Containment of Queries for Graphs with Data

Egor V. Kostylev

University of Oxford

Juan L. Reutter

Pontificia Universidad Católica de Chile and Center for Semantic Web Research

Domagoj Vrgoč

Pontificia Universidad Católica de Chile and Center for Semantic Web Research

Abstract

We consider the containment problem for regular queries with memory and regular queries with data tests: two recently proposed query languages for graph databases that, in addition to allowing the user to ask topological queries, also track how the data changes along paths connecting various points in the database. Our results show that the problem is undecidable in general. However, by allowing only positive data comparisons we find natural fragments with better static analysis properties: the containment problem is PSPACE-complete in the case of regular queries with data tests and EXPSPACE-complete in the case of regular queries with memory.

Keywords: Graph databases, query languages, data values, query containment

1. Introduction

Managing graph-structured data is one of the most active topics in the database community these days. Although first introduced in the eighties [12, 13], the model has recently gained popularity due to a high demand from services that find the relational model too restrictive, such as social networks, Semantic Web, crime detection networks, biological databases and many others. There are several vendors offering graph database systems [14, 16, 27] and a growing body of literature on the subject (for a survey see, e.g., [2, 3, 33]).

In such applications data is usually modelled as a graph, with each node describing one entity in the database, for example a user in a social network, and

the edges of the graph representing various connections between nodes, such as friends in a social network, supervisor connections in a database modelling the structure of a company, etc. Nodes can have various types of connections, so usually each edge in the graph is labelled. Finally, nodes by themselves contain the actual data, modelled as traditional relational data with values coming from an infinite domain [2].

To query graph-structured data, one can, of course, use traditional relational languages and treat the model as a relational database. What makes graph databases attractive in modern applications is the ability to query intricate navigational patterns between objects, thus obtaining more information about the *topology* of the stored data and how it relates to the actual data. Earliest graph query languages, such as regular path queries (RPQs) [13] and conjunctive regular path queries (CRPQs) [9, 12], concentrate on retrieving the topology of the graph and ignore the actual data stored. These languages have been well studied in last decades, and many extensions were defined for them, such as 2-way RPQs [9], which allow backward navigation; nested regular expressions [6], which allow existential tests; or extended CRPQs [4], which allow checks of nontrivial relations amongst paths. Industry is also taking account of navigational languages. For example, RPQs have been added to SPARQL, a query language for Semantic Web graph databases [18], as a primitive for querying navigational properties of graphs.

But purely navigational languages such as RPQs or CRPQs cannot reason on the data stored in the nodes. Thus such data was usually queried using relational languages, without a way of specifying the interplay between the data stored and various navigational patterns connecting the data.

This interplay is indeed a requirement in many applications using graph-structured data. For example, in a database modelling the inner workings of a company one might be interested in finding chains of people living in the same city that are connected by professional links, or in a social network one could look for a sequence of friends, all of which like the same type of music. Recently, several languages that can handle such queries have been proposed [23, 24, 26] and they were all built on the idea of extending RPQs, or some variation thereof, with the ability to reason about data values that appear along the navigated path.

Our goal is to study static analysis aspects of this new generation of graph query languages, understanding them as basic building blocks for more complex navigational languages. We concentrate on the query containment problem, which is the problem of deciding, given two queries in some graph language, whether the answer set of the first query is contained in the answer set of the second one.

Deciding query containment is a fundamental problem in database theory, and is relevant to several complex database tasks such as data integration [22], query optimisation [1], view definition and maintenance [17], and query answering using views [11].

The importance of this problem motivated sustained research for relational query languages (see, e.g., [1]), XML query languages (see e.g. [29]) and even extensions of RPQs and other graph query languages [4, 5, 9, 15, 20]. The overall conclusion is that containment is generally undecidable for first order logic and other similar formalisms (see, e.g., [1]), but becomes decidable if we restrict to queries with little or no negation. For example, containment of conjunctive queries is NP-complete, while containment of RPQs, 2-way RPQs and nested regular expressions is PSPACE-complete. For CRPQs it jumps to EXPSPACE-complete.

While much is known about the containment problem for the above mentioned classes of queries, no detailed study has been conducted for query languages that deal both with navigational and data aspects of graph databases. Therefore, in this work we concentrate on the containment problem for languages which mix topological properties and data values. Namely, we consider *regular queries with memory* (or *RQMs* for short) and *regular queries with data tests* (or *RQDs*), both introduced in [26]. We primarily concentrate on containment, but the techniques presented here can easily be adapted to deal with other similar problems, such as satisfiability or equivalence of queries.

The intuition behind RQMs is that one can navigate through a graph in the same way as with RPQs, but along the path it is also possible to store a data value into a register and compare it with another value encountered later on the path. This idea is very similar to the one of register automata [19, 28] and in fact one can show that these two formalisms are equivalent [25]. RQDs operate in a similar fashion, but storing and comparing values adheres to a more strict stack-like discipline, so they enjoy much better evaluation properties.

Contributions By using equivalence of RQMs with register automata, we obtain our first result: the problem of checking whether one RQM is contained in another RQM is undecidable. This, of course, opens up the question of fragments of the language that do have decidable containment problem. The class of *positive RQMs* is one of such fragments, in which we allow testing if two data values are equal, but not if they are different. We show that the problem of positive RQM query containment is decidable and, in fact, EXPSPACE-complete.

Next we move onto the class of RQDs, which was shown to be strictly contained in the class of RQMs [26]. The imposed restrictions to RQMs are quite

heavy, and computational complexity of query evaluation drops by almost one exponent when we consider RQDs instead of RQMs [26]. For this reason one may expect the containment problem to be decidable for RQDs. On the contrary, as we show, it remains undecidable even in this restricted scenario. However, this changes once again when we consider *positive RQDs*, for which a PSPACE algorithm for testing containment is obtained. This is the best possible bound for any extension of RPQs, since their containment is already PSPACE-hard [10].

Overall, we see that when containment is considered, the situation is quite different for languages handling both topology and data than it is for traditional graph languages allowing only navigational queries. While for the latter containment is generally decidable, we show that for the languages considered here the problem resembles behaviour of relational algebra, where containment is undecidable for the full language, but various restrictions on the use of negation lead to decidable fragments.

Organization In Section 2 we formally define the data model and the problem studied. In Section 3 we introduce RQMs and the model of register automata used throughout this paper and then study their containment problem. We do the same for RQDs and the corresponding automata in Section 4. We conclude with some remarks about future work in Section 5.

Remark Some of the results of this paper have been announced previously in [21] where they appeared without proofs. In this extended version we provide the missing proofs and substantial new material, including the definition an analysis of the automata versions for RQMs and RQDs (the latter being a novel class of register automata), and several additional examples.

2. Preliminaries

Data graphs Let Σ be a finite alphabet of *labels* and $\mathcal{D}$ an infinite set of *data values*. A *data graph* over labels Σ and data values $\mathcal{D}$ is a triple $\langle V, E, \rho \rangle$, where

- V is a finite set of nodes,
- $E \subseteq V \times \Sigma \times V$ is a set of labelled edges, and
- $\rho : V \to \mathcal{D}$ is a function that assigns a data value to each node in the graph.

An example of a data graph is shown in Figure 1. If data values are not important, we disregard ρ and only talk about *graphs* $\langle V, E \rangle$ over Σ.

Regarding data values, this paper follows [23, 26] and the standard convention for data trees (as a model for XML), and assumes that data values are attached to

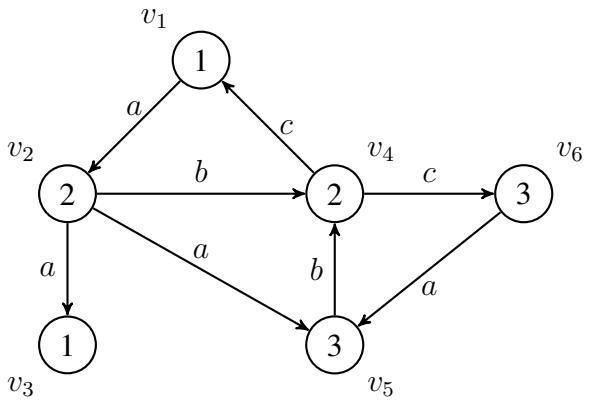

Figure 1: A data graph over labels $\{a, b, c\}$ and natural numbers as data values, in which nodes are v_i, $1 \leq i \leq 6$.

nodes. There are several ways to attach data values to graphs, such as in the edges or both in nodes and edges, but they are essentially equivalent [32].

We also assume that each node is assigned a single data value. This is not a real restriction, since a node with data values of several attributes can be modelled in our formalism as follows: for each attribute, the node is connected to a dedicated auxiliary node by an edge labelled with the attribute name, and the data value of the auxiliary node is the value of the attribute; moreover, in order to allow collecting multiple attribute values on a single path, each such auxiliary node is connected back to the original node by an edge with a special label $back$.

Paths A *path* between nodes v_1 and v_n in a graph $\langle V, E \rangle$ is a sequence

$$v_1 a_1 v_2 a_2 v_3 \ldots v_{n-1} a_{n-1} v_n,$$

such that each (v_i, a_i, v_{i+1}), for $1 \leq i < n$, is an edge in E. The *label* of the path is the word $a_1 \ldots a_{n-1}$ obtained by reading the edge labels appearing along this path.

Queries The default core of any query language for graphs is *regular path queries* (or *RPQs*), which are just regular languages over labels Σ, usually defined by regular expressions. The *evaluation* $[\![e]\!]^{\mathbb{G}}$ of an RPQ e over a graph $\mathbb{G}$, is the set of all pairs (v, v') of nodes in $\mathbb{G}$ such that there exists a path from v to v' with the label from the language of e.

There are a number of extensions of RPQs proposed in the literature. In this paper we concentrate on those that are capable of dealing with data values,

namely, RQMs and RQDs, introduced in the following sections. All queries in these classes are binary, that is, their evaluations (i.e., answers) are sets of pairs of nodes. Same as for RPQs, we denote by $[\![e]\!]^{\mathbb{G}}$ the evaluation of such a query e over a data graph $\mathbb{G}$.

Containment and Equivalence A query e is *contained* in a query e', written $e \subseteq e'$, if for each data graph $\mathbb{G}$ over labels Σ and data values $\mathcal{D}$ we have that

$$[\![e]\!]^{\mathbb{G}} \subseteq [\![e']\!]^{\mathbb{G}}.$$

Queries e and e' are *equivalent*, written $e \equiv e'$, if and only if $[\![e]\!]^{\mathbb{G}} = [\![e']\!]^{\mathbb{G}}$ for every $\mathbb{G}$.

The containment and equivalence are at the core of many static analysis tasks, such as query optimisation. All the classes of queries considered in this paper are closed under union, so these two problems are easily interreducible: $e \equiv e'$ if and only if e and e' contain each other, and $e \subseteq e'$ if and only if $e \cup e' \equiv e'$. This is why we concentrate just on the first problem, that is, consider the following decision problem parametrised by a class of queries $\mathcal{Q}$.

CONTAINMENT($\mathcal{Q}$)	
Input:	Queries e and e' from $\mathcal{Q}$.
Question:	Is e contained in e'?

The semantics of RPQs is defined for graphs, but it is straightforward to see that, for any two RPQs e and e', we have that $e \subseteq e'$ if and only if $\mathcal{L}(e) \subseteq \mathcal{L}(e')$, where $\mathcal{L}(e)$ and $\mathcal{L}(e')$ are the word languages accepted by regular expressions e and e', respectively, [10]. From this fact we obtain that containment of RPQs is PSPACE-complete, following the classic result that containment of regular expressions is PSPACE-complete. Since all of the classes of queries studied in this paper are extensions of RPQs, this establishes a lower bound for containment of any of these classes.

3. Regular Queries with Memory

Regular queries with memory, or *RQMs* for short, were introduced in [26] (where they were called *regular expressions with memory*) as a formalism for querying data graphs that allows data comparisons while navigating through the structure of the graph. They are based on *register automata*, an extension of finite-state automata for words over infinite alphabets.

The idea of RQMs is the following. They can store data values in a number of named *registers*, while parsing the input graph according to a specified regular navigation pattern. Also, they can compare the current data value with values that had previously been stored. An example of an RQM is the expression $\downarrow x.a^+[x^=]$, which returns all pairs (v_1, v_2) of nodes in a graph that have the same data value and are connected by a path labelled only with a's. Intuitively the expression works as follows: it first stores the data value of the node v_1 into the register x, and, after navigating an a-labelled path, it checks that the node v_2 at the end of this path has the same data value as the first node. This check is done via the test $x^=$, which makes sure that the data value of v_2 is the same as the one stored in register x.

The proposal of RQMs as a formalism for querying data graphs was motivated not only by their ability to handle data values, but also by the low computational complexity of their evaluation: it is PSPACE-complete in general, and NLOGSPACE-complete if the query is fixed (i.e., in *data complexity*) [26]. Hence, in the general case it is the same as for first-order or relational algebra queries, and, while slightly higher than for the latter two, the data complexity is still reasonable.

3.1. Syntax and Semantics of RQMs

Let X be a set of *registers*. A *condition* over X is a positive Boolean combination of atoms of the form $x^=$ or $x^{\neq}$, for $x \in X$.

Definition 3.1. *A* regular query with memory *(or* RQM*) over an alphabet of labels Σ and a set of registers X is an expression satisfying the grammar*

$$e \;:=\; \varepsilon \mid a \mid e \cup e \mid e \cdot e \mid e^+ \mid e[c] \mid \downarrow x.e$$

where ε is the empty word, a ranges over labels, x over registers, and c over conditions.

Before formally defining the semantics, let us give some examples of RQMs and explain their intuitive meaning.

Example 3.2.

1. The RQM $\downarrow x.(a[x^=])^+$ returns all pairs of nodes connected by a path, along which all the edges are labelled a and all data values are equal. The evaluation starts with $\downarrow x$, which stores the first data value into register x. The subexpression $(a[x^=])^+$ then checks that each subsequent label along the path is a, and that the data value of each node on this path is equal to the one of the first

$$
\begin{array}{lcl}
\mathcal{H}^{\mathbb{G}}(\varepsilon) & = & \{(s,s) \mid s \text{ is a state}\} \\
\mathcal{H}^{\mathbb{G}}(a) & = & \{([v_1,\lambda],[v_2,\lambda]) \mid (v_1,a,v_2) \in E\} \\
\mathcal{H}^{\mathbb{G}}(e_1 \cup e_2) & = & \mathcal{H}^{\mathbb{G}}(e_1) \cup \mathcal{H}^{\mathbb{G}}(e_2) \\
\mathcal{H}^{\mathbb{G}}(e_1 \cdot e_2) & = & \mathcal{H}^{\mathbb{G}}(e_1) \bullet \mathcal{H}^{\mathbb{G}}(e_2) \\
\mathcal{H}^{\mathbb{G}}(e^+) & = & \mathcal{H}^{\mathbb{G}}(e) \cup (\mathcal{H}^{\mathbb{G}}(e) \bullet \mathcal{H}^{\mathbb{G}}(e)) \cup (\mathcal{H}^{\mathbb{G}}(e) \bullet \mathcal{H}^{\mathbb{G}}(e) \bullet \mathcal{H}^{\mathbb{G}}(e)) \cup \ldots \\
\mathcal{H}^{\mathbb{G}}(e[c]) & = & \{([v_1,\lambda_1],[v_2,\lambda_2]) \mid \\
 & & \qquad ([v_1,\lambda_1],[v_2,\lambda_2]) \in \mathcal{H}^{\mathbb{G}}(e) \text{ and } \rho(v_2),\lambda_2 \models c\} \\
\mathcal{H}^{\mathbb{G}}(\downarrow x.e) & = & \{([v_1,\lambda_1],[v_2,\lambda_2]) \mid \\
 & & \qquad ([v_1,\lambda_1],[v_2,\lambda_2]) \in \mathcal{H}^{\mathbb{G}}(e) \text{ and } \lambda_1(x) = \rho(v_1)\}
\end{array}
$$

Table 1: Definition of function $\mathcal{H}^{\mathbb{G}}$ with respect to a data graph $\mathbb{G}$.

node (this is done by comparison with the value stored in the register x). The fact that this subexpression has $^+$ indicates that the sequence of checks is of arbitrary none-zero length.

2. The RQM $\downarrow x.(a[x^{\neq}])^+$ returns all pairs of nodes connected by a path where all edges are labelled with a and the first data value is different from all other data values. It works analogously as the expression above, except that it checks for inequality.

3. The RQM $\downarrow x.(a \cdot b)^+[x^{\neq}]$ returns all pairs of nodes connected by a path, whose label is of the form $ab \cdots ab$, and the first data value is different from the last. Note that the order of $^+$ and the condition check is different from the previous examples: the condition is checked only once, after verifying that the label is in $(a \cdot b)^+$, that is, at the end of the path. □

To define what it means for a data value to satisfy a condition we need the following notion. An *assignment* of registers X is a partial function from X to the set of data values $\mathcal{D}$. We will also write *Image*(λ) for the range of an assignment λ. Intuitively, an assignment models the current state of the registers at some point of the computation, with some registers containing stored data values, and some still being empty. Formally, a data value d and an assignment λ *satisfy* a condition $x^{=}$ (or $x^{\neq}$), written $d, \lambda \models x^{=}$ (or, $d, \lambda \models x^{\neq}$, respectively), if and only if $\lambda(x)$ is defined and $d = \lambda(x)$ (or $d \neq \lambda(x)$, respectively). This satisfaction relation is extended to general conditions in the usual way.

Given a data graph $\mathbb{G}$ and a set of registers X, a *state* is a pair consisting of a node of $\mathbb{G}$ and an assignment of X.

The semantics of RQMs over a data graph $\mathbb{G} = \langle V, E, \rho \rangle$ is defined in terms of a function $\mathcal{H}^{\mathbb{G}}$, which binds each RQM with a set of pairs of states in the

graph. The intuition of the set $\mathcal{H}^{\mathbb{G}}(e)$, for some RQM e, is as follows. Given states $s_1 = [v_1, \lambda_1]$ and $s_2 = [v_2, \lambda_2]$, the pair (s_1, s_2) is in $\mathcal{H}^{\mathbb{G}}(e)$ if there exists a path from v_1 to v_2, such that expression e can parse this path assuming that the registers are initialized according to λ_1, modified and tested as dictated by e, and the resulting assignment after traversing the path is λ_2.

Formally, given a data graph $\mathbb{G} = \langle V, E, \rho \rangle$, the function $\mathcal{H}^{\mathbb{G}}$ is constructed by the inductive definition in Table 1. The symbol $\bullet$ in the table refers to the usual (left) composition of binary relations:

$$\mathcal{H}^{\mathbb{G}}(e_1) \bullet \mathcal{H}^{\mathbb{G}}(e_2) = \\ \{(s_1, s_2) \mid \exists s \text{ such that } (s_1, s) \in \mathcal{H}^{\mathbb{G}}(e_1) \text{ and } (s, s_2) \in \mathcal{H}^{\mathbb{G}}(e_2)\}.$$

Finally, the *evaluation* $[\![e]\!]^{\mathbb{G}}$ of an RQM e over a data graph $\mathbb{G}$ is the following set of pairs of nodes in $\mathbb{G}$, where $\perp$ is the assignment with the empty domain:

$$\{(v, v') \mid \exists \lambda \text{ such that } ([v, \perp], [v', \lambda]) \in \mathcal{H}^{\mathbb{G}}(e)\}.$$

Example 3.3. Consider the evaluations of the expressions from Example 3.2 over the data graph in Figure 1:

1. the evaluation of $\downarrow x.(a[x^{=}])^+$ is $\{(v_6, v_5)\}$;
2. the evaluation of $\downarrow x.(a[x^{\neq}])^+$ is $\{(v_1, v_2), (v_1, v_5), (v_2, v_5), (v_2, v_3)\}$; and
3. the evaluation of $\downarrow x.(a \cdot b)^+[x^{\neq}]$ contains (v_1, v_4) and (v_6, v_4), but does not contain (v_2, v_4). □

Note that RQMs have the same algebraic properties as usual regular expressions: for example, composition $\cdot$ and union $\cup$ are associative, and $\cup$ is also commutative. We will use these properties silently, for example, omit parentheses in compositions and unions of several sub-expressions. Besides this, RQMs obey some algebraic laws that involve their specific operators. The following proposition immediately follows from the definitions.

Proposition 3.4. *The following equivalences hold for any RQMs e_1 and e_2, register x, and condition c:*

$$\begin{aligned} (\downarrow x.e_1) \cdot e_2 &\equiv \downarrow x.(e_1 \cdot e_2), \\ e_1 \cdot (e_2[c]) &\equiv (e_1 \cdot e_2)[c]. \end{aligned}$$

Same as for the associativity and commutativity laws, we will use these equivalences and omit parentheses accordingly. We will also use standard abbreviations e^* for $\varepsilon \cup e^+$ and e^k for the composition $e \cdot \ldots \cdot e$ of k copies of an RQM e.

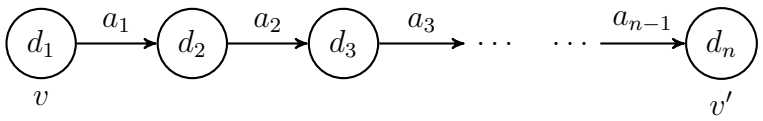

Figure 2: The data graph $\mathbb{G}_w$ corresponding to the data word $w = d_1a_1d_2 \dots a_{n-1}d_n$ (identifiers of the intermediate nodes are omitted).

3.2. From Graphs to Words

As we mentioned in the preliminaries, standard algorithms for containment of RPQs rely on the fact that two RPQs are contained if and only if the regular languages they define are contained [10]. In this section we show a similar behaviour for RQMs.

Data words are a widely studied extension of words over finite alphabets [30], in which every position carries not only a label from the finite alphabet Σ, but also a data value from the infinite domain $\mathcal{D}$. However, just for uniformity of presentation, we follow [26] and opt to the following essentially equivalent definition, by which data values are attached not to positions in a word, but "between" them.[1]

Definition 3.5. *A* data word *over a finite alphabet of labels Σ and an infinite set of data values $\mathcal{D}$ is a sequence $d_1a_1d_2a_2 \dots a_{n-1}d_n$, where $n > 0$, $d_i \in \mathcal{D}$, for each $1 \leq i \leq n$, and $a_i \in \Sigma$, for each $1 \leq i < n$.*

Every data word $w = d_1a_1d_2 \dots a_{n-1}d_n$ can be easily transformed to a data graph $\mathbb{G}_w$, consisting of n different nodes with data values $d_1, \dots, d_n$, which are connected by edges labelled with $a_1, \dots, a_{n-1}$, as illustrated in Figure 2.

The semantics of RQMs over data words is defined in the straightforward way: a data word w is *accepted* by an RQM e if and only if $(v, v') \in [\![e]\!]^{\mathbb{G}_w}$, where v and v' are the first and the last nodes of $\mathbb{G}_w$. The set of all data words accepted by an RQM e is denoted by $\mathcal{L}(e)$.

Coming back to graphs, each path

$$v_1a_1v_2a_2v_3 \dots v_{n-1}a_{n-1}v_n,$$

in a data graph $\langle V, E, \rho \rangle$ has the *corresponding* data word

$$\rho(v_1)a_1\rho(v_2)a_2\rho(v_3) \dots \rho(v_{n-1})a_{n-1}\rho(v_n).$$

[1] In [26] to distinguish this notion from the original, the term "data path" was used.

As noted in [26], for each RQM e, data graph $\mathbb{G}$, and nodes v, v' of $\mathbb{G}$, it holds that $(v, v') \in [\![e]\!]^{\mathbb{G}}$ if and only if there exists a path between v and v' such that its corresponding data word is accepted by e. Exploiting usual techniques in query containment we arrive at the following proposition, similar to the property of RPQs, mentioned in the preliminaries.

Proposition 3.6. *Given two RQMs e and e', it holds that $e \subseteq e'$ if and only if $\mathcal{L}(e) \subseteq \mathcal{L}(e')$.*

Proof. In this proof we will use the following result from [23]: a pair of nodes (v, v') of a data graph $\mathbb{G}$ belongs to $[\![e]\!]^{\mathbb{G}}$ if and only if there is a path from v to v' such that its corresponding data word belongs to $\mathcal{L}(e)$.

Assume first that $e \subseteq e'$. By definition, it means that $[\![e]\!]^{\mathbb{G}} \subseteq [\![e']\!]^{\mathbb{G}}$, for every data graph $\mathbb{G}$. Consider any data word $w = d_1 a_1 d_2 a_2 \dots a_{n-1} d_n$ such that $w \in \mathcal{L}(e)$. By definition we have that $(v, v') \in [\![e]\!]^{\mathbb{G}_w}$, where $\mathbb{G}_w$ is the data graph corresponding to w, as denoted in Figure 2. Then by our assumption we have that $(v, v') \in [\![e']\!]^{\mathbb{G}_w}$. From this and the definition of $\mathcal{L}(e')$, it follows that $w \in \mathcal{L}(e')$, as desired.

For the backward direction, suppose that $\mathcal{L}(e) \subseteq \mathcal{L}(e')$ and take any data graph $\mathbb{G}$ and pair of nodes $(v, v') \in [\![e]\!]^{\mathbb{G}}$. By the aforementioned fact, there is a path from v to v' in $\mathbb{G}$ whose corresponding data word w belongs to $\mathcal{L}(e)$. Then by our assumption we have that $w \in \mathcal{L}(e')$, so using the same fact for e' we get that $(v, v') \in [\![e']\!]^{\mathbb{G}}$. □

Note that in this proposition $e \subseteq e'$ is defined on data graphs, but $\mathcal{L}(e)$ and $\mathcal{L}(e')$ are sets of data words.

3.3. Automata for RQMs

As it is usual when dealing with query languages based on regular expressions, to establish an upper bound for the containment of RQMs we use a class of automata that subsumes RQMs—namely, register automata [19, 26]. A register automaton is essentially a finite state automaton equipped with a finite set of registers allowing it to store data values for later comparisons. It can move to the next state either by matching the current label from a finite alphabet or by comparing the current data value with the ones in the registers. Since we use data words it is best to draw the formal definition from [26].

Definition 3.7. *A* register data word automaton *with the registers X over an alphabet of labels Σ is a tuple $\langle P, r_s, q_f, \gamma \rangle$, where*

– P *is a set of* states *that is a disjoint union of* data states R *and* word states Q*;*

– $r_s \in R$ *and* $q_f \in Q$ *are the* initial *and the* final *states, respectively;*

– γ *is a pair consisting of*

  - *the* data transition relation $\delta \subseteq R \times \mathcal{C}_X \times 2^X \times Q$*, where* $\mathcal{C}_X$ *is the set of all conditions over* X *and* 2^X *is the set of all subsets of* X*, and*
  - *the* word transition relation $\alpha \subseteq Q \times \Sigma \times R$*.*

The intuition behind this definition is that since we alternate between data values and labels (word symbols) in data words, we also alternate between data states, which expect data value as the next symbol, and word states, which expect labels as the next symbol. We start with a data value, so the initial state r_s is a data state, and end with a data value, so the final state q_f, seen after reading that value, is a word state.

In a data state, the automaton checks if the current data value and assignment of the registers satisfy the condition, and if they do, moves to a word state and updates some of the registers with the read data value. In a word state a register automaton behaves as a usual nondeterministic final state automaton (NFA), except that it moves to a data state.

Example 3.8. Consider again the expression $\downarrow x.(a[x^=])^+$ from Example 3.2. An equivalent register automaton is depicted in Figure 3(a), where data transition arrows are labelled with conditions and sets of assigned registers, and word transition arrows are labelled with alphabet symbols. It uses data states $\{r_s, r\}$ and word states $\{q, q_f\}$ with initial state r_s and final state q_f, respectively, while γ is given by the data transitions $(r_s, \mathsf{true}, \{x\}, q)$ and $(r, x^=, \emptyset, q_f)$, and the word transitions (q, a, r) and (q_f, a, r).

Next, consider the expression $\downarrow x.a{\cdot}\downarrow y.(a[x^=] \cup a[y^=])^+$, over registers $\{x, y\}$, which specifies all data words over a single label a whose each data value is equal to either the first or the second data value (stored in registers x and y, respectively). An equivalent register automaton is depicted in Figure 3(b). □

The semantics of register automata is given in terms of configurations. Given a set $\mathcal{D}$ of data values, a *configuration* of a register automaton $\mathbb{A} = \langle P, r_s, q_f, \gamma \rangle$ over registers X is a pair $[p, \lambda]$, where p is either a data or word state in P and $\lambda : X \rightharpoonup \mathcal{D}$ is an assignment of the registers. A configuration is *initial* if it is of the form $[r_s, \bot]$, for the assignment $\bot$ with the empty domain, and a configuration is *final* if its state is q_f. A configuration $[p_2, \lambda_2]$ is *reachable* from a configuration

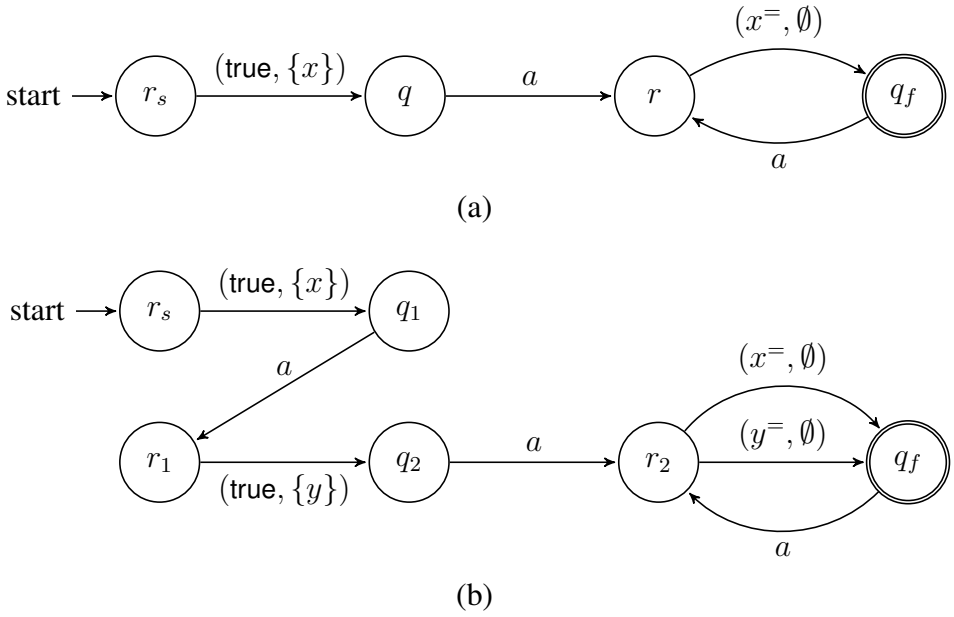

Figure 3: Register automata equivalent to RQMs in Example 3.8.

$[p_1, \lambda_1]$ *by* a symbol $u \in \mathcal{D} \cup \Sigma$, denoted as $[p_1, \lambda_1] \Rightarrow_u^{\mathbb{A}} [p_2, \lambda_2]$, if one of the following holds:

1. u is a data value in $\mathcal{D}$, and there is a data transition (p_1, c, Y, p_2) in δ such that $u, \lambda_1 \models c$ and
$$\lambda_2(x) = \begin{cases} u, & x \in Y, \\ \lambda_1(x), & x \notin Y; \end{cases}$$
2. u is a label in Σ, there is a word transition (p_1, u, p_2) in α, and $\lambda_1 = \lambda_2$.

A *run* of $\mathbb{A}$ *on* a data word $w = u_1, \ldots, u_n$ is a a sequence of configurations $[p_0, \lambda_0], \ldots, [p_n, \lambda_n]$ such that $[p_0, \lambda_0]$ is initial and $[p_{i-1}, \lambda_{i-1}] \Rightarrow_{u_i}^{\mathbb{A}} [p_i, \lambda_i]$ for each $1 \leq i \leq n$. The run is *accepting* if $[p_n, \lambda_n]$ is a final configuration. A data word w is *accepted* by $\mathbb{A}$ if there is an accepting run of $\mathbb{A}$ on w. The language of all data words accepted by $\mathbb{A}$ is denoted by $\mathcal{L}(\mathbb{A})$.

The key property of register automata that we will use is the fact that they capture RQMs, and that the translation from an RQM to its equivalent automaton can be done efficiently.

Proposition 3.9 ([25, 26])**.**

1. *For every RQM e there is a register automaton $\mathbb{A}$ such that $\mathcal{L}(e) = \mathcal{L}(\mathbb{A})$. Moreover, $\mathbb{A}$ can be constructed in time polynomial in the size of e.*
2. *For every register automaton $\mathbb{A}$ there is an RQM e such that $\mathcal{L}(\mathbb{A}) = \mathcal{L}(e)$.*

Although Proposition 3.9 was stated for general register automata, the proof in [26] also shows that if the starting expression does not have inequalities, then the resulting automaton is also free of inequalities, and vice versa.

3.4. Containment of RQMs: Main Results

The connection between RQMs and register automata means bad news, as the containment problem for these type of automata is known to be undecidable [28]. Together with Proposition 3.9, this result implies that containment is also undecidable for RQMs.

Corollary 3.10. *The problem* CONTAINMENT(RQMs) *is undecidable.*

As we see, the power that RQMs gain through its data manipulation mechanism comes with a high price for static analysis tasks. But this result naturally leads to the question of finding decidable subclasses. Previous work on automata has found that testing containment of an expression using at most one register in an expression using at most two registers is decidable [28], but this approach seems too restrictive for our case. We concentrate instead on *positive RQMs*, that is, RQMs that use only atoms of the form $x^{=}$ in the conditions. It is already known that the containment of positive RQMs is decidable [31], but so far the complexity is not known. The next theorem fills the gap.

Theorem 3.11. *Problem* CONTAINMENT(positive RQMs) *is* EXPSPACE*-complete.*

Next two sections are devoted to the proof of this theorem. We show the upper bound in Section 3.5 and then continue with the lower bound in Section 3.6.

3.5. Containment of Positive RQMs: Upper Bound

It is more convenient to study the corresponding problem for register automata, which is stated as follows: given register automata $\mathbb{A}$ and $\mathbb{A}'$ that use no inequalities in the conditions in their transitions, is it true that $\mathcal{L}(\mathbb{A}) \subseteq \mathcal{L}(\mathbb{A}')$? By Propositions 3.6 and 3.9 we know that an exponential space algorithm for

this problem automatically yields an exponential space algorithm for CONTAINMENT(positive RQMs), simply by transforming the expressions into their equivalent automata.

Lemma 3.12. *The problem of checking whether $\mathcal{L}(\mathbb{A}) \subseteq \mathcal{L}(\mathbb{A}')$ for register automata $\mathbb{A}$ and $\mathbb{A}'$ that do not use inequalities is in* EXPSPACE.

Proof. The idea of the proof is to simulate both $\mathbb{A}$ and $\mathbb{A}'$ with NFAs of exponential size, from which the EXPSPACE bound follows since containment of NFAs can be solved in polynomial space. In order to do this let us first show that the space of data words we need to analyse can be reduced to words where the number of data values is somehow bounded by the number of registers. In order to formalise this observation, we need the following notion: a data word $w = d_1 a_1 d_2 a_2 \dots a_{n-1} d_n$ is *ℓ-bounded in memory*, for a positive number ℓ, if for each position in w the number of data values appearing both in the left and in the right of w is always less than or equal to ℓ. That is, if $|\{d_1, \dots, d_i\} \cap \{d_{i+1}, \dots, d_n\}| \leq \ell$ for any i, $1 \leq i < n$. We then have the following claim, assuming that register automaton $\mathbb{A}$ has ℓ registers.

Claim 3.13. *If $\mathbb{A} \not\subseteq \mathbb{A}'$, then there is an ℓ-bounded in memory data word in $\mathcal{L}(\mathbb{A})$ but not in $\mathcal{L}(\mathbb{A}')$.*

Proof. Let $w = d_1 a_1 d_2 a_2 \dots a_{n-1} d_n$ be a data word witnessing the fact that $\mathbb{A} \not\subseteq \mathbb{A}'$, and let

$$[r_s, \bot], [q_1, \lambda_1], [r_1, \lambda_1], \dots, [r_{n-1}, \lambda_{n-1}], [q_f, \lambda_n]$$

be an accepting run of $\mathbb{A}$ on w. If w is ℓ-bounded in memory, then the claim is proved. Otherwise, there is a number i such that $|\{d_1, \dots, d_i\} \cap \{d_{i+1}, \dots, d_n\}| > \ell$. Therefore, there exists a data value $d \notin \mathit{Image}(\lambda_i)$ such that $d_j = d$ and $d_k = d$ for some $j \leq i$ and $k > i$. Consider the data word w' obtained from w by replacing all d_k such that $d_k = d$ and $k > i$ with a fresh data value. On the one hand, w' is still such that $w' \in \mathcal{L}(\mathbb{A})$, because we can obtain an accepting run from the run on w by replacing all occurrences of d in λ_k, $k > i$, with the new data value. On the other hand, $\mathbb{A}'$ uses only equalities, so we still have that $w' \notin \mathcal{L}(\mathbb{A}')$. If w' is ℓ-bounded in memory, then the claim is proved. Otherwise, we can repeat the replacement as above until the conditions are satisfied. This process is terminating, because each i can be considered at most n times. □

By this claim we can focus solely on words that are ℓ-bounded in memory. To begin with, observe that these words (for a given ℓ) can be encoded as a word

over the alphabet $\{1, \ldots, \ell\} \times \{\mathit{old}, \mathit{new}\} \cup \Sigma$; the idea is to keep labels in Σ untouched, and for data symbols a pair (i, old) represents a data value i that has been seen before in the word, whereas the pair (i, new) represents a new data value (taking up the place of the i-th data value that can be repeated on this word).

Thus, we can simulate automata $\mathbb{A}$ and $\mathbb{A}'$ by usual NFAs $\mathbb{B}$ and $\mathbb{B}'$, respectively, both over the alphabet $\Gamma = (\{1, \ldots, \ell\} \times \{\mathit{old}, \mathit{new}\}) \cup \Sigma$.

Next we give a formal construction of NFA $\mathbb{B}$ corresponding to register automaton $\mathbb{A} = \langle P, r_s, q_f, \gamma \rangle$, while the construction of $\mathbb{B}'$ corresponding to $\mathbb{A}'$ is analogous.

1. The states of $\mathbb{B}$ are all the configurations $[p, \lambda]$ of $\mathbb{A}$ such that the range of assignment λ is $\{1, \ldots, \ell\}$ (recall that λ can still be undefined for some registers).
2. The initial state of $\mathbb{B}$ is $[r_s, \bot]$ and the final states are all $[q_f, \lambda]$ for any λ.
3. The transition relation of $\mathbb{B}$ is defined as follows:
   - first, $\mathbb{B}$ copies all label transitions of $\mathbb{A}$, that is, for all labels a in Σ and states $[q, \lambda]$ and $[r, \lambda]$ of $\mathbb{B}$ such that $[q, \lambda] \Rightarrow_a^{\mathbb{A}} [r, \lambda]$, the transition relation of $\mathbb{B}$ contains the triple $([q, \lambda], a, [r, \lambda])$;
   - next, $\mathbb{B}$ also simulates $\mathbb{A}$ on its data transitions whenever we see a pair of the form (i, old), that is, with a data value we have already seen before; formally, the transition relation of $\mathbb{B}$ contains the triple $([r, \lambda_1], (i, \mathit{old}), [q, \lambda_2])$ for each pair of states $[r, \lambda_1]$ and $[q, \lambda_2]$ of $\mathbb{B}$ such that $[r, \lambda_1] \Rightarrow_i^{\mathbb{A}} [q, \lambda_2]$;
   - finally, when the read symbol is of the form (i, new), $\mathbb{B}$ simulates $\mathbb{A}$ on the assignment of registers obtained from the previous one by erasing value i from the registers; formally, given an assignment λ, let us denote by $\lambda^{[i \to \bot]}$ the new assignment obtained from λ by leaving undefined all registers x with $\lambda(x) = i$; then the transition relation of $\mathbb{B}$ contains a triple $([r, \lambda_1], (i, \mathit{new}), [q, \lambda_2])$ for each pair of states $[r, \lambda_1]$, $[q, \lambda_2]$ of $\mathbb{B}$ and $1 \leq i \leq \ell$ such that $[r, \lambda_1^{[i \to \bot]}] \Rightarrow_i^{\mathbb{A}} [q, \lambda_2]$.

To state the correctness of the encoding, consider, for each ℓ-bounded in memory data word $w = d_1 a_1 d_2 a_2 \cdots a_{n-1} d_n$, a word $w' = b_1 a_1 b_2 a_2 \cdots a_{n-1} b_n$ over Γ where each b_i is defined as follows:

1. if $d_i = d_j$ for some $j < i$ then $b_i = (m, \mathit{old})$, where m, $1 \leq m \leq \ell$, is the number mentioned in b_j;
2. if $d_i \neq d_j$ for all $j < i$, then $b_i = (m, \mathit{new})$, where m is such that

– either b_j for any $j < i$ does not mention m,
– or $d_j \neq d_k$ for all $k > i$, where j is the maximal number such that b_j mentions m.

Note that the existence of w' is guaranteed by definition of ℓ-bounded in memory data word. The following claim immediately follows from the definitions.

Claim 3.14. *Register automaton $\mathbb{A}$ (or $\mathbb{A}'$) accepts ℓ-bounded in memory data word w if and only if $\mathbb{B}$ (or $\mathbb{B}'$) accepts w'.*

Combining Claims 3.13 and 3.14, we have that $\mathbb{A} \subseteq \mathbb{A}'$ if and only if $\mathbb{B} \subseteq \mathbb{B}'$. Both $\mathbb{B}$ and $\mathbb{B}'$ are of exponential size by construction, so their containment can be checked in EXPSPACE by a standard containment algorithm for NFAs. □

From the proof above we can immediately obtain the following result for *ℓ-bounded positive RQMs*, that is, positive RQMs that can use at most ℓ registers.

Corollary 3.15. *Problem* CONTAINMENT(ℓ-bounded positive RQMs) *is* PSPACE*-complete for fixed any natural number ℓ.*

Proof. The upper bound follows from the fact that in this case the NFAs $\mathbb{B}$ and $\mathbb{B}'$ simulating $\mathbb{A}$ and $\mathbb{A}'$ are of polynomial size, and their containment can be checked in PSPACE. The lower bound is inherited from the containment problem for usual regular expressions. □

3.6. Containment of Positive RQMs: Lower Bound

Next we show the exponential space lower bound for the containment of arbitrary positive RQMs.

Lemma 3.16. *The problem* CONTAINMENT(positive RQMs) *is* EXPSPACE*-hard.*

Proof. The proof is by reduction of the complement of the acceptance problem for a Turing machine that works in EXPSPACE. Some ideas of this proof are adapted from [7, Theorem 6].

Let $\mathcal{L}$ be a language that belongs to EXPSPACE over some alphabet Γ', $\mathbb{M}$ be a deterministic Turing machine that decides $\mathcal{L}$ in EXPSPACE, and w be a word (plain, without data values) over Γ'. Next we show how to construct, in polynomial time in the size of $\mathbb{M}$ and w, RQMs e and e' such that $\mathcal{L}(e) \subseteq \mathcal{L}(e')$ if and only if $\mathbb{M}$ does not accept w as input. By Proposition 3.6 this is enough for a proof of the lemma.

Let $\mathbb{M} = \langle Q, q_s, q_{acc}, q_{rej}, \delta \rangle$, where Q is the set of states, $q_s \in Q$ is the initial state, $q_{acc}, q_{rej} \in Q$ are the accepting and the rejecting states, respectively, and

$$\delta : \ (Q \setminus \{q_{acc}, q_{rej}\}) \times \Gamma \ \rightarrow \ Q \times \Gamma \times \{-1, +1\}$$

is the transition function with $\Gamma = \Gamma' \cup \{\textvisiblespace\}$, for $\textvisiblespace$ the blank symbol of the tape. Furthermore, let w be a word over Γ. Since $\mathbb{M}$ decides $\mathcal{L}$ in EXPSPACE, there exists a polynomial P that does not depend on w such that $\mathbb{M}$ decides w using 2^n cells for $n = \mathsf{P}(|w|)$.

In the proof the following notation will be convenient: for any alphabet $\Omega = \{b_1, \ldots, b_\ell\}$, we denote by the same symbol Ω the regular expression $b_1 \cup \cdots \cup b_\ell$.

We now turn to the construction of RQMs e and e'. We also give some intuition on the reduction as we go deeper into the construction. The alphabet of e and e' is

$$\Sigma = \Gamma \cup (Q \times \Gamma) \cup \{\#, \&, \%, \triangle\}.$$

RQM e is

$$\downarrow x_0.\triangle \cdot \downarrow x_1.\big(\triangle[x_0^{=}] \cup \triangle[x_1^{=}]\big) \cdot \big(\Sigma[x_0^{=}] \cup \Sigma[x_1^{=}]\big)^*.$$

Intuitively, e ensures that the first two labels in the word are the symbol $\triangle$, and that all data values in a data word are equal to the first or the second value. In other words, $\mathcal{L}(e)$ consists of all data words $d_1 a_1 d_2 a_2 \cdots d_{k-1} a_{k-1} d_k$ such that $a_1 = a_2 = \triangle$, each a_i belongs to Σ and each d_i, for $3 \leq i \leq k$, is equal to one of d_1 or d_2.

Assume for a moment that we only consider those data words in which the first data value is different from the second (we will enforce this by means of e', as we explain later on in the proof). Then we can arbitrarily designate these values by numbers 0 and 1: 0 is the first data value of the word and 1 is the second data value. In this case, words accepted by e are all those words that use only 0 and 1 as their data values.

We need more notation. Given a number i, $0 \leq i \leq 2^n$ for $n = \mathsf{P}(|w|)$, that is written in binary as $d_n d_{n-1} \cdots d_1$, with all $d_j \in \{0, 1\}$, let $\langle i \rangle$ denote its representation as a data word

$$\#d_n \#d_{n-1} \# \cdots \#d_1.$$

For example, $\langle 0 \rangle$ is the data word $(\#0)^n$, and $\langle 2 \rangle$ is the data word $(\#0)^{n-2}\#1\#0$. (Formally, these representations are not really data words, because they start with

labels, not data values; however, we allow this inconsistency for brevity, keeping in mind that such words are used only as sub-words of proper data words.)

Then, we represent any configuration of the Turing machine with state q, the head at cell i, and the contents of the cells $a_0 \dots a_{2^n-1}$ with each $a_j \in \Gamma$, by the data word

$$\begin{aligned} &\&\, 0\, \langle 0 \rangle\, a_0\, 0 \quad \&\, 0\, \langle 1 \rangle\, a_1\, 0 \\ &\qquad \cdots \quad \&\, 0\, \langle i-1 \rangle\, a_{i-1}\, 0 \quad \&\, 0\, \langle i \rangle\, (q, a_i)\, 0 \quad \&\, 0\, \langle i+1 \rangle\, a_{i+1}\, 0 \quad \cdots \\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \&\, 0\, \langle 2^n - 1 \rangle\, a_{2^n-1}\, 0. \end{aligned} \tag{1}$$

Intuitively, the sub-words $\langle 0 \rangle, \langle 1 \rangle, \langle 2 \rangle, \dots, \langle 2^n - 1 \rangle$ index each of the 2^n used cells of $\mathbb{M}$, and the symbol following such a word represents either the content of the cell plus the state of $\mathbb{M}$, if $\mathbb{M}$ is pointing at that particular cell at the given step of the computation, or just the contents of the cell, if the head does not point here. Note that the data value 0 after each $\&$, each a_j and (q, a_i) is there just because we need some data value between labels. It does not play any other role and could be 1 instead.

Since every configuration of $\mathbb{M}$ can be represented as such a data word, any run of $\mathbb{M}$ on input w can be seen as a sequence (i.e., concatenation) of representations of consequent configurations, separated by a special label $\%$. To initialise the two different data values 0 and 1, we also add a special prefix to the representation. Formally, a run of $\mathbb{M}$ on w of length m is represented as a data word

$$0 \triangle 1 \triangle 0 \quad \%\, 0\, u_1 \quad \%\, 0\, u_2 \quad \dots \quad \%\, 0\, u_m, \tag{2}$$

where each u_j is of the form (1) and represents the j-th configuration of $\mathbb{M}$ on the run. Data value 0 after the second $\triangle$ and each $\%$ plays the same auxiliary role as in the representations of configurations.

The idea of the reduction is as follows. We have already defined RQM e, which accepts the words that have only 0 and 1 as data values. In turn, RQM e' accepts all the words from $\mathcal{L}(e)$ that are either not representations of runs of $\mathbb{M}$ on w of form (2), or representations of runs that are not accepting. Hence, there is an accepting run of $\mathbb{M}$ on w if and only if there is a data word that is in $\mathcal{L}(e)$ but not in $\mathcal{L}(e')$.

RQM e' is a union of six parts $e'_0 \cup e'_1 \cup e'_2 \cup e'_3 \cup e'_4 \cup e'_5$. These parts have the following intuitive interpretations:

- e'_0 accepts all data words that use a single data value (instead of two);
- e'_1 accepts all words with the wrong sequences of labels;

– e'_2 accepts all words that are not concatenations of sequences of numbered contents of 2^n cells, possibly paired with states, as in (1);

– e'_3 accepts all concatenations of words of the form (1) such that some of these words do not represent valid configurations for $\mathbb{M}$;

– e'_4 accepts all words whose first configuration is not the initial configuration of $\mathbb{M}$ on input w;

– e'_5 accepts all words containing two consecutive configurations that do not agree with transition function δ;

– e'_6 accepts all words whose last configuration is not a final configuration.

Next we formally define these parts of e'.

Expression e'_0: it is defined as $\downarrow x_0.(\Sigma[x^{=}])^*$.

Expression e'_1: it accepts all data words with $\triangle$ as first two labels that do not have a proper structure in the labels, that is, whose projections to labels do not satisfy the regular expression $\triangle \cdot \triangle \cdot (\% \cdot (\& \cdot (\#^n \cdot (\Gamma \cup (Q \times \Gamma)))^*)^*)^*$. Note that this is in fact an ordinary regular expression. It is also straightforward to see that the complement of this expression can be defined as an expression of polynomial size.

Expression e'_2: it accepts all words with a "configuration" in which cells are not numbered in the proper order from $\langle 0 \rangle$ to $\langle 2 \rangle^n - 1$. To this end, $e'_2 = \downarrow x_0.\triangle \cdot \downarrow x_1.\triangle \cdot e''_2$, where e''_2 is the union of

– RQMs

$$\Sigma^* \cdot \% \cdot \& \cdot \#^* \cdot \#[x_1^{=}] \cdot \Sigma^*,$$
$$\Sigma^* \cdot \#[x_0^{=}] \cdot \#^* \cdot \Sigma \cdot (\% \cdot \Sigma^* \cup \varepsilon),$$

which look for configurations starting with something different from $\langle 0 \rangle$ and ending with something different from $\langle 2^n - 1 \rangle$, respectively;

– RQMs

$$\Sigma^* \cdot \& \cdot \#^{n-1} \cdot \#[x_0^{=}] \cdot \Sigma \cdot \& \cdot \#^{n-1} \cdot \#[x_0^{=}] \cdot \Sigma^*,$$
$$\Sigma^* \cdot \& \cdot \#^{i} \cdot \#[x_0^{=}] \cdot \#^{n-i-2} \cdot \#[x_0^{=}] \cdot \Sigma \cdot \& \cdot \#^{i} \cdot \#[x_1^{=}] \cdot \Sigma^*,$$
$$\Sigma^* \cdot \& \cdot \#^{i} \cdot \#[x_1^{=}] \cdot \#^{n-i-2} \cdot \#[x_0^{=}] \cdot \Sigma \cdot \& \cdot \#^{i} \cdot \#[x_0^{=}] \cdot \Sigma^*,$$

with $0 \leq i \leq n-2$, which look for a configuration where an even number, that is, a number ending with 0 in its binary representation, is not followed by the next one: indeed, the first RQM checks that the next number does not end with 0, and the other ones check that each of the first $n-1$ digits is the same in both numbers;

– similarly, RQMs

$$\Sigma^* \cdot \& \cdot \#^{n-2} \cdot \#[x_0^=] \cdot \#[x_1^=] \cdot \Sigma \cdot \& \cdot \#^{n-2} \cdot \# \cdot \#[x_1^=] \cdot \Sigma^*,$$
$$\Sigma^* \cdot \& \cdot \#^{n-2} \cdot \#[x_0^=] \cdot \#[x_1^=] \cdot \Sigma \cdot \& \cdot \#^{n-2} \cdot \#[x_0^=] \cdot \# \cdot \Sigma^*,$$
$$\Sigma^* \cdot \& \cdot \#^{i} \cdot \#[x_0^=] \cdot \#^{n-i-3} \cdot \#[x_0^=] \cdot \#[x_1^=] \cdot \Sigma \cdot \& \cdot \#^{i} \cdot \#[x_1^=] \cdot \Sigma^*,$$
$$\Sigma^* \cdot \& \cdot \#^{i} \cdot \#[x_1^=] \cdot \#^{n-i-3} \cdot \#[x_0^=] \cdot \#[x_1^=] \cdot \Sigma \cdot \& \cdot \#^{i} \cdot \#[x_0^=] \cdot \Sigma^*,$$

with $0 \leq i \leq n-3$, which look for a configuration where a number ending with 01 in binary is not followed by the next one: the first two RQMs check that the next number does not end with anything except 10, and the other ones check that each of the first $n-2$ digits is the same in both numbers;

– similarly, RQMs that deal with numbers ending with 011, 0111, etc.: e.g., for 0111 the first four RQMs check that the next number does not end with anything except 1000, and the other ones check that each of the first $n-4$ digits is the same in both numbers.

Expression e'_3: it accepts all words with "configurations" with no heads of the machine and more than one head. To this end, e'_3 is the union of RQMs

$$\Sigma^* \cdot \% \cdot \big(\Sigma \setminus ((Q \times \Gamma) \cup \{\%\})\big)^* \cdot (\% \cdot \Sigma^* \cup \varepsilon),$$
$$\Sigma^* \cdot (Q \times \Gamma) \cdot \big(\Sigma \setminus ((Q \times \Gamma) \cup \{\%\})\big)^* \cdot (Q \times \Gamma) \cdot \Sigma^*,$$

which look for two $\%$ without a label from $Q \times \Gamma$ between them (or for the last $\%$ in the word without such a label after it), and for two labels from $\Gamma \times Q$ without $\%$ between them, respectively. Note that e'_3 does not manipulate any data values.

Expression e'_4: it accepts all words whose first configuration is not initial. To this end, if $w = a_0 \cdots a_k$ then $e'_4 = \triangle \cdot \triangle \cdot \% \cdot e''_4$, where e''_4 is the union of RQMs

$$\& \cdot \#^n \cdot \big(\Sigma \setminus \{(q_s, a_0)\}\big) \cdot \Sigma^*,$$
$$\& \cdot \#^n \cdot (q_s, a_0) \cdot \quad \& \cdot \#^n \cdot \big(\Sigma \setminus \{a_1\}\big) \cdot \Sigma^*,$$
$$\cdots$$
$$\& \cdot \#^n \cdot (q_s, a_0) \cdot \quad \& \cdot \#^n \cdot a_1 \cdot \ldots \cdot \& \cdot \#^n \cdot \big(\Sigma \setminus \{a_k\}\big) \cdot \Sigma^*,$$
$$\& \cdot \#^n \cdot (q_s, a_0) \cdot \quad \& \cdot \#^n \cdot a_1 \cdot \ldots \cdot \& \cdot \#^n \cdot a_k \cdot \big(\Sigma \setminus \{\%\}\big)^* \cdot \quad \& \cdot \#^n \cdot \textvisiblespace \cdot \Sigma^*.$$

The first of them looks for words with something different from (q_s, a_0) in the position corresponding to the first cell in the first configuration, next k look for words with something different from $a_1, \ldots, a_k$ in the positions for the next k cells, and the last one looks for words with something different from $\textvisiblespace$ in all the following cells.

Expression e'_5: it accepts all words whose consecutive configurations do not agree with the transition function δ. To this end, it is the union of

– RQMs

$$\Sigma^* \cdot \# \cdot \downarrow y_1.\# \cdot \downarrow y_2.\# \cdot \ldots \cdot \downarrow y_{n-1}.\# \cdot \downarrow y_n.a \cdot \big(\Sigma \setminus \{\%\}\big)^* \cdot \% \cdot \big(\Sigma \setminus \{\%\}\big)^* \cdot \#[y_1^=] \cdot \#[y_2^=] \cdot \ldots \cdot \#[y_n^=] \cdot \big((\Gamma \setminus \{a\}) \cup (Q \times (\Gamma \setminus \{a\}))\big) \cdot \Sigma^*$$

for each $a \in \Gamma$, which look for consecutive configurations with different contents of a cell not under the head in the first of the configurations;

– RQMs

$$\Sigma^* \cdot \# \cdot \downarrow y_1.\# \cdot \downarrow y_2.\# \cdot \ldots \cdot \downarrow y_{n-1}.\# \cdot \downarrow y_n.(q,a) \cdot \big(\Sigma \setminus \{\%\}\big)^* \cdot \% \cdot \big(\Sigma \setminus \{\%\}\big)^* \cdot \#[y_1^=] \cdot \#[y_2^=] \cdot \ldots \cdot \#[y_n^=] \cdot \big((\Gamma \setminus \{a'\}) \cup (Q \times \Gamma)\big) \cdot \Sigma^*$$

for each q, a and a' such that $\delta(q,a)$ has a' as the new symbol, which look for consecutive configurations whose contents of the cell under the head does not change according to δ;

– RQMs

$$\Sigma^* \cdot \# \cdot \downarrow y_1.\# \cdot \downarrow y_2.\# \cdot \ldots \cdot \downarrow y_{n-1}.\# \cdot \downarrow y_n.(q,a) \cdot \big(\Sigma \setminus \{\%\}\big)^* \cdot \% \cdot \big(\Sigma \setminus \{\%\}\big)^* \cdot \big(\Gamma \cup ((Q \setminus \{q'\}) \times \Gamma)\big) \cdot \& \cdot \#[y_1^=] \cdot \#[y_2^=] \cdot \ldots \cdot \#[y_n^=] \cdot \Sigma^*$$

for each q, a and q' such that $\delta(q,a)$ has -1 as the direction and q' as the new state, which look for consecutive configurations such that the head does not move left properly; and

– RQMs

$$\Sigma^* \cdot \# \cdot \downarrow y_1.\# \cdot \downarrow y_2.\# \cdot \ldots \cdot \downarrow y_{n-1}.\# \cdot \downarrow y_n.(q,a) \cdot \big(\Sigma \setminus \{\%\}\big)^* \cdot \% \cdot \big(\Sigma \setminus \{\%\}\big)^* \cdot \#[y_1^=] \cdot \#[y_2^=] \cdot \ldots \cdot \#[y_n^=] \cdot \Gamma \cdot \& \cdot \#^n \cdot \big(\Gamma \cup ((Q \setminus \{q'\}) \times \Gamma)\big) \cdot \Sigma^*$$

for each q, a and q' such that $\delta(q,a)$ has $+1$ as the direction and q' as the new state, which look for consecutive configurations such that the head does not move right properly.

Expression e'_6: it accepts all words that do not represent an accepting run. To this end, it is the union of RQMs

$$\Sigma^* \cdot (q, a) \cdot \big(\Sigma \setminus \{\%\}\big)^*$$

for all $q \in Q \setminus \{q_{acc}\}$ and $a \in \Gamma$, which look for configurations that do not have the final state (paired with a symbol from Γ) after the last $\%$.

With these definitions at hand, it is now straightforward to show that $\mathcal{L}(e) \subseteq \mathcal{L}(e')$ if and only if $\mathbb{M}$ does not accept input w: indeed, on the one hand, according to the construction, a data word in $\mathcal{L}(e)$ but not in $\mathcal{L}(e')$, if it exists, represents a run of $\mathbb{M}$ on w, and, moreover, this run is accepting; on the other hand, if $\mathbb{M}$ accepts w, then there exists its accepting run that is represented by some data word in $\mathcal{L}(e)$ but not in $\mathcal{L}(e')$. □

All in all, positive RQMs appear as a natural subclass of RQMs with decidable query containment. However, when comparing the complexity with the one for RPQs, we see that allowing positive data test comparisons results in an exponential jump (unless of course we fix the amount of registers). In the following section we consider another class of queries extending RPQs, which also allows for data value comparisons, but in a more restricted way than RQMs. As we will see, the positive subclass of this class has the same complexity of containment as RPQs.

4. Regular Queries with Data Tests

Looking for classes of queries capable of handling data values, but with better query answering properties than RQMs, the authors of [26] introduced *regular queries with data tests*, or *RQDs* for short (these were called *regular expressions with equality* in the original paper). An example of such a query is the expression $a(b^+)_{=}c$, whose intention is to return all pairs of nodes connected by a path labelled $ab \cdots bc$ and where the data values before and after the sequence of b's are the same.

All RQDs can be expressed as RQMs, but when expressing RQDs we can restrict the usage of registers: each stored data value can be retrieved and compared only once, and the order of these storing and retrieving operations is not arbitrary, but on the "first in, last out" discipline. The data complexity of RQDs' evaluation is the same as for RQMs—in NLogSpace, but the combined complexity is much better, in PTime [26].

4.1. Syntax and Semantics of RQDs

The syntax for RQDs can be defined in a direct, much simpler way than for RQMs, without even mentioning registers and conditions.

Definition 4.1. *A* regular query with data tests *(or* RQD*) over an alphabet of labels Σ is an expression satisfying the grammar*

$$e \;:=\; \varepsilon \mid a \mid e \cup e \mid e \cdot e \mid e^{+} \mid e_{=} \mid e_{\neq}$$

where a ranges over labels in Σ.

Again, before the formal definition of semantics we give some examples of RQDs and their connection to RQMs.

Example 4.2. Recall RQMs from Example 3.3 (we consider them here in different order for better understanding of the relation between RQMs and RQDs).

1. The RQM $\downarrow x.(a \cdot b)^{+}[x^{\neq}]$ can be written as the RQD $((a \cdot b)^{+})_{\neq}$: the first data value is stored, then the sequence of ab's is read, and then the value is retrieved and compared for inequality with the current one. Note that the stored value is used just once.
2. The RQM $\downarrow x.(a[x^{=}])^{+}$ can be written as the RQD $(a_{=})^{+}$: the first data value is stored; then a is read; then the stored data value is retrieved and compared with the current one for equality; if successful, this current value (equal to the original!) is stored again, another a is read, and so on. If the parsing continues, then the current data value is always equal to the original one, even if we use each stored value just once.
3. Contrary to the previous case, it can be shown that the RQM $\downarrow x.(a[x^{\neq}])^{+}$ cannot be expressed as an RQD: indeed, after the first comparison the original data value is lost, and storing the current data value (different from the original) cannot help with correct comparison on the next step.
4. The RQM $\downarrow x.a \cdot \downarrow y.b[y^{=}] \cdot a[x^{=}]$ can be written as the RQD $(a \cdot b_{=} \cdot a)_{=}$. However, the very similar RQM $\downarrow x.a \cdot \downarrow y.b[x^{=}] \cdot a[y^{=}]$ is not expressible as an RQD, since the sequence in which data values have to be retrieved does not respect the "first-in-last-out" discipline required by RQD syntax. □

The semantics of RQDs is also defined in a much simpler way than for RQMs. The *evaluation* $[\![e]\!]^{\mathbb{G}}$ of an RQD e over a data graph $\mathbb{G} = \langle V, E, \rho \rangle$ is the set of all pairs (v_1, v_2) of nodes in V defined recursively in Table 2.

$$
\begin{array}{lcl}
[\![\varepsilon]\!]^{\mathbb{G}} & = & \{(v,v) \mid v \in V\} \\
[\![a]\!]^{\mathbb{G}} & = & \{(v,v') \mid (v,a,v') \in E\} \\
[\![e_1 \cdot e_2]\!]^{\mathbb{G}} & = & [\![e_1]\!]^{\mathbb{G}} \bullet [\![e_2]\!]^{\mathbb{G}} \\
[\![e_1 \cup e_2]\!]^{\mathbb{G}} & = & [\![e_1]\!]^{\mathbb{G}} \cup [\![e_2]\!]^{\mathbb{G}} \\
[\![e^+]\!]^{\mathbb{G}} & = & [\![e]\!]^{\mathbb{G}} \cup ([\![e]\!]^{\mathbb{G}} \bullet [\![e]\!]^{\mathbb{G}}) \cup ([\![e]\!]^{\mathbb{G}} \bullet [\![e]\!]^{\mathbb{G}} \bullet [\![e]\!]^{\mathbb{G}}) \cup \ldots \\
[\![e_{=}]\!]^{\mathbb{G}} & = & \{(v,v') \mid (v,v') \in [\![e]\!]^{\mathbb{G}}, \rho(v) = \rho(v')\} \\
[\![e_{\neq}]\!]^{\mathbb{G}} & = & \{(v,v') \mid (v,v') \in [\![e]\!]^{\mathbb{G}}, \rho(v) \neq \rho(v')\}
\end{array}
$$

Table 2: Semantics of RQDs with respect to a data graph $\mathbb{G}$. The composition of binary relations is again denoted $\bullet$.

As Example 4.2 suggests, and as is formally shown in [26], the class of RQDs is strictly contained in the class of RQMs. Indeed, to transform an RQD to RQM we just need to recursively replace each subexpression of the form $e_{\sim}$, $\sim \in \{=, \neq\}$, with the subexpression $\downarrow x.e[x^{\sim}]$, where x is a previously unused register. However, as we have seen, there are RQMs which cannot be transformed to RQDs.

As any RQM, each RQD e defines the language $\mathcal{L}(e)$ of data words, which consists of all w such that $(v,v') \in [\![e]\!]^{\mathbb{G}_w}$, with $\mathbb{G}_w$ as in the Figure 2. Hence, Proposition 3.6 allows us to reduce query containment to language containment as for RQMs: $e \subseteq e'$ holds for two RQDs e and e' if and only if $\mathcal{L}(e) \subseteq \mathcal{L}(e')$. This is why we concentrate on containment of languages in the rest of this section.

4.2. Containment of General RQDs

RQDs were originally introduced as a restriction of RQMs that enjoys much better query evaluation properties. In the light of this result, one might also hope for good behaviour when query containment is considered. Surprisingly, it is not the case, and containment is undecidable for RQDs, same as for RQMs. In fact, we will prove a stronger result that the universality problem for RQDs, defined below, is undecidable. Let $\Sigma[\mathcal{D}]^*$ denote the set of all data words over the alphabet Σ and set of data values $\mathcal{D}$.

UNIVERSALITY(RQD)	
Input:	An RQD e.
Question:	Does $\mathcal{L}(e) = \Sigma[\mathcal{D}]^*$?

The undecidability of this problem immediately implies that, given two RQDs e and e', checking whether $\mathcal{L}(e) \subseteq \mathcal{L}(e')$ is undecidable: indeed, we can just take $(a_1 \cup \ldots \cup a_k)^*$ with $\Sigma = \{a_1, \ldots, a_k\}$ as e (from containment) and e (from universality) as e' for a reduction.

Theorem 4.3. *The problem* UNIVERSALITY(RQD) *is undecidable.*

Proof. The proof is by reduction of the *Post correspondence problem (PCP)*, which is well-known to be undecidable. The proof borrows from [28], where the universality of register automata was shown to be undecidable.

A *PCP instance* is a set of pairs (u, u') of non-empty words over a finite alphabet Γ. A *solution* of a PCP instance I is a sequence

$$(u_1, u'_1), \ldots, (u_k, u'_k)$$

of pairs from I (possibly with repetitions) such that

$$u_1 \cdots u_k = u'_1 \cdots u'_k.$$

PCP is the problem to check whether a PCP instance has a solution, and it is undecidable.

Let I be a PCP instance. In the rest of the proof we will show how to construct an RQD e over some alphabet Σ such that $\mathcal{L}(e) = \Sigma[\mathcal{D}]^*$ if and only if I has a solution. Throughout the reduction we will use the following notation: given a data word $w = d_1 a_1 d_2 \ldots a_{n-1} d_n$, we denote by $\bar{w}$ the reversal of w, that is, $\bar{w} = d_n a_{n-1} \ldots d_2 a_1 d_1$.

Let $\Sigma = \Gamma \cup \{\$, \#\}$, where $\$$ and $\#$ are two special symbols not in Γ. Next we describe restrictions on a word over this alphabet and then show that, on the one hand, words under these restrictions encode solutions of PCP instance I, and, on the other, the negation of these restrictions can be written as an RQD. This implies that I has a solution if and only if the RQD is not universal.

To this end, consider a data word $w \# \bar{w}'$ over Σ such that

$$\begin{aligned} w \;=\; 0 \;\; \$c_1 \; a_1 d_1 \cdots a_{n_1} d_{n_1} \;\; \$c_2 \; a_{n_1+1} d_{n_1+1} \cdots a_{n_1+n_2} d_{n_1+n_2} \cdots\cdots \\ \$c_k \; a_{n_1+\cdots+n_{k-1}+1} d_{n_1+\cdots+n_{k-1}+1} \cdots a_{n_1+\cdots+n_k} d_{n_1+\cdots+n_k}, \end{aligned}$$

and

$$\begin{aligned} w' = \; 0 \;\; \$c'_1 \; a'_1 d'_1 \cdots a'_{m_1} d'_{m_1} \;\; \$c'_2 \; a'_{m_1+1} d'_{m_1+1} \cdots a'_{m_1+m_2} d'_{m_1+m_2} \cdots\cdots \\ \$c'_\ell \; a'_{m_1+\cdots+m_{\ell-1}+1} d'_{m_1+\cdots+m_{\ell-1}+1} \cdots a'_{m_1+\cdots+m_\ell} d'_{m_1+\cdots+m_\ell}, \end{aligned}$$

with numbers $k \geq 1$, $\ell \geq 1$, $n_i \geq 1$ for $1 \leq i \leq k$, $m_i \geq 1$ for $1 \leq i \leq \ell$, data values 0, c_i for $1 \leq i \leq k$, c'_i for $1 \leq i \leq \ell$, d_i for $1 \leq i \leq n$ and $n = n_1 + \cdots + n_k$, and d'_i for $1 \leq i \leq m$ and $m = m_1 + \cdots + m_\ell$, as well as labels a_i from Γ for $1 \leq i \leq n$ and a'_i for $1 \leq i \leq m$, such that the following conditions hold:

1. all c_i are pairwise different and all d_i are pairwise different;
2. all c'_i are pairwise different and all d'_i are pairwise different;
3. $c_1 = c'_1$ and $c_k = c'_\ell$;
4. $d_1 = d'_1$ and $d_n = d'_m$;
5. if $c_i = c'_j$ then $c_{i+1} = c'_{j+1}$ for all $1 \leq i < k$, $1 \leq j < \ell$;
6. if $d_i = d'_j$ then $d_{i+1} = d'_{j+1}$ for all $1 \leq i < n$, $1 \leq j < m$;
7. if $d_i = d'_j$ then $a_i = a'_j$ for all $1 \leq i \leq n$, $1 \leq j \leq m$;
8. $a_{n_1+\cdots+n_{i-1}+1} \cdots a_{n_1+\ldots+n_i}$ is the first word of a pair in I for all $1 \leq i \leq k$;
9. if $c_i = c'_j$, then $(a_{n_1+\cdots+n_{i-1}+1} \cdots a_{n_1+\cdots+n_i}, a'_{m_1+\cdots+m_{j-1}+1} \cdots a'_{m_1+\cdots+m_j}) \in I$ for all $1 \leq i \leq k$, $1 \leq j \leq \ell$;

Having the restrictions on a data word at hand, next we show that every data word satisfying them encodes a solution of I, and, other way round, every solution can be represented by such a word.

First, note that conditions 1, 2, 4 and 6 force that $d_1 \cdots d_n = d'_1 \cdots d'_m$ and, in particular, $n = m$. Hence, by condition 7, $a_1 \cdots a_n = a'_1 \cdots a'_m$. Similarly, conditions 1, 2, 3 and 5 force that $c_1 \cdots c_k = c'_1 \cdots c'_\ell$ and, in particular, $k = \ell$. Moreover, conditions 8 and 9 say that the word of labels from Γ after any c_i is the first component of a pair in I, and it pairs with the word after c'_j that is equal to c_i (which is only possible if $i = j$). Therefore, the sequence of these pairs, for $1 \leq i \leq k$, is a solution of I.

For the other direction, let $(u_1, u'_1), \ldots, (u_k, u'_k)$ be a solution of I. Then we can take $a_1 \cdots a_n = a'_1 \cdots a'_m = u_1 \cdots u_k = u'_1 \cdots u'_k$, $c_i = c'_i = i$ for all $1 \leq i \leq k$ and $d_j = d'_j = j$ for all $1 \leq j \leq n$ to satisfy all the conditions.

We are left to construct an RQD e over Σ that accepts a data word such that it is either not of the form $w\#\bar{w}'$ as above, or at least one of conditions 1–9 is not satisfied. To this end, RQD e is the union $e_0 \cup \cdots \cup e_9$ of the following RQDs, using the usual abbreviation Δ for the regular expression $b_1 \cup \cdots \cup b_\ell$ over any alphabet $\Delta = \{b_1, \ldots, b_\ell\}$, and the abbreviation $\Sigma_\#$ for $\Sigma \setminus \{\#\}$.

– Expression e_0 accepts all data words whose projection to labels Σ do not satisfy the regular expression $(\$\Gamma^+)^+\#(\Gamma^+\$)^+$. As this is an ordinary regular expression, it is straightforward to define its complement.

– Expression e_1 accepts all data words such that condition 1 does not hold. To

this end, it is the union of

$$\Sigma_{\#}^* \cdot \$ \cdot (\Sigma_{\#}^* \cdot \$)_{=} \cdot \Sigma^*,$$
$$\Sigma_{\#}^* \cdot \Gamma \cdot (\Sigma_{\#}^* \cdot \Gamma)_{=} \cdot \Sigma^*,$$

which look for the same data value immediately after two $\$$ before $\#$ and immediately after two labels from Γ before $\#$, respectively.

– Expression e_2 accepts all data words such that condition 2 does not hold. Symmetrically to e_1, it is the union of

$$\Sigma^* \cdot (\$ \cdot \Sigma_{\#}^*)_{=} \cdot \$ \cdot \Sigma_{\#}^*,$$
$$\Sigma^* \cdot (\Gamma \cdot \Sigma_{\#}^*)_{=} \cdot \Gamma \cdot \Sigma_{\#}^*.$$

– Expression e_3 accepts all data words such that condition 3 does not hold. To this end, it is the union of

$$\$ \cdot (\Sigma^*)_{=} \cdot \$,$$
$$\Sigma_{\#}^* \cdot \$ \cdot (\Gamma^+ \cdot \# \cdot \Gamma^+)_{=} \cdot \$ \cdot \Sigma_{\#}^*.$$

– Expression e_4 accepts all data words such that condition 4 does not hold. To this end, it is the union of

$$\$ \cdot \Gamma \cdot (\Sigma^*)_{=} \cdot \Gamma \cdot \$,$$
$$\Sigma_{\#}^* \cdot (\#)_{=} \cdot \Sigma_{\#}^*.$$

– Expression e_5 accepts all data words such that condition 5 does not hold. To this end, it is

$$\Sigma_{\#}^* \cdot \$ \cdot (\Gamma^+ \cdot \$ \cdot (\Sigma^*)_{\neq} \cdot \$ \cdot \Gamma^+)_{=} \cdot \$ \cdot \Sigma_{\#}^*.$$

– Expression e_6 accepts all data words such that condition 6 does not hold. Similarly to e_5, it is

$$\Sigma_{\#}^* \cdot \Gamma \cdot (\$^* \cdot \Gamma \cdot (\Sigma^*)_{\neq} \cdot \Gamma \cdot \$^*)_{=} \cdot \Gamma \cdot \Sigma_{\#}^*.$$

– Expression e_7 accepts all data words such that condition 7 does not hold. To this end, it is the union of

$$\Sigma_{\#}^* \cdot a_i \cdot (\Sigma^*)_{=} \cdot a_j \cdot \Sigma_{\#}^*$$

for all $a_i, a_j \in \Gamma$ such that $a_i \neq a_j$.

– Expression e_8 accepts all data words such that condition 8 does not hold. To this end, it is the following RQD, where $e^{\neg}$ is the regular expression accepting the language $\Gamma^+ \setminus \{u_1, \ldots, u_p\}$ for $u_1, \ldots, u_p$ all the first components of pairs in I:

$$\Sigma_{\#}^* \cdot \$ \cdot e^{\neg} \cdot (\$ \cup \#) \cdot \Sigma_{\#}^*.$$

– Expression e_9 accepts all data words such that condition 9 does not hold. To this end, it is the union of the following RQDs for each first component u of a pair in I, where $e_u^{\neg}$ is the regular expression accepting the language $\Gamma^+ \setminus \{u'_1, \ldots, u'_p\}$ for $u'_1, \ldots, u'_p$ all the second components of pairs in I having u as the first component:

$$\Sigma_{\#}^* \cdot \$ \cdot (u \cdot (\$ \cdot \Gamma^+)^* \cdot \# \cdot (\Gamma^+ \cdot \$)^* \cdot e_u^{\neg})_{=} \cdot \$ \cdot \Sigma_{\#}^*.$$

By this construction, e represents exactly those data words for which the restrictions above do not hold. Therefore, the PCP instance I has no solution if and only if $\mathcal{L}(e) = \Sigma[\mathcal{D}]^*$, as required. □

Having this result at hand, we obtain our first result on containment of RQDs.

Corollary 4.4. *The problem* CONTAINMENT(RQDs) *is undecidable.*

4.3. Positive RQDs and pdt-Automata

The negative result of Corollary 4.4 naturally opens the search for subclasses of RQDs with decidable containment problem. Similarly to positive RQMs, we now consider the class of *positive RQDs*, that is, RQDs where subexpressions of the form $e_{\neq}$ are not allowed. We can obtain a positive RQM from a positive RQD by the procedure described above that transforms an RQD into an RQM. Hence, we again have an inclusion of the corresponding classes, and from Theorem 3.11 we conclude that containment of positive RQDs is decidable and in EXPSPACE. However, this time we show that the complexity of containment is reduced to PSPACE-complete, which is the best possible bound in light of the PSPACE lower bound for plain RPQs.

In the previous section we showed that RQMs can be captured by register automata, and then used this fact to establish the matching upper bound on containment of positive RQMs. We follow a similar strategy for positive RQDs, and begin with introducing a subclass of register automata that captures this formalism. We call this class *automata with positive data tests*, or pdt-automata for short.

Intuitively, pdt-automata impose a restriction on register automata that guarantee that the comparisons behave in a "first in, last out" discipline, in the same way as positive RQDs restrict RQMs. In particular, they have the following special policy for manipulation of registers:

1. only tests for conjunctions of equalities are allowed,
2. the registers are ordered and arranged in a stack,
3. (positive) comparisons of the current data value can be performed only with the value stored in the register on the top of the stack, and, moreover, after such a comparison the stored value is lost and the register becomes unused (i.e., the value is popped off), and
4. the current data value can be stored only in the currently unused register just above the top of the stack (i.e., pushed into).

To define pdt-automata formally, it is convenient to have the following notions. An *action* is a pair $\langle m_1, m_2 \rangle$ of nonnegative numbers m_1 and m_2 (we choose to denote pairs of numbers as actions to improve the readability of the machinery we use in this section). Let $\mathcal{M}$ denote the set of all such actions.

Definition 4.5. *A register automaton $\langle P, r_s, q_f, \gamma \rangle$ with $\gamma = (\delta, \alpha)$ and registers $X = \{x_1, \ldots, x_\ell\}$ over labels Σ is a* pdt-automaton *if there are assignments Depth : $P \to \{0, \ldots, \ell\}$ and Action : $\delta \to \mathcal{M}$ such that the following holds:*

- *Depth$(r_s) = 0$ and Depth$(q_f) = 0$;*
- *for every data transition $(r, c, Y, q) \in \delta$ with Action$(r, c, Y, q) = \langle m_1, m_2 \rangle$, we have that:*
  - *Depth$(r) - m_1 \geq 0$,*
  - *Depth$(r) - m_1 + m_2 =$ Depth(q),*
  - *$c = \bigwedge x_i^{=}$, where i ranges as Depth$(r) \geq i >$ Depth$(r) - m_1$;*
  - *$Y = \{x_i \mid$ Depth$(r) - m_1 < i \leq$ Depth$(q)\}$;*
- *Depth$(q) =$ Depth(r) for every word transition $(q, a, r) \in \alpha$.*

Therefore, assignments *Depth* and *Action* impose certain restrictions on how a pdt-automaton can manipulate registers. These restrictions reflect the difference between RQMs and RQDs: in the latter all comparisons are stack-based, in the sense that one cannot store two data values to registers x_1 and then x_2, but then compare them to current data values in the same order (e.g., one RQDs cannot define languages of data words of the form $d_1 a d_2 a d_1 a d_2 a \cdots$, which are definable

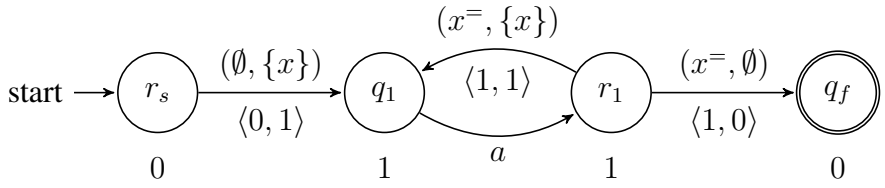

Figure 4: Pdt-automaton equivalent to the RQD in Example 4.6; *Depth* and *Action* labels are given below states and data transitions, respectively.

by RQDs). Thus, to simulate RQDs with register automata we need to impose the condition that one can never process a condition of the form $x_i^=$ unless i is actually the last register that we have used to store a data value. The depth function is key for this restriction, as it represents the index of the last register in which we have stored a data value. Moreover, note that equalities in RQDs are restricted in scope, so one cannot compare a value in a word with two values in different positions that are farther away in the word. Thus, to simulate RQDs we need to restrict register automata so that, after a condition $x^=$, one cannot compare the contents of register x again until one re-stores another value there. In order to enforce this, we assign an action to each transition, which is a pair where the first component represents which registers are being used in the condition, and the second component represents which registers are being used to store the current data value. For example, an action of the form $\langle m, 0\rangle$ states that m topmost registers (relative to the number *Depth* of currently used registers) are compared in the condition, and that the current data value is not stored anywhere; an action of the form $\langle 0, m\rangle$ states that this transition has no condition but the current data value is stored in the next m available registers; and $\langle m_1, m_2\rangle$ is a combined action, which first compares with the value in the topmost m_1 registers, and then stores the value in the next m_2 available registers once we have freed the m_1 registers used in the comparison.

Note that by this definition assignments *Depth* and *Action* unambiguously define the equality conditions c and assigned registers Y of all data transitions of an pdt-automaton. Hence, we will write $(r, \langle m_1, m_2\rangle, q)$ instead of (r, c, Y, q) for a data transition with an action $\langle m_1, m_2\rangle$, that is, assume that $\delta \subseteq R \times \mathcal{M} \times Q$.

Example 4.6. Consider again the RQD $(a_=)^+$ from Example 4.2. The pdt-automaton in Figure 4 accepts the same language as this RQD, where *Depth* and *Action* labels are given below states and data transitions.

When reading the first data value, we go from the state r_s, with depth 0, to the

state q_1, with depth 1, via the action $\langle 0, 1\rangle$, that is, we are storing a data value in the first register x_1, and now this register is on the top of the stack and ready to be used in a comparison. We then move from q_1 to r_1 by reading the label a. Here we have two options. First, we can decide that this is the end of the word and follow the transition to q_f by using the comparison $x_1^=$, that is, parsing the action $\langle 1, 0\rangle$, thus bringing the depth down to zero. Second, we can decide that we wish to continue processing the input by checking that the current data value equals to the one in x and at the same time storing this value into x_1 again in order to do further comparisons. This brings us back to the state q_1, where the process starts again. Note that in this transition we first reduce the depth to 0 (by using $x^=$), and then increase it back to 1 as reflected by the action $\langle 1, 1\rangle$. □

Remark 4.7. *Note that pdt-automata have a similar flavour as k-pebble automata of [28], which also force registers to be used in a stack-like manner. However, while we show that containment of pdt-automata is decidable (see Theorem 4.12), the problem is undecidable in the case of pebble automata [28], thus suggesting that the latter is much more expressive.*

Just as register automata capture RQMs, pdt-automata capture positive RQDs. The following proposition can be checked by immediate inspection of the construction in [25] for RQMs.

Proposition 4.8. *For every RQD e there exists an pdt-automaton $\mathbb{A}_e$ such that $\mathcal{L}(e) = \mathcal{L}(\mathbb{A}_e)$. Moreover, $\mathbb{A}_e$ can be constructed in time polynomial in $|e|$.*

4.4. Monoid of Actions

The definition of pdt-automata introduces actions, that is, pairs of non-negative numbers. Our algorithm for containment of RQDs makes use of the following binary composition operator $\circ$ on the set $\mathcal{M}$ of actions:

$$\langle m_1, m_2\rangle \circ \langle m_1', m_2'\rangle = \begin{cases} \langle m_1, m_2 - m_1' + m_2'\rangle, & \text{if } m_2 \geq m_1', \\ \langle m_1 - m_2 + m_1', m_2'\rangle, & \text{otherwise.} \end{cases}$$

It is now not difficult to show that the algebraic structure $\langle \mathcal{M}, \circ\rangle$ is a monoid. The proof is just a case by case analysis that we leave for the appendix.

Proposition 4.9. *Operation $\circ$ is associative and $\langle 0, 0\rangle$ is neutral for $\circ$.*

We take advantage of the associativity of $\circ$ to omit parentheses when writing compositions of several actions. The following property of the neutral element is immediate.

Corollary 4.10. *Let $\mu_1, \ldots, \mu_n$ be actions such that $\mu_j \circ \cdots \circ \mu_k = \langle 0, 0\rangle$ for some j and k, $1 \leq j \leq k \leq n$. Then $\mu_1 \circ \cdots \circ \mu_n = \langle 0, 0\rangle$ if and only if $\mu_1 \circ \cdots \circ \mu_{j-1} \circ \mu_{k+1} \circ \cdots \circ \mu_n = \langle 0, 0\rangle$.*

Finally, the key property of the monoid $\langle \mathcal{M}, \circ\rangle$ that we use is the following: the composition of actions along some consecutive transitions in a run of a pdt-automaton is $\langle m_1, m_2\rangle$ if and only if, first, the difference of the depths of states in the beginning and in the end of the sequence are $m_1 - m_2$, and, second, the smallest number of a register manipulated along the transitions is exactly the depth in the beginning minus $m_1 - 1$. The following lemma formalises this property; its proof is also in the appendix.

Lemma 4.11. *Let*

$$[r_0, \lambda_0], [q_1, \lambda_1], [r_1, \lambda_1], \ldots, [r_{n-1}, \lambda_{n-1}], [q_n, \lambda_n]$$

be a run of a pdt-automaton, where, for every $1 \leq i \leq n$, configuration $[q_i, \lambda_i]$ is reachable from $[r_{i-1}, \lambda_{i-1}]$ by a transition with action μ_i. For every j and k, $1 \leq j \leq k \leq n$, the numbers m_1 and m_2 in $\langle m_1, m_2\rangle = \mu_j \circ \cdots \circ \mu_k$ are such that

- *Depth$(r_{j-1}) - m_1 + m_2 =$ Depth(q_k),*
- *Depth$(r_{i-1}) - m_1' \geq$ Depth$(r_{j-1}) - m_1$ for any i, $j \leq i \leq k$, with $\mu_i = \langle m_1', m_2'\rangle$,*
- *there exists i, $j \leq i \leq k$, such that Depth$(r_{i-1}) - m_1' =$ Depth$(r_{j-1}) - m_1$ for $\mu_i = \langle m_1', m_2'\rangle$ (i.e., the inequality above becomes an equality).*

4.5. Containment of Positive RQDs

In what follows we prove that containment of pdt-automata can be done in PSPACE, which implies, together with Proposition 4.8, that containment of RQDs over data graphs is in PSPACE as well (recall that the matching lower bound follows from the PSPACE-hardness of containment of usual RPQs).

Theorem 4.12. *The problem of deciding whether $\mathcal{L}(\mathbb{A}) \subseteq \mathcal{L}(\mathbb{A}')$ for pdt-automata $\mathbb{A}$ and $\mathbb{A}'$ is in* PSPACE.

Proof. The idea of the proof is to construct a transition system (i.e., an NFA over a special alphabet) $\mathbb{U}(\mathbb{A}, \mathbb{A}')$ whose language is nonempty if and only if there is a data word that is accepted by $\mathbb{A}$ but not by $\mathbb{A}'$ (in other words, the language of the transition system corresponds to all witnesses for the non-containment of $\mathbb{A}$ in

$\mathbb{A}'$), and then show that the problem of deciding whether the language of $\mathbb{U}(\mathbb{A}, \mathbb{A}')$ is empty is in PSPACE. This last result suffices for the proof: if the language of $\mathbb{U}(\mathbb{A}, \mathbb{A}')$ is empty then there is no word that is accepted by $\mathbb{A}$ and not by $\mathbb{A}'$, and thus it must be that the language of $\mathbb{A}$ is contained in the language of $\mathbb{A}'$.

The idea of using such a transition system is not new, and is commonly used when analysing standard finite automata. However, in our case we cannot construct such a system in the direct way, because the number of data values in a data word is not bounded, and thus to simulate all possible runs for $\mathbb{A}'$ directly we would need to search over a space of configurations that can be arbitrarily big. Therefore, our transition system does not work with data values, but rather with a finite number of their representatives. Moreover, to archive desired space bounds, these representatives are compressed by means of an involved data structure.

The definition and functionality of $\mathbb{U}(\mathbb{A}, \mathbb{A}')$ are quite technical. Therefore, we proceed in the following two steps:

1. we first show how to construct an auxiliary transition system $\mathbb{S}(\mathbb{A}, \mathbb{A}')$, which possesses all the properties required for $\mathbb{U}(\mathbb{A}, \mathbb{A}')$ except that its state space is exponential, and hence its emptiness can be decided in EXPSPACE;
2. then we explain how to reduce, by means of transforming $\mathbb{S}(\mathbb{A}, \mathbb{A}')$ to $\mathbb{U}(\mathbb{A}, \mathbb{A}')$, the space usage of the algorithm from exponential to polynomial and prove that it does not affect the correctness of the construction.

Note that the upper complexity bound obtained at the first step does not give any improvement in comparison to the general case of register automata as in Theorem 3.11.

Along the proof we assume that $\mathbb{A} = \langle P, r_s, q_f, \gamma \rangle$ is a pdt-automaton with registers $X = \{x_1, \ldots, x_\ell\}$, states $P = R \cup Q$ for data and word states R and Q, respectively, and transition relation $\gamma = (\delta, \alpha)$ for data and word transitions δ and α, respectively; similarly, $\mathbb{A}' = \langle P', r'_s, q'_f, \gamma' \rangle$ is a pdt-automaton with registers $X' = \{x'_1, \ldots, x'_{\ell'}\}$, states $P' = R' \cup Q'$, and transition relation $\gamma' = (\delta', \alpha')$.

Step 1. For the beginning, assume that both $\mathbb{A}$ and $\mathbb{A}'$ are such that in each of their data transitions there is at most one register manipulation, that is, all their actions are among $\langle 0, 1 \rangle$, $\langle 1, 0 \rangle$, and $\langle 0, 0 \rangle$. A witness for non-containment of $\mathbb{A}$ in $\mathbb{A}'$ could be an accepting run of $\mathbb{A}$ on a data word w such that $\mathbb{A}'$ does not have an accepting run on w. However, we do not need to check all such data words: instead, to comprehensively model the behaviour of $\mathbb{A}$ it is enough to specify, for each data value in a word, whether it is equal to some data values currently stored in the stack of registers of $\mathbb{A}$, and assume that it is a fresh one otherwise; for $\mathbb{A}'$, it

is enough to keep track which registers of $\mathbb{A}'$ have same values as registers of $\mathbb{A}$. Moreover, since $\mathbb{A}$ and $\mathbb{A}'$ are pdt-automata with the special structure, it is enough to specify if a current data value is equal to the value on the top of the stack of registers in $\mathbb{A}$ or not; regarding $\mathbb{A}'$, we only need to track the corresponding runs that do not store any values not in the registers of $\mathbb{A}$. Therefore, a simple transition system for these type of automata would have the following structure:

- each state consists of a state of $\mathbb{A}$ and a set of states of $\mathbb{A}'$, each of which has a stack of reactions, that is, Boolean flags indicating whether the data values in the registers of $\mathbb{A}$ were pushed to the registers of $\mathbb{A}'$ or not;
- the initial state consists of the initial states of $\mathbb{A}$ and $\mathbb{A}'$ (the latter as a singleton set), while the final state conists of the final state of $\mathbb{A}$ and all subsets of states of $\mathbb{A}'$ that do not contain the final state;
- each transition corresponds to a transition of $\mathbb{A}$, while the set of states of $\mathbb{A}'$ updates taking the data manipulations into account:
  - if $\mathbb{A}$ passes action $\langle 0, 1\rangle$ then $\mathbb{A}'$ should remember whether it pushes the data value or not;
  - if $\mathbb{A}$ passes action $\langle 1, 0\rangle$ then $\mathbb{A}'$ should pop (and compare) the data value only if it was pushed when the value was pushed in $\mathbb{A}$;
  - if $\mathbb{A}$ passes action $\langle 0, 0\rangle$ then $\mathbb{A}'$ should neither push nor pop as well.

Note that in this simple case we do not to remember anything about the stack of $\mathbb{A}$, because its depth is uniquely defined by the state. This is not the case in general, because the current data value can be pushed to register stack several times. However, such subsets must contain only consecutive registers according to the order of automaton $\mathbb{A}$. Moreover, the reaction of an accepting run of $\mathbb{A}'$ to pushing a value into the stack of registers of $\mathbb{A}$ may be arbitrary: for example, expression $((e_1)_{=}e_2)_{=}$ is contained in $(e_1(e_2)_{=})_{=}$ (in fact, they are equivalent), despite the fact that after parsing e_1 the action of the former is $\langle 1, 0\rangle$, that is, the data value is popped off, while the action of the latter is $\langle 0, 1\rangle$ that is, the value is pushed into. Next we show how to generalise the intuition to the case of pdt-automata with arbitrary actions. We also formally prove the correctness of the resulting algorithm.

We define a transition system $\mathbb{S}(\mathbb{A}, \mathbb{A}')$, which is an NFA working on words of a special form, as follows.

System $\mathbb{S}(\mathbb{A}, \mathbb{A}')$ works on *action words*, that is, words of the form $\mu_1 a_1 \mu_2 a_2 \cdots a_{n-1} \mu_n$, where all a_i are labels in Σ, and all μ_i are actions in $\mathcal{M}$.

Same as in the simplified example above, the actions in such a word are the actions passed by pdt-automaton $\mathbb{A}$. So, such a word represents a data word in which all the data values are different unless they are forced to be equal by the actions.

The states S of $\mathbb{S}(\mathbb{A}, \mathbb{A}')$, called *instantaneous descriptions* (*IDs*) in order to avoid confusion with the states of register automata, are pairs $[[p, \mathtt{S}], \mathcal{F}]$ where $[p, \mathtt{S}]$ and $\mathcal{F}$ are defined as follows (note that they generalise the construction in step 1):

- the pair $[p, \mathtt{S}]$, called a *state-stack tuple*, or *s-s tuple* for short, consists of a state $p \in P$ and a stack $\mathtt{S}$ of positive numbers such that $\Sigma_{1 \leq i \leq |\mathtt{S}|} \mathtt{S}[i] = \mathit{Depth}(p)$, where $|\mathtt{S}|$ is the size of $\mathtt{S}$ and $\mathtt{S}[i]$ is its i'th element; and
- $\mathcal{F}$ is a possibly empty set of pairs $[p', \mathtt{R}]$, called *state-reaction tuples* with respect to $\mathtt{S}$, or *s-r tuples*, where $p' \in P'$ and $\mathtt{R}$ is a stack of actions in $\mathcal{M}$ (i.e., pairs of non-negative numbers) such that $|\mathtt{R}| = |\mathtt{S}|$.

The initial ID of $\mathbb{S}(\mathbb{A}, \mathbb{A}')$ is $[[r_s, \mathtt{S}_\emptyset], \{[r'_s, \mathtt{R}_\emptyset]\}]$, where $\mathtt{S}_\emptyset$ and $\mathtt{R}_\emptyset$ are empty stacks. The final IDs are all $[[q_f, \mathtt{S}_\emptyset], \mathcal{F}]$ such that $\mathcal{F}$ is a set of s-r tuples that does not contain $[q'_f, \mathtt{R}_\emptyset]$.

Intuitively, IDs of $\mathbb{S}(\mathbb{A}, \mathbb{A}')$ store the required information about a run of $\mathbb{A}$ on a data word represented by an action word as well as all runs of $\mathbb{A}'$ on this data word: an s-s tuple keeps the sizes of groups of registers in $\mathbb{A}$ that must store the same data value, while s-r tuples keep the "reactions" of $\mathbb{A}'$ as in the simplified case above.

The transitions of $\mathbb{S}(\mathbb{A}, \mathbb{A}')$ over labels in Σ are standard, but defining the transitions over actions requires a bit more of work. Intuitively, there are four types of transitions, each one representing one of the four different ways for manipulating registers in a transition of $\mathbb{A}$ in a run on a data word: no manipulation, pushing the current fresh data value into the stack of registers several times, popping the value off in such a way that the value is not in the stack any more; popping it off and, possibly, pushing it back so it is left in the stack. In turn, s-r tuples in $\mathtt{R}$ propagate all possible transitions of $\mathbb{A}'$ accordingly:

- if $\mathbb{A}$ does not manipulate registers, then $\mathbb{A}'$ should not do so as well;
- if $\mathbb{A}$ pushes a data value but does not pop anything before, then $\mathbb{A}'$ should store the corresponding action in stack $\mathtt{R}$;
- if $\mathbb{A}$ pops all copies of the data value, then $\mathbb{A}'$ should clear from this data value as well;

– if $\mathbb{A}$ pops, but not all copies, then $\mathbb{A}'$ should continue to accumulate the overall reacting action on the top of the stack.

Therefore, in the definition of the transition relation of $\mathbb{S}(\mathbb{A}, \mathbb{A}')$ we need the following notions. Given an s-s tuple $[p, \mathtt{S}]$ and a number m, $0 < m \leq \mathit{Depth}(p)$, the *level* $\mathit{Lev}(\mathtt{S}, m)$ of $\mathtt{S}$ and m is the number satisfying the inequalities

$$\Sigma_{\mathit{Lev}(\mathtt{S},m)<i\leq|\mathtt{S}|}\mathtt{S}[i] < m \leq \Sigma_{\mathit{Lev}(\mathtt{S},m)\leq i\leq|\mathtt{S}|}\mathtt{S}[i],$$

and the *remainder* $\mathit{Rem}(\mathtt{S}, m)$ is the number

$$\Sigma_{\mathit{Lev}(\mathtt{S},m)\leq i\leq|\mathtt{S}|}\mathtt{S}[i] - m.$$

Intuitively, if m is the number of registers whose contents are compared in the current transition, then $\mathit{Lev}(\mathtt{S}, m)$ is the group in $\mathtt{S}$ to which the last (i.e., the deepest) compared register belongs, and the remainder $\mathit{Rem}(\mathtt{S}, m)$ is the number of registers left non-compared in this group.

Formally, the transition relation $\gamma_{\mathbb{S}} \subseteq S \times (\mathcal{M} \cup \Sigma) \times S$ of system $\mathbb{S}(\mathbb{A}, \mathbb{A}')$ is defined as follows, where cases 1(a)–1(d) correspond to the four ways of register manipulation described above:

1. $([[r, \mathtt{S}_1], \mathcal{F}_1], \mu, [[q, \mathtt{S}_2], \mathcal{F}_2]) \in \gamma_{\mathbb{S}}$ for a data state r, word state q, stacks $\mathtt{S}_1$ and $\mathtt{S}_2$, sets of s-r tuples $\mathcal{F}_1$ and $\mathcal{F}_2$, and action $\mu \in \mathcal{M}$ if $(r, \mu, q) \in \delta$ and one of the following holds:

   (a) $\mu = \langle 0, 0\rangle$, $\mathtt{S}_1 = \mathtt{S}_2$, and $\mathcal{F}_2$ consists of all s-r tuples $[q', \mathtt{R}_2]$ for each of which there is $[r', \mathtt{R}_1] \in \mathcal{F}_1$ with $(r', \langle 0, 0\rangle, q') \in \delta'$ and $\mathtt{R}_1 = \mathtt{R}_2$;

   (b) $\mu = \langle 0, m_2\rangle$ for $m_2 > 0$, $\mathtt{S}_2 = (\mathtt{S}_1[1], \ldots, \mathtt{S}_1[|\mathtt{S}_1|], m_2)$ (i.e., $\mathtt{S}_2$ can be obtained from $\mathtt{S}_1$ by pushing m_2 into), and $\mathcal{F}_2$ consists of the s-r tuples $[q', \mathtt{R}_2]$ for each of which there is $[r', \mathtt{R}_1] \in \mathcal{F}_1$ with $(r', \mu', q') \in \delta'$ and $\mathtt{R}_2 = (\mathtt{R}_1[1], \ldots, \mathtt{R}_1[|\mathtt{R}_1|], \mu')$;

   (c) $\mu = \langle m_1, 0\rangle$ for $m_1 > 0$ such that the reminder $\mathit{Rem}(\mathtt{S}_1, m_1) = 0$, $\mathtt{S}_2 = (\mathtt{S}_1[1], \ldots, \mathtt{S}_1[\mathit{Lev}(\mathtt{S}_1, m_1) - 1])$, and $\mathcal{F}_2$ consists of the s-r tuples $[q', \mathtt{R}_2]$ for each of which there is $[r', \mathtt{R}_1] \in \mathcal{F}_1$ with $(r', \mu', q') \in \delta'$, $\mathtt{R}_2 = (\mathtt{R}_1[1], \ldots, \mathtt{R}_1[\mathit{Lev}(\mathtt{S}_1, m_1) - 1])$ and

   $$\mathtt{R}_1[\mathit{Lev}(\mathtt{S}_1, m_1)] \circ \cdots \circ \mathtt{R}_1[|\mathtt{R}_1|] \circ \mu' = \langle 0, 0\rangle; \tag{3}$$

   (d) $\mu = \langle m_1, m_2\rangle$ for $m_1 > 0$ such that at least one of m_2 and $\mathit{Rem}(\mathtt{S}_1, m_1)$ is not 0, $\mathtt{S}_2 = (\mathtt{S}_1[1], \ldots, \mathtt{S}_1[\mathit{Lev}(\mathtt{S}_1, m_1) - 1], \mathit{Rem}(\mathtt{S}_1, m_1) + m_2)$, and

$\mathcal{F}_2$ consists of the s-r tuples $[q', \mathtt{R}_2]$ for each of which there is $[r', \mathtt{R}_1] \in \mathcal{F}_1$ with $(r', \mu', q') \in \delta'$ and

$$\mathtt{R}_2 = (\mathtt{R}_1[1], \ldots, \mathtt{R}_1[\mathit{Lev}(\mathtt{S}_1, m_1) - 1], \mathtt{R}_1[\mathit{Lev}(\mathtt{S}_1, m_1)] \circ \cdots \circ \mathtt{R}_1[|\mathtt{R}_1|] \circ \mu');$$

2. $([[q, \mathtt{S}_1], \mathcal{F}_1], a, [[r, \mathtt{S}_2], \mathcal{F}_2]) \in \gamma_{\mathbb{S}}$ for a word state q, data state r, stacks $\mathtt{S}_1$ and $\mathtt{S}_2$, sets of s-r tuples $\mathcal{F}_1$ and $\mathcal{F}_2$, and label a, if $(q, a, r) \in \alpha$, $\mathtt{S}_1 = \mathtt{S}_2$ and $\mathcal{F}_2$ consists of the s-r tuples $[r', \mathtt{R}_2]$ for each of which there is an s-r tuple $[q', \mathtt{R}_1]$ in $\mathcal{F}_1$ such that $(q', a, r') \in \alpha'$ and $\mathtt{R}_1 = \mathtt{R}_2$.

A *run* of $\mathbb{S}(\mathbb{A}, \mathbb{A}')$ on an action word $w = \mu_1 a_1 \mu_2 a_2 \cdots a_{n-1} \mu_n$ is a sequence $s'_0, s_1, s'_1, s_2, s'_2, \ldots, s'_{n-1}, s_n$ of IDs such that s'_0 is initial, $(s'_{i-1}, \mu_i, s_i) \in \gamma_{\mathbb{S}}$ for each i, $1 \leq i \leq n$, and $(s_i, a_i, s'_i) \in \gamma_{\mathbb{S}}$ for each i, $1 \leq i < n$. It is *accepting* if s_n is final, and in this case $\mathbb{S}(\mathbb{A}, \mathbb{A}')$ *accepts* w.

Next we show the correctness of the transition system, that is, prove that system $\mathbb{S}(\mathbb{A}, \mathbb{A}')$ accepts at least one action word if and only if $\mathcal{L}(\mathbb{A}) \not\subseteq \mathcal{L}(\mathbb{A}')$. In particular, Claim 4.13 states the forward direction of this equivalence, where we show that a data word represented by the action word, as explained above, witnesses the non-containment; next, Claim 4.14 states the backward direction.

Claim 4.13. *For each action word accepted by $\mathbb{S}(\mathbb{A}, \mathbb{A}')$ there is a data word accepted by $\mathbb{A}$ and rejected by $\mathbb{A}'$.*

Proof. Assume that $\mathbb{S}(\mathbb{A}, \mathbb{A}')$ accepts an action word $\mu_1 a_1 \mu_2 a_2 \cdots a_{n-1} \mu_n$, and let $s'_0, s_1, s'_1, s_2, s'_2, \ldots, s_{n-1}, s'_{n-1}, s_n$ be the corresponding accepting run with all s'_i of the form $[[r_i, \mathtt{S}_i], \mathcal{F}'_i]$ and all s_i of the form $[[q_i, \mathtt{S}_i], \mathcal{F}_i]$ such that $[[r_0, \mathtt{S}_0], \mathcal{F}'_0] = [[r_s, \mathtt{S}_\emptyset], \{[r'_s, \mathtt{R}_\emptyset]\}]$, $[q_n, \mathtt{S}_n] = [q_f, \mathtt{S}_\emptyset]$ and $\mathcal{F}_n$ does not contain any s-r tuple $[q'_f, \mathtt{R}_\emptyset]$. We construct a data word $w = d_1 a_1 d_2 a_2 \cdots a_{n-1} d_n$ and then prove that $w \in \mathcal{L}(\mathbb{A})$ while $w \notin \mathcal{L}(\mathbb{A}')$.

In w we take each label a_i from the action word, so to finish the definition of w we need to define each of the data values d_i. As said above, we need data values such that any two of them are different unless they are forced to be equal by the actions in the action word.

Formally, we first set d_i as a fresh data value whenever $\mu_i = \langle 0, 0 \rangle$. Next we we iterate over all k such that $\mathit{Rem}(\mathtt{S}_{k-1}, m_1) = 0$ for $\mu_k = \langle m_1, 0 \rangle$ and $m_1 > 0$ in the increasing order. Let k_1 be the smallest of these numbers. Since $\mathit{Rem}(\mathtt{S}_{k_1 - 1}, m_1) = 0$, there must exist a number $j < k_1$ such that $\mathit{Depth}(r_{j-1}) = \mathit{Depth}(q_{k_1})$. Let j_1 be the greatest of such numbers and d^* be a fresh data value. We then set $d_i = d^*$, for all $i \in [j_1, k_1]$ such that $\mu_i \neq \langle 0, 0 \rangle$.

Continuing with the iteration, let k_2 be the second smallest number such that $\mathit{Rem}(\mathtt{S}_{k_2-1}, m_1) = 0$ for $\mu_{k_2} = \langle m_1, 0\rangle$, $m_1 > 0$, and let j_2 be defined for k_2 as j_1 for k_1. There could be two cases:

- the intervals $[j_2, k_2]$ and $[j_1, k_1]$ are disjoint, in which case we define d_i for all i with $j_2 \leq i \leq k_2$ in exactly the same way as for $[j_1, k_1]$;
- $[j_2, k_2]$ contains $[j_1, k_1]$ (that is, $j_2 < j_1$); in this case we do the same as for $[j_1, k_1]$, except that we do not redefine d_i for $j_1 \leq i \leq k_1$.

Note that the case of $[j_2, k_2]$ and $[j_1, k_1]$ having non-empty intersection but $[j_2, k_2]$ does not contain $[j_1, k_1]$ is not possible by construction.

Next we prove that $w \in \mathcal{L}(\mathbb{A})$, by showing that the following sequence of configurations is an accepting run of $\mathbb{A}$ on w:

$$[r_0, \lambda_0], [q_1, \lambda_1], [r_1, \lambda_1], [q_2, \lambda_2], [r_2, \lambda_2], \ldots, [q_{n-1}, \lambda_{n-1}], [r_{n-1}, \lambda_{n-1}], [q_n, \lambda_n],$$

where data states r_i and word states q_i are taken from the run

$$s'_0, s_1, s'_1, s_2, s'_2, \ldots, s_{n-1}, s'_{n-1}, s_n,$$

$\lambda_0 = \bot$ and λ_i, for $i > 0$, assigns registers

$$x_{\mathit{Depth}(r_{i-1})-m_1+1}, \ldots, x_{\mathit{Depth}(r_{i-1})-m_1+m_2}$$

to d_i, where $\mu_i = \langle m_1, m_2\rangle$, and all other registers as λ_{i-1}.

Since $([[q_i, \mathtt{S}_i], \mathcal{F}_i], a_i, [[r_i, \mathtt{S}_i], \mathcal{F}'_i]) \in \gamma_{\mathbb{S}}$, we have that $(q_i, a_i, r_i) \in \alpha$, and therefore $[q_i, \lambda_i] \Rightarrow^{\mathbb{A}}_{a_i} [r_i, \lambda_i]$ for all i. We also know that $\lambda_0 = \bot$, $r_0 = r_s$, and $q_n = q_f$. Hence, all that is left to prove is that $[r_{i-1}, \lambda_{i-1}] \Rightarrow^{\mathbb{A}}_{d_i} [q_i, \lambda_i]$ for every i, and we do it by showing that the transition (r_{i-1}, μ_i, q_i) in δ, existing by the fact that $([[r_{i-1}, \mathtt{S}_{i-1}], \mathcal{F}'_{i-1}], \mu_i, [[q_i, \mathtt{S}_i], \mathcal{F}_i]) \in \gamma_{\mathbb{S}}$, is applicable in this case as well. In fact, λ_i differs from λ_{i-1} only on registers $x_{\mathit{Depth}(r_{i-1})-m_1+1}, \ldots, x_{\mathit{Depth}(r_{i-1})-m_1+m_2}$, for $\mu_i = \langle m_1, m_2\rangle$, and contains d_i in all these registers by construction, so we only need to show that λ_{i-1} contains d_i in all $x_{\mathit{Depth}(r_{i-1})}, \ldots, x_{\mathit{Depth}(r_{i-1})-m_1+1}$. If $m_1 = 0$ then the check is trivial, so in what follows we assume that $m_1 > 0$.

Consider first the case when $j_1 \leq i \leq k_1$, where k_1 is the smallest number such that $\mathit{Rem}(\mathtt{S}_{k_1-1}, m_1) = 0$ for $\mu_{k_1} = \langle m_1, 0\rangle$, $m_1 > 0$, and j_1 is the greatest number such that $j_1 < k_1$ and $\mathit{Depth}(r_{j_1-1}) = \mathit{Depth}(q_{k_1})$. By construction, $d_i = d_{j_1}$. By the requirements for the run of $\mathbb{S}(\mathbb{A}, \mathbb{A}')$, $\mathit{Depth}(r_{i_1-1}) - m'_1 \geq \mathit{Depth}(r_{j_1-1})$ for any i_1, $j_1 \leq i_1 \leq k_1$, with $\mu_{i_1} = \langle m'_1, m'_2\rangle$. In particular, $\mathit{Depth}(r_{i-1}) - m_1 \geq \mathit{Depth}(r_{j_1-1})$, that is, all the registers tested on step i are above $\mathit{Depth}(r_{j_1-1})$. On

the one hand, all the values in these registers are written after step j_1; on the other, the only data value that is written between j_1 and i is d_{j_1}. In other words, λ_{i-1} contains d_i in all $x_{\mathit{Depth}(r_{i-1})}, \ldots, x_{\mathit{Depth}(r_{i-1})-m_1+1}$, as required.

Consider now the case when $j_1 \leq i \leq k_1$ does not hold, but $j_2 \leq i \leq k_2$, where k_2 is the second smallest number such that $\mathit{Rem}(\mathtt{S}_{k_2-1}, m_1) = 0$ for $\mu_{k_2} = \langle m_1, 0\rangle$, $m_1 > 0$, and j_2 is the greatest number such that $j_2 < k_2$ and $\mathit{Depth}(r_{j_2-1}) = \mathit{Depth}(q_{k_2})$. If $j_2 > k_1$ then we can reason exactly as in the previous case. If $j_2 < j_1$, then the reasoning is again similar. By construction, $d_i = d_{j_2}$. By the requirements for the run of $\mathbb{S}(\mathbb{A}, \mathbb{A}')$, $\mathit{Depth}(r_{i_1-1}) - m'_1 \geq \mathit{Depth}(r_{j_2-1})$ for any i_1, $j_2 \leq i_1 \leq k_2$, with $\mu_{i_1} = \langle m'_1, m'_2\rangle$. In particular, $\mathit{Depth}(r_{i-1}) - m_1 \geq \mathit{Depth}(r_{j_2-1})$, that is, all the registers tested on step i are above $\mathit{Depth}(r_{j_2-1})$. The only data value in these registers is d_{j_2}; indeed, if $i < j_1$, then all these registers are written between j_2 and i, and the only data value that is written on these steps is d_{j_2}; otherwise, that is, if $j_1 < k_1 < i$, d_{j_1} is also written to some registers on steps between j_1 and k_1, but all these registers are emptied by step k_1. In other words, λ_{i-1} contains d_i in all $x_{\mathit{Depth}(r_{i-1})}, \ldots, x_{\mathit{Depth}(r_{i-1})-m_1+1}$, as required.

Reasoning like this, we can consider all i except those that do not belong to any interval. In these cases, however, $m_1 = 0$, because $\mu_i = \langle 0, 0\rangle$ by construction. Hence, as already discussed, the check is trivial.

Finally, we need to show that $w \notin \mathcal{L}(\mathbb{A}')$. Assume for the sake of contradiction that it is not the case and there is an accepting run of $\mathbb{A}'$ on data word w of the form

$$[r'_0, \lambda'_0], [q'_1, \lambda'_1], [r'_1, \lambda'_1], \ldots, [r'_{n-1}, \lambda'_{n-1}], [q'_n, \lambda'_n]$$

with $[r'_0, \lambda'_0] = [r'_s, \bot]$ and $q'_n = q'_f$. Let, for each i, μ'_i be the action of the transition witnessing $[r'_{i-1}, \lambda'_{i-1}] \Rightarrow^{\mathbb{A}'}_{d_i} [q'_i, \lambda'_i]$. We prove that this implies that $\mathcal{F}_n$ contains the s-r tuple $[q'_f, \mathtt{R}_\emptyset]$, which would contradict the fact that s_n is final and the run of $\mathbb{S}(\mathbb{A}, \mathbb{A}')$ is accepting.

We construct stacks of actions $\mathtt{R}_i$, for $0 \leq i \leq n$, as follows. First, let $\mathtt{R}_0 = \mathtt{R}_\emptyset$. Then, every other $\mathtt{R}_i$ is defined on the base of $\mathtt{R}_{i-1}$ depending on μ_i as follows:

1. if $\mu_i = \langle 0, 0\rangle$ then $\mathtt{R}_i = \mathtt{R}_{i-1}$;
2. if $\mu_i = \langle 0, m_2\rangle$ for $m_2 > 0$ then $\mathtt{R}_i = (\mathtt{R}_{i-1}[1], \ldots, \mathtt{R}_{i-1}[|\mathtt{R}_{i-1}|], \mu'_i)$;
3. if $\mu_i = \langle m_1, 0\rangle$ for $m_1 > 0$ such that the reminder $\mathit{Rem}(\mathtt{S}_{i-1}, m_1) = 0$ then

$$\mathtt{R}_i = (\mathtt{R}_{i-1}[1], \ldots, \mathtt{R}_{i-1}[\mathit{Lev}(\mathtt{S}_{i-1}, m_1) - 1]);$$

4. if $\mu_i = \langle m_1, m_2 \rangle$ such that $m_1 > 0$, and m_2 or $\mathit{Rem}(\mathtt{S}_{i-1}, m_1)$ is not 0 then

$$\mathtt{R}_i = (\mathtt{R}_{i-1}[1], \ldots, \mathtt{R}_{i-1}[\mathit{Lev}(\mathtt{S}_{i-1}, m_1) - 1], \\ \mathtt{R}_{i-1}[\mathit{Lev}(\mathtt{S}_{i-1}, m_1)] \circ \cdots \circ \mathtt{R}_{i-1}[|\mathtt{R}_{i-1}|] \circ \mu'_i).$$

Next we show, by induction on i, that $[r'_i, \mathtt{R}_i] \in \mathcal{F}'_i$ and $[q'_i, \mathtt{R}_i] \in \mathcal{F}_i$. For the basis of the induction, $[r'_0, \mathtt{R}_0] \in \mathcal{F}'_0$ by construction. Let $[r'_{i-1}, \mathtt{R}_{i-1}] \in \mathcal{F}'_{i-1}$. We show that $[q'_i, \mathtt{R}_i] \in \mathcal{F}_i$. Knowing that $(r'_{i-1}, \mu'_i, q'_i) \in \delta'$, we have four cases, depending on the form of μ_i as above.

Case 1. If $\mu_i = \langle 0, 0 \rangle$, then we only need to prove that $\mu'_i = \langle 0, 0 \rangle$. Let $\mu'_i = \langle m'_1, m'_2 \rangle$. We first show that $m'_1 = 0$. On the one hand, $\mu_i = \langle 0, 0 \rangle$, so d_i is a fresh data value that is different from all $d_1, \ldots, d_{i-1}$. On the other, $\mathit{Image}(\lambda'_{i-1})$ consists only of these data values by construction. Therefore, $d_i \notin \mathit{Image}(\lambda'_{i-1})$ and no comparisons can be performed in $[r'_{i-1}, \lambda'_{i-1}] \Rightarrow^{\mathbb{A}'}_{d_i} [q'_i, \lambda'_i]$, that is, $m'_1 = 0$. Next we show that $m'_2 = 0$. Indeed, all of the values written to registers must be tested by the end of the accepting run of $\mathbb{A}'$, so d_i, which does not appear among $d_{i+1}, \ldots, d_n$, cannot be written to any register $x' \in X'$. Therefore, $m'_2 = 0$. Hence, $(r'_{i-1}, \langle 0, 0 \rangle, q'_i) \in \delta'$, and $[q'_i, \mathtt{R}_i] \in \mathcal{F}_i$ by definition.

Case 2. If $\mu_i = \langle 0, m_2 \rangle$ for $m_2 > 0$, then $[q'_i, \mathtt{R}_i] \in \mathcal{F}_i$ by definition.

Case 3. If $\mu_i = \langle m_1, 0 \rangle$ for $m_1 > 0$ and $\mathit{Rem}(\mathtt{S}_{i-1}, m_1) = 0$ then the only thing we need to check is equation (3) (for $\mathtt{R}_{i-1}$ as $\mathtt{R}_1$, $\mathtt{S}_{i-1}$ as $\mathtt{S}_1$ and μ'_i as μ'). Let first i be the smallest number with the properties, that is, $i = k_1$ in the construction above. In this case (3) boils down to $\mu'_{j_1} \circ \cdots \circ \mu'_{k_1} = \langle 0, 0 \rangle$, where j_1 is the greatest number with $\mathit{Depth}(r_{j_1 - 1}) = \mathit{Depth}(q_{k_1})$. Since the run of $\mathbb{A}'$ is accepting, we have that $\mathit{Depth}(r'_{j_1 - 1}) = \mathit{Depth}(q'_{k_1})$; the proof of this fact is very similar to the reasoning in case 1: $\mathit{Depth}(r'_{j_1 - 1}) > \mathit{Depth}(q'_{k_1})$ would mean that d_{j_1} is successfully tested for equality with some value written to a register before j_1, which, however, is not possible by construction; also $\mathit{Depth}(r'_{j_1 - 1}) < \mathit{Depth}(q'_{k_1})$ would mean that the data value d_{j_1} in the register $x'_{\mathit{Depth}(r'_{j_1 - 1}) + 1}$, which does not appear in the word after k_1, is never tested after k_1, so q'_n could not be final. Moreover, by the same reason $\mathit{Depth}(r'_{h-1}) - m'_1 \geq \mathit{Depth}(r'_{j_1 - 1})$ for any h, $j_1 \leq h \leq k_1$, and $\mu'_h = \langle m'_1, m'_2 \rangle$. Hence, by Lemma 4.11 $\mu'_{j_1} \circ \cdots \circ \mu'_{k_1} = \langle 0, 0 \rangle$ as required. Let now $i = k_2$ in the construction above. If $j_2 > k_1$, then we can check (3) in exactly the same way. If $j_2 < j_1$, that is, the interval $[j_1, k_1]$ is contained in the interval $[j_2, k_2]$, then (3) boils down to $\mu'_{j_2} \circ \cdots \circ \mu'_{j_1 - 1} \circ \mu'_{k_1 + 1} \circ \cdots \circ \mu'_{k_2} = \langle 0, 0 \rangle$. It holds by applying Corollary 4.10 to the same construction. Reasoning like this, we can check (3) for all relevant i.

Case 4. If $\mu_i = \langle m_1, m_2 \rangle$ such that $m_1 > 0$, and m_2 or ***Rem***$(\mathtt{S}_{i-1}, m_1)$ is not equal to 0, then $[q'_i, \mathtt{R}_i] \in \mathcal{F}_i$ again by definition.

Let now $[q'_i, \mathtt{R}_i] \in \mathcal{F}_i$. Then $[r'_i, \mathtt{R}_i] \in \mathcal{F}'_i$ holds because $(q'_i, a_i, r'_i) \in \alpha'$ by the properties of the accepting run of $\mathbb{A}'$.

Hence, we have that $[q'_n, \mathtt{R}_n] \in \mathcal{F}_n$. We know that $q'_n = q'_f$. Also, $\mathtt{R}_n = \mathtt{R}_\emptyset$, because $|\mathtt{R}_n| = |\mathtt{S}_n| = |\mathtt{S}_\emptyset| = 0$. So, $\mathcal{F}_n$ contains the s-r tuple $[q'_f, \mathtt{R}_\emptyset]$, which contradicts to the fact that the run of $\mathbb{S}(\mathbb{A}, \mathbb{A}')$ is accepting. It means that the assumption is wrong and $w \notin \mathcal{L}(\mathbb{A}')$, as required.

□

Having the forward direction of the correctness proved, we move to the backward direction.

Claim 4.14. *For each data word accepted by $\mathbb{A}$ and rejected by $\mathbb{A}'$ there is a action word accepted by $\mathbb{S}(\mathbb{A}, \mathbb{A}')$.*

Proof. Assume that there is a data word $w = d_1 a_1 d_2 a_2 \cdots a_{n-1} d_n$ such that $w \in \mathcal{L}(\mathbb{A})$ but $w \notin \mathcal{L}(\mathbb{A}')$. Since $w \in \mathcal{L}(\mathbb{A})$, there is an accepting run of $\mathbb{A}$ on w of the form

$$[r_0, \lambda_0], [q_1, \lambda_1], [r_1, \lambda_1], \ldots, [q_{n-1}, \lambda_{n-1}], [r_{n-1}, \lambda_{n-1}], [q_n, \lambda_n]$$

with $[r_0, \lambda_0] = [r_s, \bot]$ and $q_n = q_f$.

On the base of the data word and the run, we next construct an action word $\mu_1 a_1 \mu_2 a_2 \cdots a_{n-1} \mu_n$ and a sequence $s'_0, s_1, s'_1, s_2, s'_2, \ldots, s'_{n-1}, s_n$ of IDs of system $\mathbb{S}(\mathbb{A}, \mathbb{A}')$ with all $s'_i = [[r_i, \mathtt{S}_i], \mathcal{F}'_i]$ and all $s_i = [[q_i, \mathtt{S}_i], \mathcal{F}_i]$; then we prove that the sequence is an accepting run of $\mathbb{S}(\mathbb{A}, \mathbb{A}')$ on the action word. Note that the lengths of the action word and the sequence are the same as the lengths of data word w and the accepting run of $\mathbb{A}$, respectively; moreover, all the labels a_i, data states r_i and word states q_i are also the same.

We define the actions in the action word as the actions passed by $\mathbb{A}$ when accepting the data word: let each μ_i be the action of the transition witnessing $[r_{i-1}, \lambda_{i-1}] \Rightarrow^{\mathbb{A}}_{d_i} [q_i, \lambda_i]$ in the accepting run of $\mathbb{A}$ on w. In turn, stacks $\mathtt{S}_i$ as well as sets of s-r tuples $\mathcal{F}_i$ and $\mathcal{F}'_i$ are defined as follows: let $\mathtt{S}_0 = \mathtt{S}_\emptyset$, $\mathcal{F}'_0 = \{[r'_s, \mathtt{R}_\emptyset]\}$, and every other $\mathtt{S}_i$, $\mathcal{F}_i$ and $\mathcal{F}'_i$ be constructed as in the definition of transition relation $\gamma_{\mathbb{S}}$ by taking μ_i as μ, r_{i-1} as r, $\mathtt{S}_{i-1}$ as $\mathtt{S}_1$, $\mathcal{F}'_{i-1}$ as $\mathcal{F}_1$, q_i as q, $\mathtt{S}_i$ as $\mathtt{S}_2$, and $\mathcal{F}_i$ as $\mathcal{F}_2$ in the first case, and by taking a_i as a, q_i as q, $\mathtt{S}_i$ as $\mathtt{S}_1$, $\mathcal{F}_i$ as $\mathcal{F}_1$, r_i as r, $\mathtt{S}_i$ as $\mathtt{S}_2$, and $\mathcal{F}'_i$ as $\mathcal{F}_2$ in the second case.

It is straightforward to check that $s'_0, s_1, s'_1, s_2, s'_2, \ldots, s'_{n-1}, s_n$ is a run of $\mathbb{S}(\mathbb{A}, \mathbb{A}')$ on $\mu_1 a_1 \mu_2 a_2 \cdots a_{n-1} \mu_n$. Indeed, by definition, $s'_0 = [[r_0, \mathtt{S}_0], \mathcal{F}'_0]$ is initial, while $([[r_{i-1}, \mathtt{S}_{i-1}], \mathcal{F}'_{i-1}], \mu_i, [[q_i, \mathtt{S}_i], \mathcal{F}_i])$ and $([[q_i, \mathtt{S}_i], \mathcal{F}_i], a_i, [[r_i, \mathtt{S}_i], \mathcal{F}'_i])$ belong to $\gamma_{\mathbb{S}}$ for each i.

We are left to show that the run is accepting, that is, s_n is final. Recall first that $q_n = q_f$; moreover, the depth $\mathit{Depth}(q_n)$ is 0, so $\mathtt{S}_n = \mathtt{S}_\emptyset$. Hence, we only need to show that $\mathcal{F}_n$ does not contain the s-r tuple $[q'_f, \mathtt{R}_\emptyset]$. Assume for the sake of contradiction that it is not true and $\mathcal{F}_n$ contains this s-r tuple. We show that in this case there exists an accepting run of $\mathbb{A}'$ on w, which contradicts the fact that $w \notin \mathcal{L}(\mathbb{A}')$.

Since $\mathcal{F}_n$ contains $[q'_f, \mathtt{R}_\emptyset]$, there exist s-r tuples $[q'_i, \mathtt{R}_i] \in \mathcal{F}_i$, $0 \leq i \leq n$, and $[r'_i, \mathtt{R}_i] \in \mathcal{F}'_i$, $1 \leq i \leq n$, as well as actions μ'_i, such that $r'_0 = r'_s$, $\mathtt{R}_0 = \mathtt{R}_\emptyset$, $(r'_{i-1}, \mu'_i, q'_i) \in \delta'$, μ'_i satisfies the conditions in the definition of $\gamma_{\mathbb{S}}$, and $(q'_i, a_i, r'_i) \in \alpha'$ for all i, and $[q'_n, \mathtt{R}_n] = [q'_f, \mathtt{R}_\emptyset]$.

We define assignments λ'_i of X', for $0 \leq i \leq n$. The initial assignment λ'_0 is $\perp$, and each λ'_i, for $1 \leq i \leq n$, is defined as follows:

- it is d_i on $x'_{\mathit{Depth}(r'_{i-1})-m'_1+1}, \ldots, x'_{\mathit{Depth}(r'_{i-1})-m'_1+m'_2}$, for $\mu'_i = \langle m'_1, m'_2 \rangle$;
- it coincides with λ'_{i-1} on all other x'.

Next we prove that the sequence of configurations

$$[r'_0, \lambda'_0], [q'_1, \lambda'_1], [r'_1, \lambda'_1], \ldots, [r'_{n-1}, \lambda'_{n-1}], [q'_n, \lambda'_n]$$

is an accepting run of $\mathbb{A}'$ on data word w. Since $(q'_i, a_i, r'_i) \in \alpha'$, we know that $[q'_i, \lambda'_i] \Rightarrow^{\mathbb{A}'}_{a_i} [r'_i, \lambda'_i]$ for all i. We also know that $\lambda'_0 = \perp$, $r'_0 = r'_s$, and $q'_n = q'_f$. Hence, it is left to be proven that $[r'_{i-1}, \lambda'_{i-1}] \Rightarrow^{\mathbb{A}'}_{d_i} [q'_i, \lambda'_i]$ for every i, and we prove it by showing that the transition (r'_{i-1}, μ'_i, q'_i) in δ' is applicable in this case as well. In fact, λ'_i differs from λ'_{i-1} only on registers $x'_{\mathit{Depth}(r'_{i-1})-m'_1+1}, \ldots, x'_{\mathit{Depth}(r'_{i-1})-m'_1+m'_2}$ and contains d_i in all these registers by construction, so we only need to show that λ'_{i-1} contains d_i in all $x'_{\mathit{Depth}(r'_{i-1})}, \ldots, x'_{\mathit{Depth}(r'_{i-1})-m'_1+1}$. If $m'_1 = 0$ then the check is trivial, so in what follows we assume that $m'_1 > 0$.

Consider first the case when $j_1 \leq i \leq k_1$, where k_1 is the smallest number such that $\mathit{Rem}(\mathtt{S}_{k_1-1}, m_1) = 0$ for $\mu_{k_1} = \langle m_1, 0 \rangle$, $m_1 > 0$, and j_1 is the greatest number such that $j_1 < k_1$ and $\mathit{Depth}(r_{j_1-1}) = \mathit{Depth}(q_{k_1})$. By the requirements for the run of $\mathbb{S}(\mathbb{A}, \mathbb{A}')$, $\mu'_{j_1} \circ \cdots \circ \mu'_{k_1} = \langle 0, 0 \rangle$, so, by Lemma 4.11, $\mathit{Depth}(r'_{j_1-1}) = \mathit{Depth}(q'_{k_1})$ and $\mathit{Depth}(r'_{i_1-1}) - m''_1 \geq \mathit{Depth}(r'_{j_1-1})$ for any i_1, $j_1 \leq i_1 \leq k_1$ and $\mu_{i_1} = \langle m''_1, m''_2 \rangle$. In particular, $\mathit{Depth}(r'_{i-1}) - m'_1 \geq \mathit{Depth}(r'_{j_1-1})$, that is, all the registers tested on step i are above $\mathit{Depth}(r'_{j_1-1})$.

On the one hand, all the values in these registers are written after step j_1; on the other, the only data value that is written between j_1 and i is d_{j_1}. Since $m'_1 > 0$, that is, $\mu'_i \neq \langle 0, 0\rangle$, we have that $\mu_i \neq \langle 0, 0\rangle$ by construction. So, λ'_{i-1} contains d_i in $x'_{\mathit{Depth}(r'_{i-1})}, \ldots, x'_{\mathit{Depth}(r'_{i-1})-m'_1+1}$ as required.

Consider now the case when $j_1 \leq i \leq k_1$ does not hold, but $j_2 \leq i \leq k_2$, where k_2 is the second smallest number such that $\mathit{Rem}(\mathtt{S}_{k_2-1}, m_1) = 0$ for $\mu_{k_2} = \langle m_1, 0\rangle$, $m_1 > 0$, and j_2 is the greatest number such that $j_2 < k_2$ and $\mathit{Depth}(r_{j_2-1}) = \mathit{Depth}(q_{k_2})$. If $j_2 > k_1$ then we can reason exactly as in the previous case. If $j_2 < j_1$, then $\mu'_{j_2} \circ \cdots \circ \mu'_{j_1-1} \circ \mu'_{k_1+1} \circ \cdots \circ \mu'_{k_2} = \langle 0, 0\rangle$ by the requirements on the run. By Corollary 4.10, $\mu'_{j_2} \circ \cdots \circ \mu'_{i_2} = \langle 0, 0\rangle$, so, by Lemma 4.11, $\mathit{Depth}(r'_{j_2-1}) = \mathit{Depth}(q'_{k_2})$ and $\mathit{Depth}(r'_{i_1-1}) - m''_1 \geq \mathit{Depth}(r'_{j_2-1})$ for any i_1, $j_2 \leq i_1 \leq k_2$, and $\mu_{i_1} = \langle m''_1, m''_2\rangle$. In particular, $\mathit{Depth}(r'_{i-1}) - m'_1 \geq \mathit{Depth}(r'_{j_2-1})$. If $i < j_1$, then, as in the previous case, the only value in the tested registers is d_{j_2}, and it is the same value as d_i, as required. If $i > k_1$, then the same holds, because the value d_{j_1}, which was pushed into some registers above $\mathit{Depth}(r'_{j_2-1})$ between j_1 and k_1, is already completely popped off before $k_1 + 1$.

Reasoning like this, we can consider all i except those that do not belong to any interval. In these cases, however, $m'_1 = 0$, because $\mu'_i = \langle 0, 0\rangle$ by construction. Hence, as already discussed, the check is trivial.

We conclude that the sequence of configurations $[r'_0, \lambda'_0], [q'_1, \lambda'_1], [r'_1, \lambda'_1]$, $\ldots, [r'_{n-1}, \lambda'_{n-1}], [q'_n, \lambda'_n]$ is an accepting run of $\mathbb{A}'$ on w, which contradicts, however, the fact that $w \notin \mathcal{L}(\mathbb{A}')$. So, our assumption was wrong and the run of $\mathbb{S}(\mathbb{A}, \mathbb{A}')$ is accepting, as required. □

Claims 4.13 and 4.14 guarantee that $\mathcal{L}(\mathbb{A}) \subseteq \mathcal{L}(\mathbb{A}')$ if and only if system $\mathbb{S}(\mathbb{A}, \mathbb{A}')$ accepts an action word. However, reducing to the emptiness problem of $\mathbb{S}(\mathbb{A}, \mathbb{A}')$ gives us any essential gain in complexity in comparison with much simpler algorithm in the proof of Theorem 3.11. It is not surprising that there are reachable IDs of $\mathbb{S}(\mathbb{A}, \mathbb{A}')$ with s-r tuples in their sets $\mathcal{F}$ having the same state, but different stacks of actions. Moreover, there are examples with $\mathcal{F}$ containing exponential number of s-r tuples, so checking the emptiness of $\mathbb{S}(\mathbb{A}, \mathbb{A}')$ in the standard way can require exponential space as well. Nevertheless, it is possible to reduce the space usage from exponential to polynomial, and we will devote the second part of the proof to showing how it can be done.

Step 2. The key observation to stay in polynomial space is that even if the number of s-r tuples in the set $\mathcal{F}$ can be exponential, this set is *never arbitrary*, and a lot of the information in the stacks of actions is shared: if a stack can be seen as a unary tree, then every set of such trees that appears on a run can be represented

as a *directed acyclic graph (dag)*, whose size is polynomial. For example, if the expressions have the form $c \cdot (a \cdot (a \cdot (a \cdot b \dots)_{=} \dots)_{=} \dots)_{=} \dots$ and $c \cdot (a^* \cdot (a^* \cdot b \dots)_{=} \dots)_{=} \dots$, then before reading b there are two s-r tuples with stacks of actions $(\langle 0,1\rangle, \langle 0,0\rangle, \langle 0,1\rangle)$ and $(\langle 0,1\rangle, \langle 0,1\rangle, \langle 0,0\rangle)$. These stacks have the common first action, $\langle 0,1\rangle$, so we can keep it in memory just once, together with links to the rests of the stacks. In the rest of this proof we formalise this intuition and show its correctness.

Let $\mathcal{N}_0$ be the set of all natural numbers with 0 and $\star$ be a special symbol. A *reaction dag* $\mathbb{D}$ for pdt-automaton $\mathbb{A}'$ is a rooted labelled dag with nodes from $\mathcal{N}_0 \times (Q' \cup \{\star\})$ such that

- the root v_0 is $(0, \star)$, and this is the only node with $\star$;
- the first component g of each node (g, q'), called *level*, is such that the length of each directed path (i.e., the number of edges along this path) from the root to (g, q') is g;
- all the leaves have the same level, denoted by $\|\mathbb{D}\|$;
- every edge is labelled with an action in $\mathcal{M}$;
- every leaf v is labelled with a possibly empty set of states $B(v) \subseteq P'$ of $\mathbb{A}'$.

In fact, we will concentrate on reaction dags that have either only data states in the sets $B(v)$ of its leaves v or only word states in these sets.

A reaction dag $\mathbb{D}$ *represents* an s-r tuple $[p', \mathtt{R}]$ if it has a path from the root to a leaf v such that $p' \in B(v)$ and the actions along this path, from the root to the leaf, are the actions in the stack $\mathtt{R}$, from the bottom to the head.

Essentially, we are going to show that the sets of s-r tuples in all reachable IDs of $\mathbb{S}(\mathbb{A}, \mathbb{A}')$ are representable by reaction dags. This is enough for a PSPACE algorithm for containment, because, contrary to sets of s-r tuples, every reaction dag is of polynomial size by definition (the size of a stack in an s-r tuple is bounded by the maximal depth of a state in $\mathbb{A}'$). To this end, we define another transition system, $\mathbb{U}(\mathbb{A}, \mathbb{A}')$, that runs on action words and differs from $\mathbb{S}(\mathbb{A}, \mathbb{A}')$ by reaction dags in IDs instead of sets of s-r tuples.

The IDs U of $\mathbb{U}(\mathbb{A}, \mathbb{A}')$ are pairs $[[p, \mathtt{S}], \mathbb{D}]$ where

- $[p, \mathtt{S}]$ is an s-s tuple with $p \in P$ (i.e., a state of $\mathbb{A}$) and stack $\mathtt{S}$ of positive numbers such that $\Sigma_{1 \le i \le |\mathtt{S}|} \mathtt{S}[i] = \mathit{Depth}(p)$; and
- $\mathbb{D}$ is a reaction dag such that $\|\mathbb{D}\| = |\mathtt{S}|$.

The initial ID of $\mathbb{U}(\mathbb{A}, \mathbb{A}')$ is $[[r_s, \mathtt{S}_\emptyset], \mathbb{D}_\emptyset]$, where $\mathbb{D}_\emptyset$ is the reaction dag consisting of only the root v_0 and such that $B(v_0) = \{r'_s\}$. The final IDs are $[[q_f, \mathtt{S}_\emptyset], \mathbb{D}_f]$,

where $\mathbb{D}_f$ is any reaction dag such that $q'_f \notin B(v_0)$ (note that the $\|\mathbb{D}_f\| = |\mathtt{S}_\emptyset| = 0$, so the root is the only leaf of $\mathbb{D}_f$).

The transition relation $\gamma_{\mathbb{U}} \subseteq U \times (\mathcal{M} \cup \Sigma) \times U$ of system $\mathbb{U}(\mathbb{A}, \mathbb{A}')$ is defined as follows:

1. $([[r, \mathtt{S}_1], \mathbb{D}_1], \mu, [[q, \mathtt{S}_2], \mathbb{D}_2]) \in \gamma_{\mathbb{U}}$ for a data state r, word state q, stacks $\mathtt{S}_1$ and $\mathtt{S}_2$, reaction dags $\mathbb{D}_1$ and $\mathbb{D}_2$, and action $\mu \in \mathcal{M}$ if $(r, \mu, q) \in \delta$ and one of the following holds:
   (a) $\mu = \langle 0, 0 \rangle$, $\mathtt{S}_1 = \mathtt{S}_2$, and $\mathbb{D}_2$ is the same as $\mathbb{D}_1$ except that $q' \in B(v)$ for a leaf v in $\mathbb{D}_2$ if and only if there is $r' \in B(v)$ in $\mathbb{D}_1$ with $(r', \langle 0, 0 \rangle, q') \in \delta'$;
   (b) $\mu = \langle 0, m_2 \rangle$ for $m_2 > 0$, $\mathtt{S}_2 = (\mathtt{S}_1[1], \ldots, \mathtt{S}_1[|\mathtt{S}_1|], m_2)$, and $\mathbb{D}_2$ can be obtained from $\mathbb{D}_1$ by
      – adding all the nodes $v_{new} = (\|\mathbb{D}_1\| + 1, q'_{new})$ labelled as $B(v_{new}) = \{q'_{new}\}$ and edges (v, v_{new}) labelled by μ', such that $v = (\|\mathbb{D}_1\|, q')$, $(r', \mu', q'_{new}) \in \delta'$, and $r' \in B(q')$; and
      – removing all the labels of nodes of level $\|\mathbb{D}_1\|$ and all the nodes that are not on a path from the root to a node of level $\|\mathbb{D}_1\| + 1$;
   (c) $\mu = \langle m_1, 0 \rangle$ for $m_1 > 0$ such that the reminder $\mathit{Rem}(\mathtt{S}_1, m_1) = 0$, $\mathtt{S}_2 = (\mathtt{S}_1[1], \ldots, \mathtt{S}_1[\mathit{Lev}(\mathtt{S}_1, m_1) - 1])$, and $\mathbb{D}_2$ can be obtained from $\mathbb{D}_1$ by
      – removing all the nodes of levels greater than $\mathit{Lev}(\mathtt{S}_1, m_1) - 1$ (together with all the edges involving them); and
      – adding a word state q'_{new} to the label $B(v)$ of a new leaf $v = (\mathit{Lev}(\mathtt{S}_1, m_1) - 1, q')$ if and only if there is a path from v to a leaf v_1 in $\mathbb{D}_1$ with $r' \in B(v_1)$ such that $(r', \mu', q'_{new}) \in \delta'$ and $\mu'' \circ \mu' = \langle 0, 0 \rangle$, where μ'' is the composition of edge labels along the path;
   (d) $\mu = \langle m_1, m_2 \rangle$ for $m_1 > 0$ such that at least one of m_2 and $\mathit{Rem}(\mathtt{S}_1, m_1)$ is not 0, $\mathtt{S}_2 = (\mathtt{S}_1[1], \ldots, \mathtt{S}_1[\mathit{Lev}(\mathtt{S}_1, m_1) - 1], \mathit{Rem}(\mathtt{S}_1, m_1) + m_2)$, and $\mathbb{D}_2$ can be obtained from $\mathbb{D}_1$ by
      – removing all the nodes of levels greater than $\mathit{Lev}(\mathtt{S}_1, m_1) - 1$ (together with all the edges involving them);
      – adding all the nodes $v_{new} = (\mathit{Lev}(\mathtt{S}_1, m_1), q'_{new})$ labelled as $B(v_{new}) = \{q'_{new}\}$ and edges (v, v_{new}) labelled by $\mu'' \circ \mu'$, such that $v = (\mathit{Lev}(\mathtt{S}_1, m_1) - 1, q')$, there is a path from v to a leaf v_1 in $\mathbb{D}_1$ with $r' \in B(v_1)$ such that $(r', \mu', q'_{new}) \in \delta'$ and μ'' is the composition of edge labels along the path; and

– removing all the nodes that are not on a path from the root to a node of level $\mathit{Lev}(\mathtt{S}_1, m_1)$;

2. $([[q, \mathtt{S}_1], \mathbb{D}_1], a, [[r, \mathtt{S}_2], \mathbb{D}_2]) \in \gamma_{\mathbb{U}}$ for a word state q, data state r, stacks $\mathtt{S}_1$ and $\mathtt{S}_2$, reaction dags $\mathbb{D}_1$ and $\mathbb{D}_2$, and label a, if $(q, a, r) \in \alpha$, $\mathtt{S}_1 = \mathtt{S}_2$, and $\mathbb{D}_2$ is the same as $\mathbb{D}_1$ except that $r' \in B(v)$ for a leaf v in $\mathbb{D}_2$ if and only if $q' \in B(v)$ in $\mathbb{D}_1$ for $(q', a, r') \in \alpha'$.

Note that some new nodes v_{new} added to a reaction dag in cases 1(b) and 1(d) may have several new incoming edges.

A *run* of transition system $\mathbb{U}(\mathbb{A}, \mathbb{A}')$ on an action word $w = \mu_1 a_1 \mu_2 a_2 \cdots a_{n-1} \mu_n$ is a sequence $u'_0, u_1, u'_1, u_2, u'_2, \ldots, u'_{n-1}, u_n$ of IDs such that u'_0 is initial, $(u'_{i-1}, \mu_i, u_i) \in \gamma_{\mathbb{U}}$ for each i, $1 \leq i \leq n$, and $(u_i, a_i, u'_i) \in \gamma_{\mathbb{U}}$ for each i, $1 \leq i < n$. It is *accepting* if u_n is final, and in this case $\mathbb{U}(\mathbb{A}, \mathbb{A}')$ *accepts* w.

Claim 4.15. *An action word is accepted by system* $\mathbb{U}(\mathbb{A}, \mathbb{A}')$ *if and only if it is accepted by system* $\mathbb{S}(\mathbb{A}, \mathbb{A}')$.

Proof. We start with the forward direction. Let $w = \mu_1 a_1 \mu_2 a_2 \cdots a_{n-1} \mu_n$ be an action word and $v'_0, v_1, v'_1, v_2, v'_2, \ldots, v'_{n-1}, v_n$ be an accepting run of $\mathbb{U}(\mathbb{A}, \mathbb{A}')$ on w with all $v'_i = [[r_i, \mathtt{S}_i], \mathbb{D}'_i]$ and all $v_i = [[q_i, \mathtt{S}_i], \mathbb{D}_i]$. We will construct an accepting run $s'_0, s_1, s'_1, s_2, s'_2, \ldots, s'_{n-1}, s_n$ of $\mathbb{S}(\mathbb{A}, \mathbb{A}')$ on w, with all $s'_i = [[r_i, \mathtt{S}_i], \mathcal{F}'_i]$ and all $s_i = [[q_i, \mathtt{S}_i], \mathcal{F}_i]$. Note that r_i, $\mathtt{S}_i$, and q_i are the same in both of the runs, so we only need to define $\mathcal{F}'_i$ and $\mathcal{F}_i$.

Let $\mathcal{F}'_0 = \mathcal{F}_\emptyset$, and all the other sets of s-r tuples are iteratively defined exactly as in the definition of $\gamma_{\mathbb{S}}$, by taking, for each i, r_{i-1} as r, $\mathtt{S}_{i-1}$ as $\mathtt{S}_1$, $\mathcal{F}'_{i-1}$ as $\mathcal{F}_1$, μ_i as μ, q_i as q, $\mathtt{S}_i$ as $\mathtt{S}_2$ and $\mathcal{F}_i$ as $\mathcal{F}_2$ for a transition $(r_{i-1}, \mu_i, q_i) \in \delta$, and taking q_i as q, $\mathtt{S}_i$ as $\mathtt{S}_1$, $\mathcal{F}_i$ as $\mathcal{F}_1$, a_i as a, r_i as r, $\mathtt{S}_i$ as $\mathtt{S}_2$ and $\mathcal{F}'_i$ as $\mathcal{F}_2$ for a transition $(q_i, a_i, r_i) \in \alpha$. The sequence $s'_0, s_1, s'_1, s_2, s'_2, \ldots, s'_{n-1}, s_n$ is a run of $\mathbb{S}(\mathbb{A}, \mathbb{A}')$ on w by construction. We are left to show that it is an accepting run. Assume, for the sake of contradiction, that it is not the case, that is, $[q'_f, \mathtt{R}_\emptyset] \in \mathcal{F}_n$. We prove that then $\mathbb{D}_n$ represents $[q'_f, \mathtt{R}_\emptyset]$, that is, $q'_f \in B(v_0)$ in $\mathbb{D}_n$, which contradicts to the fact that the run of $\mathbb{U}(\mathbb{A}, \mathbb{A}')$ is accepting.

Since $\mathcal{F}_n$ contains $[q'_f, \mathtt{R}_\emptyset]$, there exist s-r tuples $[q'_i, \mathtt{R}_i] \in \mathcal{F}_i$ and $[r'_i, \mathtt{R}_i] \in \mathcal{F}'_i$, as well as actions μ'_i, such that $r'_0 = r'_s$, $\mathtt{R}_0 = \mathtt{R}_\emptyset$, $(r'_{i-1}, \mu'_i, q'_i) \in \delta'$, μ'_i satisfies the conditions in the definition of $\gamma_{\mathbb{S}}$, and $(q'_i, a_i, r'_i) \in \alpha'$ for all i. By construction, $\mathbb{D}'_0 = \mathbb{D}_\emptyset$ represents $[r'_0, \mathtt{R}_0]$, and it is straightforward to compare the definitions of $\gamma_{\mathbb{S}}$ and $\gamma_{\mathbb{U}}$, and see that $\mathbb{D}_i$ represents $[q'_i, \mathtt{R}_i]$ and $\mathbb{D}'_i$ represents $[r'_i, \mathtt{R}_i]$. In particular, $\mathbb{D}_n$ represents $[q'_f, \mathtt{R}_\emptyset]$. So, our assumption was wrong and the constructed run of $\mathbb{S}(\mathbb{A}, \mathbb{A}')$ is accepting.

Next we prove the statement in the backward direction. Let $w = \mu_1 a_1 \mu_2 a_2 \cdots a_{n-1} \mu_n$ be a action word and $s'_0, s_1, s'_1, s_2, s'_2, \ldots, s'_{n-1}, s_n$ be an accepting run of $\mathbb{S}(\mathbb{A}, \mathbb{A}')$ on w with all $s'_i = [[r_i, \mathtt{S}_i], \mathcal{F}'_i]$ and all $s_i = [[q_i, \mathtt{S}_i], \mathcal{F}_i]$. We will construct an accepting run $u'_0, u_1, u'_1, u_2, u'_2, \ldots, u'_{n-1}, u_n$ of $\mathbb{U}(\mathbb{A}, \mathbb{A}')$ on w, with all $u'_i = [[r_i, \mathtt{S}_i], \mathbb{D}'_i]$ and all $u_i = [[q_i, \mathtt{S}_i], \mathbb{D}_i]$. Note that r_i, $\mathtt{S}_i$, and q_i are the same in both of the runs, so we only need to define $\mathbb{D}'_i$ and $\mathbb{D}_i$.

Let $\mathbb{D}'_0 = \mathbb{D}_\emptyset$, and all the following reaction dags are iteratively defined exactly as in the definition of $\gamma_{\mathbb{U}}$, by taking, for each i, r_{i-1} as r, $\mathtt{S}_{i-1}$ as $\mathtt{S}_1$, $\mathbb{D}'_{i-1}$ as $\mathbb{D}_1$, μ_i as μ, q_i as q, $\mathtt{S}_i$ as $\mathtt{S}_2$ and $\mathbb{D}_i$ as $\mathbb{D}_2$ for a transition $(r_{i-1}, \mu_i, q_i) \in \delta$, and taking q_i as q, $\mathtt{S}_i$ as $\mathtt{S}_1$, $\mathbb{D}_i$ as $\mathbb{D}_1$, a_i as a, r_i as r, $\mathtt{S}_i$ as $\mathtt{S}_2$ and $\mathbb{D}'_i$ as $\mathbb{D}_2$ for a transition $(q_i, a_i, r_i) \in \alpha$. Then, the sequence $u'_0, u_1, u'_1, u_2, u'_2, \ldots, u'_{n-1}, u_n$ is a run of $\mathbb{U}(\mathbb{A}, \mathbb{A}')$ on w by construction. We are left to show that it is an accepting run. Assume, for the sake of contradiction, that it is not the case, that is, $[q'_f, \mathtt{R}_\emptyset]$ is represented by $\mathbb{D}_n$. We prove, by induction on i, that, first, if an s-r tuple $[q'_i, \mathtt{R}_i]$ is represented by $\mathbb{D}_i$ then it is in $\mathcal{F}_i$, and, second, if $[r'_i, \mathtt{R}_i]$ is represented by $\mathbb{D}'_i$ then it is in $\mathcal{F}'_i$. It will imply that $[q'_f, \mathtt{R}_\emptyset] \in \mathcal{F}_n$, which contradicts to the fact that the run of $\mathbb{S}(\mathbb{A}, \mathbb{A}')$ is accepting.

For the basis of the induction, note that $[r'_s, \mathtt{R}_\emptyset]$ is the only s-r tuple represented by $\mathbb{D}'_0$, and it is in $\mathcal{F}_0$.

For the first part of the inductive step, let, for $1 \leq i \leq n$, all s-r tuples represented by $\mathbb{D}'_{i-1}$ be in $\mathcal{F}'_{i-1}$. We need to show that any $[q'_i, \mathtt{R}_i]$ represented by $\mathbb{D}_i$ is in $\mathcal{F}_i$. There are four cases, depending on the form of μ_i.

1. If $\mu_i = \langle 0, 0 \rangle$, then, by construction, there is $[r'_{i-1}, \mathtt{R}_{i-1}]$ represented by $\mathbb{D}'_{i-1}$ such that $(r'_{i-1}, \langle 0, 0 \rangle, q'_i) \in \delta'$ and $\mathtt{R}_{i-1} = \mathtt{R}_i$. Hence, $[q'_i, \mathtt{R}_i] \in \mathcal{F}_i$ by definition and the inductive hypothesis.

2. If $\mu_i = \langle 0, m_2 \rangle$ for $m_2 > 0$, then there is $[r'_{i-1}, \mathtt{R}_{i-1}]$ represented by $\mathbb{D}'_{i-1}$ such that $(r'_{i-1}, \mu', q'_i) \in \delta'$ and $\mathtt{R}_i = (\mathtt{R}_{i-1}[1], \ldots, \mathtt{R}_{i-1}[|\mathtt{R}_{i-1}|], \mu')$ for some μ'. Again, $[q'_i, \mathtt{R}_i] \in \mathcal{F}_i$ by definition and the inductive hypothesis.

3. If $\mu_i = \langle m_1, 0 \rangle$ for $m_1 > 0$ and $\mathit{Rem}(\mathtt{S}_{i-1}, m_1) = 0$ then there is $[r'_{i-1}, \mathtt{R}_{i-1}]$ represented by $\mathbb{D}'_{i-1}$ such that $(r'_{i-1}, \mu', q'_i) \in \delta'$, $\mathtt{R}_i = (\mathtt{R}_{i-1}[1], \ldots, \mathtt{R}_{i-1}[\mathit{Lev}(\mathtt{S}_{i-1}, m_1) - 1])$ and $\mathtt{R}_{i-1}[\mathit{Lev}(\mathtt{S}_{i-1}, m_1)] \circ \cdots \circ \mathtt{R}_{i-1}[|\mathtt{R}_{i-1}|] \circ \mu' = \langle 0, 0 \rangle$ for some μ'. Again, $[q'_i, \mathtt{R}_i] \in \mathcal{F}_i$ by definition and the hypothesis.

4. If $\mu_i = \langle m_1, m_2 \rangle$ such that $m_1 > 0$, and m_2 or $\mathit{Rem}(\mathtt{S}_{i-1}, m_1)$ is not equal to 0, then there is $[r'_{i-1}, \mathtt{R}_{i-1}]$ represented by $\mathbb{D}'_{i-1}$ such that $(r'_{i-1}, \mu', q'_i) \in \delta'$, $\mathtt{R}_i = (\mathtt{R}_{i-1}[1], \ldots, \mathtt{R}_{i-1}[\mathit{Lev}(\mathtt{S}_{i-1}, m_1) - 1], \mathtt{R}_{i-1}[\mathit{Lev}(\mathtt{S}_{i-1}, m_1)] \circ \cdots \circ \mathtt{R}_{i-1}[|\mathtt{R}_{i-1}|] \circ$

Data comparisions	RQD	RQM
none	PSPACE-complete*	
positive	PSPACE-complete	EXPSPACE-complete
full	undecidable	undecidable

Table 3: Summary of results. The result, known before, are marked with an asterisk. Some classes have synonyms, not given for clarity: i.e. RQDs and RQMs with no data comparisons are RPQs.

μ') for some μ'. Again, $[q'_i, \mathtt{R}_i] \in \mathcal{F}_i$ by definition and the hypothesis.

For the second part of the inductive step, let, for $1 \leq i \leq n$, all s-r tuples represented by $\mathbb{D}_i$ are in $\mathcal{F}_i$. Therefore, any $[r'_i, \mathtt{R}_i]$ represented by $\mathbb{D}_i$ is in $\mathcal{F}'_i$ by construction and the inductive hypothesis.

Hence, our assumption was wrong, and the run of $\mathbb{U}(\mathbb{A}, \mathbb{A}')$ on w is accepting, as required. □

Claims 4.13, 4.14, and 4.15 guarantee that $\mathcal{L}(\mathbb{A}) \subseteq \mathcal{L}(\mathbb{A}')$ if and only if system $\mathbb{U}(\mathbb{A}, \mathbb{A}')$ accepts an action word. To conclude the proof of the lemma, we need to prove that emptyness of the language of $\mathbb{U}(\mathbb{A}, \mathbb{A}')$ can be decided in PSPACE. By definition, the number of IDs in $\mathbb{U}(\mathbb{A}, \mathbb{A}')$ is infinite, because the components of actions are generally unbounded. However, by Lemma 4.11, all the actions $\langle m'_1, m'_2 \rangle$ that may appear in stacks of actions $\mathtt{R}$ in reachable IDs of $\mathbb{S}(\mathbb{A}, \mathbb{A}')$, and, hence as edge labels of reaction dags $\mathbb{D}$ in reachable IDs of $\mathbb{U}(\mathbb{A}, \mathbb{A}')$, are such that $m'_1 \leq \ell'$ and $m'_2 \leq \ell'$, where ℓ' is the number of registers of $\mathbb{A}'$ (i.e., the maximal depth of a state of $\mathbb{A}'$). Hence, we can restrict ourselves only to such actions in IDs. Then, by definition, such IDs can be represented in polynomial space, because each reaction dag in such an ID has at most ℓ levels, for ℓ the number of registers in $\mathbb{A}$, and each level has at most $|P|$ nodes. Hence, emptyness of the language of $\mathbb{U}(\mathbb{A}, \mathbb{A}')$ can be decided in PSPACE using standard algorithms for NFAs. □

Since pdt-automata capture RQDs and the translations are polynomial, the PSPACE upper bound transfers to containment of RQDs. The matching lower bound is inherited from containment of standard regular expressions.

Corollary 4.16. *Problem* CONTAINMENT(positive RQDs) *is* PSPACE*-complete.*

5. Conclusions and Future Work

After conducting a detailed study of query containment for main classes of queries for graphs with data, we conclude that the picture here is quite different from the one for traditional navigational languages. In particular, there is a sharp contrast between RPQs or CRPQs, where containment is decidable, and any of the known extension of RPQs that handle data values. Undecidability for the class of RQMs comes as not a surprise, due to high complexity of query evaluation and powerful data manipulation mechanism, but we have seen that even the class of RQDs with good query evaluation properties can have undecidable containment.

The main observation of this paper is that the presence of inequality tests is a major detractor for the static analysis of data comparison queries. We summarise all of the results in Table 3.

As far as future work is concerned, we think there is much to be done in terms of static analysis of graph query languages. First, the obvious extension is to try and work on two-way (positive) RQMs and RQDs, that is, the languages where backward traversals of edges is allowed. Some results in this direction are given in [21], where two-way RQMs were considered. Doing two-way RQDs would possibly require extending register automata and pdt-automata into some form of two-way automata, and it is not clear that all of our results continue to hold under this extension. Second possibility is to study conjunctions of (positive) RQMs and RQDs, which would probably require a combination of our techniques and that of [9]. Finally, it could be interesting to look at graph queries over various description logics, where some results are known, but only about 2RPQs and C2RPQs [8].

Acknowledgements. Reutter and Vrgoč were funded by the Millennium Nucleus Center for Semantic Web Research under Grant NC120004. Vrgoč was also funded by the FONDECYT grant N11160383. We thank the reviewers for their helpful comments; and in particular for their suggestions on simplifying and improving the proofs of Theorems 3.11 and 4.12.

References

[1] Abiteboul, S., Hull, R., and Vianu, V. (1995). *Foundations of Databases*. Addison-Wesley.

[2] Angles, R. and Gutierrez, C. (2008). Survey of graph database models. *ACM Computing Surveys*, 40(1).

[3] Barceló, P. (2013). Querying graph databases. In *32th ACM Symposium on Principles of Database Systems (PODS)*.

[4] Barceló, P., Libkin, L., Lin, A. W., and Wood, P. T. (2012). Expressive languages for path queries over graph-structured data. *ACM Trans. Database Syst.*, 37(4):31.

[5] Barceló, P., Libkin, L., and Reutter, J. (2011). Querying graph patterns. In *30th ACM Symposium on Principles of Database Systems (PODS)*, pages 199–210.

[6] Barceló, P., Pérez, J., and Reutter, J. (2012). Relative expressiveness of nested regular expressions. In *AMW*, pages 180–195.

[7] Barceló, P., Reutter, J. L., and Libkin, L. (2013). Parameterized regular expressions and their languages. *Theor. Comput. Sci.*, 474:21–45.

[8] Bienvenu, M., Ortiz, M., and Šimkus, M. (2013). Conjunctive regular path queries in lightweight description logics. In *IJCAI*.

[9] Calvanese, D., De Giacomo, G., Lenzerini, M., and Vardi, M. (2000). Containment of conjunctive regular path queries with inverse. In *7th International Conference on Principles of Knowledge Representation and Reasoning (KR)*, pages 176–185.

[10] Calvanese, D., De Giacomo, G., Lenzerini, M., and Vardi, M. (2003). Reasoning on regular path queries. *ACM SIGMOD Record*, 32(4):83–92.

[11] Calvanese, D., De Giacomo, G., Lenzerini, M., and Vardi, M. Y. (2001). View-based query answering and query containment over semistructured data. In *DBPL*, pages 40–61.

[12] Consens, M. and Mendelzon, A. (1990). Graphlog: A visual formalism for real life recursion. In *9th ACM Symposium on Principles of Database Systems (PODS)*, pages 404–416.

[13] Cruz, I., Mendelzon, A., and Wood, P. (1987). A graphical query language supporting recursion. In *ACM Special Interest Group on Management of Data 1987 Annual Conference (SIGMOD)*, pages 323–330.

[14] Dex (2013). DEX query language, Sparsity Technologies. http://www.sparsity-technologies.com/dex.php.

[15] Florescu, D., Levy, A. Y., and Suciu, D. (1998). Query containment for conjunctive queries with regular expressions. In *PODS*, pages 139–148.

[16] Gremlin (2013). Gremlin Language. https://github.com/tinkerpop/gremlin/wiki.

[17] Gupta, A. and Mumick, I. S. (1995). Maintenance of materialized views: Problems, techniques, and applications. *IEEE Data Eng. Bull.*, 18(2):3–18.

[18] Harris, S. and Seaborne, A. (2013). SPARQL 1.1 query language. W3C recommendation. `http://www.w3.org/TR/sparql11-query/`.

[19] Kaminski, M. and Francez, N. (1994). Finite memory automata. *Theoretical Computer Science*, 134(2):329–363.

[20] Kostylev, E. V., Reutter, J. L., and Vrgoc, D. (2016). Static analysis of navigational XPath over graph databases. *Inf. Process. Lett.*, 116(7):467–474.

[21] Kostylev, E. V., Reutter, J. L., and Vrgoč, D. (2014). Containment of data graph queries. In *ICDT*.

[22] Lenzerini, M. (2002). Data integration: a theoretical perspective. In *PODS*, pages 233–246.

[23] Libkin, L., Martens, W., and Vrgoč, D. (2013a). Querying Graph Databases with XPath. In *ICDT*.

[24] Libkin, L., Reutter, J. L., and Vrgoč, D. (2013b). TriAL for rdf: Adapting graph query languages for rdf data. In *PODS*.

[25] Libkin, L. and Vrgoč, D. (2012a). Regular expressions for data words. In *LPAR*, pages 274–288.

[26] Libkin, L. and Vrgoč, D. (2012b). Regular Path Queries on Graphs with Data. In *ICDT*, pages 74–85.

[27] Neo4j (2013). Neo4j, The graph database. `http://www.neo4j.org/`.

[28] Neven, F., Schwentick, T., and Vianu, V. (2004). Finite state machines for strings over infinite alphabets. *ACM Trans. Comput. Log.*, 5(3):403–435.

[29] Schwentick, T. (2004). XPath query containment. *SIGMOD Record*, 33(1):101–109.

[30] Segoufin, L. (2006). Automata and logics for words and trees over an infinite alphabet. In *CSL*, pages 41–57.

[31] Tal, A. (1999). Decidability of inclusion for unification based automata. Master's thesis, Department of Computer Science, Technion - Israel Institute of Technology.

[32] Vrgoč, D. (2014). *Querying graphs with data*. PhD thesis, School of Informatics, University of Edinburgh.

[33] Wood, P. (2012). Query languages for graph databases. *Sigmod Record*, 41(1):50–60.

Appendix: Proofs of Proposition 4.9 and Lemma 4.11

Proposition 4.9. *Operation* $\circ$ *is associative and* $\langle 0,0\rangle$ *is neutral for* $\circ$.

Proof. For associativity, note that for any $\langle m_1, m_2\rangle$, $\langle m_1', m_2'\rangle$ and $\langle m_1'', m_2''\rangle$ in $\mathcal{M}$

$$\big(\langle m_1, m_2\rangle \circ \langle m_1', m_2'\rangle\big) \circ \langle m_1'', m_2''\rangle$$

$$= \begin{cases} \langle m_1, m_2 - m_1' + m_2'\rangle \circ \langle m_1'', m_2''\rangle, & \text{if } m_2 \geq m_1', \\ \langle m_1 - m_2 + m_1', m_2'\rangle \circ \langle m_1'', m_2''\rangle, & \text{if } m_2 \leq m_1', \end{cases}$$

$$= \begin{cases} \langle m_1, m_2 - m_1' + m_2' - m_1'' + m_2''\rangle, & \text{if } m_2 \geq m_1', m_2 - m_1' + m_2' \geq m_1'', \\ \langle m_1 - m_2 + m_1' - m_2' + m_1'', m_2''\rangle, & \text{if } m_2 \geq m_1', m_2 - m_1' + m_2' \leq m_1'', \\ \langle m_1 - m_2 + m_1', m_2' - m_1'' + m_2''\rangle, & \text{if } m_2 \leq m_1', m_2' \geq m_1''. \\ \langle m_1 - m_2 + m_1' - m_2' + m_1'', m_2''\rangle, & \text{if } m_2 \leq m_1', m_2' \leq m_1'', \end{cases}$$

$$= \begin{cases} \langle m_1, m_2 - m_1' + m_2' - m_1'' + m_2''\rangle, & \text{if } m_2 \geq m_1', m_2 - m_1' + m_2' \geq m_1'', \\ \langle m_1 - m_2 + m_1', m_2' - m_1'' + m_2''\rangle, & \text{if } m_2 \leq m_1', m_2' \geq m_1'', \\ \langle m_1 - m_2 + m_1' - m_2' + m_1'', m_2''\rangle, & \text{if } m_2 - m_1' + m_2' \leq m_1'', m_2' \leq m_1''. \end{cases}$$

The last equality holds because

$$\big((m_2 \geq m_1') \wedge (m_2 - m_1' + m_2' \leq m_1'')\big) \vee ((m_2 \leq m_1') \wedge (m_2' \leq m_1''))$$

is equivalent to

$$(m_2 - m_1' + m_2' \leq m_1'') \wedge (m_2' \leq m_1'').$$

Indeed, if $m_2 \geq m_1'$ and $m_2 - m_1' + m_2' \leq m_1''$ then $m_2' \leq m_1''$, and if $m_2 \leq m_1'$ and $m_2' \leq m_1''$ then $m_2 - m_1' + m_2' \leq m_1''$. Symmetrically, the last system is equal to $\langle m_1, m_2\rangle \circ \big(\langle m_1', m_2'\rangle \circ \langle m_1'', m_2''\rangle\big)$.

The neutrality axioms $\mu \circ \langle 0,0\rangle = \mu$ and $\langle 0,0\rangle \circ \mu = \mu$ for any $\mu \in \mathcal{M}$ hold by construction. □

Lemma 4.11. *Let*

$$[r_0, \lambda_0], [q_1, \lambda_1], [r_1, \lambda_1], \ldots, [r_{n-1}, \lambda_{n-1}], [q_n, \lambda_n]$$

be a run of a pdt-automaton, where, for every $1 \leq i \leq n$, *configuration* $[q_i, \lambda_i]$ *is reachable from* $[r_{i-1}, \lambda_{i-1}]$ *by a transition with action* μ_i. *For every* j *and* k, $1 \leq j \leq k \leq n$, *the numbers* m_1 *and* m_2 *in* $\langle m_1, m_2\rangle = \mu_j \circ \cdots \circ \mu_k$ *are such that*

– *Depth*$(r_{j-1}) - m_1 + m_2 =$ *Depth*(q_k),

– *Depth*$(r_{i-1}) - m'_1 \geq$ *Depth*$(r_{j-1}) - m_1$ *for any* i, $j \leq i \leq k$, *with* $\mu_i = \langle m'_1, m'_2 \rangle$,

– *there exists* i, $j \leq i \leq k$, *such that* *Depth*$(r_{i-1}) - m'_1 =$ *Depth*$(r_{j-1}) - m_1$ *for* $\mu_i = \langle m'_1, m'_2 \rangle$ *(i.e., the inequality above becomes an equality).*

Proof. Let us fix j and prove the statement by induction on k.

If $k = j$, then it holds by definition of pdt-automata.

Assume that the statement holds for $k - 1 \geq j$, that is, m^1_1 and m^1_2 in $\langle m^1_1, m^1_2 \rangle = \mu_j \circ \cdots \circ \mu_{k-1}$ are such that *Depth*$(r_{j-1}) - m^1_1 + m^1_2 =$ *Depth*(q_{k-1}); *Depth*$(r_{i-1}) - m'_1 \geq$ *Depth*$(r_{j-1}) - m^1_1$ for any i, $j \leq i \leq k - 1$, and $\mu_i = \langle m'_1, m'_2 \rangle$; and there is i such that *Depth*$(r_{i-1}) - m'_1 =$ *Depth*$(r_{j-1}) - m^1_1$. Let also $\mu_k = \langle m^2_1, m^2_2 \rangle$. Next we show that the statement holds for k as well. We need to show that m_1 and m_2 in $\langle m_1, m_2 \rangle = \langle m^1_1, m^1_2 \rangle \circ \langle m^2_1, m^2_2 \rangle = \mu_j \circ \cdots \circ \mu_k$ satisfy the conditions on the used registers. We have two cases.

Assume first that $m^1_2 \geq m^2_1$. Then $m_1 = m^1_1$ and $m_2 = m^1_2 - m^2_1 + m^2_2$ by the definition of $\circ$. On the one hand, *Depth*$(q_k) =$ *Depth*$(r_{k-1}) - m^2_1 + m^2_2$ by the definition of pdt-automata. On the other, *Depth*$(r_{k-1}) =$ *Depth*(q_{k-1}) and *Depth*$(q_{k-1}) =$ *Depth*$(r_{j-1}) - m^1_1 + m^1_2$ by the assumption. Therefore, *Depth*$(q_k) =$ *Depth*$(r_{j-1}) - m^1_1 + m^1_2 - m^2_1 + m^2_2 =$ *Depth*$(r_{j-1}) - m_1 + m_2$, as required. Next, by the assumption, the smallest number of a manipulated register between j and $k - 1$ is *Depth*$(r_{j-1}) - m^1_1 + 1$, and this manipulation happens on the step from $[r_{i-1}, \lambda_{i-1}]$ to $[q_i, \lambda_i]$. On the one hand, *Depth*$(r_{j-1}) - m^1_1 + 1 =$ *Depth*$(r_{k-1}) - m^1_2 + 1$. On the other, the smallest number manipulated on the step from $[r_{k-1}, \lambda_{k-1}]$ to $[q_k, \lambda_k]$ is *Depth*$(r_{k-1}) - m^2_1 + 1$. Since $m^1_2 \geq m^2_1$, the smallest number between j and $k - 1$ is the smallest between j and k as well, as required.

The other case is when $m^1_2 < m^2_1$. Then $m_1 = m^1_1 - m^1_2 + m^2_1$ and $m_2 = m^2_2$, and the proof goes in exactly the same lines as for the case $m^1_2 \geq m^2_1$ except that now the overall smallest number is *Depth*$(r_{k-1}) - m^2_1 + 1$ on the step from $[r_{k-1}, \lambda_{k-1}]$ to $[q_k, \lambda_k]$. □