[e]) &= v \triangleq w \wedge \exists v\ \lambda(e);\\
\lambda(\overline{\pi}_1[e]) &= v \triangleq w \wedge \neg\exists w\ \lambda(e); & \lambda(\overline{\pi}_2[e]) &= v \triangleq w \wedge \neg\exists v\ \lambda(e);\\
\lambda(e_1 \ltimes e_2) &= \lambda(e_1) \wedge \exists v\ (v \triangleq w \wedge \exists w\ \lambda(e_2)); & \lambda(e_1 \rtimes e_2) &= \lambda(e_2) \wedge \exists w\ (v \triangleq w \wedge \exists v\ \lambda(e_1));\\
\lambda(e_1 \cup e_2) &= \lambda(e_1) \vee \lambda(e_2); & \lambda(e_1 \cap e_2) &= \lambda(e_1) \wedge \lambda(e_2);\\
\lambda(e_1 - e_2) &= \lambda(e_1) \wedge \neg\lambda(e_2).
\end{aligned}$$

Correctness of these translations follows from a straightforward induction argument. □

The next example illustrates the rewriting κ as used in the above proof:

Example 8.4. Consider the FO[2] query $\{(v, w) \mid \varphi(v, w)\}$ with

$$\varphi = v \triangleq w \wedge \exists w\ \mathit{ParentOf}(v, w).$$

This query yields parents. Notice that $e_1 \wedge e_2 = \neg(\neg e_1 \vee \neg e_2)$. Hence, φ is equivalent to

$$\varphi' = \neg(\neg(v \triangleq w) \vee \neg(\exists w\ \mathit{ParentOf}(v, w))).$$

Next, we rewrite φ' to an expression in $\mathcal{M}$:

$$\begin{aligned}
\kappa(\varphi') &= \kappa(\neg(\neg(v \triangleq w) \vee \neg(\exists w\ \mathit{ParentOf}(v,w))))\\
&= \text{all} - (\kappa(\neg(v \triangleq w) \vee \neg(\exists w\ \mathit{ParentOf}(v,w))))\\
&= \text{all} - (\kappa(\neg(v \triangleq w)) \cup \kappa(\neg(\exists w\ \mathit{ParentOf}(v,w))))\\
&= \text{all} - ((\text{all} - \kappa(v \triangleq w)) \cup (\text{all} - \kappa(\exists w\ \mathit{ParentOf}(v,w))))\\
&= \text{all} - ((\text{all} - \text{id}) \cup (\text{all} - (\kappa(\mathit{ParentOf}(w,v)) \rtimes \text{all})))\\
&= \text{all} - ((\text{all} - \text{id}) \cup (\text{all} - (\mathit{ParentOf}^\frown \rtimes \text{all})))
\end{aligned}$$

Some straightforward simplifications yield:

$$\begin{aligned}
&= \text{all} - (\text{di} \cup (\text{all} - (\mathit{ParentOf}^\frown \rtimes \text{all})))\\
&= (\text{all} - \text{di}) \cap (\text{all} - (\text{all} - (\mathit{ParentOf}^\frown \rtimes \text{all})))\\
&= \text{id} \cap (\mathit{ParentOf}^\frown \rtimes \text{all})\\
&= \text{id} \cap (\pi_2[\mathit{ParentOf}^\frown] \rtimes \text{all})\\
&= \text{id} \cap (\pi_1[\mathit{ParentOf}] \rtimes \text{all})\\
&= \pi_1[\mathit{ParentOf}].
\end{aligned}$$

We can generalize Theorem 8.1 to also cover fixpoints:

Proposition 8.2. *$\mathcal{M}^{\text{fp}}$ is path-subsumed by* FO[2] *to which inflationary fixpoints have been added.*

Proof (sketch). Observe that the semantics of the fixpoint operator fp we use is based on the semantics of the usual inflationary fixpoints [39]. As such, the translation is relatively straightforward. Let $\text{fp}_{j,\mathfrak{R}}[e \text{ union } b]$ be a fixpoint expression in $\mathcal{M}^{\text{fp}}$. We translate the fixpoint as follows

$$\begin{aligned}
\lambda(\text{fp}_{1,\mathfrak{R}}[e \text{ union } b]) &= (v \triangleq w) \wedge [\mathbf{ifp}_{v,\mathfrak{R}}\ \exists w\ (\lambda(\pi_1[e]) \vee \lambda(\pi_1[b]))](v);\\
\lambda(\text{fp}_{2,\mathfrak{R}}[e \text{ union } b]) &= (v \triangleq w) \wedge [\mathbf{ifp}_{w,\mathfrak{R}}\ \exists v\ (\lambda(\pi_2[e]) \vee \lambda(\pi_2[b]))](w),
\end{aligned}$$

and translate node variables by $\lambda(\mathfrak{N}) = (v \triangleq w) \wedge \mathfrak{N}(v)$. □

From Proposition 8.2, we also conclude that $\mathcal{M}^{\text{fp}}$ is path-subsumed by $\text{L}^2_{\infty\omega}$, the two-variable fragment of infinitary logic [39, 79].

8.4 Rewriting $\mathcal{N}$ towards using semi-joins

In this section, we explore ways to automatically rewrite expressions with compositions and Kleene-stars into expression with semi-joins and fixpoints. We start by identifying two situations in which the presence of a node expression, as a subexpression of an expression e, allows for the elimination of composition from e in favor of semi-joins:

1. The expression e itself is a node expression due to the use of projection or coprojection at the outer level. An example is the expression $\pi_1[e_1 \circ e_2]$, where a straightforward rewriting yields $\pi_1[e_1 \circ e_2] \equiv_{\text{path}} \pi_1[e_1 \ltimes e_2]$.

2. The expression e is a composition $e_1 \circ e_2$, in which e_1 or e_2 is a node expression. An example is the expression $e_1 \circ \pi_1[e_2]$, where a straightforward rewriting yields $e_1 \circ \pi_1[e_2] \equiv_{\text{path}} e_1 \ltimes \pi_1[e_2]$.

Since the semantics of the Kleene-star is defined using composition, similar observations can be made with respect to the Kleene-star.

The first observation above relies on the freedom to rewrite expressions to expressions that agree on either the first or the second column. We formalize this by introducing the following additional equivalence notions:

Definition 8.2. Let q_1 and q_2 be queries. We say that q_1 and q_2 are *left-projection-equivalent*, denoted by $q_1 \equiv_{\pi_1} q_2$, if, for every graph $\mathcal{G}$, $[\![q_1]\!]_{\mathcal{G}}|_1 = [\![q_2]\!]_{\mathcal{G}}|_1$ and are *right-projection-equivalent*, denoted by $q_1 \equiv_{\pi_2} q_2$, if, for every graph $\mathcal{G}$, $[\![q_1]\!]_{\mathcal{G}}|_2 = [\![q_2]\!]_{\mathcal{G}}|_2$.

Let $z \in \{\pi_1, \pi_2\}$. We say that the class of queries $\mathcal{L}_1$ is *z-subsumed* by the class of queries $\mathcal{L}_2$, denoted by $\mathcal{L}_1 \preceq_z \mathcal{L}_2$, if every query in $\mathcal{L}_1$ is z-equivalent to a query in $\mathcal{L}_2$. We say that the class of queries $\mathcal{L}_1$ is *projection-subsumed* by the class of queries $\mathcal{L}_2$, denoted by $\mathcal{L}_1 \preceq_\pi \mathcal{L}_2$, if $\mathcal{L}_1 \preceq_{\pi_1} \mathcal{L}_2$ and $\mathcal{L}_1 \preceq_{\pi_2} \mathcal{L}_2$. Let $z' \in \{\pi_1, \pi_2, \pi\}$. We say that the classes of queries $\mathcal{L}_1$ and $\mathcal{L}_2$ are *z'-equivalent*, denoted by $\mathcal{L}_1 \equiv_{z'} \mathcal{L}_2$, if $\mathcal{L}_1 \preceq_{z'} \mathcal{L}_2$ and $\mathcal{L}_2 \preceq_{z'} \mathcal{L}_1$.

By definition, expressions that are path-equivalent must also be left-projection-equivalent and right-projection-equivalent. Expressions that are left-projection-equivalent or right-projection-equivalent must also be Boolean-equivalent. The reverse is generally not true. (We will look at this in more detail in Section 8.8). To illustrate these equivalence notions, we extend Example 1.4:

Example 8.5. We have $e_1 \cap e_2 \equiv_{\text{path}} e_1 - (e_1 - e_2)$. Consequently, also $e_1 \cap e_2 \equiv_{\pi_1} e_1 - (e_1 - e_2)$ and $e_1 \cap e_2 \equiv_{\pi_2} e_1 - (e_1 - e_2)$. We have $\pi_1[\ell] \equiv_{\text{bool}} \ell \equiv_{\text{bool}} \pi_2[\ell]$. We have $\pi_1[\ell] \equiv_{\pi_1} \ell$, but not $\ell \equiv_{\pi_1} \pi_2[\ell]$. Likewise, we have $\ell \equiv_{\pi_2} \pi_2[\ell]$, but not $\pi_1[\ell] \equiv_{\pi_2} \ell$. Finally, we have $\pi_1[\ell] \equiv_{\text{bool}} \pi_2[\ell]$, but not $\pi_1[\ell] \equiv_{\pi_1} \pi_2[\ell]$ or $\pi_1[\ell] \equiv_{\pi_2} \pi_2[\ell]$.

To support the rewriting of compositions and Kleene-stars to semi-joins and fixpoints in situations similar to the ones discussed above, we will develop ways to rewrite (parts of) expressions in $\mathcal{N}$ to path-equivalent (parts of) expressions in $\mathcal{M}$. As a first step, we consider basic expressions built without using the composition, Kleene-star, semi-join, and fixpoint operators.

Definition 8.3. The *basic expressions* are defined by the grammar

$$e := \emptyset \mid \text{id} \mid \text{di} \mid \ell \mid \ell^\frown \mid \pi_j[e] \mid \overline{\pi}_j[e] \mid e \cup e \mid e \cap e \mid e - e,$$

in which $\ell \in \Sigma$ and $j \in \{1, 2\}$.

These basic expressions are equally expressible in $\mathcal{N}$, in $\mathcal{M}$, and in $\mathcal{M}^{\text{fp}}$. More precisely, every basic expression in $\mathcal{N}(\mathcal{F})$, $\mathcal{F} \subseteq \{\text{di}, ^\frown, \pi, \overline{\pi}, \cap, -\}$, is also in $\mathcal{M}(\mathcal{F})$. To deal with composition and Kleene-star operators, we propose the rewrite rules of Figure 8.1. We notice that these rewrite rules do not change basic expressions. We will argue later that $\tau(e) \equiv_{\text{path}} e$, $\tau_{\pi_1}(e) \equiv_{\pi_1} e$, and $\tau_{\pi_2}(e) \equiv_{\pi_2} e$.

Not every expression in $\mathcal{N}$ can be rewritten to an expressions in either $\mathcal{M}$ or $\mathcal{M}^{\text{fp}}$. Therefore, the rewrite rules of Figure 8.1 aim at partially rewriting subexpressions to the semi-join algebra instead of rewriting entire expressions. As such, the rewrite rules yield expressions that use both relation algebra and semi-join algebra operators. In Section 8.5, we shall investigate which fragments of $\mathcal{N}$ are fully rewritten into $\mathcal{M}$ or $\mathcal{M}^{\text{fp}}$, and for which fragments this is only partially possible.

$$\tau(b) = b \qquad \tau_{\pi_i}(b) = b$$
$$\tau(f_j[e]) = f_j[\tau_{\pi_j}(e)] \qquad \tau_{\pi_i}(f_j[e]) = f_j[\tau_{\pi_j}(e)]$$
$$\tau(e_1 \circ e_2) = \circ_{\text{path}}(e_1; e_2) \qquad \tau_{\pi_i}(e_1 \circ e_2) = \circ_{\pi_i}(e_1; e_2)$$
$$\tau(e_1 \cup e_2) = \tau(e_1) \cup \tau(e_2) \qquad \tau_{\pi_i}(e_1 \cup e_2) = \tau_{\pi_i}(e_1) \cup \tau_{\pi_i}(e_2)$$
$$\tau(e_1 \oplus e_2) = \tau(e_1) \oplus \tau(e_2) \qquad \tau_{\pi_i}(e_1 \oplus e_2) = \tau(e_1) \oplus \tau(e_2)$$
$$\tau([e]^*) = [\tau(e)]^* \qquad \tau_{\pi_i}([e]^*) = \text{id}$$

$$\tau_{\circ_1}(b; \varepsilon) = b \ltimes \varepsilon \qquad \tau_{\circ_2}(b; \varepsilon) = \varepsilon \rtimes b$$
$$\tau_{\circ_1}(f_j[e]; \varepsilon) = f_j[\tau_{\pi_j}(e)] \ltimes \varepsilon \qquad \tau_{\circ_2}(f_j[e]; \varepsilon) = \varepsilon \rtimes f_j[\tau_{\pi_j}(e)]$$
$$\tau_{\circ_1}(e_1 \circ e_2; \varepsilon) = \tau_{\circ_1}(e_1; \tau_{\circ_1}(e_2; \varepsilon)) \qquad \tau_{\circ_2}(e_1 \circ e_2; \varepsilon) = \tau_{\circ_2}(e_2; \tau_{\circ_2}(e_1; \varepsilon))$$
$$\tau_{\circ_1}(e_1 \cup e_2; \varepsilon) = \tau_{\circ_1}(e_1; \varepsilon) \cup \tau_{\circ_1}(e_2; \varepsilon) \qquad \tau_{\circ_2}(e_1 \cup e_2; \varepsilon) = \tau_{\circ_2}(e_1; \varepsilon) \cup \tau_{\circ_2}(e_2; \varepsilon)$$
$$\tau_{\circ_1}(e_1 \oplus e_2; \varepsilon) = (\tau(e_1) \oplus \tau(e_2)) \ltimes \varepsilon \qquad \tau_{\circ_2}(e_1 \oplus e_2; \varepsilon) = \varepsilon \rtimes (\tau(e_1) \oplus \tau(e_2))$$
$$\tau_{\circ_1}([e]^*; \varepsilon) = \text{fp}_{\mathfrak{R},1}[\tau_{\circ_1}(e; \mathfrak{R}) \text{ union } \varepsilon] \qquad \tau_{\circ_2}([e]^*; \varepsilon) = \text{fp}_{\mathfrak{R},2}[\tau_{\circ_2}(e; \mathfrak{R}) \text{ union } \varepsilon]$$

$$\circ_{\text{path}}(e_1; e_2) = \begin{cases} \tau(e_1) \circ \tau(e_2) & \text{if } e_1 \text{ and } e_2 \text{ are not node expressions;} \\ \tau(e_1) \ltimes \tau_{\pi_1}(e_2) & \text{if } e_2 \text{ is a node expression;} \\ \tau_{\pi_2}(e_1) \rtimes \tau(e_2) & \text{if } e_1 \text{ is a node expression.} \end{cases}$$

$$\circ_{\pi_i}(e_1; e_2) = \begin{cases} \tau_{\circ_1}(e_1; \tau_{\pi_1}(e_2)) & \text{if } i = 1; \\ \tau_{\circ_2}(e_2; \tau_{\pi_2}(e_1)) & \text{if } i = 2. \end{cases}$$

Figure 8.1: Rewrite rules aimed at rewriting compositions to semi-joins and Kleene-stars to fixpoints. In these rules, b is a basic expression, ε is an already rewritten expression, $f \in \{\pi, \overline{\pi}\}$, $i \in \{1, 2\}$, $j \in \{1, 2\}$, $\oplus \in \{\cap, -\}$, and $\mathfrak{R}$ is a fresh variable.

Example 8.6. Consider the expression

$$e = \pi_1[((\mathit{WorksOn} \circ \mathit{WorksOn}^\frown) \cap \mathit{FriendOf}) \circ \mathit{EditorOf}] \circ \mathit{StudentOf}\,.$$

This expression returns pairs of professors and their students, but only for professors that are friends with an editor with whom they collaborate on a project. For clarity, we abbreviate each edge label in e, resulting in $\pi_1[((W \circ W^\frown) \cap F) \circ E] \circ S$. We have the following:

$$\begin{aligned} \tau(e) &= \tau_{\pi_2}(\pi_1[((W \circ W^\frown) \cap F) \circ E]) \rtimes \tau(S) \\ &= \pi_1[\tau_{\pi_1}(((W \circ W^\frown) \cap F) \circ E)] \rtimes S \\ &= \pi_1[\tau_{\circ_1}((W \circ W^\frown) \cap F; \tau_{\pi_1}(E))] \rtimes S \\ &= \pi_1[(\tau(W \circ W^\frown) \cap \tau(F)) \ltimes E] \rtimes S \\ &= \pi_1[((\tau(W) \circ \tau(W^\frown)) \cap F) \ltimes E] \rtimes S \\ &= \pi_1[((W \circ W^\frown) \cap F) \ltimes E] \rtimes S. \end{aligned}$$

We shall prove (Theorem 8.4) that e and $\tau(e)$ are path-equivalent. This rewriting results in an expression in which two out of three applications of composition are eliminated in favor of semi-joins. In Proposition 8.9 (Section 8.6), we shall show that the last remaining composition step is unavoidable.

When applied on expressions with subexpressions of the form $[e]^*$, the rewrite rules can introduce fixpoint operators. Consequently, certain rewrite rules yield subexpressions with free node variables. For these expressions with free node variables, we have not defined the semantics of query evaluation and, hence, we have also not defined when such expressions are left-projection-equivalent or right-projection-equivalent. To enable reasoning about expressions with free node variables, we generalize Definition 8.2:

Definition 8.4. Let e_1 and e_2 be expressions with a single free node variable $\mathfrak{N}$. We say that e_1 and e_2 are *left-projection-equivalent*, denoted by $e_1 \equiv_{\pi_1} e_2$, if, for every graph $\mathcal{G} = (\mathcal{V}, \Sigma, \mathbf{E})$ and every $s \subseteq \mathcal{V}$, $[\![e]\!]_{\mathcal{G}+s}|_1 = [\![e]\!]_{\mathcal{G}+s}|_1$ in which $\mathcal{G} + s$ is the graph $\mathcal{G}$ augmented with the edge relation $\{(n, n) \mid n \in s\}$ labeled with $\mathfrak{N}$. Likewise, we say that e_1 and e_2 are *right-projection-equivalent*, denoted by $e_1 \equiv_{\pi_2} e_2$, if, for every graph $\mathcal{G} = (\mathcal{V}, \Sigma, \mathbf{E})$ and every $s \subseteq \mathcal{V}$, $[\![e]\!]_{\mathcal{G}+s}|_2 = [\![e]\!]_{\mathcal{G}+s}|_2$.

To prove soundness of the rewrite rules of Figure 8.1, we use the following properties:

Lemma 8.3. *Let g, h, ϕ, and ψ be expressions. We have:*

(i) If $g \equiv_{\pi_j} h$ with $j \in \{1, 2\}$, then $\pi_j[g] \equiv_{\text{path}} \pi_j[h]$ and $\overline{\pi}_j[g] \equiv_{\text{path}} \overline{\pi}_j[h]$.

(ii) Let $g \equiv_{\text{path}} h$. If $\phi \equiv_{\pi_1} \psi$, then $g \ltimes \phi \equiv_{\text{path}} h \ltimes \psi$, $g \circ \phi \equiv_{\pi_1} h \circ \psi$, and $g \circ \phi \equiv_{\pi_1} h \ltimes \psi$. If $\phi \equiv_{\pi_2} \psi$, then $\phi \rtimes g \equiv_{\text{path}} \psi \rtimes h$, $\phi \circ g \equiv_{\pi_2} \psi \circ h$, and $\phi \circ g \equiv_{\pi_2} \psi \rtimes h$.

(iii) Let $g \equiv_{\text{path}} h$ and let ϕ be a node expression. If $\phi \equiv_{\pi_1} \psi$, then $g \circ \phi \equiv_{\text{path}} h \ltimes \psi$. If $\phi \equiv_{\pi_2} \psi$, then $\phi \circ g \equiv_{\text{path}} \psi \rtimes h$.

(iv) If $g \equiv_z h$ and $\phi \equiv_z \psi$ with $z \in \{\text{path}, \pi_1, \pi_2\}$, then $g \cup \phi \equiv_z h \cup \psi$.

(v) If $g \equiv_{\text{path}} h$ and $\phi \equiv_{\text{path}} \psi$, then $g \cap \phi \equiv_{\text{path}} h \cap \psi$ and $g - \phi \equiv_{\text{path}} h - \psi$.

(vi) $\text{id} \equiv_{\pi_1} [\phi]^*$ and $\text{id} \equiv_{\pi_2} [\phi]^*$.

(vii) If $g \ltimes \mathfrak{N} \equiv_{\pi_1} h$ and $\phi \equiv_{\pi_1} \psi$, then $[g]^* \circ \phi \equiv_{\pi_1} \text{fp}_{1,\mathfrak{N}}[h \text{ union } \psi]$. If $\mathfrak{N} \rtimes g \equiv_{\pi_2} h$ and $\phi \equiv_{\pi_2} \psi$, then $\phi \circ [g]^* \equiv_{\pi_2} \text{fp}_{2,\mathfrak{N}}[h \text{ union } \psi]$.

Proof. Statements (i)–(vi) follow from the semantics of the operators involved. We prove the first case of Statement (vii), the second case is analogous. Assume $g \ltimes \mathfrak{N} \equiv_{\pi_1} h$ and $\phi \equiv_{\pi_1} \psi$. We shall prove $[g]^* \circ \phi \equiv_{\pi_1} \text{fp}_{1,\mathfrak{N}}[h \text{ union } \psi]$ using induction on the stages of the evaluation of the fixpoint operator fp. By s_i we denote the i-th stage of the evaluation of the fixpoint and by e_i, we denote the expression $(g^0 \cup \ldots \cup g^i) \circ \phi$. By induction, we shall prove that $[\![e_i]\!]_{\mathcal{G}}|_1 = s_i$. The base case is $i = 0$. We have $e_0 = g^0 \circ \phi \equiv_{\text{path}} \text{id} \circ \phi \equiv_{\text{path}} \phi \equiv_{\pi_1} \psi$. Hence, we have $[\![e_0]\!]_{\mathcal{G}}|_1 = [\![\phi]\!]_{\mathcal{G}}|_1 = [\![\psi]\!]_{\mathcal{G}}|_1 = s_0$. Now assume that, for every j with $0 \leq j < i$, $[\![e_j]\!]_{\mathcal{G}}|_1 = s_j$. Consider the case for e_i. We have

$$\begin{aligned} e_i &\equiv_{\text{path}} (g^0 \cup \ldots \cup g^i) \circ \phi \\ &\equiv_{\text{path}} (g^0 \cup \ldots \cup g^{i-1}) \circ \phi \cup g \circ ((g^0 \cup \ldots \cup g^{i-1}) \circ \phi) \\ &\equiv_{\text{path}} e_{i-1} \cup (g \circ e_{i-1}) \end{aligned}$$

$$\equiv_{\pi_1} e_{i-1} \cup (g \ltimes e_{i-1}).$$

Due to the induction hypothesis, we have $[\![e_{i-1}]\!]_{\mathcal{G}}|_1 = s_{i-1}$. Hence, during computation of s_i, we have

$$\begin{aligned} s_i &= s_{i-1} \cup [\![h]\!]_{\mathcal{G}+s_{i-1}}|_1 \\ &= [\![e_{i-1}]\!]_{\mathcal{G}}|_1 \cup [\![g \ltimes \mathfrak{R}]\!]_{\mathcal{G}+s_{i-1}}|_1. \end{aligned}$$

By the semantics of $\mathcal{G} + s_{i-1}$ and $[\![e_{i-1}]\!]_{\mathcal{G}}|_1 = s_{i-1}$, we can replace $\mathfrak{R}$ by e_{i-1}, resulting in

$$\begin{aligned} &= [\![e_{i-1} \cup (g \ltimes e_{i-1})]\!]_{\mathcal{G}}|_1 \\ &= [\![e_i]\!]_{\mathcal{G}}|_1. \end{aligned}$$

We conclude $[g]^* \circ \phi \equiv_{\pi_1} \mathsf{fp}_{1,\mathfrak{R}}[h \text{ union } \psi]$. □

The rules of Figure 8.1 depend on the ability to determine if an expression e is a node expression. This is, in general, hard to determine without evaluation of e. We can, however, use the semantics of $\mathcal{N}$, $\mathcal{M}$, and $\mathcal{M}^{\mathsf{fp}}$ to define a predicate $\mathsf{ns}(e)$ that evaluates to true *only if* the expression e is a node expression:

$$\begin{aligned} \mathsf{ns}(\emptyset) = \mathsf{ns}(\mathrm{id}) &= \mathtt{True}; \\ \mathsf{ns}(\mathrm{di}) &= \mathtt{False}; \\ \mathsf{ns}(\ell) = \mathsf{ns}(\ell^\frown) &= \mathtt{False}; \\ \mathsf{ns}(f_j[e]) &= \mathtt{True}; \\ \mathsf{ns}(e_1 \circ e_2) &= \mathsf{ns}(e_1) \wedge \mathsf{ns}(e_2); \\ \mathsf{ns}(e_1 \ltimes e_2) = \mathsf{ns}(e_2 \rtimes e_1) &= \mathsf{ns}(e_1); \\ \mathsf{ns}(e_1 \cup e_2) &= \mathsf{ns}(e_1) \wedge \mathsf{ns}(e_2); \\ \mathsf{ns}(e_1 \cap e_2) &= \mathsf{ns}(e_1) \vee \mathsf{ns}(e_2); \\ \mathsf{ns}(e_1 - e_2) &= \mathsf{ns}(e_1); \\ \mathsf{ns}([e]^*) &= \mathsf{ns}(e); \\ \mathsf{ns}(\mathsf{fp}_{j,\mathfrak{R}}[e \text{ union } b]) &= \mathtt{True}, \end{aligned}$$

in which $f \in \{\pi, \overline{\pi}\}$, and $j \in \{1, 2\}$. Using Lemma 8.3 and induction on the length of relation algebra expressions, we can establish the following main result about the soundness of the rewrite rules in Figure 8.1:

Theorem 8.4. *Let e be an expression in $\mathcal{N}$. We have $\tau(e) \equiv_{\text{path}} e$, we have $\tau_{\pi_1}(e) \equiv_{\pi_1} e$, and we have $\tau_{\pi_2}(e) \equiv_{\pi_2} e$.*

The rewrite rules of Figure 8.1 are sound, as stated in Theorem 8.4, but not complete, as illustrated by the following example:

Example 8.7. Consider the expression

$$e = (\mathit{FriendOf} \cap (\mathit{FriendOf} \circ \mathit{FriendOf})) - \text{all}.$$

Due to the presence of intersection, The rewritings $\tau(e)$, $\tau_{\pi_1}(e)$, and $\tau_{\pi_2}(e)$ do not result in an expression in $\mathcal{M}$ or $\mathcal{M}^{\mathsf{fp}}$. Since e always evaluates to $\emptyset$, however, we have $e \equiv_{\text{path}} \emptyset$, and $\emptyset$ is, by definition, in $\mathcal{M}$ and $\mathcal{M}^{\mathsf{fp}}$.

$$
\begin{aligned}
\rho(b) &= b;\\
\rho(f_j[e]) &= f_j[\rho(e)];\\
\rho(e_1 \ltimes e_2) &= \rho(e_1) \circ \pi_1[\rho(e_2)];\\
\rho(e_1 \rtimes e_2) &= \pi_2[\rho(e_1)] \circ \rho(e_2);\\
\rho(e_1 \cup e_2) &= \rho(e_1) \cup \rho(e_2);\\
\rho(e_1 \oplus e_2) &= \rho(e_1) \oplus \rho(e_2);\\
\rho(\mathsf{fp}_{1,\mathfrak{R}}[e \text{ union } b]) &= \pi_1[[\rho_{\text{right-}\mathfrak{R}}(e)]^* \circ \rho(b)];\\
\rho(\mathsf{fp}_{2,\mathfrak{R}}[e \text{ union } b]) &= \pi_2[\rho(b) \circ [\rho_{\text{left-}\mathfrak{R}}(e)]^*];
\end{aligned}
$$

$$
\begin{aligned}
\rho_{z\text{-}\mathfrak{R}}(\mathfrak{R}) &= \mathrm{id};\\
\rho_{z\text{-}\mathfrak{R}}(e_1 \cup e_2) &= \rho_{z\text{-}\mathfrak{R}}(e_1) \cup \rho_{z\text{-}\mathfrak{R}}(e_2);\\
\rho_{\text{left-}\mathfrak{R}}(e_1 \rtimes e_2) &= \rho_{\text{left-}\mathfrak{R}}(e_1) \circ \rho(e_2);\\
\rho_{\text{right-}\mathfrak{R}}(e_1 \ltimes e_2) &= \rho(e_1) \circ \rho_{\text{right-}\mathfrak{R}}(e_2);\\
\rho_{\text{right-}\mathfrak{R}}(\mathsf{fp}_{1,\mathfrak{R}'}[e' \text{ union } b']) &= [\rho_{\text{right-}\mathfrak{R}'}(e')]^* \circ \rho_{\text{right-}\mathfrak{R}}(b');\\
\rho_{\text{left-}\mathfrak{R}}(\mathsf{fp}_{2,\mathfrak{R}'}[e' \text{ union } b']) &= \rho_{\text{left-}\mathfrak{R}}(b') \circ [\rho_{\text{left-}\mathfrak{R}'}(e')]^*.
\end{aligned}
$$

Figure 8.2: Rewrite rules aimed at rewriting semi-joins to compositions and fixpoints to Kleene-stars. In these rules, b is a basic expression, $f \in \{\pi, \overline{\pi}\}$, $j \in \{1, 2\}$, $\oplus \in \{\cap, -\}$, and $z \in \{\text{left}, \text{right}\}$.

8.5 Relative expressive power of $\mathcal{N}$ and $\mathcal{M}^{\mathsf{fp}}$

The rewrite rules of Figure 8.1 do not fully rewrite every expression in $\mathcal{N}$ to $\mathcal{M}$ or $\mathcal{M}^{\mathsf{fp}}$. To better understand the limitations of these rewrite rules, we take a look at how they rewrite fragments of $\mathcal{N}$. As with the relation algebra, we write $\mathcal{F} \subseteq \{\mathrm{di}, \smile, \pi, \overline{\pi}, \cap, -\}$ to denote a set of operators. By $\mathcal{M}(\mathcal{F})$ we denote the fragment of $\mathcal{M}$ that only allows $\emptyset$, $\ell \in \Sigma$, id, $\ltimes$, $\rtimes$, $\cup$, and all operators in $\mathcal{F}$ and by $\mathcal{M}^{\mathsf{fp}}(\mathcal{F})$ we denote the fragment of $\mathcal{M}^{\mathsf{fp}}$ that only allows fixpoint iteration and the operators allowed by $\mathcal{M}(\mathcal{F})$.

Observe that we have already provided rewrite rules that can rewrite fragments of the relation algebra to the semi-join algebra. Before we take an in-depth look at how these rules operate on fragments of $\mathcal{N}$, we first take a look at the reverse: expressing the semi-join algebra using the relation algebra. To do so, we propose the rewrite rules of Figure 8.2.

Using a straightforward induction on the length of expressions, in which the base cases are basic expressions and Lemma 8.3 is used to prove the inductive cases, we conclude:

Proposition 8.5. *Let $\{\pi\} \subseteq \mathcal{F} \subseteq \{\mathrm{di}, \smile, \pi, \overline{\pi}, \cap, -\}$ and let e be an expression.*

(i) If e is in $\mathcal{M}(\mathcal{F})$, then $e \equiv_{\text{path}} \rho(e)$ and $\rho(e)$ is in $\mathcal{N}(\mathcal{F})$;

(ii) If e is in $\mathcal{M}^{\mathsf{fp}}(\mathcal{F})$, then $e \equiv_{\text{path}} \rho(e)$ and $\rho(e)$ is in $\mathcal{N}(\mathcal{F} \cup \{^*\})$.

A careful analysis of the rewrite rules of Figure 8.1, Theorem 8.4, and Proposition 8.5 allows us to conclude:

Theorem 8.6. *Let $\{\pi\} \subseteq \mathcal{F} \subseteq \{\mathrm{di}, \frown, \pi, \overline{\pi}\}$. We have*

(i) $\mathcal{M}(\mathcal{F}) \preceq_{\mathrm{path}} \mathcal{N}(\mathcal{F})$ *and* $\mathcal{M}(\mathcal{F}) \equiv_{\pi} \mathcal{N}(\mathcal{F})$*;*

(ii) $\mathcal{M}^{\mathrm{fp}}(\mathcal{F}) \preceq_{\mathrm{path}} \mathcal{N}(\mathcal{F} \cup \{^*\})$ *and* $\mathcal{M}^{\mathrm{fp}}(\mathcal{F}) \equiv_{\pi} \mathcal{N}(\mathcal{F} \cup \{^*\})$.

Proof. The rewrite rules of Figure 8.1 satisfy two basic properties. First, no rewrite rule introduces operators not yet in the original expressions, except for semi-joins (introduced when rewriting compositions) and fixpoints (introduced when rewriting Kleene-stars). Besides composition and the Kleene-star operators, rewriting semi-joins and fixpoints only introduces projections, which are included in every fragment. Second, compositions are only kept in path-equivalent rewritings. During left-projection-equivalent rewriting using $\tau_{\pi_1}(\cdot)$ and right-projection-equivalent rewriting using $\tau_{\pi_2}(\cdot)$, path-equivalent rewritings are only enforced by the usage of intersection and difference outside of basic expressions. □

8.6 The role of intersection and difference

Observe that Theorem 8.6 excludes the use of intersection or difference. This is a very severe restriction, however, as intersection and difference are allowed in the semi-join algebra. Careful analysis of the rewrite rules of Figure 8.1 reveals that rewriting intersection and difference only causes issues in conjunction with compositions, but not when used in basic expressions. As a consequence, we can extend Theorem 8.6 slightly.

Definition 8.5. Let $\mathcal{F} \subseteq \{\mathrm{di}, \frown, \pi, \overline{\pi}, \cap, -\}$. We write $\mathcal{B}(\mathcal{F})$, $\mathcal{B}(\mathcal{F} \cup \{^*\})$, $\mathcal{A}(\mathcal{F})$, and $\mathcal{A}^{\mathrm{fp}}(\mathcal{F})$ to denote the *basic fragments* of $\mathcal{N}(\mathcal{F})$, $\mathcal{N}(\mathcal{F} \cup \{^*\})$, $\mathcal{M}(\mathcal{F})$, and $\mathcal{M}^{\mathrm{fp}}(\mathcal{F})$, respectively, in which intersection and difference occur in basic expressions only.

Theorem 8.7. *Let $\{\pi\} \subseteq \mathcal{F} \subseteq \{\mathrm{di}, \frown, \pi, \overline{\pi}, \cap, -\}$. We have*

(i) $\mathcal{A}(\mathcal{F}) \preceq_{\mathrm{path}} \mathcal{B}(\mathcal{F})$, $\mathcal{A}(\mathcal{F}) \equiv_{\pi} \mathcal{B}(\mathcal{F})$, *and* $\mathcal{A}(\underline{\mathcal{F}}) \equiv_{\mathrm{path}} \mathcal{M}(\mathcal{F})$*;*

(ii) $\mathcal{A}^{\mathrm{fp}}(\mathcal{F}) \preceq_{\mathrm{path}} \mathcal{B}(\mathcal{F} \cup \{^*\})$, $\mathcal{A}^{\mathrm{fp}}(\mathcal{F}) \equiv_{\pi} \mathcal{B}(\mathcal{F} \cup \{^*\})$, *and* $\mathcal{A}^{\mathrm{fp}}(\underline{\mathcal{F}}) \equiv_{\mathrm{path}} \mathcal{M}^{\mathrm{fp}}(\mathcal{F})$.

Proof. It suffices to observe that in the semi-join algebra we may assume without loss of generality that intersection and difference occur in basic expressions only, since we can push down intersection and difference through projections, coprojections, semi-joins, and unions. Let $f \in \{\pi, \overline{\pi}\}$ and let $j \in \{1, 2\}$. We push down intersection using the following properties:

$$
\begin{aligned}
e \cap f_j[e'] &\equiv_{\mathrm{path}} f_j[e'] \cap e \equiv_{\mathrm{path}} (e \cap \mathrm{id}) \ltimes f_j[e'];\\
e \cap (e_1 \ltimes e_2) &\equiv_{\mathrm{path}} (e_1 \ltimes e_2) \cap e \equiv_{\mathrm{path}} (e \cap e_1) \ltimes e_2;\\
e \cap (e_1 \rtimes e_2) &\equiv_{\mathrm{path}} (e_1 \rtimes e_2) \cap e \equiv_{\mathrm{path}} e_1 \rtimes (e \cap e_2);\\
e \cap (e_1 \cup e_2) &\equiv_{\mathrm{path}} (e_1 \cup e_2) \cap e \equiv_{\mathrm{path}} (e \cap e_1) \cup (e \cap e_2);\\
e \cap (e_1 - e_2) &\equiv_{\mathrm{path}} (e_1 - e_2) \cap e \equiv_{\mathrm{path}} (e \cap e_1) - e_2
\end{aligned}
$$

We push down difference using the following properties:

$$
\begin{aligned}
e - f_j[e'] &\equiv_{\mathrm{path}} (e \cap \mathrm{di}) \cup ((e \cap \mathrm{id}) \ltimes \overline{\pi}_j[f_j[e']]);\\
f_j[e'] - e &\equiv_{\mathrm{path}} f_j[e'] \ltimes \overline{\pi}_j[e \cap \mathrm{id}];\\
e - (e_1 \ltimes e_2) &\equiv_{\mathrm{path}} (e - e_1) \cup ((e \cap e_1) \ltimes \overline{\pi}_1[e_2]);
\end{aligned}
$$

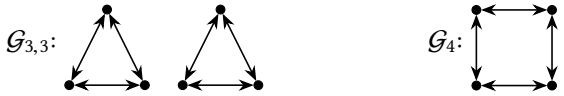


Figure 8.3: On the *left*, a two-3-cycle graph $\mathcal{G}_{3,3}$. On the *right*, a single-4-cycle graph $\mathcal{G}_4$.

$$(e_1 \ltimes e_2) - e \equiv_{\text{path}} (e_1 - e) \ltimes e_2;$$
$$e - (e_1 \rtimes e_2) \equiv_{\text{path}} (e - e_2) \cup (\overline{\pi}_2[e_1] \rtimes (e \cap e_2));$$
$$(e_1 \rtimes e_2) - e \equiv_{\text{path}} e_1 \rtimes (e_2 - e);$$
$$e - (e_1 \cup e_2) \equiv_{\text{path}} (e - e_1) - e_2;$$
$$(e_1 \cup e_2) - e \equiv_{\text{path}} (e_1 - e) \cup (e_2 - e).$$

We observe that fixpoints are node expressions and we can treat them as if they are projections. By repeatedly pushing down intersection and difference until this is no longer possible, all intersections and differences occur in basic expressions only. □

The above also implies a collapse of semi-join algebra fragments to the corresponding basic semi-join algebra fragments:

Corollary 8.8. *Let* $\{\pi\} \subseteq \mathcal{F} \subseteq \{\text{di}, \frown, \pi, \overline{\pi}, \cap, -\}$. *We have*

(i) $\mathcal{M}(\mathcal{F}) \preceq_{\text{path}} \mathcal{B}(\underline{\mathcal{F}})$ *and* $\mathcal{M}(\mathcal{F}) \equiv_{\pi} \mathcal{B}(\underline{\mathcal{F}})$*;*

(ii) $\mathcal{M}^{\text{fp}}(\mathcal{F}) \preceq_{\text{path}} \mathcal{B}(\underline{\mathcal{F}} \cup \{^*\})$ *and* $\mathcal{M}(\mathcal{F}) \equiv_{\pi} \mathcal{B}(\underline{\mathcal{F}} \cup \{^*\})$.

The result of Corollary 8.8 does not generalize to a collapse of relation algebra fragments to the corresponding basic relation algebra fragments. Indeed, we have already seen in Part II that intersection and difference play crucial roles in the expressive power of the non-downward and non-local fragments of the relation algebra (e.g., Proposition 3.9). Next, we will formally show that all basic fragments of $\mathcal{N}$ that include intersection have less expressive power than their non-basic counterparts:

Proposition 8.9. *Let* $\mathcal{F} \subseteq \{\text{di}, \frown, \pi, \overline{\pi}, \cap, -\}$ *with* $\cap \in \underline{\mathcal{F}}$. *We have*

(i) $\mathcal{N}(\mathcal{F}) \not\preceq_{\text{bool}} \mathcal{B}(\underline{\mathcal{F}})$ *and* $\mathcal{N}(\mathcal{F}) \not\preceq_{\text{bool}} \mathcal{M}(\underline{\mathcal{F}})$*;*

(ii) $\mathcal{N}(\mathcal{F} \cup \{^*\}) \not\preceq_{\text{bool}} \mathcal{B}(\underline{\mathcal{F}} \cup \{^*\})$ *and* $\mathcal{N}(\mathcal{F} \cup \{^*\}) \not\preceq_{\text{bool}} \mathcal{M}^{\text{fp}}(\underline{\mathcal{F}})$.

Proof. Consider the expression $e' = (\mathcal{E} \circ \mathcal{E}) \cap \mathcal{E}$. This expression is based on the part of e in Example 8.6 that could not be rewritten without using composition. The expression e' has an occurrence of intersection beyond the scope of basic expressions. We claim that no expression in $\mathcal{B}(\mathcal{F})$, for any $\mathcal{F} \subseteq \{\text{di}, \frown, \pi, \overline{\pi}, \cap, -, ^*\}$, is path-equivalent, left-projection-equivalent, right-projection-equivalent, or Boolean-equivalent to e'.

To show this, consider the graphs $\mathcal{G}_{3,3}$ and $\mathcal{G}_4$ of Figure 8.3 and observe that $[\![e']\!]_{\mathcal{G}_{3,3}} \neq \emptyset$ and $[\![e']\!]_{\mathcal{G}_4} = \emptyset$. To show that no expression in $\mathcal{B}(\mathcal{F})$ can distinguish between $\mathcal{G}_{3,3}$ and $\mathcal{G}_4$, we show that no expression in $\mathcal{M}$ or $\mathcal{M}^{\text{fp}}$ can distinguish between $\mathcal{G}_{3,3}$ and $\mathcal{G}_4$. We can do so using standard two-pebble game results for the FO[2]-variant of the infinitary finite variable logics [39, Example 3.10]. □

On graphs, the intersection and difference operators used only within the basic fragments of $\mathcal{N}$ still have useful roles, as shown in the next example.

Example 8.8. Consider again the relationships *FriendOf* and *WorksWith* and consider the following basic expressions:

$$e_1 = \mathit{FriendOf} \cap \mathit{WorksWith}, \qquad e_2 = \mathit{FriendOf} - \mathit{WorksWith},$$
$$e_3 = \mathit{WorksWith} - \mathrm{id}, \qquad e_4 = \mathit{WorksWith} \cap \mathrm{di},$$

Expression e_1 yields work-friends, whereas expression e_2 yields non-work-friends. These examples show how intersection and difference can be used to select specific combinations of edge labels. Both expression e_3 and e_4 yields people who work with each other, while excluding self-loops (people that work with themselves).

The above use cases do rely on graph data models that allow multi-labeled edges and/or allows self-loops. Although the general graph data model allows these features, this is not the case for several more restrictive data models. Examples of these restrictive data models include the trees and chains studied in Part II and other hierarchical and tree data models such as XML and JSON [29, 58, 74]. For these restrictive data models, intersection and difference do not add any expressive power to the basic fragments of the relation algebra or the semi-join algebra. Next, we formalize and prove this.

Definition 8.6. Let $\mathcal{G} = (\mathcal{V}, \Sigma, \mathbf{E})$ be a graph. We say that an edge label $\ell \in \Sigma$ represents a *node label* if ℓ evaluates to a subset of id. An edge label $\ell \in \Sigma$ has *self-loops* if ℓ does not represent a node label and $\mathbf{E}(\ell)$ overlaps with a non-empty subset of $[\![\mathrm{id}]\!]_{\mathcal{G}}$. Graph $\mathcal{G}$ has *self-loops* if there exists an edge label $\ell \in \Sigma$ that is a self-loop. Graph $\mathcal{G}$ has *multi-edges* if there exists a pair of distinct edge labels $\ell_1, \ell_2 \in \Sigma$, both not being node labels, such that either $[\![\ell_1]\!]_{\mathcal{G}} \cap [\![\ell_2]\!]_{\mathcal{G}} \neq \emptyset$ or $[\![\ell_1]\!]_{\mathcal{G}} \cap [\![\ell_2{}^\frown]\!]_{\mathcal{G}} \neq \emptyset$.

Proposition 8.10. *Let* $\mathcal{F} \subseteq \{\mathrm{di}, {}^\frown, \pi, \overline{\pi}, \cap, -\}$.

(i) On the class of graphs without self-loops and multi-edges, we have $\mathcal{M}(\mathcal{F}) \equiv_{\mathrm{path}} \mathcal{M}(\underline{\mathcal{F}} - \{\cap, -\})$ *and* $\mathcal{M}^{\mathrm{fp}}(\mathcal{F}) \equiv_{\mathrm{path}} \mathcal{M}^{\mathrm{fp}}(\underline{\mathcal{F}} - \{\cap, -\})$ *if either* $\overline{\pi} \in \mathcal{F}$ *or* $- \notin \mathcal{F}$.

(ii) On the class of graphs without self-loops and multi-edges, and in which no edge labels represent node labels, we have $\mathcal{M}(\mathcal{F}) \equiv_{\mathrm{path}} \mathcal{M}(\underline{\mathcal{F}} - \{\cap, -\})$ *and* $\mathcal{M}^{\mathrm{fp}}(\mathcal{F}) \equiv_{\mathrm{path}} \mathcal{M}^{\mathrm{fp}}(\underline{\mathcal{F}} - \{\cap, -\})$.

Proof. By Corollary 8.8, we assume that expressions $\mathcal{M}(\mathcal{F})$ and $\mathcal{M}^{\mathrm{fp}}(\mathcal{F})$ are also expressions in $\mathcal{A}(\mathcal{F})$ and $\mathcal{A}^{\mathrm{fp}}(\mathcal{F})$. Hence, we only need to consider the usage of intersection and difference within basic expressions. Using the push down rules for intersection and difference as proposed in the proof of Theorem 8.7 and Lemma 3.2 (i), we can push down intersection and difference to the atoms $\emptyset$, id, di, ℓ, and $\ell^\frown$. Any combination of $\emptyset$, $\cap$, and di using intersection and/or difference is path-equivalent to either $\emptyset$, $\cap$, or di.

Next, we consider expressions utilizing a single edge label ℓ. If ℓ does not represent node labels, then, due to the graph not having self-loops, we have

$$\emptyset \equiv_{\mathrm{path}} \ell \cap \emptyset \equiv_{\mathrm{path}} \ell \cap \mathrm{id} \equiv_{\mathrm{path}} \ell - \mathrm{di} \equiv_{\mathrm{path}} \ell - \ell;$$
$$\emptyset \equiv_{\mathrm{path}} \ell^\frown \cap \emptyset \equiv_{\mathrm{path}} \ell^\frown \cap \mathrm{id} \equiv_{\mathrm{path}} \ell^\frown - \mathrm{di} \equiv_{\mathrm{path}} \ell^\frown - \ell^\frown;$$
$$\ell \equiv_{\mathrm{path}} \ell - \emptyset \equiv_{\mathrm{path}} \ell - \mathrm{id} \equiv_{\mathrm{path}} \ell \cap \mathrm{di} \equiv_{\mathrm{path}} \ell \cap \ell;$$
$$\ell^\frown \equiv_{\mathrm{path}} \ell^\frown - \emptyset \equiv_{\mathrm{path}} \ell^\frown - \mathrm{id} \equiv_{\mathrm{path}} \ell^\frown \cap \mathrm{di} \equiv_{\mathrm{path}} \ell^\frown \cap \ell^\frown.$$

Else, if ℓ is an edge label that represents a node label, then we have

$$\begin{aligned}
\emptyset &\equiv_{\text{path}} \ell \cap \emptyset \equiv_{\text{path}} \ell - \text{id} \equiv_{\text{path}} \ell \cap \text{di} \equiv_{\text{path}} \ell - \ell \\
\emptyset &\equiv_{\text{path}} \ell^\frown \cap \emptyset \equiv_{\text{path}} \ell^\frown - \text{id} \equiv_{\text{path}} \ell^\frown \cap \text{di} \equiv_{\text{path}} \ell^\frown - \ell^\frown; \\
\ell &\equiv_{\text{path}} \ell - \emptyset \equiv_{\text{path}} \ell \cap \text{id} \equiv_{\text{path}} \ell - \text{di} \equiv_{\text{path}} \ell \cap \ell; \\
\ell^\frown &\equiv_{\text{path}} \ell^\frown - \emptyset \equiv_{\text{path}} \ell^\frown \cap \text{id} \equiv_{\text{path}} \ell^\frown - \text{di} \equiv_{\text{path}} \ell^\frown \cap \ell^\frown.
\end{aligned}$$

Finally, we consider expressions utilizing distinct edge labels ℓ_1 and ℓ_2. If e_1 and e_2 do not both represent node labels, then, due to the graph not having multi-edges, we have $\ell_1 \cap \ell_2 \equiv_{\text{path}} \emptyset$ and $\ell_1 - \ell_2 \equiv_{\text{path}} \ell_1$. Else, if ℓ_1 and ℓ_2 represent node labels, then we have $\ell_1 \cap \ell_2 \equiv_{\text{path}} \ell_1 \circ \ell_2$ and $\ell_1 - \ell_2 \equiv_{\text{path}} \ell_1 \circ \overline{\pi}_1[\ell_2]$. We observe that only when we use differences on edge labels that represent node labels, we introduce the need for coprojections, explaining why we require coprojections in this case. □

8.7 Results on trees and chains

Part II already presented many results on the expressive power of the relation algebra on trees and chains. These results can easily be combined with Theorems 8.6 and 8.7 and Corollary 8.8 to draw conclusions on the expressive power of the semi-join algebra on trees and chains.

Corollary 8.11. *Let $\mathcal{F} \subseteq \{^\frown, \pi, \overline{\pi}, \cap, -\}$. We have the following*

(i) On trees, we have $\mathcal{N}(\underline{\mathcal{F}}) \preceq_\pi \mathcal{M}(\underline{\mathcal{F}})$ and $\mathcal{N}(\underline{\mathcal{F}} \cup \{^*\}) \preceq_\pi \mathcal{M}^{\text{fp}}(\underline{\mathcal{F}})$ if $\mathcal{N}(\mathcal{F})$ is downward.

(ii) On chains, we have $\mathcal{N}(\underline{\mathcal{F}}) \preceq_\pi \mathcal{M}(\underline{\mathcal{F}})$ if $\mathcal{N}(\mathcal{F})$ is local.

We can also sharpen the result of Theorem 3.29 significantly:

Corollary 8.12. *On unlabeled chains, every first-order logic query is Boolean-equivalent to an expression in $\mathcal{M}(\overline{\pi})$.*

Next, we consider the node-labeled sibling-ordered XML data model and the Conditional XPath query language. On finite node-labeled sibling-ordered trees, Conditional XPath is path-equivalent to FO^{tree}: first-order logic on tree structures represented by a descendant and a following-sibling relation, unary node-label predicates, and equality [74, Proposition 2.7]. We shall prove that this collapse can be sharpened to a projection equivalence collapse of FO^{tree} queries to $\mathcal{M}^{\text{fp}}$. To prove this, we first provide a minimal introduction to Conditional XPath.

Conditional XPath is a syntactical fragment of Regular XPath, and Regular XPath is a query language for querying node-labeled sibling-ordered XML data [74]. Regular XPath distinguishes path formulae, which evaluate to binary relations, and node formulae, which evaluate to unary relations (sets of nodes). Path formulae are defined by the grammar[18]

$$\text{p\_wff} = \mathit{Edge} \mid \text{p\_wff} \circ \text{p\_wff} \mid \text{p\_wff} \cup \text{p\_wff} \mid [\text{p\_wff}]^* \mid ?\text{n\_wff},$$

in which $\mathit{Edge} \in \{\mathit{Child}, \mathit{Parent}, \mathit{Left}, \mathit{Right}\}$ denotes the edge relations (the parent-child axis and the ordered sibling axis), n_wff is a node formula, and ?n_wff interprets the node formulae as a binary relation. Node formulae are defined by the grammar

$$\text{n\_wff} = \ell \mid \text{id} \mid \pi_1[\text{p\_wff}] \mid \overline{\text{n\_wff}} \mid \text{n\_wff} \cup \text{n\_wff} \mid \text{n\_wff} \cap \text{n\_wff},$$

[18] We have slightly adapted the Regular XPath syntax to better match the syntax of $\mathcal{N}$ where possible.

in which ℓ denotes a node label.

As a first step towards the collapse of Boolean $\mathrm{FO}^{\mathrm{tree}}$ queries to the semi-join algebra, we claim that Regular XPath is path-equivalent to $\mathcal{N}(^\frown, \overline{\pi}, ^*)$. We prove this claim by rewriting path formulae to expressions in $\mathcal{N}(^\frown, \overline{\pi}, ^*)$ and node formulae to node expressions in $\mathcal{N}(^\frown, \overline{\pi}, ^*)$. We represent node labels using edge labels (see Definition 8.6). These choices result in a straightforward rewriting $\tau_{\mathsf{p\_wff}}(\mathsf{p\_wff})$ for path formulae p_wff. For rewritings involving node formulae, we have:

$$\begin{aligned}
\tau_{\mathsf{p_wff}}(?\mathsf{n_wff}) &= \pi_1[\tau_{\mathsf{n_wff}}(\mathsf{n_wff})];\\
\tau_{\mathsf{n_wff}}(\ell) &= \ell;\\
\tau_{\mathsf{n_wff}}(\mathrm{id}) &= \mathrm{id};\\
\tau_{\mathsf{n_wff}}(\pi_1[\mathsf{p_wff}]) &= \pi_1[\tau_{\mathsf{p_wff}}(\mathsf{p_wff})];\\
\tau_{\mathsf{n_wff}}(\overline{\mathsf{n_wff}}) &= \overline{\pi}_1[\tau_{\mathsf{n_wff}}(\mathsf{n_wff})];\\
\tau_{\mathsf{n_wff}}(\mathsf{n_wff}_1 \cup \mathsf{n_wff}_2) &= \tau_{\mathsf{n_wff}}(\mathsf{n_wff}_1) \cup \tau_{\mathsf{n_wff}}(\mathsf{n_wff}_2);\\
\tau_{\mathsf{n_wff}}(\mathsf{n_wff}_1 \cap \mathsf{n_wff}_2) &= \tau_{\mathsf{n_wff}}(\mathsf{n_wff}_1) \circ \tau_{\mathsf{n_wff}}(\mathsf{n_wff}_2).
\end{aligned}$$

As Conditional XPath is a restriction of Regular XPath in which the Kleene-star can only be applied to *steps* instead of generic expressions,[19] we conclude the following:

Proposition 8.13. *With respect to queries yielding binary relations evaluated on finite node-labeled sibling-ordered trees, we have Regular XPath* $\equiv_{\mathrm{path}} \mathcal{N}(^\frown, \overline{\pi}, ^*)$*, Conditional XPath* $\preceq_{\mathrm{path}} \mathcal{N}(^\frown, \overline{\pi}, ^*)$*, and* $\mathcal{N}(^\frown, \overline{\pi}, ^*) \not\preceq_{\mathrm{bool}}$ *Conditional XPath.*

Proof. To translate from Regular XPath to $\mathcal{N}(^\frown, \overline{\pi}, ^*)$, we use the rewrite rules $\tau_{\mathsf{p\_wff}}(\mathsf{p\_wff})$. For the other direction, we adapt the above rewrite rules. The only difficulty in this are subexpressions of the form $\pi_2[e]$ and $\overline{\pi}_2[e]$. We deal with these subexpressions by rewriting them towards $\pi_1[e']$ and $\overline{\pi}_1[e']$, respectively, in which e' is obtained from $[e]^{-1}$ by pushing down the converse step to the edge labels *Child* and *Right* (see Section 1.3). The remainder of the statement of the Proposition follows from the well-known relationships between Regular XPath and Conditional XPath [74]. □

We combine Theorem 8.6, Proposition 8.13, and a result from Marx [74]:

Corollary 8.14. *With respect to queries yielding binary relations evaluated on finite node-labeled sibling-ordered trees, we have Regular XPath* $\equiv_\pi \mathcal{M}^{\mathrm{fp}}(^\frown, \overline{\pi})$*, Conditional XPath* $\preceq_\pi \mathcal{M}^{\mathrm{fp}}(^\frown, \overline{\pi})$*, and* $\mathcal{M}^{\mathrm{fp}}(^\frown, \overline{\pi}) \not\preceq_{\mathrm{bool}}$ *Conditional XPath.*

Finally, we combine Corollary 8.14 with the collapse of $\mathrm{FO}^{\mathrm{tree}}$ to Conditional XPath [74, Proposition 2.7] to conclude the following:

Proposition 8.15. *With respect to queries yielding binary relations evaluated on finite node-labeled sibling-ordered trees, we have* $\mathrm{FO}^{\mathrm{tree}} \preceq_\pi \mathcal{M}^{\mathrm{fp}}(^\frown, \overline{\pi})$.

Unfortunately, it is not possible to strengthen Proposition 8.15 by showing that one can translate Conditional XPath to the two-variable fragment of $\mathrm{FO}^{\mathrm{tree}}$ via $\mathcal{M}^{\mathrm{fp}}(^\frown, \overline{\pi})$. This follows from the simple fact that the two-variable fragment of $\mathrm{FO}^{\mathrm{tree}}$ cannot express basic Conditional XPath constructions, including the edge relations *Child* and *Right*, and step-based conditional iteration via the descendant and the following-sibling relations.

[19] A step is an edge relation to which, optionally, a test is applied of the form ?n_wff.

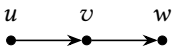


Figure 8.4: An unlabeled chain of length three.

8.8 The role of projection equivalence

In this chapter, we introduced projection equivalence to study rewriting and simplifying expressions while keeping either the first projection or second projection of the rewritten expression equivalent to the original expression. One can ask how projection equivalence relates to the more commonly studied path equivalence and Boolean equivalence.

Proposition 8.16. *Let* $\mathcal{F}_1, \mathcal{F}_2 \subseteq \{\mathrm{di}, \hat{}, \pi, \overline{\pi}, \cap, -, ^*\}$. *On labeled graphs, we have* $\mathcal{N}(\mathcal{F}_1) \preceq_\pi \mathcal{N}(\mathcal{F}_2)$ *if and only if* $\mathcal{N}(\mathcal{F}_1) \preceq_{\mathrm{bool}} \mathcal{N}(\mathcal{F}_2)$.

Proof. We always have that $\mathcal{N}(\mathcal{F}_1) \preceq_\pi \mathcal{N}(\mathcal{F}_2)$ implies $\mathcal{N}(\mathcal{F}_1) \preceq_{\mathrm{bool}} \mathcal{N}(\mathcal{F}_2)$. To prove that $\mathcal{N}(\mathcal{F}_1) \preceq_{\mathrm{bool}} \mathcal{N}(\mathcal{F}_2)$ implies $\mathcal{N}(\mathcal{F}_1) \preceq_\pi \mathcal{N}(\mathcal{F}_2)$, we distinguish two cases:

1. $\mathcal{N}(\mathcal{F}_1) \preceq_{\mathrm{path}} \mathcal{N}(\mathcal{F}_2)$. Then both $\mathcal{N}(\mathcal{F}_1) \preceq_\pi \mathcal{N}(\mathcal{F}_2)$ and $\mathcal{N}(\mathcal{F}_1) \preceq_{\mathrm{bool}} \mathcal{N}(\mathcal{F}_2)$.

2. $\mathcal{N}(\mathcal{F}_1) \not\preceq_{\mathrm{path}} \mathcal{N}(\mathcal{F}_2)$ *and* $\mathcal{N}(\mathcal{F}_1) \preceq_{\mathrm{bool}} \mathcal{N}(\mathcal{F}_2)$. In this case, there exists a fragment $\mathcal{F} \subseteq \{\mathrm{di}, \hat{}, \pi, \overline{\pi}\}$ such that $\mathcal{F}_1 = \mathcal{F} \cup \{\hat{}\}$ and $\mathcal{F}_2 = \mathcal{F} \cup \{\pi\}$. The proof of $\mathcal{N}(\mathcal{F}_1) \preceq_{\mathrm{bool}} \mathcal{N}(\mathcal{F}_2)$ in Fletcher et al. [31, Proof of Proposition 4.2] reveals that, for every expression e in $\mathcal{N}(\mathcal{F}_1)$, there exist expressions e_1 and e_2 in $\mathcal{N}(\mathcal{F}_2)$ such that $e_1 \equiv_{\mathrm{path}} \pi_1[e]$ and $e_2 \equiv_{\mathrm{path}} \pi_2[e]$. Hence, by definition, $\mathcal{N}(\mathcal{F}_1) \preceq_\pi \mathcal{N}(\mathcal{F}_2)$. □

We observe that Proposition 8.16 implies that we may not conclude from $\mathcal{L}_1 \equiv_\pi \mathcal{L}_2$ that $\mathcal{L}_1 \equiv_{\mathrm{path}} \mathcal{L}_2$. Next, we show that we may not conclude from $\mathcal{L}_1 \equiv_{\mathrm{bool}} \mathcal{L}_2$ that $\mathcal{L}_1 \equiv_\pi \mathcal{L}_2$.

Proposition 8.17. *Let* $\mathcal{F} \subseteq \{^*\}$. *On labeled chains, we have* $\mathcal{N}(\mathcal{F} \cup \{\pi\}) \preceq_{\mathrm{bool}} \mathcal{N}(\mathcal{F})$ *and* $\mathcal{N}(\mathcal{F} \cup \{\pi\}) \not\preceq_\pi \mathcal{N}(\mathcal{F})$.

Proof. By Theorem 4.12, we have $\mathcal{N}(\mathcal{F} \cup \{\pi\}) \preceq_{\mathrm{bool}} \mathcal{N}(\mathcal{F})$. We prove $\mathcal{N}(\mathcal{F} \cup \{\pi\}) \not\preceq_\pi \mathcal{N}(\mathcal{F})$ via a simple counterexample. Consider the query $e = \pi_2[\ell] \circ \pi_1[\ell]$ evaluated on the chain C of Figure 8.4. This query will return the node-pair (v, v). An exhaustive search shows that no expression in $\mathcal{N}(\mathcal{F})$ is j-test-equivalent to e, $j \in \{1, 2\}$. □

Using Proposition 8.16 and Proposition 8.17, we conclude the following:

Corollary 8.18. *Let* $\mathcal{L}_1$ *and* $\mathcal{L}_2$ *be query languages. Then* $\mathcal{L}_1 \equiv_{\mathrm{bool}} \mathcal{L}_2$ *does not imply* $\mathcal{L}_1 \equiv_\pi \mathcal{L}_2$, *and* $\mathcal{L}_1 \equiv_\pi \mathcal{L}_2$ *does not imply* $\mathcal{L}_1 \equiv_{\mathrm{path}} \mathcal{L}_2$.

CHAPTER 9

Optimizing graph query evaluation[20]

In this chapter, we review the composition-to-semi-join rewrite rules presented in Chapter 8 and investigate their usefulness with respect to *graph query optimization*. To do so, we first provide a simple cost model for graph query evaluation and show why we consider the composition and Kleene-plus operators to be expensive, and the semi-join and fixpoint operators to be inexpensive. Our main results are as follows:

1. We propose an efficient algorithm to evaluate the fixpoint operators we consider.
2. We revisit the analysis of the semi-join rewrite rules (Theorem 8.6) and add to this analysis by providing strict bounds on the size of rewritten expressions. We show that if the input of the semi-join rewrite rules is an expression of length s that uses at most u union-operators, then application of the rewrite rules we proposed yields a rewritten expression that can be evaluated in at most $s + u \leq 2 \cdot s$ evaluation steps, demonstrating the practical feasibility of our rewrite techniques.
3. We show that the semi-join rewrite rules provide strict guarantees on the *data complexity*—the complexity in terms of the size of the graph—of evaluating queries by evaluating each of the operators involved. The application of the semi-join rewrite rules can decrease the data complexity of query evaluation significantly, and will never increase it.
4. We also identify the identity, diversity, and coprojection operators as expensive to evaluate. We show how common usage of these operators can be replaced by specialized, efficient to evaluate, operators.

9.1 Organization

In Section 9.2, we introduce a simple cost model for evaluating operators in the relation algebra and the semi-join algebra. In Section 9.3, we show that fixpoints can be evaluated efficiently. In Section 9.4, we apply the developed cost model to the rewrite rules of Chapter 8. In Section 9.5, we take a look at usages of identity, diversity, and coprojections. Finally, in Section 9.6, we conclude on the above findings by casting them in terms of the impact on the complexity of query evaluation.

[20]The results in this chapter are partly based on the paper "From relation algebra to semi-join algebra: an approach for graph query optimization" [55].

9.2 The cost of evaluating operators

The operators in the relation algebra and semi-join algebra can easily and efficiently be evaluated using specialized versions of the many query evaluation algorithms that are used in traditional relational database management systems [16, 38, 60, 62, 69, 73, 85, 94]. The only exception is evaluation of the fixpoint operators, which we discuss in-depth in Section 9.3. Here, we focus on giving a cost model for the other operators. A detailed cost model for the evaluation cost of queries evaluation involves many factors and is outside the scope of this work. Luckily, the worst-case cost of each operator we consider is almost entirely determined by the size of the evaluation result of its operands and the size of its result, even if we use naive algorithms for evaluating these operators [38, 85, 94]. Hence, based on this observation, we will develop and use a high-level worst-case evaluation cost model in which the cost of evaluating operators is based on the size of the evaluation results of its operands and on the size of the evaluation result of the operator. We have the following:

Proposition 9.1. *Let $\mathcal{G} = (\mathcal{V}, \Sigma, \mathbf{E})$ be a graph and let $i \in \{1, 2\}$. We have*

$$
\begin{aligned}
|[\![\emptyset]\!]_{\mathcal{G}}| &= 0;\\
|[\![\mathrm{id}]\!]_{\mathcal{G}}| &= |\mathcal{V}|;\\
|[\![\mathrm{di}]\!]_{\mathcal{G}}| &= |\mathcal{V}|^2 - |\mathcal{V}|;\\
|[\![\pi_i[e]]\!]_{\mathcal{G}}| &= |[\![e]\!]_{\mathcal{G}}|_i|;\\
|[\![\overline{\pi}_i[e]]\!]_{\mathcal{G}}| &= |\mathcal{V}| - |[\![e]\!]_{\mathcal{G}}|_i|;\\
|[\![e_1 \circ e_2]\!]_{\mathcal{G}}| &\leq |[\![e_1]\!]_{\mathcal{G}}|_1| \cdot |[\![e_2]\!]_{\mathcal{G}}|_2|;\\
|[\![e_1 \ltimes e_2]\!]_{\mathcal{G}}| &\leq |[\![e_1]\!]_{\mathcal{G}}|;\\
|[\![e_1 \rtimes e_2]\!]_{\mathcal{G}}| &\leq |[\![e_2]\!]_{\mathcal{G}}|;\\
|[\![e_1 \cup e_2]\!]_{\mathcal{G}}| &\leq |[\![e_1]\!]_{\mathcal{G}}| + |[\![e_2]\!]_{\mathcal{G}}|;\\
|[\![e_1 \cap e_2]\!]_{\mathcal{G}}| &\leq \min(|[\![e_1]\!]_{\mathcal{G}}|, |[\![e_2]\!]_{\mathcal{G}}|);\\
|[\![e_1 - e_2]\!]_{\mathcal{G}}| &\leq |[\![e_1]\!]_{\mathcal{G}}|;\\
|[\![[e]^*]\!]_{\mathcal{G}}| &\leq |[\![e]\!]_{\mathcal{G}}|_1| \cdot |[\![e]\!]_{\mathcal{G}}|_2|.
\end{aligned}
$$

Based on Proposition 9.1, it is tempting to categorize the operators in three complexity classes.[21]

Definition 9.1. An operator is *expression-linear* if it guarantees to yield a result linearly upper-bounded by the size of the evaluation result of its operands. An operator is *node-linear* if it is not expression-linear, but still guarantees to yield a result linearly upper-bounded by the number of nodes (independent of the size of the evaluation result of any operands). An operator is *non-linear* if it does not fall in the above two categories.

We deem expression-linear operators to be the least expensive and the non-linear operators to be most expensive. We classify the relation algebra and semi-join algebra operators as follows:

Proposition 9.2. *The operators π_1, π_2, $\ltimes$, $\rtimes$, $\cup$, $\cap$, and $-$ are expression-linear, the operators* id, *$\overline{\pi}_1$, and $\overline{\pi}_2$ are node-linear, and the operators* di, *$\circ$, and * are non-linear.*

[21] The categorization is similar to the categorization of Leinders and Van den Bussche [69].

Proof. From Proposition 9.1 it already follows that π_1, π_2, $\ltimes$, $\rtimes$, $\cup$, $\cap$, and $-$ are expression-linear. To show that the operators id, $\overline{\pi}_1$, and $\overline{\pi}_2$ are node-linear, we provide an example showing that they are not expression-linear. To show that the operators di, $\circ$, and * are non-linear, we provide an example showing that they are not expression-linear and node-linear. Let $\mathcal{G} = (\mathcal{V}, \Sigma, \mathbf{E})$ be a graph with $\mathcal{V} = \{m, n_1, \ldots, n_{|\mathcal{V}|-1}\}$, $\Sigma = \{\ell, \ell'\}$, $\mathbf{E}(\ell) = \{(m, n_i), (n_i, m) \mid 1 \leq i \leq |\mathcal{V}| - 1\}$, and $\mathbf{E}(\ell') = \{(m, m)\}$. We have

$$\begin{aligned}
[\![\mathrm{id}]\!]_{\mathcal{G}} &= \{(n', n') \mid n' \in \mathcal{V}\};\\
[\![\mathrm{di}]\!]_{\mathcal{G}} &= \mathcal{V}^2 - \{(v, v) \mid v \in \mathcal{V}\};\\
[\![\overline{\pi}_1[\ell']]\!]_{\mathcal{G}} &= [\![\overline{\pi}_2[\ell']]\!]_{\mathcal{G}} = \{(n_i, n_i) \mid 1 \leq i \leq |\mathcal{V}| - 1\};\\
[\![\ell \circ \ell]\!]_{\mathcal{G}} &= \mathcal{V}^2 - \mathbf{E}(\ell);\\
[\![[\ell]^*]\!]_{\mathcal{G}} &= \mathcal{V}^2.
\end{aligned}$$

Observe that $|[\![\mathrm{id}]\!]_{\mathcal{G}}| = |\mathcal{V}|$, $|[\![\mathrm{di}]\!]_{\mathcal{G}}| = |\mathcal{V}|^2 - |\mathcal{V}|$, $|[\![\overline{\pi}_1[\ell']]\!]_{\mathcal{G}}| = |\mathcal{V}| - 1$, and $|[\![\ell \circ \ell]\!]_{\mathcal{G}}| = |\mathcal{V}|^2 - 2 \cdot (|\mathcal{V}| - 1)$. □

Proposition 9.2 illustrates why we consider composition and the Kleene-star to be expensive and why also identity, diversity, and coprojections are to be avoided.

9.3 Efficient evaluation of fixpoints

Due to the restrictions put on fixpoints, they can be evaluated very efficiently, which we will show next. Let $f = \mathsf{fp}_{j,\mathfrak{N}}[e \text{ union } b]$ be an expression without free variables. The complexity of evaluating f is determined by the *recursion steps of* f, denoted by $\mathcal{R}(f)$, and the cost of evaluating the *non-recursive terms of* f. We denote the non-recursive terms of f by $\mathcal{T}(f)$. We define $\mathcal{R}(f) = \mathcal{R}(e)$ with

$$\begin{aligned}
\mathcal{R}(\mathfrak{N}) &= 1;\\
\mathcal{R}(e_1 \ltimes e_2) = \mathcal{R}(e_2 \rtimes e_1) &= 1 + \mathcal{R}(e_1);\\
\mathcal{R}(e_1 \cup e_2) &= \mathcal{R}(e_1) + \mathcal{R}(e_2) + 1;\\
\mathcal{R}(\mathsf{fp}_{j,\mathfrak{N}'}[e' \text{ union } b']) &= \mathcal{R}(b') + \mathcal{R}(e') + 1,
\end{aligned}$$

with $\mathfrak{N}$ a variable, and we define $\mathcal{T}(f)$ to be the multiset $\mathcal{T}(f) = [b] + \mathcal{T}(e)$ with

$$\begin{aligned}
\mathcal{T}(\mathfrak{N}) &= [\,]\\
\mathcal{T}(e_1 \ltimes e_2) = \mathcal{T}(e_2 \rtimes e_1) &= [e_2] + \mathcal{T}(e_1);\\
\mathcal{T}(e_1 \cup e_2) &= \mathcal{T}(e_1) + \mathcal{T}(e_2);\\
\mathcal{T}(\mathsf{fp}_{j,\mathfrak{N}'}[e' \text{ union } b']) &= \mathcal{T}(b') + \mathcal{T}(e').
\end{aligned}$$

Proposition 9.3. *Let $\mathcal{G} = (\mathcal{V}, \Sigma, \mathbf{E})$ be a graph and let $f = \mathsf{fp}_{j,\mathfrak{N}}[e \text{ union } b]$ be an expression without free variables. The worst-case cost for evaluating $[\![f]\!]_{\mathcal{G}}$ is $O(\mathcal{R}(f) \cdot n + s + c)$, in which*

$$\begin{aligned}
n &= \max\{|[\![t]\!]_{\mathcal{G}}|_j \mid t \in \mathcal{T}(f)\};\\
s &= \sum\{|[\![t]\!]_{\mathcal{G}}| \mid t \in \mathcal{T}(f)\},
\end{aligned}$$

and c is the total cost of evaluating the expressions in $\mathcal{T}(f)$.

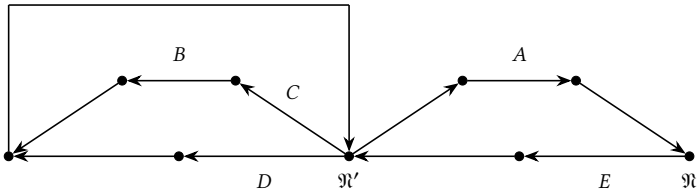


Figure 9.1: Graph representing the expression $e = \mathsf{fp}_{1,\mathfrak{N}}[A \ltimes e' \text{ union } F]$ with $e' = \mathsf{fp}_{1,\mathfrak{N}'}[(B \ltimes (C \ltimes \mathfrak{N}')) \cup (D \ltimes \mathfrak{N}') \text{ union } E \ltimes \mathfrak{N}]$. This graph is obtained by applying the graph representation construction of the proof of Proposition 9.3 on e.

Proof. We observe that, in expression e, there is no negation on the path towards the variable $\mathfrak{N}$: we only allow union, semi-joins, and fixpoints, and we do not allow difference and coprojections. Hence, if we interpret the expressions in $\mathcal{T}(f)$ as pre-computed edge labels, then the restricted language we consider is expressible in a subset of the alternation-free μ-calculus, for which very efficient evaluation algorithms exist [22].

Based on these algorithms, we sketch how to efficiently evaluate the fixpoint expression f when $j = 1$. The case for $j = 2$ is analogous. To evaluate the fixpoint expression f, we first translate the expression into a graph representation. We do so by making edge-connections between expressions in the following way.

1. Add an unlabeled connection from the expression e to the expression $\mathfrak{N}$.
2. For any right-recursive subexpression $e_1 \ltimes e_2$, add a connection labeled e_1 from the expression e_2 to the expression $e_1 \ltimes e_2$.
3. For any right-recursive subexpression $e_1 \cup e_2$, add unlabeled connections from the expressions e_1 and e_2 to the expression $e_1 \cup e_2$.
4. For any right-recursive subexpression $\mathsf{fp}_{1,\mathfrak{N}'}[e' \text{ union } b']$, add an unlabeled connection from the expression $\mathfrak{N}'$ to the expression $\mathsf{fp}_{1,\mathfrak{N}'}[e' \text{ union } b']$ and from the expression b' to the expression $\mathfrak{N}'$.

Figure 9.1 provides an example of the resulting graph representation of a fixpoint expression.

The graph representation is used for a message-passing evaluation algorithm in which each expression-node maintains a set of received graph-nodes. When an expression-node receives a graph-node v it has not yet received, then it sends v to every expression-node to which it has an unlabeled connection and it sends w to every expression-node to which it has a connection labeled e' with $(w, v) \in [\![e']\!]_{\mathcal{G}}$. We initialize this process by sending each graph-node in $[\![b]\!]_{\mathcal{G}}|_1$ to the expression-node $\mathfrak{N}$. Let S be the set of all graph-nodes received by $\mathfrak{N}$ after all messages have been processed. We have $[\![f]\!]_{\mathcal{G}} = \{(v, v) \mid v \in S\}$.

Over every unlabeled connection, at most n messages are sent and over every connection labeled with e, at most $|[\![e]\!]_{\mathcal{G}}|$ messages are sent. The number of expression-nodes and the number of unlabeled connections are both worst-case $O(\mathcal{R}(f))$ and for every non-recursive term in $\mathcal{T}(f)$ there is exactly one labeled connection. Hence, at most $O(\mathcal{R}(f) \cdot n + s)$ messages need to be sent. □

Fixpoints do not really suit the classification we used for the other operators, as its operands cannot be evaluated independently due to the recursion involved. To obtain a

classification in the spirit of Definition 9.1, we can view the set of non-recursive terms $\mathcal{T}(f)$ as the operands of a fixpoint. Using this view, we can classify the fixpoint operator as an *expression-linear* operator.

9.4 Revising the analysis of the semi-join rewriting

We already claimed soundness of the rewrite rules of Figure 8.1. To claim their usability for query optimization, we will analyze the complexity of the expression resulting from the rewrite next. We do so in terms of the expression size, the number of steps needed for evaluation, and the complexity of the operators involved. In this analysis, we use the following terminology:

Definition 9.2. The *size* of an expression e, denoted by $\|e\|$, is the number of operations in e. We have

$$\begin{aligned}
\|\emptyset\| = \|\mathrm{id}\| = \|\mathrm{di}\| = \|\ell\| = \|\ell^\frown\| &= 0;\\
\|f_j[e]\| &= 1 + \|e\|;\\
\|e_1 \otimes e_2\| &= 1 + \|e_1\| + \|e_2\|;\\
\|e_1 \oplus e_2\| &= 1 + \|e_1\| + \|e_2\|;\\
\|[e]^*\| &= 1 + \|e\|;\\
\|\mathfrak{N}\| &= 0;\\
\|\mathsf{fp}_{j,\mathfrak{N}}[e \text{ union } b]\| &= 1 + \|e\| + \|b\|,
\end{aligned}$$

with $f \in \{\pi, \overline{\pi}\}$, $j \in \{1, 2\}$, $\otimes \in \{\circ, \ltimes, \rtimes\}$, and $\oplus \in \{\cup, \cap, -\}$. The *subexpression set* of e, denoted by $\mathcal{S}(e)$, is the set of all unique non-atomic subexpressions that must be evaluated:

$$\begin{aligned}
\mathcal{S}(\emptyset) = \mathcal{S}(\mathrm{id}) = \mathcal{S}(\mathrm{di}) = \mathcal{S}(\ell) = \mathcal{S}(\ell^\frown) &= \emptyset;\\
\mathcal{S}(f_j[e]) &= \{f_j[e]\} \cup \mathcal{S}(e);\\
\mathcal{S}(e_1 \otimes e_2) &= \{e_1 \otimes e_2\} \cup \mathcal{S}(e_1) \cup \mathcal{S}(e_2);\\
\mathcal{S}(e_1 \oplus e_2) &= \{e_1 \oplus e_2\} \cup \mathcal{S}(e_1) \cup \mathcal{S}(e_2);\\
\mathcal{S}([e]^*) &= \{[e]^*\} \cup \mathcal{S}(e);\\
\mathcal{S}(\mathfrak{N}) &= \emptyset;\\
\mathcal{S}(\mathsf{fp}_{j,\mathfrak{N}}[e \text{ union } b]) &= \{\mathsf{fp}_{j,\mathfrak{N}}[e \text{ union } b]\} \cup \mathcal{S}(e) \cup \mathcal{S}(b).
\end{aligned}$$

The *evaluation size* of e is defined by $\text{eval-steps}(e) = |\mathcal{S}(e)|$.

Example 9.1. Consider the expression

$$e = ((\ell \circ \ell) \circ (\ell \circ \ell)) \circ ((\ell \circ \ell) \circ (\ell \circ \ell)).$$

We have $\|e\| = 7$, whereas we have $\text{eval-steps}(e) = 3$. Indeed, this expression can be evaluated in three steps, namely by first evaluating $e_1 = \ell \circ \ell$, next $e_2 = e_1 \circ e_1$, and, finally, $e = e_2 \circ e_2$. Now consider the expression

$$e' = \ell \circ (\ell \circ (\ell \circ (\ell \circ (\ell \circ (\ell \circ \ell))))),$$

for which we have $e \equiv_{\text{path}} e'$ and $\|e'\| = \text{eval-steps}(e') = 7$.

Next, we characterize the impact of the rewrite rule of Figure 8.1 on the evaluation size and expression size of rewritten expressions. Observe that the only rewrite rules of Figure 8.1 that increases the expression size significantly are the rewrite rules $\tau_{\circ_i}(e_1 \cup e_2; \varepsilon) = \tau_{\circ_i}(e_1; \varepsilon) \cup \tau_{\circ_1}(e_2; \varepsilon)$, $i \in \{1, 2\}$, as these rewrite rules duplicate the expression ε. By $\mathsf{u}(\tau(e))$, $\mathsf{u}(\tau_{\pi_j}(e))$, and $\mathsf{u}(\tau_{\circ_j}(e; \varepsilon))$, $j \in \{1, 2\}$, we denote the number of times the rewrite rules $\tau_{\circ_i}(e_1 \cup e_2; \varepsilon) = \tau_{\circ_i}(e_1; \varepsilon) \cup \tau_{\circ_1}(e_2; \varepsilon)$, $i \in \{1, 2\}$, have been applied in the rewriting of e using $\tau(e)$, $\tau_{\pi_j}(e)$, or $\tau_{\circ_j}(e; \varepsilon')$, respectively. We have the following:

Theorem 9.4. *Let e be an expression in $\mathcal{N}$.*

(i) We have $\tau(e) \equiv_{\text{path}} e$, $\text{eval-steps}(\tau(e)) \leq \mathsf{u}(\tau(e)) + \|e\|$, and $\|\tau(e)\| = \Theta(\|e\| \cdot 2^{\mathsf{u}(\tau(e))})$ in the worst case.

(ii) Let $i \in \{1, 2\}$. We have $\tau_{\pi_i}(e) \equiv_{\pi_i} e$, $\text{eval-steps}(\tau_{\pi_i}(e)) \leq \mathsf{u}(\tau_{\pi_i}(e)) + \|e\|$, and $\|\tau_{\pi_i}(e)\| = \Theta(\|e\| \cdot 2^{\mathsf{u}(\tau_{\pi_i}(e))})$ in the worst case.

Proof. We first prove $\|\tau(e)\| = \Omega(\|e\| \cdot 2^{\mathsf{u}(\tau(e))})$ in the worst case. In the worst case, we have $\mathsf{u}(\tau(e)) = \Theta(\|e\|)$. Let $e = \pi_1[(\ell_1 \circ \ell_2 \cup \ell_2 \circ \ell_1)^p \circ \ell^{p+1}]$. We have $\|(\ell_1 \circ \ell_2 \cup \ell_2 \circ \ell_1)^p\| = 4p - 1$, $\|\ell^{p+1}\| = p$, $\|e\| = 5p$, and $\mathsf{u}(\tau(e)) = p$. This expression is rewritten into $e' = \pi_1[e_1]$ with, for i, $1 \leq i < p$, $e_i = \ell_1 \ltimes (\ell_2 \ltimes e_{i+1}) \cup \ell_2 \ltimes (\ell_1 \ltimes e_{i+1})$, and $e_p = \ell \ltimes (\ell \ltimes (\ldots \ltimes \ell) \ldots)$. We have $\|e'\| = 1 + \|e_1\|$ with $\|e_i\| = 5 + 2 \cdot \|e_{i+1}\|$ and $\|e_p\| = p$. Hence,

$$\begin{aligned}\|e'\| &= 1 + 5 \cdot (2^0 + 2^1 + \cdots + 2^{p-2}) + p \cdot 2^{p-1} \\ &\geq (p \cdot 2^p)/2 \\ &= ((\|e\|/5) \cdot 2^{(\|e\|/5)})/2 = \Omega(\|e\| \cdot 2^{\mathsf{u}(\tau(e))}).\end{aligned}$$

To prove that $\|\tau_{\pi_i}(e)\| = \Omega(\|e\| \cdot 2^{\mathsf{u}(\tau_{\pi_i}(e))})$ in the worst case, it now suffices to observe that $\tau_{\pi_i}(e) = \tau(e)$.

To prove the remainder of the Theorem, it suffices to show that $\tau(e)$, $\tau_{\pi_i}(e)$, $\tau_{\circ_1}(e; \varepsilon)$, and $\tau_{\circ_1}(e; \varepsilon)$, with e an expression in $\mathcal{N}$ and ε an expression, satisfy the following conditions:

1. If $x = \tau(e)$ and $u = \mathsf{u}(\tau(e))$, then $x \equiv_{\text{path}} e$, $\|x\| \leq \|e\| \cdot 2^u$, and $\text{eval-steps}(x) \leq u + \|e\|$.
2. If $i \in \{1, 2\}$, $x = \tau_{\pi_i}(e)$, and $u = \mathsf{u}(\tau_{\pi_i}(e))$, then $x \equiv_{\pi_i} e$, $\|x\| \leq \|e\| \cdot 2^u$, and $\text{eval-steps}(x) \leq u + \|e\|$.
3. If ε is non-recursive or right-recursive in variable $\mathfrak{N}$, $x = \tau_{\circ_1}(e; \varepsilon)$, and $u = \mathsf{u}(\tau_{\circ_1}(e; \varepsilon))$, then $x \equiv_{\pi_1} e \ltimes \varepsilon$, $\|x\| \leq (\|e\| + \|\varepsilon\| + 1) \cdot 2^u$, and $\mathcal{S}(x) = \mathcal{S}(\varepsilon) \cup T$ with $|T| \leq \|e\| + u + 1$. If ε is right-recursive in variable $\mathfrak{N}$, then so is x, else x is non-recursive.
4. If ε is non-recursive or left-recursive in variable $\mathfrak{N}$, $x = \tau_{\circ_2}(e; \varepsilon)$, and $u = \mathsf{u}(\tau_{\circ_2}(e; \varepsilon))$, then $x \equiv_{\pi_2} \varepsilon \rtimes e$, $\|x\| \leq (\|e\| + \|\varepsilon\| + 1) \cdot 2^u$, and $\mathcal{S}(x) = \mathcal{S}(\varepsilon) \cup T$ with $|T| \leq \|e\| + u + 1$. If ε is left-recursive in variable $\mathfrak{N}$, then so is x, else x is non-recursive.

These properties are straightforward to prove using induction on the length of e. The base cases are the basic expressions and Lemma 8.3 is used to prove the inductive steps. □

The rules of Figure 8.1 are introduced with the sole purpose of proving relationships between fragments of the relation algebra and the semi-join algebra. Consequently, these rewrite rules are not fully designed with respect to optimizing query evaluation, and will not necessarily improve query evaluation performance at every evaluation step, as illustrated next.

Example 9.2. Consider the expression $e = \pi_1[\ell_1 \circ \ell_2]$. We have $\tau(e) = \pi_1[\ell_1 \ltimes \ell_2]$. Now consider the graph $\mathcal{G} = (\mathcal{V}, \Sigma, \mathbf{E})$ with $\mathcal{V} = \{m, n_1, \ldots, n_{|\mathcal{V}|-1}\}$, $\Sigma = \{\ell_1, \ell_2\}$, $\mathbf{E}(\ell_1) = \{(m, n_i) \mid 1 \leq i \leq |\mathcal{V}| - 1\}$, and $\mathbf{E}(\ell_2) = \{(n_i, m) \mid 1 \leq i \leq |\mathcal{V}| - 1\}$. We shall argue that evaluation of e by first evaluating the composition and then evaluating the projection is less costly than evaluation of $\tau(e)$. Observe that we have

$$[\![\ell_1 \circ \ell_2]\!]_{\mathcal{G}} = \{(m, m)\};$$
$$[\![\ell_1 \ltimes \ell_2]\!]_{\mathcal{G}} = \mathbf{E}(\ell_1).$$

Due to the intermediate result of evaluating $\ell_1 \circ \ell_2$ being much smaller than the intermediate result of evaluating $\ell_1 \ltimes \ell_2$, straightforward projection algorithms will perform the projection step in e at a much lower cost than the projection step in $\tau(e)$. Moreover, in this specific example, algorithms for computing the composition $\ell_1 \circ \ell_2$ can easily achieve comparable performance to algorithms for computing the semi-join $\ell_1 \ltimes \ell_2$.

In the following, we explore how the rewrite rules of Figure 8.1 can be adjusted and used for graph query optimization. Remember that the cost of all operators is influenced primarily by the size of the evaluation results of its operands. From this observation, the issue shown in Example 9.2 can easily be explained: the rewrite rules of Figure 8.1 can rewrite expression e into an expression e' such that $|[\![e]\!]_{\mathcal{G}}| < |[\![e']\!]_{\mathcal{G}}|$.

We change the rewrite rules of Figure 8.1 in such a way that the resulting rules provide the following strong guarantee: the size of the evaluation results of a rewritten expression will always be upper bounded by the size of the evaluation results of the original expression. We do so by modifying $\tau_{\circ_i}(e; \varepsilon)$ by replacing rewritings of the form $g \ltimes \varepsilon$ by $\pi_1[g \ltimes \varepsilon]$ and of the form $\varepsilon \rtimes g$ by $\pi_2[\varepsilon \rtimes g]$.

Proposition 9.5. *Let $\mathcal{G}$ be a graph, let e be an expression in $\mathcal{N}$, and let ε be an expression. If we use the rewrite rules of Figure 8.1 with the above modifications, then $\tau_{\circ_1}(e; \varepsilon)$ and $\tau_{\circ_2}(e; \varepsilon)$ are node expressions and their evaluation yields the smallest possible sets such that $[\![\tau_{\circ_1}(e; \varepsilon)]\!]_{\mathcal{G}}|_1 = [\![e \ltimes \varepsilon]\!]_{\mathcal{G}}|_1$ and $[\![\tau_{\circ_2}(e; \varepsilon)]\!]_{\mathcal{G}}|_2 = [\![\varepsilon \rtimes e]\!]_{\mathcal{G}}|_2$.*

With a minimal modification to the rewrite rules for $\tau_{\pi_i}(e)$, $i \in \{1, 2\}$, we can also guarantee that $\tau_{\pi_i}(e)$ minimizes intermediate evaluation results in the same way as the modified version of $\tau_{\circ_i}(e; \varepsilon)$ does. We do so by, additionally, applying the following two changes to $\tau_{\pi_i}(e)$:

$$\tau_{\pi_i}(b) = \pi_i[b];$$
$$\tau_{\pi_i}(e_1 \oplus e_2) = \pi_i[\tau(e_1) \oplus \tau(e_2)],$$

in which b is a basic expression and $\oplus \in \{\cap, -\}$. Using a straightforward induction argument, we obtain

Proposition 9.6. *Let $\mathcal{G}$ be a graph and let e be an expression in $\mathcal{N}$. If we use the rewrite rules of Figure 8.1 with the above modifications, then $\tau_{\pi_1}(e)$ and $\tau_{\pi_2}(e)$ are node expressions and their evaluation yields the smallest possible sets such that $[\![\tau_{\pi_1}(e)]\!]_{\mathcal{G}}|_1 = [\![e]\!]_{\mathcal{G}}|_1$ and $[\![\tau_{\pi_2}(e)]\!]_{\mathcal{G}}|_2 = [\![e]\!]_{\mathcal{G}}|_2$.*

From Proposition 9.6, we conclude:

Corollary 9.7. *Let $\mathcal{G}$ be a graph, let e be an expression in $\mathcal{N}$, and $i \in \{1, 2\}$. If we use the rewrite rules of Figure 8.1 with the above modifications, then we have $|[\![\tau_{\pi_i}(e)]\!]_{\mathcal{G}}| \leq |[\![e]\!]_{\mathcal{G}}|$.*

We observe that, in practice, the projection-steps introduced by modifying $\tau_{\circ_i}(e;\varepsilon)$ and $\tau_{\pi_i}(e)$ can easily be integrated into specialized versions of the operators $\ltimes$, $\rtimes$, $\cup$, $\cap$, and $-$. These specialized single-column operators can be evaluated at least as efficient as the original operators. This also holds for single-column specializations of all other operators used by the relation algebra and the semi-join algebra, which makes an attractive proposition to eliminate every usage of the projection operators. Hence, even though strictly speaking the number of evaluation steps slightly increases by the above changes, this does not translate into an increase in the evaluation cost of the resultant rewritten expressions if these operators are properly implemented.

9.5 Dealing with other expensive operators

The semi-join rewrite rules introduced are aimed at optimizing query evaluation for common usages of compositions and Kleene-stars in cases where their full expressive power is unnecessary. In Section 9.2, we argued that also identity, diversity, and coprojections are to be avoided. Next, we recognize common usages in which the full expressive power of identity, diversity, and coprojections is unnecessary. To enable optimization of query evaluation in these common cases, we introduce the selection operators $\sigma_{=}$ and $\sigma_{\neq}$ and the anti-semi-join operators $\bar{\ltimes}$ and $\bar{\rtimes}$. The semantics of evaluation of these operators is defined by:

$$\begin{aligned}
[\![\sigma_{=}(e)]\!]_{\mathcal{G}} &= \{(n_1,n_2) \mid (n_1,n_2) \in [\![e]\!]_{\mathcal{G}} \wedge (n_1 = n_2)\};\\
[\![\sigma_{\neq}(e)]\!]_{\mathcal{G}} &= \{(n_1,n_2) \mid (n_1,n_2) \in [\![e]\!]_{\mathcal{G}} \wedge (n_1 \neq n_2)\};\\
[\![e_1 \mathbin{\bar{\ltimes}} e_2]\!]_{\mathcal{G}} &= \{(m,n) \mid (m,n) \in [\![e_1]\!]_{\mathcal{G}} \wedge \neg\exists z\,(n,z) \in [\![e_2]\!]_{\mathcal{G}}\};\\
[\![e_1 \mathbin{\bar{\rtimes}} e_2]\!]_{\mathcal{G}} &= \{(m,n) \mid (m,n) \in [\![e_2]\!]_{\mathcal{G}} \wedge \neg\exists z\,(z,m) \in [\![e_1]\!]_{\mathcal{G}}\}.
\end{aligned}$$

Before we use these newly introduced operators, we classify these operators in the style of Proposition 9.1 and Proposition 9.2.

Proposition 9.8. *Let $\mathcal{G}$ be a graph and let e, e_1, and e_2 be expressions. We have:*

$$\begin{aligned}
|[\![\sigma_{=}(e)]\!]_{\mathcal{G}}| &\leq |[\![e]\!]_{\mathcal{G}}|;\\
|[\![\sigma_{\neq}(e)]\!]_{\mathcal{G}}| &\leq |[\![e]\!]_{\mathcal{G}}|;\\
|[\![e_1 \mathbin{\bar{\ltimes}} e_2]\!]_{\mathcal{G}}| &\leq |[\![e_1]\!]_{\mathcal{G}}|;\\
|[\![e_1 \mathbin{\bar{\rtimes}} e_2]\!]_{\mathcal{G}}| &\leq |[\![e_2]\!]_{\mathcal{G}}|.
\end{aligned}$$

The operators $\sigma_{=}$, $\sigma_{\neq}$, $\bar{\ltimes}$, and $\bar{\rtimes}$ are expression-linear.

Next, we look at ways to rewrite common usages of identity and diversity. As illustrated in Section 1.2 and by Example 8.8, common usages of identity and diversity involve intersection and difference. In these usages, identity and diversity are used to restrict query results to keep only node pairs of the form (n,n) or to restrict query results to filter out node pairs of the form (n,n). In these use cases, we can introduce selection operators:

Proposition 9.9. *Let e be an expression. We have:*

(i) $e \cap \mathrm{id} \equiv_{\mathrm{path}} \mathrm{id} \cap e \equiv_{\mathrm{path}} e - \mathrm{di} \equiv_{\mathrm{path}} \sigma_{=}(e)$;

(ii) $e \cap \mathrm{di} \equiv_{\mathrm{path}} \mathrm{di} \cap e \equiv_{\mathrm{path}} e - \mathrm{id} \equiv_{\mathrm{path}} \sigma_{\neq}(e)$.

We observe that expression of the form $e - \mathrm{di}$ and $\sigma_{=}(e)$ are node expressions. Hence, we can add these cases to the definition of $\mathsf{ns}(\cdot)$. Besides the above usages of identity, we observe that Lemma 3.2 (iii) already specifies how to eliminate redundant id terms from relation algebra expressions. These elimination rules can easily be extended to also cover the semi-join algebra operators.

Finally, we look at ways to rewrite common usages of coprojections. In Section 8.3, we observed that $e_1 \circ \pi_1[e_2] \equiv_{\text{path}} e_1 \ltimes \pi_1[e_2] \equiv_{\text{path}} e_1 \ltimes e_2$. We use anti-semi-joins to generalize these rewritings to also cover coprojections:

Proposition 9.10. *Let e and e' be expressions. We have*

(i) $e \circ \overline{\pi}_1[e'] \equiv_{\text{path}} e \bar{\ltimes} e'$ *and* $e \circ \overline{\pi}_2[e'] \equiv_{\text{path}} e \bar{\ltimes} \pi_2[e']$;

(ii) $\overline{\pi}_1[e'] \circ e \equiv_{\text{path}} \pi_1[e'] \bar{\rtimes} e$ *and* $\overline{\pi}_2[e'] \circ e \equiv_{\text{path}} e' \bar{\rtimes} e$.

If, additionally, e is a node expression, then also

(iii) $e \cap \overline{\pi}_1[e'] \equiv_{\text{path}} e \bar{\ltimes} e'$ *and* $e \cap \overline{\pi}_2[e'] \equiv_{\text{path}} e \bar{\ltimes} \pi_2[e']$;

(iv) $e - \overline{\pi}_1[e'] \equiv_{\text{path}} e \ltimes e'$ *and* $e - \overline{\pi}_2[e'] \equiv_{\text{path}} e \ltimes \pi_2[e']$.

9.6 Complexity of rewritten expressions

We will consider the complexity of evaluating expressions e and the complexity of evaluating the rewritten expression $\tau(e)$ (using the rewrite rules of Section 9.4). To do so, we use the usual query evaluation complexity framework [95]. The worst-case complexity of evaluating any expression e' is $O(\text{eval-steps}(e') \cdot c)$, where $\text{eval-steps}(e')$ is the number of evaluation steps and c is the maximum cost for performing a single evaluation step. Hence, the *query complexity*—the cost of evaluating an expression in terms of the size of the expression given a fixed graph—is $O(\text{eval-steps}(e))$ and the *data complexity*—the cost of evaluating a query in terms of the size of the graph given a fixed query—is $O(c)$.

We can verify that the rewrite rules $\tau(e)$, $\tau_{\pi_1}(e)$, and $\tau_{\pi_2}(e)$ reduce the data-complexity in two distinct ways. First, the number of expensive non-linear operators (composition and Kleene-star) is reduced in favor of cheaper expression-linear operators (possibly reducing c, but never increasing it). Second, by Corollary 9.7, the size of evaluation results for subexpressions is minimized whenever possible, reducing the cost of evaluating non-rewritten operators.

The cost of the reduction in the data-complexity of evaluating an expression optimized by $\tau(e)$, $\tau_{\pi_1}(e)$, or $\tau_{\pi_2}(e)$ is an increase in the query-complexity of evaluating the optimized expression. This increase in the query-complexity is caused by an increase in the evaluation size and, in the worst case, this is an exponential increase:

Example 9.3. Consider the expressions e and $e' = \tau_{\pi_1}(e)$ of Example 9.1. By Theorem 8.4, we have $e' \equiv_{\pi_1} e'$. During rewriting, the expression size did not increase, while the evaluation size did sharply increase: we have $\|e\| = \|e'\| = 7$, $\text{eval-steps}(e) = 3$, and $\text{eval-steps}(e') = 7$. As a consequence, evaluating e and e' by evaluating each of the operators involved is possible in worst-case $O(3 \cdot |\mathbf{E}(\ell)|^2)$ and $O(7 \cdot |\mathbf{E}(\ell)|)$, respectively. Hence, any increase in the query-complexity is accompanied by a sharp decrease in the data-complexity.

Even with a worst-case exponential increase in the query-complexity, Theorem 8.4 guarantees that the query complexity is linearly upper-bounded by the size of the original

query. Hence, when queries are small and the data graphs are large, which is usually the case, the increase of the query complexity is a good trade-off if the data complexity decreases significantly.

CHAPTER 10

Graph query optimization beyond semi-joins

As motivated in Chapter 1, the relation algebra is an abstract query language that represents cleanly the core of graph query languages with respect to recognizing structures in graphs. Unfortunately, this abstract nature does not make the relation algebra directly suited for real world graph querying. To illustrate this, we consider graph querying in social networks such as visualized in Figure 1.1.

For users, a core feature of social networks is to find and connect with old and new friends. To support this, several social networks will suggest new friends to users by suggesting friends-of-friends that are not already friends. These friends-of-friends can be retrieved via the query

$$\mathit{SuggestFriends} = (\mathit{FriendOf} \circ \mathit{FriendOf}) - (\mathit{FriendOf} \cup \mathrm{id}).$$

If we want to provide Alice with friend suggestions, we simply evaluate the above query and then select all nodes m such that the pair (Alice, m) is in the result of query *SuggestFriends*. This approach is far from optimal: we ran a complex query on the social network, after which we selected and used only a very small portion of the retrieved data. As social networks tend to be extremely large graphs, this approach is unacceptably expensive.

A more practical query language would allow us to *select* the node Alice within the query, which allows the query optimizer to take advantage of the selection to simplify query evaluation. To study the effects of selection on optimization of graph query evaluation, we add a simple *node-selection operator* to the relation algebra and study how this operator affects query optimization. The node-selection $\langle n \rangle$, with n a node in the graph, will return only the single pair $\{(n, n)\}$. For example, we can use the node-selection operator to retrieve friend suggestions for Alice:

$$\mathit{SuggestAliceFriends} = \langle \mathit{Alice} \rangle \circ ((\mathit{FriendOf} \circ \mathit{FriendOf}) - (\mathit{FriendOf} \cup \mathrm{id})).$$

We observe that just evaluating the query *SuggestAliceFriends* as-is will be as inefficient as the original approach. In this chapter, we shall show that the above query does provide significant opportunities for query optimization. This is not surprising, as selections play a significant role in query optimization for, e.g., traditional relational database management systems [85, 94]. For these systems, selections give rise to the frequently used push-down rewrite rules, which can simplify queries and strongly reduce the size of intermediate results during query evaluation. Indeed, in the optimizations for node-selections we consider we can recognize the application of the concepts underlying the push-down rewrite rules.

In this chapter, we highlight query rewrite techniques we believe can aid graph querying and, at the same time, highlight how existing rewrite techniques from the relational database

world apply to graph querying. In particular, we show that in the setting of the relation algebra node-selections give rise to expressions that evaluate to relations of a specific form, which we call *Cartesian relations*. We show that these expressions that evaluate to Cartesian relations can often be simplified and optimized significantly, while also giving rise to additional opportunities for applying the semi-join optimizations we studied in Chapters 8 and 9.

10.1 Organization

In Section 10.2, we introduce Cartesian relations, Cartesian expressions, and the Cartesian product operator that we shall use in optimizing Cartesian expressions. In Section 10.3, we take a look at basic rewrite rules involving composition, intersection, and difference. In Section 10.4, we show that the Cartesian expressions are closed under composition and intersection, which can be used to derive that complex expressions are Cartesian. Finally, in Section 10.5, we give an overview of techniques that can support the rewrites proposed for Cartesian expressions.

10.2 Cartesian relations and products

In the graph data model, evaluating expressions build using selections will result in relations that have a distinctive structure:

Definition 10.1. We say that a binary relation R is *Cartesian* if and only if there exist sets U and V such that $\mathrm{R} = U \times V = \{(u, v) \mid u \in U \wedge v \in V\}$. We say that an expression e is *Cartesian* on graph $\mathcal{G}$ if $[\![e]\!]_{\mathcal{G}}$ is a Cartesian relation (hence, if $[\![e]\!]_{\mathcal{G}} = [\![e]\!]_{\mathcal{G}}|_1 \times [\![e]\!]_{\mathcal{G}}|_2$). We say that an expression e is *Cartesian* if it is Cartesian on all graphs.[22]

Example 10.1. Consider the graph in Figure 1.1. The expression $\langle \mathit{Alice} \rangle$ is Cartesian, as we can choose $U = V = \{\text{Alice}\}$. Likewise, the expression $\langle \mathit{Alice} \rangle \circ [\mathit{ParentOf}]^+$ evaluates to a Cartesian relation, as we can choose $U = \{\text{Alice}\}$ and $V = \{\text{Carol}, \text{Dan}, \text{Faythe}, \text{Grace}\}$.

If we recognize that an expression is Cartesian, then we can use this to our advantage during query evaluation. To simplify reasoning on and enable manipulation of Cartesian expressions, we introduce the *Cartesian product* operator. If e_1 and e_2 are expressions, then $e_1 \mathbin{\dot{\times}} e_2$ denotes the Cartesian product. The semantics of evaluation of the Cartesian product is defined by[23]

$$[\![e_1 \mathbin{\dot{\times}} e_2]\!]_{\mathcal{G}} = [\![e_1]\!]_{\mathcal{G}}|_1 \times [\![e_2]\!]_{\mathcal{G}}|_2.$$

The Cartesian product operator is non-linear, just like composition. When compared to composition, the Cartesian product is much easier to evaluate efficiently, however.

By definition, the Cartesian product always yields a Cartesian expression. We also have $e_1 \mathbin{\dot{\times}} e_2 \equiv_{\text{path}} \pi_1[e_1] \mathbin{\dot{\times}} \pi_2[e_2]$. Hence, we have

Lemma 10.1. *Let e_1, e_2, e_1', and e_2' be expressions with $e_1 \equiv_{\pi_1} e_1'$ and $e_2 \equiv_{\pi_2} e_2'$. We have $e_1 \mathbin{\dot{\times}} e_2 \equiv_{\text{path}} e_1' \mathbin{\dot{\times}} e_2'$.*

[22] Let R be the relation represented by a relational database table $R(A, B)$. Observe that R is Cartesian if and only if the join dependency $\bowtie [\{A\}, \{B\}]$ holds, and if and only if the multivalued dependency $\emptyset \twoheadrightarrow A$ holds [1, 85].

[23] We observe that the Cartesian product is a straightforward specialization of the usual product operator for relations to the setting in which all relations are binary.

Example 10.2. Consider the expression $e = \mathit{FriendOf} \circ \langle \mathit{Alice} \rangle \circ \mathit{FriendOf}$. We can express this expression using the Cartesian product, after which we can apply Lemma 10.1:

$$\begin{aligned} e &\equiv_{\text{path}} \pi_1[\mathit{FriendOf} \circ \langle \mathit{Alice} \rangle] \dot{\times} \pi_2[\langle \mathit{Alice} \rangle \circ \mathit{FriendOf}] \\ &\equiv_{\text{path}} (\mathit{FriendOf} \ltimes \langle \mathit{Alice} \rangle) \dot{\times} (\langle \mathit{Alice} \rangle \rtimes \mathit{FriendOf}). \end{aligned}$$

Observe that, in this rewrite, we use the expressions $\mathit{FriendOf} \ltimes \langle \mathit{Alice} \rangle$ and $\langle \mathit{Alice} \rangle \rtimes \mathit{FriendOf}$ to find all nodes that participate in the first and second column of the output. The Cartesian product simply combines these two sets of nodes.

Cartesian expressions do not necessary have to be the result of evaluating some selection, also specific (parts of) relations in a graph can be Cartesian. In particular, every binary relation can be written as a union of Cartesian relations, which is a special case of *factorized representations*. Consequently, Cartesian expressions and edge relations stored as union-of-Cartesian relations can have significant benefits beyond the query rewrite optimizations we study in this work, both at the storage level and at the query evaluation level [4, 78].

10.3 Cartesian relations and rewritings

As a first step to using Cartesian expressions in optimizing graph query evaluation, we consider composition.

Proposition 10.2. *Let e be an expression that is Cartesian on graph $\mathcal{G}$ and let e_1 and e_2 be arbitrary expressions. We have*

(i) $[\![e_1 \circ e \circ e_2]\!]_{\mathcal{G}} = [\![\pi_1[e_1 \circ e] \dot{\times} \pi_2[e \circ e_2]]\!]_{\mathcal{G}}$;

(ii) $[\![e_1 \circ e]\!]_{\mathcal{G}} = [\![\pi_1[e_1 \circ e] \dot{\times} \pi_2[e]]\!]_{\mathcal{G}}$;

(iii) $[\![e \circ e_2]\!]_{\mathcal{G}} = [\![\pi_1[e] \dot{\times} \pi_2[e \circ e_2]]\!]_{\mathcal{G}}$.

Proof. We only prove Statement (i), Statements (ii) and (iii) can be derived from Statement (i) by choosing $e_1 = \text{id}$ or $e_2 = \text{id}$ and applying the equivalences $\text{id} \circ e' \equiv_{\text{path}} e' \equiv_{\text{path}} e' \circ \text{id}$.

1. $[\![e_1 \circ e \circ e_2]\!]_{\mathcal{G}} \subseteq [\![\pi_1[e_1 \circ e] \dot{\times} \pi_2[e \circ e_2]]\!]_{\mathcal{G}}$. We have $(m, n) \in [\![e_1 \circ e \circ e_2]\!]_{\mathcal{G}}$ if there exists z_1 and z_2 such that $(m, z_1) \in [\![e_1]\!]_{\mathcal{G}}$, $(z_1, z_2) \in [\![e]\!]_{\mathcal{G}}$, and $(z_2, n) \in [\![e_2]\!]_{\mathcal{G}}$. Hence, we also have $(m, z_2) \in [\![e_1 \circ e]\!]_{\mathcal{G}}$ and $(z_1, n) \in [\![e \circ e_2]\!]_{\mathcal{G}}$ and we conclude $(m, n) \in [\![\pi_1[e_1 \circ e] \dot{\times} \pi_2[e \circ e_2]]\!]_{\mathcal{G}}$.

2. $[\![e_1 \circ e \circ e_2]\!]_{\mathcal{G}} \supseteq [\![\pi_1[e_1 \circ e] \dot{\times} \pi_2[e \circ e_2]]\!]_{\mathcal{G}}$. We have $(m, n) \in [\![\pi_1[e_1 \circ e] \dot{\times} \pi_2[e \circ e_2]]\!]_{\mathcal{G}}$ if there exists y_1 and y_2 such that $(m, y_1) \in [\![e_1 \circ e]\!]_{\mathcal{G}}$ and $(y_2, n) \in [\![e \circ e_2]\!]_{\mathcal{G}}$, which is the case if there exists z_1 and z_2 such that $(m, z_1) \in [\![e_1]\!]_{\mathcal{G}}$, $(z_1, y_1) \in [\![e]\!]_{\mathcal{G}}$, $(y_2, z_2) \in [\![e]\!]_{\mathcal{G}}$, and $(z_2, n) \in [\![e_2]\!]_{\mathcal{G}}$. As $[\![e]\!]_{\mathcal{G}}$ is a Cartesian relation, $(z_1, y_1) \in [\![e]\!]_{\mathcal{G}}$ and $(y_2, z_2) \in [\![e]\!]_{\mathcal{G}}$ imply $(z_1, z_2) \in [\![e]\!]_{\mathcal{G}}$. We conclude $(m, n) \in [\![e_1 \circ e \circ e_2]\!]_{\mathcal{G}}$. □

Next, we look at Cartesian expressions as operands of intersections and differences.

Proposition 10.3. *Let e be an expression that is Cartesian on graph $\mathcal{G}$ and let e' be an arbitrary expression. We have*

(i) $[\![e \cap e']\!]_{\mathcal{G}} = [\![e' \cap e]\!]_{\mathcal{G}} = [\![\pi_1[e] \circ e' \circ \pi_2[e]]\!]_{\mathcal{G}}$;

(ii) $[\![e' - e]\!]_{\mathcal{G}} = [\![\overline{\pi}_1[e] \circ e' \cup e' \circ \overline{\pi}_2[e]]\!]_{\mathcal{G}}$.

Proof. First, we prove Statement (i). We observe that intersection is commutative, hence we have $[\![e \cap e']\!]_{\mathcal{G}} = [\![e' \cap e]\!]_{\mathcal{G}}$. We prove $[\![e \cap e']\!]_{\mathcal{G}} = [\![\pi_1[e] \circ e' \circ \pi_2[e]]\!]_{\mathcal{G}}$:

1. $[\![e \cap e']\!]_{\mathcal{G}} \subseteq [\![\pi_1[e] \circ e' \circ \pi_2[e]]\!]_{\mathcal{G}}$. If $(m,n) \in [\![e \cap e']\!]_{\mathcal{G}}$, then $(m,n) \in [\![e]\!]_{\mathcal{G}}$ and $(m,n) \in [\![e']\!]_{\mathcal{G}}$. Hence, we have $(m,m) \in [\![\pi_1[e]]\!]_{\mathcal{G}}$ and $(n,n) \in [\![\pi_2[e]]\!]_{\mathcal{G}}$. We conclude $(m,n) \in [\![\pi_1[e] \circ e' \circ \pi_2[e]]\!]_{\mathcal{G}}$.

2. $[\![e \cap e']\!]_{\mathcal{G}} \supseteq [\![\pi_1[e] \circ e' \circ \pi_2[e]]\!]_{\mathcal{G}}$. If $(m,n) \in [\![\pi_1[e] \circ e' \circ \pi_2[e]]\!]_{\mathcal{G}}$, then $(m,m) \in [\![\pi_1[e]]\!]_{\mathcal{G}}$, $(m,n) \in [\![e']\!]_{\mathcal{G}}$, and $(n,n) \in [\![\pi_2[e]]\!]_{\mathcal{G}}$. Hence, there exists z_1 and z_2 such that $(m,z_1) \in [\![e]\!]_{\mathcal{G}}$ and $(z_2,n) \in [\![e]\!]_{\mathcal{G}}$. As $[\![e]\!]_{\mathcal{G}}$ is a Cartesian relation, $(m,z_1) \in [\![e]\!]_{\mathcal{G}}$ and $(z_2,n) \in [\![e]\!]_{\mathcal{G}}$ imply $(m,n) \in [\![e]\!]_{\mathcal{G}}$. We conclude $(m,n) \in [\![e \cap e']\!]_{\mathcal{G}}$.

Next, we prove Statement (ii).

1. $[\![e' - e]\!]_{\mathcal{G}} \subseteq [\![\overline{\pi}_1[e] \circ e' \cup e' \circ \overline{\pi}_2[e]]\!]_{\mathcal{G}}$. If $(m,n) \in [\![e' - e]\!]_{\mathcal{G}}$, then $(m,n) \in [\![e']\!]_{\mathcal{G}}$ and $(m,n) \notin [\![e]\!]_{\mathcal{G}}$. As $[\![e]\!]_{\mathcal{G}}$ is a Cartesian relation, $(m,n) \notin [\![e]\!]_{\mathcal{G}}$ implies either $\forall z\ (m,z) \notin [\![e]\!]_{\mathcal{G}}$ or $\forall z\ (z,n) \notin [\![e]\!]_{\mathcal{G}}$. Hence, we have $(m,m) \in [\![\overline{\pi}_1[e]]\!]_{\mathcal{G}}$ and $(m,n) \in [\![\overline{\pi}_1[e] \circ e']\!]_{\mathcal{G}}$ or $(n,n) \in [\![\overline{\pi}_2[e]]\!]_{\mathcal{G}}$ and $(m,n) \in [\![e' \circ \overline{\pi}_2[e]]\!]_{\mathcal{G}}$. We conclude $(m,n) \in [\![\overline{\pi}_1[e] \circ e' \cup e' \circ \overline{\pi}_2[e]]\!]_{\mathcal{G}}$.

2. $[\![e' - e]\!]_{\mathcal{G}} \supseteq [\![\overline{\pi}_1[e] \circ e' \cup e' \circ \overline{\pi}_2[e]]\!]_{\mathcal{G}}$. If $(m,n) \in [\![\overline{\pi}_1[e] \circ e' \cup e' \circ \overline{\pi}_2[e]]\!]_{\mathcal{G}}$, then $(m,m) \in [\![\overline{\pi}_1[e]]\!]_{\mathcal{G}}$ and $(m,n) \in [\![e']\!]_{\mathcal{G}}$, or $(m,n) \in [\![e']\!]_{\mathcal{G}}$ and $(n,n) \in [\![\overline{\pi}_2[e]]\!]_{\mathcal{G}}$. By $(m,m) \in [\![\overline{\pi}_1[e]]\!]_{\mathcal{G}}$ or $(n,n) \in [\![\overline{\pi}_2[e]]\!]_{\mathcal{G}}$, we have $\forall z\ (m,z) \notin [\![e]\!]_{\mathcal{G}}$ or $\forall z\ (z,n) \notin [\![e]\!]_{\mathcal{G}}$. In both cases, we have $(m,n) \notin [\![e]\!]_{\mathcal{G}}$. We conclude $(m,n) \in [\![e' - e]\!]_{\mathcal{G}}$. □

We did not consider the case $e - e'$ with e Cartesian. In this case, no general rewriting exists that eliminates the difference operator, as is shown by the following example.

Example 10.3. Observe that the expression $\mathrm{all} = \mathrm{id} \cup \mathrm{di}$ is Cartesian. Let e be an arbitrary expression. The expression $\mathrm{all} - e$ evaluates to the complement of e (with respect to the set of all nodes). In the relation algebra, the complement is not expressible without difference. Hence, in this case we cannot use the fact that "all" is Cartesian to eliminate the need for the difference operator.

The rewrite rules of Proposition 10.2 and Proposition 10.3 do not necessary lead to optimizations; no operations are removed and additional operations are added. Proposition 10.2 does, however, push compositions into projections. Likewise, Proposition 10.3 eliminates intersection and difference operators. Hence, these rewrite rules introduce opportunities for the application of the semi-join rewrite rules:

Example 10.4. Consider the expression

$$e = \langle \mathit{Alice} \rangle \circ (\mathit{FriendOf} \circ \mathit{FriendOf} \circ \mathit{FriendOf}).$$

This expression returns pairs (Alice, m) with m a friend-of-friend-of-friend of Alice. The expression $\langle \mathit{Alice} \rangle$ is Cartesian. Hence, we apply Proposition 10.2 (iii) to eliminate the composition, after which we apply Lemma 10.1 and the semi-join rewrites on the resultant projection-term, yielding:

$$\begin{aligned} e &\equiv_{\mathrm{path}} \langle \mathit{Alice} \rangle \circ (\mathit{FriendOf} \circ \mathit{FriendOf} \circ \mathit{FriendOf}) \\ &\equiv_{\mathrm{path}} \pi_1[\langle \mathit{Alice} \rangle] \mathbin{\dot{\times}} \pi_2[\langle \mathit{Alice} \rangle \circ (\mathit{FriendOf} \circ \mathit{FriendOf} \circ \mathit{FriendOf})] \\ &\equiv_{\mathrm{path}} \langle \mathit{Alice} \rangle \mathbin{\dot{\times}} \tau(\pi_2[\langle \mathit{Alice} \rangle \circ (\mathit{FriendOf} \circ \mathit{FriendOf} \circ \mathit{FriendOf})]) \\ &= \langle \mathit{Alice} \rangle \mathbin{\dot{\times}} \pi_2[((\langle \mathit{Alice} \rangle \rtimes \mathit{FriendOf}) \rtimes \mathit{FriendOf}) \rtimes \mathit{FriendOf}]. \end{aligned}$$

The above rewrite manages to eliminate all compositions in e in favor of semi-joins and a single usage of the Cartesian product. We observe that, in this case, the Cartesian product will yield exactly the query result. Hence, due to its simplicity, its usage can be considered optimal.

In Proposition 10.2 and Proposition 10.3, we looked at binary operators with a single Cartesian operand. Next, consider binary operators with two Cartesian operands. If both operands of a composition are Cartesian, then this allows us to choose between applying Proposition 10.2 (ii) or Proposition 10.2 (iii). For difference, we cannot take advantage of the extra Cartesian operand, and we simply apply Proposition 10.3 (ii). For intersections of two Cartesian expressions, we have:

Proposition 10.4. *Let e_1 and e_2 be expressions that are Cartesian on graph $\mathcal{G}$. We have $[\![e_1 \cap e_2]\!]_{\mathcal{G}} = [\![e_2 \cap e_1]\!]_{\mathcal{G}} = [\![\pi_1[e_1] \circ \pi_1[e_2] \dot{\times} \pi_2[e_1] \circ \pi_2[e_2]]\!]_{\mathcal{G}}$.*

Finally, we look at simplifying Kleene-star and Kleene-plus operators applied on Cartesian operands:

Proposition 10.5. *Let e be an expression that is Cartesian on graph $\mathcal{G}$. We have $[\![[e]^*]\!]_{\mathcal{G}} = [\![e \cup \mathrm{id}]\!]_{\mathcal{G}}$ and $[\![[e]^+]\!]_{\mathcal{G}} = [\![e]\!]_{\mathcal{G}}$.*

10.4 Closure results for Cartesian relations

Up till now, we have only seen basic examples of Cartesian expressions. Next, we show that new Cartesian expressions can be build using such basic Cartesian expressions. First, we consider compositions with Cartesian operands:

Proposition 10.6. *Let e be an expression that is Cartesian on graph $\mathcal{G}$ and let e' be an arbitrary expressions. The expressions $e \circ e'$ and $e' \circ e$ are Cartesian on $\mathcal{G}$.*

A similar closure result does not hold for intersections, however.

Example 10.5. Consider the expression $\mathrm{all} \cap e$. This expression has at least a single Cartesian operand (all), but the expression is Cartesian only if e is Cartesian.

For intersections, a weaker result does hold:

Proposition 10.7. *Let e_1 and e_2 be expressions that are Cartesian on graph $\mathcal{G}$. The expression $e_1 \cap e_2$ is Cartesian on $\mathcal{G}$.*

The above results can be used to increase the scope of Cartesian-based rewrites.

Example 10.6. Consider the expression

$$e = \pi_1[(\mathit{WorksWith} \circ \langle \mathit{Alice} \rangle \circ \mathit{WorksWith}) \cap (\mathit{FriendOf} \circ \mathit{MarriedWith})].$$

This expression returns pairs (m, n) of people that both work with Alice and such that m is a friend of the partner of n. For brevity, we write $e = \pi_1[e_1 \cap e_2]$. The expression e is in $\mathcal{N}$ and, due to the usage of intersection and composition, the compositions cannot be eliminated by the semi-join rewrite rules. The expression $\langle \mathit{Alice} \rangle$ is Cartesian. Hence, we

apply Proposition 10.6 to conclude that also e_1 is Cartesian. Next, we use Proposition 10.3 (i) to remove the intersection, after which we apply the semi-join rewrites, yielding

$$\begin{aligned} e &\equiv_{\text{path}} \pi_1[e_1 \cap e_2] \\ &\equiv_{\text{path}} \pi_1[\pi_1[e_1] \circ e_2 \circ \pi_2[e_1]] \\ &\equiv_{\text{path}} \tau(\pi_1[\pi_1[e_1] \circ e_2 \circ \pi_2[e_1]]) \\ &= \pi_1[e_{1,1} \ltimes (\mathit{FriendOf} \ltimes (\mathit{MarriedWith} \ltimes e_{1,2}))], \end{aligned}$$

with

$$\begin{aligned} e_{1,1} &= \pi_1[\mathit{WorksWith} \ltimes (\langle \mathit{Alice} \rangle \ltimes \mathit{WorksWith})]; \\ e_{1,2} &= \pi_2[(\mathit{WorksWith} \rtimes \langle \mathit{Alice} \rangle) \rtimes \mathit{WorksWith}]. \end{aligned}$$

We observe that this rewrite manages to eliminate the intersection and all compositions in favor of only semi-joins.

We have shown that compositions and intersections can be used to build Cartesian expressions. Using Proposition 10.5, we can also directly conclude that $[e]^+$ is Cartesian on graph $\mathcal{G}$ whenever e is Cartesian on $\mathcal{G}$. Next, we show that no such closure results exist for the cases we did not yet consider:

Example 10.7. We consider projections, coprojections, unions, differences, and Kleene-stars. Let e_1 and e_2 be expressions with

$$\begin{aligned} [\![e_1]\!]_{\mathcal{G}} &= \{(\alpha, u), (\alpha, v), (\beta, u), (\beta, v)\}; \\ [\![e_2]\!]_{\mathcal{G}} &= \{(\alpha, u), (\gamma, u)\} \end{aligned}$$

on some graph $\mathcal{G}$. Both e_1 and e_2 are Cartesian on $\mathcal{G}$. Next, consider:

$$\begin{aligned} [\![\pi_1[e_1]]\!]_{\mathcal{G}} &= \{(\alpha, \alpha), (\beta, \beta)\}; \\ [\![\overline{\pi}_1[e_1]]\!]_{\mathcal{G}} &= \{(\gamma, \gamma), (u, u), (v, v)\}; \\ [\![e_1 \cup e_2]\!]_{\mathcal{G}} &= \{(\alpha, u), (\alpha, v), (\beta, u), (\beta, v), (\gamma, u)\}; \\ [\![e_1 - e_2]\!]_{\mathcal{G}} &= \{(\alpha, v), (\beta, u), (\beta, v)\}; \\ [\![[e_2]^*]\!]_{\mathcal{G}} &= \{(\alpha, \alpha), (\alpha, u), (\beta, \beta), (\gamma, \gamma), (\gamma, u), (u, u), (v, v)\}. \end{aligned}$$

By inspection on the above results, one can verify that the expressions $\pi_1[e_1]$, $\overline{\pi}_1[e_1]$, $e_1 \cup e_2$, $e_1 - e_2$, and $[e_2]^*$ are not Cartesian on $\mathcal{G}$.

10.5 Other practical query optimizations

In the previous sections, we have seen that Cartesian expressions can be used to rewrite compositions, intersections, and differences, and in several cases these rewrites clearly lead to improvements. Unfortunately, the presented rewrites on their own are not strong enough to optimize all queries that can benefit from Cartesian rewrites, semi-join rewrites, or both. First, the rewrites might not recognize cases where Cartesian expressions can be introduced and, hence, miss cases where Cartesian rewrites are possible. Second, rewrites can also make queries unnecessary complicated, especially when intersections or differences are involved. We illustrate both issues with a simple example:

Example 10.8. Consider the expression

$$e = \langle Alice \rangle \circ ((FriendOf \circ FriendOf) \cap FriendOf).$$

The expressions $FriendOf \circ FriendOf$ and $FriendOf$ are both not Cartesian. The node-selection $\langle Alice \rangle$ can be pushed inside the intersection, however, resulting in an intersection of two Cartesian expressions, after which we can apply Proposition 10.4, and the semi-join rewrites:

$$\begin{aligned}
e &\equiv_{\text{path}} (\langle Alice \rangle \circ FriendOf \circ FriendOf) \cap (\langle Alice \rangle \circ FriendOf) \\
&\equiv_{\text{path}} \pi_1[\langle Alice \rangle \circ FriendOf \circ FriendOf] \circ \pi_1[\langle Alice \rangle \circ FriendOf] \mathbin{\dot{\times}} \\
&\qquad \pi_2[\langle Alice \rangle \circ FriendOf \circ FriendOf] \circ \pi_2[\langle Alice \rangle \circ FriendOf] \\
&\equiv_{\text{path}} \tau(\pi_1[\langle Alice \rangle \circ FriendOf \circ FriendOf] \circ \pi_1[\langle Alice \rangle \circ FriendOf]) \mathbin{\dot{\times}} \\
&\qquad \tau(\pi_2[\langle Alice \rangle \circ FriendOf \circ FriendOf] \circ \pi_2[\langle Alice \rangle \circ FriendOf]) \\
&= \pi_1[\langle Alice \rangle \ltimes (FriendOf \ltimes FriendOf)] \ltimes \pi_1[\langle Alice \rangle \ltimes FriendOf] \mathbin{\dot{\times}} \\
&\qquad \pi_2[(\langle Alice \rangle \rtimes FriendOf) \rtimes FriendOf] \circ \pi_2[\langle Alice \rangle \rtimes FriendOf].
\end{aligned}$$

We observe that the left-hand side of the Cartesian product will evaluate to {(Alice, Alice)} if and only if the right-hand side does not evaluate to the empty-set. Hence, the above query can significantly be simplified to

$$\langle Alice \rangle \mathbin{\dot{\times}} \pi_2[(\langle Alice \rangle \rtimes FriendOf) \rtimes FriendOf] \circ \pi_2[\langle Alice \rangle \rtimes FriendOf].$$

The final rewrite used in Example 10.8 is a special case of Proposition 10.4, which we formalize next.

Proposition 10.8. *Let e_1 and e_2 be expressions that are Cartesian on graph $\mathcal{G}$ and let $e_1 = e \circ e'$. We have:*

(i) if $[\![e]\!]_{\mathcal{G}}|_1 \leq 1$, then $[\![e_1 \cap e_2]\!]_{\mathcal{G}} = [\![e_2 \cap e_1]\!]_{\mathcal{G}} = e \mathbin{\dot{\times}} \pi_2[e_1] \circ \pi_2[e_2]$;

(ii) if $[\![e']\!]_{\mathcal{G}}|_2 \leq 1$, then $[\![e_1 \cap e_2]\!]_{\mathcal{G}} = [\![e_2 \cap e_1]\!]_{\mathcal{G}} = \pi_1[e_1] \circ \pi_1[e_2] \mathbin{\dot{\times}} e'$.

The main technique used in Example 10.8 is the introduction of Cartesian expressions as operands for the intersection operator. We did so by pushing a node-selection step downwards, which is possible due to node-selections not only being Cartesian expressions, but also node expressions. This technique is very similar to the well-known push-down rules for selection in relational database management systems [85, 94]. For node expressions, we have the following push-down rules:

Proposition 10.9. *Let e be a node expressions, let e', e_1, and e_2 be arbitrary expressions, and let $j, j' \in \{1, 2\}$. We have:*

$$\begin{aligned}
e \circ e' &\equiv_{\text{path}} e \circ e \circ e'; \\
e \circ \pi_1[e'] &\equiv_{\text{path}} \pi_1[e \circ e']; \\
e \circ \pi_2[e'] &\equiv_{\text{path}} \pi_2[e' \circ e]; \\
e \circ \overline{\pi}_{j'}[e'] &\equiv_{\text{path}} \overline{\pi}_{j'}[\overline{\pi}_j[e] \cup e']; \\
e \cup \overline{\pi}_{j'}[e'] &\equiv_{\text{path}} \overline{\pi}_{j'}[\overline{\pi}_j[e] \circ e']; \\
e \circ (e_1 \cup e_2) &\equiv_{\text{path}} (e \circ e_1) \cup (e \circ e_2);
\end{aligned}$$

$$(e_1 \cup e_2) \circ e \equiv_{\text{path}} (e_1 \circ e) \cup (e_2 \circ e);$$
$$e \circ (e_1 \cap e_2) \equiv_{\text{path}} (e \circ e_1) \cap e_2 \equiv_{\text{path}} e_1 \cap (e \circ e_2);$$
$$(e_1 \cap e_2) \circ e \equiv_{\text{path}} (e_1 \circ e) \cap e_2 \equiv_{\text{path}} e_1 \cap (e_2 \circ e);$$
$$e \circ (e_1 - e_2) \equiv_{\text{path}} (e \circ e_1) - e_2 \equiv_{\text{path}} (e \circ e_1) - (e \circ e_2);$$
$$(e_1 - e_2) \circ e \equiv_{\text{path}} (e_1 \circ e) - e_2 \equiv_{\text{path}} (e_1 \circ e) - (e_2 \circ e).$$

To further reduce the complexity of rewritten expressions, we can attempt to eliminate subexpressions. Observe that Lemma 3.2 (ii) and 3.2 (iii) already provide basic rewrite rules for the elimination of superfluous usage of $\emptyset$ and id and of subexpressions that contain $\emptyset$. Using the semantics of the relation algebra operators, we can obtain many other cases in which subexpressions can be eliminated.

Proposition 10.10. *Let $\mathcal{G}$ be a graph, let e_1 and e_2 be expressions, let $j_1, j_2 \in \{1, 2\}$, and let $\otimes \in \{\circ, \ltimes, \rtimes\}$.*

(i) If $[\![e_1]\!]_{\mathcal{G}}|_{j_1} \subseteq [\![e_2]\!]_{\mathcal{G}}|_{j_2}$, then

$$[\![\pi_{j_1}[e_1] \otimes \pi_{j_2}[e_2]]\!]_{\mathcal{G}} = [\![\pi_{j_1}[e_1]]\!]_{\mathcal{G}}, \qquad [\![\pi_{j_1}[e_1] \otimes \overline{\pi}_{j_2}[e_2]]\!]_{\mathcal{G}} = \emptyset;$$
$$[\![\pi_{j_1}[e_1] \cap \pi_{j_2}[e_2]]\!]_{\mathcal{G}} = [\![\pi_{j_1}[e_1]]\!]_{\mathcal{G}}, \qquad [\![\pi_{j_1}[e_1] \cap \overline{\pi}_{j_2}[e_2]]\!]_{\mathcal{G}} = \emptyset;$$
$$[\![\pi_{j_1}[e_1] - \overline{\pi}_{j_2}[e_2]]\!]_{\mathcal{G}} = [\![\pi_{j_1}[e_1]]\!]_{\mathcal{G}}, \qquad [\![\pi_{j_1}[e_1] - \pi_{j_2}[e_2]]\!]_{\mathcal{G}} = \emptyset.$$

(ii) If $[\![e_1]\!]_{\mathcal{G}}|_2 \cap [\![e_2]\!]_{\mathcal{G}}|_1 = \emptyset$, then $[\![e_1 \otimes e_2]\!]_{\mathcal{G}} = \emptyset$.

(iii) If $[\![e_1]\!]_{\mathcal{G}} \cap [\![e_2]\!]_{\mathcal{G}} = \emptyset$, then $[\![e_1 \cap e_2]\!]_{\mathcal{G}} = \emptyset$ and $[\![e_1 - e_2]\!]_{\mathcal{G}} = [\![e_1]\!]_{\mathcal{G}}$.

(iv) If $[\![e_1]\!]_{\mathcal{G}} \subseteq [\![e_2]\!]_{\mathcal{G}} = \emptyset$, then $[\![e_1 \cap e_2]\!]_{\mathcal{G}} = [\![e_1]\!]_{\mathcal{G}}$ and $[\![e_1 - e_2]\!]_{\mathcal{G}} = \emptyset$.

The conditions of Proposition 10.10 are not always straightforward to detect. Observe that $e_1 \equiv_{\pi_j} e_2$, $j \in \{1, 2\}$, implies, for every graph $\mathcal{G}$, $[\![e_1]\!]_{\mathcal{G}}|_j = [\![e_2]\!]_{\mathcal{G}}|_j$, $[\![e_1]\!]_{\mathcal{G}}|_j \subseteq [\![e_2 \cup e']\!]_{\mathcal{G}}|_j$, $[\![e_1 \cap e']\!]_{\mathcal{G}}|_j \subseteq [\![e_2]\!]_{\mathcal{G}}|_j$, and $[\![e_1 - e']\!]_{\mathcal{G}}|_j \subseteq [\![e_2]\!]_{\mathcal{G}}|_j$. Likewise, $e_1 \equiv_{\text{path}} e_2$ implies, for every graph $\mathcal{G}$, $[\![e_1]\!]_{\mathcal{G}} = [\![e_2]\!]_{\mathcal{G}}$, $[\![e_1]\!]_{\mathcal{G}} \subseteq [\![e_2 \cup e']\!]_{\mathcal{G}}$, $[\![e_1 \cap e]\!]_{\mathcal{G}} \subseteq [\![e_2]\!]_{\mathcal{G}}$, and $[\![e_1 - e]\!]_{\mathcal{G}} \subseteq [\![e_2]\!]_{\mathcal{G}}$. Finally, $[\![e_1 \circ e_2]\!]_{\mathcal{G}}|_1 = [\![e_1 \ltimes e_2]\!]_{\mathcal{G}}|_1 \subseteq [\![e_1]\!]_{\mathcal{G}}|_1$ and $[\![e_1 \circ e_2]\!]_{\mathcal{G}}|_2 = [\![e_1 \rtimes e_2]\!]_{\mathcal{G}}|_2 \subseteq [\![e_2]\!]_{\mathcal{G}}|_2$.

To conclude, we show how the techniques discussed in this chapter can be used to rewrite the query *SuggestAliceFriends* to a query that is much more efficient to evaluate:

Example 10.9. Consider the expression

$$e = \langle Alice \rangle \circ ((FriendOf \circ FriendOf) - (FriendOf \cup \text{id})).$$

We observe that $\langle Alice \rangle$ is a node expression. We use Proposition 10.9 to push $\langle Alice \rangle$ through the difference operator. We obtain

$$e \equiv_{\text{path}} ((\langle Alice \rangle \circ FriendOf \circ FriendOf) - (\langle Alice \rangle \circ (FriendOf \cup \text{id}))).$$

We use Proposition 10.6 on $\langle Alice \rangle$ to conclude that $\langle Alice \rangle \circ (FriendOf \cup \text{id})$ is a Cartesian expression. We apply Proposition 10.3 (ii) to obtain

$$e \equiv_{\text{path}} (\overline{\pi}_1[\langle Alice \rangle \circ (FriendOf \cup \text{id})] \circ \langle Alice \rangle \circ FriendOf \circ FriendOf) \cup$$
$$(\langle Alice \rangle \circ FriendOf \circ FriendOf \circ \overline{\pi}_2[\langle Alice \rangle \circ (FriendOf \cup \text{id})]).$$

By Proposition 10.9 and Lemma 3.2 (iii), we have $\langle Alice \rangle \circ (FriendOf \cup \mathrm{id}) \equiv_{\mathrm{path}} (\langle Alice \rangle \circ FriendOf) \cup \langle Alice \rangle$. Hence, we have $[\![\langle Alice \rangle \circ (FriendOf \cup \mathrm{id})]\!]_{\mathcal{G}}|_1 \supseteq [\![\langle Alice \rangle]\!]_{\mathcal{G}}|_1$. We apply Proposition 10.10 to conclude that $\overline{\pi}_1[\langle Alice \rangle \circ (FriendOf \cup \mathrm{id})] \circ \langle Alice \rangle \equiv_{\mathrm{path}} \emptyset$. We apply Lemma 3.2 (ii) to obtain

$$e \equiv_{\mathrm{path}} \langle Alice \rangle \circ FriendOf \circ FriendOf \circ \overline{\pi}_2[(\langle Alice \rangle \circ FriendOf) \cup \langle Alice \rangle].$$

On $\langle Alice \rangle$, we apply Proposition 10.2 (iii) to obtain

$$e \equiv_{\mathrm{path}} \pi_1[\langle Alice \rangle] \mathbin{\dot{\times}} \pi_2[\langle Alice \rangle \circ FriendOf \circ FriendOf \circ \overline{\pi}_2[(\langle Alice \rangle \circ FriendOf) \cup \langle Alice \rangle]].$$

Finally, we simplify $\pi_1[\langle Alice \rangle]$ and apply the semi-join rewritings and Proposition 9.10 (i) on the right-hand side of the Cartesian product. We obtain

$$e \equiv_{\mathrm{path}} \langle Alice \rangle \mathbin{\dot{\times}} \pi_2[(\langle Alice \rangle \rtimes FriendOf) \rtimes (FriendOf \mathbin{\bar{\ltimes}} \pi_2[(\langle Alice \rangle \rtimes FriendOf) \cup \langle Alice \rangle])].$$

We observe that in the resulting expression, all usages of the identity, composition, and difference operators have been eliminated in favor of only semi-joins, anti-semi-joins, and a single simple application of the Cartesian product.

CHAPTER 11

Conclusion, discussion, and future work

In Part III, we set out to improve our understanding of the relationship between the relation algebra and the semi-join algebra, this with applications to graph query optimization in mind. Indeed, we have shown that, in many cases, costly composition and Kleene-star operators can be rewritten into the less costly semi-join and fixpoint operators. We have also identified sufficient conditions on relation algebra expressions that allow us to perform these rewrites automatically. In addition, we have shown that our rewrite rules can significantly lower the data complexity—the complexity in terms of the size of the graph—of query evaluation, while never increasing it. Finally, we briefly looked at optimizations of other expensive operators and at optimization opportunities arising from the introduction of operators necessary for practical use cases.

We believe that our work can improve the state of the art in efficient graph query evaluation. Our work does not provide a full approach for efficient query evaluation for the relation algebra, however. Hence, the design and implementation of graph database systems and graph query engines deserves further attention in future research.

As a first step in the direction of future graph database systems, one can take a look at the design and implementation of existing graph database systems and at the vast body of work on relational database management systems. To see what is necessary in this regard, we will take a brief look in this chapter at established relational database management systems, and identify challenges relevant to graph database systems. We believe that addressing the challenges of graph query optimization not only benefits graph database systems, but also relational database management systems, as further advancements in graph query evaluation can pave the way for further advancements in relational query evaluation. In this regards, a worthwhile avenue for future research is investigating if and how the semi-join optimizations presented in this part can be translated to the richer setting of SQL queries (with multiset semantics and aggregations).

11.1 Organization

In Section 11.2, we argue why traditional relational database management systems and graph database systems are related, and we argue what the main differences are. In Section 11.3, we take a look at the shortcomings of state-of-the-art database systems with respect to graph querying, establishing that state-of-the-art database systems struggle to execute even simple navigation-based graph queries. In Section 11.4, we take a look at the operations of traditional query engines, and identify directions that need further exploration when applied to the setting of graph querying. Finally, in Section 11.5, we look at the most important aspect

of efficient query evaluation, evaluating joins, and discuss what parts of join evaluation need additional attention in the setting of graph query evaluation.

11.2 Relational data and graph data

It is easy to see parallels between relational database management systems and graph database systems, be it the formal edge-labeled graphs we use in this work (see Definition 1.1) or richer graph data models such as the *property graph model* [82].

The relational data model boils down to a set of arbitrary named relations. The (edge-labeled) graph data model of Definition 1.1 that we used in this work boils down to a set of many-to-many named binary relations (the edges), as shown in Table 1.1. In richer data models such as the property graph model, these many-to-many named binary relations are combined with a node-attribute-value relation. Hence, in essence, both the relational data model and the graph data model use relations as the main representation of data. This similarity extends to the query language level. Relational database management systems typically use SQL, whose core is formalized by Codd's relational algebra [23, 40, 59]. The relation algebra we studied can be seen as a binary-relation-only specialization of Codd's relational algebra. Likewise, the semi-join algebra we introduced can be seen as a binary-relation-only specialization of the standard semi-join relational algebra [66]. Even the Kleene-star and fixpoint operators we considered have parallels in modern relational database managements system via '`WITH RECURSIVE`' SQL queries [59]. As such, it is not surprising that all example queries considered in this work can be expressed in SQL in a rather straightforward way.

As just argued, we can identify many similarities between the relational data model and the graph data model, especially at the data model level (the data they can or cannot model) and at the query level (the queries they can or cannot express). This does not imply that these data models are similar, however. We believe that the main distinction between these data models are found in their practical usage: the type of relationships relational database management systems and graph database systems are expected to operate on. On the one hand, typical data in relational database management systems is normalized and structured around relations representing *entities* and *relationships* between these entities [85]. Normalizing relational data usually results in many *foreign key constraints* that each imply important *many-to-one* or *one-to-one* relationships relating tuples from one relation to a single tuple from another relation, while only few relationships are *many-to-many*. On the other hand, typical graph data is centered around important *many-to-many* relationships [5, 6, 82], as is already illustrated by the edge relationships *FriendOf*, *ParentOf*, and *WorksWith* of Chapter 1.

To further illustrate this difference between, on the one hand, a focus on one-to-one and many-to-one relationships and, on the other hand, many-to-many relationships, we take a look at the TPC-H benchmark [26]. In this standardized relational database management system benchmark, we have the relations *Part*, *Supplier*, *Customer*, *Nation*, *Region*, and *Order* representing entities. Between these entities, there are several many-to-one foreign key relationships, e.g., suppliers and customers are both located in a single nation, each nation belongs to a single region, and each order is placed by a single customer. Only the relations *Partsupp*—relating parts to suppliers that sell them—and *Lineitem*—relating orders to individual parts and suppliers—represent *many-to-many* relationships. This affects the complexity of typical joins. Indeed, a quick look at the queries in the TPC-H benchmark shows that only two out of 22 queries (query Q17 and Q21) require many-to-many joins in which

each tuples of either relation can be joined with several tuples of the other relation.[24] Contrast this with even the simplest graph queries seen in this work such as *FriendOf* ∘ *FriendOf* and *ParentOf* ∘ *ParentOf* ∘ *ParentOf*, in which many-to-many joins are at the very core of the query.

Due to the differences in the relationships relational database management systems are expected to operate on and the type of relationships that are central in graph data, one can expect that relational database management systems have troubles evaluating certain graph queries efficiently. In the next section, we shall briefly look at this expectation and conform that this is, indeed, the case.

11.3 Graph queries in modern database systems

In the previous section we observed that relational database management systems and graph database systems are strongly related, but might have a different focus in practice, especially with regards to many-to-many joins. To illustrate this observed difference in focus in practice, we present a few performance measurements on executing graph queries expressed in SQL. As the first example, we consider the basic expression

$$q = \pi_1[\mathit{Edges} \circ \mathit{Edges} \circ \mathit{Edges} \circ \mathit{Edges}],$$

which yields nodes that have outgoing paths of length at-least four. The SQL query

```
SELECT DISTINCT R.nfrom
FROM edges R, edges S, edges T, edges U
WHERE R.nto = S.nfrom AND
      S.nto = T.nfrom AND
      T.nto = U.nfrom;
```

expresses exactly the same query. We have executed the above SQL query on three modern relational database management systems, namely PostgreSQL 9.6.1, Microsoft SQL Server 2016 (SP1) 13.0.4001.0, and Oracle Database 12c Release 12.1.0.2.0. We constructed a single *Edges* table with $n = 1\,000$ nodes and $e = 75\,000$ randomly generated edges connecting these nodes. Our measurements can be found in the columns labeled *(original)* in Table 11.1.

We directly conclude that the above SQL query is evaluated impractically slow in modern relational database management systems. Fortunately, it is well-known that manually applying semi-join rewritings of natural joins into '`WHERE IN`' or '`WHERE EXISTS`' clauses (see, e.g., [77]) can improve query evaluation performance drastically. As an illustration, we have rewritten the above query to

```
SELECT DISTINCT nfrom FROM edges
WHERE nto IN (
      SELECT nfrom FROM edges
      WHERE nto IN (
            SELECT nfrom FROM edges
            WHERE nto IN (
                  SELECT nfrom FROM edges)));
```

[24] Queries Q1 and Q6 do not involve joins at all. The queries Q9, Q17, Q20, and Q21 involve multiple many-to-many relations. Of these four queries, queries Q9 and Q20 join the many-to-many relations *Lineitem* and *Partsupp* on a many-to-one foreign key relationship between *Lineitem* and *Partsupp*, effectively making these joins many-to-one joins. The remaining 16 queries all only involve one-to-one and many-to-one joins.

Table 11.1: Performance measurements on executing the SQL-version of query q and the manually optimized semi-join SQL version of query q. The columns labeled *t (ms)* shows the time it takes to execute the specified query (executing the query and retrieving all rows in the result, queries where aborted after 20 minutes). The columns labeled #⋈/⋉ show the number of joins in the underlying query plan, and how many of these joins are performed using semi-join-like algorithms.

	(original)		(rewritten)	
Relational Database Management System	t (ms)	#⋈/⋉	t (ms)	#⋈/⋉
PostgreSQL 9.6.1	∞	3/0	376	3/3
SQL Server 2016 (SP1) 13.0.4001.0	312 783	3/1	278	3/3
Oracle Database 12c Release 12.1.0.2.0	264 373	3/1	269 815	3/1

This rewritten query yields the same result and most tested relational database management systems are able to evaluate this rewritten query in less than a second, as shown in Table 11.1, columns labeled *(rewritten)*. To see if a dedicated modern graph database system would perform significantly better, we have also implemented the query q in Neo4j using the Cypher query language [82]. The query q is equivalent to the Cypher query

```
MATCH (a)-[:edge]->(b)-[:edge]-(c)-[:edge]->(d)-[:edge]->(e)
RETURN DISTINCT (a)
```

We ran this query in Neo4j 3.3.2 and, as with PostgreSQL, this query did not finish executing in any reasonable amount of time.

Also with respect to graph-like aggregation queries, our initial experience shows that there is still significant room for improvement. Consider the following SQL query, based on query q of the previous section, that yields, per node, the number of distinct outgoing paths of length three.

```
SELECT R.nfrom, COUNT(*) AS n
FROM edges R, edges S, edges T
WHERE R.nto = S.nfrom AND S.nto = T.nfrom
GROUP BY R.nfrom;
```

Alternatively, this query can be rewritten in a more semi-join optimized fashion as follows:

```
SELECT R.nfrom, SUM(SS.n) AS n
FROM edges R,
     (SELECT S.nfrom, SUM(TT.n) AS n
      FROM edges S,
           (SELECT T.nfrom, COUNT(*) AS n
            FROM edges T
            GROUP BY T.nfrom) TT
      WHERE S.nto = TT.nfrom
      GROUP BY S.nfrom) SS
WHERE R.nto = SS.nfrom
GROUP BY R.nfrom;
```

Table 11.2: Performance measurements on executing an SQL implementation of the aggregated version of query q and the manually optimized version of this SQL query. The columns labeled *t (ms)* show the time it takes to execute the specified query (executing the query and retrieving all rows in the result).

	(original)	(rewritten)
Relational Database Management System	t (ms)	t (ms)
PostgreSQL 9.6.1	411 200	397
SQL Server 2016 (SP1) 13.0.4001.0	1 636	230
Oracle Database 12c Release 12.1.0.2.0	10 280	2631

Also this rewrite results in significant performance gains in several relational database management systems, as shown in Table 11.2. Also in this case, we implemented the query in Neo4j using the Cypher query language; yielding the query

```
MATCH (a)-[:edge]->(b)-[:edge]-(c)-[:edge]->(d)
RETURN a, COUNT(a)
```

We ran this query in Neo4j 3.3.2 and although it did yield the proper result, it only did so after 118s.

Although the above measurements are limited in scope, they do underline that modern relational database management systems and modern graph database systems have severe limitations with respect to evaluating even basic graph queries. At the same time, we see that manual rewriting of queries can improve the performance significantly. The need for this manual fine-tuning of queries is unfortunately, as we believe that these kind of rewritings should be performed *for* the users, rather than *by* the users. We believe that in the construction of practical high-performance graph query engines, our semi-join optimizations can play a crucial role in reducing the need for this type of manual fine-tuning. Additionally, we believe that our semi-join optimizations can be adapted to the setting of relational database management systems. This will require extending our results to the relation algebra with counting operators, aggregations, multiset semantics, and richer forms of iteration (see also Section 6.4), and study how these extensions interact with our semi-join-based rewrite rules. As a starting point in this direction, we observe that basic forms of aggregation rewrite rules in the multiset relational algebra have received some attention (e.g., [18, 19, 84]).

11.4 Query evaluation in relational database management systems

There exists a vast literature on query optimization and evaluation for traditional relational database management systems (e.g., [16, 38, 60, 62, 73, 85, 94]). Query evaluation of SQL queries can be summarized in three steps:

1. *Parsing, validation, and view resolution.* First, the SQL query is parsed into an internal form and the used relations, attributes, and views are verified against the schema of the database.

2. *Query planning and optimization.* Then, the query optimizer translates the internal representation of the query into an execution plan that describes exactly how the query needs to be executed. These plans include specifications where data should be read

from (e.g., relations, indices), which algorithms are used to perform each operator (e.g., hash-joins or merge-joins), how intermediate results are passed between operators (e.g., pipelining results or via result materialization), and the order in which operations are performed.

Finding a good or optimal query plan with respect to query execution time (or other cost measurements such as memory usage, disk IOs, or network communication) often involves a costly and complex search procedure. Usually, the first step in this search process is to put the query into a normal form in which selection and projection operators are performed as early as possible with the aim to reduce the size and cost to compute intermediate results (not unlike the aim of the semi-join rewrite rules we study in this part). Next, the most important step in this search process is determining the order in which joins are performed. To do so, different join orders are enumerated and, using a cost estimation for each join order, a join order is selected that should minimize query execution time. These cost estimations are usually based on available metadata, sampling, and other techniques [73].

3. *Plan compilation and execution.* Finally, after constructing the query plan, the plan is translated into an executable format and executed.

Efficient evaluation of joins is central to high-performance query evaluation of SQL queries in relational database management systems: joins of many-to-many relationships can significantly blow up the size of intermediate query results, especially when the query plan performs joins in a sub-optimal order. Luckily, query planning and optimization usually leads to a decent query plan, especially for typical joins involving mainly many-to-one relationships.

Unfortunately, as we have presented in the previous section, we cannot rely on the current state-of-the-art query optimizers to always generate optimal query plans for graph queries. Especially for complex and large graph queries involving many-to-many relationships, the quality of the resultant query plan seems insufficient, which can have many causes (e.g., a lack of considered query alternatives, an insufficient cost model, or an inaccurate cost model that deviates significantly from the real costs).

For relational systems, several techniques have been proposed to deal with such issues [17, 99]. In contrast, we believe that cost estimation for graph database systems is especially challenging and deserves more in-depth research. An obvious challenge in cost estimation for graph data is the lack of schema information, primary keys, and foreign keys. Hence, cost estimators need to rely more on collected summaries and samples of the data.[25] Furthermore, typical big graph datasets have significant data skew, as the evolution of real-world graph usually adheres to a power law [5, 6].[26] In such real-world graph data sets, the number of incoming edges and outgoing edges of nodes is not uniformly distributed: a few nodes are expected to participate in many edges, whereas most nodes only participate in a few edges. This skew can negatively influence the usability of collected summaries and samples of the data.

[25] Summaries of the data distribution in edge relations can also directly aid the semi-join rewrites. Using this information, we can determine whether an edge label is a node expression; which would enable additional rewrite opportunities.

[26] Graphs whose evolution adhere to a power law are referred to as *scale-free networks*. A recent study on many real-world graphs shows that the assumption that many real-world graphs are scale-free is an oversimplification [14]. Still, it remains true that most real-world graphs have significant data skew.

Taking into account the typical structure of real-world graph data will not only improve the accuracy of cost estimation, but can also be exploited in the design and implementation of specialized algorithms for evaluating complex operators. It is, for example, well-known that the diameter of real-world graphs is typically low [5, 6]. This strongly limits the number of steps necessary to evaluate Kleene-star and fixpoint operators. Hence, algorithms for evaluating these operators can assume that they only need to perform a few steps, and optimize their evaluation approach accordingly.

11.5 Efficient graph query evaluation and joins

As already noted, the efficient evaluation of joins is central to high-performance query evaluation. To conclude our overview of, on the one hand, available techniques for query evaluation, and, on the other hand, necessary directions of future work for high-performance graph database systems and graph query engines, we take a closer look at the types of joins resulting from relation algebra operators. In the relation algebra, where joins are primarily the result of composition steps, we can distinguish the following types of joins:

1. *Necessary sequential composition (acyclic joins).* We cannot rule out legitimate use of the composition operator, such as in the query $GrandparentOf = ParentOf \circ ParentOf$ of Chapter 1. Consequently, the only real way to deal with these acyclic joins is by finding a good query plan and execute it. As acyclic joins have a central role in dealing with normalized relations in relational databases, there is a vast collection of query planning strategies, query optimizations, query heuristics, and query algorithms to aid in answering these kinds of composition efficiently (e.g., [98]). Hence, with respect to these joins, graph database systems can rely largely on established query planning strategies and join algorithms.

2. *Unnecessary sequential composition.* As seen in Chapter 8, usages of compositions within projections and coprojections are wasteful, as computed information is thrown out later during evaluation. For these forms of sequential composition, we have presented in this part techniques to replace these unnecessary compositions by semi-joins. These rewrite rules can yield complex subexpressions that are repeatedly used at several places in the final expression. Hence, proper implementation needs to take this into account, which can complicate query planning and (pipelined) query evaluation beyond what is normally seen in relational database management systems.

3. *Parallel compositions (cyclic joins).* The only relation algebra queries that involve *cyclic joins* are queries that rely on compositions and intersections or compositions and differences. Take, for example, the non-basic expression $(FriendOf \circ FriendOf) \cap FriendOf$ that yields pairs (m, n) such that m is both a friend and a friend-of-a-friend of n. Let $R(A, B)$ be a relational database table representing the edge relation $FriendOf$. The above relation algebra query is equivalent to the relational algebra query

$$\pi_{R.A,S.B}(\sigma_{R.A=T.A \wedge R.B=S.A \wedge T.B=S.B}(R \bowtie \rho_S(R) \bowtie \rho_T(R))),$$

which is a classic example of a cyclic join [3].

We observe that these cyclic joins prevent application of the semi-join rewrite rules of Figure 8.1. It it also well-known that these cyclic joins do not have full semi-join reducers [11, 12], limiting applicability of distributed join evaluation techniques.

Recently, it was proven that cyclic joins cannot be answered optimally by a sequence of binary join steps, whereas multi-way join algorithms can answer these optimally [3, 96]. Hence, to fully support intersection and compositions efficiently, beyond the use cases of Proposition 9.9, such multi-way join algorithms need to be incorporated in graph query engines. The development of relational database management systems that incorporate these multi-way join algorithms is still ongoing. Therefore, developing graph query engines that successfully incorporate these multi-way algorithms during query planning and query evaluation is a clear avenue of future work that can benefit not only graph database systems, but also relational database management systems.

As seen in Chapter 10, certain queries that appear cyclic can still be optimized heavily. Take, for example, the relation algebra query

$$\langle \mathit{Alice} \rangle \circ ((\mathit{FriendOf} \circ \mathit{FriendOf}) \cap \mathit{FriendOf}).$$

The above query is equivalent to the relational algebra query

$$\pi_{R.A,S.B}(\sigma_{R.A=\text{'Alice'} \wedge R.A=T.A \wedge R.B=S.A \wedge T.B=S.B}(R \bowtie \rho_S(R) \bowtie \rho_T(R))).$$

By transitivity, we have $T.A =$ 'Alice' after which we can eliminate the term $R.A = T.A$, breaking the cycle. We observe that breaking up cycles is a well-known technique in relational database management systems that, unfortunately, can only be applied in rather limited cases (e.g., [12, 61]).[27]

Fortunately, we have already been able to establish in this work two cases in which cyclic graph queries can be made acyclic. First, for several restricted graph data models, redundancy of intersection and difference can be exploited to break up cyclic joins by eliminating intersection and difference (see e.g., [74], Corollary 5.6, and Theorems 5.12 and 4.10). Second, in Chapter 10, we have explored using node selections to break up cycles. Unfortunately, both techniques can blow up the size of queries significantly, and we have not yet determined in which cases these techniques make graph query evaluation more efficient. Still, it is worthwhile to investigate further opportunities for eliminating cyclic joins and to explorer when such cycle-elimination techniques have practical applications.

Overall, the above join classification does imply that one can rely on established techniques used in relational database management systems for most of the join requirements of graph database systems. At the same time, we also identified important roles for the semi-join rewrite rules we proposed, the node selection optimizations we briefly looked at, for multi-way join algorithms, and for cost estimation of graph queries. To integrate all these aspects in the query optimizers of future graph database systems is in itself challenging. This is especially the case if the query optimizer not only has to find an *efficient* query plan, but also needs to do so *fast*. Hence, as stated at the begin of this chapter, the design and implementation of high-performance graph database systems and graph query engines will require future research.

[27] Cyclic joins not only have a huge effect on the complexity of query evaluation and the complexity of distributed query evaluation [3, 11], but also on many other related problems that conceptually involve joins, e.g. checking integrity of relational data with respect to join dependencies [41].

Part IV

On Tarski's Relation Algebra

A GENERAL CONCLUSION

CHAPTER 12

Conclusion

Superficially, Parts II and III present results on two seemingly unrelated formal aspects of Tarski's relation algebra. Looking more in depth, however, one can see that the two studied aspects are actually closely related. Indeed, both parts concern the expressive power of fragments of Tarski's relation algebra in a non-traditional restricted setting.

In Part II, we focus on the expressive power of the relation algebra if we restrict the types of the *structures* we query. Specifically, we restrict ourselves to querying labeled and unlabeled trees and chains rather than graphs. Compared to querying graphs, as studied by Fletcher et al. [31–34, 87], this restriction has a dramatic influence on the expressive power of several fragments of the relation algebra. Indeed, many typical tree queries can be expressed in downward or local fragments for which we have shown that the intersection and difference operators are redundant. This is due to the structural limitations of trees and chains, which proved to be both a blessing and a curse, the latter because separations results turned out to be much harder to establish on trees and chains than on graphs. We can apply the results of Part II for *query optimization*, as we have shown that graph queries can often be simplified when evaluated over simpler structures.

In Part III, we focus on the expressive power of the relation algebra if we restrict ourselves to *projection equivalence* rather than path equivalence or Boolean equivalence of fragments. In this way, we were able to relate the expressive power of many fragments of the relation algebra with fragments of the semi-join algebra, showing that the complexity of these fragments is very limited. We translated these results to *query optimization* techniques that can simplify the evaluation of relation algebra queries, especially of queries that use projections.

In conclusion, we study in both Parts II and III the effects on the expressive power of fragments of the relation algebra (and, consequently, of many practical graph query languages), if we either restrict the graph structures we query or simplify the semantics of evaluation. In both cases, we observe that these restrictions influence the expressive power of the query language under consideration. Hence, in both cases, these restrictions can be used for optimizing graph query evaluation. Unfortunately, we could only touch briefly on the practical applications within the scope of this work. Hence, we believe that the further development of graph query evaluation and optimization techniques based on our results is a prime avenue for future work.

Bibliography

[1] Serge Abiteboul, Richard Hull, and Victor Vianu, editors. *Foundations of Databases: The Logical Level*. Addison-Wesley Longman Publishing Co., Inc., 1st edition, 1995.

[2] Waseem Akhtar, Álvaro Cortés-Calabuig, and Jan Paredaens. Constraints in RDF. In *Semantics in Data and Knowledge Bases*, pages 23–39. Springer Berlin Heidelberg, 2011.

[3] Albert Atserias, Martin Grohe, and Dániel Marx. Size bounds and query plans for relational joins. *SIAM Journal on Computing*, 42(4):1737–1767, 2013.

[4] Nurzhan Bakibayev, Dan Olteanu, and Jakub Závodný. FDB: A query engine for factorised relational databases. *Proceedings of the VLDB Endowment*, 5(11):1232–1243, 2012.

[5] Albert-László Barabási. *Network Science*. Cambridge University Press, 1st edition, 2016.

[6] Albert-László Barabási and Réka Albert. Emergence of scaling in random networks. *Science*, 286(5439):509–512, 1999.

[7] Pablo Barceló. Querying graph databases. In *Proceedings of the 32nd Symposium on Principles of Database Systems*, pages 175–188. ACM, 2013.

[8] Pablo Barceló, Jorge Pérez, and Juan L Reutter. Relative expressiveness of nested regular expressions. In *Proceedings of the 6th Alberto Mendelzon International Workshop on Foundations of Data Management*, pages 180–195. CEUR Workshop Proceedings, 2012.

[9] Michael Benedikt, Wenfei Fan, and Gabriel Kuper. Structural properties of XPath fragments. *Theoretical Computer Science*, 336(1):3–31, 2005.

[10] Michael Benedikt and Christoph Koch. XPath leashed. *ACM Computing Surveys*, 41(1):3:1–3:54, 2009.

[11] Philip A. Bernstein and Dah-Ming W. Chiu. Using semi-joins to solve relational queries. *Journal of the ACM*, 28(1):25–40, 1981.

[12] Philip A. Bernstein and Nathan Goodman. Power of natural semijoins. *SIAM Journal on Computing*, 10(4):751–771, 1981.

[13] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, François Yergeau, and John Cowan. Extensible markup language (XML) 1.1 (second edition). W3C recommendation, W3C, 2006. URL: `http://www.w3.org/TR/2006/REC-xml11-20060816`.

[14] Anna D. Broido and Aaron Clauset. Scale-free networks are rare. Technical report, University of Colorado, 2018. URL: https://arxiv.org/abs/1801.03400.

[15] Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Moshe Y. Vardi. Containment of conjunctive regular path queries with inverse. In *Proceedings of the Seventh International Conference on Principles of Knowledge Representation and Reasoning*, pages 176–185. Morgan Kaufmann Publishers Inc., 2000.

[16] Surajit Chaudhuri. An overview of query optimization in relational systems. In *Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 34–43, 1998.

[17] Surajit Chaudhuri. Query optimizers: Time to rethink the contract? In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, pages 961–968, 2009.

[18] Surajit Chaudhuri and Kyuseok Shim. Including group-by in query optimization. In *Proceedings of the 20th International Conference on Very Large Data Bases*, pages 354–366. Morgan Kaufmann Publishers Inc., 1994.

[19] Surajit Chaudhuri and Kyuseok Shim. *Optimizing queries with aggregate views*, pages 167–182. Springer Berlin Heidelberg, 1996.

[20] James Clark and Steve DeRose. XML path language (XPath) version 1.0. W3C recommendation, W3C, 1999. URL: http://www.w3.org/TR/1999/REC-xpath-19991116.

[21] Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. The MIT Press, 1999.

[22] Rance Cleaveland and Bernhard Steffen. A linear-time model-checking algorithm for the alternation-free modal mu-calculus. *Formal Methods in System Design*, 2(2):121–147, 1993.

[23] E. F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6), 1970.

[24] Latha S. Colby. A recursive algebra for nested relations. *Information Systems*, 15(5):567–582, 1990.

[25] Mariano P. Consens and Alberto O. Mendelzon. GraphLog: A visual formalism for real life recursion. In *Proceedings of the Ninth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 404–416. ACM, 1990.

[26] Transaction Processing Performance Council. TPC benchmark^{TM} H (decision support), 2017. URL: http://www.tpc.org/tpch/.

[27] Isabel F. Cruz, Alberto O. Mendelzon, and Peter T. Wood. A graphical query language supporting recursion. In *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data*, pages 323–330. ACM, 1987.

[28] R. J. Duffin. Topology of series-parallel networks. *Journal of Mathematical Analysis and Applications*, 10(2):303–318, 1965.

[29] Ecma International. The JSON data interchange syntax, 2nd edition, 2017. URL: `http://www.ecma-international.org/publications/standards/Ecma-404.htm`.

[30] Michael J. Fischer and Richard E. Ladner. Propositional dynamic logic of regular programs. *Journal of Computer and System Sciences*, 18(2):194–211, 1979.

[31] George H. L. Fletcher, Marc Gyssens, Dirk Leinders, Dimitri Surinx, Jan Van den Bussche, Dirk Van Gucht, Stijn Vansummeren, and Yuqing Wu. Relative expressive power of navigational querying on graphs. *Information Sciences*, 298:390–406, 2015.

[32] George H. L. Fletcher, Marc Gyssens, Dirk Leinders, Jan Van den Bussche, Dirk Van Gucht, Stijn Vansummeren, and Yuqing Wu. Relative expressive power of navigational querying on graphs. In *Proceedings of the 14th International Conference on Database Theory*, pages 197–207. ACM, 2011.

[33] George H. L. Fletcher, Marc Gyssens, Dirk Leinders, Jan Van den Bussche, Dirk Van Gucht, Stijn Vansummeren, and Yuqing Wu. The impact of transitive closure on the boolean expressiveness of navigational query languages on graphs. In *Proceedings of the 7th International Symposium on Foundations of Information and Knowledge Systems*, volume 7153, pages 124–143. Springer Berlin Heidelberg, 2012.

[34] George H. L. Fletcher, Marc Gyssens, Dirk Leinders, Jan Van den Bussche, Dirk Van Gucht, Stijn Vansummeren, and Yuqing Wu. The impact of transitive closure on the expressiveness of navigational query languages on unlabeled graphs. *Annals of Mathematics and Artificial Intelligence*, 73(1-2):167–203, 2015.

[35] George H. L. Fletcher, Marc Gyssens, Jan Paredaens, Dirk Van Gucht, and Yuqing Wu. Structural characterizations of the navigational expressiveness of relation algebras on a tree. *Journal of Computer and System Sciences*, 82(2):229–259, 2016.

[36] Steven Givant. The calculus of relations as a foundation for mathematics. *Journal of Automated Reasoning*, 37(4):277–322, 2006.

[37] Erich Grädel and Martin Otto. On logics with two variables. *Theoretical Computer Science*, 224(1–2):73–113, 1999.

[38] Goetz Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2):73–169, 1993.

[39] Martin Grohe. Finite variable logics in descriptive complexity theory. *The Bulletin of Symbolic Logic*, 4:345–398, 1998.

[40] Paolo Guagliardo and Leonid Libkin. A formal semantics of SQL queries, its validation, and applications. *Proceedings of the VLDB Endowment*, 11(1):27–39, 2017.

[41] Marc Gyssens. On the complexity of join dependencies. *ACM Transactions on Database Systems*, 11(1):81–108, 1986.

[42] Marc Gyssens, Jelle Hellings, Jan Paredaens, Dirk Van Gucht, Jef Wijsen, and Yuqing Wu. Calculi for symmetric queries.

[43] Marc Gyssens, Jan Paredaens, Dirk Van Gucht, and George H. L. Fletcher. Structural characterizations of the semantics of XPath as navigation tool on a document. In *Proceedings of the Twenty-fifth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 318–327. ACM, 2006.

[44] Steven Harris and Andy Seaborne. SPARQL 1.1 query language. W3C recommendation, W3C, 2013. URL: `http://www.w3.org/TR/2013/REC-sparql11-query-20130321`.

[45] Jelle Hellings. Bisimulation partitioning and partition maintenance on very large directed acyclic graphs. Master's thesis, Eindhoven University of Technology, 2011.

[46] Jelle Hellings. Conjunctive context-free path queries. In *Proceedings of the 17th International Conference on Database Theory*, pages 119–130, 2014.

[47] Jelle Hellings. Path results for context-free grammar queries on graphs. Technical report, Hasselt University, 2016. URL: `https://arxiv.org/abs/1502.02242`.

[48] Jelle Hellings, George H.L. Fletcher, and Herman Haverkort. Efficient external-memory bisimulation on dags. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 553–564, 2012.

[49] Jelle Hellings, Marc Gyssens, Dirk Van Gucht, and Yuqing Wu. First-order definable counting-only queries. In *Proceedings of the 10th International Symposium on Foundations of Information and Knowledge Systems*, 2018.

[50] Jelle Hellings, Marc Gyssens, Jan Paredaens, and Yuqing Wu. Implication and axiomatization of functional constraints on patterns with an application to the RDF data model. In *Proceedings of the 8th International Symposium on Foundations of Information and Knowledge Systems*, pages 250–269. Springer International Publishing, 2014.

[51] Jelle Hellings, Marc Gyssens, Jan Paredaens, and Yuqing Wu. Implication and axiomatization of functional and constant constraints. *Annals of Mathematics and Artificial Intelligence*, 76(3):251–279, 2016.

[52] Jelle Hellings, Marc Gyssens, Yuqing Wu, Dirk Van Gucht, Jan Van den Bussche, Stijn Vansummeren, and George H. L. Fletcher. Relative expressive power of downward fragments of navigational query languages on trees and chains. In *Proceedings of the 15th Symposium on Database Programming Languages*, pages 59–68, 2015.

[53] Jelle Hellings, Marc Gyssens, Yuqing Wu, Dirk Van Gucht, Jan Van den Bussche, Stijn Vansummeren, and George H. L. Fletcher. Comparing downward fragments of the relational calculus with transitive closure on trees. Technical report, Hasselt University, 2018. URL: `https://arxiv.org/abs/1803.01390`.

[54] Jelle Hellings, Bart Kuijpers, Jan Van den Bussche, and Xiaowang Zhang. Walk logic as a framework for path query languages on graph databases. In *Proceedings of the 16th International Conference on Database Theory*, pages 117–128, 2013.

[55] Jelle Hellings, Catherine L. Pilachowski, Dirk Van Gucht, Marc Gyssens, and Yuqing Wu. From relation algebra to semi-join algebra: An approach for graph query optimization. In *Proceedings of the 16th International Symposium on Database Programming Languages*, pages 5:1–5:10, 2017.

[56] Jelle Hellings and Yuqing Wu. Stab-forests: Dynamic data structures for efficient temporal query processing.

[57] Jelle Hellings, Yuqing Wu, Marc Gyssens, and Dirk Van Gucht. The power of Tarski's relation algebra on trees. In *Proceedings of the 10th International Symposium on Foundations of Information and Knowledge Systems*, 2018.

[58] Jan Hidders, Jan Paredaens, and Jan Van den Bussche. J-Logic: Logical foundations for JSON querying. In *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pages 137–149, 2017.

[59] International Organization for Standardization. ISO/IEC 9075-1: Information technology – database languages – SQL, 2016.

[60] Yannis E. Ioannidis. Query optimization. *ACM Computing Surveys*, 28(1):121–123, 1996.

[61] Matthias Jarke and Jürgen Koch. Range nesting: A fast method to evaluate quantified queries. *SIGMOD Record*, 13(4):196–206, 1983.

[62] Matthias Jarke and Jurgen Koch. Query optimization in database systems. *ACM Computing Surveys*, 16(2):111–152, 1984.

[63] Raghav Kaushik, Philip Bohannon, Jeffrey F Naughton, and Henry F Korth. Covering indexes for branching path queries. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, pages 133–144. ACM, 2002.

[64] Aviel Klausner and Nathan Goodman. Multirelations: Semantice and languages. In *Proceedings of the 11th International Conference on Very Large Data Bases*, pages 251–258. VLDB Endowment, 1985.

[65] Dexter Kozen. Kleene algebra with tests. *ACM Transactions on Programming Languages and Systems*, 19(3):427–443, 1997.

[66] Dirk Leinders. *The semijoin algebra.* PhD thesis, Hasselt University and transnational University of Limburg, 2008.

[67] Dirk Leinders, Maarten Marx, Jerzy Tyszkiewicz, and Jan Van den Bussche. The semijoin algebra and the guarded fragment. *Journal of Logic, Language and Information*, 14(3):331–343, 2005.

[68] Dirk Leinders, Jerzy Tyszkiewicz, and Jan Van den Bussche. On the expressive power of semijoin queries. *Information Processing Letters*, 91(2):93–98, 2004.

[69] Dirk Leinders and Jan Van den Bussche. On the complexity of division and set joins in the relational algebra. *Journal of Computer and System Sciences*, 73(4):538–549, 2007. Special Issue: Database Theory 2005.

[70] Leonid Libkin. *Elements of Finite Model Theory.* Springer Berlin Heidelberg, 2004.

[71] Leonid Libkin, Wim Martens, and Domagoj Vrgoč. Querying graph databases with XPath. In *Proceedings of the 16th International Conference on Database Theory*, pages 129–140. ACM, 2013.

[72] Peter Linz. *An Introduction to Formal Languages and Automata.* Jones and Bartlett Publishers, Inc., 5th edition, 2012.

[73] Michael V. Mannino, Paicheng Chu, and Thomas Sager. Statistical profile estimation in database systems. *ACM Computing Surveys*, 20(3):191–221, 1988.

[74] Maarten Marx. Conditional XPath. *ACM Transactions on Database Systems*, 30(4):929–959, 2005.

[75] Maarten Marx and Maarten de Rijke. Semantic characterizations of navigational XPath. *SIGMOD Record*, 34(2):41–46, 2005.

[76] Maarten Marx and Yde Venema. *Multi-Dimensional Modal Logic.* Springer Netherlands, 1997.

[77] MySQL Documentation Team. MySQL 5.7 reference manual – 8.2.2.1 optimizing subqueries, derived tables, and view references with semi-join transformations, 2017. URL: `https://dev.mysql.com/doc/refman/5.7/en/semi-joins.html` [cited 21 December 2017].

[78] Dan Olteanu and Jakub Závodný. Size bounds for factorised representations of query results. *ACM Transactions on Database Systems*, 40(1):2:1–2:44, 2015.

[79] Martin Otto. The expressive power of fixed-point logic with counting. *Journal of Symbolic Logic*, 61(1):147––176, 1996.

[80] Martin Otto. Bounded variable logics: two, three, and more. *Archive for Mathematical Logic*, 38(4):235–256, 1999.

[81] Juan L. Reutter, Miguel Romero, and Moshe Y. Vardi. Regular queries on graph databases. *Theory of Computing Systems*, 61(1):31–83, 2017.

[82] Ian Robinson, Jim Webber, and Emil Eifrem. *Graph Databases: New Opportunities for Connected Data.* O'Reilly Media, Inc., 2nd edition, 2015.

[83] Guus Schreiber and Yves Raimond. RDF 1.1 primer. W3C working group note, W3C, 2014. URL: `http://www.w3.org/TR/2014/NOTE-rdf11-primer-20140624`.

[84] Praveen Seshadri, Joseph M. Hellerstein, Hamid Pirahesh, T. Y. Cliff Leung, Raghu Ramakrishnan, Divesh Srivastava, Peter J. Stuckey, and S. Sudarshan. Cost-based optimization for magic: Algebra and implementation. *SIGMOD Record*, 25(2):435–446, 1996.

[85] Avi Silberschatz, Henry F. Korth, and S Sudarshan. *Database System Concepts.* McGraw-Hill, 6 edition, 2011.

[86] Dimitri Surinx. *A Framework for Comparing Query Languages in Their Ability to Express Boolean Queries.* PhD thesis, Hasselt University and transnational University of Limburg, 2017.

[87] Dimitri Surinx, George H. L. Fletcher, Marc Gyssens, Dirk Leinders, Jan Van den Bussche, Dirk Van Gucht, Stijn Vansummeren, and Yuqing Wu. Relative expressive power of navigational querying on graphs using transitive closure. *Logic Journal of the IGPL*, 23(5):759–788, 2015.

[88] Dimitri Surinx, Jan Van den Bussche, and Dirk Van Gucht. The primitivity of operators in the algebra of binary relations under conjunctions of containments. *2017 32nd Annual ACM/IEEE Symposium on Logic in Computer Science*, pages 1–10, 2017.

[89] Alfred Tarski. On the calculus of relations. *The Journal of Symbolic Logic*, 6(3):73–89, 1941.

[90] Balder ten Cate. The expressivity of XPath with transitive closure. In *Proceedings of the Twenty-fifth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 328–337. ACM, 2006.

[91] Balder ten Cate and Maarten Marx. Navigational XPath: Calculus and algebra. *SIGMOD Record*, 36(2):19–26, 2007.

[92] The PostgreSQL Global Development Group. JSON functions and operators, 2017. URL: `https://www.postgresql.org/docs/10/static/functions-json.html` [cited 21 December 2017].

[93] D. C. Tsichritzis and F. H. Lochovsky. Hierarchical data-base management: A survey. *ACM Computing Surveys*, 8(1):105–123, 1976.

[94] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems: Volume II: The New Technologies*. W. H. Freeman & Co., 1990.

[95] Moshe Y. Vardi. The complexity of relational query languages (extended abstract). In *Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing*, pages 137–146. ACM, 1982.

[96] Todd L. Veldhuizen. Leapfrog triejoin: A simple, worst-case optimal join algorithm. In *Proceedings of the 17th International Conference on Database Theory*, pages 96–106, 2014.

[97] Yuqing Wu, Dirk Van Gucht, Marc Gyssens, and Jan Paredaens. A study of a positive fragment of path queries: Expressiveness, normal form and minimization. *The Computer Journal*, 54(7):1091–1118, 2011.

[98] Mihalis Yannakakis. Algorithms for acyclic database schemes. In *Proceedings of the Seventh International Conference on Very Large Data Bases - Volume 7*, pages 82–94. VLDB Endowment, 1981.

[99] Shaoyi Yin, Abdelkader Hameurlain, and Franck Morvan. Robust query optimization methods with respect to estimation errors: A survey. *SIGMOD Record*, 44(3):25–36, 2015.

APPENDIX A

Index on the relative expressive power

In Part II, we study the relative expressive power of fragments of the relation algebra when querying unlabeled chains, labeled chains, unlabeled paths, or labeled paths. We study the relative expressive power with respect to Boolean semantics and path semantics. To make it easier to locate specific results, the following pages consist of index tables to the results presented in Part II.

There are eight tables in total. Each table is an index to the separation and collapse results discussed in Part II. Consider the table titled "z semantics on s" with $z \in \{\text{bool}, \text{path}\}$ and $s \in \{\text{unbaled chains}, \text{labeled chains}, \text{unlabeled paths}, \text{labeled paths}\}$, and let $(\mathcal{N}(\mathcal{F}), op)$ be a field in this table. This field has one of the following forms:

. Indicates non-applicability, as $op \in \mathcal{F}$.

3.24 ✗ . The cross ✗ indicates that already on s, we have $\mathcal{N}(\mathcal{F} \cup \{op\}) \not\leq_z \mathcal{N}(\mathcal{F})$. We refer to Reference 3.24 for details.

3.35 ✗• . The cross ✗ indicates that already on s, we have $\mathcal{N}(\mathcal{F} \cup \{op\}) \not\leq_z \mathcal{N}(\mathcal{F})$. This result is provided by related work and we refer to Reference 3.35 for details.

4.10 ✓ . The check mark ✓ indicates that on s, we have $\mathcal{N}(\mathcal{F} \cup \{op\}) \leq_z \mathcal{N}(\mathcal{F})$. We refer to Reference 4.10 for details.

3.36 ✓• . The check mark ✓ indicates that on s, we have $\mathcal{N}(\mathcal{F} \cup \{op\}) \leq_z \mathcal{N}(\mathcal{F})$. This result is provided by related work and we refer to Reference 3.36 for details.

? . The question mark **?** indicates an open problem.

The referenced result identifies either a theorem, corollary, proposition, or lemma, which all use a single chapter-based continuous counter. For example, Reference 3.1 uniquely references the first result in Chapter 3.

For brevity, some results are only obtained by combining the referenced result with a general collapse result such as Theorem 4.10 or Corollary 5.6. As an example, we state $\mathcal{N}(\overline{\pi}) \not\leq_{\text{bool}} \mathcal{N}(\cap, -, ^*)$ by Reference 3.24. This result is not directly proven by Theorem 3.24: by Theorem 4.10, we have $\mathcal{N}(\cap, -, ^*) \leq_{\text{path}} \mathcal{N}(^*)$, after which Theorem 3.24 shows that $\mathcal{N}(\overline{\pi}) \not\leq_{\text{bool}} \mathcal{N}(^*)$.

A.1 Boolean semantics on unlabeled chains

	di	$\frown$	π	$\overline{\pi}$	$\cap$	$-$	*
$\mathcal{N}()$	3.23 ✓	4.12 ✓	4.12 ✓	3.24 ✗	4.10 ✓	4.10 ✓	3.34 ✓•
$\mathcal{N}(\cap)$	3.23 ✓	4.12 ✓	4.12 ✓	3.24 ✗		4.10 ✓	3.23 ✓
$\mathcal{N}(\cap, -)$	3.24 ✗	3.24 ✗	3.24 ✗	3.24 ✗			3.23 ✓
$\mathcal{N}(\pi)$	3.23 ✓	3.34 ✓•		3.24 ✗	4.10 ✓	3.24 ✗	3.34 ✓•
$\mathcal{N}(\pi, \cap)$	3.23 ✓	3.23 ✓		3.24 ✗		3.24 ✗	3.23 ✓
$\mathcal{N}(\pi, \overline{\pi})$	3.30 ✓	3.34 ✓•			4.10 ✓	4.10 ✓	3.26 ✗
$\mathcal{N}(\pi, \overline{\pi}, \cap)$	3.30 ✓	3.30 ✓				4.10 ✓	3.26 ✗
$\mathcal{N}(\pi, \overline{\pi}, \cap, -)$	3.30 ✓	3.30 ✓					3.26 ✗
$\mathcal{N}(\frown)$	3.23 ✓		3.13 ✓	3.24 ✗	3.13 ✓	3.24 ✗	3.23 ✓
$\mathcal{N}(\frown, \pi)$	3.23 ✓			3.24 ✗	3.36 ✓•	3.24 ✗	3.23 ✓
$\mathcal{N}(\frown, \pi, \cap)$	3.23 ✓			3.24 ✗		3.24 ✗	3.23 ✓
$\mathcal{N}(\frown, \pi, \overline{\pi})$	3.30 ✓				5.6 ✓	5.13 ✓	3.26 ✗
$\mathcal{N}(\frown, \pi, \overline{\pi}, \cap)$	3.30 ✓					5.13 ✓	3.26 ✗
$\mathcal{N}(\frown, \pi, \overline{\pi}, \cap, -)$	3.30 ✓						3.26 ✗
$\mathcal{N}(\text{di})$		3.23 ✓	3.23 ✓	3.24 ✗	3.23 ✓	3.24 ✗	3.34 ✓•
$\mathcal{N}(\text{di}, \pi)$		3.34 ✓•		3.24 ✗	3.23 ✓	3.24 ✗	3.34 ✓•
$\mathcal{N}(\text{di}, \pi, \cap)$		3.23 ✓		3.24 ✗		3.24 ✗	3.23 ✓
$\mathcal{N}(\text{di}, \pi, \overline{\pi})$		3.34 ✓•			3.30 ✓	3.30 ✓	3.26 ✗
$\mathcal{N}(\text{di}, \pi, \overline{\pi}, \cap)$		3.30 ✓				3.30 ✓	3.26 ✗
$\mathcal{N}(\text{di}, \pi, \overline{\pi}, \cap, -)$		3.30 ✓					3.26 ✗
$\mathcal{N}(\text{di}, \frown)$			3.23 ✓	3.24 ✗	3.23 ✓	3.24 ✗	3.23 ✓
$\mathcal{N}(\text{di}, \frown, \pi)$				3.24 ✗	3.23 ✓	3.24 ✗	3.23 ✓
$\mathcal{N}(\text{di}, \frown, \pi, \cap)$				3.24 ✗		3.24 ✗	3.23 ✓
$\mathcal{N}(\text{di}, \frown, \pi, \overline{\pi})$					3.30 ✓	3.30 ✓	3.26 ✗
$\mathcal{N}(\text{di}, \frown, \pi, \overline{\pi}, \cap)$						3.30 ✓	3.26 ✗
$\mathcal{N}(\text{di}, \frown, \pi, \overline{\pi}, \cap, -)$							3.26 ✗

	di	$\frown$	π	$\overline{\pi}$	$\cap$	$-$	*
$\mathcal{N}(^*)$	3.23 ✓	3.23 ✓	3.23 ✓	3.24 ✗	4.10 ✓	4.10 ✓	
$\mathcal{N}(\cap, ^*)$	3.23 ✓	3.23 ✓	3.23 ✓	3.24 ✗		4.10 ✓	
$\mathcal{N}(\cap, -, ^*)$	3.24 ✗	3.24 ✗	3.24 ✗	3.24 ✗			
$\mathcal{N}(\pi, ^*)$	3.23 ✓	3.23 ✓		3.24 ✗	4.10 ✓	3.24 ✗	
$\mathcal{N}(\pi, \cap, ^*)$	3.23 ✓	3.23 ✓		3.24 ✗		3.24 ✗	
$\mathcal{N}(\pi, \overline{\pi}, ^*)$	?	?			4.10 ✓	4.10 ✓	
$\mathcal{N}(\pi, \overline{\pi}, \cap, ^*)$	?	?				4.10 ✓	
$\mathcal{N}(\pi, \overline{\pi}, \cap, -, ^*)$	?	?					
$\mathcal{N}(\frown, ^*)$	3.23 ✓		3.23 ✓	3.24 ✗	3.23 ✓	3.24 ✗	
$\mathcal{N}(\frown, \pi, ^*)$	3.23 ✓			3.24 ✗	3.23 ✓	3.24 ✗	
$\mathcal{N}(\frown, \pi, \cap, ^*)$	3.23 ✓			3.24 ✗		3.24 ✗	
$\mathcal{N}(\frown, \pi, \overline{\pi}, ^*)$	3.1 ✓				?	?	
$\mathcal{N}(\frown, \pi, \overline{\pi}, \cap, ^*)$	3.1 ✓					?	
$\mathcal{N}(\frown, \pi, \overline{\pi}, \cap, -, ^*)$	3.1 ✓						
$\mathcal{N}(\text{di}, ^*)$		3.23 ✓	3.23 ✓	3.24 ✗	3.23 ✓	3.24 ✗	
$\mathcal{N}(\text{di}, \pi, ^*)$		3.23 ✓		3.24 ✗	3.23 ✓	3.24 ✗	
$\mathcal{N}(\text{di}, \pi, \cap, ^*)$		3.23 ✓		3.24 ✗		3.24 ✗	
$\mathcal{N}(\text{di}, \pi, \overline{\pi}, ^*)$		?			?	?	
$\mathcal{N}(\text{di}, \pi, \overline{\pi}, \cap, ^*)$		?				?	
$\mathcal{N}(\text{di}, \pi, \overline{\pi}, \cap, -, ^*)$		3.1 ✓					
$\mathcal{N}(\text{di}, \frown, ^*)$			3.23 ✓	3.24 ✗	3.23 ✓	3.24 ✗	
$\mathcal{N}(\text{di}, \frown, \pi, ^*)$				3.24 ✗	3.23 ✓	3.24 ✗	
$\mathcal{N}(\text{di}, \frown, \pi, \cap, ^*)$				3.24 ✗		3.24 ✗	
$\mathcal{N}(\text{di}, \frown, \pi, \overline{\pi}, ^*)$					?	?	
$\mathcal{N}(\text{di}, \frown, \pi, \overline{\pi}, \cap, ^*)$						?	
$\mathcal{N}(\text{di}, \frown, \pi, \overline{\pi}, \cap, -, ^*)$							

	di	$\frown$	π	$\overline{\pi}$	$\cap$	$-$	*

A.2 Boolean semantics on labeled chains

	di	$\frown$	π	$\overline{\pi}$	$\cap$	$-$	*
$\mathcal{N}()$	3.14 ✗	4.12 ✓	4.12 ✓	3.24 ✗	4.10 ✓	4.10 ✓	3.26 ✗
$\mathcal{N}(\cap)$	3.14 ✗	4.12 ✓	4.12 ✓	3.24 ✗		4.10 ✓	3.26 ✗
$\mathcal{N}(\cap, -)$	3.24 ✗	3.24 ✗	3.24 ✗	3.24 ✗			3.26 ✗
$\mathcal{N}(\pi)$	3.14 ✗	3.34 ✓•		3.24 ✗	4.10 ✓	3.24 ✗	3.26 ✗
$\mathcal{N}(\pi, \cap)$	3.14 ✗	5.6 ✓		3.24 ✗		3.24 ✗	3.26 ✗
$\mathcal{N}(\pi, \overline{\pi})$	3.14 ✗	3.34 ✓•			4.10 ✓	4.10 ✓	3.26 ✗
$\mathcal{N}(\pi, \overline{\pi}, \cap)$	3.14 ✗	5.6 ✓				4.10 ✓	3.26 ✗
$\mathcal{N}(\pi, \overline{\pi}, \cap, -)$	3.14 ✗	5.13 ✓					3.26 ✗
$\mathcal{N}(\frown)$	3.14 ✗		3.13 ✓	3.24 ✗	3.13 ✓	3.24 ✗	3.26 ✗
$\mathcal{N}(\frown, \pi)$	3.14 ✗			3.24 ✗	3.36 ✓•	3.24 ✗	3.26 ✗
$\mathcal{N}(\frown, \pi, \cap)$	3.14 ✗			3.24 ✗		3.24 ✗	3.26 ✗
$\mathcal{N}(\frown, \pi, \overline{\pi})$	3.14 ✗				5.6 ✓	5.13 ✓	3.26 ✗
$\mathcal{N}(\frown, \pi, \overline{\pi}, \cap)$	3.14 ✗					5.13 ✓	3.26 ✗
$\mathcal{N}(\frown, \pi, \overline{\pi}, \cap, -)$	3.14 ✗						3.26 ✗
$\mathcal{N}(\text{di})$		3.16 ✗	3.16 ✗	3.24 ✗	3.16 ✗	3.24 ✗	3.26 ✗
$\mathcal{N}(\text{di}, \pi)$		3.34 ✓•		3.24 ✗	3.31 ✗	3.24 ✗	3.26 ✗
$\mathcal{N}(\text{di}, \pi, \cap)$		?		3.24 ✗		3.24 ✗	3.26 ✗
$\mathcal{N}(\text{di}, \pi, \overline{\pi})$		3.34 ✓•			3.31 ✗	3.31 ✗	3.26 ✗
$\mathcal{N}(\text{di}, \pi, \overline{\pi}, \cap)$		?				?	3.26 ✗
$\mathcal{N}(\text{di}, \pi, \overline{\pi}, \cap, -)$		?					3.26 ✗
$\mathcal{N}(\text{di}, \frown)$			?	3.24 ✗	3.31 ✗	3.24 ✗	3.26 ✗
$\mathcal{N}(\text{di}, \frown, \pi)$				3.24 ✗	3.31 ✗	3.24 ✗	3.26 ✗
$\mathcal{N}(\text{di}, \frown, \pi, \cap)$				3.24 ✗		3.24 ✗	3.26 ✗
$\mathcal{N}(\text{di}, \frown, \pi, \overline{\pi})$					3.31 ✗	3.31 ✗	3.26 ✗
$\mathcal{N}(\text{di}, \frown, \pi, \overline{\pi}, \cap)$						?	3.26 ✗
$\mathcal{N}(\text{di}, \frown, \pi, \overline{\pi}, \cap, -)$							3.26 ✗

	di	$\frown$	π	$\overline{\pi}$	$\cap$	$-$	*
$\mathcal{N}(^*)$	?	?	4.12 ✓	3.24 ✗	4.10 ✓	4.10 ✓	
$\mathcal{N}(\cap, ^*)$	?	?	4.12 ✓	3.24 ✗		4.10 ✓	
$\mathcal{N}(\cap, -, ^*)$	3.24 ✗	3.24 ✗	3.24 ✗	3.24 ✗			
$\mathcal{N}(\pi, ^*)$	?	?		3.24 ✗	4.10 ✓	3.24 ✗	
$\mathcal{N}(\pi, \cap, ^*)$	?	?		3.24 ✗		3.24 ✗	
$\mathcal{N}(\pi, \overline{\pi}, ^*)$	?	?			4.10 ✓	4.10 ✓	
$\mathcal{N}(\pi, \overline{\pi}, \cap, ^*)$	?	?				4.10 ✓	
$\mathcal{N}(\pi, \overline{\pi}, \cap, -, ^*)$	?	?					
$\mathcal{N}(\frown, ^*)$	3.1 ✓		?	3.24 ✗	?	3.24 ✗	
$\mathcal{N}(\frown, \pi, ^*)$	3.1 ✓			3.24 ✗	?	3.24 ✗	
$\mathcal{N}(\frown, \pi, \cap, ^*)$	3.1 ✓			3.24 ✗		3.24 ✗	
$\mathcal{N}(\frown, \pi, \overline{\pi}, ^*)$	3.1 ✓				?	?	
$\mathcal{N}(\frown, \pi, \overline{\pi}, \cap, ^*)$	3.1 ✓					?	
$\mathcal{N}(\frown, \pi, \overline{\pi}, \cap, -, ^*)$	3.1 ✓						
$\mathcal{N}(\text{di}, ^*)$		?	?	3.24 ✗	?	3.24 ✗	
$\mathcal{N}(\text{di}, \pi, ^*)$		?		3.24 ✗	?	3.24 ✗	
$\mathcal{N}(\text{di}, \pi, \cap, ^*)$		?		3.24 ✗		3.24 ✗	
$\mathcal{N}(\text{di}, \pi, \overline{\pi}, ^*)$		?			?	?	
$\mathcal{N}(\text{di}, \pi, \overline{\pi}, \cap, ^*)$		?				?	
$\mathcal{N}(\text{di}, \pi, \overline{\pi}, \cap, -, ^*)$		3.1 ✓					
$\mathcal{N}(\text{di}, \frown, ^*)$			?	3.24 ✗	?	3.24 ✗	
$\mathcal{N}(\text{di}, \frown, \pi, ^*)$				3.24 ✗	?	3.24 ✗	
$\mathcal{N}(\text{di}, \frown, \pi, \cap, ^*)$				3.24 ✗		3.24 ✗	
$\mathcal{N}(\text{di}, \frown, \pi, \overline{\pi}, ^*)$					?	?	
$\mathcal{N}(\text{di}, \frown, \pi, \overline{\pi}, \cap, ^*)$						?	
$\mathcal{N}(\text{di}, \frown, \pi, \overline{\pi}, \cap, -, ^*)$							

	di	$\frown$	π	$\overline{\pi}$	$\cap$	$-$	*

A.3 Boolean semantics on unlabeled trees

	di	$\frown$	π	$\overline{\pi}$	$\cap$	$-$	*
$\mathcal{N}()$	3.9 ✗	3.23 ✓	3.23 ✓	3.24 ✗	4.10 ✓	4.10 ✓	3.34 ✓•
$\mathcal{N}(\cap)$	3.9 ✗	3.23 ✓	3.23 ✓	3.24 ✗		4.10 ✓	3.23 ✓
$\mathcal{N}(\cap, -)$	3.24 ✗	3.24 ✗	3.24 ✗	3.24 ✗			3.23 ✓
$\mathcal{N}(\pi)$	3.9 ✗	3.34 ✓•		3.24 ✗	4.10 ✓	3.24 ✗	3.34 ✓•
$\mathcal{N}(\pi, \cap)$	3.9 ✗	5.6 ✓		3.24 ✗		3.24 ✗	3.23 ✓
$\mathcal{N}(\pi, \overline{\pi})$	3.9 ✗	3.34 ✓•			4.10 ✓	4.10 ✓	3.26 ✗
$\mathcal{N}(\pi, \overline{\pi}, \cap)$	3.9 ✗	5.6 ✓				4.10 ✓	3.26 ✗
$\mathcal{N}(\pi, \overline{\pi}, \cap, -)$	3.9 ✗	3.9 ✗					3.26 ✗
$\mathcal{N}(\frown)$	3.9 ✗		3.23 ✓	3.24 ✗	3.23 ✓	3.24 ✗	3.23 ✓
$\mathcal{N}(\frown, \pi)$	3.9 ✗			3.24 ✗	3.36 ✓•	3.24 ✗	3.23 ✓
$\mathcal{N}(\frown, \pi, \cap)$	3.9 ✗			3.24 ✗		3.24 ✗	3.23 ✓
$\mathcal{N}(\frown, \pi, \overline{\pi})$	3.9 ✗				5.6 ✓	3.9 ✗	3.26 ✗
$\mathcal{N}(\frown, \pi, \overline{\pi}, \cap)$	3.9 ✗					3.9 ✗	3.26 ✗
$\mathcal{N}(\frown, \pi, \overline{\pi}, \cap, -)$	3.15 ✗						3.26 ✗
$\mathcal{N}(\text{di})$		3.33 ✗	3.33 ✗	3.24 ✗	3.9 ✗	3.24 ✗	3.34 ✓•
$\mathcal{N}(\text{di}, \pi)$		3.34 ✓•		3.24 ✗	3.9 ✗	3.24 ✗	3.34 ✓•
$\mathcal{N}(\text{di}, \pi, \cap)$		3.33 ✗		3.24 ✗		3.24 ✗	3.26 ✗
$\mathcal{N}(\text{di}, \pi, \overline{\pi})$		3.34 ✓•			3.9 ✗	3.9 ✗	3.26 ✗
$\mathcal{N}(\text{di}, \pi, \overline{\pi}, \cap)$		3.33 ✗				?	3.26 ✗
$\mathcal{N}(\text{di}, \pi, \overline{\pi}, \cap, -)$		3.33 ✗					3.26 ✗
$\mathcal{N}(\text{di}, \frown)$			3.33 ✗	3.24 ✗	3.9 ✗	3.24 ✗	3.28 ✓
$\mathcal{N}(\text{di}, \frown, \pi)$				3.24 ✗	3.9 ✗	3.24 ✗	3.28 ✓
$\mathcal{N}(\text{di}, \frown, \pi, \cap)$				3.24 ✗		3.24 ✗	3.26 ✗
$\mathcal{N}(\text{di}, \frown, \pi, \overline{\pi})$					3.9 ✗	3.9 ✗	3.26 ✗
$\mathcal{N}(\text{di}, \frown, \pi, \overline{\pi}, \cap)$						?	3.26 ✗
$\mathcal{N}(\text{di}, \frown, \pi, \overline{\pi}, \cap, -)$							3.26 ✗

	di	$\frown$	π	$\overline{\pi}$	$\cap$	$-$	*
$\mathcal{N}(^*)$	3.9 ✗	3.23 ✓	3.23 ✓	3.24 ✗	4.10 ✓	4.10 ✓	
$\mathcal{N}(\cap, ^*)$	3.9 ✗	3.23 ✓	3.23 ✓	3.24 ✗		4.10 ✓	
$\mathcal{N}(\cap, -, ^*)$	3.24 ✗	3.24 ✗	3.24 ✗	3.24 ✗			
$\mathcal{N}(\pi, ^*)$	3.9 ✗	3.23 ✓		3.24 ✗	4.10 ✓	3.24 ✗	
$\mathcal{N}(\pi, \cap, ^*)$	3.9 ✗	3.23 ✓		3.24 ✗		3.24 ✗	
$\mathcal{N}(\pi, \overline{\pi}, ^*)$	3.9 ✗	?			4.10 ✓	4.10 ✓	
$\mathcal{N}(\pi, \overline{\pi}, \cap, ^*)$	3.9 ✗	?				4.10 ✓	
$\mathcal{N}(\pi, \overline{\pi}, \cap, -, ^*)$	3.9 ✗	3.9 ✗					
$\mathcal{N}(\frown, ^*)$	3.9 ✗		3.23 ✓	3.24 ✗	3.23 ✓	3.24 ✗	
$\mathcal{N}(\frown, \pi, ^*)$	3.9 ✗			3.24 ✗	3.23 ✓	3.24 ✗	
$\mathcal{N}(\frown, \pi, \cap, ^*)$	3.9 ✗			3.24 ✗		3.24 ✗	
$\mathcal{N}(\frown, \pi, \overline{\pi}, ^*)$	3.9 ✗				?	3.9 ✗	
$\mathcal{N}(\frown, \pi, \overline{\pi}, \cap, ^*)$	3.9 ✗					3.9 ✗	
$\mathcal{N}(\frown, \pi, \overline{\pi}, \cap, -, ^*)$	3.1 ✓						
$\mathcal{N}(\text{di}, ^*)$		3.33 ✗	3.33 ✗	3.24 ✗	3.9 ✗	3.24 ✗	
$\mathcal{N}(\text{di}, \pi, ^*)$		?		3.24 ✗	3.9 ✗	3.24 ✗	
$\mathcal{N}(\text{di}, \pi, \cap, ^*)$		3.33 ✗		3.24 ✗		3.24 ✗	
$\mathcal{N}(\text{di}, \pi, \overline{\pi}, ^*)$		?			3.9 ✗	3.9 ✗	
$\mathcal{N}(\text{di}, \pi, \overline{\pi}, \cap, ^*)$		3.33 ✗				?	
$\mathcal{N}(\text{di}, \pi, \overline{\pi}, \cap, -, ^*)$		3.33 ✗					
$\mathcal{N}(\text{di}, \frown, ^*)$			3.33 ✗	3.24 ✗	3.9 ✗	3.24 ✗	
$\mathcal{N}(\text{di}, \frown, \pi, ^*)$				3.24 ✗	3.9 ✗	3.24 ✗	
$\mathcal{N}(\text{di}, \frown, \pi, \cap, ^*)$				3.24 ✗		3.24 ✗	
$\mathcal{N}(\text{di}, \frown, \pi, \overline{\pi}, ^*)$					3.9 ✗	3.9 ✗	
$\mathcal{N}(\text{di}, \frown, \pi, \overline{\pi}, \cap, ^*)$						?	
$\mathcal{N}(\text{di}, \frown, \pi, \overline{\pi}, \cap, -, ^*)$							

	di	$\frown$	π	$\overline{\pi}$	$\cap$	$-$	*

A.4 Boolean semantics on labeled trees

	di	⌢	π	$\overline{\pi}$	$\cap$	$-$	*
$\mathcal{N}()$	3.9 ✗	3.5 ✗	3.5 ✗	3.24 ✗	4.10 ✓	4.10 ✓	3.26 ✗
$\mathcal{N}(\cap)$	3.9 ✗	3.5 ✗	3.5 ✗	3.24 ✗		4.10 ✓	3.26 ✗
$\mathcal{N}(\cap, -)$	3.24 ✗	3.24 ✗	3.24 ✗	3.24 ✗			3.26 ✗
$\mathcal{N}(\pi)$	3.9 ✗	3.34 ✓•		3.24 ✗	4.10 ✓	3.24 ✗	3.26 ✗
$\mathcal{N}(\pi, \cap)$	3.9 ✗	5.6 ✓		3.24 ✗		3.24 ✗	3.26 ✗
$\mathcal{N}(\pi, \overline{\pi})$	3.9 ✗	3.34 ✓•			4.10 ✓	4.10 ✓	3.26 ✗
$\mathcal{N}(\pi, \overline{\pi}, \cap)$	3.9 ✗	5.6 ✓				4.10 ✓	3.26 ✗
$\mathcal{N}(\pi, \overline{\pi}, \cap, -)$	3.9 ✗	3.9 ✗					3.26 ✗
$\mathcal{N}(⌢)$	3.9 ✗		5.9 ✓	3.24 ✗	5.9 ✓	3.24 ✗	3.26 ✗
$\mathcal{N}(⌢, \pi)$	3.9 ✗			3.24 ✗	3.36 ✓•	3.24 ✗	3.26 ✗
$\mathcal{N}(⌢, \pi, \cap)$	3.9 ✗			3.24 ✗		3.24 ✗	3.26 ✗
$\mathcal{N}(⌢, \pi, \overline{\pi})$	3.9 ✗				5.6 ✓	3.9 ✗	3.26 ✗
$\mathcal{N}(⌢, \pi, \overline{\pi}, \cap)$	3.9 ✗					3.9 ✗	3.26 ✗
$\mathcal{N}(⌢, \pi, \overline{\pi}, \cap, -)$	3.15 ✗						3.26 ✗
$\mathcal{N}(\text{di})$		3.33 ✗	3.33 ✗	3.24 ✗	3.9 ✗	3.24 ✗	3.26 ✗
$\mathcal{N}(\text{di}, \pi)$		3.34 ✓•		3.24 ✗	3.9 ✗	3.24 ✗	3.26 ✗
$\mathcal{N}(\text{di}, \pi, \cap)$		3.33 ✗		3.24 ✗		3.24 ✗	3.26 ✗
$\mathcal{N}(\text{di}, \pi, \overline{\pi})$		3.34 ✓•			3.9 ✗	3.9 ✗	3.26 ✗
$\mathcal{N}(\text{di}, \pi, \overline{\pi}, \cap)$		3.33 ✗				?	3.26 ✗
$\mathcal{N}(\text{di}, \pi, \overline{\pi}, \cap, -)$		3.33 ✗					3.26 ✗
$\mathcal{N}(\text{di}, ⌢)$			3.33 ✗	3.24 ✗	3.9 ✗	3.24 ✗	3.26 ✗
$\mathcal{N}(\text{di}, ⌢, \pi)$				3.24 ✗	3.9 ✗	3.24 ✗	3.26 ✗
$\mathcal{N}(\text{di}, ⌢, \pi, \cap)$				3.24 ✗		3.24 ✗	3.26 ✗
$\mathcal{N}(\text{di}, ⌢, \pi, \overline{\pi})$					3.9 ✗	3.9 ✗	3.26 ✗
$\mathcal{N}(\text{di}, ⌢, \pi, \overline{\pi}, \cap)$						?	3.26 ✗
$\mathcal{N}(\text{di}, ⌢, \pi, \overline{\pi}, \cap, -)$							3.26 ✗

	di	⌢	π	$\overline{\pi}$	$\cap$	$-$	*
$\mathcal{N}(^*)$	3.9 ✗	3.5 ✗	3.5 ✗	3.24 ✗	4.10 ✓	4.10 ✓	
$\mathcal{N}(\cap, ^*)$	3.9 ✗	3.5 ✗	3.5 ✗	3.24 ✗		4.10 ✓	
$\mathcal{N}(\cap, -, ^*)$	3.24 ✗	3.24 ✗	3.24 ✗	3.24 ✗			
$\mathcal{N}(\pi, ^*)$	3.9 ✗	?		3.24 ✗	4.10 ✓	3.24 ✗	
$\mathcal{N}(\pi, \cap, ^*)$	3.9 ✗	?		3.24 ✗		3.24 ✗	
$\mathcal{N}(\pi, \overline{\pi}, ^*)$	3.9 ✗	?			4.10 ✓	4.10 ✓	
$\mathcal{N}(\pi, \overline{\pi}, \cap, ^*)$	3.9 ✗	?				4.10 ✓	
$\mathcal{N}(\pi, \overline{\pi}, \cap, -, ^*)$	3.9 ✗	3.9 ✗					
$\mathcal{N}(⌢, ^*)$	3.9 ✗		?	3.24 ✗	?	3.24 ✗	
$\mathcal{N}(⌢, \pi, ^*)$	3.9 ✗			3.24 ✗	?	3.24 ✗	
$\mathcal{N}(⌢, \pi, \cap, ^*)$	3.9 ✗			3.24 ✗		3.24 ✗	
$\mathcal{N}(⌢, \pi, \overline{\pi}, ^*)$	3.9 ✗				?	3.9 ✗	
$\mathcal{N}(⌢, \pi, \overline{\pi}, \cap, ^*)$	3.9 ✗					3.9 ✗	
$\mathcal{N}(⌢, \pi, \overline{\pi}, \cap, -, ^*)$	3.1 ✓						
$\mathcal{N}(\text{di}, ^*)$		3.33 ✗	3.33 ✗	3.24 ✗	3.9 ✗	3.24 ✗	
$\mathcal{N}(\text{di}, \pi, ^*)$		?		3.24 ✗	3.9 ✗	3.24 ✗	
$\mathcal{N}(\text{di}, \pi, \cap, ^*)$		3.33 ✗		3.24 ✗		3.24 ✗	
$\mathcal{N}(\text{di}, \pi, \overline{\pi}, ^*)$		?			3.9 ✗	3.9 ✗	
$\mathcal{N}(\text{di}, \pi, \overline{\pi}, \cap, ^*)$		3.33 ✗				?	
$\mathcal{N}(\text{di}, \pi, \overline{\pi}, \cap, -, ^*)$		3.33 ✗					
$\mathcal{N}(\text{di}, ⌢, ^*)$			3.33 ✗	3.24 ✗	3.9 ✗	3.24 ✗	
$\mathcal{N}(\text{di}, ⌢, \pi, ^*)$				3.24 ✗	3.9 ✗	3.24 ✗	
$\mathcal{N}(\text{di}, ⌢, \pi, \cap, ^*)$				3.24 ✗		3.24 ✗	
$\mathcal{N}(\text{di}, ⌢, \pi, \overline{\pi}, ^*)$					3.9 ✗	3.9 ✗	
$\mathcal{N}(\text{di}, ⌢, \pi, \overline{\pi}, \cap, ^*)$						?	
$\mathcal{N}(\text{di}, ⌢, \pi, \overline{\pi}, \cap, -, ^*)$							

	di	⌢	π	$\overline{\pi}$	$\cap$	$-$	*

A.5 Path semantics on unlabeled chains

	di	$\frown$	π	$\overline{\pi}$	$\cap$	$-$	*
$\mathcal{N}()$	3.11 ✗	3.11 ✗	3.32 ✗	3.24 ✗	4.10 ✓	4.10 ✓	3.25 ✗
$\mathcal{N}(\cap)$	3.11 ✗	3.11 ✗	3.32 ✗	3.24 ✗		4.10 ✓	3.25 ✗
$\mathcal{N}(\cap, -)$	3.24 ✗	3.24 ✗	3.24 ✗	3.24 ✗			3.25 ✗
$\mathcal{N}(\pi)$	3.11 ✗	3.11 ✗		3.24 ✗	4.10 ✓	3.24 ✗	3.25 ✗
$\mathcal{N}(\pi, \cap)$	3.11 ✗	3.11 ✗		3.24 ✗		3.24 ✗	3.25 ✗
$\mathcal{N}(\pi, \overline{\pi})$	3.11 ✗	3.11 ✗			4.10 ✓	4.10 ✓	3.26 ✗
$\mathcal{N}(\pi, \overline{\pi}, \cap)$	3.11 ✗	3.11 ✗				4.10 ✓	3.26 ✗
$\mathcal{N}(\pi, \overline{\pi}, \cap, -)$	3.11 ✗	3.11 ✗					3.26 ✗
$\mathcal{N}(\frown)$	3.11 ✗		3.13 ✓	3.24 ✗	3.13 ✓	3.24 ✗	3.25 ✗
$\mathcal{N}(\frown, \pi)$	3.11 ✗			3.24 ✗	3.36 ✓•	3.24 ✗	3.25 ✗
$\mathcal{N}(\frown, \pi, \cap)$	3.11 ✗			3.24 ✗		3.24 ✗	3.25 ✗
$\mathcal{N}(\frown, \pi, \overline{\pi})$	3.11 ✗				5.6 ✓	5.13 ✓	3.26 ✗
$\mathcal{N}(\frown, \pi, \overline{\pi}, \cap)$	3.11 ✗					5.13 ✓	3.26 ✗
$\mathcal{N}(\frown, \pi, \overline{\pi}, \cap, -)$	3.11 ✗						3.26 ✗
$\mathcal{N}(\text{di})$		3.20 ✗	3.32 ✗	3.24 ✗	3.20 ✗	3.24 ✗	3.25 ✗
$\mathcal{N}(\text{di}, \pi)$		3.20 ✗		3.24 ✗	3.20 ✗	3.24 ✗	3.25 ✗
$\mathcal{N}(\text{di}, \pi, \cap)$		?		3.24 ✗		3.24 ✗	3.25 ✗
$\mathcal{N}(\text{di}, \pi, \overline{\pi})$		?			?	?	3.26 ✗
$\mathcal{N}(\text{di}, \pi, \overline{\pi}, \cap)$		?				?	3.26 ✗
$\mathcal{N}(\text{di}, \pi, \overline{\pi}, \cap, -)$		?					3.26 ✗
$\mathcal{N}(\text{di}, \frown)$			3.19 ✓	3.24 ✗	3.20 ✗	3.24 ✗	3.25 ✗
$\mathcal{N}(\text{di}, \frown, \pi)$				3.24 ✗	3.20 ✗	3.24 ✗	3.25 ✗
$\mathcal{N}(\text{di}, \frown, \pi, \cap)$				3.24 ✗		3.24 ✗	3.25 ✗
$\mathcal{N}(\text{di}, \frown, \pi, \overline{\pi})$					?	?	3.26 ✗
$\mathcal{N}(\text{di}, \frown, \pi, \overline{\pi}, \cap)$						?	3.26 ✗
$\mathcal{N}(\text{di}, \frown, \pi, \overline{\pi}, \cap, -)$							3.26 ✗

	di	$\frown$	π	$\overline{\pi}$	$\cap$	$-$	*
$\mathcal{N}(^*)$	3.11 ✗	3.11 ✗	3.32 ✗	3.24 ✗	4.10 ✓	4.10 ✓	
$\mathcal{N}(\cap, ^*)$	3.11 ✗	3.11 ✗	3.32 ✗	3.24 ✗		4.10 ✓	
$\mathcal{N}(\cap, -, ^*)$	3.24 ✗	3.24 ✗	3.24 ✗	3.24 ✗			
$\mathcal{N}(\pi, ^*)$	3.11 ✗	3.11 ✗		3.24 ✗	4.10 ✓	3.24 ✗	
$\mathcal{N}(\pi, \cap, ^*)$	3.11 ✗	3.11 ✗		3.24 ✗		3.24 ✗	
$\mathcal{N}(\pi, \overline{\pi}, ^*)$	3.11 ✗	3.11 ✗			4.10 ✓	4.10 ✓	
$\mathcal{N}(\pi, \overline{\pi}, \cap, ^*)$	3.11 ✗	3.11 ✗				4.10 ✓	
$\mathcal{N}(\pi, \overline{\pi}, \cap, -, ^*)$	3.11 ✗	3.11 ✗					
$\mathcal{N}(\frown, ^*)$	3.1 ✓		?	3.24 ✗	?	3.24 ✗	
$\mathcal{N}(\frown, \pi, ^*)$	3.1 ✓			3.24 ✗	?	3.24 ✗	
$\mathcal{N}(\frown, \pi, \cap, ^*)$	3.1 ✓			3.24 ✗		3.24 ✗	
$\mathcal{N}(\frown, \pi, \overline{\pi}, ^*)$	3.1 ✓				?	?	
$\mathcal{N}(\frown, \pi, \overline{\pi}, \cap, ^*)$	3.1 ✓					?	
$\mathcal{N}(\frown, \pi, \overline{\pi}, \cap, -, ^*)$	3.1 ✓						
$\mathcal{N}(\text{di}, ^*)$		3.20 ✗	3.32 ✗	3.24 ✗	?	3.24 ✗	
$\mathcal{N}(\text{di}, \pi, ^*)$		3.20 ✗		3.24 ✗	?	3.24 ✗	
$\mathcal{N}(\text{di}, \pi, \cap, ^*)$		?		3.24 ✗		3.24 ✗	
$\mathcal{N}(\text{di}, \pi, \overline{\pi}, ^*)$		?			?	?	
$\mathcal{N}(\text{di}, \pi, \overline{\pi}, \cap, ^*)$		?				?	
$\mathcal{N}(\text{di}, \pi, \overline{\pi}, \cap, -, ^*)$		3.1 ✓					
$\mathcal{N}(\text{di}, \frown, ^*)$			?	3.24 ✗	?	3.24 ✗	
$\mathcal{N}(\text{di}, \frown, \pi, ^*)$				3.24 ✗	?	3.24 ✗	
$\mathcal{N}(\text{di}, \frown, \pi, \cap, ^*)$				3.24 ✗		3.24 ✗	
$\mathcal{N}(\text{di}, \frown, \pi, \overline{\pi}, ^*)$					?	?	
$\mathcal{N}(\text{di}, \frown, \pi, \overline{\pi}, \cap, ^*)$						?	
$\mathcal{N}(\text{di}, \frown, \pi, \overline{\pi}, \cap, -, ^*)$							

	di	$\frown$	π	$\overline{\pi}$	$\cap$	$-$	*

A.6 Path semantics on labeled chains

	di	⌢	π	$\overline{\pi}$	$\cap$	–	*
$\mathcal{N}()$	3.11 ✗	3.11 ✗	3.32 ✗	3.24 ✗	4.10 ✓	4.10 ✓	3.25 ✗
$\mathcal{N}(\cap)$	3.11 ✗	3.11 ✗	3.32 ✗	3.24 ✗		4.10 ✓	3.25 ✗
$\mathcal{N}(\cap, -)$	3.24 ✗	3.24 ✗	3.24 ✗	3.24 ✗			3.25 ✗
$\mathcal{N}(\pi)$	3.11 ✗	3.11 ✗		3.24 ✗	4.10 ✓	3.24 ✗	3.25 ✗
$\mathcal{N}(\pi, \cap)$	3.11 ✗	3.11 ✗		3.24 ✗		3.24 ✗	3.25 ✗
$\mathcal{N}(\pi, \overline{\pi})$	3.11 ✗	3.11 ✗			4.10 ✓	4.10 ✓	3.26 ✗
$\mathcal{N}(\pi, \overline{\pi}, \cap)$	3.11 ✗	3.11 ✗				4.10 ✓	3.26 ✗
$\mathcal{N}(\pi, \overline{\pi}, \cap, -)$	3.11 ✗	3.11 ✗					3.26 ✗
$\mathcal{N}(⌢)$	3.11 ✗		3.13 ✓	3.24 ✗	3.13 ✓	3.24 ✗	3.25 ✗
$\mathcal{N}(⌢, \pi)$	3.11 ✗			3.24 ✗	3.36 ✓•	3.24 ✗	3.25 ✗
$\mathcal{N}(⌢, \pi, \cap)$	3.11 ✗			3.24 ✗		3.24 ✗	3.25 ✗
$\mathcal{N}(⌢, \pi, \overline{\pi})$	3.11 ✗				5.6 ✓	5.13 ✓	3.26 ✗
$\mathcal{N}(⌢, \pi, \overline{\pi}, \cap)$	3.11 ✗					5.13 ✓	3.26 ✗
$\mathcal{N}(⌢, \pi, \overline{\pi}, \cap, -)$	3.11 ✗						3.26 ✗
$\mathcal{N}(\text{di})$		3.20 ✗	3.32 ✗	3.24 ✗	3.20 ✗	3.24 ✗	3.25 ✗
$\mathcal{N}(\text{di}, \pi)$		3.20 ✗		3.24 ✗	3.20 ✗	3.24 ✗	3.25 ✗
$\mathcal{N}(\text{di}, \pi, \cap)$		?		3.24 ✗		3.24 ✗	3.25 ✗
$\mathcal{N}(\text{di}, \pi, \overline{\pi})$		?			3.31 ✗	3.31 ✗	3.26 ✗
$\mathcal{N}(\text{di}, \pi, \overline{\pi}, \cap)$		?				?	3.26 ✗
$\mathcal{N}(\text{di}, \pi, \overline{\pi}, \cap, -)$		?					3.26 ✗
$\mathcal{N}(\text{di}, ⌢)$			?	3.24 ✗	3.20 ✗	3.24 ✗	3.25 ✗
$\mathcal{N}(\text{di}, ⌢, \pi)$				3.24 ✗	3.20 ✗	3.24 ✗	3.25 ✗
$\mathcal{N}(\text{di}, ⌢, \pi, \cap)$				3.24 ✗		3.24 ✗	3.25 ✗
$\mathcal{N}(\text{di}, ⌢, \pi, \overline{\pi})$					3.31 ✗	3.31 ✗	3.26 ✗
$\mathcal{N}(\text{di}, ⌢, \pi, \overline{\pi}, \cap)$						?	3.26 ✗
$\mathcal{N}(\text{di}, ⌢, \pi, \overline{\pi}, \cap, -)$							3.26 ✗

	di	⌢	π	$\overline{\pi}$	$\cap$	–	*
$\mathcal{N}(^*)$	3.11 ✗	3.11 ✗	3.32 ✗	3.24 ✗	4.10 ✓	4.10 ✓	
$\mathcal{N}(\cap, ^*)$	3.11 ✗	3.11 ✗	3.32 ✗	3.24 ✗		4.10 ✓	
$\mathcal{N}(\cap, -, ^*)$	3.24 ✗	3.24 ✗	3.24 ✗	3.24 ✗			
$\mathcal{N}(\pi, ^*)$	3.11 ✗	3.11 ✗		3.24 ✗	4.10 ✓	3.24 ✗	
$\mathcal{N}(\pi, \cap, ^*)$	3.11 ✗	3.11 ✗		3.24 ✗		3.24 ✗	
$\mathcal{N}(\pi, \overline{\pi}, ^*)$	3.11 ✗	3.11 ✗			4.10 ✓	4.10 ✓	
$\mathcal{N}(\pi, \overline{\pi}, \cap, ^*)$	3.11 ✗	3.11 ✗				4.10 ✓	
$\mathcal{N}(\pi, \overline{\pi}, \cap, -, ^*)$	3.11 ✗	3.11 ✗					
$\mathcal{N}(⌢, ^*)$	3.1 ✓		?	3.24 ✗	?	3.24 ✗	
$\mathcal{N}(⌢, \pi, ^*)$	3.1 ✓			3.24 ✗	?	3.24 ✗	
$\mathcal{N}(⌢, \pi, \cap, ^*)$	3.1 ✓			3.24 ✗		3.24 ✗	
$\mathcal{N}(⌢, \pi, \overline{\pi}, ^*)$	3.1 ✓				?	?	
$\mathcal{N}(⌢, \pi, \overline{\pi}, \cap, ^*)$	3.1 ✓					?	
$\mathcal{N}(⌢, \pi, \overline{\pi}, \cap, -, ^*)$	3.1 ✓						
$\mathcal{N}(\text{di}, ^*)$		3.20 ✗	3.32 ✗	3.24 ✗	?	3.24 ✗	
$\mathcal{N}(\text{di}, \pi, ^*)$		3.20 ✗		3.24 ✗	?	3.24 ✗	
$\mathcal{N}(\text{di}, \pi, \cap, ^*)$		?		3.24 ✗		3.24 ✗	
$\mathcal{N}(\text{di}, \pi, \overline{\pi}, ^*)$		?			?	?	
$\mathcal{N}(\text{di}, \pi, \overline{\pi}, \cap, ^*)$		?				?	
$\mathcal{N}(\text{di}, \pi, \overline{\pi}, \cap, -, ^*)$		3.1 ✓					
$\mathcal{N}(\text{di}, ⌢, ^*)$			?	3.24 ✗	?	3.24 ✗	
$\mathcal{N}(\text{di}, ⌢, \pi, ^*)$				3.24 ✗	?	3.24 ✗	
$\mathcal{N}(\text{di}, ⌢, \pi, \cap, ^*)$				3.24 ✗		3.24 ✗	
$\mathcal{N}(\text{di}, ⌢, \pi, \overline{\pi}, ^*)$					?	?	
$\mathcal{N}(\text{di}, ⌢, \pi, \overline{\pi}, \cap, ^*)$						?	
$\mathcal{N}(\text{di}, ⌢, \pi, \overline{\pi}, \cap, -, ^*)$							

	di	⌢	π	$\overline{\pi}$	$\cap$	–	*

A.7 Path semantics on unlabeled trees

	di	$\frown$	π	$\overline{\pi}$	$\cap$	$-$	*
$\mathcal{N}()$	3.9 ✗	3.11 ✗	3.32 ✗	3.24 ✗	4.10 ✓	4.10 ✓	3.25 ✗
$\mathcal{N}(\cap)$	3.9 ✗	3.11 ✗	3.32 ✗	3.24 ✗		4.10 ✓	3.25 ✗
$\mathcal{N}(\cap, -)$	3.24 ✗	3.24 ✗	3.24 ✗	3.24 ✗			3.25 ✗
$\mathcal{N}(\pi)$	3.9 ✗	3.11 ✗		3.24 ✗	4.10 ✓	3.24 ✗	3.25 ✗
$\mathcal{N}(\pi, \cap)$	3.9 ✗	3.11 ✗		3.24 ✗		3.24 ✗	3.25 ✗
$\mathcal{N}(\pi, \overline{\pi})$	3.9 ✗	3.11 ✗			4.10 ✓	4.10 ✓	3.26 ✗
$\mathcal{N}(\pi, \overline{\pi}, \cap)$	3.9 ✗	3.11 ✗				4.10 ✓	3.26 ✗
$\mathcal{N}(\pi, \overline{\pi}, \cap, -)$	3.9 ✗	3.9 ✗					3.26 ✗
$\mathcal{N}(\frown)$	3.9 ✗		3.32 ✗	3.24 ✗	3.32 ✗	3.24 ✗	3.25 ✗
$\mathcal{N}(\frown, \pi)$	3.9 ✗			3.24 ✗	3.36 ✓•	3.24 ✗	3.25 ✗
$\mathcal{N}(\frown, \pi, \cap)$	3.9 ✗			3.24 ✗		3.24 ✗	3.25 ✗
$\mathcal{N}(\frown, \pi, \overline{\pi})$	3.9 ✗				5.6 ✓	3.9 ✗	3.26 ✗
$\mathcal{N}(\frown, \pi, \overline{\pi}, \cap)$	3.9 ✗					3.9 ✗	3.26 ✗
$\mathcal{N}(\frown, \pi, \overline{\pi}, \cap, -)$	3.15 ✗						3.26 ✗
$\mathcal{N}(\text{di})$		3.33 ✗	3.33 ✗	3.24 ✗	3.9 ✗	3.24 ✗	3.25 ✗
$\mathcal{N}(\text{di}, \pi)$		3.20 ✗		3.24 ✗	3.9 ✗	3.24 ✗	3.25 ✗
$\mathcal{N}(\text{di}, \pi, \cap)$		3.33 ✗		3.24 ✗		3.24 ✗	3.26 ✗
$\mathcal{N}(\text{di}, \pi, \overline{\pi})$		3.32 ✗			3.9 ✗	3.9 ✗	3.26 ✗
$\mathcal{N}(\text{di}, \pi, \overline{\pi}, \cap)$		3.33 ✗				?	3.26 ✗
$\mathcal{N}(\text{di}, \pi, \overline{\pi}, \cap, -)$		3.33 ✗					3.26 ✗
$\mathcal{N}(\text{di}, \frown)$			3.33 ✗	3.24 ✗	3.9 ✗	3.24 ✗	3.25 ✗
$\mathcal{N}(\text{di}, \frown, \pi)$				3.24 ✗	3.9 ✗	3.24 ✗	3.25 ✗
$\mathcal{N}(\text{di}, \frown, \pi, \cap)$				3.24 ✗		3.24 ✗	3.26 ✗
$\mathcal{N}(\text{di}, \frown, \pi, \overline{\pi})$					3.9 ✗	3.9 ✗	3.26 ✗
$\mathcal{N}(\text{di}, \frown, \pi, \overline{\pi}, \cap)$						?	3.26 ✗
$\mathcal{N}(\text{di}, \frown, \pi, \overline{\pi}, \cap, -)$							3.26 ✗

	di	$\frown$	π	$\overline{\pi}$	$\cap$	$-$	*
$\mathcal{N}(^*)$	3.9 ✗	3.11 ✗	3.32 ✗	3.24 ✗	4.10 ✓	4.10 ✓	
$\mathcal{N}(\cap, ^*)$	3.9 ✗	3.11 ✗	3.32 ✗	3.24 ✗		4.10 ✓	
$\mathcal{N}(\cap, -, ^*)$	3.24 ✗	3.24 ✗	3.24 ✗	3.24 ✗			
$\mathcal{N}(\pi, ^*)$	3.9 ✗	3.11 ✗		3.24 ✗	4.10 ✓	3.24 ✗	
$\mathcal{N}(\pi, \cap, ^*)$	3.9 ✗	3.11 ✗		3.24 ✗		3.24 ✗	
$\mathcal{N}(\pi, \overline{\pi}, ^*)$	3.9 ✗	3.11 ✗			4.10 ✓	4.10 ✓	
$\mathcal{N}(\pi, \overline{\pi}, \cap, ^*)$	3.9 ✗	3.11 ✗				4.10 ✓	
$\mathcal{N}(\pi, \overline{\pi}, \cap, -, ^*)$	3.9 ✗	3.9 ✗					
$\mathcal{N}(\frown, ^*)$	3.9 ✗		3.32 ✗	3.24 ✗	3.32 ✗	3.24 ✗	
$\mathcal{N}(\frown, \pi, ^*)$	3.9 ✗			3.24 ✗	?	3.24 ✗	
$\mathcal{N}(\frown, \pi, \cap, ^*)$	3.9 ✗			3.24 ✗		3.24 ✗	
$\mathcal{N}(\frown, \pi, \overline{\pi}, ^*)$	3.9 ✗				?	3.9 ✗	
$\mathcal{N}(\frown, \pi, \overline{\pi}, \cap, ^*)$	3.9 ✗					3.9 ✗	
$\mathcal{N}(\frown, \pi, \overline{\pi}, \cap, -, ^*)$	3.1 ✓						
$\mathcal{N}(\text{di}, ^*)$		3.33 ✗	3.33 ✗	3.24 ✗	3.9 ✗	3.24 ✗	
$\mathcal{N}(\text{di}, \pi, ^*)$		3.20 ✗		3.24 ✗	3.9 ✗	3.24 ✗	
$\mathcal{N}(\text{di}, \pi, \cap, ^*)$		3.33 ✗		3.24 ✗		3.24 ✗	
$\mathcal{N}(\text{di}, \pi, \overline{\pi}, ^*)$		3.32 ✗			3.9 ✗	3.9 ✗	
$\mathcal{N}(\text{di}, \pi, \overline{\pi}, \cap, ^*)$		3.33 ✗				?	
$\mathcal{N}(\text{di}, \pi, \overline{\pi}, \cap, -, ^*)$		3.33 ✗					
$\mathcal{N}(\text{di}, \frown, ^*)$			3.33 ✗	3.24 ✗	3.9 ✗	3.24 ✗	
$\mathcal{N}(\text{di}, \frown, \pi, ^*)$				3.24 ✗	3.9 ✗	3.24 ✗	
$\mathcal{N}(\text{di}, \frown, \pi, \cap, ^*)$				3.24 ✗		3.24 ✗	
$\mathcal{N}(\text{di}, \frown, \pi, \overline{\pi}, ^*)$					3.9 ✗	3.9 ✗	
$\mathcal{N}(\text{di}, \frown, \pi, \overline{\pi}, \cap, ^*)$						?	
$\mathcal{N}(\text{di}, \frown, \pi, \overline{\pi}, \cap, -, ^*)$							

	di	$\frown$	π	$\overline{\pi}$	$\cap$	$-$	*

A.8 Path semantics on labeled trees

	di	$\frown$	π	$\overline{\pi}$	$\cap$	$-$	*
$\mathcal{N}()$	3.9 ✗	3.35 ✗•	3.35 ✗•	3.24 ✗	4.10 ✓	4.10 ✓	3.35 ✗•
$\mathcal{N}(\cap)$	3.9 ✗	3.5 ✗	3.5 ✗	3.24 ✗		4.10 ✓	3.26 ✗
$\mathcal{N}(\cap, -)$	3.24 ✗	3.24 ✗	3.24 ✗	3.24 ✗			3.26 ✗
$\mathcal{N}(\pi)$	3.9 ✗	3.35 ✗•		3.24 ✗	4.10 ✓	3.24 ✗	3.35 ✗•
$\mathcal{N}(\pi, \cap)$	3.9 ✗	3.11 ✗		3.24 ✗		3.24 ✗	3.26 ✗
$\mathcal{N}(\pi, \overline{\pi})$	3.9 ✗	3.11 ✗			4.10 ✓	4.10 ✓	3.26 ✗
$\mathcal{N}(\pi, \overline{\pi}, \cap)$	3.9 ✗	3.11 ✗				4.10 ✓	3.26 ✗
$\mathcal{N}(\pi, \overline{\pi}, \cap, -)$	3.9 ✗	3.9 ✗					3.26 ✗
$\mathcal{N}(\frown)$	3.9 ✗		3.35 ✗•	3.24 ✗	3.32 ✗	3.24 ✗	3.35 ✗•
$\mathcal{N}(\frown, \pi)$	3.9 ✗			3.24 ✗	3.36 ✓•	3.24 ✗	3.35 ✗•
$\mathcal{N}(\frown, \pi, \cap)$	3.9 ✗			3.24 ✗		3.24 ✗	3.26 ✗
$\mathcal{N}(\frown, \pi, \overline{\pi})$	3.9 ✗				5.6 ✓	3.9 ✗	3.26 ✗
$\mathcal{N}(\frown, \pi, \overline{\pi}, \cap)$	3.9 ✗					3.9 ✗	3.26 ✗
$\mathcal{N}(\frown, \pi, \overline{\pi}, \cap, -)$	3.15 ✗						3.26 ✗
$\mathcal{N}(\text{di})$		3.33 ✗	3.33 ✗	3.24 ✗	3.9 ✗	3.24 ✗	3.26 ✗
$\mathcal{N}(\text{di}, \pi)$		3.20 ✗		3.24 ✗	3.9 ✗	3.24 ✗	3.26 ✗
$\mathcal{N}(\text{di}, \pi, \cap)$		3.33 ✗		3.24 ✗		3.24 ✗	3.26 ✗
$\mathcal{N}(\text{di}, \pi, \overline{\pi})$		3.32 ✗			3.9 ✗	3.9 ✗	3.26 ✗
$\mathcal{N}(\text{di}, \pi, \overline{\pi}, \cap)$		3.33 ✗				?	3.26 ✗
$\mathcal{N}(\text{di}, \pi, \overline{\pi}, \cap, -)$		3.33 ✗					3.26 ✗
$\mathcal{N}(\text{di}, \frown)$			3.33 ✗	3.24 ✗	3.9 ✗	3.24 ✗	3.26 ✗
$\mathcal{N}(\text{di}, \frown, \pi)$				3.24 ✗	3.9 ✗	3.24 ✗	3.26 ✗
$\mathcal{N}(\text{di}, \frown, \pi, \cap)$				3.24 ✗		3.24 ✗	3.26 ✗
$\mathcal{N}(\text{di}, \frown, \pi, \overline{\pi})$					3.9 ✗	3.9 ✗	3.26 ✗
$\mathcal{N}(\text{di}, \frown, \pi, \overline{\pi}, \cap)$						?	3.26 ✗
$\mathcal{N}(\text{di}, \frown, \pi, \overline{\pi}, \cap, -)$							3.26 ✗

	di	$\frown$	π	$\overline{\pi}$	$\cap$	$-$	*
$\mathcal{N}(^*)$	3.9 ✗	3.35 ✗•	3.35 ✗•	3.24 ✗	4.10 ✓	4.10 ✓	
$\mathcal{N}(\cap, ^*)$	3.9 ✗	3.5 ✗	3.5 ✗	3.24 ✗		4.10 ✓	
$\mathcal{N}(\cap, -, ^*)$	3.24 ✗	3.24 ✗	3.24 ✗	3.24 ✗			
$\mathcal{N}(\pi, ^*)$	3.9 ✗	3.35 ✗•		3.24 ✗	4.10 ✓	3.24 ✗	
$\mathcal{N}(\pi, \cap, ^*)$	3.9 ✗	3.11 ✗		3.24 ✗		3.24 ✗	
$\mathcal{N}(\pi, \overline{\pi}, ^*)$	3.9 ✗	3.11 ✗			4.10 ✓	4.10 ✓	
$\mathcal{N}(\pi, \overline{\pi}, \cap, ^*)$	3.9 ✗	3.11 ✗				4.10 ✓	
$\mathcal{N}(\pi, \overline{\pi}, \cap, -, ^*)$	3.9 ✗	3.9 ✗					
$\mathcal{N}(\frown, ^*)$	3.9 ✗		3.35 ✗•	3.24 ✗	3.32 ✗	3.24 ✗	
$\mathcal{N}(\frown, \pi, ^*)$	3.9 ✗			3.24 ✗	?	3.24 ✗	
$\mathcal{N}(\frown, \pi, \cap, ^*)$	3.9 ✗			3.24 ✗		3.24 ✗	
$\mathcal{N}(\frown, \pi, \overline{\pi}, ^*)$	3.9 ✗				?	3.9 ✗	
$\mathcal{N}(\frown, \pi, \overline{\pi}, \cap, ^*)$	3.9 ✗					3.9 ✗	
$\mathcal{N}(\frown, \pi, \overline{\pi}, \cap, -, ^*)$	3.1 ✓						
$\mathcal{N}(\text{di}, ^*)$		3.33 ✗	3.33 ✗	3.24 ✗	3.9 ✗	3.24 ✗	
$\mathcal{N}(\text{di}, \pi, ^*)$		3.20 ✗		3.24 ✗	3.9 ✗	3.24 ✗	
$\mathcal{N}(\text{di}, \pi, \cap, ^*)$		3.33 ✗		3.24 ✗		3.24 ✗	
$\mathcal{N}(\text{di}, \pi, \overline{\pi}, ^*)$		3.32 ✗			3.9 ✗	3.9 ✗	
$\mathcal{N}(\text{di}, \pi, \overline{\pi}, \cap, ^*)$		3.33 ✗				?	
$\mathcal{N}(\text{di}, \pi, \overline{\pi}, \cap, -, ^*)$		3.33 ✗					
$\mathcal{N}(\text{di}, \frown, ^*)$			3.33 ✗	3.24 ✗	3.9 ✗	3.24 ✗	
$\mathcal{N}(\text{di}, \frown, \pi, ^*)$				3.24 ✗	3.9 ✗	3.24 ✗	
$\mathcal{N}(\text{di}, \frown, \pi, \cap, ^*)$				3.24 ✗		3.24 ✗	
$\mathcal{N}(\text{di}, \frown, \pi, \overline{\pi}, ^*)$					3.9 ✗	3.9 ✗	
$\mathcal{N}(\text{di}, \frown, \pi, \overline{\pi}, \cap, ^*)$						?	
$\mathcal{N}(\text{di}, \frown, \pi, \overline{\pi}, \cap, -, ^*)$							

	di	$\frown$	π	$\overline{\pi}$	$\cap$	$-$	*