Structural Recursions on Edge-Labelled Graphs

Thesis

Balázs Kósa

Supervisor: András Benczúr D.Sc.

Eötvös Loránd University, Faculty of Informatics
Department of Information Systems

Ph.D. School of Computer Science
András Benczúr D.Sc.

Ph.D. Program of Information Systems
András Benczúr D.Sc.

Budapest, 2013

Abstract

In the dissertation we introduce a family of graph transformators which for brevity we call *structural recursions* and then examine some of their mathematical properties. The practical importance of structural recursions comes from their usability in the field of semi-structured and XML databases, where data is modelled by labelled graphs and trees respectively.

First, a more general definition of structural recursions is introduced which resembles to the definition of tree automata [12]. This definition is far from being trivial, since the evaluation of conditions should be developed elaborately owing to the presence of cycles. Then five different classes of structural recursions are differentiated and their expressive power is compared. Afterwards, the usual operations, i.e., complement, union and intersection are defined over structural recursions.

Next, in order to understand the functioning of structural recursions in a deeper level they are compared with two well known types of automata, namely non-deterministic finite state string automata and alternating tree automata. In these comparisons structural recursions are considered as acceptors, which means that only the emptiness or non-emptiness of the result is of importance. It is shown that structural recursions can be simulated by alternating tree automata and vice versa. However, despite this strong connection there are several important differences between the two formalisms.

Finally, the usual static analytical questions, i.e., the problems of emptiness and containment, are addressed for the different classes of structural recursions. The complexity of the problems ranges from PTIME through coNP-completeness and $\Sigma_k P$-hardness to DEXPTIME-completeness.

Acknowledgments

First and foremost, I would like to thank András Benczúr for valuable feedback, support and guidance. Many thanks go to Attila Kiss as well, it has been a great privilege to work with and learn from him.

Finally, I thank first of all my Mum and all members of my family for their unconditional support and love.

Contents

1 Introduction **1**

2 Preliminaries **9**

2.1 Automata 9

2.2 The data model of structural recursions 12

2.3 Structural recursions with natural semantics 18

3 Structural recursions with operational semantics **22**

3.1 Simple structural recursions 22

3.2 Structural recursions with conditions 28

3.3 Syntax 29

3.4 Semantics 30

4 Basic notions and statements **37**

4.1 Static analytical questions 40

4.2 Classes of structural recursions 46

4.3 Operational homomorphism 50

4.4 Simulating data graphs with data trees 57

5 Operations on structural recursions **69**

5.1 Complement 69

5.1.1 Complement of structural recursions in (SR(n.i., i.)). . 69

5.1.2 Complement of structural recursions in $SR(n.i., i., el)$. 75

5.2 Intersection, union 76

6 Structural recursions and automata **80**

6.1 Structural recursions in $SR()$ and NDFSA 80

6.1.1 Rewriting of a structural recursion in $SR()$ into an NDFSA 80

6.1.2 Rewriting of an NDFSA into a structural recursions . . 82

6.2 Alternating tree automata and structural recursions 84

6.2.1 Rewriting structural recursion to alternating tree automata 84
6.2.2 Rewriting alternating tree automata to structural recursion 90

7 The problem of emptiness **100**
7.1 Structural recursions in $SR(n.i.)$ 102
7.2 Structural recursion in $SR(n.i., i.)$ 104
7.3 Structural recursions in $SR(n.i., i., \leq k)$. 113

8 The problem of containment **123**
8.1 Structural recursions in $SR()$ 124
8.1.1 Separators 125
8.1.2 Deterministic structural recursions 130
8.1.3 Structural recursions in $SR(n.i., \vee)$ 131
8.2 Structural recursions in $SR(n.i.)$ 135

9 Summary and future work **139**

List of Figures

1 Summary of the complexities of the emptiness and containment problems 6
2 Preliminaries 13
3 Intersection of schema graphs 16
4 An example for simulation 18
5 The syntax of transformation rules 18
6 The natural semantics of structural functions 21
7 Simple structural recursions 25
8 Exponential output 28
9 Syntax of transformation rules with conditions 29
10 The semantics of structural recursions with conditions 32
11 The `Complete_Structural_Recursion` algorithm 39
12 A run of the `Tree_Constructor` algorithm 57
13 The `Tree_Constructor` algorithm 58
14 `Tree_Simulator` algorithm 64
15 Rewriting of node- and edge-labelled data trees 91
16 Truth assignment instances 117
17 Separators 126
18 Reductions 135

1 Introduction

In the dissertation we introduce a family of graph transformators which for brevity we call *structural recursions* and then examine some of their mathematical properties. The practical importance of structural recursions comes from their usability in the field of semi-structured and XML databases, where data is modelled by labelled graphs and trees respectively. In general structural recursions are applied on such (mathematical) objects that are built up by certain constructors. For example in propositional logic these constructors are the logical connectives $\neg, \wedge, \vee$. In this context a mapping ζ over propositional formulae can be defined as a structural recursion. First ζ is defined over the propositional variables. For a complex formula $\neg P$ or $P_1 \vartheta P_2$, $\zeta(\neg P)$, $\zeta(P_1 \vartheta P_2)$ is defined by combining the values of $\zeta(P)$ and $\zeta(P_1), \zeta(P_2)$ respectively [31] ($\vartheta \in \{\wedge, \vee\}$).

In the context of databases structural recursions working on sets were recommended as a query language alternative in the early 90's to be able to overstep the limitations of the relational data model, and to move toward to the expressive capability of a relational query language embedded into a general purpose programming language [7]. In the following years structural recursions were used to query (nested) collections based on sets [10], bags [23] and lists [17]. However, in these works a more general form of structural recursions was introduced than that of used here. Namely, for an element of such a collection an arbitrary function interpreted over these elements may be called that obeys some certain restrictions like commutativity and idempotence [29]. As it has been summarized in [27] such structural recursions lead to highly expressive query languages. For instance in [10] it was shown that structural recursion on nested relations is equivalent with the powerset algebra of Abiteboul and Beeri [1].

In [9] structural recursions were applied to process unordered trees and graphs. Here, query language UnQL was introduced and recommended as an alternative with many advantageous properties to handle semi-structured data. Structural recursions formed the basis of the algebra UnCal of UnQL.

The definition of the structural recursions formulated in this dissertation (without conditions) is inspired by the definition given in this paper. The slight variances in the syntax will be indicated on the appropriate places. The semantics will be defined in a different manner, though the equivalence of the approaches could be proven.

Beside structural recursions UnCal consisted of graph constructors and a conditional branching in *if-then-else* form. In the condition the non-emptiness of an UnCal query could be tested. In [18] this possibility was extended to be able to take the Boolean-combination of such checks. Furthermore, in [27] the aforementioned approaches were synthesized to develop a language with which both embedded lists and ordered trees can be handled.

In contrast to UnCal in our version of structural recursions *if-then-else* conditions are moved inside structural recursions. What is more, although structural recursions are graph transformators in this context, in most of our previous works as well as here we have been focusing on the question of constructing, hence structural recursions have being considered rather as a special kind of automaton which accepts its input if a non-empty output is returned.

In the following paragraphs our previous papers as well as the new results of this dissertation are summarized.

Our previous and recent results. In [4] we introduced a new way of defining the semantics of structural recursions. A structural recursion comprises structural functions which may call each other. In our semantics we make use of a special kind of schema graph with which the interactions of structural functions are displayed. It is similar in nature to the graph representation of non-deterministic finite state string automata, where the nodes represent states, while the edges transitions from one state into another as a result of a symbol. We called this representation of structural recursions *operational graph* and although several details have been changed since the first definition the basic idea is still preserved in this dissertation. The input

data graph is intersected with the operational graph and the result of this operation is used then to construct the final result. An edge of this intersection illustrates that the corresponding edge of the input is processed by the structural function represented by the starting node of the corresponding edge in the operational graph and then the structural function represented by the endnode is called. Note that in our data model edge-labelled graphs are used.

We also showed how structural recursions can intertwine with restrictions imposed on the structure of the data. These are expressed by means of schema graphs [8] and since the operational graphs are also special schema graphs using intersection certain constraints can be incorporated into the structural recursion. In some scenarios this can lead to significant optimization. The connection between data and schema graphs is based on simulation, in which nodes of a data graph are mapped to nodes of a schema graph and thus "types" can be assigned to the edges of the data graph. With this technique one can prescribe the allowable labels of the outgoing edges of an edge of a certain type. By defining a slightly different variant of the simulation we described how the same idea could be used to specify the compulsory labels that the outgoing edges of an edge have to have. Our method based on intersection also proved to be useful in the typing question, where the structure of the output is restricted. Here we examined both kinds of restricting mechanisms. Finally, the emptiness question was also addressed and it was shown to remain tractable, i.e., solvable in polynomial time, both for typed and untyped data.

In [19] we extended structural recursions with conditions in which the Boolean combination of not-isempty and isempty logical functions and equality comparisons can be taken. Applying a (non-)isempty function one can check whether a structural function returns a (non-)empty output or not. In an equality comparison the equality of a given constant and the label of the edge being processed can be checked. The conditions are formulated in an if-then-else style. Using these enhanced structural recursions a core fragment

of XPath 1.0 [13] was simulated. We have also explained, in a similar fashion as in [4], how Document Type Definitions (DTD) [6] and Extended Document Type Definitions (EDTD) [24] can be incorporated into the simulating structural recursions. In the XPath expressions only axis `child, parent, descendant, ancestor` were allowed to apply. Since structural recursions process the data in a top-down manner, in order to simulate the functioning of upward axis `parent, ancestor` registers were introduced. A register is of the form $X^a_{f_1} = X^a_{f_2}$ and this example specifies that only those edges a should be considered in the construction of the output that are processed by both f_1 and f_2. The use of the registers is only necessary when there are at least two upward axis s.t. the first one is not immediately followed by the second one but there is a downward axes between them. In such cases by using a structural recursion the output can be obtained by a single traverse of the input data graph which again may result in considerable optimization in certain cases.

In general our simulation offers an efficient implementation working in time $O(|D||Q|)$, where $|D|$ denotes the size of the data, whereas $|Q|$ the size of the query. This coincides with the speed of the algorithm developed by Gottlob et al. [15].

In [22] this simulation was extended to a core fragment of XSLT 1.0 [11]. Here we made use of the formal model of XSLT offered in [5]. The presence of variables whose values are given by means of XPath expressions made the simulation rather intricate. In this case the implementation entailed by this representation works in time $O(|D^2||Q|^2)$.

In our next paper [20] we focused on the usual static analytical questions, namely the questions of emptiness and containment. In the first case for a given structural recursion f it is asked whether there is a data graph I s.t. $f(I)$ is not empty. In the second case, for two structural recursions f and g it is checked whether there is a data graph for which g returns a non-empty, while f an empty output. In this dissertation a variant of this question will also be addressed. This distinction is inspired by a similar differentiation in

the context of XPath expressions [28]. In [20] we only examined a fragment of structural recursions in which the conditions consisted of a single not-isempty logical function. In other words, neither isempty logical functions nor equality comparisons were allowed to use. On the other hand we distinguished two cases on the basis that whether the application of the else-branches is permitted or not. We found that in the second case the emptiness question can be answered in quadratic time, while the problem of containment is coNP-hard. In the first case we showed how the two problems can be reduced to each other in polynomial time and proved that both questions are PSPACE-hard in general.

In this dissertation the previous research is extended in several different ways. First of all the isempty logical functions are also involved in the investigations. Secondly, in the conditions arbitrary Boolean combination of the not-isempty, isempty conditions may be formulated. Thirdly, in [20] structural recursions worked as "edge rewriters". More precisely, they could either change the label of the processed edge or they could delete it. In both cases they could also invoke another structural function to process the subgraph under the actual edge. In contrast, in this case as a result of traversing an edge an arbitrary forest may be constructed, furthermore, arbitrary number of structural functions may be called as well. Note that in [9] structural recursions were also defined in this more general way. In the simulation of XPath and XSLT we also used this variant [19, 22]. Here, several interesting subclasses of structural recursions of this type are examined. The summary of our results is to be found in Figure 1.

In the table $SR(n.i., i., el)$ denotes the most general class of structural recursions, where not-isempty, isempty and else-branches are all allowed to use. The meaning of classes $SR()$, $SR(n.i.)$, $SR(n.i., i)$ is similar. In addition in $SR(n.i., \vee)$ only disjunctions may be applied in the conditions, whereas in $SR(n.i., i., \leq k)$ the conditions may be embedded up to the k^{th} level. Except from the question of containment in $SR(n.i.)$ and the questions of emptiness and containment in $SR(n.i., i., \leq k)$ all problems enumerated in the table are

	EMPTINESS	CONTAINMENT
deterministic	PTIME	PTIME
$SR()$	PTIME	coNP
$SR(n.i., \vee)$	PTIME	coNP
$SR(n.i.)$	PTIME	PSPACE-hard
$SR(n.i., i., \leq k)$	$\Sigma_k P$-hard	$\Pi_k P$-hard
$SR(n.i., i.)$	DEXPTIME	DEXPTIME
$SR(n.i., i., el)$	DEXPTIME	DEXPTIME

Figure 1: The summary of the results of this dissertation with respect to the emptiness and containment problems.

complete for the corresponding complexity classes.

In [21] we extended XPath expressions with *named expressions* by means of which a name can be assigned to an XPath expression, and then this name can be used wherever a location step may occur. Named expressions may contain other named expressions or themselves, which gives rise to a new kind of recursivity different from the recursivity already involved in axis `descendant`, `ancestor` etc. The idea was inspired by the possibility of defining functions in XQuery, which may call other user-defined functions in their body. Named expressions can be regarded as a simplified version of user-defined functions in the context of XPath.

Going into details it was proven that it is not possible to write such an XPath expression that selects each node of a path of arbitrary length in which a and b nodes follow each other in turn. On the other hand in the following XQuery function F the XPath expression of the `for` clause select exactly these nodes, when F invokes itself recursively.

```
define function F($x)
{
    for $y in $x/(child::a | child::  b)
    return <r>{string($y/@id), F($y)}</r>
}
```

The corresponding XPath query with named expressions is as follows

$$\texttt{F}: \{(\texttt{self} :: *|(\texttt{child} :: \texttt{a}|\texttt{child} :: \texttt{b})/\texttt{F}).\}$$

It is not difficult to see that $\texttt{self} :: */\texttt{F}$ selects the same nodes of an input as the preceding XQuery function in its `for` clause.

In our work only downward axis were taken into consideration, however the use of negation was permitted in the predicates. First, we clarified the relationship between the different types of recursions including transitive closure, then we explained how XPath expressions with named expressions can be simulated with structural recursions and vice versa. The possibility of this two-way rewriting enabled us to directly use the aforementioned static analytical results to the appropriate classes of the extended XPath expressions.

Reverting to the content of this dissertation besides the complexity results the relationship between structural recursions and different types of automata is also discussed. In these comparisons structural recursions are considered as acceptors, which means that only the emptiness or non-emptiness of the result is of importance. First, the connection between the structural recursions without conditions and the non-deterministic finite state string automata is clarified. Secondly, it is shown that in the general case structural recursions can be simulated by alternating tree automata [12] and vice versa. This relationship may enable us to apply the complexity results of Figure 1. to the appropriate classes of the latter formalism. However, the details are not given here.

On the other hand, despite this strong connection there are several important differences between the two formalisms. Firstly, structural recursions process edge-labelled graphs with possible cycles, while alternating tree automata work on node-labelled trees. More importantly, structural recursions are applied on such data graphs where the number of the outgoing edges of an arbitrary node is not limited, in addition no order is defined among these edges, whereas alternating tree automata only accept ranked ordered trees as inputs. This second difference distinguishes structural recursions from

unranked tree automata [12] as well, since the latter although also works on unranked trees, these trees should be ordered too. However, in many theoretical investigations [2] as well as in practical applications it is more advantageous to handle unordered data especially in the field of databases. All together these differences underlie the importance of structural recursions of their own and make them a significant candidate whenever a practical application working on graphs is to be modeled formally.

The dissertation is organized in the following way. In Section 2 the basic notions are explained. First, the definitions of string automata and alternating tree automata are given. Then the data model on which structural recursions work and the relating concepts are described. Lastly, structural recursions without conditions are introduced. The semantics is based on the more general and usual meaning of structural recursions and it is only applicable on trees. In Section 3 the semantics of structural recursions is defined in a different way enabling the process of cycles. This definition relies on operational graphs. After proving the equivalence of the two approaches it is shown how the operational graphs should be extended to be able to handle conditions. In Section 4 first the different classes of structural recursions (cf. Figure 1.) are compared in terms of expressive power. Secondly, an extension of the simulation relation is introduced with which among others the containment of structural recursions can be characterized. Finally, in the third part of this section two algorithms are developed constructing data trees from data graphs which in certain aspects behave in the same way as the original data graphs. The intersection, union and complement of structural recursions are defined then in Section 5. The relationship with the previously mentioned automata is clarified in Section 6. In Section 7 the emptiness whereas in Section 8 the containment problem is addressed. The conclusions are drawn in the last section. In order to improve the readability of the main text the most intricate proofs are placed in the Appendix attached to the end of this work.

All of the results of this work and the aforementioned papers [4][19][22][20][21]

are my own, however, I am deeply indebted for my co-writer colleagues András Benczúr and Attila Kiss for their useful comments and advice.

2 Preliminaries

2.1 Automata

Finite state (string) automata. Let Σ be a finite set of constants. Σ^* denotes the set of finite strings of symbols from Σ. A non-deterministic, finite state string automaton is a tuple $A = (Q, \Sigma, Q_I, Q_f, \Phi)$, where Q is the finite set of states, $Q_I, Q_f \subseteq Q$ are the sets of initial and final states and Φ is the set of transition rules. A transition rule is of the form:

$$a(q_1) \to q_2, \text{ where } a \in \Sigma, q_1, q_2 \in Q.$$

A *run* of A on word $w = (a_1 \dots a_n) \in \Sigma^*$ is a sequence of states $q_{i_1}, \dots, q_{i_{n+1}}$ s.t. q_{i_1} is an initial state, and $a_j(q_{i_j}) \to q_{i_{j+1}}$ is a transition rule $(1 \leq j \leq n)$. A run is *accepting*, if $q_{i_{n+1}}$ is in Q_f. A finite state string automaton is *deterministic*, if there are no two transition rules in Φ with the same left-hand side [16]. In the sequel, finite state non-deterministic automata will be referred as *NDFSA-s*.

Node-labelled, ranked data trees. In a node-labelled, ranked tree the nodes are represented as finite lists over natural numbers, $\mathbb{N}^*$, where $\mathbb{N}$ denotes the set of natural numbers. Furthermore, every node is allowed to have only a limited number of outgoing edges. This number depends on the label of the node in question.

Formally, a ranked alphabet is a couple

$$\Upsilon = (\Omega, Arity),$$

where Ω is a finite set of constants and $Arity$ is a function from Ω to $\mathbb{N}$. The arity of symbol $\upsilon \in \Omega$ is denoted by $Arity(\upsilon)$. The set of Υ-trees, in notation $\mathcal{T}^{\Upsilon}_{node}$, is inductively defined as follows [12]:

(i) every $u \in \Omega$ is an Υ-tree, where $Arity(u) = 0$,

(ii) if $Arity(u) = n$ and $t_1, \ldots, t_n \in \mathcal{T}^{\Upsilon}_{node}$, then $u(t_1, \ldots, t_n)$ is an Υ-tree, $n \geq 1$.

Here, *node* in the subscript indicates, that these trees are node-labelled. As it has been already mentioned nodes are represented as finite lists of natural numbers. In detail, the set of nodes of t denoted by $V.t$ is defined as follows: if $t = u(t_1, \ldots, t_n)$, where $u \in \Omega$, $n \geq 0$, then

$$V.t = \{\epsilon\} \cup \{iv \mid i \in \{1, \ldots, n\}, v \in V.t_i\}.$$

Here, ϵ denotes the empty list and it is the root of t in this case. A node v of t_i is substituted with iv. Thus, for example the root of t_2, which was ϵ before this construction, is substituted with 2, while the first child of this root, which was 1, is substituted with 21 etc. With $lab^t(u)$ we denote the label of node u in t. An example of a node labelled tree over ranked alphabet $\{a, b\}$, $Arity(a) = 2$, $Arity(b) = 0$, can be found in Figure 2.(a). The symbols in parenthesis indicate the labels of the corresponding nodes.

Alternating tree automata. This definition of the alternating tree automata is based on the definition given in [12]. Let P be a set of symbols. With $\mathcal{B}^+(P)$ we denote the set of positive propositional formulas over P. For example

$$q_1 \vee q_2 \wedge q_3 \in \mathcal{B}^+(\{q_1, q_2, q_3\}).$$

An alternating automaton over ranked alphabet $\Upsilon = (\Omega, Arity)$ is a tuple $A = (Q, \Upsilon, I, \Psi)$, where Q is the finite set of states, $I \subseteq Q$ is the set of initial states, and Ψ is the set of transition rules, which, here, are mappings:

$$(Q, \Omega) \to \mathcal{B}^+(Q \times \mathbb{N}) \cup \{true, false\}.$$

Remark 2.1. Here and in the rest of this work we use the term mapping in a broader sense. Namely, a mapping may assign several different values to a member of its domain. In other words, a mapping is not necessarily a function.

Reverting to the introduction of the alternating tree automata for an arbitrary $\upsilon \in \Omega$, if $Arity(\upsilon) > 0$, (q, υ) is always mapped into $\mathcal{B}^+(Q \times \{1, \ldots, Arity(\upsilon)\})$. On the other hand, if $Arity(\upsilon) = 0$, $true$ or $false$ is assigned to (q, υ). Informally, the intended meaning of the rule:

$$(q, a) \rightarrow (q_1, 2) \wedge (q_3, 1) \vee (q_2, 3)$$

is that after processing an a labelled node in state q, the corresponding tree will be accepted, if the second and first branch in states q_1, q_3 are both accepted, or the third branch in state q_2 is accepted. Here $Arity(a) = 3$. Note that this definition is given in a top-down manner, which is more convenient in the alternating case.

A run of an automaton A on tree t is a node-labelled tree λ, whose labels are from $(Q \times \mathbb{N}^*) \cup \{true, false\}$. Informally, node label $(q, 11)$ in a run represents that node 11 of the input tree was processed in state q. Formally, suppose that for node u of λ, $lab^\lambda(u) = (q, w)$ and $lab^t(w) = v$, then

(i) if $Arity(v) > 0$, then $u.j \in V.\lambda$, $lab^\lambda(u.j) = (q_{i_j}, w.k_j)$, where $(q, v) \rightarrow \phi \in \Phi$, (q_{i_j}, k_j) is in ϕ, $\{(q_{i_1}, k_1), \ldots, (q_{i_n}, k_n)\} \subseteq Q \times \{1 \ldots Arity(v)\}$.

(ii) if $Arity(v) = 0$, then $u.1 \in V.\lambda$, $lab^\lambda(u.1) = \phi$, where $(q, v) \rightarrow \phi \in \Phi$, $\phi \in \{true, false\}$.

Here, for sake of transparency $u.j$ has been used for denoting the j^{th} child of u instead of uj.

Example 2.2. An example for automaton $(\{q_1, q_2, q_3\}, \Upsilon, \{q_1\}, \Psi)$ can be found in Figure 2.(b)-(c). Here Ω is $\{a, b, c\}$ with $Arity(a) = 2$, $Arity(b) = 1$, $Arity(c) = 0$ and Ψ consists of the following rules:

$(q_1, a) \rightarrow (q_2, 1) \wedge (q_3, 2) \vee (q_2, 2)$	$(q_2, a) \rightarrow (q_3, 1) \vee (q_1, 2)$	$(q_3, a) \rightarrow (q_3, 1)$
$(q_1, b) \rightarrow (q_2, 1)$	$(q_2, b) \rightarrow (q_2, 1)$	$(q_3, b) \rightarrow (q_3, 1)$
$(q_1, c) \rightarrow false$	$(q_2, c) \rightarrow false$	$(q_3, c) \rightarrow true$

Note that the nodes of λ are also from $\mathbb{N}^*$.

In the *evaluation* of the run λ we assign truth values to the nodes in a bottom-up manner. This function is denoted Λ ($\Lambda : V.\lambda \to \{true, false\}$). If a node of λ is of case (ii), then it has a child with a $true$ or $false$ label. We assign the corresponding truth value to this node. For node u of case (i) consider its children one after the other. Keeping the notations of case (i), suppose that for the j^{th} child, $u.j$, a truth value has been already assigned. Assume that $lab^{\lambda}(u.j) = (q_{i_j}, w.k_j)$, which means that (q_{i_j}, k_j) occurs in ϕ at least once. Substitute this (these) instance(s) of $(q_{i_j}, w.k_j)$ with the assigned truth value of $u.j$. If at the end ϕ becomes $true$, then assign $true$ to u. Similarly, if ϕ becomes $false$, then assign $false$ to u. The algorithm stops, if a truth value to the root of λ is assigned, or no other truth value can be assigned to any of the nodes of λ.

A run is *accepting*, (i) if $lab^{\lambda}(\epsilon) = (q_i, \epsilon)$, $q_i \in I$, (ii) $\Lambda(\epsilon) = true$. In other words the run should start with an initial state, and the evaluation should assign $true$ to the root of λ. For an example consider Figure 2.(d).

The complement of an alternating tree automaton can be constructed as follows [12]. For $A = (Q, \Upsilon, Q_I, \Psi)$ consider $\tilde{A} = (\tilde{Q}, \Upsilon, \tilde{Q}_I, \tilde{\Psi})$. Here, each state q_i of A has a correspondence $\tilde{q}_i$ in $\tilde{A}$. Moreover, $\tilde{Q}_I$ contains $\tilde{q}_i$, if q_i is in Q_I. In $\tilde{\Psi}$ each transitional rule $(q_i, a) \to \phi$ should be changed to $(\tilde{q}_i, a) \to \tilde{\phi}$, where $\tilde{\phi}$ is

(i) $\neg\phi$, if $\phi \in \{true, false\}$.

(ii) Otherwise, each (q_j, k) in ϕ is changed to $(\tilde{q}_j, k)$, then each conjunction is substituted with a disjunction and each disjunction with a conjunction.

2.2 The data model of structural recursions

Data graphs. Originally, contrast to the XML tree model, where the nodes are labelled, in the context of semistructured databases structural recursions were defined on edge-labelled graphs [9]. In this dissertation we decided to adhere to this tradition.

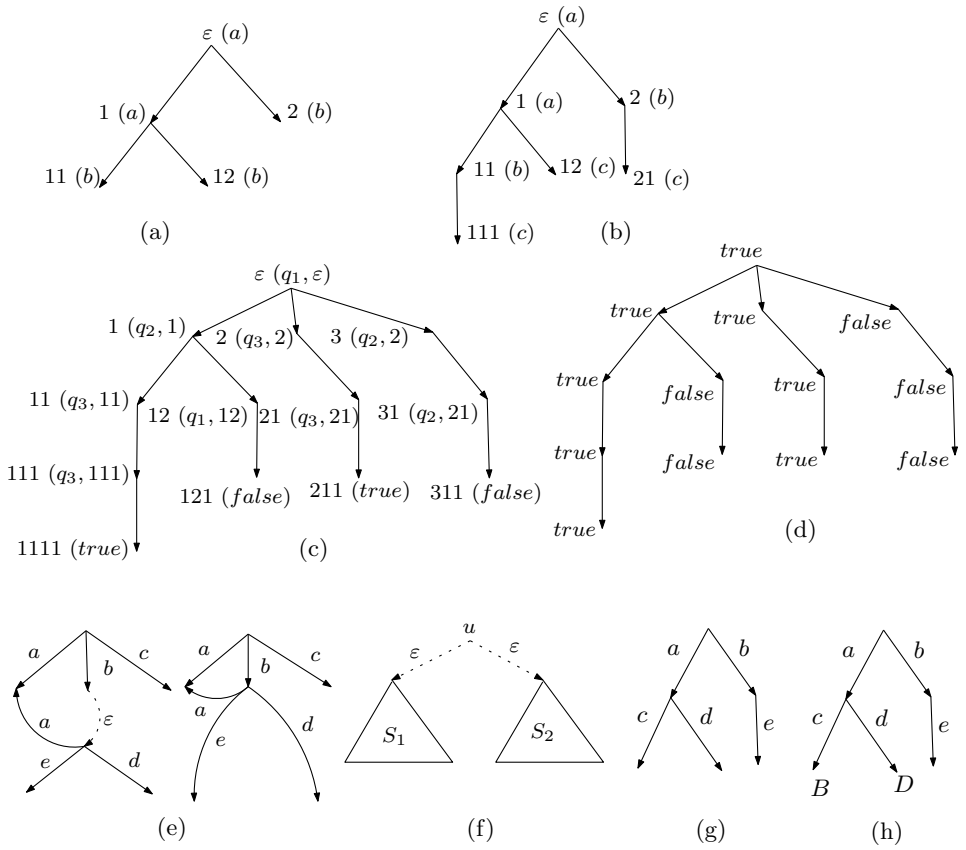

Figure 2: (a) A node-labelled tree. (b) A node-labelled tree. (c) A run of the alternating tree automaton of Example 2.2. on the tree on (b). (d) The evaluation of the run on (c). (e) An example of eliminating ε edges. (f) The union of data graphs. (g) A data tree. (h) A data tree with leaf labels.

Let Σ be a recursively enumerable set of constants. Then a *data graph* I is a triplet $I = (V, E, v_0)$, where V is the set of nodes, $E \subset V \times \Sigma \times V$ is the set of labelled, directed edges and v_0 is the distinguished root [9]. In the sequel data graphs will also be called *instances*. The sets of nodes and edges of an instance I will be denoted $V.I$, $E.I$. Furthermore, in what follows we respectively denote $\mathcal{D}^{\Sigma}_{edge}$, $\mathcal{T}^{\Sigma}_{edge}$, $\mathcal{F}^{\Sigma}_{edge}$ the set of data graphs, data trees and data forests with edge-labels from Σ. Here, *edge* in the subscript indicates that in this case edges are labelled instead of nodes. If it is clear from the context, then we will omit this notation. With $\mathcal{D}^{\Sigma}_{edge}(B), \mathcal{T}^{\Sigma}_{edge}(B), \mathcal{F}^{\Sigma}_{edge}(B)$

we denote that the leaves are labelled with labels from set B. Note that no order has been defined among the edges or the nodes of the data graphs.

Basic notions of graphs. A sequence of subsequent edges of a graph, $a_1 \dots a_n$ in notation, is called *path*. By *cycle* we mean a path whose first and last node are the same. Looping edges will also be considered as cycles. In what follows we will only consider *connected* graphs as data graphs. This means that all nodes of the data graph, possibly with the exception of the root itself, is reachable from the root. A *subgraph* $\hat{I}$ of a data graph $I = (V, E, v_0)$ is a triple $(\hat{V}, \hat{E}, \hat{v}_0)$, where $\hat{V} \subseteq V$, $\hat{E} \subseteq E$, $\hat{v}_0 \in \hat{V}$. Again, we will take into consideration only connected subgraphs. By *pregraph* we mean a subgraph, whose root is the same as that of the original graph. Two edges having the same starting node are called *neighbours*. Edge e_1 is a *child* of edge e_2 and e_2 is a *parent* of e_1, if the starting node of e_1 is the same as the end node of e_2. S A *root-edged* instance has only one outgoing edge from the root. This edge will be referred as the *root edge*.

ε edges. ε edges are introduced in order to make the explanations more transparent. By means of them graphs will be connected by the contraction of certain nodes of these graphs. At the end of the constructions ε edges should always be eliminated. The elimination is accomplished in the following way: in an arbitrary instance I let $(u, \varepsilon, v) \in E.I$ be an ε edge. For every edge $(v, a, w) \in E.I$ (the starting node is the same as the end node of (u, ε, v)) add (u, a, w) to $E.I$. Afterwards the preceding (v, a, w) edges and (u, ε, v) should be deleted. As an example consider Figure 2.(e).

Union. Let I_1, I_2 be arbitrary instances and let u be a new node different from all of the nodes of I_1 and I_2. Add ε edges from u to the roots of I_1 and I_2 and then eliminate these ε edges. The resulting graph is defined to be the *union* of I_1 and I_2 [9]. For a graphical representation consider Figure 2.(f).

Data trees and semistructured data expressions. Beside the definition given earlier for data graphs data trees can be represented in a different way. Namely, they can be built up using three constructors: the empty graph $\{\}$ consisting of a node only, the singleton set $\{l : t\}$, which is a directed l edge with subtree t in its end node, and the aforementioned union operation. This representation is called the *ssd-expression* of the tree [3] (ssd: semistructured data). As an example consider

$$\{a : \{c : \{\}\} \cup \{d : \{\}\}\} \cup \{b : \{e : \{\}\}\},$$

which is the ssd-expression of the tree in Figure 2.(g).

The *subtrees* of a data tree t are defined as follows:

(i) t is a subtree of t.

(ii) If $t = t_1 \cup \ldots \cup t_n$, then t_i is a subtree of t $(1 \leq i \leq n)$.

(iii) If $t = \{a : \hat{t}\}$, then $\hat{t}$ is also a subtree of t.

We also introduce the disjunctive union [9] of trees, $t_1 \oplus t_2$, which as its name suggests, returns a forest constituted by t_1 and t_2.

Finally, in the definition of the structural recursions special data trees will be used, whose leaves may be labelled as well. To represent such trees *free ssd-expressions* are introduced, in which the empty graph can be substituted with a label. As an example consider the data tree on Figure 2.(h), whose free ssd-expression is as follows:

$$\{a : \{c : \{B\}\} \cup \{d : \{D\}\}\} \cup \{b : \{e : \{\}\}\}.$$

In the sequel we will blur the distinction between (free) ssd-expressions and data trees (with leaf labels).

Schema graphs. Originally, schema graphs were introduced in order to be able to impose restrictions on the structure of data graphs [8]. Here, they will be used to represent the structure of structural recursions, i.e., how

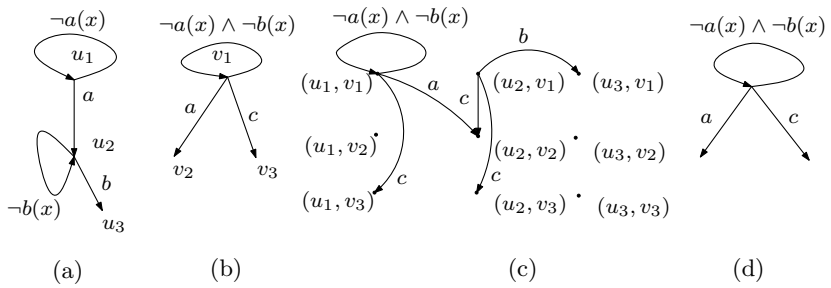

Figure 3: (a) A data graph. (b) A data graph. (c) The intersection of the data graphs of (a) and (b). (d) The final result of this intersection.

the structural functions of a structural recursion call each other. Several beneficence of this representation will be explored. For example using this representation we will be able to characterize those data graphs on which a given structural recursion returns a non-empty output.

Formally, schema graphs are rooted, directed graphs, whose edges are labelled with formulas, which are conjugations of possibly negated predicates of the form: $a(x)$ ($a \in \Sigma$) or $\top(x)$. The former predicates will be called *constant predicates*. We fix an interpretation I_Σ throughout this dissertation, in which $a(x)$ becomes true iff $x = a$ and $\top(x)$ is satisfied by all constants in Σ. Note that the evaluation of p over this fixed interpretation can be accomplished in linear time. Here, the size of a formula is the number of its predicates. With $I_\Sigma \models p(a)$ we denote that a ($a \in \Sigma$) satisfies formula p.

Remark 2.3. Note that, if we blur the distinction between a and $a(x)$, then instances can be taken as schema graphs.

A schema graph is called *deterministic*, if for all pairs of neighbouring edges, with labels p_1 and p_2, there is not any $a \in \Sigma$ s.t. $I_\Sigma \models p_1(a) \wedge p_2(a)$. Furthermore, a schema graph is *semi-deterministic*, if either p_1 is the same as p_2 syntactically, or there is not any $a \in \Sigma$ s.t. $I_\Sigma \models p_1(a) \wedge p_2(a)$.

Simulation. For two schema graphs S_1, S_2 a mapping $\mu : V.S_1 \to V.S_2$ is called *simulation* [8], if the followings hold:

(i) if u is the root of S_1, then $\mu(u)$ is the root of S_2.

(ii) For all edge $e = (u_1, p_1, v_1) \in E.S_1$ and for all $u_2 \in \mu(u_1), v_2 \in \mu(v_1)$ $(u_2, p_2, v_2) \in E.S_2$ s.t. for all $a \in \Sigma$ to which $I_\Sigma \models p_1(a)$, $I_\Sigma \models p_2(a)$ also holds. As a shorthand notation with $\mu(e)$ we will denote the set of the aforementioned edges (u_2, p_2, v_2).

An example can be found in Figure 4.

In [8] simulations were used to establish a connection between data graphs and schema graphs and to impose restrictions on the structure of the data graphs. Namely, if there is a simulation μ from instance I to schema graph S, then an arbitrary node u of I may only have an outgoing a-labelled edge, if $\mu(u)$ also has an a-labelled outgoing edge. In Section 4 a generalization of simulation will be defined.

Equivalence. Two instances I_1, I_2 will be considered *equivalent*, if there is a simulation from I_1 to I_2 whose inverse is also a simulation [9]. These simulations are called *bisimulations*. In [25] it was shown that there is always a maximal bisimulation between two instances, what is more it can be found in time $O(m \log(m+n))$, where $m = |E.I_1| + |E.I_2|$ and $n = |V.I_1| + |V.I_2|$.

Intersection of schema graphs. The intersection of schema graphs were introduced in order to catch those restrictions that both operands of the intersection prescribe [8].

Formally, let S_1, S_2 be two schema graphs. The intersection of S_1 and S_2, $S_1 \sqcap S_2$ is defined as follows:

$$V.S_1 \sqcap S_2 := \{(u, v) \mid u \in V.S_1, v \in V.S_2\},$$
$$E.S_1 \sqcap S_2 := \{((u_1, u_2), p_1 \wedge p_2, (v_1, v_2)) \mid (u_1, p_1, v_1) \in E.S_1, (u_2, p_2, v_2) \in E.S_2\}.$$

The root of $S_1 \sqcap S_2$ is (u_0, v_0), where u_0, v_0 are the roots of S_1 and S_2 respectively. Edges with unsatisfiable formulas should be eliminated. Note that in our fixed interpretation the unsatisfiability of such formulas can be decided in linear time again. As for data graphs we will consider only the subgraph that is reachable from the root through directed edges. An example for the

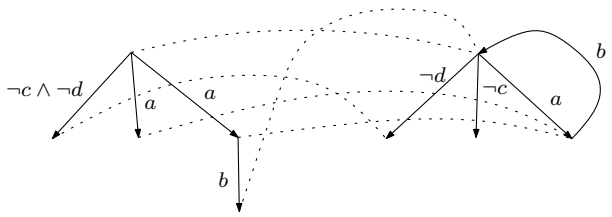

Figure 4: An example for simulation.

$f = (\{f_1, \ldots, f_n\}, \Sigma, F_I, \Gamma)$: structural function, $\gamma_i \in \Gamma$: a transformation rule, $L = \{f_1(t), \ldots, f_n(t)\}$: set of labels, $t \in \mathcal{T}^\Sigma$	
$\gamma_i ::=$	$(t_1 \cup t_2) = f_i(t_1) \cup f_i(t_2) \mid (\{\}) = \{\} \mid (\{a : t\}) = R \mid$ $(\{* : t\}) = R^d$
$R ::=$	$frst$, where $frst \in \mathcal{F}^\Delta(L)$
$R^d ::=$	$frst$, where $frst \in \mathcal{F}^{\Delta \cup \{*\}}(L)$

Figure 5: The syntax of transformation rules.

intersection of schema graphs can be found in Figure 3.(a)-(d). It is easy to see that the intersection is an associative and commutative operation. Sometimes for edges $e_i \in E.S_i$ $(i = 1, 2)$ we will denote (e_1, e_2) their pair in $E.S_1 \sqcap S_2$.

The *ancestor image* of $S_1 \sqcap S_2$ in S_1, here it is denoted G, is defined as follows:

$$V.G := \{u \mid u \in V.S_1, \exists v \in V.S_2 \text{ s.t. } (u, v) \in V.S_1 \sqcap S_2\},$$
$$E.G := \{e_1 \mid e_1 \in E.S_1, \exists e_2 \in E.S_2 \text{ s.t. } (e_1, e_2) \in E.S_1 \sqcap S_2\}.$$

For node $(u_1, u_2) \in V.S_1 \sqcap S_2$, edge $(e_1, e_2) \in E.S_1 \sqcap S_2$, u_1 and e_1 are also called *ancestor images in* S_1.

2.3 Structural recursions with natural semantics

This definition of structural recursions is based on the usual interpretation of structural recursions (cf. Section 1) [9], however, it is only applicable on trees.

A structural recursion is a tuple $f = (F, \Sigma, F_I, \Gamma)$, where $F = \{f_1, \ldots, f_n\}$ is the set of *structural functions*, $F_I = \{f_{i_1}, \ldots, f_{i_k}\}$ is the set of *initial* structural functions and Γ is the set of *transformation rules*. A structural recursion processes a data tree in a top-down manner. F_I gives those structural functions, which begin this processing. Note that in our previous works [22, 4, 19, 20] F_I consisted of a single structural function f_1. On the contrary, in [9] F_I was equal to F, i.e., every structural function was considered to be initial as well. Thus, F_I was not given explicitly there (besides the notion of transformation rules was also not introduced). Our approach here is between these two extremes and it is inspired by the definition of alternating tree automata.

Example 2.4. As an example consider $f = (\{f_1, f_2\}, \{f_1\}, \Gamma)$ that copies the subgraphs under the Ann edges. Γ consists of the following rules:

$$
\begin{array}{lll}
f_1\colon (t_1 \cup t_2) & = & f_1(t_1) \cup f_1(t_2) \\
f_1\colon (\{Ann\colon t\}) & = & \{Ann\colon f_2(t)\} \\
f_1\colon (\{*\colon t\}) & = & f_1(t) \\
f_1\colon (\{\}) & = & \{\}
\end{array}
\qquad
\begin{array}{lll}
f_2\colon (t_1 \cup t_2) & = & f_2(t_1) \cup f_2(t_2) \\
f_2\colon (\{*\colon t\}) & = & \{*\colon f_2(t)\} \\
f_2\colon (\{\}) & = & \{\}.
\end{array}
$$

In the sequel with $\gamma_{i,a}$ we denote the transformation rule for structural function f_i and singleton $\{a : t\}$ (or edge a). Similarly, $\gamma_{i,*}$ denotes the transformation rule for the default case.

In what follows, we briefly explain the meaning of the different transformation rules.

(i) If $t = t_1 \cup t_2$, then both t_1 and t_2 should be processed, and the union of the results should be taken $(f_i(t_1) \cup f_i(t_2))$. Note that if

$$t = \{a_1\colon t_1, \ldots, a_k\colon t_k\},$$

then it does not matter how we split t into branches, f_i will be called on every subtree $\{a_j\colon t_j\}$ $(1 \leq j \leq k)$.

(ii) If $t = \{\theta\colon \tilde{t}\}$, $\theta \in \Sigma \cup \{*\}$, then a forest in $\mathcal{F}^{\Delta}(L)$ or in $\mathcal{F}^{\Delta\cup\{*\}}(L)$ (the edge-labels are from set Δ or $\Delta \cup \{*\}$, besides the leaves may be labelled with elements of L) is constructed. Here, $*$ stands for the default case, $L = (f_1(t), \ldots, f_n(t))$. This forest is represented by its ssd-expression. For example in $\gamma_{1,Ann}$, $\{Ann\colon f_2(t)\}$ shows that as a result of calling f_1 on an Ann edge followed by subtree t an Ann edge should be constructed and its end node should be labelled with $f_2(t)$. On the other hand $\gamma_{2,*}$ says that in the default case an edge with the same label that is being processed should be constructed. Finally, $f_1(t)$ in $\gamma_{1,*}$ shows that only a node is to be constructed with label $f_1(t)$.

(iii) If $t = \{\}$, then an empty graph should be constructed.

Calls like $f_1(f_2(t))$, $f(\{a\colon \{b\colon t\}\})$ are not allowed. Note that these restrictions together with rule (iii) guarantee termination.

Remark 2.5. For constructors $\cup, \{\}$ the transformation rules are always the same for all structural functions, thus they will be omitted in the sequel.

The syntax of transformation rules can be found in Figure 5. γ_i denotes a transformation rule belonging to structural function f_i. As we have seen, when a tree is processed by a structural function, its ssd-expression is considered. In a transformation rule it is given how a structural function should work when it encounters a given constructor in the ssd-expression. When a structural function processes a singleton it creates a forest in $\mathcal{F}^{\Delta}(L)$. In [4, 20] only the construction of single edges were allowed. We introduced this extension in [22] in order to be able to simulate a core language of XSLT. However, this more general approach is not our invention since it has been already developed in [9].

For structural recursion $f = (\{f_1, \ldots, f_n\}, \Sigma, \{f_{i_1}, \ldots, f_{i_k}\}, \Gamma)$ the semantics of its structural functions is defined in Figure 6. The semantic function ε^{nat} assigns a forest in $\mathcal{F}^{\Delta}$ to each pair of a structural function and a tree in $\mathcal{T}^{\Sigma}$ (the edges are labelled with elements of Σ). In the superscript *nat* abbreviates natural. If the tree is the union of two trees, or it is the empty

$f = (\{f_1, \ldots, f_n\}, \Sigma, \{f_{i_1}, \ldots, f_{i_k}\}, \Gamma)$ is a structural recursion, $\varepsilon^{nat} : \{f_1, \ldots, f_n\} \times \mathcal{T}^{\Sigma} \to \mathcal{F}^{\Delta}$	
$\varepsilon^{nat}[\![f_i(t_1 \cup t_2)]\!] =$	$\varepsilon^{nat}[\![f_i(t_1)]\!] \cup \varepsilon^{nat}[\![f_i(t_2)]\!]$
$\varepsilon^{nat}[\![f_i(\{a : t\})]\!] =$	$\varepsilon^{nat}[\![frst]\!]$, $frst$ is constructed in $\gamma_{i,a}$ or if $\gamma_{i,a}$ is not given, then in $\gamma_{i,*}$. If neither $\gamma_{i,a}$, nor $\gamma_{i,*}$ are given, then $\varepsilon^{nat}[\![f_i(\{a : t\})]\!]$ is the empty graph.
$\varepsilon^{nat}[\![f_i(\{\})]\!] =$	$\{\}$

Figure 6: The natural semantics of structural functions.

graph, then the semantics is defined in a straightforward way. If this tree is a singleton$\{a : t\}$, then for structural function f_i first the appropriate transformation rule should be found. This is $\gamma_{i,a}$ or, if it is not given, then $\gamma_{i,*}$. If neither $\gamma_{i,a}$ nor $\gamma_{i,*}$ are given, then the semantic function assigns the empty graph to f_i and $\{a : t\}$. Otherwise, suppose that forest $frst$ is to be constructed on the right hand side of the preceding transformation rule. In Figure 6. $\varepsilon^{nat}[\![frst]\!]$ is a shorthand notation. If a leaf is labelled with $f_j(t)$, then it should be substituted with $\varepsilon^{nat}[\![f_j(t)]\!]$. More precisely, the roots of the trees of forest $\varepsilon^{nat}[\![f_j(t)]\!]$ should be contracted, and then this new root should be contracted with the leaf of $frst$ labelled with $f_j(t)$. Furthermore, if $frst$ is constructed in the default transformation rule ($\gamma_{i,*}$), then edge labels $*$ should be changed to edge label a. Consider the following example:

$$\varepsilon^{nat}[\![\{a : f_1(t)\} \cup \{b : f_2(t)\}]\!] = \{a : \varepsilon^{nat}[\![f_1(t)]\!]\} \cup \{b : \varepsilon^{nat}[\![f_2(t)]\!]\}.$$

Then, the result of calling structural recursion f on tree t, in notation $[\![f(t)]\!]_{nat}$, is defined to be

$$\varepsilon^{nat}[\![f_{i_1}(t)]\!] \oplus \ldots \oplus \varepsilon^{nat}[\![f_{i_k}(t)]\!].$$

Here $f_{i_1}, \ldots, f_{i_k}$ are the initial structural functions of f. This way of evaluation will be referred as the *natural semantics* of structural recursions.

Remark 2.6. Note that under this semantics the size of the output may grow too large. Namely, consider the following structural recursion f with transformation rules:

$f_1 : (\{a : t\}) = \{a : \{\{a : f_1(t)\} \cup \{a : f_1(t)\}\}\}$
$(\{* : t\}) = f_1(t).$

In Figure 8.(b)-(c) one can see how a simple path of a edges results an output of exponential size.

3 Structural recursions with operational semantics

Here, the semantics of structural recursions are given in a different way to be able to handle arbitrary data graphs. This definition was introduced in our first paper [4]. However, till now we have not proven the equivalence of the natural and operational semantics in any of our previous works. At the end of Section 3.1 this gap will be filled. Afterwards, in Section 3.2 *if-then-else* statements will be introduced, in whose conditions the Boolean combination of conditions *isempty* and *not-isempty* can be taken. This extension was introduced in [19] and to the best of our knowledge this is our own invention. In what follows structural recursions without conditions will be called *simple structural recursions*.

3.1 Simple structural recursions

Informally, in the operational semantics each cycle of a data graph will be traversed only once. In [9], when the semantics was extended to handle arbitrary data graphs the same consideration was in the background. Two different approaches were developed there, whose equivalence was also proven. Here, we elaborate a third way of defining the semantics which is based on the intersection of a graph representing the structural recursion and the data graph being processed. The equivalence of this alternative and the previously mentioned definitions could be shown, however, the proof is omitted owing to the lengthy and refined details.

Furthermore, since the syntax of transformation rules does not change here, it should be noted that in a transformation rule

$$f_i\colon\ (\{t_1 \cup t_2\}) \ =\ f_i(t_1) \cup f_i(t_2),$$

where f_i denotes a structural function, data graphs t_1 and t_2 may have common nodes and edges, in fact, it may happen that except from one edge from both data graphs, the rest of the edges coincides.

Operational graphs. First, *operational graphs* are defined [4], which will represent the "relationships" among structural functions. For structural recursion $f = (\{f_1, \ldots, f_n\}, \Sigma, F_I, \Gamma)$ we denote its operational graph with U_f. In U_f for each f_i we assign a node with label f_i $(1 \leq i \leq n)$. Besides, we take another node w_{end}. The edges of U_f are given with respect to the transformation rules given for singletons. As a warm-up consider transformation rule:

$$f_i : (\{a : t\}) = \{b : f_j(t)\}.$$

Here an $a(x)$ edge is added from f_i to f_j. The intended meaning is obvious, we represent that as a result of processing singleton $\{a : t\}$ f_i calls f_j.

Formally, for transformation rule

$$(\{\theta : t\}) = frst\ (frst \in \mathcal{F}^{\Delta \cup \{*\}}(L), \theta \in \Sigma \cup \{*\})$$

an (f_i, p, f_j) edge is added from node f_i to node f_j, if $f_j(t)$ is among the labels of the leaves of $frst$. Here, if $\theta = a$, then $p = a(x)$ $(a \in \Sigma)$. Otherwise, if $\theta = *$,

$$p = \neg a_1(x) \wedge \ldots \wedge \neg a_l(x),$$

where $a_1, \ldots, a_l$ are the symbols occurring in the non-default transformation rules for singletons of f_i. If there are no such transformation rules, then $p = \top(x)$. Remember that in our fixed interpretation $\top(x)$ is satisfied by all constants in Σ. Furthermore, if $f_j(t)$ appears more than once among the labels, then a separate edge (f_i, p, f_j) is added to represent each occurrence.

If $frst$ has no leaf labels, i.e., no structural function is called, then an (f_i, p, w_{end}) edge is added. If $frst$ is the empty graph, then no edges are added at all.

Nodes labelled with structural functions from F_I are defined to be the roots of U_f. Note that operational graphs are very similar to the graph representation of NDFSA-s.

Example 3.1. As an example the operational graph of structural recursion $f = (\{f_1, f_2, f_3\}, \{a, b, c, d\}, \{f_1, f_2\}, \Gamma)$ can be seen in Figure 7.(a).

$f_1 \colon (\{a : t\}) = \{b : \{\{a : f_2(t)\} \cup \{c : f_3(t)\}\}\}$ $\quad f_2 \colon (\{* : t\}) = \{* : f_2(t)\}$
$(\{* : t\}) = f_1(t)$

$f_3 \colon (\{* : t\}) = \{c : \{\}\}$

Here the nodes with labels f_1, f_2 are the roots of U_f.

In order to evaluate $f = (\{f_1, \ldots, f_n\}, \Sigma, F_I, \Gamma)$ on a given input I the intersection of U_f and I should be taken. Clearly, operational graphs may be considered as schema graphs. Additionally, as we have already mentioned, instances can be taken as schema graphs as well, hence the intersection defined for schema graphs can be applied here.

The intuition is straightforward. For instance, edge

$$((f_i, u), p(x) \wedge a(x), (f_j, v))$$

in $E.U_f \sqcap I$ represents that edge (u, a, v) is processed by f_i, and then f_j is called. In the sequel whenever we say that f_i *processes edge* (u, a, v), formally we mean that there is an edge (f_i, p, ϕ) in U_f s.t. $((f_i, u), p(x) \wedge a(x), (\phi, v))$ is in $E.U_f \sqcap I$ ($\phi \in \{f_1, \ldots, f_n, w_{end}\}$). Note that according to the definition of the intersection of schema graphs, if $p(x) \wedge a(x)$ is not satisfiable, then the aforementioned edge should be deleted. Moreover, this formula can be simply substituted with $a(x)$ or sloppily a. As an example for the intersection of the operational graph of Example 3.1. and an instance consider Figure 7.(a)-(c).

The size of an operational graph U_f, in notation $|U_f|$, is defined to be $\max\{|V.U_f|, |E.U_f|\}$.

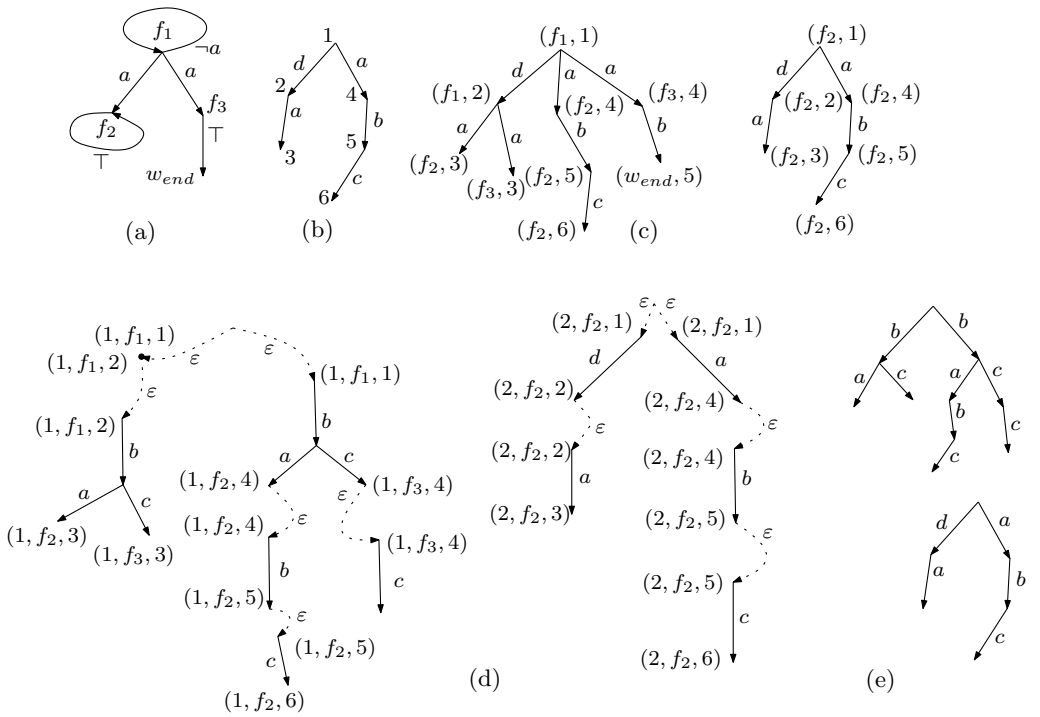

Figure 7: (a) The operational graph of the structural recursion of Example 3.1. (b) A data graph. (c) The intersection of the operational graph of (a) and data graph of (b). (d) Construction of the result. (e) The result after the elimination of ε edges.

Proposition 3.2. *Let f and g be two structural recursions. Then $U_f \sqcap U_g$ can be constructed in $|U_f||U_g|$ time.*

Proof. Consider node (f_i, g_j) in $U_f \sqcap U_g$, where f_i and g_j are structural functions of f and g respectively. In order to construct its outgoing edges the transformation rules of f_i and g_j should be coupled. First, we try to couple the non-default rules. If $\gamma^f_{i,a}$ matches $\gamma^g_{j,b}$, which means that a is the same as b, then the corresponding edge(s) should be drawn from (f_i, g_j). Clearly, neither of these transformation rules should be coupled with the default transformation rule of the other structural function, for the conjunction of the formulas of the representing edges is unsatisfiable.

On the other hand, if $\gamma^f_{i,a}$ does not match any of the non-default trans-

formation rules of g_j, then it should be coupled with the default rule, since $a(x) \wedge \neg b_1(x) \wedge \ldots \wedge \neg b_k(x)$ is surely satisfiable, because a is different from each b_l $(1 \leq l \leq k)$. Here $\neg b_1(x) \wedge \ldots \wedge \neg b_k(x)$ is the formula of the edge(s) representing the default transformation rule of g_j. Furthermore, the default rules should also be coupled, since a formula of the form

$$\neg a_1(x) \wedge \ldots \wedge \neg a_r(x) \wedge \neg b_1(x) \wedge \ldots \wedge \neg b_k(x)$$

is always satisfiable. Remember that we have assumed that Σ contains infinite number of elements. Here $\neg a_1(x) \wedge \ldots \wedge \neg a_r(x)$ is the formula of the representing edge(s) of the default transformation rule of f_i.

All in all, each transformation rule of f should be coupled with each transformation rule of g. Each such step requires constant time. The number of the transformation rules of a structural recursion is clearly less than equal than the size of its operational graph. ■

Corollary 3.3. *Let f be a structural recursion and I an instance, then $U_f \sqcap I$ can be constructed in time $|U_f||I|$, where $|I| = \max\{|V.I|, |E.I|\}$*

Construction of the result. As we have already seen, when an edge of the input is processed by a transformation rule, then a forest with possible leaf labels is to be constructed. In order to be able to construct the final result we should describe how these forests should be connected to each other. Consider Example 3.1. again. If an edge (u, a, v) is processed according to f_1, then tree

$$\hat{t} = \{b : \{\{a : \{f_2(t)\}\} \cup \{c : \{f_3(t)\}\}\}\},$$

should be constructed. However, in this case, the leaves are labelled with $(1, f_2, v)$ and $(1, f_3, v)$, i.e., the subgraph on which f_2 and f_3 are to be called is represented by its root v. The first number of the label, in this case 1, represents that $\hat{t}$ belongs to the result of f_1 called on the input. Remember that f has two initial structural functions f_1 and f_2. (However, when f_2 is called on an input first, then since f_2 only calls itself, f_1 will be never called during this construction.) Besides, the root of $\hat{t}$ should also be labelled with

$(1, f_1, u)$. The labels of the leaves indicate that the results of f_2 and f_3 applied on the subgraph under (u, a, v) should be connected to $\hat{t}$. Similarly, the label of the root shows that to which fragments of the result $\hat{t}$ should be connected. The connection of these fragments is accomplished through ε edges. As an example, consider Figure 7.(a)-(e). Note that here, when f_1 is applied to $(1, d, 2)$, only a node is constructed with two labels (cf. Figure 7.(d)).

Formally, consider structural recursion $f = (\{f_1, \ldots, f_n\}, \Sigma, F_I, \Gamma)$. Suppose that (u, a, v) is processed by f_i, and as a consequence forest $frst \in \mathcal{F}^{\Delta \cup \{*\}}(L)$ should be constructed. Suppose that this forest will be in that subgraph of the final result which belongs to initial structural function f_j. In the construction an instance of $frst$ should be taken, the $*$ labels should be substituted, with a, the root should be labelled with (j, f_i, u) and every leaf label $f_k(t)$ should be changed to (j, f_k, v). The result is called *basic forest*.

After the construction of the basic forests, the union of those of them should be taken, whose roots have the same label. In other words, for each such group a new node should be added to the result, and it should be connected to the aforementioned roots with ε edges. In the next step the labelled leaves of the forests should be connected with ε edges to the roots with the same label. Finally the ε edges should be eliminated. An example can be found in Figure 7.(d)-(e).

If for a given instance I, $U_f \sqcap I$ does not contain any edges, recall that in an intersection only those edges are considered which are reachable from a root, then $f(I)$ is defined to be the empty graph.

This semantics will be called *operational semantics*, and the result of calling structural recursion f on data graph I is denoted $[\![f(I)]\!]_{op}$.

Example 3.4. Consider now the structural recursion of Remark 2.6. Figure 8.(a)-(e) shows how operational semantics avoids outputs of exponential size by allowing the presence of undirected cycles. It is easy to see however that the result of the natural semantics applied for the same structural recursion and data tree in Remark 2.6. is equivalent to the output of the operational

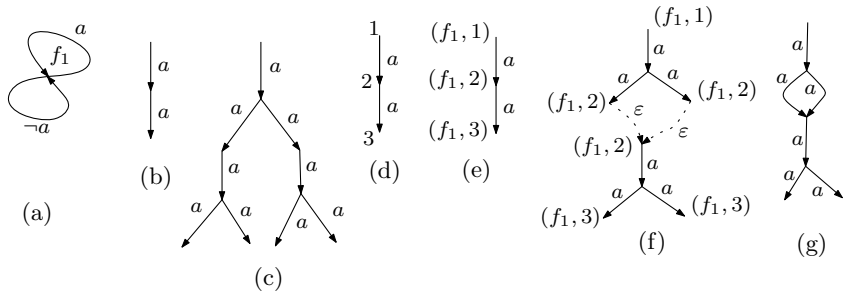

Figure 8: (a) The operational graph of the structural recursion of Example 3.4. (b) An input. (c) The result of the structural recursion of (a) called on the input of (b) according to natural semantics. (d) An input. (e) The intersection of the operational graph of (a) and the input of (d). (f) Construction and connection of the basic forests. (g) The final result.

semantics. The next proposition shows that this coincidence is not by chance.

Proposition 3.5. *Let $f = (\{f_1, \ldots, f_n\}, \Sigma, F_I, \Gamma)$ be a simple structural recursion . Then for all data tree $\hat{t}$, $[\![f(\hat{t})]\!]_{nat}$ is equivalent to $[\![f(\hat{t})]\!]_{op}$.*

The proof can be found in the Appendix (A 3.1.).

Number of steps. The size of a simple structural f, $|f|$ in notation, is defined to be $\max\{|V.U_f|, |E.U_f|\}$. In other words, since each call of a structural function in a transformation rule is represented by a separate edge in the operational graph, the size of a structural recursion is proportional to the number of its structural functions or the number of the structural function calls. Using the result of Corollary 3.3. it is easy to see that $f(I)$ can be constructed in $O(|f||I|)$ time.

3.2 Structural recursions with conditions

In this section we introduce conditions in structural recursions. In such a condition by examining whether a set of structural functions returns an empty or non-empty result on the subgraph under the edge being processed one can "look forward" and depending on the result it can be controlled which

$f = (\{f_1, \ldots, f_n\}, \Sigma, F_I, \Gamma)$: structural recursion, γ_i: transformation rule, C: condition, $L = \{f_1(t), \ldots, f_n(t)\}$, $t \in \mathcal{T}^\Sigma$, i., n.i. respectively stands for *isempty*, *not isempty*	
$\gamma_i ::=$	$(t_1 \cup t_2) = f_i(t_1) \cup f_i(t_2) \mid (\{\}) = \{\} \mid (\{a : t\}) = R \mid$ $(\{* : t\}) = R^d \mid$ $(\{a : t\}) =$ if C then R_1 else $R_2 \mid$ $(\{* : t\}) =$ if C then R_1^d; else R_2^d
$R ::=$	$frst$, where $frst \in \mathcal{F}^\Delta(L)$
$R^d ::=$	$frst$, where $frst \in \mathcal{F}^{\Delta \cup \{*\}}(L)$
$C ::=$	i.$(f_j(t))$ | n.i.$(f_j(t))$ | $(C_1 \wedge C_2)$ | $(C_1 \vee C_2)$

Figure 9: The syntax of transformation rules of structural recursions with conditions.

set of structural functions is to be applied in the next step. We suppose again that the results of structural recursions will be elements of $\mathcal{D}^\Delta$ (data graphs whose edge labels are from Δ). First, we define logical function

$$empty : \mathcal{D}^\Delta \to \{true, false\}.$$

It returns *true* iff its parameter is the empty graph. It will be applied on structural functions called on data graphs. In the conditions the Boolean combination of such "atomic" conditions will be taken. The negation of *empty* is *not isempty*. Obviously, by using tautologies

$$\neg(A \wedge B) \sim \neg A \vee \neg B \text{ and } \neg(A \vee B) \sim \neg A \wedge \neg B$$

each condition can be rewritten into an equivalent form without negation operators, in which only functions *empty* and *not isempty* may occur. In what follows we will always use this form of the conditions.

3.3 Syntax

The syntax of structural recursions with conditions can be found in Figure 9. As one can see, conditions can be given in transformation rules in *if-then-*

else statements. Both in the then- and else-branches, as in the case of simple structural recursions, forests in $\mathcal{F}^{\Delta}(L)$ or in the default case $\mathcal{F}^{\Delta\cup\{*\}}(L)$ can be constructed. A transformation rule may not contain any conditions. In this case its syntax is the same as it was given for simple structural recursions.

Example 3.6. Consider structural recursion $f = (\{f_1, f_2, f_3\}, \Sigma, \{f_1\}, \Gamma)$ as an example, which copies a subtree $\{a : t\}$ if the a edge has an *Ann* child. Here n.i. abbreviates not-isempty.

f_1: $(\{a : t\})$= if n.i.$(f_2(t))$ then $\{a : f_3(t)\}$
$(\{* : t\})$= $f_1(t)$

f_2: $(\{Ann : t\})$= $\{\psi : \{\}\}$
$(\{* : t\})$ = $\{\}$.

f_3: $(\{* : t\})$= $\{* : f_3(t)\}$

3.4 Semantics

Operational graphs. As in the previous section we represent the relationships among structural functions with operational graphs. A transformation rule without a condition will be represented in the same way as formerly, whereas for transformation rule

$$(\{a : t\}) = \text{ if n.i.}(f_j(t)) \text{ then } f_k(t) \text{ else } f_l(t),$$

an $a(x)$ edge will be added from f_i to f_j and two other $a(x)$ edges to f_k and f_l. These edges will be called *premise*, *then-* and *else-edge* respectively.

Formally, let $f = (\{f_1, \ldots, f_n\}, \Sigma, F_I, \Gamma)$ be a structural recursion with conditions. In U_f again for each f_i we assign a node with label f_i $(1 \leq i \leq n)$. Besides, we take another node w_{end}. The edges of U_f are given with respect to the transformation rules given for singletons. For a transformation rule without a condition the construction works exactly in the same way as for simple structural recursions. On the other hand, for transformation rule with a condition:

$$(\{\theta : t\}) = \text{ if } C \text{ then } frst_1 \text{ else } frst_2$$

if n.i.$(f_j(t))$ occurs in C, then edge (f_i, p, f_j) should be added. Such edges will be called *premises*. Here, again, if $\theta = a$, then $p = a(x)$. Otherwise, if $\theta = *$,

$$p = \neg a_1(x) \wedge \ldots \wedge \neg a_l(x),$$

where $a_1, \ldots, a_l$ are the symbols appearing in non-default transformation rules of f_i. If there are no such rows in the definition, then $p = \top(x)$.

Furthermore, edge (f_i, p, f_k) should also be added, if $f_k(t)$ occurs among the leaf labels of $frst_1$ or $frst_2$. In the first case it is called *then-edge*, in the second *else-edge*. The premise, then- and else-edges together will be called *conditional edges*. If leaf label $f_k(t)$ appears more than once in $frst_s$, a separate (f_i, p, f_k) edge should be added for each such occurrence $(s = 1, 2)$. Again edge (f_i, p, w_{end}) is added, if the leaves of $frst_s$ are not labelled.

Example 3.7. The operational graph of structural recursion

$$(\{f_1, \ldots, f_5\}, \Sigma, \{f_1\}, \Gamma)$$

can be found in Figure 10.(a). Its transformation rules are as follows:

$f_1\colon (\{a : t\}) =$ if i.$(f_2(t))$ then $\{f_5(t)\}$ else $\{b : \{\}\}$

$f_2\colon (\{b : t\}) =$ if n.i.$(f_3(t))$ then $\{f_4(t)\}$ else $\{b : \{\}\}$

$f_3\colon (\{a : t\}) =$ if n.i.$(f_2(t))$ then $\{a : \{\}\}$
$(\{c : t\}) = \{c : \{\}\}$

$f_4\colon (\{c : t\}) = \{c : \{\}\}$

$f_5\colon (\{* : t\}) = \{* : \{\}\}$

In what follows, for transformation rule $\gamma_{i,\vartheta}$, with $Form(\gamma_{i,\vartheta})$ we denote the formula of its condition (Boolean combination of i. and n.i. conditions) $(\vartheta \in \Sigma \cup \{*\})$. Furthermore $Pr(\gamma_{i,\vartheta}), Th(\gamma_{i,\vartheta}), El(\gamma_{i,\vartheta})$ will denote the premises, then- and else-edges belonging to $\gamma_{i,\vartheta}$. In what follows, for transformation rule $\gamma_{i,\vartheta}$, with $Form(\gamma_{i,\vartheta})$ we denote the formula of its condition (Boolean combination of i. and n.i. conditions) $(\vartheta \in \Sigma \cup \{*\})$. Furthermore

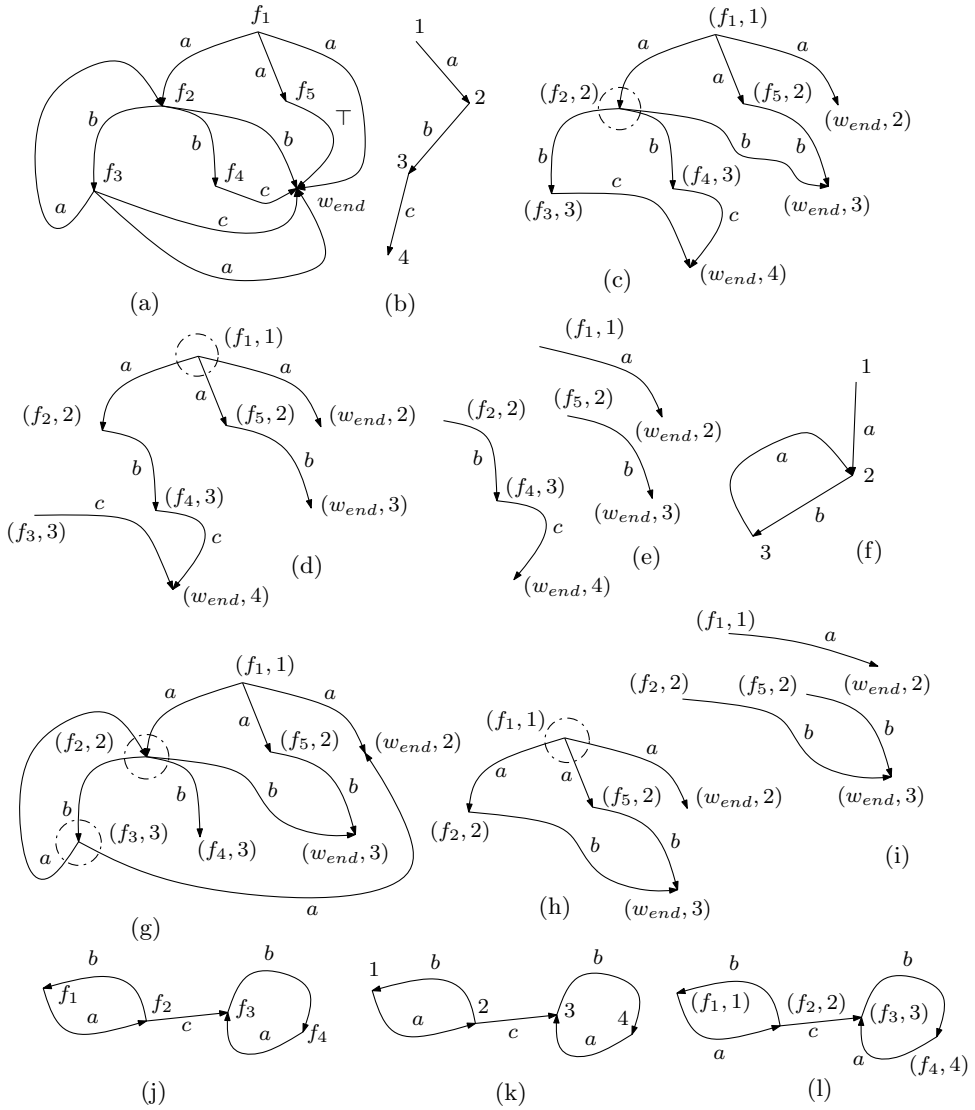

Figure 10: (a) The operational graph of the structural recursion of Example 3.7. (b) A tree input. (c) The intersection of the operational graph of (a) and the instance of (b). The dashed-dotted circle indicates that b.o.c., whose condition can be evaluated, and the appropriate conditional edges can be deleted. (d) The result of the first step of the condition evaluation algorithm. (e) The result of the second step. (f) An input with a cycle. (g) The intersection of the operational graph of (a) and the instance of (f). The dashed-dotted circles indicate b.o.c.-s whose premises form a cycle as it is described in the (iii) part of the condition evaluation algorithm. (h) The result of the first step of the condition evaluation algorithm. (i) The result of the second step. (j) An operational graph without then- and else-edges. (k) A data graph. (l) The intersection of the operational graph of (j) and the data graph of (k).

$Pr(\gamma_{i,\vartheta}), Th(\gamma_{i,\vartheta}), El(\gamma_{i,\vartheta})$ will denote the premises, then- and else-edges belonging to $\gamma_{i,\vartheta}$, while $Frst(\gamma_{i,\vartheta})$ denotes the forest constructed in this transformation rule.

Processing the input. Let f be a structural recursions and I an instance. In the construction of $f(I)$ in the first step the intersection of U_f and I should be taken. In this intersection an edge is a *premise*, (*then-*, *else-*, *constructor edge*), if its ancestor image in U_f is also a premise (then-, else-, constructor edge).

Evaluation of the conditions. In the next step, in $U_f \sqcap I$ we delete the premises and those then- and else-edges, whose condition is not satisfied. Note that for a condition

$$Cond = \text{ if } C \text{ then } frst_1 \text{ else } frst_2$$

and for an edge e_I of I, if (e_f, e_I) is in $E.U_f \sqcap I$ s.t. e_f is a conditional edge of $Cond$, then e_I is also coupled with the rest of the conditional edges of $Cond$. The set of these edges of $U_f \sqcap I$ will be referred as $Cond^{e_I}$. The conditional edges of $Cond$ may be paired with other edges of I therefore to be able to decide whether condition C is satisfied by the appropriate subgraph an instance of C should be assigned to each such edge pair sets. The instance assigned to e_I is denoted C^{e_I}.

(i) If from a premise $pr = (e_f, e_I)$ of $U_f \sqcap I$, where e_f, e_I are edges of U_f, I respectively, a constructor edge is reachable through a directed path not containing any conditional edges, this means that a non-empty output will be constructed. Thus, if pr belongs to a n.i. condition, it should be substituted with constant *true* in C^{e_I}. If C^{e_I} becomes *true*, then clearly, the then-branch should be executed, hence except for the then-edges, we delete all other conditional edges belonging to $Cond^{e_I}$. The rest of the remaining then-edges of $Cond^{e_I}$ are considered as normal (non-conditional) edges in the further steps of the algorithm. Note that the aforementioned path may traverse cycles.

If pr belongs to an i. condition, then it is substituted with constant $false$. If C^{e_I} becomes $false$, then the else-edges are kept and the rest of the condition edges belonging to $Cond^{e_I}$ are deleted. In Figure 10.(c) the n.i. condition of $((f_2, 2), b, (f_3, 3))$ is satisfied, while in the second step the i. condition of $((f_1, 1), a, (f_2, 2))$ is not satisfied (Figure 10.(d)). The final result of condition evaluation can be found in Figure 10.(e), here, since $(f_1, 1)$ is the root, only edge $((f_1, 1), a, (w_{end}, 2))$ should be considered. Note that in the example the operational graph of Example 3.7. has been used.

(ii) If there are neither constructor, nor conditional edges reachable from pr, then there is no further possibility for construction. Hence the appropriate i. condition is substituted with $true$, while the n.i. condition is substituted with $false$. Again, as in the previous case, if C^{e_I} becomes $false$, the else-edges, otherwise the then-edges are kept and considered as normal edges further on.

(iii) However, it may happen that steps (i), (ii) cannot be applied and there are still premises in the graph. Obviously, in this case from each premise at least another premise is reachable through a directed path without any conditional edge, consequently some of the premises must form cycles. Consider a maximal strongly connected subgraph of the intersect (the non-evaluated part of $U_f \sqcap I$) without any then- or else-edges from which no other premise is reachable through a directed path without conditional edges. Then the truth values of the premises – or rather the represented i. or n.i. condition – of this strongly connected subgraph mutually depend on each other. Since there is no escape from this infinite circle of dependencies, there does not remain any possibility for construction, hence the represented i. conditions should become $true$, whereas the n.i. conditions should become $false$. If as a result of this step a condition becomes $true$ or $false$, then its premises and its then- or else-edges should be deleted in the same way as in the previously described steps. The remaining else- or then-edges should be considered as normal edges in further steps of the algorithm. Note that here it was important that no other premise was reachable through a di-

rected path without conditional edges from the examined strongly connected component. An example can be found in Figure 10.(f)-(i). In Figure 10.(g) premises $((f_2, 2), a, (f_3, 3))$, $((f_3, 3), b, (f_2, 2))$ form a cycle, furthermore, neither step (i) not step (ii) can be applied. For a more complex example consider Figure 10.(j)-(l). Here, it is assumed that each edge of the operational graph is a premise. In the intersect (Figure 10.(l)) in the first step only cycle $((f_3, 3), a, (f_4, 4))$, $((f_4, 4), b, (f_3, 3))$ should be processed in the way given in this paragraph. Afterwards, the remaining then- or else-edges may offer an exit from the trap of cycle $((f_1, 1), a, (f_2, 2))$, $((f_2, 2), b, (f_1, 1))$.

To find the maximal strongly connected components Tarjan's algorithm [30] can be applied. In the algorithm the input graph is traversed in a depth-first search order. To each node a *depth search index* is assigned, which numbers the nodes consecutively in the order in which they are discovered. In addition, another value *lowlink* is given to each node that is equal to the smallest depth search index of a node reachable from the node in question through a directed path. Note that for each node the lowlink value is always less than or equal to the depth search index. At the end of the traverse all nodes of a strongly connected component have the same lowlink value. It can be shown that the algorithm requires $O(|V.G| + |E.G|)$ steps, where G denotes the input graph of the algorithm.

Now, in order to find those strongly connected components from which no other strongly connected component is reachable through a directed path consider an auxiliary graph G' whose nodes represent maximal strongly connected components of the former graph G. For two nodes u_1, u_2 of G' add an edge from u_1 to u_2, if in the represented maximal strongly connected components U_1, U_2 there are two nodes v_1, v_2 belonging to U_1, U_2 respectively s.t. there is an edge from v_1 to v_2. Then, clearly, G' is a directed acyclic graph. What is more, from the strongly connected components represented by the leaves no other strongly component is reachable. It is easy to see that G' can be constructed in $O(|E.G|)$ time.

Thus, when step (iii) is to be accomplished on the non-evaluated subgraph

of $U_f \sqcap I$ consider a copy of this graph, delete the then- and else-edges and apply the former algorithm for the remaining graph. For those premises that belong to a maximal strongly connected component from which no other strongly connected component can be reached through a directed path, as it has been already described, the represented i. conditions should become $true$, while the n.i. conditions should become $false$. Note that if such a maximal strongly connected component does not contain any premises, then its nodes should be deleted and the "leaf" maximal strongly connected components of the next level should be considered.

Construction of the result. The evaluation of the conditions stops, when there are no premises left. Since the resulting graph is without conditional edges, the construction of the final result can be accomplished exactly in the same way as for simple structural recursions.

Number of steps. Remember that $|f| = \max\{|V.U_f|, |E.U_f|\}$, $|I| = \max\{|V.I|, |E.I|\}$. By Corollary 3.3. $U_f \sqcap I$ can be constructed in $|f||I|$ time. The size of $U_f \sqcap I$ is $O(|f||I|)$. One iteration of the condition evaluation algorithm consists of considering the premises consecutively and checking whether step (i) or (ii) can be applied to them. If it is so, then the appropriate truth value should be substituted into the corresponding formula. If neither step (i) nor (ii) can be applied to any of the premises, then step (iii) should be accomplished. It is easy to see that in one iteration at least one premise is deleted. Since the number of premises is at most $|f||I|$, the number of iterations is also at most $|f||I|$.

Step (i) and (ii) can be accomplished by applying a breadth-first search starting from the premises, i.e., the first level consists of the premises, the second level consists of those edges that has a premise parent etc. During the traverse to each edge a list of premises is added from which it is reachable through a directed path. In the first step we assume that each premise is reachable from itself. Then for an arbitrary edge this list can be constructed by appending the lists of its non-conditional parent edges. At the end of a

traverse if a premise occurs in a list of a non-conditional constructor edge, then it fulfills the requirement of step (i). On the other hand, if it does not appear in any of the lists belonging to a constructor, then- or else-edge, then it satisfies the condition of step (ii). Clearly, this part of the algorithm can be accomplished in $O(|f||I|)$ steps all together. Besides, we have already mentioned that step (iii) also requires $O(|f||I|)$ time. Finally, the result can be constructed in $O(|f||I|)$ time. All in all the following statement has been proven.

Proposition 3.8. *For an arbitrary structural function f and instance I, $f(I)$ can be constructed in $O(|f|^2|I|^2)$ time.*

Remark 3.9. Note that the steps of the condition evaluation algorithm can be applied without any changes to operational graphs as well. This version of the algorithm will be used in the solution of the emptiness problem in Section 7.

Remark 3.10. Sometimes instead of structural recursion

$$f = (\{f_1, \ldots, f_n\}, \Sigma, F_I, \Gamma)$$

structural recursion $(\{f_1, \ldots, f_n\}, \Sigma, \{f_i\}, \Gamma)$ will be considered, which differs from f only in the set of initial structural functions. This is analogous to that case when instead of instance I instance I' is taken, whose node and edge set is the same as that of I only its root has been chosen to be different.

4 Basic notions and statements

In the rest of this dissertation structural recursions are treated as acceptors. This means that only the fact of a construction is of importance and the shape or other properties of the constructed graph are not taken into account. In Section 4.1 the usual static analytical questions are defined in the context of structural recursions. In [28] two types of containment were introduced for XPath expressions. Here, these definitions are reformulated and it will also

be proven that the related static analytical questions can be reduced to each other in polynomial time. This result has been already published in [21], however, the proof given here slightly improves the proof explained there. In Section 4.2 five different classes of structural recursions are introduced and their expressive power is compared. In Section 4.3 a mapping between operational graphs or intersections of operational graphs and instances will be introduced, which generalizes the simulation relation between schema graphs. By means of this relation we will be able to characterize the containment of structural recursions. Finally, in Section 4.4 an algorithm is developed, with which from an instance resulting a non-empty output for a structural recursion a tree can be constructed also returning a non-empty result for the same structural recursion. In the second part a refinement of the former algorithm is described, whose output data tree simulates the behaviour of the input data graph, when it is processed by a given structural recursion (which is also an element of the input).

Complete structural recursions. However, before going into the details in order to ease the explanations the definition of *complete* structural recursions is introduced, which was inspired by a similar notion of tree automata [12].

Let $f = (\{f_1, \ldots, f_n\}), \Sigma, F_I, \Gamma)$ be a structural recursion. The set of constants appearing in the transformation rules is defined as follows:

$$\Sigma_f := \{a \mid a \in \Sigma, \exists i \text{ s.t. } \gamma_{i,a} \in \Gamma, 1 \leq i \leq n\}.$$

Definition 4.1. *Keeping the preceding notation, we call* f complete*, if*

(i) each structural function has a transformation rule for the default case,

(ii) each structural function has a transformation rule for each constant in Σ_f*.*

The algorithm that makes a structural recursion complete can be found in Figure 11. First, if a structural recursion does not have a transformation

```
/* f = ({f_1, ..., f_n}), Σ, F_I, Γ) is a structural function,
   f^comp = ({f_1^comp, ..., f_n^comp}), Σ, F_I^comp, Γ^comp) is the result.  */

Complete_Structural_Recursion(f)
{
   f^comp := f;
   for i = 1...n
     if f_i^comp does not have a transformation rule for the default case,
       then define γ_(f_i^comp,*) to be {* : t} = {};
       add γ_(f_i^comp,*) to Γ^comp;
     for each constant a in Σ_f
       if f_i^comp does not have a transformation rule for a, then
         define γ_(f_i^comp,a) to be {a : t} = ϑ;
         (here ϑ denotes the right side of γ_(f_i^comp,*))
         add γ_(f_i^comp,a) to Γ^comp;
   return f^comp;
}
```

Figure 11: The algorithm of making a structural recursion complete.

rule for the default case, then it is added. All of these default transformation rules are without conditions, no structural function is called inside them and nothing is constructed. Afterwards the constants of Σ_f are considered one after the other. If the aforementioned structural function does not have a transformation rule for a constant, then it is created. These transformation rules work in the same way as the transformation rule for the default case. The subsequent proposition is trivial, therefore its proof is omitted.

Proposition 4.2. *For a structural recursion $f = (\{f_1, \ldots, f_n\}), \Sigma, F_I, \Gamma)$ in $SR(n.i., i., el)$ denote f^{comp} the result of the* `Complete_Structural_Recursion` *algorithm applied on f. Then, for an arbitrary instance I, $f(I)$ is not empty $\Leftrightarrow$ $f^{comp}(I)$ is not empty.*

4.1 Static analytical questions

Definition 4.3. *Let f be a structural recursion. The question of whether there is an instance I s.t. $f(I)$ is not empty is called the* problem of emptiness.

Definition 4.4. *For given structural recursions f_1, f_2 we say that f_1* contains *f_2, if for all instances I, from the non-emptiness of $f_2(I)$ the non-emptiness of $f_1(I)$ follows. The question that whether f_1 contains f_2 is called the* problem of containment.

Definition 4.5. *Let $f = (\{f_1, \ldots, f_n\}, \Sigma, F_I, \Gamma_f)$ be a structural recursion, I an instance and e an edge of I.*

(i) We say that f_i constructs on edge *$e = (u, a, v)$, if $((f_i, u), a, (\varphi, v))$ remains in $U_f \sqcap I$ after condition elimination, and in the appropriate transformation rule of f_i ($\gamma_{i,a}$ or $\gamma_{i,*}$) at least one edge is constructed ($\varphi \in \{f_1, \ldots, f_n, w_{end}\}$).*

(ii) Furthermore, f constructs on *e, if at least one of its structural function constructs on e.*

Definition 4.6. *For given structural recursions f_1, f_2 we say that f_1* strictly contains *f_2, if for all instances I, the set of those edges of I on which f_2 constructs is contained by the set of those edges of I on which f_1 constructs. The question that whether f_1 contains f_2 is called the* problem of strict containment.

Definition 4.7. *Two structural recursions f_1, f_2 are* (strictly) equivalent, *if f_1 (strictly) contains f_2 and f_2 (strictly) contains f_1.*

In the rest of this subsection the relationship between the two types of containment is clarified. It will be shown that the related static analytical problems can be reduced to each other in polynomial time.

Let f and g be structural recursions with initial structural functions $\{f_{i_1}, \ldots, f_{i_k}\}$ and $\{g_{j_1}, \ldots, g_{j_s}\}$. Suppose first that we have an algorithm which for two arbitrary structural recursions decides whether the first strictly contains the second. Then consider two additional structural functions f_0 and g_0 with the following transformation rules:

$f_0\colon (\{a : t\}) =$ if n.i.$(f_{i_1}(t)) \vee \ldots \vee$ n.i.$(f_{i_k}(t))$ then $\{\psi : \{\}\}$

$g_0\colon (\{a : t\}) =$ if n.i.$(g_{j_1}(t)) \vee \ldots \vee$ n.i.$(g_{j_s}(t))$ then $\{\psi : \{\}\}$.

Extend f and g with f_0 and g_0 respectively s.t. they are defined to be the only initial structural functions of the new structural recursions. Denote them f' and g'.

Proposition 4.8. *For arbitrary structural recursions f and g, f contains g $\Leftrightarrow$ f' strictly contains g'.*

Proof. Note that first if f' or g' constructs on edge e of an instance I, then e must be an outgoing a-labelled edge from the root followed by an instance I' s.t. f or g returns a non-empty output for I'. Assume now that f' strictly contains g', however, there is an instance $\hat{I}$ s.t. $g(\hat{I})$ in not empty, whereas $f(\hat{I})$ is empty. Then the previous observation shows that g' constructs on the

root edge of $\{a : \hat{I}\}$, while f' returns the empty graph for the same output, which is a contradiction. The proof of the reverse direction is similar. ■

Secondly, suppose that we have an algorithm which decides the containment problem for structural recursions. To show how this algorithm can be applied to solve the strict containment problem a transformed version of structural recursions will be used again. However, in this case the details are a bit trickier.

Let $f = (\{f_1, \ldots, f_n\}, \Sigma, F_I, \Gamma)$ be a structural recursion in $SR(n.i., i.)$. In the first step f should be extended with a copy of each of its structural functions, i.e., for each structural function f_i another structural function $\breve{f}_i$ is taken, whose transformation rules are the same as that of f_i's $(1 \leq i \leq n)$. Next, the transformation rules should be changed for both kinds of structural functions. In the transformation rules of the copies each occurrence of a structural function f_j should be substituted with $\breve{f}_j$ $(1 \leq j \leq n)$. For the original structural functions only the conditions should be changed in this way.

Let $\sharp$ be a symbol now that is different from all elements of Σ. We define a new structural function $f_\sharp$, whose single transformation rule is as follows:

$f_\sharp$: $(\{\sharp : t\})= \{\psi : \{\}\}$.

This transformation rule is intended to be the only transformation rule of the new structural recursion through which a construction can be done outside the conditions. To accomplish this modification consider those transformation rules of the original structural functions in which a non-empty graph is constructed. In the substitute of such a rule nothing will be constructed, however, if a structural function is called in the original transformation rule, then it should also be called in the new version as well as $f_\sharp$. Afterwards to each structural function different from $f_\sharp$ an extra transformation rule handling $\sharp$ edges should be added, which is without any condition, and in which nothing is to be constructed. Denote $\breve{f}$ this new structural recursion. As an example consider $f = (\{f_1, f_2\}, \Sigma, \{f_1\}, \Gamma)$ with transformation rules:

f_1: $(\{a:t\}) = \{a : f_1(t)\} \cup \{b : f_2(t)\}$ $\quad$ f_2: $(\{*:t\}) = \{* : f_1(t)\}$.
$(\{*:t\}) =$ if n.i.$(f_1(t))$ then $f_2(t)$

Then $\breve{f} = (\{f_1, f_2, \breve{f}_1, \breve{f}_2, f_\sharp\}, \Sigma \cup \{\sharp\}, \{f_1\}, \breve{\Gamma})$ and the transformation rules are as follows:

f_1: $(\{a:t\}) = f_1(t) \cup f_2(t) \cup f_\sharp(t)$ $\quad$ f_2: $(\{\sharp:t\}) = \{\}$
$(\{\sharp:t\}) = \{\}$ $\quad$ $(\{*:t\}) = f_1(t) \cup f_\sharp(t)$
$(\{*:t\}) =$ if n.i.$(\breve{f}_1(t))$ then $f_2(t)$

$\breve{f}_1$: $(\{a:t\}) = \{a : \breve{f}_1(t)\} \cup \{b : \breve{f}_2(t)\}$ $\quad$ $\breve{f}_2$: $(\{\sharp:t\}) = \{\}$
$(\{\sharp:t\}) = \{\}$ $\quad$ $(\{*:t\}) = \{* : \breve{f}_1(t)\}$.
$(\{*:t\}) =$ if n.i.$(\breve{f}_1(t))$ then $\breve{f}_2(t)$

To be able to describe the connection between f and $\breve{f}$ the original instances in $\mathcal{D}^\Sigma$ on which f works should be extended with $\sharp$ edges. For an arbitrary instance I in $\mathcal{D}^\Sigma$ denote $I_\sharp$ the instance which is constructed by adding an edge $\sharp$ to every node of I. The endnode of such a new edge should be always different from the rest of the nodes. For edge e of I, denote $e_\sharp$ the $\sharp$ edge, which has been added to the endnode of e. Note that I cannot contain any edges $\sharp$, since its edge labels are taken from Σ. In this subsection $I_\sharp$ will be called $\sharp$-*hedgehog*.

Lemma 4.9. *Let f be a structural recursion in $SR(n.i., i., el)$ and I an arbitrary instance. Then*

(i) f constructs on edge e of $I \Leftrightarrow \breve{f}$ constructs on $e_\sharp$ in $I_\sharp$.

(ii) Let e be an arbitrary edge $\sharp$ in $I_\sharp$. Consider an arbitrary set of edges $\sharp$ in $I_\sharp$, which does not contain e, delete these edges and denote $I'_\sharp$ the result. Then $\breve{f}$ constructs on e for $I_\sharp \Leftrightarrow \breve{f}$ constructs on e for $I'_\sharp$.

Proof. (i) Suppose that f constructs on e. This means that f has a transformation rule γ by which e is processed during the computation of $f(I)$. Then e is also processed by the correspondence of γ, when $\breve{f}$ is invoked on I. As a result $f_\sharp$ is called on $e_\sharp$ and finally an edge ψ is constructed. The proof of the reverse direction is similar.

(ii) Note that the presence of edges $\sharp$ do not affect the behaviour of the conditions in the intersection of $U_{\breve{f}}$ and an instance, since only the copy structural functions ($\breve{f}_i$) are reachable from a premise, which never call $f_\sharp$ and stop the computation returning the empty graph whenever an edge $\sharp$ is processed. Furthermore, if there is a path to a constructor edge in $U_{\breve{f}} \sqcap I_\sharp$, then its correspondence obviously exist in $U_{\breve{f}} \sqcap I'_\sharp$. All together this means that a condition is satisfied in $U_{\breve{f}} \sqcap I_\sharp$ iff its correspondence is also satisfied in $U_{\breve{f}} \sqcap I'_\sharp$.

Suppose now that $\breve{f}$ constructs on e. This means that there is a path from a root to an edge (e_f, e) in $U_{\breve{f}} \sqcap I_\sharp$ after condition elimination, where in the transformation rule which e_f represents a non-empty graph is constructed. We have just proven that the correspondence of this path does exist in $U_{\breve{f}} \sqcap I'_\sharp$, what is more its conditional edges are all kept during the condition elimination. In other words, $\breve{f}$ constructs on e for $I'_\sharp$. The reverse direction can be proven in the same way. ■

By means of the subsequent lemmas it will be shown that if for structural recursions f, g $\breve{f}$ does not strictly contain $\breve{g}$, then a counter example can be found even among the $\sharp$-hedgehogs.

Lemma 4.10. *Let I be an arbitrary instance in $\mathcal{D}^{\Sigma\cup\{\sharp\}}$ and f an arbitrary structural recursion in $SR(n.i., i., el)$. Delete all edges $\sharp$ of I and denote I' the resulting instance.*

(i) If for an edge e of I with label $\sharp$ its starting node is not reachable from the root of I', then $\breve{f}$ does not construct on e, when it is invoked on I.

(ii) Let e_1 be an edge of I s.t. in $I'_\sharp$ there is an edge $\sharp$ with the same starting node as e_1, where $I'_\sharp$ denotes the $\sharp$-hedgehog constructed from I'. Then

$\breve{f}$ *constructs on* e_1 *for* $I \Leftrightarrow \breve{f}$ *constructs on* e_2 *for* $I'_\sharp$.

Proof. (i) From the construction of $\breve{f}$ follows that whenever an edge $\sharp$ is processed, the computation stops on the corresponding branch. In this case this means that even if e is coupled with an edge e_f of U_f in $U_f \sqcap I$, (e_f, e) will not be reachable from any root of the intersection. Consequently, even if a non-empty graph is constructed through (e_f, e), it will not be reachable from any of the roots of the final result, hence by definition it does not belong to this result.

(ii) Suppose first that $\breve{f}$ constructs on e_1. Since $\breve{f}$ only constructs on edges $\sharp$, e_1 is a $\sharp$-labelled edge. Apart from e_1 leave all edges $\sharp$ of I and denote I_1 the result. By applying a similar reasoning to that of used in the proof of Lemma 4.9. it can be shown that $\breve{f}$ still constructs on e for I_1. Except from e_2 delete now all edges $\sharp$ of $I'_\sharp$ and denote I_2 the result. Note that the only difference between I_1 and I_2 is that the endnodes of e_1 and e_2 may not coincide, since the endnode of e_2 should be different from the rest of the nodes of I_2, while the endnode of e_1 may not fulfill this requirement. Nevertheless, this slight difference is without any particular importance, thus $\breve{f}$ will construct on e_2 for I_2. From Lemma 4.9. it follows that $\breve{f}$ also constructs on e_2 when it is called on $I'_\sharp$. The reverse direction can be proven in a similar way. ■

Lemma 4.11. *Let* f, g *be structural recursions in* $SR(n.i., i., el)$. *Then, if there is an instance* I_1 *in* $\mathcal{D}^{\Sigma \cup \{\sharp\}}$ *and edge* e_1 *of* I_1 *s.t.* $\breve{f}$ *does not construct on* e_1, *while* $\breve{g}$ *does, then there is a* $\sharp$*-hedgehog* I_2 *and an edge* e_2 *of* I_2 *for which* $\breve{f}$ *still does not construct on* e_2, *whereas* $\breve{g}$ *does.*

Proof. Statement (i) of Lemma 4.10. implies that since $\breve{g}$ constructs on e_1, if all edges $\sharp$ were deleted from I_1 the starting node of e_1 would be still reachable from the root of this new instance. Consequently, the $\sharp$-hedgehog of this instance contains an edge $\sharp$ with the same staring node as that of e_1. The statement follows now from statement (ii) of Lemma 4.10. ■

Proposition 4.12. *Let* f *and* g *be arbitrary structural recursions. Then* f *strictly contains* $g \Leftrightarrow \breve{f}$ *contains* $\breve{g}$.

Proof. From statement (i) of Lemma 4.9. it follows that f strictly contains g if and only if $\breve{f}$ strictly contains $\breve{g}$. In what follows we prove that $\breve{f}$ strictly contains $\breve{g}$ if and only if $\breve{f}$ contains $\breve{g}$, which obviously implies the proposition.

Suppose that $\breve{f}$ contains $\breve{g}$ and there is an instance I and edge e of I s.t. g constructs on e, whereas f does not. By Lemma 4.11. we may assume that I is a $\sharp$-hedgehog. Except from e leave all $\sharp$ edges of I and denote I' the resulting instance. Statement (ii) of Lemma 4.9. entails that g still constructs on e, when it is invoked on I', hence $f(I')$ cannot be empty. However, e is the only $\sharp$ edge of I', thus, since there is no other possibility, f must construct on e. This contradiction shows that for $\breve{f}$ and $\breve{g}$ from containment the strict containment follows. The reverse direction trivially holds. ■

4.2 Classes of structural recursions

With $SR(n.i., i., el)$ we denote the class of those structural recursions, in which n.i., i. conditions and else-branches may all appear in the transformation rules. Similarly, $SR(n.i., i.)$ denotes the class of those structural recursions, in which both n.i. and i. conditions are allowed to occur in the transformation rules, while none of them may contain any else-branch. Besides, notations $SR(n.i.)$ and $SR()$ will also be used, whose meaning should be self-describing in the light of the previous explanation.

Definition 4.13. *A structural recursion in $SR()$ is called* deterministic, *if*

(i) it has only one initial structural function,

(ii) at most one structural function is called in each transformation rule.

For structural recursion f denote $\mathcal{L}(f)$ the class of those instances in $\mathcal{D}^{\Sigma}$ (instances whose edges are labelled with elements of Σ) for which f returns a non-empty output. It is easy to see that for an arbitrary subclass $\mathcal{H}$ of $\mathcal{D}^{\Sigma}$ there does not necessarily exist a structural recursion f s.t. $\mathcal{L}(f)$ is equal to $\mathcal{H}$.

Proposition 4.14. *Let $\mathcal{H}$ be a set of two elements*

$$\{a : \{\}\} \text{ and } \{a : \{\}\} \cup \{a : \{\}\} \cup \{a : \{\}\}.$$

Then there is not any structural recursion f s.t. $\mathcal{L}(f)$ would be equal to $\mathcal{H}$.

Proof. Since $f(t_1 \cup t_2)$ is $f(t_1) \cup f(t_2)$, if for an arbitrary structural recursion f $\mathcal{L}(f)$ contains $\{a : \{\}\}$, then $\mathcal{L}(f)$ must contain $\{a : \{\}\} \cup \{a : \{\}\}$ as well. ■

In what follows we compare the expressive power of structural recursions belonging to the just defined classes, if we consider them as acceptors. It will turn out that for each structural recursion f in $SR()$ there is a deterministic structural recursion g s.t. $\mathcal{L}(f) = \mathcal{L}(g)$. Moreover, in this sense the expressive power of class $SR(n.i.)$ is stronger than the expressive power of class $SR()$, and $SR(n.i., i.)$ is more expressive than $SR(n.i.)$. Finally, $SR(n.i., i.el)$ and $SR(n.i., i.)$ will be proven to be equally expressive.

Deterministic structural recursions and $SR()$. First, we show that how the functioning of several structural functions can be simulated by a single structural function. For structural functions $f_{i_1}, \ldots, f_{i_k}$, we denote with $f_{i_1,\ldots,i_k}$ this representing structural function. Let $f = (\{f_1, \ldots, f_n\}, \Sigma, F_I, \Gamma)$ be a structural recursion in $SR()$, by Proposition 4.2. we may assume that f is complete. If it is not so, then in the first step f should be made complete. Let ϑ be a symbol in $\Sigma \cup \{*\}$ s.t. there is a structural function f_i of f that has a transformation rule for ϑ. Note that in this case since f is complete all structural functions of f have a transformation rule for ϑ. Consider a set of these structural functions $f_{i_1}, \ldots, f_{i_k}$ and suppose that in the transformation rules structural functions $f_{j_1}, \ldots, f_{j_s}$ are called (f_{j_o} is not necessarily called in every transformation rule) ($1 \leq o \leq s$). Then, if a non-empty graph is constructed in at least one of the aforementioned transformation rules, then in $\gamma_{i_1,\ldots,i_k,\vartheta}$ (the transformation rule of $f_{i_1,\ldots,i_k}$ for ϑ) edge $\{\psi : f_{j_1,\ldots,j_s}(t)\}$ should be constructed. Otherwise, if nothing is constructed in these transformation rules, then simply structural function $f_{j_1,\ldots,j_s}$ should be called.

Now, in order to simulate f with a deterministic structural recursion construct each structural function $f_{j_1,\ldots,j_s}$, where $\{j_1,\ldots,j_s\} \subseteq \{1,\ldots,n\}$. If the initial structural functions of f are $f_{i_1},\ldots,f_{i_k}$, then define $f_{i_1,\ldots,i_k}$ to be the initial structural function in the simulation. Denote $Det(f)$ the resulting structural recursion. Clearly, $Det(f)$ is deterministic, on the other hand its size is exponential in the size of f.

Example 4.15. Consider structural recursion $f = (\{f_1, f_2\}, \Sigma, \{f_1, f_2\}, \Gamma)$, whose transformation rules are as follows:

$$\begin{array}{ll} f_1\colon (\{a:t\}) = \{b:f_1(t)\} \cup \{c:f_2(t)\} & f_2\colon (\{b:t\}) = f_1(t) \\ \quad (\{*:t\}) = f_1(t) & \quad (\{*:t\}) = \{*:f_2(t)\} \end{array}$$

Then $Det(f) = (\{f_1, f_2, f_{1,2}\}, \Sigma, \{f_{1,2}\}, \Gamma)$, where the transformation rules are the following:

$$\begin{array}{lll} f_1\colon & (\{a:t\}) = \{\psi:f_{1,2}(t)\} & f_2\colon (\{a:t\}) = \{\psi:f_2(t)\} \\ & (\{b:t\}) = f_1(t) & \quad (\{b:t\}) = f_1(t) \\ & (\{*:t\}) = f_1(t) & \quad (\{*:t\}) = \{\psi:f_2(t)\} \end{array}$$

$$\begin{array}{l} f_{1,2}\colon (\{a:t\}) = \{\psi:f_{1,2}(t)\} \\ \quad (\{b:t\}) = f_1(t) \\ \quad (\{*:t\}) = \{\psi:f_{1,2}(t)\}. \end{array}$$

Proposition 4.16. *Let $f = (\{f_1,\ldots,f_n\}, \Sigma, F_I, \Gamma)$ be an arbitrary structural recursion in $SR()$. Then $\mathcal{L}(f) = \mathcal{L}(Det(f))$.*

The proof of the proposition can be found in the Appendix (A.4.2).

Remark 4.17. Note that for structural recursions f and g, $\mathcal{L}(f) = \mathcal{L}(g)$ if and only if f is equivalent to g.

$SR(n.i.)$ **and** *$SR()$*.

Lemma 4.18. *Let f be an arbitrary structural recursion in $SR()$ and I an instance. Then $f(I)$ is not empty $\Leftrightarrow$ there is a path pa of I, whose root is the same as that of I s.t. $f(pa)$ is also not empty.*

Proof. Suppose first that $f(I)$ is not empty. Then there is a path pa' from the root of $U_f \sqcap I$ to a constructor edge. Let pa be the ancestor image of this path in I. Obviously, $U_f \sqcap pa$ still contains pa' as a pregraph, hence $f(pa)$ is non-empty. The reverse direction is trivial. ■

Proposition 4.19. *There is a structural recursion f in $SR(n.i.)$ for which there is not any structural recursion g in $SR()$ s.t. $\mathcal{L}(f) = \mathcal{L}(g)$.*

Proof. Consider structural recursion $f = (\{f_1, f_2, f_3\}, \Sigma, \{f_1\}, \Gamma)$, whose transformation rules are as follows:

$f_1 \colon (\{a : t\}) =$ if n.i.$(f_2(t))$ then $f_3(t)$ $\quad f_2 \colon (\{b : t\}) = \{\psi : \{\}\}$

$f_3 \colon (\{c : t\}) = \{\psi : \{\}\}$.

It is not difficult to see that if I is in $\mathcal{L}(f)$, then its root should have an outgoing edge a with children edges b and c. In other words $\mathcal{L}(f)$ does not contain any paths. On the other hand, Lemma 4.18. implies that for every structural recursion g in $SR()$, if $\mathcal{L}(g)$ is not empty, then it should contain a path. ■

$SR(n.i., i.)$ **and** *$SR(n.i.)$* In Proposition 4.27. we will show that non-emptiness is a monotonous property for structural recursions without *isempty* conditions, i.e., if an instance I_1 results a non-empty output for a structural recursion in $SR(n.i.)$, then every instance I_2 containing I_1 as a pregraph results a non-empty output.

On the other hand, for structural recursion $f = (\{f_1, f_2\}, \Sigma, \{f_1\}, \Gamma)$ with transformation rules:

f_1: $(\{a : t\})$= if i.$(f_2(t))$ then $\{\psi : \{\}\}$ $\quad f_2$: $(\{* : t\})$= $\{\psi : \{\}\}$

$\{a : \{\}\}$ results an edge ψ, whereas $\{a : \{b : \{\}\}\}$ results the empty graph. This example proves the following statement:

Proposition 4.20. *There is a structural recursion f in $SR(n.i., i.)$ for which there is not any structural recursion g in $SR(n.i.)$ s.t. $\mathcal{L}(f) = \mathcal{L}(g)$.*

Finally, in Corollary 5.11. in Section 5.1. it will be proven that for each structural recursion in $SR(n.i., i., el)$ a structural recursion in $SR(n.i., i.)$ can be constructed which returns a non-empty output for the same set of instances.

4.3 Operational homomorphism

In this subsection a mapping between operational graphs or intersections of operational graphs and instances will be introduced, which generalizes the simulation relation between schema graphs. Afterwards, by means of this relation a necessary and sufficient condition will be formulated for the containment of structural recursions.

First, an auxiliary notion is introduced which for two propositional formulae establish a connection between two subsets of their variables that determine the truth value of the formulae in a somewhat similar manner. Let F be a propositional formula with variables $X_1, \ldots, X_n$ and $\hat{X} \subseteq \{X_1, \ldots, X_n\}$.

$Int(F, \hat{X})$:= $\{I \mid I$ is an interpretation over $\{X_1, \ldots, X_n\}$ s.t. $I(F)$ can be calculated by using solely the truth values of $X_i, X_i \in \hat{X}\}$.

Here, $I(F)$ denotes the truth value of F for interpretation I.

Definition 4.21. *Let F_1, F_2 be two propositional formulae with variables $X_1, \ldots, X_n$ and $Y_1, \ldots, Y_m$. A mapping $\varphi : \hat{X} \to \hat{Y}$ $(\hat{X} \subseteq \{X_1, \ldots X_n\}, \hat{Y} \subseteq \{Y_1, \ldots, Y_m\})$ is called* truth preserving with respect to $F_1, F_2, \hat{X}, \hat{Y}$, *if for all interpretations I_1 in $Int(F_1, \hat{X})$ and interpretation I_2 over $Y_1, \ldots, Y_m$, where*

for $Y_j \in \hat{Y}$ $I_2(Y_j)$ is true iff there is at least one $X_i \in \hat{X}$ s.t. $Y_j \in \rho(X_i)$ and $I_1(X_i)$ is also true, $I_2 \in Int(F_2, \hat{Y})$ and $I_1(F_1) = I_2(F_2)$ $(1 \leq i \leq n, 1 \leq j \leq m)$.

As an example consider $F_1 = X_1 \vee X_2$, $F_2 = (Y_1 \wedge Y_2) \vee Y_3 \vee Y_4$. Here φ, $\varphi(X_1) = \{Y_1, Y_2\}$, $\varphi(X_2) = Y_3$, is truth preserving with respect to $F_1, F_2, \{X_1, X_2\}$, $\{Y_1, Y_2, Y_3\}$.

Definition 4.22. *For two graphs G_1 and G_2 that are either operational graphs or intersections of an operational graph and an instance, a mapping $\rho : V.G_1 \to V.G_2$ is called* operational homomorphism, *if the followings hold:*

(i) $\rho(u)$ is a root of G_2, if u is a root of G_1.

(ii) For all edge $e = (u_1, p_1, v_1) \in E.G_1$, $\rho(e) = (u_2, p_2, v_2) \in E.G_2$ s.t. for all $a \in \Sigma$, if $p_1(a)$ is true, then $p_2(a)$ is also true. Here, $u_2 \in \rho(u_1)$, $v_2 \in \rho(v_1)$.

(iii) If $e \in E.G_1$ is a premise, then-, else- edge belonging to an i. (n.i.) condition, then $\rho(e)$ is also premise, then-, else-edge in G_2 belonging to an i. (n.i.) condition. What is more, if e is a constructor edge, then $\rho(e)$ is also a constructor edge.

(iv) Suppose that for transformation rule γ_1, $Pr(\gamma_1) = \{pr_1, \ldots pr_n\}$. Denote γ_2 a transformation rule to which an element of $\rho(pr_1)$ belongs. Then relation φ, $\varphi(pr_i) = \rho(pr_i)$ $1 \leq i \leq n$, is truth preserving with respect to $Form(\gamma_1)$, $Form(\gamma_2)$, $\{pr_1, \ldots pr_n\}$, $\{\rho(pr_1), \ldots, \rho(pr_n)\}$.

Note that in the definition with a slight abuse of notions premises are treated as propositional variables. As a notational convenience for path $pa = (u_1, a_1, v_1) \ldots (u_m, a_m, v_m)$,

$$(\rho(u_1), a_1, \rho(v_1)) \ldots (\rho(u_m), a_m, \rho(v_m))$$

will be denoted by $\rho(pa)$, where ρ is an operational homomorphism.

The following lemma establishes the connection between simulation and operational homomorphism.

Lemma 4.23. *Let f be a structural recursion in $SR(n.i., i., el)$ and I_1, I_2 instances s.t. there exists a simulation μ from I_1 to I_2. Then there also exists an operational homomorphism ρ from $U_f \sqcap I_1$ to $U_f \sqcap I_2$.*

Proof. ρ is defined as follows: for a given node u_1 of I_1 and for all nodes u_2 of I_2 where $\mu(u_1) = u_2$

$$\rho((u_f, u_1)) := (u_f, u_2), \text{ where } (u_f, u_1) \in V.U_f \sqcap I_1.$$

We have to show that conditions (i)-(iv) of Definition 4.22. apply to ρ.

(i) If $(u_f, u_1) \in V.U_f \sqcap I_1$ is a root, then u_f, u_1, u_2 are also roots of U_f, I_1, I_2 respectively, where $u_2 \in \mu(u_1)$. Consequently, (u_f, u_2) is a root of $U_f \sqcap I_2$.

(ii)-(iii) For edge $e = ((u_f, u_1), p \wedge a, (v_f, v_1)) \in E.U_f \sqcap I_1$,

$$e_f = (u_f, p, v_f),\ e_2 = (u_2, a, v_2),\ u_2 \in \mu(u_1), v_2 \in \mu(v_1)$$

are also edges in U_f and I_2. With a straightforward induction it can be shown that $(e_f, e_2) \in \rho(e)$ is an edge in $U_f \sqcap I_2$. What is more, if e is a premise (then-, else-, constructor edge), then e_f and as a direct consequence (e_f, e_2) is also a premise (then-, else-, constructor edge). In addition, if e is a conditional edge belonging to an i. (n.i.) condition, then $\rho(e)$ also belongs to an i. (n.i.) condition.

(iv) Suppose that edge $e = (e_f, e_1)$ in $U_f \sqcap I_1$ is a premise belonging to transformation rule γ. Then with a similar reasoning to that of the previous case it can be shown that $(e_f, e_2) \in \rho(e)$ also belongs to γ, hence this condition trivially holds. ■

Next we show how the containment of structural recursions in $SR(n.i.)$ can be characterized by operational homomorphism.

Lemma 4.24. *For structural recursions f_1, f_2 in $SR(n.i.)$, instances I_1, I_2, if there is an operational homomorphism ρ from $V.U_{f_1} \sqcap I_1$ to $V.U_{f_2} \sqcap I_2$, then for an arbitrary then-edge $th \in E.U_{f_1} \sqcap I_1$, if th is kept in the n^{th} step of condition evaluation, then $\rho(th)$ is also kept in the m^{th} step, where $m \leq n$.*

Proof. Without loss of generality we may suppose that ρ is a function. We prove the statement by using induction on n. Suppose first that $n = 1$. Denote γ_1, γ_2 the transformation rules which th and $\rho(th)$ respectively belongs to. Since th is kept, $Form(\gamma_1)$ is satisfied, i.e., there is a subset $pr_1, \dots pr_k$ of $Pr(\gamma_1)$ s.t. there is a path from pr_i to a constructor edge that does not contain any conditional edges ($1 \leq i \leq k$). Using induction on the length of pa_i by means of conditions (ii)-(iii) of Definition 4.22. it can be proven that $\rho(pa_i)$ is also a path from premise $\rho(pr_i)$ to a constructor edge without conditional edges, i.e., $\rho(pr_i)$ also becomes $true$. By condition (iv) of Definition 4.22. a truth preserving mapping can be given from pr_i to $\rho(pr_i)$. If I_1, I_2 are interpretations over $Pr(\gamma_1)$ and $Pr(\gamma_2)$ respectively that assign a $true$ value to each $pr_i, \rho(pr_i)$, then clearly the conditions of Definition 4.21. are all satisfied, hence, since $Form(\gamma_1)$ becomes $true$ over I_1, $Form(\gamma_2)$ also becomes $true$ over I_2, consequently $\rho(th)$ is also be kept.

Note that there may be premises in $U_{f_2} \sqcap I_2$ that become $true$ and they have no ancestor image according to ρ. Consequently, there may be then-edges that have been kept after the first step in $U_{f_2} \sqcap I_2$, however, at this stage of the algorithm it cannot be decided yet, if their ancestor images according to ρ should be kept or not. Our lemma states that these then-edges will also be kept in a later step.

In order to prove the general case we formulate an additional statement. We slightly modify the algorithm of condition evaluation in case of $U_{f_2} \sqcap I_2$ s.t. in the k^{th} step we do not delete those premises and then-edges in $E.U_{f_2} \sqcap I_2$ which should be deleted, but it cannot be decided whether their ancestor images according to ρ should be deleted or not. (†) Then for G_1, G_2, where G_i denotes $U_{f_i} \sqcap I_i$, after the k^{th} step of this modified condition evaluation, an operational homomorphism can be defined from $V.G_1$ to $V.G_2$ ($i = 1, 2$). Indeed, the definition of this relation is quite simple, namely ρ should be restricted to those nodes of G_1 that are still reachable from a root. It is easy to see that conditions (i)-(iv) of Definition 4.22. remain true for this restriction.

Suppose now that the statement of the lemma holds for $n = k$ and suppose that $n = k + 1$ ($k \geq 1$). Since there is an operational homomorphism from $V.G_1$ to $V.G_2$, where G_1, G_2 denote the graphs of statement (†) after the k^{th} step of condition evaluation, the proof of this case can be given exactly in the same way as in the base case. This concludes the proof. ■

Theorem 4.25. *For structural recursions f_1, f_2 in $SR(n.i.)$, instances I_1, I_2, if there is an operational homomorphism ρ from $V.U_{f_1} \sqcap I_1$ to $V.U_{f_2} \sqcap I_2$, then from the non-emptiness of $f_1(I_1)$ the non-emptiness of $f_2(I_2)$ follows.*

Proof. Since $f_1(I)$ is non-empty, we know that there is a path pa from a root to a constructor edge in $V.U_{f_1} \sqcap I_1$ after condition evaluation. It is easy to show that before condition evaluation $\rho(pa)$ is also a path from a root to a constructor edge in $V.U_{f_2} \sqcap I_2$. Furthermore, if $th \in E.pa$ was a then-edge before condition evaluation, then $\rho(th)$ was also a then-edge in $\rho(pa)$. Since th has been kept, from Lemma 4.24. it follows that $\rho(th)$ has also been kept. Hence, none of the edges of $\rho(pa)$ has been deleted, which means that $f_2(I)$ is also non-empty. ■

Corollary 4.26. *Let f_1, f_2 be two structural recursions in $SR(n.i.)$. If there is an operational homomorphism ρ from $V.U_{f_1}$ to $V.U_{f_2}$, then f_2 contains f_1.*

Proof. Let I be an arbitrary instance s.t. $f_1(I)$ is not empty. In order to prove the statement, we should show that $f_2(I)$ is also non-empty. We define $\hat{\rho} : V.U_{f_1} \sqcap I \to V.U_{f_2} \sqcap I$ as follows. Let u_1, u_I be arbitrary nodes of U_{f_1}, I respectively s.t. (u_1, u_I) is in $V.U_{f_1} \sqcap I$. For all node u_2 in U_{f_2}, where $\rho(u_1) = u_2$, $\hat{\rho}((u_1, u_I)) := (u_2, u_I)$. Obviously, $\hat{\rho}$ fulfills the requirements of conditions (i)-(iii) in Definition 4.22. Since premise $(e_{pr}, e_I) \in E.U_{f_i} \sqcap I$ belongs to the same transformation rule as e_{pr}, condition (iv) also holds straightforwardly for $\hat{\rho}$ ($i = 1, 2$). This means that $\hat{\rho}$ is an operational homomorphism, hence from Theorem 4.25. the non-emptiness of $f_2(I)$ follows. ■

Using Theorem 4.25. we prove that non-emptiness is a monotonous property for structural recursions in $SR(n.i.)$.

Proposition 4.27. *For a structural recursion f in $SR(n.i.)$ and two instances I_1, I_2, where I_1 is a pregraph of I_2, from the non-emptiness of $f(I_1)$ the non-emptiness of $f(I_2)$ follows.*

Proof. Trivially, since I_1 is a pregraph of I_2 a simulation can be given from I_1 to I_2. Hence, by Lemma 4.23. an operational homomorphism can be defined from $U_f \sqcap I_1$ to $U_f \sqcap I_2$. The statement of the proposition follows now from Theorem 4.25. ■

The subsequent two examples highlight that Corollary 4.26. and Proposition 4.27. do not hold for structural recursions with else branches in general. A counter example for Theorem 4.25. could be given in a similar manner. It can also be shown that neither of the aforementioned statements hold for structural recursions in $SR(n.i., i.)$.

Example 4.28. The transformation rules of $f = (\{f_1, f_2, f_3\}, \Sigma, \{f_1\}, \Gamma_f)$ are as follows:

$f_1 \colon (\{a : t\}) =$ if n.i.$(f_2(t))$ then $f_3(t)$ else $\{a : f_1(t)\}$ $\qquad$ $f_3 \colon (\{* : t\}) = f_3(t)$

$f_2 \colon (\{* : t\}) = \{\}$.

For $g = (\{g_1, g_2, g_3\}, \Sigma, \{g_1\}, \Gamma_g)$ the transformation rules for g_1, g_3 are the same as for f_1, f_3 (only structural functions f_i are changed to g_i $(i = 1, 3)$). Structural function g_2 has a single transformation rule:

$g_2 \colon (\{* : t\}) = \{\psi : \{\}\}$.

The operational graph of f can be found in Figure 12.(a). It is easy to see that aside from the node labels the operational graph of g has the same structure. Clearly, $\rho : V.U_f \to V.U_g$ is an operational homomorphism, where

$$\rho(f_1) = g_1,\ \rho(f_2) = g_2,\ \rho(f_3) = g_3,\ \rho(w^f_{end}) = w^g_{end}.$$

Obviously, for instance $I = \{a : \{b : \{\}\}\}$, $f(I)$ is not empty, while $g(I)$ is empty.

Example 4.29. Here, g is the same as in the previous example.

$$I_1 := \{a : \{\}\}, I_2 := \{a : \{b : \{\}\}\}.$$

Straightforwardly, I_1 is a pregraph of I_2, yet, $g(I_1)$ is non-empty, while $g(I_2)$ is empty.

Next we show how the previous reasonings should be extended to be able to characterize the containment of structural recursions $SR(n.i., i., el)$. The proof of the following lemma is very similar to that of Lemma 4.24., therefore it is placed in the Appendix (A 4.3.).

Lemma 4.30. *For structural recursions f_1, f_2 in $SR(n.i., i., el)$, instances I_1, I_2, if there is a surjective operational homomorphism ρ from $V.U_{f_1} \sqcap I_1$ to $V.U_{f_2} \sqcap I_2$ s.t. the inverse of ρ, ρ^{-1}, is also an operational homomorphism, then for an arbitrary then- or else-edge $e \in E.U_{f_1} \sqcap I_1$, if e is deleted in the n^{th} step of condition evaluation, then $\rho(e)$ is also deleted in this step.*

Again, the subsequent theorem and its corollary can be proven in similar manner as their counterparts Theorem 4.25. and Corollary 4.26.

Theorem 4.31. *For structural recursions f_1, f_2 in $SR(n.i., i., el)$, instances I_1, I_2, if there is a surjective operational homomorphism ρ from $V.U_{f_1} \sqcap I_1$ to $V.U_{f_2} \sqcap I_2$ s.t. the its inverse is also an operational homomorphism, then from the non-emptiness of $f_1(I)$ the non-emptiness of $f_2(I)$ follows and vice versa.*

Corollary 4.32. *Let f_1, f_2 be two structural recursions in $SR(n.i., i., el)$. If there is a surjective operational homomorphism ρ from $V.U_{f_1}$ to $V.U_{f_2}$ s.t. its inverse is also an operational homomorphism, then f_1 is equivalent to f_2.*

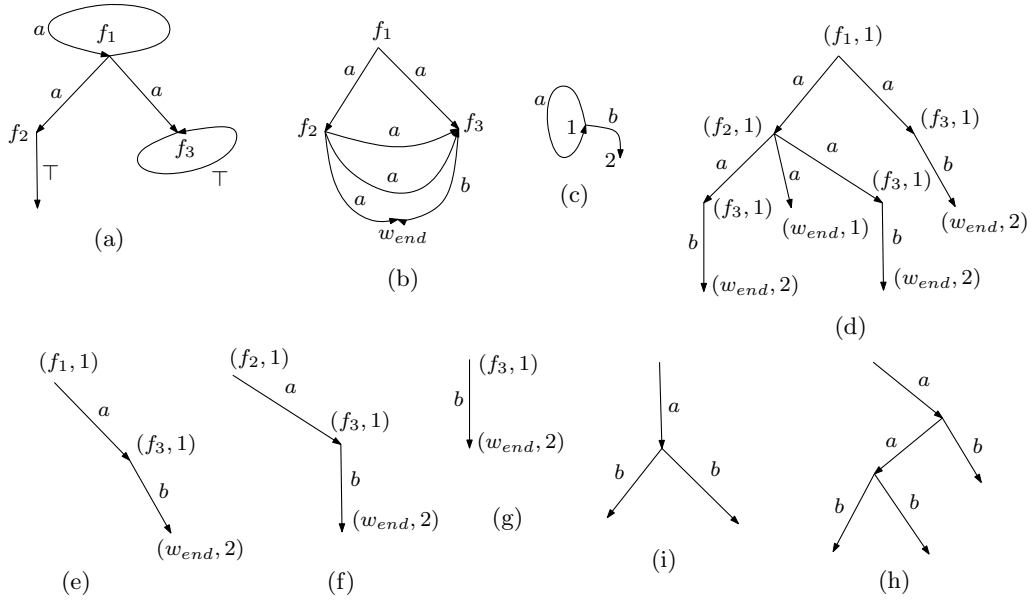

Figure 12: (a) The operational graph of the structural recursion of Example 4.28. (b) The operational graph of the structural recursion of Example 4.33. (c) A data graph with a cycle resulting a non-empty output for the structural recursion of (b). (d) The intersection of the operational graph of (b) and the data graph of (c). (e) A path from the root to a constructor in the intersection of (d). (f) The path through which the n.i. logical function of the condition of the then-edge of the path of (e) is satisfied. (g) The path through which a construction is done and as a result the else-edge of the path of (f) is kept. (h) The result of the second (recursive) call of the `Tree_Constructor` algorithm. (i) The final result of the algorithm, when it is called for the structural recursion of (b) and the data graph of (c).

4.4 Simulating data graphs with data trees

In this subsection an algorithm is developed, with which from an instance resulting a non-empty output for a structural recursion a tree can be constructed also returning a non-empty result for the same structural recursion. This achievement will gain its full importance, when the connection between structural recursions and alternating tree automata formulated in Section 7 will be used to find the complexity class to which the emptiness and containment problems of structural recursions belong in the general case.

```
/* f = ({f_1, ..., f_n}), Σ, F_I, Γ) is a structural recursion,
   I is an instance,
   pa= e_1 ... e_m is a path from a root to a constructor edge in
     U_f ⊓ I s.t. none of its edges is deleted in the condition evaluation,
   e_i = ((u_{f_{j_i}}, u_i), a_i, (u_{f_{j_{i+1}}}, u_{i+1})) ∈ E.pa  (1 ≤ i ≤ m, j_1, ..., j_{m+1} ∈ {1, ... n})),
   if e_i is a then-edge (e_i ∈ Th(γ)), then pr^i_1, ..., pr^i_{k_i} denote those
     premises in Pr(γ) belonging to n.i. conditions that becomes true,
     n.i.pa^i_j denote the path through which pr^i_j becomes true,
     n.i.pa^i_j does not contain pr^i_j  (1 ≤ j ≤ k_i),
   if e_i is an else-edge (e_i ∈ El(γ)), then pr^i_1, ..., pr^i_{s_i} denote
     the premises belonging to i. conditions that becomes false,
     i.pa^i_j denote the path through which pr^i_j becomes false,
     i.pa^i_j does not contain pr^i_j, (1 ≤ j ≤ s_i),
   f^i = ({f_1, ..., f_n}, Σ, {f_{j_i}}, Γ) (Remark 3.10.),
   I^{u_i} has the same nodes and edges as I, u_i is designated as the root,
   T_C abbreviates Tree_Constructor. */

Tree_Constructor(f, I, pa)
{
  for i = 1 ... n
    if (e_i is a then-edge) then
       aux_Tree_i := {e_i : T_C(f^i, I^{u_i}, n.i. pa^i_1) ∪ ... ∪ T_C(f^i, I^{u_i}, n.i. pa^i_{k_i})};
    else if (e_i is an else-edge) then
       aux_Tree_i := {e_i : T_C(f^i, I^{u_i}, i. pa^i_1) ∪ ... ∪ T_C(f^i, I^{u_i}, i. pa^i_{s_i})};
    else
       aux_Tree_i := e_i;
  for i = 1..n − 1
    add an ε edge from the end node of e_i in aux_Tree_i to the
    starting node of e_{i+1} in aux_Tree_{i+1};
  eliminate the ε edges from the result of the previous for loop;
  I_tree := the result after this elimination;
  return I_tree;
}
```


Figure 13: The algorithm that for structural recursion f, instance I, where $f(I)$ is non-empty, constructs a tree I_{tree} to which $f(I_{tree})$ is also non-empty.

The pseudo code of the algorithm can be found in Figure 13. The structural recursion f and the instance I on which f returns a non-empty result are parts of the input. Informally, in the construction an appropriate pregraph of I is taken and its cycles are substituted with paths that one can construct by traversing these cycles one or more times.

More precisely, a path $pa = e_1 \dots e_m$ from a root of $U_f \sqcap I$ to a constructor edge is chosen through which an edge is constructed in $f(I)$, i.e., none of its then- or else-edges are deleted during the condition elimination. If e_i is a then edge, with

$$^{n.i.}pa^i_1, \dots, ^{n.i.}pa^i_{k_i} \, (1 \leq i \leq m)$$

we denote those paths through which the n.i. conditions of the condition of e_i that become *true* in the evaluation are satisfied. Here, $^{n.i.}pa^i_{k_j}$ does not contain the premise representing the corresponding n.i. condition ($1 \leq j \leq k_i$). Similarly, if e_i is an else-edge, then

$$^{i.}pa^i_1, \dots, ^{i.}pa^i_{s_i}$$

denote those paths through which the i. conditions of the condition of e_i that become $false$ in the evaluation are falsified. Again, $^{i.}pa^i_{k_j}$ does not contain the premises representing these i. conditions ($1 \leq j \leq s_i$). With the appropriately modified structural recursion (see Figure 13. for the exact details), the method is called recursively on each $^{n.i.}pa^i_j$,$^{i.}pa^i_r$ and the results of these calls are connected together in an appropriate order to get the final result I_{tree} ($1 \leq j \leq k_i, 1 \leq r \leq s_i$). Basically, the aforementioned paths are straightened, i.e., if in $^{n.i.}pa^i_j$ ($^{i.}pa^i_j$) some cycles are traversed one or more times, then these cycles are substituted with these traversals.

Example 4.33. Consider structural recursion $f = (\{f_1, f_2, f_3\}, \Sigma, \{f_1\}, \Gamma)$, whose transformation rules are as follows:

$f_1 \colon (\{a : t\}) =$ if n.i.$(f_2(t))$ then $f_3(t)$ $\qquad$ $f_3 \colon (\{b : t\}) = \{b : \{\}\}$

$f_2 \colon (\{a : t\}) =$ if i.$(f_3(t))$ then $\{\}$
else $f_3(t)$.

The operational graph of f can be found in Figure 12.(b). The intersection of U_f and the instance I of Figure 12.(c) can be seen in Figure 12.(d). In Figure 12.(e)-(h) one can track the steps of the `Tree_Constructor` algorithm called on f, I and the path of (e). Clearly, this path is from the root of $U_f \sqcap I$ and it ends in a constructor edge. It contains a single conditional edge $((f_1, 1), a, (f_3, 1))$. The corresponding n.i. condition is satisfied by the path of (f), here we omitted the premise. This path contains a single conditional edge $((f_1, 1), a, (f_3, 1))$, which happens to be an else-edge. The corresponding i. condition becomes $false$ through the path of (g), where the premise was omitted again. This path is a single non-conditional edge, hence when the `Tree_Constructor` algorithm is called, it returns an edge with the same label. This is appended under the else-edge of (f), consider (g), and the result of this construction is appended to the endnode of the then-edge of (e). The final result can be seen in (i).

Lemma 4.34. *For an arbitrary structural recursion f and instance I to which $f(I)$ is not empty,*

(i) I_{tree} is a tree,

(ii) $|I_{tree}|$ is in $O(|U_f|^2|I|)^k$, where k denotes the number of steps necessary to accomplish condition evaluation in $U_f \sqcap I$.

Proof. Throughout the proof the notation of the `Tree_Constructor` algorithm is used. The statement is proven by applying induction on k. First suppose that $k = 0$. This means that pa is without then- or else-edges, hence $I_{tree} = pa$. Clearly, both (i) and (ii) hold here.

Suppose now that the statement is *true* for $k \leq n$ and $k := n + 1$. According to our supposition, each T_C is a tree, hence aux_Tree_i is also a tree, where T_C is either $T\_C(f^i, I^{u_i}, ^{n.i.}\, pa^i_j)$ or $T\_C(f^i, I^{u_i}, ^{i.}\, pa^i_r)$ or e_i. When they are connected, obviously, the result remains a tree. This proves (i). On the other hand the size of T_C is $O(|U_f|^2|I|)^{k-1}$, thus $|Aux\_Tree_i| \leq |U_f| * O(|U_f|^2|I|)^{k-1}$. Since $|pa| \leq |U_f||I|$, $|I_{tree}|$ is in $O(|U_f|^2|I|)^k$. This concludes the proof. ■

Corollary 4.35. *Keeping the above notations* $|I_{tree}|$ *is in* $O(|U_f|^2|I|)^{|U_f|}$

Proof. Straightforwardly, $k \leq U_f$ (where k denotes the number of steps necessary to accomplish condition evaluation in $U_f \sqcap I$). ■

This means that if we consider combined complexity, i.e., the sizes of both U_f and I are taken into account, then there is an exponential growth in the size of U_f. At first glance, this seems to be unacceptable from a practical point of view, but we would like to emphasize that in what follows I_{tree} will be never constructed, only the fact of its existence will be used.

Next we prove that $f(I_{tree})$ is not empty, if $f(I)$ is not empty.

Lemma 4.36. *Let* f *and* I *be a structural recursion in* $SR(n.i., i., el)$ *and an instance s.t.* $f(I)$ *is not empty. Then there is an operational homomorphism* ρ *from* $V.U_f \sqcap I_{tree}$ *to* $U_f \sqcap I$.

Proof. Recall that the nodes of I_{tree} is taken from the nodes of $U_f \sqcap I$. With $\mu : V.I_{tree} \to V.I$ we denote the mapping assigning to each $(u_f, u_I) \in V.I_{tree}$ its ancestor image u_I in $V.I$. In what follows we prove that μ is a simulation from I_{tree} to I, then the statement will follow from Lemma 4.23.

If (u_f, u_I) is a root of I_{tree}, then $\mu((u_f, u_I)) = u_I$ is the root of I. Thus condition (i) of the definition of simulation on Page 16 is satisfied.

(ii) For an arbitrary edge $((u_f, u_I), p \wedge a, (v_f, v_I))$ of I_{tree},

$$\mu(((u_f, u_I), p \wedge a, (v_f, v_I))) = (u_I, a, v_I)$$

is obviously an edge of I. ■

Lemma 4.37. *Let* f *and* I *be a structural recursion and an instance s.t.* $f(I)$ *is not empty. Denote* ρ *the operational homomorphism given by Lemma 4.36. Then, if a then- or else-edge* e_I *having an ancestor image according to* ρ *is kept in the condition evaluation over* $U_f \sqcap I$, *then* $\rho^{-1}(e_I) = e_{tree}$ *is also kept in* $U_f \sqcap I_{tree}$.

Proof. Suppose that the lemma does not hold for edge $e_I \in E.U_f \sqcap I$, and in the previous steps of the condition evaluation over $U_f \sqcap I$ none of the edges

have violated the statement. In the proof γ will denote the condition which e_I belongs to.

Assume first that e_I is a then-edge. Since this is the first step, when the statement of the lemma is violated, for each path from the starting node of e_I to a constructor edge that satisfies a premise in $Pr(\gamma)$ belonging to a n.i. condition, its correspondence still exists in $U_f \sqcap I_{tree}$ under the starting node of e_{tree}. (Remember the aux_Tree constructed for e_I contains these paths.) Thus, the ancestor images of these premises according to ρ are also satisfied. On the other hand, if there is a path pa through premise pr belonging to an arbitrary i. condition in $U_f \sqcap I_{tree}$ to a constructor edge which is without conditional edges at one point of the condition evaluation, i.e., the i. condition becomes $false$, then from our supposition follows that $\rho(pa)$ is also without conditional edges at that point of the condition evaluation over $U_f \sqcap I$. Since ρ is an operational homomorphism, $\rho(pr)$ also belongs to an i. condition and $\rho(pa)$ also ends in a constructor edge, hence $\rho(pr)$ also becomes $false$. From this it follows that if a premise in $Pr(\gamma)$ belonging to an i. condition becomes $true$, then its ancestor image also becomes $true$. All together, we get that if a premise of any kind in $Pr(\gamma)$ becomes $true$, then its correspondence also becomes $true$. This means that the correspondence of γ in $U_f \sqcap I_{tree}$ is also satisfied, consequently e_{tree} should also be kept, which is a contradiction.

If e_I is an else-edge, we have to prove that if a premise in $Pr(\gamma)$ becomes $false$, then its correspondence also becomes $false$. Consider first those premises of $Pr(\gamma)$, which belongs to an i. condition and becomes $false$. Again, an aux_Tree has been constructed containing the correspondences of those paths that results a non-empty result. Thus, from our supposition it follows that the ancestor images of the aforementioned premises also become $false$. On the other hand, in a similar way as in the previous case, it can be shown that if a premise in $Pr(\gamma)$ belonging to a n.i. condition becomes $false$, then its correspondence cannot evaluate to $true$. All in all, e_{tree} should be kept in this case as well. ■

Theorem 4.38. *For structural recursion f and instance I, from the non-*

emptiness of $f(I)$ the non-emptiness of $f(I_{tree})$ follows. In other words: if there is an instance on which a structural recursion returns a non-empty output, then there should be a tree which also results a non-empty output for this structural recursion.

Proof. Suppose that the `Tree_Constructor` algorithm has been called with pa, which, remember, is a path from a root of $U_f \sqcap I$ to a constructor edge, whose conditional edges are all kept in the condition evaluation. Clearly, $\rho^{-1}(pa)$ is also a path from a root to a constructor edge in $U_f \sqcap I_{tree}$. According to Lemma 4.37. its conditional edges are also kept during the condition evaluation, which by definition means that the result of $f(I_{tree})$ is not empty. ■

Remark 4.39. Since aux_Tree_i-s are connected to the endnode of e_i-s (consider Figure 13.), the result of the `Tree_Constructor` algorithm is always root-edged.

Remark 4.40. Denote $Out(G, a, u)$ the number of outgoing a edges from node u of graph G. Then from the construction of aux_Tree-s it clearly follows that for each node (f_i, u) of I_{tree},

$$Out(I_{tree}, a, (f_i, u)) \leq Out(U_f, a, f_i).$$

The generalized algorithm. Using the just developed techniques we give another algorithm, `Tree_Simulator` (Figure 14.), constructing a tree with which, for a given structural recursion f and instance I, the run of the condition evaluation over $U_f \sqcap I$ can be simulated (in the spirit of Lemma 4.37.). In this case, however, any pregraph of I can be used for the construction, and the resulting tree not necessarily returns a non-empty output for f.

We start with a tree I_{par} to which there exists a simulation μ from I_{par} to I. According to Lemma 4.23. there is an operational homomorphism ρ from $U_f \sqcap I_{par}$ to $U_f \sqcap I$. In the algorithm we consider the steps of condition

```
/* f = ({f_1,...,f_n}, Σ, F_I, Γ) is a structural recursion,
   I is an instance,
   I_par is a tree data graph to which possibly subtrees are added,
     and Tree_Simulator is called with this new tree data graph,
   μ is a simulation from I_par to I,
   ρ is an operational homomorphism from U_f ⊓ I_par to U_f ⊓ I
     given by Lemma 4.23.,
   M is the number of steps of the condition evaluation over U_f ⊓ I,
   γ is the condition to which ρ(e) belongs,
   in (†) pa_1,...,pa_k are paths in U_f ⊓ I from the end nodes of
     premises of Pr(γ) belonging to n.i. conditions that satisfy
     these premises and do not have an ancestor image according to
     ρ in U_f ⊓ I_par,
   in (‡) pa_1,...,pa_k are paths of U_f ⊓ I from the end nodes of
     premises of Pr(γ) belonging to i. conditions that makes
     these premises false and do not have an ancestor image according to
     ρ in U_f ⊓ I_par,
   u_1,...,u_k denote the starting nodes of pa_1,...,pa_k,
   f^{i_j} = ({f_1,...,f_n}, Σ, {f_{i_j}}, Γ), where f_{i_j} is the structural functions called
     on the first edge of pa_j (1 ≤ j ≤ k),
   I^{u_i} has the same nodes and edges as I, u_i is designated to be the root,
   T_C abbreviates Tree_Constructor (Figure 13.),
   μ_new denotes the simulation from I_new to I (see Lemma 4.41). */

Tree_Simulator(f, I, I_par, μ)
{
  for i = 1..M
    execute the i^th step of condition evaluation over U_f ⊓ I and U_f ⊓ I_par;
      if during this execution there is an edge e = (e_f, e_par) s.t.
  (†) e is an else-edge in U_f ⊓ I_par, e is kept, while ρ(e) is deleted or
  (‡) e is a then-edge s.t. it is kept, while ρ(e) is deleted, then
        aux_Tree := T_C(f^{i_1}, I^{u_1}, pa_1) ∪ ... ∪ T_C(f^{i_k}, I^{u_k}, pa_k);
        connect the root of aux_Tree to the endnode of e_par,
        denote I_new this new tree;
        return Tree_Simulator(f, I, I_new, μ_new);
  return I_par;
}
```

evaluations over these graphs one after the other. In (†) we handle that case, when a then-edge e in $U_f \sqcap I_{par}$ is deleted, while its counterpart $\rho(e)$ is kept. Note that this does not necessarily happen in the same step. Denote γ_1, γ_2 the transformation rules which e and $\rho(e)$ respectively belongs to. The deletion of e and the non-deletion of $\rho(e)$ means that there are paths from premises of $Pr(\gamma_2)$ belonging to n.i. conditions to constructor edges without counterparts in $U_f \sqcap I_{par}$. Since our aim is to simulate the run of condition evaluation on $U_f \sqcap I$, I_{par} should be extended in such a way that in the intersection of U_f and the new tree these counterparts could not be missing. Thus, the `Tree_Constructor` algorithm is invoked on the aforementioned paths, whose role, remember, is to construct trees from those data graphs that results a non-empty output for a structural recursion. Afterwards the results of these calls are appended under the appropriate node of I_{par}.

In (‡) reversely e is kept and $\rho(e)$ is deleted. This means that there are paths from premises of $Pr(\gamma_2)$ belonging to i. conditions to constructor edges without counterparts in $U_f \sqcap I_{par}$, which make these premises $false$. In other words, in this respect the situation is the same as was in (†). Again, the `Tree_Constructor` algorithm is called with these paths, and the result is connected to the appropriate edge of I_{par}.

With this new tree I_{new} the `Tree_Simulator` algorithm is called again. In this case condition (†), (‡) will not apply to the correspondence of e in $U_f \sqcap I_{new}$, hence at the end of the computation the condition evaluation of $U_f \sqcap I$ can be simulated with the final result.

Formally, the following statements should be proven.

Lemma 4.41. *Keeping the notations of the `Tree_Simulator` algorithm, there exists a simulation μ_{new} from I_{new} to I.*

Proof. I_{new} consists of two parts, I_{par} and aux_Tree. We know that there is a simulation μ from I_{par} to I. From the structure of the `Tree_Constructor` algorithm it turns out that aux_Tree is constructed using the edges of $U_f \sqcap I$. Define relation $\hat{\mu}$, with which μ will be extended, from $V.aux\_Tree$

to $V.I$ in the following way: $\hat{\mu}((u_f, u_I)) := u_I$. Let

$$((u_f, u_I), a, (v_f, v_I))$$

be an edge in aux_Tree. Then

$$(\flat)\ \hat{\mu}(((u_f, u_I), a, (v_f, v_I))) = \mu((u_I, a, v_I))$$

is also an edge in I. Recall that aux_Tree is connected to the end node of an edge of I_{par}. In other words, nodes u and u_0 are contracted, where u, u_0 respectively denote this end node and the root of aux_Tree. It is easy to see that $\mu(u)$ and $\hat{\mu}(u_0)$ is the same node (in I). Hence μ can be extended with $\hat{\mu}$. Denote μ_{new} this new mapping.

We show now that μ_{new} fulfills requirements (i)-(ii) of the definition of simulation on page 16. Since the root of I_{par}, w_0, and I_{new} is the same, and $\mu_{new}(w_0)$ is equal to $\mu(w_0)$, condition (i) trivially holds. Condition (ii) follows from observation ($\flat$) and from the fact that μ is also a simulation. ■

Lemma 4.42. *The* `Tree_Simulator` *algorithm terminates.*

Proof. Keeping again the notations of the algorithm, consider $U_f \sqcap I_{new}$. According to Lemma 4.37. condition (†) or (‡) does not apply to any of the edges having an ancestor image in aux_Tree, where remember I_{new} is constructed from I_{par} and aux_Tree. Furthermore, consider that edge of $U_f \sqcap I_{par}$ which fulfilled the requirements of (†) or (‡). Again, from Lemma 4.37. it follows that in this case the correspondence of e in $U_f \sqcap I_{new}$ does neither satisfy (†) nor (‡). Hence, after each step, the number of those edges that satisfy condition (†) or (‡) decreases. Note that there are not any edges that would satisfy both (†) and (‡). ■

Lemma 4.43. *Keeping the notations of the* `Tree_Simulator` *algorithm, I_{new} is a tree.*

Proof. The statement trivially follows from the fact that both I_{par} and aux_Tree are trees. ■

Lemma 4.44. *Let f be a structural recursion and I, I_{par} instances s.t. there is a simulation from I_{par} to I. Denote I_{tree} the result of the* `Tree_Simulator` *algorithm applied on f, I, I_{par}. Then, if a then- or else-edge e_I is kept during condition evaluation in $U_f \sqcap I$, then $e_{tree} = \rho^{-1}(e_I)$ is also kept in $U_f \sqcap I_{tree}$. Here, ρ denotes the operational homomorphism given by the simulation of Lemma 4.41.*

Proof. Essentially, the proof works in the same way as the proof of Lemma 4.37. As there, suppose that the lemma does not hold for edge $e_I \in E.U_f \sqcap I$, and in the previous steps of condition evaluation none of the edges have violated the statement. Denote γ_1, γ_2 the condition to which e_{tree}, e_I respectively belongs.

Suppose first that e_I is a then-edge. Then e_{tree} fulfills the requirements of condition (†), thus in one of the instantiations of the `Tree_Constructor` algorithm the appropriate subtree aux_Tree has been connected under the ancestor image of e_{tree} according to I_{par}. From the result of Lemma 4.37. it follows that if a premise in $Pr(\gamma_2)$ belonging to a n.i. condition is satisfied, then its ancestor image according to ρ is also satisfied. On the other hand, in the same way as in the proof of Lemma 4.37. one can show that if a premise pr in $Pr(\gamma_1)$ belonging to an i. condition becomes $false$, then $\rho(pr)$ also becomes $false$. All together we get that if a premise of any kind in $Pr(\gamma_2)$ becomes $true$, then its ancestor image according to ρ also becomes $true$. This again implies that if e_I is kept, then e_{tree} should also be kept, which is a contradiction.

The proof of the other case, when e_I is an else-edge, can be given along the same lines as for the similar case in Lemma 4.37., hence we omit the details. ■

Proposition 4.45. *Let f be a structural recursion, I an arbitrary instance and I_{par} a data tree from which there is a simulation μ to I. Denote I_{tree} the result of the* `Tree_Simulator` *algorithm called on f, I, I_{par}, μ.*

1. *Then, from the emptiness of $f(I)$ the emptiness of $f(I_{tree})$ follows.*

2. *Denote I' the image of I_{par} according to μ in I. Then, if $U_f \sqcap I'$ (which is clearly a pregraph of $U_f \sqcap I$) contains a path from the root to a constructor edge whose edges are kept in condition elimination in $U_f \sqcap I$, then $f(I_{tree})$ is not empty.*

Proof. (i) Suppose that on the contrary there is a path from a root of $U_f \sqcap I_{tree}$ to a constructor edge after condition evaluation, i.e., the result of $f(I_{tree})$ is not empty. Since there is an operational homomorphism from $U_f \sqcap I_{tree}$ to $U_f \sqcap I$ this path did exist in $U_f \sqcap I$ before condition evaluation. According to Lemma 4.44. none of its edges is deleted during condition evaluation, otherwise the corresponding edge of the path in $U_f \sqcap I_{tree}$ would also be deleted, which entails the non-emptiness of $f(I)$. A contradiction.

The proof of statement (ii) can be given in a similar way. ■

Remark 4.46. Similarly to Remark 4.39. the result of the `Tree_Simulator` will be always root-edged.

Remark 4.47. Again, similarly to Remark 4.40., if data graph I_{par}, to which algorithm `Tree_Simulator` is firstly initialized, is deterministic, i.e., each of its neighbouring edges has a different label, then

$$Out(I_{par}, a, (f_i, u)) \leq Out(U_f, a, f_i),$$

where remember $Out(G, a, u)$ denotes the number of outgoing a edges from node u of graph G. This is again obvious from the construction of aux_Tree-s in the algorithm.

Remark 4.48. Note that in both algorithms aux_Tree-s are defined in the same way and in the same situation. Thus, if `Tree_Simulator` is called with a path in $U_f \sqcap I$ from the root to a constructor edge, whose edges are kept during condition evaluation, remember that `Tree_Constructor` is always called with such paths, then the two algorithms construct the same tree.

5 Operations on structural recursions

In this section the usual operations [12]: complement, intersection and union are introduced for structural recursions. As it has been already mentioned throughout this dissertation structural recursions are treated as acceptors, which means that only the fact of construction is of importance, other properties of the result will not be taken into consideration. The definitions of operations are formulated with this in mind.

5.1 Complement

5.1.1 Complement of structural recursions in (SR(n.i., i.)).

In order to be able to define the complement structural recursions should be transformed into a special equivalent form, the *conditional form*. Here, in the transformation rules at most a single edge will be constructed, and all calls of structural functions outside the conditions will be moved into the conditions. Beside its role, the conditional form of structural recursions will be widely used in the sequel.

The second modification, the removal of the calls of the structural functions into the conditions, is achieved by assigning a formula to a forests constructed in a transformation rule. Formally, let $f = (\{f_1, \ldots, f_n\}, \Sigma, F_I, \Gamma)$ be a structural recursion in $SR(n.i., i.)$. For a transformation rule γ this formula is denoted $FoForm(\gamma)$. Suppose that $f_{i_1}, \ldots, f_{i_k}$ are the structural functions that are called in $Frst(\gamma)$. Then $FoForm(\gamma)$ is

$$\text{n.i.}(f_{i_1}) \vee \ldots \vee \text{n.i.}(f_{i_k}),$$

where $(\{i_1, \ldots, i_k\} \subseteq \{1, \ldots n\})$. Clearly, $FoForm(\gamma)$ becomes *true*, if at least one of the called structural functions results a construction.

The conditional form of f is denoted $f^{cf} = (\{f_1^{cf}, \ldots, f_n^{cf}\}, \Sigma, F_I^{cf}, \Gamma^{cf})$. As the notation shows each structural function of f has its counterpart in f^{cf}, besides f_i^{cf} is in F_I^{cf}, if f_i is in F_I. For transformation rule $\gamma_{i,a}$, $\gamma_{i,a}^{cf}$ is defined as follows $(a \in \Sigma \cup \{*\})$:

1. if $\gamma_{i,a}$ is without a condition, then

   (i) if there is a construction, then

   $$Form(\gamma^{cf}_{i,a}) \text{ is not given, } Frst(\gamma^{cf}_{i,a}) = \{\psi : \{\}\}.$$

   (ii) If there is no construction, but at least one structural function is called, then

   $$Form(\gamma^{cf}_{i,a}) = FoForm(\gamma_{i,a}),\ Frst(\gamma^{cf}_{i,a}) = \{\psi : \{\}\}.$$

   (iii) If there is neither construction, nor structural functions are called, then

   $$Form(\gamma^{cf}_{i,a}) \text{ is not given, } Frst(\gamma^{cf}_{i,a}) = \{\}.$$

2. If $\gamma_{i,a}$ is with a condition, then

   (i) if there is a construction, then

   $$Form(\gamma^{cf}_{i,a}) = Form(\gamma_{i,a}),\ Frst(\gamma^{cf}_{i,a}) = \{\psi : \{\}\}.$$

   (ii) If there is no construction, but at least one structural function is called, then

   $$Form(\gamma^{cf}_{i,a}) = Form(\gamma_{i,a}) \wedge FoForm(\gamma_{i,a}),\ Frst(\gamma^{cf}_{i,a}) = \{\psi : \{\}\}.$$

   (iii) If there is neither construction, nor structural functions are called, then

   $$Form(\gamma^{cf}_{i,a}) \text{ is not given, } Frst(\gamma^{cf}_{i,a}) = \{\}.$$

The rationale behind the rewriting rules is quite obvious. For example in rule 2.(ii), if the condition and the formula of the forest of $\gamma_{i,a}$ are both satisfied, which guarantees that the then-branch is to be executed and as a result at least an edge is constructed, then a ψ edge constructed in the conditional form. The next proposition trivially holds:

Proposition 5.1. *Let f be a structural recursion in $SR(n.i., i.)$ and I an arbitrary instance. Then $f(I)$ is not empty $\Leftrightarrow$ $f^{cf}(I)$ is not empty.*

Remark 5.2. It is easy to see that f and f^{cf} are not equivalent to each other in general.

Remark 5.3. Clearly, for an arbitrary structural recursion f, $|f^{cf}| \leq c|f|$, where remember, $|f|$ was defined as $\max\{|V.U_f|, |E.U_f|\}$ and $0 < c \leq 2$.

Now we show that if only the non-emptiness of the result is in question, then we may substitute the initial structural functions with a single structural function. For structural recursion $f = (\{f_1, \ldots, f_n\}, \Sigma, F_I, \Gamma)$ consider its conditional form. Suppose that $f^{cf}_{i_1}, \ldots, f^{cf}_{i_k}$ are the initial structural functions of f ($\{i_1, \ldots, i_k\} \subseteq \{1, \ldots, n\}$). Denote f_{one} the structural function, with which we will substitute $f^{cf}_{i_1}, \ldots, f^{cf}_{i_k}$. The transformation rules of f_{one} are defined as follows. Let a be a symbol in $\Sigma \cup \{*\}$, and assume that $f^{cf}_{j_1}, \ldots, f^{cf}_{j_s}$ are those initial structural functions that have a transformation rule for a, in which a ψ-edge is constructed ($\{j_1, \ldots, j_s\} \subseteq \{i_1, \ldots, i_k\}$). Then

$$Form(\gamma_{one,a}) := Form(\gamma^{cf}_{j_1,a}) \vee \ldots \vee Form(\gamma^{cf}_{j_s,a}),\ Frst(\gamma_{one,a}) := \{\psi : \{\}\}.$$

Here $Form(\gamma^{cf}_{j_o,a})$ is *true*, if $\gamma^{cf}_{j_o,a}$ does not have a condition ($1 \leq o \leq s$). In the next step f_{one} should be added to the set of structural functions and Γ should be extended with its transformation rules. Furthermore, f_{one} should be defined to be the only initial structural function. Denote f^{one} the resulting structural recursion. The subsequent proposition is straightforward.

Proposition 5.4. *Let f be a structural recursion and I an arbitrary instance. Then $f(I)$ is not empty $\Leftrightarrow$ $f^{one}(I)$ is not empty.*

Definition 5.5. *When a structural recursion f is in conditional form and it has a single initial structural function, then we say that it is in* conditional normal form *(*CNF*). If f is also complete, then it is in* complete conditional normal form *(*CCNF*).*

Remark 5.6. In Propositions 4.2., 5.1. and 5.4. it has been shown that to every structural recursion in $SR(n.i., i.)$ there is an equivalent structural recursion in CCNF. Later in this subsection this result will be extended to all structural recursions.

After these preparations the complement of a structural recursion can be defined. For the definition of the transformation rules of the complement, the *negation* of a formula of a transformation rule is introduced. Consider the formula $Form(\gamma)$ of an arbitrary transformation rule γ. In $\neg Form(\gamma)$ applying rules:

$$\neg(A \wedge B) \equiv \neg A \vee \neg B,\ \neg(A \vee B) \equiv \neg A \wedge \neg B$$

(here A, B are propositional formulas) first the negations are moved directly before the n.i., i. conditions. Afterwards $\neg$n.i.$(f_j(t))$ is substituted with i.$(f_j(t))$ and $\neg$i.$(f_j(t))$ with n.i.$(f_j(t))$.

Let $f = (\{f_1, \ldots, f_n\}, \Sigma, \{f_i\}, \Gamma)$ be a structural recursion in $SR(n.i., i.)$. In the rewriting we will assume that f is in CCNF. If it is not so, then first f should be rewritten into CCNF. In the *complement structural recursion*

$$\tilde{f} = (\{\tilde{f}_i, f_1, \ldots, f_n\}, \Sigma, \{\tilde{f}_i\}, \tilde{\Gamma})$$

only one extra structural function $\tilde{f}_i$ is added to the set of structural functions (Γ is extended with the transformation rules of $\tilde{f}_i$), which is defined to be the single initial structural function. In the definition of $\tilde{f}_i$ only the transformation rules of f_i are considered, where f_i is the initial structural function of f.

Understandably, $\tilde{\gamma}_{i,a}$ will be defined in the opposite way as $\gamma^{cf}_{i,a}$ was. If there is a condition in $\gamma^{cf}_{i,a}$, then the negation of this condition will be taken. On the other hand, if there is no condition in $\gamma^{cf}_{i,a}$, then $\tilde{\gamma}_{i,a}$ will be also without a condition. However, in this case, if $\gamma^{cf}_{i,a}$ constructs a ψ edge, then $\tilde{\gamma}_{i,a}$ will return the empty graph and vice versa. Formally, for $\gamma_{i,a}$, $\tilde{\gamma}_{i,a}$ is given in the following way ($a \in \Sigma \cup \{*\}$):

1. if $\gamma_{i,a}$ is without a condition, then

(i) if there is a construction, then

$$Form(\tilde{\gamma}_{i,a}) \text{ is not given, } Frst(\tilde{\gamma}_{i,a}) = \{\}.$$

(ii) If there is no construction, but at least one structural function is called, then

$$Form(\tilde{\gamma}_{i,a}) = \neg FoForm(\gamma_{i,a}),\ Frst(\tilde{\gamma}_{i,a}) = \{\psi : \{\}\}.$$

(iii) If there is neither construction, nor structural functions are called, then

$$Form(\tilde{\gamma}_{i,a}) \text{ is not given, } Frst(\tilde{\gamma}_{i,a}) = \{\psi : \{\}\}.$$

2. If $\gamma_{i,a}$ is with a condition, then

(i) if there is a construction, then

$$Form(\tilde{\gamma}_{i,a}) = \neg Form(\gamma_{i,a}),\ Frst(\tilde{\gamma}_{i,a}) = \{\psi : \{\}\}.$$

(ii) If there is no construction, but at least one structural function is called, then

$$Form(\tilde{\gamma}_{i,a}) = \neg Form(\gamma_{i,a}) \vee \neg FoForm(\gamma_{i,a}),\ Frst(\tilde{\gamma}_{i,a}) = \{\psi : \{\}\}.$$

(iii) If there is neither construction, nor structural functions are called, then

$$Form(\tilde{\gamma}_{i,a}) \text{ is not given, } Frst(\tilde{\gamma}_{i,a}) = \{\psi : \{\}\}.$$

The following proposition trivially holds.

Proposition 5.7. *Let f be a structural recursion in CNF and in $SR(n.i., i.)$, whose single initial structural function is f_i. If $\widetilde{\tilde{f}}_i$ is substituted with f_i in $\widetilde{\tilde{f}}$, then $\widetilde{\tilde{f}}$ becomes syntactically the same as f.*

Proposition 5.8. *Let $f = (\{f_1, \ldots, f_n\}, \Sigma, \{f_i\}, \Gamma)$ be a structural recursion in $SR(n.i., i.)$ and I root-edged instance, then $f(I)$ is not empty $\Leftrightarrow$ $\tilde{f}(I)$ is empty.*

Proof. We may assume that the root edge of I is labelled with a. We prove the statement by using induction on the number of steps k in the condition evaluation over $U_{f^{cf}} \sqcap I$. (Here, if f is already in conditional form, then f^{cf} is the same as f.) Suppose that $k = 1$. This means that in the construction of $\gamma^{cf}_{i,a}$ (f^{cf}_i is the initial structural function of f^{cf}) either rule 1.(i) or 1.(iii) or 2.(iii) was used. (Otherwise, there would be a condition in $\gamma^{cf}_{i,a}$, and the condition evaluation could not be executed in one step.) If $f(I)$ is not empty, then rule 1.(i) was used. In this case there is no condition in $\tilde{\gamma}_{i,a}$ and no other structural function is called, hence $\tilde{f}(I)$ is trivially empty. On the other hand, if $f(I)$ is empty, then either rule 1.(iii) or 2.(iii) was applied. In both cases $\tilde{f}(I)$ is obviously non-empty.

Suppose that the statement holds for $k \leq m$ and $k := m+1$. In this case 1.(ii), 2.(i) or 2.(ii) could have been applied in the construction of $\gamma^{cf}_{i,a}$. In each cases $Form(\tilde{\gamma}_{i,a})$ is the negation of $Form(\gamma^{cf}_{i,a})$. Hence, if n.i.$f_j(t)$ (i.$(f_j(t))$) occurs in $Form(\gamma^{cf}_{i,a})$, then i.$f_j(t)$ (n.i.$(f_j(t))$) appears in $Form(\tilde{\gamma}_{i,a})$ in the corresponding position ($1 \leq j \leq n$). Straightforwardly, if n.i.$f_j(t)$ becomes $true$, then i.$f_j(t)$ becomes $false$ and vice versa. Note that now, if an interpretation $\mathcal{I}$ makes proposition formula $P\ true$, then interpretation $\tilde{\mathcal{I}}$, which assigns the opposite truth value to each propositional variable that I has assigned, makes $\neg P\ false$. From this observation it follows that if $Form(\gamma^{cf}_{i,a})$ becomes $true$ ($f(I)$ is not empty), then $Form(\tilde{\gamma}_{i,a})$ becomes $false$, ($\tilde{f}(I)$ is empty) and vice versa.

The reverse direction of the statement is a straightforward consequence of Proposition 5.7. ■

Remark 5.9. The following example shows that if I is not root-edged, then both $f(I)$ and $\tilde{f}(I)$ can be non-empty. Let $f = (\{f_1\}, \Sigma, \{f_1\}, \Gamma)$ be a structural recursion, whose transformation rules are as follows:

$f_1 \colon (\{a : t\}) =$ if n.i.$(f_2(t))$ then $\{\psi : \{\}\}$
$(\{* : t\}) = \{\}$

$f_2 \colon (\{b : t\}) = \{\psi : \{\}\}$
$(\{* : t\}) = \{\}$.

Note that f constructs on instance I if and only if its root has an outgoing a edge followed by a b edge.

On the other hand, $\tilde{f} = (\{\tilde{f}_1, f_1, f_2\}, \Sigma, \{\tilde{f}_1\}, \tilde{\Gamma})$, where the transformation rules of $\tilde{f}_1$ are as follows:

$\tilde{f}_1$: $(\{a : t\})$= if i.$(f_2(t))$ then $\{\psi : \{\}\}$
$(\{* : t\})$= $\{\psi : \{\}\}$

Let I be $\{a : \{b : \{\}\}\} \cup \{b : \{\}\}$. Clearly, both $f(I)$ and $\tilde{f}(I)$ are not empty.

5.1.2 Complement of structural recursions in $SR(n.i., i., el)$.

As in the previous case the conditional form is defined first. For transformation rules without else-branches the construction works in the same way. Let $\gamma_{i,a}$ be a transformation rule with an else-branch. Then $\gamma^{cf}_{i,a}$ is defined as follows:

$$\begin{aligned} Form(\gamma^{cf}_{i,a}) = & (Form(\gamma_{i,a}) \wedge FoForm_{th}(\gamma_{i,a})) \vee \\ & (\neg Form(\gamma_{i,a}) \wedge FoForm_{el}(\gamma_{i,a})) \\ Frst(\gamma^{cf}_{i,a}) = & \{\psi : \{\}\} \end{aligned}$$

Here, $FoForm_{th}(\gamma_{i,a})$, $FoForm_{el}(\gamma_{i,a})$ respectively denote the forest formula belonging to the then- and else-branch. The rationale behind this rewriting rule is again obvious, in $Form(\gamma^{cf}_{i,a})$ the possible ways of construction are enumerated. A construction is done, if either the condition is satisfied and in the then-branch a non-empty graph is constructed, or the condition becomes $false$ and the else-branch results a non-empty output.

Note that in the previous subsection the forest of a formula was not defined for that case when no structural function is called in the then- or else-branch. Thus, in order to be able to interpret the preceding formula in these cases, this definition should be extended. If no structural structural function is called in $Frst(\gamma_{i,a})$, then,

(i) if there is a construction in $Frst(\gamma_{i,a})$, then $FoForm(\gamma_{i,a})$ is $true$,

(ii) otherwise $FoForm(\gamma_{i,a})$ is $false$.

From Proposition 5.1. and 5.8. the subsequent statement straightforwardly follows.

Proposition 5.10. *Let f be a structural recursion in $SR(n.i., i., el)$ and I an instance. Then $f(I)$ is not empty $\Leftrightarrow$ $f^{cf}(I)$ is not empty.*

Corollary 5.11. *For an arbitrary structural recursion f in $SR(n.i., i., el)$ there is a structural recursion g in $SR(n.i., i.)$ s.t. $\mathcal{L}(f) = \mathcal{L}(g)$.*

Here, remember that $\mathcal{L}(f)$ denotes the set of instances for which f returns a non-empty output.

Corollary 5.12. *Every structural recursion can be rewritten into CCNF.*

Proof. Note that the conditional form is without else-branches. Hence using the result of Proposition 5.4. it can be transform into a structural recursion with a single initial structural function. ■

Thus, when the complement of a structural recursion with else-branches is to be constructed, first it should be transformed into CCNF form. This is without else-branches, hence the rules for constructing the complement of structural recursions in $SR(n.i., i.)$ can be applied.

5.2 Intersection, union

First, the intersection and union of transformation rules should be introduced. Let $\gamma_{i,\vartheta_1}, \gamma_{j,\vartheta_2}$ be transformation rules belonging to structural recursions in conditional forms, where $\vartheta_1, \vartheta_2 \in \Sigma \cup \{*\}$. If ϑ_1 and ϑ_2 are different symbols from Σ, then neither the intersection $\gamma_{i \sqcap j,\vartheta}$ nor the union $\gamma_{i \sqcup j,\vartheta}$ is defined. Otherwise the construction rules of $\gamma_{i \sqcap j,\vartheta}$ and $\gamma_{i \sqcup j,\vartheta}$ are as follows:

$$Form(\gamma_{i \sqcap j,\vartheta}) = Form(\gamma_{i,\vartheta_1}) \wedge Form(\gamma_{j,\vartheta_2}),\ Frst(\gamma_{i \sqcap j,\vartheta}) = Frst(\gamma_{i,\vartheta_1}) \sqcap Frst(\gamma_{j,\vartheta_2})$$
$$Form(\gamma_{i \sqcup j,\vartheta}) = Form(\gamma_{i,\vartheta_1}) \vee Form(\gamma_{j,\vartheta_2}),\ Frst(\gamma_{i \sqcup j,\vartheta}) = Frst(\gamma_{i,\vartheta_1}) \sqcup Frst(\gamma_{j,\vartheta_2}).$$

Here, as before, if $Form(\gamma)$ is not given, then it is taken to be *true*. Since both $\gamma_{i,\vartheta_1}, \gamma_{j,\vartheta_2}$ belong to structural recursions in conditional forms, $Frst(\gamma_{i,\vartheta_1})$ and $Frst(\gamma_{j,\vartheta_2})$ either consist of a single ψ edge or it is the empty graph.

$$Frst(\gamma_{i,\vartheta_1}) \sqcap Frst(\gamma_{j,\vartheta_2})$$

is a ψ edge, if both $Frst(\gamma_{i,\vartheta_1})$ and $Frst(\gamma_{j,\vartheta_2})$ are ψ edges, and it is the empty graph otherwise. On the other hand

$$Frst(\gamma_{i,\vartheta_1}) \sqcup Frst(\gamma_{j,\vartheta_2})$$

is a ψ edge, if $Frst(\gamma_{i,\vartheta_1})$ or $Frst(\gamma_{j,\vartheta_2})$ is a ψ edge, and it is the empty graph otherwise. ϑ is taken to be a, if ϑ_1 or ϑ_2 stand for a, and $*$ otherwise (remember that if both ϑ_1 and ϑ_2 are different from $*$, then in this case they denote the same symbol).

Let $f = (\{f_1, \ldots, f_n\}, \Sigma, \{f_i\}, \Gamma_f)$ and $g = (\{g_1, \ldots, g_m\}, \Sigma, \{g_j\}, \Gamma_g)$ be structural recursions in CCNF and in $SR(n.i., i., el)$. The intersection of f and g is as follows:

$$f \sqcap g = (\{f_{i\sqcap j}, f_1, \ldots, f_n, g_1, \ldots, g_m, \Sigma, \{f_{i\sqcap j}\}, \Gamma\}).$$

As the notation reveals all structural functions of f and g are kept and only one structural function $f_{i\sqcap j}$ is added, which is defined to be the only initial structural function. The transformation rules belonging to $f_{i\sqcap j}$ are constructed by taking the intersections of the transformation rules of the initial structural functions of f and g, f_i and g_j. First, we consider the transformation rules $\gamma^f_{i,\vartheta}$ of f_i one after the other ($\vartheta \in \Sigma \cup \{*\}$). If $\vartheta = a$, then if there exists transformation rule $\gamma^g_{j,a}$, then $\gamma_{i\sqcap j,a}$ is taken to be the intersection of these two transformation rules. Otherwise $\gamma^f_{i,a}$ should be intersected with $\gamma^g_{j,*}$. If $\vartheta = *$, then $\gamma^f_{i,*}$ should be intersected with $\gamma^g_{j,*}$. Remember that both f and g are in CCNF, hence $\gamma^f_{i,*}$, $\gamma^g_{i,*}$ surely exist. Next, we consider those transformation rules $\gamma^g_{j,b}$ of g_j that have not been coupled with any of the transformation rules belonging to f_i yet ($b \in \Sigma$). Each of these transformation rules should be intersected with $\gamma^f_{i,*}$.

Example 5.13. Suppose that the transformation rules belonging to f_i are $\gamma^f_{i,a}$, $\gamma^f_{i,*}$, while the transformation rules belonging to g_j are $\gamma^g_{j,a}$, $\gamma^g_{j,b}$, $\gamma^g_{j,*}$. Then the transformation rules of $f_{i\sqcap j}$ are: $\gamma_{i\sqcap j,a}$, $\gamma_{i\sqcap j,b}$, $\gamma_{i\sqcap j,*}$.

The union of f and g

$$f \sqcup g = (\{f_{i\sqcup j}, f_1, \ldots, f_n, g_1, \ldots, g_m, \Sigma, \{f_{i\sqcup j}\}, \Gamma\})$$

is defined in a similar way. Here, however, instead of the intersections of the transformation rules of f_i and g_j the unions should be considered.

Proposition 5.14. *Let f and g be two structural recursions and I an arbitrary instance, then*

(i) $(f \sqcup g)(I)$ *is not empty* $\Leftrightarrow$ $f(I)$ *or* $g(I)$ *is not empty.*

(ii) $(f \sqcap g)(I)$ *is not empty* $\Rightarrow$ $f(I)$ *and* $g(I)$ *are not empty.*

(iii) If I is root-edged, then $f(I)$ and $g(I)$ are not empty $\Leftrightarrow$ $(f \sqcap g)(I)$ *is not empty.*

Proof. The statements can be proven by using a straightforward induction on the number of steps of the condition evaluation in $U_{f\sqcup I} \sqcap I$ and $U_{f\sqcap I} \sqcap I$. ■

Denote $\mathcal{D}^{\Sigma}_{root}$ the class of root-edged instances, whose edge labels are from Σ. A subclass $\mathcal{H}$ of $\mathcal{D}^{\Sigma}_{root}$ is called *structural recursion recognizable*, if there is a structural recursion f s.t. $\mathcal{L}(f)$ is equal to $\mathcal{H}$.

Corollary 5.15. *Let $\mathcal{H}_1, \mathcal{H}_2$ be two subclasses of $\mathcal{D}^{\Sigma}_{root}$ that are structural recursion recognizable. Then the complement of $\mathcal{H}_i$ over $\mathcal{D}^{\Sigma}_{root}$, $\mathcal{H}_1 \cup \mathcal{H}_2$ and $\mathcal{H}_1 \cup \mathcal{H}_2$ are also structural recursion recognizable ($i = 1, 2$).*

Remark 5.16. Note that in the general case (I is not necessarily root-edged) the non-emptiness of $f(I)$ and $g(I)$ does not necessarily entail the non-emptiness of $(f \sqcap g)(I)$. Namely, it may happen that for $I = t_1 \cup t_2$, $f(t_1)$ is non-empty, $f(t_2)$ is empty, while $g(t_1)$ is empty and $g(t_2)$ is not empty. In this case $(f \sqcap g)(I)$ is obviously empty. In fact, there are cases, when there does

not exist any instance on which $f \sqcap g$ would return a non-empty result. As an example consider $f = (\{f_1, f_2\}, \Sigma, \{f_1\}, \Gamma)$ and $g = (\{g_1, g_2\}, \Sigma, \{g_1\}, \Gamma)$, where the transformation rules are the following:

$$\begin{aligned} f_1\colon (\{a:t\}) &= \text{if n.i.}(f_2(t)) \text{ then } \{\psi:\{\}\} \quad f_2\colon (\{b:t\}) = \{\psi:\{\}\} \\ (\{b:t\}) &= \text{if i.}(f_2(t)) \quad \text{then } \{\psi:\{\}\} \qquad (\{*:t\}) = \{\} \\ (\{*:t\}) &= \{\} \end{aligned}$$

$$\begin{aligned} g_1\colon (\{a:t\}) &= \text{if i.}(g_2(t)) \quad \text{then } \{\psi:\{\}\} \quad g_2\colon (\{b:t\}) = \{\psi:\{\}\} \\ (\{b:t\}) &= \text{if n.i.}(g_2(t)) \text{ then } \{\psi:\{\}\} \qquad (\{*:t\}) = \{\} \\ (\{*:t\}) &= \{\}. \end{aligned}$$

It is easy to see f constructs on instance I iff its root has an outgoing a edge followed by a b edge, or it has an outgoing b edge without any b labelled children. On the other hand, g constructs on I iff its root has an outgoing a edge without any b labelled children or it has an outgoing b edge with a b labelled children.

The transformation rules of $f_{1\sqcap 1}$ are:

$$\begin{aligned} f_{1\sqcap 1}\colon (\{a:t\}) &= \text{if n.i.}(f_2(t)) \wedge \text{i.}(g_2(t)) \text{ then } \{\psi:\{\}\} \\ (b:t\}) &= \text{if i.}(f_2(t)) \vee \text{n.i.}(g_2(t)) \text{ then } \{\psi:\{\}\} \\ (\{*:t\}) &= \{\}. \end{aligned}$$

Clearly, since f_2 and g_2 are the same, there is not any instance that would result a non-empty output for $f \sqcap g$. On the other hand, for instance $I = \{a:\{b:\{\}\}\} \cup \{b:\{b:\{\}\}\}$, $f(I)$ and $g(I)$ are both non-empty.

We conclude this section by formulating the De Morgan's laws for structural recursions.

Proposition 5.17. *Let*

$$f = (\{f_1, \ldots, f_n\}, \Sigma, F_I, \Gamma_f) \text{ and } g = (\{g_1, \ldots, g_m\}, \Sigma, G_I, \Gamma_g)$$

be two structural recursions. Then

(i) $\widetilde{f \sqcup g}$ *is equivalent to* $\tilde{f} \sqcap \tilde{g}$.

(ii) $\widetilde{f \sqcap g}$ *is equivalent to* $\tilde{f} \sqcup \tilde{g}$.

A stronger result will be proven. Namely, it will be shown that the appropriate structural recursions are not only equivalent but they are essentially the same syntactically. The proof can be found in the Appendix (A 5.2.).

6 Structural recursions and automata

In this section structural recursions are compared with two different types of automata. Firstly, the relationship between NDFSA-s and structural recursions in $SR()$ are examined in Section 6.1. In [9] it has been already shown how simple structural recursions can be simulated by NDFSA-s. Nevertheless, since the reverse direction of the simulation works in a very similar way, it is useful to give this rewriting here as well. It will turn out that NDFSA-s cannot be simulated by simple structural recursions, however, if we allow the application of a special structural function in whose transformation rules a single i. condition is used, then the simulation can be accomplished. The reason of this failure is that the acceptance by an NDFSA is not necessarily a monotonous property. Namely, it may happen that an NDFSA accepts a word w whereas it does not accept another word containing w as a prefix. On the other hand, as it has been shown in Proposition 4.27. resulting a non-empty output for a given structural recursion is a monotonous property, when the structural recursion in question is in $SR()$.

Secondly, structural recursions will be represented by alternating tree automata and vice versa. In this case the simulations are less trivial.

6.1 Structural recursions in $SR()$ and NDFSA

6.1.1 Rewriting of a structural recursion in $SR()$ into an NDFSA

Structural recursions process data graphs, while NDFSA-s work on strings, thus it may seem illegitimate to try to simulate the first with the second.

However, Lemma 4.18. shows that each instance that results a non-empty output for a structural recursion in $SR()$ contains a path as a pregraph which also returns a non-empty result. In other words, a simple structural recursion can be characterized by paths in terms of acceptance. Clearly, there is a straightforward correspondence between paths and strings. For path

$$pa = (u_1, a_1, v_1) \dots (u_n, a_n, v_n), w_{pa} := a_1 \dots a_n, \text{ reversely, for a word}$$
$$w = b_1 \dots b_m, pa_w := (u_1, b_1, v_1) \dots (u_m, b_m, v_m),$$

where $(u_i, a_i, v_i) \in E.pa, (u_j, b_j, v_j) \in E.pa_w$ $(1 \leq i \leq n, 1 \leq j \leq m)$. This justifies at least the *raison d'être* of such a representation. In fact as it was indicated the method of simulation is quite simple.

Let $f = (\{f_1, \dots, f_n\}, \Sigma, F_I, \Gamma)$ be a simple structural recursion. The corresponding string automaton is as follows:

$$A_f = (Q^f, \Sigma_f \cup \{\S\}, Q_I^f, Q_f^f, \Phi^f).$$

Here, remember that

$$\Sigma_f = \{a \mid a \in \Sigma, \exists i \text{ s.t. } \gamma_{i,a} \in \Gamma\},$$

in other words Σ_f consists of those symbols of Σ for which at least one transformation rule is given in f. Besides, § is a symbol that is different from all elements of Σ.

$$Q^f = \{q_1, \dots q_n, q_{deny}, q_{accept}\} \text{ and } Q_f^f = q_{accept}, \text{ where}$$

as the notation reveals q_{accept} is the only acceptor state of A_f, while q_{deny} is a special state s.t. whenever A_f gets into this state the processed input will be surely rejected. Besides, to each f_i state q_i is assigned $(1 \leq i \leq n)$. Q_I^f is defined as follows:

$$Q_I^f := \{q_i \mid f_i \in F_I, 1 \leq i \leq n\}.$$

For the definition of the transition rules of A_f the transformation rules of f should be considered one after the other. For transformation rule

$$f_i : (\{\theta : t\}) = frst,\ frst \in \mathcal{F}^{\Delta \cup \{*\}}(L), \theta \in \{a, *\}$$

we add transition rule $\vartheta(q_i) \to q_j$ to Φ^f, if f_j occurs among the leaf labels of $frst$ $(1 \leq i, j \leq n)$. If $\theta = *$ (the default case), then $\vartheta = \S$. Otherwise θ and ϑ denotes the same symbol of Σ. Here i, j are not necessarily different from each other. If $frst$ contains a constructor edge, then we extend Φ^f with $\vartheta(q_i) \to q_{accept}$. On the other hand, if there is no constructor edge in $frst$ and none of its leaves are labelled, $\vartheta(q_i) \to q_{deny}$ should be added to Φ^f. The transition rules for q_{accept} and q_{deny} are as follows:

$$a(q_{accept}) \to q_{accept},\ a(q_{deny}) \to q_{deny}\,(a \in \Sigma_f \cup \{\S\}).$$

Example 6.1. For structural recursion $f = (\{f_1, f_2\}, \Sigma, \{f_1\}, \Gamma)$ whose transformation rules are:

f_1: $(\{a : t\}) = \{a : f_1(t)\}$ $\quad$ f_2: $(\{b : t\}) = \{\}$
$\quad$ $(\{* : t\}) = f_2(t)$ $\quad\quad$ $(\{* : t\}) = f_1(t)$

the transition rules of A_f are as follows:

$a(q_1) \to q_{accept}$ $\quad$ $a(q_1) \to q_1$ $\quad$ $\S(q_1) \to q_2$
$b(q_2) \to q_{deny}$ $\quad$ $\S(q_2) \to q_1$.

Proposition 6.2. *For given structural recursion in $SR()$ f and instance I, $f(I)$ is not empty $\Leftrightarrow$ I contains a path pa as a pregraph s.t. A_f accepts w_{pa}.*

The proof of the proposition are to be found in the Appendix (A 6.1.1.).

6.1.2 Rewriting of an NDFSA into a structural recursions

The simulation of a string automata with a simple structural recursion works in a similar way. Let $A = (Q, \Sigma_A, Q_I, Q_f, \Phi)$ be an arbitrary string automaton, where $Q = \{q_1, \ldots q_n\}$. Note that Σ_A is a finite alphabet here. Then

$f_A = (F^A, \Sigma, F_I^A, \Gamma^A)$ is defined in the following way. For each state q_i we add f_i to F^A. If q_i is an initial state, then f_i will also be an initial function. For a given state q_i and a symbol a in Σ, denote $q_{i_1}, \ldots q_{i_m}$ the set of those states that appear on the right side of the transition rules for $a(q_i)$. The transformation rule belonging to f_i and a is defined as follows:

$$f_i : (\{a : t\}) = f_{i_1}(t) \cup \ldots \cup f_{i_m}(t) \text{ or } (\{a : t\}) = \{\Psi : \{\}\}.$$

In the first case, there is not any acceptor state among $q_{i_1}, \ldots q_{i_m}$, in the second case there is at least one.

Lemma 6.3. *Keeping the preceding notations, if for a given word $w = a_1 \ldots a_m$, there exists a run $\lambda = q_{k_0}, q_{k_1}, \ldots, q_{k_m}$ of A on w s.t. g_{k_j} is not an acceptor state for $j < i \Leftrightarrow$*

(i) if q_{k_i} is not an acceptor state, $((f_{k_{i-1}}, u_i), a_i, (f_{k_i}, v_i)) \in E.U_{f_A} \sqcap pa_w$,

(ii) if q_{k_i} is an acceptor state, $((f_{k_{i-1}}, u_i), a_i, (w_{end}, v_i)) \in E.U_{f_A} \sqcap pa_w$ and $f_A(pa_w)$ is not-empty $(1 \leq i \leq m)$.

Proof. We prove the statement by using induction on i. Suppose that $i = 1$. From the definition of a run of a string automation (Page 9) we know that q_{k_0} is an initial state, hence f_{k_0} is an initial structural function. Assume first that run λ exists. Since $a_1(q_{k_0}) = q_{k_1} \in \Phi$, γ_{k_0,a_1} is:

$$(\{a_1 : t\}) = \ldots \cup f_{k_1} \cup \ldots \text{ or } (\{a_1 : t\}) = \{\Psi : \{\}\},$$

depending on whether q_{k_1} is an acceptor state or not. Straightforwardly,

$$((f_{k_0}, u_1), a_1, (f_{k_1}, v_1)) \in E.U_{f_A} \sqcap pa_w \text{ or } ((f_{k_0}, u_1), a_1, (w_{end}, v_1)) \in E.U_{f_A} \sqcap pa_w.$$

In the second case $f_A(pa_w)$ is not-empty. The proof of the other direction is again similar and obvious. The general case works in the same manner, thus we omit the details of this part of the proof. ■

From the lemma the next proposition obviously follows.

Proposition 6.4. *Let A be a string automaton with alphabet Σ and $w \in \Sigma^*$ a word. If A accepts w, then $f_A(pa_w)$ is not empty.*

It is easy to see that the reverse direction of the proposition does not hold. Namely, if during the process of w A reaches an acceptor state, then no matter whether this process continues or not f_A constructs an edge on pa_w.

In order to make the simulation complete i. conditions should be applied. First a structural function f_{next} is defined, which constructs an edge ψ for all elements of Σ_A, where remember Σ denotes the finite alphabet over which A works:

$$f_{next} : (\{a : t\}) = \{\psi : \{\}\},\ a \in \Sigma_A.$$

By means of f_{next} we will test whether the edge that is being processed is without outgoing edges, i.e., is it the last edge of a branch, or not. Thus, the simulating structural recursion will only construct, when it reaches an acceptor state and it is processing the last edge of the input path. Formally, the transformation rules of f_A should be changed in the following way, when there is an acceptor state among $q_{i_1}, \ldots, q_{i_m}$ (where q_{i_j}-s are the set of states that occur on the right side of the transition rules for $a(q_i)$):

$$\gamma_{f_i,a} : (\{a : t\}) = \text{if i.}(f_{next}(t)) \text{ then } \{\psi : \{\}\} \text{ else } f_{i_1}(t) \cup \ldots \cup f_{i_m}(t).$$

Denote $f_{A,i.}$ the resulting structural recursion. Using the previous reasonings the next proposition can be proven in a straightforward way.

Proposition 6.5. *Let A be a string automaton with alphabet Σ_A and $w \in \Sigma_A^*$ a word. Then A accepts $w \Leftrightarrow f_{A,i.}(pa_w)$ is not empty.*

6.2 Alternating tree automata and structural recursions

6.2.1 Rewriting structural recursion to alternating tree automata

If one is to rewrite a structural recursion to an alternating tree automaton, roughly two main differences should be taken into account. First of all a

connection should be established between the two types of inputs on which the two formalisms work. Namely, edge-labelled trees over unranked, infinite alphabets should be represented with node-labelled trees, where the labels are taken from a ranked finite alphabet. Secondly, structural functions are called on all branches of a data graph, while in an alternating tree automaton states are usually "called" only on certain branches.

It is not possible to offer an appropriate ranked alphabet for a given unranked alphabet in general. What we can do is to define this correspondent for each structural recursion separately. Let $f = (\{f_1, \ldots, f_n\}, \Sigma, F_I, \Gamma)$ be a structural recursion. Remember that Σ_f has been defined as:

$$\Sigma_f = \{a \mid a \in \Sigma, \exists i \text{ s.t. } \gamma_{i,a} \in \Gamma\}.$$

In Σ_f^{ext} we extend Σ_f with three additional symbols each of them is different from all elements of Σ:

$$\Sigma_f^{ext} := \Sigma_f \cup \{\S, a_{acc}, a_{deny}\}.$$

Here, $\S$ is introduced to be able to represent the default case. On he other hand, a_{acc}, a_{deny} are added so as to represent acceptance and refusal in the simulations by automata. In these simulations we will only consider special trees, whose inner nodes that are different from the root have the same number of outgoing edges. This number m will depend on f. Besides, the leaf edges will be labelled with a_{acc} or a_{deny}. The set of these trees is denoted by $\mathcal{T}^{\Sigma_f^{ext},m}$. The corresponding ranked alphabet $\Upsilon_f = (\Sigma_f^{ext}, Arity_f)$ is defined as follows:

(i) $Arity_f(a) = 0$, if $a \in \{a_{acc}, a_{deny}\}$.

(ii) $Arity_f(a) = m$, $a \in \Sigma_f^{ext} \setminus \{a_{acc}, a_{deny}\}$.

Next a transformation should be introduced from edge-labelled trees to node-labelled ones. Mapping $\phi_{edge \to node} : \mathcal{T}_{edge}^{(\Sigma_f^{ext},m)} \to T_{node}^{\Upsilon_f}$ is defined in the following way:

(i) $\phi_{edge\to node}(\{a:\{\}\}) = a$, here $a \in \{a_{acc}, a_{deny}\}$, thus $Arity(a) = 0$,

(ii) $\phi_{edge\to node}(\{b : t_1 \cup \ldots \cup t_m\}) = b(\phi_{edge\to node}(t_1) \ldots \phi_{edge\to node}(t_m))$, here $Arity(b) = m$.

In the following step it is described how an arbitrary instance resulting a non-empty output for f should be represented by a set of trees in $\mathcal{T}^{\Sigma_f^{ext},m}$. Suppose that f is still of the following form: $(\{f_1, \ldots, f_n\}, \Sigma, F_I, \Gamma)$. By Proposition 4.2. we may assume that f is complete. Furthermore, let I be an arbitrary instance s.t. $f(I)$ is not empty. Consider a path pa from the root to a constructor edge in $U_f \sqcap I$, whose edges are kept during condition evaluation. Apply the `Tree_Simulator` algorithm (Figure 14.) to pa. Then from Proposition 4.45. we know that $f(pa_{tree})$ is not empty. In the next step change those edge labels of pa_{tree} to § that are not in Σ_f. Clearly, since § is not in Σ_f, in each structural function the default transformation rules are called for both the original and the relabelled edges, thus f still results a non-empty output. Next, substitute each default transformation rule $\gamma_{i,*}$ of f with $\gamma_{i,\S}$ in which only the $*$ symbols are changed to § symbols ($1 \leq i \leq n$). Clearly, this new structural recursion still results a not-empty output for the relabelled pa_{tree}. In the rest of of this subsection we will always assume that structural recursions and instances are relabelled in this way.

Recall that the edges of pa_{tree} are taken from $U_f \sqcap I$. According to Remark 4.47. for each node (f_i, u) and symbol a

$$Out(pa_{tree}, a, (f_i, u)) \leq Out(U_f, a, f_i),$$

where remember $Out(G, a, u)$ denotes the number of outgoing a edges from node u of graph G. Clearly, this also holds for the relabelled versions of pa_{tree} and U_f.

$$M_f := \max_{a \in \Sigma_f \cup \{\S\}} \{ \max_{u \in V.U_f} \{Out(U_f, a, u)\}\}.$$

Obviously, $Out(pa_{tree}, a, (f_i, u)) \leq M_f$ for each node and symbol. Using this upper limit we complete pa_{tree} s.t. each of its node will have the same number of outgoing edges except from the root. Note that the root has only

one outgoing edge now (Remark 4.46.), and no other outgoing edge should be added, otherwise we could not rewrite the resulting graph into a node-labelled tree. For each node u of pa_{tree} different from the root and for each label a in $\Sigma_f \cup \{\S\}$ add $M_f - Out(pa_{tree}, a, u)$ number of a_{deny} edges to u. Here, $Out(pa_{tree}, a, u) = 0$, if u does not have any outgoing a edge. After this step, straightforwardly, each node (except from the root), including the former leaves, has $m_f = M_f|\Sigma_f \cup \{\S\}|$ outgoing edges. In order to able to simulate construction (acceptance) by a tree automata to each node of pa_{tree} (different from the root) we add an a_{acc} edge in the $(m_f + 1)^{th}$ position. An a_{deny} edge is also added in the $(m_f + 2)^{th}$ position. With these latter edge we will handle those cases, when in a transformation rule nothing is constructed and no other structural function is called. Denote pa_{aut} this new tree. Obviously, pa_{aut} is in $\mathcal{T}^{\Sigma_f^{ext}, m_f+2}$. As an example consider Figure 15.(a)-(b). Here $\Sigma_f = \{a, b\}$ and $M_f = 2$.

The following definition collects those properties that are crucial for the successful simulation.

Definition 6.6. *A root-edged tree in $\mathcal{T}^{\Sigma_f^{ext}, m_f+2}$ is called f* simulation tree,

(i) if each of the $(m_f + 1)^{th}$ child edges is labelled with a_{acc}, while the $(m_f + 2)^{th}$ child edges are labelled with a_{deny}.

(ii) If a node has solely a_{acc} and a_{deny} labelled child edges, then the label of its first m_f child edges is a_{deny}.

To complete this construction we add transformation rules $\gamma_{i,a_{acc}}, \gamma_{i,a_{deny}}$ to each structural function of f, which are defined as follows $(1 \leq i \leq n)$:

$$Form(\gamma_{i,\vartheta}) \text{ is not given, } Frst(\gamma_{i,\vartheta}) = \{\}, \ \vartheta \in \{a_{acc}, a_{deny}\}.$$

This means that whenever a structural functions of f processes an edge a_{acc} or a_{deny} the computation stops on that branch. Denote f_{aut} this new structural recursion. Clearly, $f_{aut}(pa_{aut})$ is still not empty. Now

$\mathcal{I}_{f,I} := \{pa_{aut} \mid pa$ is a path in $U_f \sqcap I$ from the root to a constructor edge after condition evaluation.$\}$

Note that if there is a cycle in $U_f \sqcap I$, then $\mathcal{I}_{f,I}$ is not necessarily a finite set. The following lemma summarizes our results up till now.

Lemma 6.7. *Let f be a structural recursion and I an instance. Then $f(I)$ is not empty $\Leftrightarrow$ there exists a tree pa_{aut} in $\mathcal{I}_{f,I}$ s.t. $f(pa_{aut})$ is not empty.*

Recall again that by Remark 4.46. the results of `Tree_Simulator` algorithm are always root-edged. Thus, each element of $\mathcal{I}_{f,I}$ is root-edge, so it can be transformed into a node-labelled tree. For pa_{aut}, $\phi_{edge\to node}(pa_{aut})$ is denoted by pa_{aut}^{node}.

Construction of the simulating automata. For structural recursion $f = (\{f_1, \ldots, f_n\}, \Sigma, F_I, \Gamma)$ the corresponding alternating tree automaton will be denoted by $A_f = (Q_f, \Upsilon_f, Q_I^f, \Psi_f)$. Here, ranked alphabet $\Upsilon_f = (\Sigma_f^{ext}, Arity_f)$ has been already defined at the beginning of this subsection. Remember that

(i) $Arity_f(a) = 0$, if $a \in \{a_{acc}, a_{deny}\}$,

(ii) $Arity_f(a) = m_f + 2$, $a \in \Sigma_f \setminus \{a_{acc}, a_{deny}\}$.

In the construction we suppose that f is in CCNF. In Q_f for each structural function f_i we assign two states q_i and $\tilde{q}_i$ $(1 \leq i \leq n)$. They will represent f_i, when it is called in a *not-isempty*, *isempty* condition respectively. Accordingly, the first set of states will be referred as *not-isempty*, while the second as *isempty* states. Furthermore, q_i is in Q_I^f, if f_i is in F_I. For the definition of transitional rules of Ψ_f, consider a transformation rule $\gamma_{i,a}$ with a condition. For $Form(\gamma_{i,a})$ the corresponding formula $Form_{aut}(\gamma_{i,a})$ is defined as follows $(1 \leq j \leq n)$:

(i) each occurrence of a structural function in a not-isempty condition, i.e, n.i.$(f_j(t))$, should be substituted with $(q_j, 1) \vee \ldots \vee (q_j, m_f)$.

(ii) On the other hand, i.$(f_j(t))$ should be substituted with $(\tilde{q}_j, 1) \wedge \ldots \wedge (\tilde{q}_j, m_f)$.

This means that in the simulation of a (not-)isempty condition, the appropriate state will be called on all branches except from the last two edges: a_{acc} and a_{deny}. The negation of $Form_{aut}(\gamma_{i,a})$ is constructed in a similar way. The former rewriting rules should be applied to $\neg Form(\gamma_{i,a})$. As an example, suppose that $Form(\gamma_{i,a})$ is

$$\text{n.i.}(f_1(t)) \wedge (\text{i.}(f_2(t)) \vee \text{n.i.}(f_3(t)))$$

and $m_f = 2$, then

$$Form_{aut}(\gamma_{i,a}) = ((q_1,1) \vee (q_1,2)) \wedge (((\tilde{q}_2,1) \wedge (\tilde{q}_2,2)) \vee ((q_3,1) \vee (q_3,2)))$$
$$\neg Form_{aut}(\gamma_{i,a}) = ((\tilde{q}_1,1) \wedge (\tilde{q}_1,2)) \vee (((q_2,1) \vee (q_2,2)) \wedge ((\tilde{q}_3,1) \wedge (\tilde{q}_3,2)))$$

Now, for transformation rule $\gamma_{i,a}$ the corresponding transitional rules are defined in the following way ($a \in \Sigma_f \cup \{\S\}$):

(i) if there is a condition in $\gamma_{i,a}$, then $(q_i, a) \to Form_{aut}(\gamma_{i,a})$ and $(\tilde{q}_i, a) \to \neg Form_{aut}(\gamma_{i,a})$.

(ii) If there is no condition in $\gamma_{i,a}$ and a ψ edge is constructed, then $(q_i, a) \to (q_i, m_f + 1)$, $(\tilde{q}_i, a) \to (\tilde{q}_i, m_f + 1)$.

(iii) If there is no condition in $\gamma_{i,a}$ and nothing is constructed, then $(q_i, a) \to (q_i, m_f + 2)$, $(\tilde{q}_i, a) \to (\tilde{q}_i, m_f + 2)$ $(1 \leq i \leq n)$.

Note that in case (ii), where a construction without condition is to be represented both q_i and $\tilde{q}_i$ are called on the $m_f + 1$ branch, which is a single a_{acc} edge. Obviously, q_i, since it represents the non-emptiness of f_i should "accept" this branch, while $\tilde{q}_i$ should refuse it. In case (iii), where an empty result is returned, the aforementioned states are called on an a_{deny} edge. It goes without saying that here q_i should refuse and $\tilde{q}_i$ should accept. Furthermore, remember that if a node of pa_{aut} has solely a_{acc} and a_{deny} labelled edges, then the label of the first m_f of edges is a_{deny}. If a process reaches such an a_{deny} edge, then it means that nothing is to be constructed on the

corresponding branch. However, this just fits to our previous observation that $\tilde{q}_i$ should accept a_{deny}, whereas q_i should refuse it. Hence, for a_{acc} and a_{deny} the transitional rules are as follows ($1 \leq i \leq n$):

$$(q_i, a_{acc}) \to true,\ (q_i, a_{deny}) \to false,$$
$$(\tilde{q}_i, a_{acc}) \to false,\ (\tilde{q}_i, a_{deny}) \to true.$$

For $A_f = (Q_f, \Upsilon_f, \{q_i\}, \Psi_f)$ define $\tilde{A}_f$ to be $(Q_f, \Upsilon_f, \{\tilde{q}_i\}, \Psi_f)$, i.e., only the set of initial states is changed. The following two lemmas are only formulated here, owing to their intricacy the proofs are moved to the Appendix (A 6.2.1.).

Lemma 6.8. *For an arbitrary structural recursion f $\tilde{A}_f$ is the complement of A_f.*

Lemma 6.9. *Let $f = (\{f_1, \ldots, f_n\}, \Sigma, \{f_i\}, \Gamma)$ be a structural recursion and t an f simulation tree, then $f(t)$ is not empty $\Leftrightarrow$ A_f accepts $t^{node} = \phi_{edge \to node}(t)$.*

Theorem 6.10. *Let $f = (\{f_1, \ldots, f_n\}, \Sigma, F_I, \Gamma)$ be a structural recursion and I an instance. Then $f(I)$ is not empty $\Leftrightarrow$ for each tree t in $\mathcal{I}_{f,I}$ A_f accepts t^{node}.*

Proof. The theorem is a straightforward consequence of Lemma 6.7. and 6.9. and the fact that each element of $\mathcal{I}_{f,I}$ is an f simulation tree. ■

6.2.2 Rewriting alternating tree automata to structural recursion

Transformation of the input trees. If we are to simulate an alternating tree automaton with a structural recursion, the first difficulty we should cope with is that node-labelled trees over a finite ranked alphabet should be transformed into edge-labelled tree over an unranked alphabet. Contrary to the reverse transformation this can be achieved in a quite similar manner. To a given ranked alphabet $\Upsilon = (\Omega, Arity)$ we assign Ω as an unranked alphabet. Then mapping $\phi_{node \to edge} : T^{\Upsilon}_{node} \to T^{\Omega}_{edge}$ is recursively defined as follows:

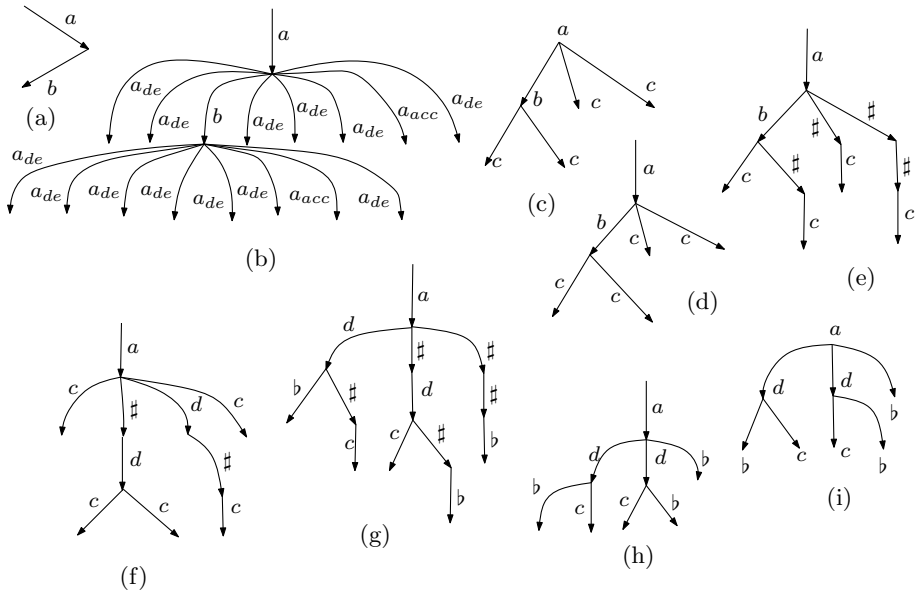

Figure 15: (a) A data graph. (b) The f-simulation tree constructed from the data graph of (a). Here a_{de} abbreviates a_{deny}, $\Sigma_f = \{a, b\}, M_f = 3$, hence $m_f = 6$. (c) A node-labelled tree. (d) The result of applying $\phi_{node \to edge}$ to the tree of (c). (e) The result of applying transformation Tr to the tree of (c). (f) A root-edge tree in $\mathcal{T}_{edge}^{\Omega \cup \{\sharp\}}$. (g) An intermediary result in transformation $ReTr$ applied on the tree of (f). (h) The result of deleting the paths of $\sharp$-edges of the data graph of (g). (i) A possible final result of transformation $ReTr$ applied on the tree of (f). This tree is the result of mapping $\phi_{edge \to node}$ accomplished on the data graph of (h).

(i) $\phi_{node\to edge}(a) = \{a : \{\}\}$, where $Arity(a) = 0$,

(ii) $\phi_{node\to edge}(a(t_1 \dots t_n)) = \{a : \phi_{node\to edge}(t_1) \cup \dots \cup \phi_{node\to edge}(t_n)\}$, where $Arity(a) = n$.

Secondly, in transitional rules the states are called on branches given with their numbers which indicate their positions in the order given among the children of a node. This number should be represented in an understandable way for a structural recursion. Informally, the i^{th} branch will be encoded by adding $i-1$ $\sharp$ edges before the first edge of this branch.

Formally, let t be a tree in $\mathcal{T}^{\Upsilon}_{node}$ ($\Upsilon = (\Omega, Arity)$). First apply $\phi_{node\to edge}$ to t. Denote t_{edge} the result. Then define an order $\prec$ on the neighbouring edges of t_{edge} s.t. if node u of t is represented by edge e_u in t_{edge} and v is the j^{th} child of u, then e_v (the representation of v) should be the j^{th} according to $\prec$ among the outgoing edges of e_u. Consider then the nodes of t_{edge} consecutively. For node u consider the outgoing edges one after the other. For the i^{th} edge (u, a, v) take a new node w, add an a edge from w to v and then delete (u, a, v). Next construct a path pa of $i-1$ $\sharp$ edges, add an ε edge from u to the starting node of pa and another ε edge from the end node of pa to w. In the last step eliminate these ε edges. With this construction a path of $i-1$ $\sharp$ edges has been added between u and its i^{th} outgoing edge. Here we assume that $\sharp$ is not in Ω. As an example consider Figure 15.(c)-(e). Here $\Omega = \{a, b, c\}$, $Arity(a) = 3$, $Arity(b) = 2$, $Arity(c) = 0$. In the rest of this subsection with $Tr(t)$ (transformation for structural recursion simulation) we denote the result of the described transformation.

Definition 6.11. *A tree in $\mathcal{T}^{\Omega\cup\{\sharp\}}_{edge}$ is called Υ simulation tree ($\Upsilon = (\Omega, Arity)$), if for an arbitrary symbol a of Ω, where $Arity(a) = m$, each a-labelled edges of this tree has m outgoing edges. Besides, for all i each of these edges has an outgoing path of $i-1$ $\sharp$ edges followed by a non-$\sharp$ edge ($1 \le i \le m$).*

Remark 6.12. Clearly, $Tr(t)$ is an Υ simulation tree, where $t \in \mathcal{T}^{\Upsilon}_{node}$. What is more, since the results of mapping $\phi_{node\to edge}$ are always root-edged, $Tr(t)$ is root-edged.

Construction of the simulating structural recursion. Let

$$A = (Q = \{q_1, \ldots, q_n\}, \Upsilon, Q_I, \Psi)$$

be an alternating tree automaton ($\Upsilon = (\Omega, Arity)$). We may assume that for each state q_i and symbol $a \in \Omega$ there is at least one transitional rule with (q_i, a) left-hand side ($1 \leq i \leq n$). If there is more than one such rule, then we take the disjunction of the formulas on the right sides. On the other hand, if there is no such rule, then we add $(q_i, a) \to false$ to Ψ. Additionally, we may also assume that A has only a single initial state [12]. Suppose on the contrary that q_i, q_j are both initial states. Let q be a new state, which is defined to be the initial state instead of q_i and q_j. Each pair of transition rules $(q_i, a) \to \phi_i$, $(q_j, a) \to \phi_j$ should be substituted with $(q, a) \to \phi_i \vee \phi_j$. When there are more than two initial states, the construction works in a similar way [12].

In the rewriting first the structural functions traversing the paths of $\sharp$ edges are defined. Denote $f^1_{\sharp,j,i}, \ldots, f^j_{\sharp,j,i}$ the set of structural functions that will traverse a path of j $\sharp$ edges and then call f_i. Their transformation rules can be given in an obvious way:

$f^k_{\sharp,j,i}$: $(\{\sharp : t\}) = f^{k+1}_{\sharp,j,i}(t)$ $\quad$ $(\{* : t\}) = \{\}$ $\quad$ $f^j_{\sharp,j,i}$: $(\{\sharp : t\}) = f_i(t)$ $\quad$ $(\{* : t\}) = \{\}$ $(1 \leq k \leq j-1)$.

Next, for each state q_i structural function f_i is assigned ($1 \leq i \leq n$). In the simulation of transitional rules belonging to q_i consider first a transitional rule $(q_i, a) \to \phi$, where $Arity(a) > 0$. In the rewriting of ϕ, substitute each (q_k, s) with n.i.$(f^1_{\sharp,s-1,k}(t))$, and denote $Form(\phi)$ this new formula ($1 \leq k \leq n, 1 \leq s \leq Arity(a)$). In other words, f_k is called after traversing a path of $s-1$ $\sharp$ edges, which, remember, represents the s^{th} branch. Note that if a path contains more or less $\sharp$ edges consecutively than $s-1$, then n.i.$(f^1_{\sharp,s-1,k}(t))$ returns $false$. Here, $f_{\sharp,0,k}$ is taken to be f_k (f_k should be called after 0 $\sharp$ edges). Afterwards $\gamma_{i,a}$ is defined as follows:

$$Form(\gamma_{i,a}) := Form(\phi),\ Frst(\gamma_{i,a}) := \{\psi : \{\}\}.$$

On the other hand, if $Arity(a) = 0$, then

$$Form(\gamma_{i,a}) \text{ is not given, } Frst(\gamma_{i,a}) := \vartheta,$$

where ϑ is a ψ edge, when ϕ is *true*, and it is the empty graph, if ϕ is $false$.

For the formal definition of the *semi-simulating* structural recursion, f_A^{sem}, yet an additional notation should be introduced. (In Example 6.14. it will become clear why this structural recursion is called only semi-simulating instead of simulating.) With $F_\sharp$ we denote the set of structural functions that are used to traverse the paths of $\sharp$ edges.

$$F_\sharp := \{f^k_{\sharp,j,i} \mid 1 \leq j \leq M-1, 1 \leq i \leq n, 1 \leq k \leq j,\}$$

where M denotes the maximal arity of Υ. Now

$$f_A^{sem} \text{ is defined to be } (F_\sharp \cup \{f_1, \ldots, f_n\}, \Omega \cup \{\sharp\}, F_I, \Gamma).$$

As it is usual f_i will be in F_I, if q_i is in Q_I $(1 \leq i \leq n)$. This completes the definition of f_A^{sem}. Note that f_A^{sem} is in CNF, what is more, it is without i. conditions.

Lemma 6.13. *For alternating tree automaton $A = (\{q_1, \ldots, q_n\}, \Upsilon, \{q_i\}, \Phi)$ and tree t in $\mathcal{T}^{\Upsilon}_{node}$, A accepts $t \Leftrightarrow f_A^{sem}(Tr(t))$ is not empty.*

Proof. Assume that $Tr(t)$ is of the form $\{a : t_1 \cup \ldots \cup t_s\}$ $(Arity(a) = s)$. Again, we use induction on the number of steps k in the condition evaluation over $U_{f_A^{sem}} \sqcap Tr(t)$. In the base case $(k = 1)$ suppose first that $f_A^{sem}(Tr(t))$ is not empty. Then, there is no condition in $\gamma_{i,a}$ and a ψ edge is constructed (f_i is the initial structural function of f_A^{sem}, since q_i is the initial state of A). This means that $\gamma_{i,a}$ represents transition rule $(q_i, a) \to true$. Since q_i is the single initial state of A and the root of t is labelled with a, A clearly accepts t. On the other hand, if $f_A^{sem}(Tr(t))$ is empty, then $\gamma_{i,a}$ corresponds to transition rule $(q_i, a) \to false$. In this case, obviously, A refuses t.

Assume now that the statement holds for $k \leq m$ and $k := m + 1$. In the run λ of A on t, transition rule $(q_i, a) \to \phi$ is used to construct the children of

the root. In what follows we show that n.i.$f^1_{\sharp,j-1,k}(t)$ becomes $true$ ($false$) in $Form(\gamma_{i,a})$ if and only if (q_k, j) in ϕ becomes $true$ ($false$) in the evaluation of λ ($1 \leq k \leq n, 1 \leq j \leq Arity(a)$). From this observation the lemma will obviously follow. However, the statement in question is a straightforward consequence of the induction hypothesis. Namely, it is easy to see that f^k, which is defined to be the same structural recursion as f^{sem}_A, except that here f_k is taken to be the only initial structural function, is the rewriting of A^k, which, again, is constructed by changing the initial state of A to q_k. From the induction hypothesis it follows that A^k accepts t^j_{node}, if and only if $f^k(Tr(t_j))$ is not empty. Here, t^j_{node} denotes the reachable subtree from the j^{th} child of the root of t. From this, the statement, which we are to prove, clearly follows. ■

The next example reveals the shortcomings of our simulation at this stage.

Example 6.14. Consider ranked alphabet $\Upsilon = (\{a, b, c\}, Arity)$, where $Arity(a) = Arity(b) = 1$ and $Arity(c) = 0$. Thus, each tree in $\mathcal{T}^{\Upsilon}_{node}$ is a path. The transition rules of alternating tree automaton

$$A = (\{q_1, q_2, q_3, q_4, q_5\}, \Upsilon, \{q_1\}, \Phi)$$

are defined as follows:

$$\begin{array}{lll}
(q_1, a) \to (q_2, 1) \wedge (q_3, 1), & (q_1, b) \to (q_2, 1) \wedge (q_3, 1), & (q_1, c) \to false \\
(q_2, a) \to (q_4, 1), & (q_2, b) \to (q_2, 1), & (q_2, c) \to false \\
(q_3, a) \to (q_5, 1), & (q_3, b) \to (q_3, 1), & (q_3, c) \to true \\
(q_4, a) \to (q_4, 1), & (q_4, b) \to (q_4, 1), & (q_4, c) \to true \\
(q_5, a) \to (q_5, 1), & (q_5, b) \to (q_5, 1), & (q_5, c) \to false.
\end{array}$$

A clearly refuses the single node labelled with c. For the rest of the inputs (their roots are either labelled with a or b) it calls both states q_2 and q_3 on the subtree under the root and both thread should be evaluated to $true$. However, it is easy to see that in the first case this subpath is only accepted, if it contains an a node, while in the second case subtrees with an a

node are refused. Hence A does not accept anything. Consider on the other hand $f_A^{sem} = (\{f_1, f_2, f_3, f_4, f_5\}, \Omega, \{f_1\}, \Gamma\})$, whose transformation rules are as follows:

f_1: $(\{a:t\}) =$ if n.i.$(f_2(t)) \wedge$ n.i.$(f_3(t))$ then $\{\psi : \{\}\}$
$(\{b:t\}) =$ if n.i.$(f_2(t)) \wedge$ n.i.$(f_3(t))$ then $\{\psi : \{\}\}$
$(\{c:t\}) = \{\}$

f_2: $(\{a:t\}) =$ if n.i.$(f_4(t))$ then $\{\psi : \{\}\}$
$(\{b:t\}) =$ if n.i.$(f_2(t))$ then $\{\psi : \{\}\}$
$(\{c:t\}) = \{\}$

f_3: $(\{a:t\}) =$ if n.i.$(f_5(t))$ then $\{\psi : \{\}\}$
$(\{b:t\}) =$ if n.i.$(f_3(t))$ then $\{\psi : \{\}\}$
$(\{c:t\}) = \{\psi : \{\}\}$

f_4: $(\{a:t\}) =$ if n.i.$(f_4(t))$ then $\{\psi : \{\}\}$
$(\{b:t\}) =$ if n.i.$(f_4(t))$ then $\{\psi : \{\}\}$
$(\{c:t\}) = \{\psi : \{\}\}$

f_5: $(\{a:t\}) =$ if n.i.$(f_5(t))$ then $\{\psi : \{\}\}$
$(\{b:t\}) =$ if n.i.$(f_5(t))$ then $\{\psi : \{\}\}$
$(\{c:t\}) = \{\}$.

Clearly, f_A^{sem} accepts $I = \{b : \{a : \{c : \{\}\}\} \cup \{b : \{c : \{\}\}\}\}$, which is a b edge followed by paths $a.c$ and $b.c$.

Note that in the example A works solely on paths, while for f_A^{sem} no such restriction is imposed. In general, f_A^{sem} may be called on arbitrary trees that are not necessarily Υ-simulation trees. For example a node may have more outgoing paths of $i-1$ $\sharp$ edges followed by a non-$\sharp$ edge, in our understanding all of them should represent the i^{th} branch. Even if for a tree t in $\mathcal{T}_{edge}^{\Omega\cup\{\sharp\}}$ $f_A^{sem}(t)$ is not empty, still it cannot be decided whether A will accept the correspondence of t or not. Nevertheless, as it will turn out, if $f_A^{sem}(t)$ is empty, then the corresponding node-labelled trees of t are all refused by A. This observation will enable us to solve the problem presented in Example 6.14.

Let $A = (\{q_1, \ldots, q_n\}, \Upsilon, Q_I, \Phi)$ be an alternating tree automaton and t a root-edged tree in $\mathcal{T}_{edge}^{\Omega\cup\{\sharp\}}$ ($\Upsilon = (\Omega, Arity)$). Informally, t will be transformed into an Υ-simulation tree (or rather set of Υ-simulation trees) and this tree will be rewritten to its correspondence in $\mathcal{T}_{node}^{\Upsilon}$.

Formally, first we show that we may assume that there is a symbol in Υ that is rejected by all of the states of A. We denote this symbol $\flat$. If this is not the case add $\flat$ to Υ s.t. $Arity(\flat) = 0$. Φ should also be extended with transition rules: $(q_i, \flat) \to false$ $(1 \leq i \leq n)$. Denote A' the resulting alternating tree automaton. Trivially, if A accepts a tree, then A' also accepts this tree. On the other hand, if A' accepts a tree, then by changing the $\flat$ labels to any other symbols in Υ with zero $Arity$, the resulting tree is accepted by A.

Secondly, consider the non-leaf edges of t (their endnodes have at leeast one outgoing edge) that are different from the root one after the other. Let (u, a, v) be such an edge. If for an i there does not exist an outgoing path of i $\sharp$-edges followed by a non-$\sharp$ edge, then add a path of i $\sharp$ edges followed by a $\flat$ edge to v $(0 \leq i \leq Arity(a) - 1)$.

(†) In the next step the unnecessary edges should be deleted. If there is an outgoing path of k $\sharp$ edges from v, where $Arity(a) \leq k$, then leave the first edge of this path. The remaining edges of this path will not be reachable from the root, thus with this deletion effectively the whole path is removed. Similarly, if for a k, $0 \leq k \leq Arity(a) - 1$, there are more outgoing paths from v with k $\sharp$ edges followed by a non-$\sharp$ edge, then except one path leave the first edges of the rest of the paths. It goes without saying that this is a non-deterministic step. If $k = 0$, then the last rule says that if there are more outgoing edges with a non-$\sharp$ label, then apart from one of these edges the rest should be deleted. Denote t' the result of this construction (the reachable subtree from the root). Obviously, t' is an Υ-simulation tree. As an example consider Figure 15.(f)-(g). Here, $\Upsilon = \{a, c, d, \sharp, \flat\}$, $Arity(a) = 3$, $Arity(d) = 2$, $Arity(\sharp) = 1$, $Arity(c) = Arity(\flat) = 0$.

In the next step the order of the neighbouring edges in t' should be defined. The outgoing edges with the non-$\sharp$ label precedes all of their neighbours. Otherwise, for edges e_1, e_2, e_1 precedes e_2, if the path beginning with e_1 contains less $\sharp$ edges before the non-$\sharp$ labelled edge, than the path beginning with e_2.

Now, to be able to transform an Υ-simulation tree to a tree in $\mathcal{T}^{\Upsilon}_{node}$ the paths of $\sharp$-edges should be deleted. Let pa be such a path, whose first edge is $(u, \sharp, v)$ and the endnode of its last edge is w. To remove pa delete edge $(u, \sharp, v)$, add an ε edge from u to w and then eliminate this edge. The preceding example continues in Figure 15.(h).

Before transforming this tree into a node-labelled one in $\mathcal{T}^{\Upsilon}_{node}$ an order should be defined among its neighbouring edges, which preserves the intended meaning of the order introduced among the neighbouring edges of t'. Let e_1, e_2 be neighbouring edges. Then e_1 precedes e_2, if it was preceded by a shorter path of $\sharp$ edges in t' than e_2. Now mapping $\phi_{edge \to node}$ can be applied to this tree, denote t'' the result. Clearly, t'' is in $\mathcal{T}^{\Upsilon}_{node}$. Consider Figure 15.(i) for the final result of our example.

It is easy to see that $Tr(t'')$ is t', i.e., if t'' is transformed into an Υ-simulation tree, then the result is t'. In what follows with $ReTr$ we refer to the previous method, which transforms a root-edged tree in $\mathcal{T}^{\Omega \cup \{\sharp\}}_{edge}$ to a set of trees in $\mathcal{T}^{\Upsilon}_{node}$. Note that if t was not root-edged, then we would not be able to apply transformation $\phi_{edge \to node}$ to the appropriate intermediary trees.

Lemma 6.15. *Let $A = (\{q_1, \ldots, q_n\}, \Upsilon, Q_I, \Phi)$ be an alternating tree automaton and t a root-edged tree in $\mathcal{T}^{\Omega \cup \{\sharp\}}_{edge}$ ($\Upsilon = (\Omega, Arity)$). Then, if $f^{sem}_A(t)$ is empty, then A refuses all elements of $ReTr(t)$.*

Proof. Suppose that on the contrary there is an element t'' in $ReTr(t)$ s.t. A accepts t''. According to Lemma 6.13. $f^{sem}_A(Tr(t''))$ is not empty. Denote t' the tree what we get from t after adding the paths of i $\sharp$ edges followed by a $\flat$ edge to those (u, a, v) edges that had not got any outgoing paths of $\sharp$ edges of length i ($0 \leq i \leq Arity(a) - 1$). Note that this is the first step of the construction of the elements of $ReTr(t)$. Recall that $Tr(t'')$ is among the results of the next step of this construction (step (†)). Since in this step only edges are deleted $Tr(t'')$ is a pregraph of t'. From Proposition 4.27. it follows then that $f^{sem}_A(t')$ is also not empty. (Remember that f^{sem}_A is in $SR(n.i.)$.) Recall that t' is constructed from t by adding $\sharp$ and $\flat$ edges. The transformation rules of f^{sem}_A belonging to $\sharp$, $\flat$ edges always return the

empty graph, hence $f_A^{sem}(t)$ should also be non-emtpy, which contradicts to our previous supposition. ■

Now, consider the complement of the semi-simulating structural recursion of the complement of A, in notation $\tilde{f}_{\tilde{A}}^{sem}$. The simulating structural recursion of A is defined as follows:

$$f_A := f_A^{sem} \sqcap \tilde{f}_{\tilde{A}}^{sem}.$$

Theorem 6.16. *Let $A = (Q, \Upsilon, Q_I, \Phi)$ be an alternating tree automaton ($\Upsilon = (\Omega, Arity)$). Then,*

(i) for tree t in $\mathcal{T}_{node}^{\Upsilon}$, A accepts $t \Leftrightarrow f_A(Tr(t))$ is not empty.

(ii) For a root-edged tree t in $\mathcal{T}_{edge}^{\Omega \cup \{\sharp\}}$, $f_A(t)$ is not empty $\Leftrightarrow$ A accepts all elements of $ReTr(t)$.

Proof. (i) If A accepts t, then according to Lemma 6.13. $f_A^{sem}(Tr(t))$ is not empty. On the other hand, $\tilde{A}$ refuses t, hence $f_{\tilde{A}}^{sem}(Tr(t))$ is empty (Lemma 6.13. again), consequently, since $Tr(t)$ is root-edged (Remark 6.12.) Proposition 5.8. implies that $\tilde{f}_{\tilde{A}}^{sem}(Tr(t))$ is not empty. This proves the non-emptiness of $f_A(Tr(t))$.

For the reverse direction it is enough to note that the non-emptiness of $f_A(Tr(t))$ entails the non-emptiness of $f_A^{sem}(Tr(t))$, from which using the result of Lemma 6.13. again the acceptance of t by A follows.

(ii) Suppose first that $f_A(t)$ is not empty. This means that by definition $\tilde{f}_{\tilde{A}}^{sem}(t)$ is also not empty, consequently, since t is root-edged $f_{\tilde{A}}^{sem}$ is empty (Proposition 5.8). Hence by Lemma 6.15. $\tilde{A}$ refuses all elements of $ReTr(t)$, which implies that A has to accept these elements.

On the other hand, if A accepts the elements of $ReTr(t)$, then $f_A^{sem}(Tr(t'))$ is not empty, where t' denotes an arbitrary element of $ReTr(t)$. With the same reasoning as that of used in Lemma 6.15. it can be shown that from this the non-emptiness of $f_A^{sem}(t)$ follows. If we assumed that $\tilde{f}_{\tilde{A}}^{sem}(Tr(t'))$ is empty, then, since $Tr(t')$ is root-edged (Remark 6.12.) this would entail

the non-emptiness of $f^{sem}_{\tilde{A}}(Tr(t'))$, which in turn would mean the acceptance of t' by $\tilde{A}$ resulting a contradiction. Thus $\tilde{f}^{sem}_{\tilde{A}}(Tr(t'))$ is not empty, from which, in the same way as for f^{sem}_{A}, the non-emptiness of $\tilde{f}^{sem}_{\tilde{A}}(t)$ follows. All together we get that $f_A(t)$ is not empty. ■

7 The problem of emptiness

In this section the complexity of the emptiness problem for the different classes of structural recursions is examined. It will be shown that the question remains practically tractable even for structural recursions in $SR(n.i.)$. On the other hand, the introduction of isempty conditions induces a surprisingly large increase in the complexity. Namely, using a similar result proven for alternating tree automata [12] it will be proven that the emptiness problem in $SR(n.i., i.)$ as well as the containment problem is DEXPTIME-complete in general. Besides, an interesting class of structural recursion whose conditions may be embedded up to a certain level will be introduced. We conjecture that both the emptiness and containment problems are complete for the appropriate class of the polynomial hierarchy [26] in accordance with the level of the embedding of the conditions, nevertheless, unfortunately, we have been only able to prove these questions are hard with respect to the aforementioned complexity class.

In [20] we have already examined a fragment of structural recursions in which the conditions consisted of a single not-isempty condition. Two cases were distinguished on the basis that whether the application of the else-branches is permitted or not. We found that in the second case the emptiness question can be answered in quadratic time. In the first case we showed how the two problems can be reduced to each other in polynomial time and proved that both questions are PSPACE-hard in general.

Before going into the details of the new results we prove a proposition which rather belongs to the topic of the containment problem, however, its corollary will play an important role in this section

Proposition 7.1. *Let f and g be arbitrary structural recursions. Then f does not contain $g \Leftrightarrow$ there is a tree t s.t. $f(t)$ is empty, while $g(t)$ is not empty.*

Proof. If f does not contain g, then there is an instance I s.t. $f(I)$ is empty and $g(I)$ is not empty. If I is a tree, then there is nothing left to prove. Otherwise, we may suppose that I is root-edged. Namely, if $I = I_1 \cup \ldots \cup I_k$, then for all i, $f(I_i)$ is still empty, while there is at least one I_j to which $g(I_j)$ is not empty $(1 \leq i, j \leq k)$. Since I is root-edged, $\tilde{f}(I)$ is not empty (Proposition 5.8.) and from Proposition 5.14. it follows that $(\tilde{f} \sqcap g)(I)$ is also not empty. By Theorem 4.38. there is a tree I_{tree} s.t. $(\tilde{f} \sqcap g)(I_{tree})$ is not empty. From Remark 4.39. we know that I_{tree} is also root-edged. Now, Proposition 5.14. and 5.8. imply that $f(I_{tree})$ is empty whereas $g(I_{tree})$ is not empty. ■

In the proof of Proposition 7.1. the following statements have been already proven.

Corollary 7.2. *For arbitrary structural recursions f and g,*

(i) if there is an instance I s.t. $f(I)$ is not empty, then there is a root-edged instance I' s.t. $f(I')$ is also not empty.

(ii) If there is an instance I s.t. $f(I)$ is empty and $g(I)$ is not empty, then there is a root-edged instance I' to which $f(I')$ is empty and $g(I')$ is not empty.

Note that the first statement is the special case of the second.

In some cases instead of the emptiness (containment) problem, it is better to examine a variant of the question.

Definition 7.3. *For an arbitrary structural recursion f and a symbol b in Σ the question that whether there is a root-edged instance I, whose root-edge is labelled with b, s.t. $f(I)$ is not empty is called the* emptiness problem for fixed b root-edged instances.

The problem of containment with fixed b root-edged instances can be defined in a similar way.

Proposition 7.4. *The emptiness and containment problems of structural recursions in Θ and the corresponding problems for fixed b root-edged instances can be reduced to each other in polynomial time*

$$\Theta \in \{SR(), SR(n.i.), SR(n.i., i.)\}$$

.

Proof. Let $f = (\{f_1, \ldots, f_n\}, \Sigma, \{f_i\}, \Gamma)$ be an arbitrary structural recursion in CCNF. From Corollary 7.2. it follows that for f it is enough to solve the emptiness problem for fixed b root-edged instances for all b in $\Sigma_f \cup \{\S\}$. Here, recall that Σ_f contains those a symbols of Σ to which there is a transformation rule $\gamma_{j,a}$ in Γ $(1 \leq j \leq n)$. Besides, § is a symbol different from the elements of Σ_f. For reducing the containment problem to the problem for fixed b root-edged instances the same reasoning can be applied.

To prove the reverse direction, except from $\gamma_{i,b}$ delete all transformation rules of f_i (here as the notation shows f_i is the only initial structural function of f). Then in order to solve the emptiness (containment) problem for fixed b root-edged instances for f it is enough to solve the emptiness (containment) problem for this new structural recursion. ■

7.1 Structural recursions in $SR(n.i.)$

As it has been already noted in Remark 3.9. the steps of condition evaluation can also be applied on operational graphs. The emptiness problem for structural recursions in $SR(n.i.)$ will be solved by means of this algorithm. In order to prove its applicability it should be shown that the condition evaluation over an operational graph encompasses all important aspects of the conditions evaluations over the intersection of the operational graph in question and an arbitrary instance. This property can be formulated properly by using operational homomorphism.

Lemma 7.5. *Let f be an arbitrary structural recursion in $SR(n.i., i)$ and I an instance. Then there is an operational homomorphism ρ from $U_f \sqcap I$ to U_f.*

Proof. For node (f_i, u_I) in $U_f \sqcap I$, $\rho((f_i, u_I)) := f_i$. Trivially, requirements (i)-(iv) of Definition 4.22. are fulfilled by ρ. ■

Note that the structural recursion of the preceding lemma may contain *isempty* conditions.

Definition 7.6. *Let S be a schema graph. We say that I* matches *S, if there exists a one-to-one mapping $\mu : V.I \to V.S$ s.t. for all edges (u, a, v) of I, $(\mu(u), p, \mu(v)) \in E.S$ and $p(a)$ is true.*

For an operational graph U_f, if we substitute each predicate labels with a satisfying constant, then clearly the result is a matching instance of U_f.

Lemma 7.7. *For structural recursion $f = (\{f_1, \ldots, f_n\}, \Sigma, F_I, \Gamma)$ in $SR(n.i., i.)$ and matching instance I, there is an operational homomorphism ρ from U_f to $U_f \sqcap I$.*

Proof. Denote μ the one-to-one mapping from I to U_f given in Definition 7.6. It is easy to prove by using a straightforward induction on the number of the edges of I that $(\mu(u), u)$ is in $V.U_f \sqcap I$ for all u, where u is a node of I. Using this observation $\rho(f_i)$ is defined to be $(\mu(u), u)$, where $\mu(u) = f_i$ $(1 \leq i \leq n)$. Since μ is a one-to-one mapping $\rho(f_i)$ is unique. Again, it is straightforward that ρ fulfills the requirements of Definition 4.22. ■

We say that structural recursion f is *constructing*, if there is at least one path from a root to a constructor edge in U_f after the condition evaluation.

Theorem 7.8. *Let f be a structural recursion $SR(n.i.)$. Then f is constructing $\Leftrightarrow$ there is an instance I s.t. $f(I)$ is not empty.*

Proof. Suppose first that f is constructing and let I be a matching instance. By Lemma 7.7. there is an operational homomorphism from U_f to $U_f \sqcap I$. The non-emptiness of $f(I)$ follows then from Lemma 4.24.

For the reverse direction, consider an instance I s.t. $f(I)$ is not empty. By Lemma 7.5. there is an operational homomorphism from $U_f \sqcap I$ to U_f. Since $f(I)$ is not empty there is a path from a root to a conditional edge in $U_f \sqcap I$ after condition evaluation. Again, Lemma 4.24. implies that there is also a path to a constructor edge in U_f after condition evaluation. ■

Corollary 7.9. *Let f be a structural recursion in $SR(n.i.)$.*

(i) Then, f is constructing $\Leftrightarrow$ for an arbitrary matching instance I of f, $f(I)$ is not empty.

(ii) The emptiness problem can be decided in $O(|f|^2)$ time.

(iii) If f is in $SR()$, then the emptiness problem can be solved in linear time.

Proof. Statement (i) has been already proven in the proof of Theorem 7.8.

(ii) Applying the reasoning about the number of steps necessary to construct $f(I)$ on Page 36 it is easy to see that condition evaluation can be accomplished in $O(|f|^2)$ time over f. Afterwards, it is enough to check whether a constructor edge is still reachable from the root of the remaining graph, which can be accomplished in linear time. This observation also proves statement (iii), because if f is without condition, then it is enough to check the latter property. ■

7.2 Structural recursion in $SR(n.i., i.)$

First we prove that the emptiness and containment problem can be reduced to each other in this case.

Lemma 7.10. *Let f and g be arbitrary structural recursions, then f contains $g \Leftrightarrow$ for all instances I, $(\tilde{f} \sqcap g)(I)$ is empty.*

Proof. Suppose that f contains g and still, there is an instance I to which $(\tilde{f} \sqcap g)(I)$ is not empty. Using the result of Corollary 7.2. we may assume that I is root-edged. From Proposition 5.14. it follows that neither $\tilde{f}(I)$

nor $g(I)$ is empty. However, Proposition 5.8. shows that in this case $f(I)$ is empty, which is a contradiction. The reverse direction can be proven in a similar way. ■

Proposition 7.11. *In case of structural recursions in $SR(n.i., i.)$ the problems of emptiness and containment can be reduced to each other in polynomial time.*

Proof. In Lemma 7.10. we have just shown how the containment problem can be reduced to the question of emptiness. For the reverse direction consider structural recursion $f^{empty} = (\{f_1\}, \Sigma, \{f_1\}, \Gamma^{empty})$, whose single transformation rule is as follows:

$f_1 \colon (\{* : t\}) = \{\}$.

Then, clearly, for an arbitrary structural recursion f, there is an instace I s.t. $f(I)$ is not empty iff f^{empty} does not contain f. ■

Theorem 7.12. *The emptiness and containment problem of structural recursions in $SR(n.i., i.)$ are DEXPTIME-complete in general.*

Proof. The emptiness problem of alternating tree automata is known to be DEXPTIME-complete [12]. From Theorem 6.10. and 6.16. the statement of this theorem immediately follows. ■

Corollary 7.13. *The emptiness and containment problem of structural recursions in $SR(n.i., i., el)$ are DEXPTIME-complete in general.*

Proof. The statement is a straightforward consequence of Corollary 5.11. ■

In the rest of this subsection a Turing machine will be constructed, which will decides the emptiness problem of structural recursions in $SR(n.i., i.)$. We believe that Claim 7.19. and 8.15. could be proven by means of this Turing machine. The details will be given after the aforementioned claims.

First, we give a characterization of the case, when an instance returns a non-empty output. This characterization will be used to construct a tree

that shows which n.i. and i. conditions are satisfied during a process of an input. The aforementioned Turing machine will be built upon the idea of these trees.

Lemma 7.14. *For structural recursion* $f = (\{f_1, \ldots, f_n\}, \Sigma, \{f_i\}, \Gamma)$ *in CCNF and in* $SR(n.i., i.)$*, there is an instance* I *s.t.* $f(I)$ *is not empty* $\Leftrightarrow$ *there is a transformation rule* $\gamma_{i,\vartheta}$ *s.t. in* $\gamma_{i,\vartheta}$ *a* ψ *edge is constructed* ($\vartheta \in \Sigma_f \cup \{*\}$). *Besides,*

(i) either $\gamma_{i,\vartheta}$ *is without a condition.*

(ii) Or $\gamma_{i,\vartheta}$ *is with a condition, and there are structural functions* $f_{i_1}, \ldots, f_{i_k}$ *and* $f_{j_1}, \ldots, f_{j_s}$ *in* $Form(\gamma_{i,\vartheta})$ ($\{i_1, \ldots, i_k\}, \{j_1, \ldots, j_s\} \subseteq \{1, \ldots, n\}$) *s.t.*

a) if each condition $n.i.(f_{i_o}(t))$, $i.(f_{j_p}(t))$ *becomes true, then* $Form(\gamma_{i,\vartheta})$ *becomes true* ($1 \leq o \leq k, 1 \leq p \leq s$).

b) For each o *there is an instance* $\hat{I}$ *s.t.* $(f^{i_o} \sqcap (f_{j_1} \widetilde{\sqcup \ldots \sqcup} f_{j_s}))(\hat{I})$ *is not empty.*

Proof. Suppose first there is an instance I s.t. $f(I)$ is not empty. By Corollary 7.2. we may assume that I is root-edged, and this root-edge is labelled with a, i.e., $I = \{a : I'\}$. The statement will be proven by using induction on the steps m of the condition evaluation in $U_f \sqcap I$.

Assume that m is 0. Then, trivially $\gamma_{i,a}$ must be without any condition and a ψ-edge must be constructed there, in other words it fulfills the requirement of condition (i).

Suppose now that the statement holds for $m < M + 1$ and let m be $M + 1$. Then $\gamma_{i,a}$ surely contains a condition. We show that the structural functions called in n.i. conditions resulting a non-empty output for I' can be taken as $f_{i_1}, \ldots, f_{i_k}$, while the structural functions of i. conditions returning an empty output for I' can be taken as $f_{j_1}, \ldots, f_{j_s}$. In this case condition (ii) a) is clearly satisfied. Suppose that $I' = I_1 \cup \ldots \cup I_q$, where all I_l-s are

root-edged ($1 \leq l \leq q$). For each o, there is an index l s.t. $f_{i_o}(I_l)$ is non-empty ($1 \leq o \leq k, 1 \leq l \leq q$). Trivially, $f_{j_p}(I_l)$ is still empty for all p, thus by Proposition 5.8. and 5.14. $(f_{j_1} \widetilde{\sqcup \ldots \sqcup} f_{j_s})(I_l)$ is non-empty. Since I_l is root-edged Proposition 5.14. also implies that $(f^{i_o} \sqcap (f_{j_1} \widetilde{\sqcup \ldots \sqcup} f_{j_s}))(I_l)$ is not empty.

On the other hand, if there is transformation rule $\gamma_{i,\vartheta}$ without a condition, in which a ψ edge is constructed (requirement (i)), then for a ϑ-labelled edge f clearly returns a non-empty output.

Finally, if $\gamma_{i,\vartheta}$ meets the requirements of condition (ii), then denote I_{i_o} the instance for which $(f^{i_o} \sqcap (f_{j_1} \widetilde{\sqcup \ldots \sqcup} f_{j_s}))(I_{i_o})$ is non-empty ($1 \leq o \leq k$). We may assume again that I_{i_o} is root-edged. Then, for instance

$$I = I_{i_1} \cup \ldots \cup I_{i_k},$$

$f_{i_o}(I)$ remains non-empty. Since $(f_{j_1} \widetilde{\sqcup \ldots \sqcup} f_{j_s})(I_{i_o})$ is non-empty and I_{i_o} is root-edged, $(f_{j_1} \sqcup \ldots \sqcup f_{j_s})(I_{i_o})$ is empty for all o (Proposition 5.8.), hence $(f_{j_1} \sqcup \ldots \sqcup f_{j_s})(I)$ is also empty. Suppose that ϑ is a, where a is in Σ. From condition (ii) a) it follows now that when $\{a : I\}$ is processed by f, $Form(\gamma_{i,\vartheta})$ is satisfied and a ψ edge is constructed. Note that if ϑ is $*$ ($\gamma_{i,\vartheta}$ is the transformation rule for the default case), then $\{b : I\}$ results a non-empty output for f, where b is in Σ and f_i does not have any transformation rule for b. This concludes the proof. ■

Note that in condition (ii) both set $\{i_1, \ldots, i_k\}$ and $\{j_1, \ldots, j_s\}$ can be empty (not at the same time).

For an arbitrary structural recursion $f = (\{f_1, \ldots, f_n\}, \Sigma, \{f_i\}, \Gamma)$ in CCNF for which there is an instance returning a non-empty result, consider a root-edged tree I s.t. $f(I)$ is not empty (Theorem 4.38. and Corollary 7.2.). Suppose that I is of the form $\{a : I'\}$. A tree $PossRun(f)$ (a possible run of f) will be constructed which shows that when a condition of a transformation rule is satisfied in the condition evaluation of $U_f \sqcap I$, which n.i.$(f_j(t))$, i.$(f_k(t))$ conditions becomes *true* in this condition ($1 \leq j, k \leq n$).

A node u of $PossRun(f)$ will be labelled with either *true* or two sets of structural functions $N.I.(u)$ and $I.(u)$. For the root u_0 of $PossRun(f)$ $N.I.(u_0)$ is f_i, while $I.(u_0)$ is the empty set. Consider then $\gamma_{i,a}$. Since $f(I)$ is not empty, it must satisfy either condition (i) or (ii) of Lemma 7.14. If the requirements of condition (i) are fulfilled, then an edge a should be added to u_0, whose endnode is labelled with *true*. Otherwise assume that in those n.i. conditions that become *true* in the formula of $\gamma_{i,a}$ structural functions $f_{i_1}, \ldots, f_{i_k}$ are called, while in the i. conditions that also become *true* $f_{j_1}, \ldots, f_{j_s}$ are invoked. Then an edge (u_0, a, u) should be added to u_0. In addition $N.I.(u)$ should be $\{f_{i_1}, \ldots, f_{i_k}\}$, whereas $I.(u)$ should be $\{f_{j_1}, \ldots, f_{j_s}\}$. Obviously, there is no structural function that would be an element of both sets. Furthermore, we may assume that no subtree $I''$of I' satisfies all of these n.i. and i. conditions, otherwise we would consider $\{a : I''\}$ instead of I. Clearly, this new tree would also return a non-empty output for f. This supposition guarantees that u does not have any descendant u' s.t. $N.I.(u) \subseteq N.I.(u')$ and $I.(u) \subseteq I.(u')$ would hold.

In the general construction step suppose that for node u $N.I.(u)$ is $f_{i_1}, \ldots, f_{i_k}$ and $I.(u)$ is $f_{j_1}, \ldots, f_{j_s}$. We know that there is a subtree $I'' = I_1 \cup \ldots \cup I_q$ of I s.t. for all j there is a p, where

$$(f_{i_j} \sqcap (f_{j_1} \widetilde{\sqcup \ldots \sqcup} f_{j_s}))(I_p)$$

is not empty. Here, I_p is root-edged $(1 \leq j \leq k, 1 \leq p \leq q)$. We may suppose again that there is no subtree I'_p of I_p s.t.

$$(f_{i_j} \sqcap (f_{j_1} \widetilde{\sqcup \ldots \sqcup} f_{j_s}))(I'_p)$$

would also be non-empty. Otherwise, we may take $I'' \cup I'_p$ instead of I'' and we may assume that $f_{i_j} \sqcap (f_{j_1} \widetilde{\sqcup \ldots \sqcup} f_{j_s})$ returns a non-empty result through I'_p. Again, this supposition guarantees that u does not have any descendant u' s.t. $N.I.(u) \subseteq N.I.(u')$ and $I.(u) \subseteq I.(u')$ would hold. Furthermore, suppose that the root-edge of I_p is labelled with a. Denote γ_a the transformation rule of $f_{i_j} \sqcap (f_{j_1} \widetilde{\sqcup \ldots \sqcup} f_{j_s})$ for a. If γ_a is without a condition (a ψ edge is

surely constructed in its then branch), then an edge a should be added from u whose endnode is labelled with *true*. Otherwise, assume that in those n.i. conditions that become *true* in the formula of γ_a structural functions $f_{o_1}, \ldots, f_{o_m}$ are called, while in the i. conditions that also become *true* $f_{p_1}, \ldots, f_{p_r}$ are invoked. Then an edge (u, a, v) with label a should be added to u, where $N.I.(v)$ is $f_{o_1}, \ldots, f_{o_m}$, whereas $I.(v)$ is $f_{p_1}, \ldots, f_{p_r}$.

The next definition summarizes the properties of $PossRun(f)$.

Definition 7.15. *Let $f = (\{f_1, \ldots, f_n\}, \Sigma, \{f_i\}, \Gamma)$ be an arbitrary structural recursion in CCNF and in $SR(n.i., i.)$. Then a tree t, whose inner nodes are labelled with two sets of structural functions and the leaves with value true called* possible run *of f, if for an arbitrary node u of t with $N.I.(u) = \{f_{i_1}, \ldots, f_{i_k}\}, I.(u) = \{f_{j_1}, \ldots, f_{j_s}\}$ the following requirements are fulfilled.*

(i) There is not any structural function f_k s.t. f_k belongs to both $N.I.(u)$ and $I.(u)$.

(ii) There is not any descendant v of u s.t. $N.I.(u) \subseteq N.I.(v)$ and $I.(u) \subseteq I.(v)$.

(iii) u has k outgoing edges. Consider the m^{th} outgoing edge. Denote v its endnode and suppose that its label is a. Furthermore denote $f_{i_m} \sqcap \widetilde{(f_{j_1} \sqcup \ldots \sqcup f_{j_s})}$ with h.

(a) If v is a leaf, then $\gamma_{h,a}$ is without a condition and it constructs an edge ψ.

(b) Otherwise, for each element f_o, f_p of $N.I.(v)$ and $I.(v)$ respectively $n.i.(f_o(t))$, $i.(f_p(t))$ is a condition of the formula of $\gamma_{h,a}$. What is more, if all of these conditions become true, then this formula is also satisfied.

Note that for structural recursion f with structural functions $f_1, \ldots, f_n$ each possible run is finite, since a path from the root to a leaf may contain

less than

$$\sum_{k=0}^{n}\left(\binom{n}{k}\left(\sum_{i=0}^{n-k}\binom{n-k}{i}\right)\right)$$

nodes, furthermore each node has at most n outgoing edges.

Proposition 7.16. *For an arbitrary structural recursion*

$$f = (\{f_1, \ldots, f_n\}, \Sigma, \{f_i\}, \Gamma)$$

in CCNF and in SR(n.i., i.), there is an instance resulting a non-empty output for $f \Leftrightarrow$ there is also a possible run of f.

Proof. Let t be a possible run of f. Delete all of its node labels. Obviously, the result after this deletion is an instance in $\mathcal{T}^{\Sigma}$ (data trees whose edges are labelled by elements of Σ). Furthermore, it can be proven by a straightforward induction on the number of edges of t that this instance returns a non-empty output for f. The reverse direction of the proposition has been already proven in the description of $PossRun(f)$. ■

Let $f = (\{f_1, \ldots, f_n\}, \Sigma, \{f_i\}, \Gamma)$ be a structural recursion in CCNF. The Turing machine that decides the containment problem is non-deterministic and it contains a single tape. In the first step it writes

$$\sharp f_i \flat \S f_i$$

to its tape. Symbol $\sharp$ will always be followed by two sets of structural functions, symbol $\flat$ functions as a delimiter between these sets. The elements of these sets will belong to those n.i. and i. conditions respectively that should become *true* in the corresponding transformation rule. This enumeration will be closed by symbol §. Suppose that the preceding two sets consists of elements $f_{i_1}, \ldots, f_{i_k}$ and $f_{j_1}, \ldots, f_{j_s}$. Then symbol f_{i_o} after § will mean that in the rest of the tape the emptiness problem of structural function $f_{i_o} \sqcap (f_{j_1} \widetilde{\sqcup \ldots \sqcup} f_{j_s})$ is to be decided ($1 \leq o \leq k$). In this case the corresponding section of the tape contains the following sequence of symbols

$$\sharp f_{i_1} \ldots f_{i_k} \flat f_{j_1} \ldots f_{j_s} \S f_{i_o}.$$

In the description of the general step suppose that the emptiness problem of the preceding structural function $f_{i_o} \sqcap (f_{j_1} \widetilde{\sqcup \ldots \sqcup} f_{j_s})$ is to be decided. Non-deterministically choose one of its transformation rule γ.

Suppose first that γ contains a condition. Denote $N.I.(\gamma)$, $I.(\gamma)$ the two sets of structural functions that are called in the n.i. and i. conditions in the formula of γ respectively. Non-deterministically choose two subsets $N.I.'(\gamma)$ and $I.'(\gamma)$ of the preceding sets. If there is a common element of these subsets, then the computation should stop and the input should be rejected. Secondly, if all of the corresponding n.i. and i. conditions become *true* and the formula of γ is still not satisfied, then the input should be rejected as well. Thirdly, if in the preceding part of the tape there is symbol sequence $\sharp f_{l_1}, \ldots, f_{l_p} \flat f_{o_1}, \ldots, f_{o_r} \S$ s.t. $N.I.(\gamma)$ contains $\{f_{l_1}, \ldots, f_{l_p}\}$ and $I.(\gamma)$ contains $f_{o_1}, \ldots, f_{o_r}$, then the input should be rejected again. Otherwise, write a symbol $\sharp$ and afterwards enumerate the elements of $N.I.'(\gamma)$ and $I.'(\gamma)$ separated by symbol $\flat$. Write symbols $\S$ and f_p then and start to decide the emptiness problem for $f_p \sqcap (. \widetilde{\sqcup \ldots \sqcup} .)$, where f_p is the first element of $N.I.'(\gamma)$, while the dots represent the elements of $I.'(\gamma)$.

If γ is without any condition and it constructs the empty graph, then the computation should stop and the input should be rejected.

Finally, if γ is without a condition, but an edge ψ is constructed, then this obviously means that there is an instance returning a non-empty output for $f_{i_o} \sqcap (f_{j_1} \widetilde{\sqcup \ldots \sqcup} f_{j_s})$. Thus, if f_{i_o} is not the last element of set $\{f_{i_1}, \ldots, f_{i_k}\}$, then substitute f_{i_o} with $f_{i_{o+1}}$ after symbol $\S$ and start to decide the emptiness problem for $f_{i_{o+1}} \sqcap (f_{j_1} \widetilde{\sqcup \ldots \sqcup} f_{j_s})$. On the other hand, if f_{i_o} is the last element, then the condition of the corresponding transformation rule can be satisfied, hence the elements of the aforementioned two sets, i.e., sequence

$$f_{i_1}, \ldots, f_{i_k} \flat f_{j_1}, \ldots, f_{j_s}$$

should be deleted. Remember that this sequence is preceded by a symbol $\sharp$. If this symbol is not preceded by any other symbol, which means that

$$\sharp f_{i_1} \ldots f_{i_k} \flat f_{j_1} \ldots f_{j_s} \S f_{i_o} \text{ is } \sharp f_i \flat \S f_i,$$

then the computation should stop and the Turing machine should accept its input. Otherwise, it is preceded by a structural function symbol suppose that this is f_p $(1 \leq p \leq n)$. This means that the transformation rule, from which it has just been shown that it is satisfiable, belongs to structural function $f_p \sqcap (. \widetilde{\sqcup \ldots \sqcup} .)$. Here the dots represent structural functions that belongs to i. conditions that should be satisfied in the condition of the appropriate transformation rule. Again, if f_p is not the last among those structural functions that are called in the n.i. conditions of the preceding transformation rule, then the computation should continue with substituting f_p with the next element of this set. If it is the last element, then again both sets of structural functions should be deleted and the deletion of elements should be continued in the same way until the input is accepted or a structural function is found that is not the last among those structural functions that are called in the n.i. conditions of a transformation rule. Denote M_f this Turing machine.

Proposition 7.17. *For an arbitrary structural recursion f in CCNF, there is a run of M_f in which the input is accepted $\Leftrightarrow$ there is an instance I s.t. $f(I)$ is not empty.*

Proof. It can be shown by using a straightforward induction on the number of steps of the computation that at each moment the content of the tape of M_f represents a path whose endnode is not necessarily labelled with *true* but otherwise satisfies the requirements of Definition 7.15. Hence by following the steps of the computation successively, if the input is accepted, a possible run of f can be built. Reversely, if f has a possible run, then using this tree an accepting run of M_f can be constructed. To sum up M_f has an accepting run iff f has a possible run. On the other hand, by Proposition 7.16. f has a possible run iff there is an input returning a non-empty output for f. ■

7.3 Structural recursions in $SR(n.i., i., \leq k)$.

Definition 7.18. *Let f be a structural recursion in CNF. Then f belongs to $SR(n.i., i., \leq k)$, if all paths not traversing any looping edge in U_f contain at most k premises ($k \geq 0$).*

Note that the operational graphs of structural recursion in $SR(n.i., i., \leq k)$ may contain looping edges that are premises, but cannot contain any other directed cycles containing one or more premises. Thus, in such structural recursions the conditions may be embedded at most to the k^{th} level. A structural recursion is in $SR(n.i., i., = k)$, if there is a path in its operational graph, which traverses each looping edge at most once, moreover, it contains k different premises, in other words the conditions are embedded to the k^{th} level. Similarly, a structural function of this structural recursion is in $SR(n.i., i., = k)$, if it is the starting node of such a path. Furthermore, if for a structural recursion f there is a k s.t. f belongs to $SR(n.i., i., \leq k)$, then we say that f is *restrictively embedded.*

Claim 7.19. *Let $f^1, \ldots, f^n$, $g^1, \ldots, g^m$ be structural recursions in $SR(n.i., i., \leq k)$. Then the question that whether there exists an instance I s.t. for all i and j $f^i(I)$ is not empty, while $g^j(I)$ is empty, is in $\Sigma_k P$ ($1 \leq i \leq n, 1 \leq j \leq m$).*

Note that the proof of Proposition 7.11., which was based on Lemma 7.10. and Corollary 7.2., can be applied to restrictively embedded structural recursions without any changes, hence if Claim 7.19. would proven to be true, then this result would also show that the containment problem belongs to $\Pi_k P$ in this case.

It is not difficult to see that the construction of a possible run can be extended to the case of Claim 7.19. Simply, for the root u_0 of such a tree $N.I.(u_0)$ should be chosen to be $f^1_{k_1}, \ldots, f^n_{k_n}$, while $I.(u_0)$ to $g^1_{r_1}, \ldots, g^m_{r_m}$, where $f^i_{k_i}, g^j_{r_j}$ denote an initial function of f^i, g^j respectively ($1 \leq i \leq n, 1 \leq j \leq m$). Now, it is also not difficult to see that Claim 7.19. is a consequence of the following statement.

(†) Keeping the notations of Claim 7.19. there is a possible run t of $f^1, \ldots, f^n, g^1, \ldots, g^m$ s.t. the number of those nodes u of t, whose labels $N.I.(u)$, $I(u)$ contain at least one structural function in $SR(n.i., i, = k)$ is polynomial in the size of $f^1, \ldots, f^n, g^1, \ldots, g^m$.

However, unfortunately, up to now we have not been able to prove (†).

The emptiness problem is $\Sigma_k P$-hard. Next, we consider the problem of *quantified satisfiability with i alternations of quantifiers*, QSAT_i in notation, which is known to be a $\Sigma_i P$-complete problem [26]. Here, for a given propositional formula ϕ, whose logical variables are divided into disjunct sets $Y_1, \ldots, Y_n$, the question is whether it is true that there is a partial truth assignment of the variables of Y_1 s.t. for all partial truth assignments of the variables in Y_2 there is a partial truth assignment of the variables of Y_3 and so on up to Y_i, ϕ is satisfied by the overall truth assignment. In other words the truth value of formula

$$\exists Y_1 \forall Y_2 \exists Y_3 \ldots Q Y_i \phi$$

should be decided, where $\exists Y_j$ ($\forall Y_j$) denotes $\exists X_{j_1} \ldots \exists X_{j_k}$ ($\forall X_{j_1} \ldots \forall X_{j_k}$), $Y_j = \{X_{j_1}, \ldots, X_{j_k}\}$. Furthermore, Q is the existential quantifier, if i is odd and the universal quantifier otherwise. In what follows, capitals X, Y with a possible subindex will denote a logical variable and a set of logical variables respectively, whereas x with a possible subindex will denote an edge label representing variable X.

This problem will be reduced to the emptiness problem for fixed b root-edged instances. Note that the proof of Proposition 7.4. can be applied without any changes for structural recursions in $SR(n.i., i., \leq k)$, hence the emptiness problem and the emptiness problem for fixed b root-edged instances can be reduced to each other in polynomial time in this case as well.

First, for formula ϕ we define a corresponding condition, in notation $Form(\phi)$, which can be used in the condition of a transformation rule. In $Form(\phi)$, simply, every instance of X_i ($\neg X_i$) should be changed to n.i.($f^{ch}_{x_i}(t)$)

(n.i.($f^{ch}_{\neg x_i}(t)$)) ($1 \leq i \leq n$). Here the single transformation rule belonging to $f^{ch}_{x_i}$ is as follows:

$f^{ch}_{x_i}$: ($\{x_i : t\}$)= $\{\psi : \{\}\}$,

ch in upper index abbreviates *child* and it indicates that the edge right under the edge being processed should have label x_i. The definition of $f^{ch}_{\neg x_i}$ is similar. Then ϕ is simulated with the following structural recursion

$$f^{\phi} = (\{f_{\phi}, f^{ch}_{x_1}, f^{ch}_{\neg x_1}, \ldots, f^{ch}_{x_n}, f^{ch}_{\neg x_n}\}, \Sigma, \{f_{\phi}\}, \Gamma),$$

where the single transformation rule belonging to f_{ϕ} is as follows:

f_{ϕ}: ($\{\flat : t\}$)= if $Form(\phi)$ then $\{\psi : \{\}\}$.

Next the representations of truth assignments are introduced. A star, i.e., a node with outgoing edges that have no further outgoing edges, is called *truth assignment star*, if the outgoing edges are labelled with x_i, $\neg x_i$. Furthermore, if an edge is labelled with x_i ($\neg x_i$), then none of the edges is labelled with $\neg x_i$ (x_i). We say that a truth assignment star *encodes* a truth assignment Θ, if for each variable X_i, if $\Theta(X_i) = true$, then it contains a x_i-labelled edge, otherwise a $\neg x_i$-labelled edge. An example can be found in Figure 16.(a). The following lemma is straightforward.

Lemma 7.20. *Let I be $\{\flat : t\}$, where t is an truth assignment star. Then $f^{\phi}(I)$ is not empty $\Leftrightarrow$ t encodes an a truth assignment that satisfies ϕ.*

Next, the quantifiers is to be simulated. A literal L_i is either X_i or $\neg X_i$. Accordingly, l_i will denote either an x_i- or an $\neg x_i$-labelled edge. We say that path $l_1 \ldots l_n$ *encodes* a truth assignment Θ, if for each variable X_i, where $\Theta(X_i) = true$ ($\Theta(X_i) = false$) l_i is x_i ($\neg x_i$). These paths will be called *truth assignment paths*. Furthermore, we say that a truth assignment star *matches* a truth assignment path, if they encode the same truth assignment. In a Θ-*assignment instance* a truth assignment path encoding Θ and a matching truth assignment star is joined, i.e, from the endnode of the truth assignment

path an edge ε should be added to the root of the truth assignment star and it should be eliminated afterwards. For expression $\forall Y$ the $\forall Y$-*assignment instance* contains each possible Θ-assignment instance as a pregraph and apart from the edges of these instances it does not contain any other edges. Here Θ is a truth assignment over the variables of Y. Similarly, an $\exists Y$-*assignment instance* is a Θ-assignment instance s.t. Θ is again a truth assignment over the variables of Y. Moreover, an $\exists Y_1 \forall Y_2$-*assignment instance* consists of a truth assignment path encoding a truth assignment Θ over the variables of Y_1 followed by the $\forall Y_2$-assignment instance, in which the the truth assignment stars are extended to encode Θ as well. Finally, a $\forall Y_1 \exists Y_2$-*assignment instance* begins with the $\forall Y_1$-assignment instance, however here, each truth assignment star is substituted with an arbitrary $\exists Y_2$-assignment instance. Obviously, a path from the root to a root of a truth assignment star of an $\exists Y_2$-assignment instance encodes a truth assignment over the variables of Y_1 and Y_2. The assignment star should be extended to encode this truth assignment. In general the construction of an $\exists Y_1 \forall Y_2 \exists Y_3 \ldots QY_n$-*assignment instance* can be given along the same line. Examples can be found in Figure 16.(b)-(e).

We begin the simulation with the base cases. Suppose first that P is of the form: $\forall Y \phi$, where $Y = \{X_1, \ldots, X_n\}$. The corresponding structural recursion

$$f_\phi^{\forall Y} = \{(f_{\forall Y}, \ldots\}, \Sigma, \{f_{\forall Y}\}, \Gamma)$$

has the following transformation rules:

$$\begin{aligned}
f_{\forall Y}: \quad (\{\flat : t\}) &= \text{if n.i.}(f_{x_1}^{ch}(t)) \wedge \text{n.i.}(f_{\neg x_1}^{ch}(t)) \wedge \text{i.}(f_{x_1,\neg x_1}^{not_ch}(t)) \wedge \text{i.}(f_{\forall Y}(t)) \\
&\quad \text{then } \{\psi : \{\}\} \\
(\{x_i : t\}) &= \text{if i.}(f_{all}(t)) \vee \text{i.}(f_{x_{i+1}}^{ch}(t)) \vee \text{i.}(f_{\neg x_{i+1}}^{ch}(t)) \vee \text{n.i.}(f_{\neg x_i}^{desc}(t)) \vee \\
&\quad \text{n.i.}(f_{x_{i+1},\neg x_{i+1}}^{not_ch}(t)) \vee \text{n.i.} f_{\forall Y}(t) \text{ then } \{\psi : \{\}\} \\
(\{\neg x_i : t\}) &= \text{if i.}(f_{all}(t)) \vee \text{i.}(f_{x_{i+1}}^{ch}(t)) \vee \text{i.}(f_{\neg x_{i+1}}^{ch}(t)) \vee \text{n.i.}(f_{x_i}^{desc}(t)) \vee \\
&\quad \text{n.i.}(f_{x_{i+1},\neg x_{i+1}}^{not_ch}(t)) \vee \text{n.i.} f_{\forall Y}(t) \text{ then } \{\psi : \{\}\} \\
(\{x_n : t\}) &= \text{if i.}(f_{x_1}^{ch}(t)) \wedge \text{i.}(f_{\neg x_1}^{ch}(t)) \vee \ldots \vee \text{i.}(f_{x_{n-1}}^{ch}(t)) \wedge \text{i.}(f_{\neg x_{n-1}}^{ch}(t)) \vee \\
&\quad \text{i.}(f_{x_n}^{ch}(t)) \vee \text{n.i.}(f_{\neg x_n}^{ch}(t)) \vee Form(\neg\phi) \text{ then } \{\psi : \{\}\}
\end{aligned}$$

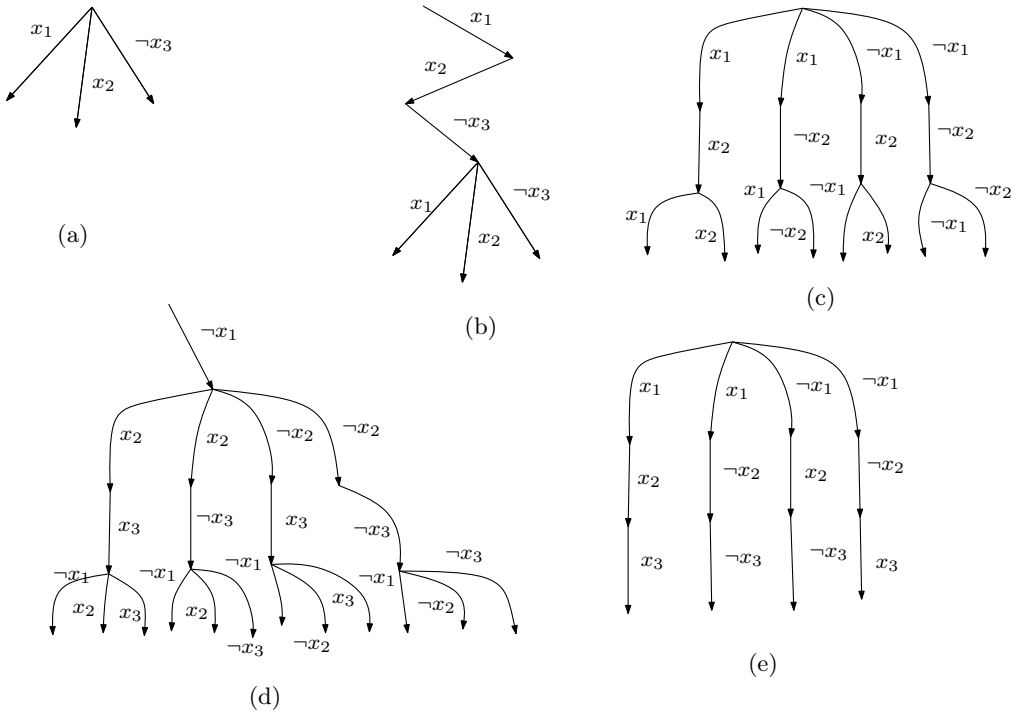

Figure 16: (a) A truth assignment star, which encodes Θ, where $\Theta(X_1) = true$, $\Theta(X_2) = true$, $\Theta(X_3) = false$. (b) A Θ-assignment instance for the truth assignment of (a). (c) The $\forall X_1 \forall X_2$-instance. (d) An $\exists X_1 \forall X_2 \forall X_3$-instance. (e) A $\forall X_1 \forall X_2 \exists X_3$-instance.

$$(\{\neg x_n : t\}) = \text{if i.}(f^{ch}_{x_1}(t)) \wedge \text{i.}(f^{ch}_{\neg x_1}(t)) \vee \ldots \vee \text{i.}(f^{ch}_{x_{n-1}}(t)) \wedge \text{i.}(f^{ch}_{\neg x_{n-1}}(t)) \vee \text{i.}(f^{ch}_{\neg x_n}(t)) \vee \text{n.i.}(f^{ch}_{x_n}(t)) \vee Form(\neg\phi) \text{ then } \{\psi : \{\}\}$$

$(1 \leq i \leq n-1)$, where

$$f^{desc}_{x_i}: \quad (\{x_i : t\}) = \{\psi : \{\}\} \qquad (\{* : t\}) = f^{desc}_{x_i}(t)$$

$$f^{desc}_{\neg x_i}: (\{\neg x_i : t\}) = \{\psi : \{\}\} \qquad (\{* : t\}) = f^{desc}_{\neg x_i}(t)$$

$$f^{not\_ch}_{x_i,\neg x_i}: (\{x_i : t\}) = \{\} \qquad (\{\neg x_i : t\}) = \{\} \qquad (\{* : t\}) = \{\psi : \{\}\}.$$

$$f_{all}: \quad (\{* : t\}) = \{\psi : \{\}\}$$

Here, f^{all} returns a ψ-edge for all input that is different from the empty graph, f^{desc}_{ϑ} returns a ψ-edge, if the root-edge of its input has a ϑ-labelled descendant $\vartheta \in \{x_i, \neg x_i\}$, while $f^{not\_ch}_{x_i,\neg x_i}$ returns a ψ-edge, if there is not any outgoing edge x_i or $\neg x_i$ from the root of its input. Trivially, $f^{\forall X}_{\phi}$ is in $SR(n.i., i., \leq 1)$.

Lemma 7.21. *Let P be a formula of the form $\forall Y \phi$. Then, P is true $\Leftrightarrow$ there is a $\flat$ root-edged instance for which $f^{\forall Y}_{\phi}$ returns a non-empty output.*

Proof. Suppose that $Y = \{X_1, \ldots, X_n\}$ and there is a $\flat$ root-edged instance I s.t. $f^{\forall Y}_{\phi}(I)$ is not empty. Clearly this root edge must have both x_1- and $\neg x_1$-labelled children, but it cannot have any children with a different label, moreover $f_{\forall Y}$ must return an empty output for the subgraph under the root edge.

It is easy to see that $f_{\forall Y}$ returns an empty result for subgraph $\{x_i : t\}$ $(\{\neg x_i : t\})$, if t is not the empty graph, and the x_i- $(\neg x_i)$-labelled edge have an x_{i+1}- and a $\neg x_{i+1}$-labelled child, and it does not have an $\neg x_i$- (x_i)-labelled descendant, and it does not have any children whose label is different from both x_{i+1} and $\neg x_{i+1}$, and $f_{\forall Y}$ returns an empty output for t $(1 \leq i \leq n-1)$. Moreover, $f_{\forall Y}$ returns an empty result for subgraph $\{x_n : t\}$ $(\{\neg x_n : t\})$, if the x_n- $(\neg x_n)$-labelled edge have an x_i- or a $\neg x_i$-labelled child $(1 \leq i \leq n-1)$, it also has an x_n- $(\neg x_n)$-labelled child, but it does not have a $\neg x_n$- (x_n)-labelled child, finally $Form(\neg\phi)$ is not satisfied by t $(1 \leq i \leq n)$. From

this explanation it follows that each x_n- or $\neg x_n$-labelled edge of I has an x_i or $\neg x_i$ labelled ancestor, but it cannot have both. On the subgraph under this ancestor structural function $f^{desc}_{\neg x_i}$ or $f^{desc}_{x_i}$ was invoked which ensures that this x_n or $\neg x_n$ edge cannot have an x_i- and an $\neg x_i$-labelled children at the same time. In other words, the x_n- or $\neg x_n$-labelled edge is followed by a truth assignment star, furthermore this edge is preceded by a truth assignment path which this truth assignment star matches. In addition this truth assignment star returns a non-empty output for $Form(\phi)$.

To sum up, since $f_{\forall Y}$ returns a non-empty result for I the root edge of I must be followed by the $\forall X$-assignment instance, whose truth assignment stars all return a non-empty output for $Form(\phi)$. By Lemma 7.20. this implies that all truth assignment over the variables of Y satisfy ϕ, i.e., $\forall Y \phi$ is *true*.

The previous explanation also shows that if $\forall Y \phi$ is *true*, then instance $\{\flat : I\}$ returns a non-empty output for $f^{\forall Y}_{\phi}$, where I is the $\forall Y$-assignment instance. This concludes the proof. ■

Next, suppose that P is $\exists Y \phi$, where Y is still $\{X_1, \ldots, X_n\}$, then its simulation

$$f^{\exists Y}_{\phi} = \{(f_{\exists Y}, \ldots\}, \Sigma, \{f_{\exists Y}\}, \Gamma)$$

has the following transformation rules:

$$\begin{aligned}
f_{\exists Y} : (\{\flat : t\}) &= \text{if } (\text{n.i.}(f^{ch}_{x_1}(t)) \vee \text{n.i.}(f^{ch}_{\neg x_1}(t))) \wedge \text{i.}(f^{not_ch}_{x_1, \neg x_1}(t)) \wedge \text{n.i.}(f_{\exists Y}(t)) \\
&\quad \text{then } \{\psi : \{\}\} \\
(\{x_i : t\}) &= \text{if } (\text{n.i.}(f^{ch}_{x_{i+1}}(t)) \vee \text{n.i.}(f^{ch}_{\neg x_{i+1}}(t))) \wedge \text{i.}(f^{desc}_{\neg x_i}(t)) \wedge \\
&\quad \text{i.}(f^{not_ch}_{x_{i+1}, \neg x_{i+1}}(t)) \wedge \text{n.i.} f_{\exists Y}(t) \text{ then } \{\psi : \{\}\} \\
(\{\neg x_i : t\}) &= \text{if } (\text{n.i.}(f^{ch}_{x_{i+1}}(t)) \vee \text{n.i.}(f^{ch}_{\neg x_{i+1}}(t))) \wedge \text{i.}(f^{desc}_{x_i}(t)) \wedge \\
&\quad \text{i.}(f^{not_ch}_{x_{i+1}, \neg x_{i+1}}(t)) \wedge \text{n.i.} f_{\exists Y}(t) \text{ then } \{\psi : \{\}\} \\
(\{x_n : t\}) &= \text{if } (\text{n.i.}(f^{ch}_{x_1}(t)) \wedge \text{i.}(f^{ch}_{\neg x_1}(t)) \vee \text{n.i.}(f^{ch}_{\neg x_1}(t)) \wedge \text{i.}(f^{ch}_{x_1}(t))) \wedge \ldots \wedge \\
&\quad (\text{n.i.}(f^{ch}_{x_{n-1}}(t)) \wedge \text{i.}(f^{ch}_{\neg x_{n-1}}(t)) \vee \text{n.i.}(f^{ch}_{\neg x_{n-1}}(t)) \wedge \text{i.}(f^{ch}_{x_{n-1}}(t))) \wedge \\
&\quad \text{n.i.}(f^{ch}_{x_n}(t)) \wedge \text{i.}(f^{ch}_{\neg x_n}(t))) \wedge Form(\phi) \text{ then } \{\psi : \{\}\} \\
(\{\neg x_n : t\}) &= \text{if } (\text{n.i.}(f^{ch}_{x_1}(t)) \wedge \text{i.}(f^{ch}_{\neg x_1}(t)) \vee \text{n.i.}(f^{ch}_{\neg x_1}(t)) \wedge \text{i.}(f^{ch}_{x_1}(t))) \wedge \ldots \wedge
\end{aligned}$$

$$(\text{n.i.}(f^{ch}_{x_{n-1}}(t)) \wedge \text{i.}(f^{ch}_{\neg x_{n-1}}(t)) \vee \text{n.i.}(f^{ch}_{\neg x_{n-1}}(t)) \wedge \text{i.}(f^{ch}_{x_{n-1}}(t))) \wedge$$
$$\text{n.i.}(f^{ch}_{\neg x_n}(t)) \wedge \text{i.}(f^{ch}_{x_n}(t))) \wedge Form(\phi) \text{ then } \{\psi : \{\}\}$$

$(1 \leq i \leq n-1)$.

As in the previous case, $f^{\exists Y}_{\phi}$ is trivially in $SR(n.i., i., \leq 1)$.

Lemma 7.22. *Let P be a formula of the form $\exists Y \phi$. Then, P is true $\Leftrightarrow$ there is a $\flat$-labelled root-edged instance for which $f^{\exists Y}_{\phi}$ returns a non-empty output.*

Proof. Suppose that $Y = \{Y_1, \ldots, Y_n\}$ and there is a $\flat$-labelled root-edged instance I s.t. $f^{\exists Y}_{\phi}$ is not empty. Then this root-edge must have an x_1 or a $\neg x_1$ child, but it cannot have any children with a different label and $f_{\exists Y}$ must return a non-empty output for the subgraph under this root edge. $f_{\exists Y}$ returns a non-empty result for subgraph $\{x_i : t\}$ ($\{\neg x_i : t\}$), if the x_i ($\neg x_i$) labelled root edge has an x_{i+1}- or $\neg x_{i+1}$-labelled child, but it does not have any children with a different label, and it does not have any $\neg x_i$- (x_i)-labelled descendant, finally $f_{\exists Y}$ returns a non-empty output for t. $f_{\exists Y}$ returns a non-empty result for subgraph $\{x_n : t\}$ ($\{\neg x_n : t\}$), if the x_n- ($\neg x_i$)-labelled edge has either an x_i- or a $\neg x_i$-labelled child ($1 \leq i \leq n-1$), it also has an x_n- ($\neg x_n$)-labelled child, but it does not have a $\neg x_n$- (x_n)-labelled child, and t returns a non-empty output for $Form(\phi)$. All in all, since $f_{\exists Y}$ returns a non-empty output for I, the root-edge of I must be followed by an $\exists Y$-assignment instance, where the encoded truth assignment satisfies ϕ. By Lemma 7.20. this implies that $\exists Y \phi$ is *true*.

The previous explanation also shows that if $\exists Y \phi$ is *true*, then instance $\{\flat : I\}$ returns a non-empty output for $f^{\forall Y}_{\phi}$, where I is an $\exists Y$-assignment instance s.t. the encoded truth assignment satisfies $\exists Y \phi$. ■

After the base cases, we explain how the construction continues for

$$\exists Y_1 \forall Y_2 \phi \text{ and } \forall Y_1 \exists Y_2 \phi.$$

Suppose that Y_1 is $\{X_1, \ldots, X_n\}$ and Y_2 is $\{X_{n+1}, \ldots, X_m\}$. Consider first $\exists Y_1 \forall Y_2 \phi$. Then in the simulation

$$f_\phi^{\exists Y_1 \forall Y_2} = (\{f_{\exists Y_1}, f_{\forall Y_2} \ldots\}, \Sigma, \{f_{\exists Y_1}\}, \Gamma).$$

The transformation rules of $f_{\exists Y_1}$ are the same as were the transformation rules of $f_{\exists Y}$. The only difference is that in the transformation rules for x_n and $\neg x_n$ the formula should be the same as in the transformation rule for $\flat$ in $f_{\forall Y_2}$. This transformation rule should be omitted from $f_{\forall Y_2}$, otherwise the transformations rules of $f_{\forall Y_2}$ are the same as the transformation rules of $f_{\forall Y}$. The only difference in this case is that in the transformation rules for x_m and $\neg x_m$ all variables of Y_1 and Y_2 should be taken into account. More concretely, these transformation rules are of the following form:

$$\begin{aligned}
(\{x_m : t\}) \;=\; & \text{if i.}(f_{x_1}^{ch}(t)) \wedge \text{i.}(f_{\neg x_1}^{ch}(t)) \vee \ldots \vee \text{i.}(f_{x_{m-1}}^{ch}(t)) \wedge \text{i.}(f_{\neg x_{m-1}}^{ch}(t)) \vee \\
& \text{i.}(f_{x_m}^{ch}(t)) \vee \text{n.i.}(f_{\neg x_m}^{ch}(t)) \vee Form(\neg\phi) \text{ then } \{\psi : \{\}\} \\
(\{\neg x_m : t\}) = \; & \text{if i.}(f_{x_1}^{ch}(t)) \wedge \text{i.}(f_{\neg x_1}^{ch}(t)) \vee \ldots \vee \text{i.}(f_{x_{m-1}}^{ch}(t)) \wedge \text{i.}(f_{\neg x_{m-1}}^{ch}(t)) \vee \\
& \text{i.}(f_{\neg x_m}^{ch}(t)) \vee \text{n.i.}(f_{x_m}^{ch}(t)) \vee Form(\neg\phi) \text{ then } \{\psi : \{\}\}
\end{aligned}$$

Obviously, $f_\phi^{\exists Y_1 \forall Y_2}$ is in $SR(n.i., i., \leq 2)$.

Lemma 7.23. *Let P be a formula of the form $\exists Y_1 \forall Y_2 \phi$. Then, P is true $\Leftrightarrow$ there is a $\flat$-labelled root-edged instance I for which $f_\phi^{\exists Y_1 \forall Y_2}$ returns a non-empty output.*

Proof. Suppose first that $f_\phi^{\exists Y_1 \forall Y_2}$ returns a non-empty output for I, where I is a $\flat$-labelled root-edged instance. Then in a similar way as in the proof of Lemma 7.21. and 7.22. one can show that the root edge must be followed by a truth assignment path encoding a truth assignment Θ over the variables of Y_1. In turn this path must be followed by the $\forall Y_2$-assignment instance, but in this case the truth assignment stars must also match Θ, i.e., they should be extended with x_i-, $\neg x_i$-labelled edges appropriately ($1 \leq i \leq n$). What is more, all of these truth assignment stars must return a non-empty output for $Form(\phi)$. From Lemma 7.20. it follows now that $\exists Y_1 \forall Y_2 \phi$ is *true*.

Again, the previous explanation shows that if $\exists Y_1 \forall Y_2 \phi$ is *true*, then instance $\{\flat : I\}$ returns a non-empty output for $f_\phi^{\exists Y_1 \forall Y_2}$, where I is an $\exists Y_1 \forall Y_2$-assignment instance s.t. the encoded truth assignment over Y_1 makes $\exists Y_1 \forall Y_2 \phi$ *true*. ■

The construction for $\forall Y_1 \exists Y_2 \phi$ works in a similar way. In

$$f_\phi^{\forall Y_1 \exists Y_2} = (\{f_{\forall Y_1}, f_{\exists Y_2} \dots\}, \Sigma, \{f_{\forall Y_1}\}, \Gamma).$$

The transformation rules of $f_{\forall Y_1}$ are the same as were the transformation rules of $f_{\forall Y}$. The only difference is that in the transformation rules for x_n and $\neg x_n$ the formula should be the same as in the transformation rule for $\flat$ in $f_{\exists Y_2}$. This latter transformation rule should be omitted, otherwise the transformations rules of $f_{\exists Y_2}$ are the same as the transformation rules of $f_{\exists Y}$. The only difference in this case is that in the transformation rules for x_m and $\neg x_m$ all variables of Y_1 and Y_2 should be taken into account. More concretely, these transformation rules are of the following form:

$$\begin{aligned}
(\{x_m : t\}) \;\; = \text{if } & (\text{n.i.}(f_{x_1}^{ch}(t)) \wedge \text{i.}(f_{\neg x_1}^{ch}(t)) \vee \text{n.i.}(f_{\neg x_1}^{ch}(t)) \wedge \text{i.}(f_{x_1}^{ch}(t))) \wedge \dots \wedge \\
& (\text{n.i.}(f_{x_{m-1}}^{ch}(t)) \wedge \text{i.}(f_{\neg x_{m-1}}^{ch}(t)) \vee \text{n.i.}(f_{\neg x_{m-1}}^{ch}(t)) \wedge \text{i.}(f_{x_{m-1}}^{ch}(t))) \wedge \\
& \text{n.i.}(f_{x_m}^{ch}(t)) \wedge \text{i.}(f_{\neg x_m}^{ch}(t))) \wedge Form(\phi) \text{ then } \{\psi : \{\}\} \\
(\{\neg x_m : t\}) = \text{if } & (\text{n.i.}(f_{x_1}^{ch}(t)) \wedge \text{i.}(f_{\neg x_1}^{ch}(t)) \vee \text{n.i.}(f_{\neg x_1}^{ch}(t)) \wedge \text{i.}(f_{x_1}^{ch}(t))) \wedge \dots \wedge \\
& (\text{n.i.}(f_{x_{m-1}}^{ch}(t)) \wedge \text{i.}(f_{\neg x_{m-1}}^{ch}(t)) \vee \text{n.i.}(f_{\neg x_{m-1}}^{ch}(t)) \wedge \text{i.}(f_{x_{m-1}}^{ch}(t))) \wedge \\
& \text{n.i.}(f_{\neg x_m}^{ch}(t)) \wedge \text{i.}(f_{x_m}^{ch}(t))) \wedge Form(\phi) \text{ then } \{\psi : \{\}\}.
\end{aligned}$$

Clearly, $f_\phi^{\forall Y_1 \exists Y_2}$ is in $SR(n.i., i., \leq 2)$.

Lemma 7.24. *Let P be a formula of the form $\forall Y_1 \exists Y_2 \phi$. Then, P is true $\Leftrightarrow$ there is a $\flat$-labelled root-edged instance for which $f_\phi^{\forall Y_1 \exists Y_2}$ returns a non-empty output.*

Proof. Suppose first that $f_\phi^{\forall Y_1 \exists Y_2}$ returns a non-empty output for I, where I is a $\flat$-labelled root-edged instance. Then in a similar way as in the proof of Lemma 7.21. and 7.22. one can show that the root edge must be followed by the $\forall Y_1$-assignment instance whose truth assignment stars are substituted

with $\exists Y_2$-assignment instances. Each truth assignment star is substituted with a single $\exists Y_2$-assignment instance. Furthermore, the truth assignment star of such an $\exists Y_2$-assignment instance should be extended to encode that truth assignment over the variables of Y_1 which the truth assignment path over the $\exists Y_2$-assignment instance in question encodes. In other words, the root edge of I must be followed by a $\forall Y_1 \exists Y_2$-assignment instance whose truth assignment stars all return a non-empty output for ϕ. Lemma 7.20. implies now that each truth assignment over Y_1 can be extended with a truth assignment over Y_2 s.t. ϕ becomes *true* in the overall truth assignment.

The previous explanation shows that if $\forall Y_1 \exists Y_2 \phi$ is *true*, then instance $\{\flat : I\}$ returns a non-empty output for $f_\phi^{\forall Y_1 \exists Y_2}$, where I is an $\forall Y_1 \exists Y_2$-assignment instance s.t. for each encoded truth assignment over Y_1 the corresponding truth assignment over Y_2 makes ϕ *true*. ■

Theorem 7.25. *For a given formula P of the form $\exists Y_1 \forall Y_2 \exists Y_3 \ldots Q Y_j \phi$, P is true $\Leftrightarrow$ there is a $\flat$-labelled root-edged instance for which $f_\phi^{\exists Y_1 \forall Y_2 \exists Y_3 \ldots Q Y_j}$ returns true.*

Proof. The statement can be proven by using an induction on the number of the quantifiers. The base case has been already proven in Lemma 7.21. and 7.22. The general case be proven in a similar way as in Lemma 7.23. and 7.24. ■

8 The problem of containment

In the previous section the complexity of the containment problem for structural recursions in $SR(n.i., i., \leq k)$, $SR(n.i., i.)$ and $SR(n.i., i., el)$ has been determined. Here, the same question will be addressed for the rest of the classes of structural recursions. However, instead of class $SR()$ another class $SR(n.i., \vee)$ will be examined, where as the notation reveals the disjunction of not isempty conditions is allowed to be taken in the conditions of transformation rules. It is easy to see that the conditional form of an arbitrary structural recursion in $SR()$ is always in $SR(n.i, \vee)$. It will turn out that the

containment problem for deterministic structural functions can be solved in polynomial time, while for structural recursions in $SR(n.i., \vee)$, $SR(n.i.)$ the problem is coNP-complete and PSPACE-hard respectively.

In [20] we have already addressed the containment problem for a fragment of structural recursions in which the conditions consisted of a single not-isempty logical function. In this case without else-branches the problem has been proven to be coNP-complete. With else-branches we could only show the PSPACE-hardness of the question.

8.1 Structural recursions in $SR()$

Earlier we have already seen that Corollary 4.26. provides a sufficient condition for the containment of structural recursions in $SR(n.i.)$. Namely, if there is an operational homomorphism from U_g to U_f, where f and g are structural recursions in $SR(n.i.)$, then f contains g. On the other hand, the next example shows that this condition is not necessary.

Example 8.1. Consider $f = (\{f_1, f_2\}, \Sigma, \{f_1\}, \Gamma_f)$ and $g = (\{g_1, g_2\}, \Sigma, \{g_1\}, \Gamma_g)$, where the transformation rules are as follows:

$f_1 \colon (\{a : t\}) = \{\psi : f_2(t)\}$ $\qquad$ $f_2 \colon (\{a : t\}) = \{\}$

$g_1 \colon (\{a : t\}) = g_2(t)$ $\qquad$ $g_2 \colon (\{a : t\}) = \{\psi : \{\}\}$.

Clearly, f contains g. On the other hand $\rho : V.U_g \to V.U_f$, where

$$\rho(g_1) = f_1,\ \rho(g_2) = f_2,\ \rho(w^g_{end}) = w^f_{end},$$

is not an operational homomorphism, since (g_2, a, w^g_{end}) is a constructor edge, whereas $\rho((g_2, a, w^g_{end}))$ is not. Here, the path from the root to a constructor edge in U_f contains a constructor edge sooner as its correspondence in U_g, hence if an instance results a construction for g, then it must "traverse" this constructor edge, when it is processed by f.

8.1.1 Separators

Now, we give another criterion with which the containment problem can be decided. Even thought it does not provide a necessary and sufficient condition in general, it can be used to figure out the complexity of the containment problem of some important special cases. Informally, as a guarantee of non-containment for structural recursions f and g we will try to find such a pregraph of U_g that is constructing, and it has a matching instance I to which $f(I)$ is empty (Corollary 7.9.).

Formally, consider two structural recursions f, g in $SR(n.i.)$ and a pregraph G of $U_f \sqcap U_g$. An edge e_g of U_g is a *separator edge*, if there is a symbol a_{e_g} that satisfies the predicate of e_g, while for each e_f, where $(e_f, e_g) \in U_f \sqcap U_g$, a_{e_g} does not satisfy the predicate of e_f. Clearly, if e_g does not have a pair in $U_f \sqcap U_g$, then it is a separator edge.

By means of the subsequent algorithm using G a candidate pregraph of U_g will be constructed that might have the aforementioned property, i.e., it is constructing and it has a matching instance to which $f(I)$ is empty. Denote G_0 the ancestor image of G in U_g.

(i) In the i^{th} step add those separator edges to G_{i-1} that have a parent edge in G_{i-1} $(i > 0)$.

(ii) Besides, if all of the parent edges of a non-separator edge are in G_{i-1} s.t. those parent edges that are in G_0 are all separator edges, then this edge should also be added to G_{i-1}. Denote G_i the result of step (i) and (ii).

Trivially, this construction should end after at most $|V.U_g|$ steps. Denote G_g the resulting graph. In what follows the edges of G_0 will be called *primary edges*, and the ancestor image of G according to U_f will be denoted by G_f.

Definition 8.2. *Keeping the preceding notation, we call* G separator *with respect to* g*,*

(i) if G_g is constructing, while G_f is not.

(ii) For each non-separator, primary edge e_g of G_g and for each edge e_f of U_f, where (e_f, e_g) is an edge of $U_f \sqcap U_g$, (e_f, e_g) is in $E.G$.

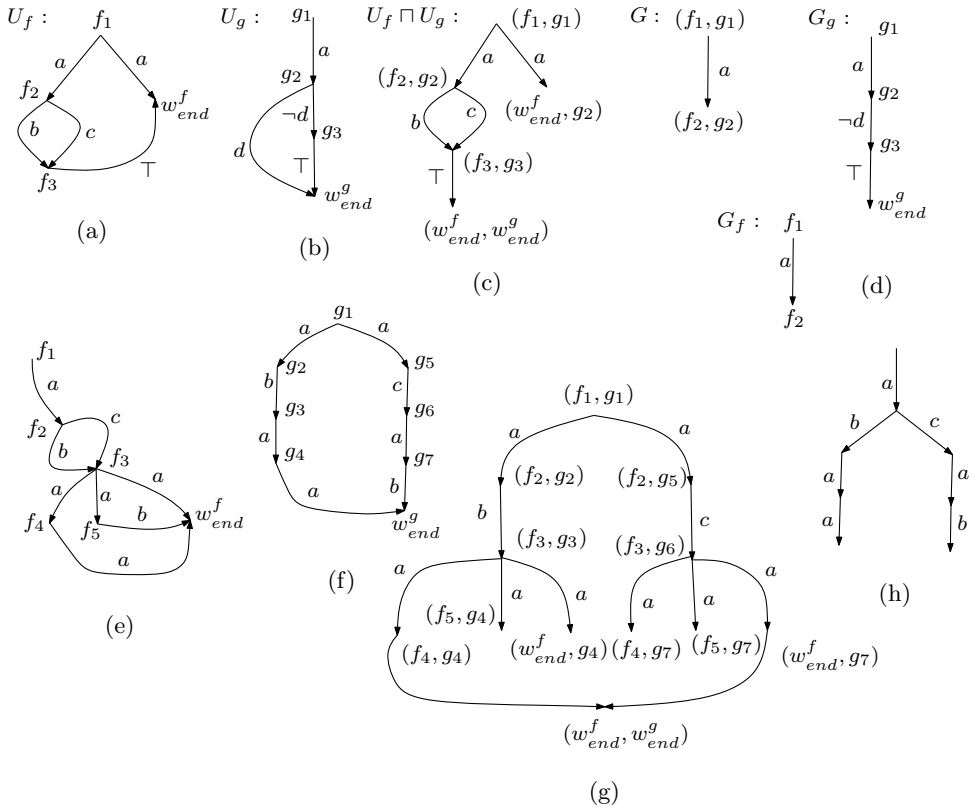

Figure 17: (a) The operational graph of structural recursion f in Example 8.3. (U_f). (b) The operational graph of structural recursion g in Example 8.3. (U_g). (c) The intersection of the operational graphs of (a) and (b). (d) A separator and the corresponding pregraphs of U_f and U_g. (e) The operational graph of structural recursion f in Example 8.6. (f) The operational graph of structural recursion g in Example 8.6. (g) The intersection of the operational graphs of (f) and (g). (h) An instance that results an empty output for the structural recursion of (e) and a non-empty output for the structural recursion of (f).

In requirement (ii) note that since e_g is in G_0 it has at least one pair e_f s.t. (e_f, e_g) is in G by definition.

Example 8.3. For structural recursions $f = (\{f_1, f_2, f_3\}, \Sigma, \{f_1\}, \Gamma_f)$, $g = (\{g_1, g_2, g_3\}, \Sigma, \{g_1\}, \Gamma_g)$ an example for a separator w.r.t. g can be found in Figure 17.(a)-(d). The transformation rules are as follows:

$f_1\colon (\{a : t\}) =$ if n.i.$(f_2(t))$ then $\{a : \{\}\}$ $\quad f_2\colon (\{b : t\}) = f_3(t)$
$\quad (\{c : t\}) = f_3(t)$

$f_3\colon (\{* : t\}) = \{* : \{\}\}$

$g_1\colon (\{a : t\}) = g_2(t)$ $\quad g_2\colon (\{d : t\}) = \{\}$
$\quad (\{* : t\}) = g_3(t)$

$g_3\colon (\{* : t\}) = \{* : \{\}\}$.

Lemma 8.4. *Let f, g be structural recursions in SR(n.i.), and G_1, G_2 are two pregraphs of $U_f \sqcap U_g$ s.t. G_1 is also a pregraph of G_2. If G_1 is a separator, and the ancestor image of G_2 in U_f is still non-constructing, then G_2 is also a separator.*

Proof. From the method of the construction used for separators (Page 125) the lemma straightforwardly follows. ■

Lemma 8.5. *Let f and g be structural recursion in SR(n.i.). If there is a separator w.r.t. g, then f does not contain g.*

Proof. Throughout the proof the notation of Definition 8.2. will be used. For each edge e_g of G_g to which there is a symbol s.t. it satisfies the predicate of e_g, but it does not satisfy any of the predicates of those edges of U_f that are coupled with e_g in $U_f \sqcap U_g$, assign this symbol. To the rest of the edges of G_g assign a symbol that satisfies their predicates. Then construct a matching instance I of G_g, where each edge of G_g is substituted with this assigned symbol. In the same way as in the proof of Theorem 7.8. one can prove that $g(I)$ is still not empty.

Next, we show that apart from the edges of G_f no other edge of U_f can affect the construction of $f(I)$. Suppose that there is an edge e_f s.t. e_f is not in G_f, but it is in the ancestor image of $U_f \sqcap I$ according to U_f. Denote e_I a pair of e_f in $U_f \sqcap I$. Furthermore, denote e_g the correspondence of e_I in G_g (according to mapping μ in the definition of the matching instance in Definition 7.6). By a straightforward induction it can be shown that (e_f, e_g) is in $U_f \sqcap U_g$. More precisely, there is a path

$$(e_1^f, e_1^g) \dots (e_m^f, e_m^g),$$

where (e_1^f, e_1^g) is the root of $U_f \sqcap U_g$ and $(e_m^f, e_m^g) = (e_f, e_g)$, s.t. for each e_i^f there is an edge e_i^I to which (e_i^f, e_i^I) is in $U_f \sqcap I$ $(1 \leq i \leq m)$. Clearly, e_g is a non-separator edge. If it was also a primary edge, then according to requirement (ii) of Definition 8.2. (e_f, e_g) would be in G, thus e_f would be in G_f, which would be a contradiction. Thus, from the definition of the separator it follows that there is a j s.t. e_j^g is a separator edge $(1 \leq j \leq m)$. We also know that the label of e_j^I was chosen in such a way that it does not satisfy the predicate of e_j^f. Consequently (e_j^f, e_j^I) cannot be an edge of $U_f \sqcap I$ and this contradiction shows that the ancestor image of $U_f \sqcap I$ according to U_f is a pregraph of G_f.

By Lemma 7.5. there is an operational homomorphism from $U_f \sqcap I$ to this pregraph and hence to G_f as well. Thus, if $f(I)$ was not empty, then using the reasoning of Lemma 4.24. one could show that G_f would be constructing. Consequently, $f(I)$ is surely empty. ■

Example 8.6. This example shows that existence of a separator is not a necessary condition for containment. Consider structural recursions

$$f = (\{f_1, \dots, f_5\}, \Sigma, \{f_1\}, \Gamma) \text{ and } g = (\{g_1, \dots, g_7\}, \Sigma, \{g_1\}, \Gamma)$$

with the following transformation rules:

$f_1 \colon (\{a : t\}) = f_2(t)$

$f_2 \colon (\{b : t\}) = f_3(t)$
$(\{c : t\}) = f_3(t)$

$f_3\colon (\{a:t\}) = \text{if n.i.}(f_4(t)) \wedge \text{ n.i.}(f_5(t)) \text{ then } \{\psi:\{\}\}$

$f_4\colon (\{a:t\}) = \{\psi:\{\}\}$ $\qquad$ $f_5\colon (\{b:t\}) = \{\psi:\{\}\}$

$g_1\colon (\{a:t\}) = \text{if n.i.}(g_2(t)) \wedge \text{ n.i.}(g_5(t)) \text{ then } \{\psi:\{\}\}$ $\qquad$ $g_2\colon (\{b:t\}) = g_3(t)$

$g_3\colon (\{a:t\}) = g_4(t)$ $\qquad$ $g_4\colon (\{a:t\}) = \{\psi:\{\}\}$

$g_5\colon (\{c:t\}) = g_6(t)$ $\qquad$ $g_6\colon\ (\{a:t\}) = g_7(t)$ $\qquad$ $g_7\colon (\{b:t\}) = \{\psi:\{\}\}.$

The operational graphs of f and g can be found in Figure 17.(e)-(f). After a thorough examination of $U_f \sqcap U_g$ in Figure 17.(g) it turns out there is not any separator edge in U_g. Moreover, if we left an arbitrary edge of U_g, then the remaining graph would not be constructing. This shows that only $U_f \sqcap U_g$ could be a separator, however, here G_f, which is U_f, is constructing. Consequently, there is no separator in this case, still the instance on Figure 17.(h) results a non-empty output for g and an empty output for f. Here the difference of the "structure" of the conditions is the cause of the non-containment.

$SR(n.i., \vee)$**.** Next, we distinguish an important class of structural recursions to which the existence of a separator will proven to be not just a sufficient but a necessary condition as well. Denote $SR(n.i., \vee)$ the class of structural recursions without *isempty* conditions and conjunctions in their conditional form. Note that the conditional forms of structural recursions in $SR()$ all fall to this class, which underlines its importance. In the next theorem we show that for structural recursions of this kind, the non-existence of separators guarantees containment. The key observation here is that in this case even a single path from a root to a constructor edge in the intersection of the operational graph and the instance ensures the non-emptiness of the result.

Theorem 8.7. *Let f and g be two structural recursions in $SR(n.i., \vee)$. Then f contains $g \Leftrightarrow$ there is not any separator in $U_f \sqcap U_g$.*

Proof. The sufficiency has been already proven in Lemma 8.5. For the necessity assume that there is no separator in $U_f \sqcap U_g$, nevertheless there is an instance I s.t. $f(I)$ is empty, while $g(I)$ is not. We may suppose that the ancestor image of $U_g \sqcap I$ according to I is I itself, i.e., every edge of I is used in the construction of the intersection. Denote G_g the ancestor image of $U_g \sqcap I$ according to U_g. Again, in the same way as in the proof of Theorem 7.8. one can show that G_g is constructing. For each edge e_g of G_g take one of its pairs e_I in $U_g \sqcap I$. Denote a the label of e_I. From $U_f \sqcap G_g$ leave those (e_f, e_g) edges, where the predicate of e_f is not satisfied by a. Note that these e_g edges are all separator edges. Respectively denote G and G_f the remaining graph and the ancestor image of this graph according to U_f. Straightforwardly, if (e_f, e_g) is in G, then (e_f, e_I) is in $U_f \sqcap I$. Hence G_f is a pregraph of the ancestor image of $U_f \sqcap I$ according to U_f. Here, one should note that there cannot be any constructor edge in $U_f \sqcap I$, otherwise $f(I)$ would not be empty. Thus G_f is also without constructor edges, consequently G_f is not constructing. Furthermore, from $U_f \sqcap G_g$ only such edges were deleted, whose ancestor images according to G_g were separator edges. Thus, for each non-separator edge e_g of G_g and an arbitrary edge e_f s.t. (e_f, e_g) is in $U_f \sqcap G_g$, (e_f, e_g) remains in G. All together, this reasoning shows that G is a separator, which is a contradiction. ■

8.1.2 Deterministic structural recursions

First note that the conditional forms of deterministic structural recursions are all in $SR(n.i., \vee)$, consequently Theorem 8.7. can be applied to this case. Secondly, for deterministic structural recursions f and g, their operational graphs are also deterministic, where remember an operational graph is called deterministic, if for each pair of neighbouring edges with predicates p_1 and p_2, $p_1 \wedge p_2$ is unsatisfiable. Hence, for each edge e_f in U_f there is at most one edge e_g s.t. (e_f, e_g) is in $U_f \sqcap U_g$ and vice versa.

For arbitrary deterministic structural recursions f and g leave those edges (e_f, e_g) of $U_f \sqcap U_g$, where e_f is a constructor edge and denote G the result.

Lemma 8.8. *Keeping the preceding notations, f contains $g \Leftrightarrow G$ is not a separator.*

Proof. Note that G is the maximal pregraph of $U_f \sqcap U_g$ that can be a separator. So, if it is not a separator, then Lemma 8.4. implies that there is not any other separator in $U_f \sqcap U_g$. Hence, by Theorem 8.7. f contains g. The other direction is a trivial consequence of Theorem 8.7. ■

Theorem 8.9. *For arbitrary deterministic structural recursions f and g, the containment problem can be decided in $O(|f|^2|g|^2)$ time.*

Proof. Consider the method of the construction used in the introduction of separators (Page 125). According to Proposition 3.2. $U_f \sqcap U_g$ can be constructed in $O(|f||g|)$ time. The constructor edges of U_f can be found in $O(|f|)$ time, their correspondences in $U_f \sqcap U_g$ can be deleted in $O(|f||g|)$ time. Denote G the result after this deletion. Remember that an edge label of an operational graph is either a constant predicate or it is the conjunction of negated constant predicates. Thus it is easy to show that for two edge labels p_1, p_2 of operational graphs the question of the existence of a constant a satisfying p_1 and not satisfying p_2 can be decided at most $O(|f||g|)$ time. Hence the separator edges in U_g can be found in $O(|f|^2|g|^2)$ time. Afterwards the ancestor image of G in U_g can be extended to be a separator candidate (using the method explained in Page 125) in $O(g^2)$ time. Furthermore, by Corollary 7.9. the emptiness problem can be decided in linear time. Condition (ii) of Definition 8.2. can also be checked in $O(|f||g|^2)$ time. All together, the question that whether G is a separator or not can be decided in $O(|f|^2|g|^2)$ time. ■

8.1.3 Structural recursions in $SR(n.i., \vee)$

Next we show that the containment problem is coNP-complete for structural recursions in $SR(n.i., \vee)$. As it is well-known, the satisfiability problem of

propositional formulas in conjuctive normal form (CNF) is NP-complete in general [14]. Hence the non-satisfiability problem of a CNF formula is coNP-complete, thus the question of whether a DNF formula is a tautology or not is also coNP-complete. In the proof of coNP-hardness we will use this latter problem.

Let $P = C_1 \vee \ldots \vee C_m$ be a DNF formula, whose variables are $X_1, \ldots, X_n$. Here C_i is a conjunctive chain ($1 \leq i \leq m$). We may suppose that in each conjuction chain C_i, if both X_j and X_k appears in C_i, then X_j precedes X_k ($1 \leq j < k \leq n$). If it is not so, then the positions of X_j and X_k should be changed.

Recall the description of truth assignment paths on Page 115. Here, the *interpretation path* terminology will be used instead. For labels $x_1, \ldots, x_n$ an interpretation path is of the form: $l_1 \ldots l_n$, where l_i is either x_i or $\neg x_i$ ($1 \leq i \leq n$). An *interpretation path of* P is an interpretation path over the variables of P. Furthermore, an instance is called *interpretation instance of* P, if it contains an interpretation path of P as a pregraph.

First, we construct a structural recursion

$$f_P^{int} = (\{f_\flat, f_{x_1}, \ldots, f_{x_n}\}, \Sigma, \{f_\flat\}, \Gamma),$$

which returns a non-empty output if and only if its input is a $\flat$-labelled edge followed by an interpretation instance. The transformation rules are as follows:

$$\begin{array}{lll} f_{x_i}\colon (\{x_i : t\}) & = f_{x_{i+1}}(t) & \qquad f_{x_n}\colon (\{x_n : t\}) \quad = \{\psi : \{\}\} \\ \quad (\{\neg x_i : t\}) & = f_{x_{i+1}}(t) & \qquad\quad (\{\neg x_n : t\}) = \{\psi : \{\}\} \end{array}$$

$$f_\flat\colon \ (\{\flat : t\}) \quad = f_{x_1}(t) \ (1 \leq i \leq n-1).$$

The operational graph of f_P^{int} can be found in Figure 18.(a). The next lemma trivially holds.

Lemma 8.10. *Let P be a propositional formula. Then, for an arbitrary instance I, $f_P^{int}(I)$ is not empty $\Leftrightarrow$ I has an outgoing edge $\flat$ from its root followed by an interpretation instance of P.*

Next, we define a structural recursion f_P s.t. it returns a non-empty result only for those inputs whose root has an outgoing edge $\flat$ followed by an interpretation instance, where the represented interpretation satisfies P. In the definition of f_P we construct the representations of C_i-s first ($1 \leq i \leq m$). Instead of enumerating the transformation rules, in this case we rather describe the transformation rules by giving the appropriate subgraph of the operational graph. For a fixed C_i consider nodes $u_1^i, \ldots, u_{n+1}^i$. If X_j does not appear in C_i, then add an x_j and an $\neg x_j$ edge from u_j^i to u_{j+1}^i ($1 \leq j \leq n$). If X_j appears only without (with) negation, then a single x_j ($\neg x_j$) edge should be added. Finally, if X_j appears both with and without negation, then C_i cannot be satisfied, hence no edge is added between u_j^i and u_{j+1}^i. The edge between u_n and u_{n+1} should be defined to be a constructor edge. Next, to each u_j^i a label f_j^i should be assigned ($1 \leq j \leq n$).

In the last step these structural functions representing C_i-s should be connected. This is accomplished by means of structural function $f_\flat$, which is defined to be the only initial structural function of f_P. It has only one transformation rule:

$f_\flat$: $(\{\flat : t\})$= if n.i.$(f_1^1(t)) \vee \ldots \vee$ n.i.$(f_1^m(t))$ then $\{\psi : \{\}\}$.

As an example for f_P consider Figure 18.(b). Here $P = (X \wedge \neg Z) \vee (X \wedge Y \wedge Z)$, and $(f_3^1, \neg z, f_4^1), (f_3^2, z, f_4^2)$ are constructor edges.

Lemma 8.11. *Let P be a DNF formula and I an arbitrary instance. Then $f_P(I)$ is not empty $\Leftrightarrow$ I has an outgoing $\flat$-labelled edge from its root followed by an interpretation instance of P s.t. the encoded interpretation satisfies P.*

Proof. Suppose that P is of the form $C_1 \vee \ldots \vee C_m$. Clearly, path $pa = l_1 \ldots l_n$ results a non-empty output for structural recursion $(\{f_1^i, \ldots, f_n^i\}, \Sigma, \{f_1^i\}, \Gamma)$ (representing C_i) if and only if pa encodes an interpretation that satisfies C_i ($1 \leq i \leq m$). From this observation the statement of the lemma straightforwardly follows. ■

Theorem 8.12. *Let P be a DNF formula, then P is a tautology $\Leftrightarrow$ f_P contains f_P^{int}.*

Proof. Suppose first that P is a tautology. Then from Lemma 8.11. it follows that every instance which has an outgoing $\flat$-labelled edge followed by an arbitrary interpretation instance of P results a non-empty output, when it is processed by f_P. From Lemma 8.10. we know that except from those instances that have an outgoing $\flat$-labelled edge from their root followed by an interpretation instance of P f_P^{int} returns an empty output for all other instances. This implies that f_P contains f_P^{int}.

Assume now that f_P contains f_P^{int}, and suppose that P is not a tautology. Then there is an interpretation that does not satisfy P. Denote pa the path encoding this interpretation. By Lemma 8.11. for path $pa' = \flat.pa$, $f_P(pa')$ is empty. On the other hand, since pa is an interpretation instance of P, f_P^{int} is not empty. A contradiction. ■

With $SR(i., \wedge)$ we denote the class of those structural recursions, whose conditional form is without not-isempty conditions and disjunctions.

Corollary 8.13. *The containment problem for structural recursions in $SR(n.i., \vee)$ is coNP-complete. It is also coNP-complete for structural recursions in $SR(i., \wedge)$.*

Proof. Let f and g be structural recursions in $SR(n.i., \vee)$. From Theorem 8.7. we know that f contains g if and only if there is not any separator in $U_f \sqcap U_g$. Since a separator is a pregraph of $U_f \sqcap U_g$, its size is trivially polynomial in the sizes of f and g. Moreover, for a given pregraph it can be checked in polynomial time, whether it is a separator or not. This proves that the containment problem is in coNP. The coNP-hardness has just been proven in Theorem 8.12. For the proof of the second statement it is enough to note that f contains g if and only if $\tilde{g}$ contains $\tilde{f}$. Furthermore, the complement of a structural recursion in $SR(n.i., \vee)$ is in $SR(i., \wedge)$. ■

Remark 8.14. Note that f_P^{int} is deterministic. Thus the containment problem is still coNP-complete, when the contained structural recursion is deterministic and the "container" structural recursion is in $SR(n.i., \vee)$.

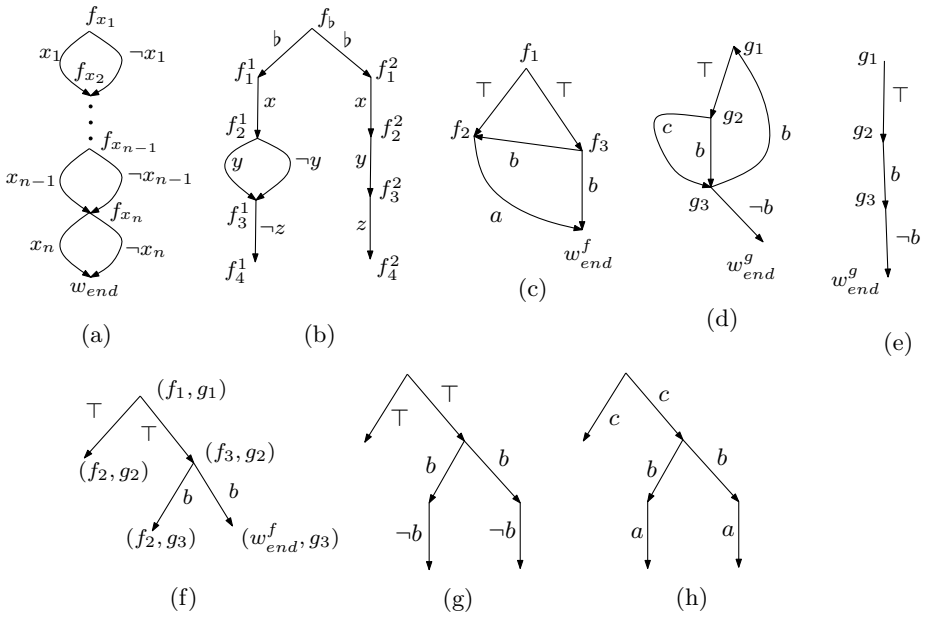

Figure 18: (a) f_P^{int}. (b) An example for f_P ($P = (X \wedge \neg Z) \vee (X \wedge Y \wedge Z)$).

8.2 Structural recursions in $SR(n.i.)$

In this section it will be shown that the containment problem of structural recursions in $SR(n.i.)$ is PSPACE-hard in general. However, before giving the proof we formulate our conjecture that the containment problem is PSPACE-complete indeed.

Claim 8.15. *The containment problem of structural recursions in $SR(n.i.)$ lies in PSPACE.*

It is easy to see that it would be enough to show that for arbitrary structural recursions f, g in $SR(n.i.)$, if f does not contain g, then there is a tree I s.t. $f(I)$ is empty, whereas $g(I)$ is not empty, moreover, each path of I from the root to a leaf is no longer than M, where M can be expressed as a polynom of $|f|$. Namely, suppose that this statement is true. Denote t the possible run of g and f constructed from I. Here, $N.I.(u_0)$ is g, whereas $I.(u_0)$ is f, where u_0 denotes the root of t. From the construction method of this possible run follows that the length of an arbitrary path in t from the root to a leaf

is at most M. Consequently, the complement of the containment problem can be solved by a Turing machine working in NPSPACE (consider the proof of Proposition 7.17.). However, complexity class NPSPACE is closed for the complement, i.e., if a problem lies in NPSPACE, then its complement also lies in NPSPACE [26]. Furthermore, it is a well-known fact that NPSPACE coincides with PSPACE [26].

The containment problem is PSPACE-hard. To prove the PSPACE-hardness the quantified Boolean formula problem will be reduced to the containment problem of structural recursions in $SR(n.i.)$. In this case the question is that for a given formula

$$P = \exists X_1 \forall X_2 \exists X_3 \ldots Q X_n \phi,$$

whether there is a truth assignment of variable X_1 s.t. for all truth assignments of variable X_2 there is a truth assignment of variable X_3 and so on up to X_n s.t. ϕ is satisfied by the overall truth assignment. It is well-known that this problem is PSPACE-complete in general [26].

P is represented by structural recursion

$$f^P = (\{f_\flat, f_{x_1}, f_{\neg x_1}, \ldots, f_{x_n}, f_{\neg x_n}, \ldots\}, \Sigma, \{f_\flat\}, \Gamma),$$

whose transformation rules are as follows:

$f_\flat$: $(\{\flat : t\}) = \text{if n.i.}(f_{x_1}(t)) \vee \text{n.i.}(f_{\neg x_1}(t)) \text{ then } \{\psi : \{\}\}$

f_{x_i}: $(\{x_i : t\}) = \text{if n.i.}(f_{x_{i+1}}(t)) \vee \text{n.i.}(f_{\neg x_{i+1}}(t)) \text{ then } \{\psi : \{\}\}$, if i is even
$f_{\neg x_i}$: $(\{\neg x_i : t\}) = \text{if n.i.}(f_{x_{i+1}}(t)) \vee \text{n.i.}(f_{\neg x_{i+1}}(t)) \text{ then } \{\psi : \{\}\}$, if i is even

f_{x_i}: $(\{x_i : t\}) = \text{if n.i.}(f_{x_{i+1}}(t)) \wedge \text{n.i.}(f_{\neg x_{i+1}}(t)) \text{ then } \{\psi : \{\}\}$, if i is odd
$f_{\neg x_i}$: $(\{\neg x_i : t\}) = \text{if n.i.}(f_{x_{i+1}}(t)) \wedge \text{n.i.}(f_{\neg x_{i+1}}(t)) \text{ then } \{\psi : \{\}\}$, if i is odd

f_{x_n}: $(\{x_n : t\}) = \text{if } Form(\phi) \text{ then } \{\psi : \{\}\}$
$f_{\neg x_n}$: $(\{\neg x_n : t\}) = \text{if } Form(\phi) \text{ then } \{\psi : \{\}\}$ $(1 \leq i \leq n-1)$.

Remember that the existential variables are indexed by odd, whereas the universal variables by even numbers.

Remark 8.16. Suppose that there is an instance that results a non-empty output for f^P. Then denote I a minimal, root-edged tree instance to which $f^P(I)$ is still not empty. (Proposition 4.27., Corollary 7.2. and Theorem 4.38. guarantees the existence of this tree.) Clearly, the root edge should be labelled with $\flat$, which should be followed by either an x_1 or an $\neg x_1$ edge. Furthermore, both of these edges should have outgoing edges $x_2, \neg x_2$ etc. In other words each path from the root of I to an edge x_n, $\neg x_n$ of this sequence is a truth assignment path. What is more each of them is followed by a truth assignment star s.t. the encoded truth assignment satisfies ϕ. However, these truth assignment stars not necessarily encodes the same truth assignments as the truth assignment paths which precede them. If each of them encoded the same truth assignment, then I would be a $\exists X_1 \forall X_2 \exists X_3 \dots Q X_n$-assignment instance (Page 116).

This observation implies the following lemma.

Lemma 8.17. *For a given quantified Boolean formula*

$$P = \exists X_1 \forall X_2 \exists X_3 \dots Q X_n \phi,$$

if there is a $\exists X_1 \forall X_2 \exists X_3 \dots Q X_n$-assignment instance I for P s.t. $\{\flat : I\}$ results a non-empty output for f^P, then P is satisfiable.

Next, we construct a structural recursion with which we can enforce the matching of the aforementioned truth assignment paths and stars in Remark 8.16. The structural recursion checks whether an x_i ($\neg x_i$) labelled edge has a $\neg x_i$ (x_i) descendant or not. It is of the following form:

$$f^n = (\{f_\flat, f_{x_1,\neg x_1}, \dots, f_{x_n,\neg x_n}, \dots\}, \Sigma, \{f_\flat\}, \Gamma),$$

where the transformation rules are:

$$f_\flat: \qquad (\{\flat : t\}) \quad = \text{if n.i.}(f_{x_1,\neg x_1}(t)) \vee \dots \vee \text{n.i.}(f_{x_n,\neg x_n}(t)) \text{ then } \{\psi : \{\}\}$$

$$
\begin{aligned}
f_{x_i,\neg x_i}\colon(\{x_i : t\}) &= \text{if n.i.}(f^{desc}_{\neg x_i}(t)) \text{ then } \{\psi : \{\}\}\\
(\{\neg x_i : t\}) &= \text{if n.i.}(f^{desc}_{x_i}(t)) \text{ then } \{\psi : \{\}\}\\
(\{* : t\}) &= f_{x_i,\neg x_i}(t)\ (1 \leq i \leq n), \text{ where}
\end{aligned}
$$

$$
\begin{aligned}
f^{desc}_{x_i}\colon\ (\{x_i : t\}) &= \{\psi : \{\}\}\\
(\{* : t\}) &= f^{desc}_{x_i}(t)
\end{aligned}
$$

$$
\begin{aligned}
f^{desc}_{\neg x_i}\colon\ (\{\neg x_i : t\}) &= \{\psi : \{\}\}\\
(\{* : t\}) &= f^{desc}_{\neg x_i}(t)
\end{aligned}
$$

Here, obviously $f^{desc}_{x_i}$ ($f^{desc}_{x_i}$) returns a non-empty output for an instance, if it has an x_i ($\neg x_i$) labelled descendant ($1 \leq i \leq n$). (†) Hence, it is easy to see that f^n returns a non-empty result for an instance I, if it has an outgoing $\flat$ edge from the root that have an x_i ($\neg x_i$) descendant, which in turn has a $\neg x_i$ (x_i) descendant.

Theorem 8.18. *Let P be a quantified Boolean formula. Then f^n does not contain $f^P \Leftrightarrow P$ is satisfiable.*

Proof. First suppose that there is an instance I s.t. $f^P(I)$ is not empty, while $f^n(I)$ is empty. As in Remark 8.16. we may assume that I is a root-edged, tree s.t. neither of its pregraphs return a non-empty result for f^P. By Remark 8.16. I consists of truth assignment paths followed by truth assignment stars which do not necessarily encode the same truth assignment. However, since $f^n(I)$ is empty, (†) implies that in this case they do encode the same truth assignment, hence I must be a $\flat$-labelled edge followed by an $\exists X_1 \forall X_2 \exists X_3 \ldots QX_n$-assignment instance. From Lemma 8.16. it follows now that P is satisfiable.

If P is satisfiable, then fix the truth values of the existential variables s.t. every truth assignment which orders these truth values to the existential variables satisfy P. Denote I' the corresponding $\exists X_1 \forall X_2 \exists X_3 \ldots QX_n$-assignment instance. Consider instance I of the form $\{\flat : I'\}$. From the

previous reasoning it follows that $f^P(I)$ is not empty, while $f^n(I)$ is empty, in other words f^n does not contain f^P. ■

9 Summary and future work

In the dissertation we have introduced and then examined some of the mathematical properties of structural recursions working on edge-labelled graphs. We have differentiated several important classes of structural recursions and compared their expressive power. Besides, the complexity of the emptiness and containment problems for these classes has been also determined, nevertheless some details still need to be clarified. Containment has been characterized by using operational homomorphism, the developed techniques and results were extensively used in the foregoing reasonings.

Furthermore, we have developed algorithms by means of which for a given structural recursion and data graph a data tree can be constructed, whose processing by the structural recursion in question simulates the processing of the data graph. Among others, for a data graph returning a non-empty output a data tree can be constructed also having this property. This observation enabled us to examine the relationship between alternating tree automata and structural recursions. It has turned out that structural recursions can be simulated by alternating tree automata and vice versa. Beside alternating tree automata structural recursions were also compared with non-deterministic, finite state automata. Finally but not lastly, the usual operations, i.e., complement ($\tilde{}$), union ($\sqcup$), intersection ($\sqcap$), have been defined.

The most important task would be to fully determine the complexity of those static analytical questions (containment for $SR(n.i.)$, emptiness and containment for $SR(n.i., i., \leq k)$), where we only have partial results. Secondly, the correspondences of the classes of structural recursions should be defined for alternating tree automata. Then, the aforementioned complexity results could also be applied to them. To the best of our knowledge up to now no similar examination of alternating automata has been published,

hence these results would be new in the field.

References

[1] Serge Abiteboul and Catriel Beeri. The Power of Languages for the Manipulation of Complex Values. *The VLDB Journal*, 4(4):727–794, 1995.

[2] Serge Abiteboul, Pierre Bourhis, and Victor Vianu. Highly Expressive Query Languages for Unordered Data Trees. In *Proceedings of the 15th International Conference on Database Theory*, ICDT '12, pages 46–60. ACM, 2012.

[3] Serge Abiteboul, Peter Buneman, and Dan Suciu. *Data on the Web: From Relations to Semistructured Data and XML*. Morgan Kaufmann Publishers, 2000.

[4] András Benczúr and Balázs Kósa. Static Analysis of Structural Recursion in Semistructured Databases and Its Consequences. In *ADBIS*, pages 189–203, 2004.

[5] Geert Jan Bex, Sebastian Maneth, and Frank Neven. A Formal Model for an Expressive Fragment of XSLT. *Inf. Syst.*, 27(1):21–39, 2002.

[6] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, François Yergeau, and John Cowan. Extensible Markup Language (XML) 1.1 W3C Recommendation, The World Wide Web Consortium. `http://www.w3.org/TR/2004/REC-xml11-20040204/`, 2004.

[7] Val Breazu-Tannen, Peter Buneman, and Shamim Naqvi. Structural recursion as a query language. In *Proceedings of the third international workshop on Database programming languages : bulk types & persistent data: bulk types & persistent data*, DBPL3, pages 9–19. Morgan Kaufmann Publishers Inc., 1992.

[8] Peter Buneman, Susan Davidson, Mary Fernandez, and Dan Suciu. Adding structure to unstructured data. In *In 6th Int. Conf. on Database Theory (ICDT '97), LNCS 1186*, pages 336–350. Springer, 1997.

[9] Peter Buneman, Mary Fernandez, and Dan Suciu. UnQL: a Query Language and Algebra for Semistructured Data Based on Structural Recursion. *The VLDB Journal*, 9:76–110, 2000.

[10] Peter Buneman, Shamim Naqvi, Val Tannen, and Limsoon Wong. Principles of Programming with Complex Objects and Collection Types. *Theor. Comput. Sci.*, 149(1):3–48, 1995.

[11] James Clark. XSL Transformations (XSLT) Version 1.0 W3C Recommendation, The World Wide Web Consortium, 1999.

[12] Huber Comon, Max Dauchet, Rémi Gilleron, Florent Löding, Christofand Jacquemard, Denise Lugiez, Sophie Tison, and Marc Tommasi. Tree Automata Techniques and Applications. Available on: `http://www.grappa.univ-lille3.fr/tata`, 2007.

[13] Alin Deutsch, Liying Sui, and Victor Vianu. XML Path Language (XPath) Version 1.0 W3C Recommendation, The World Wide Web Consortium. `http://www.w3.org/TR/xpath/`, 1999.

[14] Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness.* W. H. Freeman & Co., 1990.

[15] Georg Gottlob, Christoph Koch, and Reinhard Pichler. Efficient Algorithms for Processing XPath Queries. *ACM Trans. Database Syst.*, 30(2):444–491, 2005.

[16] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Prentice Hall, 2006.

[17] Neil Immerman, Sushant Patnaik, and David Stemple. The Expressiveness of a Family of Finite Set Languages. *Theoretical Computer Science*, 155(1):111 – 140, 1996.

[18] Kazuhiro Inaba, Soichiro Hidaka, Zhenjiang Hu, Hiroyuki Kato, and Keisuke Nakano. Graph-transformation Verification Using Monadic Second-order Logic. In *Proceedings of the 13th international ACM SIGPLAN symposium on Principles and practices of declarative programming*, pages 17–28, 2011.

[19] Balás Kósa, András Benczúr, and Attila Kiss. An Efficient Implementation of an Expressive Fragment of XPath for Typed and Untyped XML Documents Using Extended Structural recursion. In *Local Proceedings ADBIS 2009*, pages 156–176, 2009.

[20] Balás Kósa, András Benczúr, and Attila Kiss. Satisfiability and Containment Problem of Structural Recursions with Conditions. In *Proceedings of the 14th east European conference on Advances in databases and information systems*, pages 336–350. Springer-Verlag, 2010.

[21] Balázs Kósa. Containment and Satisfiability Problem for XPath with Recursion. In *Proceedings of the 16th East European conference on Advances in Databases and Information Systems*, pages 240–253. Springer-Verlag, 2012.

[22] Balázs Kósa, András Benczúr, and Attila Kiss. Extended Structural Recursion and XSLT. *Acta Univ. Sapientiae, Inform.*, 1(2):165–213, 2009.

[23] Leonid Libkin and Limsoon Wong. Query languages for bags and aggregate functions. *J. Comput. Syst. Sci.*, 55(2):241–272, 1997.

[24] Wim Martens, Frank Neven, Thomas Schwentick, and Geert Jan Bex. Expressiveness and Complexity of XML Schema. *ACM Trans. Database Syst.*, 31:770–813, 2006.

[25] Robert Paige and Robert E. Tarjan. Three Partition Refinement Algorithms. *SIAM J. Comput.*, 16(6):973–989, 1987.

[26] Christos M. Papadimitriou. *Computational Complexity.* Addison-Wesley, 1994.

[27] Edward L. Robertson, Lawrence V. Saxton, Dirk Van Gucht, and Stijn Vansummeren. Structural Recursion on Ordered Trees and List-based Complex Objects. In *Proceedings of the 11th international conference on Database Theory*, ICDT'07, pages 344–358. Springer-Verlag, 2006.

[28] Thomas Schwentick. XPath Query Containment. *SIGMOD Rec.*, 33(1):101–109, 2004.

[29] Dan Suciu and Limsoon Wong. On two Forms of Structural Recursion. *Database Theory ICDT '95*, 893:111–124, 1995.

[30] Robert Tarjan. Depth-First Search and Linear Graph Algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[31] Katalin Pásztorné Varga and Magda Várterész. *A matematikai logika alkalmazás szemléletü tárgyalása.* Panem, 2003.

Appendix

A 3.1

Proposition 3.5. Let $f = (\{f_1, \ldots, f_n\}, \Sigma, F_I, \Gamma)$ be a simple structural recursion. Then for all data tree $\hat{t}$, $[\![f(\hat{t})]\!]_{nat}$ is equivalent to $[\![f(\hat{t})]\!]_{op}$.

Proof. We will prove that for each structural function f_i, $\varepsilon^{nat}[\![f_i(\hat{t})]\!]$ is equivalent with $[\![f^i(\hat{t})]\!]_{op}$ $(1 \leq i \leq n)$. Here $f^i = (\{f_1, \ldots, f_n\}, \Sigma, \{f_i\}, \Gamma)$, in other words f^i is the same as f, only the set of initial structural functions is changed to $\{f_i\}$.

To see why it is enough to prove the preceding statement first note that the disjunctive union constructor can be straightforwardly extended to data graphs. Obviously, if $f_{i_1}, \ldots, f_{i_k}$ are the initial structural functions of f, then $[\![f(\hat{t})]\!]_{op}$ could have also been defined as

$$[\![f^{i_1}(\hat{t})]\!]_{op} \oplus \ldots \oplus [\![f^{i_k}(\hat{t})]\!]_{op}.$$

On the other hand $[\![f(\hat{t})]\!]_{nat}$ was defined as

$$\varepsilon^{nat}[\![f_{i_1}(\hat{t})]\!] \oplus \ldots \oplus \varepsilon^{nat}[\![f_{i_k}(\hat{t})]\!].$$

Hence, if we show that $\varepsilon^{nat}[\![f_{i_j}(\hat{t})]\!]$ is equivalent with $[\![f^{i_j}(\hat{t})]\!]_{op}$ $(1 \leq j \leq k)$, then the equivalence of $[\![f(t)]\!]_{nat}$ and $[\![f(t)]\!]_{op}$ will be a trivial consequence of this observation.

In the proof we use structural function f_1 and f^1, however the statement can be proven for the rest of the structural functions in the same way. We use induction on the number of edges k of $\hat{t}$. In the base case suppose that $k = 0$. This means that $\hat{t}$ is the empty graph. Then by definition both $\varepsilon^{nat}[\![f_1(\hat{t})]\!]$ and $[\![f^1(\hat{t})]\!]_{op}$ are equal to the empty graph.

Suppose that the statement holds for $k = n$ and let k be $n + 1$. Assume first that $\hat{t} = t_1 \cup t_2$. By definition

$$\varepsilon^{nat}[\![f_1(\hat{t})]\!] = \varepsilon^{nat}[\![f_1(t_1)]\!] \cup \varepsilon^{nat}[\![f_1(t_2)]\!].$$

From the induction hypothesis we know that $\varepsilon^{nat}[\![f_1(t_i)]\!]$ is equivalent with $[\![f^1(t_i)]\!]_{op}$ $(i = 1, 2)$. Now note that when the basic forests are connected with ε edges in the operational semantics, the order of these connections is irrelevant. Therefore $[\![f^1(\hat{t})]\!]_{op}$ can be constructed by first constructing $[\![f^1(t_1)]\!]_{op}$ and $[\![f^1(t_2)]\!]_{op}$. Clearly, the roots of these trees are labelled with the same label, hence according to the construction rules, in $[\![f^1(\hat{t})]\!]_{op}$ their union is taken.

Assume that $\hat{t} = \{a : t\}$. Denote $frst$ the forest to be constructed in the transformation rule of f_1 according to which the root edge of $\hat{t}$ (with label a) should be processed. Obviously, this is the same transformation

rule in both semantics. Suppose that the leaves of $frst$ are labelled with labels $f_{j_1}(t), \ldots, f_{j_r}(t)$. If the root edge were processed by the default rule, then in both semantics the edge labels $*$ should be substituted with edge labels a. For sake of convenience we also denote with $frst$ this new forest. According to our induction hypothesis each $\varepsilon^{nat}[\![f_{j_i}(t)]\!]$ is equivalent with $[\![f^{j_i}(t)]\!]_{op}$ $(1 \leq i \leq r)$. Clearly, the set of $f_{j_i}(t)$-labelled leaves of $frst$ in the natural semantics is the same as the set of $(1, f_{j_i}, v)$-labelled leaves of $frst$ in the operational semantics. In both semantics $\varepsilon^{nat}[\![f_{j_i}(t)]\!]$ and $[\![f^{j_i}(t)]\!]_{op}$ should be connected to each element of this set, hence the final results are trivially equivalent to each other. ■

A.4.2.

Proposition 4.16. Let $f = (\{f_1, \ldots, f_n\}, \Sigma, F_I, \Gamma)$ be an arbitrary structural recursion in $SR()$. Then $\mathcal{L}(f) = \mathcal{L}(Det(f))$.

Note that by Proposition 4.2. we may assume that f is complete. In order to prove the proposition we formulate two lemmas. Consider structural functions $f_{i_1}, \ldots, f_{i_k}$ of f. Let $a_1, \ldots, a_m$ be constants in Σ. Then denote $A_i^{i_1,\ldots,i_k}$ the list of those indexes $\langle j_1, \ldots, j_s \rangle$ $(\{j_1, \ldots, j_s\} \subseteq \{1, \ldots, n\})$, where for each j_r there is an i_o s.t. there is a directed path with labels $p_1 \ldots p_m$ from f_{i_o} to f_{j_r} in U_f for which $p_l(a_l)$ is true $(1 \leq o \leq k, 1 \leq r \leq s, 1 \leq l \leq m)$.

Lemma 9.1. *For an arbitrary, complete structural recursion*

$$f = (\{f_1, \ldots, f_n\}, \Sigma, F_I, \Gamma)$$

in $SR()$ and constants $a_1, \ldots a_m$ in Σ consider structural functions $f_{i_1}, \ldots, f_{i_k}$ of f. $A_i^{i_1,\ldots,i_k} = \langle j_1, \ldots, j_s \rangle \Leftrightarrow f_{j_1,\ldots,j_s}$ is reachable from $f_{i_1,\ldots,i_k}$ in $U_{Det(f)}$ through a directed path with labels $p_1 \ldots p_m$, where $p_i(a_i)$ is true $(\{i_1, \ldots, i_k\}$, $\{j_1, \ldots, j_s\} \subseteq \{1, \ldots, n\}$, $1 \leq i \leq m)$.

Proof. First, we formulate another statement, which is a straightforward consequence of the construction rules of $Det(f)$. Consider structural function f_S of $Det(f)$, where S denotes a list of indexes.

(†) For an arbitrary index j in S, structural function f_r of f and symbol ϑ in $\Sigma \cup \{*\}$ to which a transformation rule of f_j belongs in f, f_r is called in this transformation rule if and only if r is in the list of indexes P, where f_P is called in the transformation rule of f_S belonging to ϑ.

Now, the lemma can be proven by applying an induction on i. Both the base and the general step can shown in an obvious way by using (†). ■

Lemma 9.2. *Let $f_{i_1}, \ldots, f_{i_k}$ be the initial structural functions of a given complete structural recursion f, and let $a_1, \ldots, a_m$ be constants in Σ. Then there is a path with labels $p_1 \ldots p_m$ from f_{i_j} to a constructor edge in U_f s.t. $p_r(a_r)$ is true $\Leftrightarrow$ there is also a path with the same labels $p_1 \ldots p_m$ from $f_{i_1,\ldots,i_k}$ to a constructor edge in $U_{Det(f)}$ $(1 \leq j \leq k, 1 \leq r \leq m)$.*

Proof. Without the requirement that the paths must end in a constructor edge Lemma 9.2. would be a simple reformulation of Lemma 9.1. Therefore, in order to prove this lemma, statement (†) of the former proof should be extended to involve constructor edges. However, as we will see this extension is again a straightforward consequence of the construction rules of $Det(f)$.

(‡) Let S be a list of indexes s.t. f_S is a structural function of $Det(f)$ and let ϑ be a symbol in $\Sigma \cup \{*\}$ to which a transformation rule of f_S belongs. Then a ψ edge is constructed in this transformation rule if and only if there is a j in S s.t. in $\gamma_{j,\vartheta}$ a non-empty graph is constructed, where $\gamma_{j,\vartheta}$ is the transformation rule of f_j for ϑ in f.

Using (‡) the lemma can be proven by a straightforward induction on m. ■

Returning to the proof of Proposition 4.16., let f be a structural recursion in $SR()$ and I an instance. Lemma 9.1. and 9.2. imply that there is a directed path from a root of $U_f \sqcap I$ with labels $a_1 \ldots a_m$ ending in a constructor edge if and only if there is a directed path with the same labels from the root of $U_{Det(f)} \sqcap I$, whose last edge is also a constructor edge. This observation proves the proposition.

A 4.3.

Lemma 4.30. For structural recursions f_1, f_2 in $SR(n.i., i., el)$, instances I_1, I_2, if there is a surjective operational homomorphism ρ from $V.U_{f_1} \sqcap I_1$ to $V.U_{f_2} \sqcap I_2$ s.t. the inverse of ρ, ρ^{-1}, is also an operational homomorphism, then for an arbitrary then- or else-edge $e \in E.U_{f_1} \sqcap I_1$, if e is deleted in the n^{th} step of condition evaluation, then $\rho(e)$ is also deleted in this step.

Proof. As in the proof of Lemma 4.24. we use induction on n. Denote γ_1, γ_2 the transformation rules of e and $\rho(e)$ respectively. Suppose first that $n = 1$.

Assume first that e is a then-edge. In this case its deletion means that there are premises $pr_1, \ldots pr_k$ in $Pr(\gamma_1)$ that have become $false$ and as a consequence $Form(\gamma_1)$ also becomes $false$. Let pr be a premise in $\{pr_1, \ldots, pr_k\}$. Suppose first that pr belongs to an i. condition. Then there is a path pa from pr to a constructor edge in $U_{f_1} \sqcap I_1$ without conditional edges. Clearly, $\rho(pa)$ is also a path from $\rho(pr)$ to a constructor edge without conditional edges, thus the corresponding i. condition also becomes $false$.

Assume now pr belongs to a n.i. condition. Remember that pr may become $false$ in two ways in this case: (a) there is not any constructor edge reachable from pr, or (b) each of the paths through which a constructor edge is reachable contains at least one then- or else-edge, to which the corresponding condition cannot be evaluated, because the truth value of some of its premises cannot be decided, i.e., property (b) also holds for these premises.

Suppose that a premise pr has become false owing to case (a). If we assumed now that there exists a constructor edge reachable from $\rho(pr)$ through path pa, then, since ρ is surjective and ρ^{-1} is also an operational homomorphism, using induction on the number of the edges of pa and condition (ii) of Definition 4.22. it could be shown that there also exists a path from pr to a constructor edge resulting a contradiction.

In case (b) step (iii) of condition evaluation should be applied (Page 35). This means that pr belongs to a strongly connected component in which the truth value of conditions mutually depends on each other. Furthermore, this

strongly connected component is a leaf of the tree representing the hierarchy of such strongly connected components. Since both ρ and ρ^{-1} are operational homomorphisms $\rho(pr)$ also belongs to such a strongly connected component in $U_{f_2} \sqcap I_2$, hence it also becomes $false$ in this step.

All in all, we get that for all i $\rho(pr_i)$ also becomes $false$ $(1 \leq i \leq k)$. From condition (iv) of Definition 4.22. it follows that $Form(\gamma_2)$ becomes $false$ too, thus $\rho(e)$ should also be deleted.

The case, when e is an else-edge, can be proven using the same observations. This concludes the proof of the base case.

The proof of the general case is very similar to that of the proof of Lemma 4.24. Here, however, due to the symmetric nature of the lemma statement (†) becomes simpler.

(†) After each step of conditional evaluation a surjective operational homomorphism can be defined between $V.G_1$, $V.G_2$ s.t. its inverse is also an operational homomorphism. Here, G_i denotes the graph that remains from $U_{f_i} \sqcap I_i$ after the aforementioned step of condition evaluation $(i = 1, 2)$.

We have just proved that if an arbitrary then- or else-edge e, is to be deleted, then its counterpart $\rho(e)$ or $\rho^{-1}(e)$ should also be deleted in the same step. Thus, to get the operational homomorphism of (†) simply ρ should be restricted to those nodes of $V.U_{f_1} \sqcap I_1$ and $V.U_{f_2} \sqcap I_2$ that are reachable from a root.

With (†) the general case can be proven exactly in the same way as the first step of the induction. ■

A 5.2.

Proposition 5.17. Let $f = (\{f_1, \ldots, f_n\}, \Sigma, F_I, \Gamma_f)$ and $g = (\{g_1, \ldots, g_m\}, \Sigma, G_I, \Gamma_g)$ be two structural recursions. Then

(i) $\widetilde{f \sqcup g}$ is equivalent to $\tilde{f} \sqcap \tilde{g}$.

(ii) $\widetilde{f \sqcap g}$ is equivalent to $\tilde{f} \sqcup \tilde{g}$.

Proof. A stronger result will be proven. Namely, it will be shown that the appropriate structural recursions are not only equivalent but they are essentially the same syntactically. The proof is very similar for both cases, hence we only prove statement (i). Denote f' ang g' the CCNF rewriting of f and g respectively. It is easy to see that since both union and intersection are defined over structural recursions in CCNF except from some notational difference of the structural functions $\widetilde{f \sqcup g}$ and $\tilde{f} \sqcap \tilde{g}$ is syntactically the same as $\widetilde{f' \sqcup g'}$ and $\tilde{f'} \sqcap \tilde{g'}$ respectively. Thus, we may assume that both f and g are in CCNF.

First note that

$$\widetilde{f \sqcup g} = (\{f_{i \sqcup j}, \widetilde{f_{i \sqcup j}}, f_1, \ldots, f_n, g_1, \ldots, g_m\}, \Sigma, \{\widetilde{f_{i \sqcup j}}\}, \Gamma_{\widetilde{i \sqcup j}}) \text{ and}$$
$$\tilde{f} \sqcap \tilde{g} = (\{f_{\tilde{i} \sqcap \tilde{j}}, \tilde{f}_i, \tilde{g}_j, f_1, \ldots, f_n, g_1, \ldots, g_m\}, \Sigma, \{f_{\tilde{i} \sqcap \tilde{j}}\}, \Gamma_{\tilde{i} \sqcap \tilde{j}}).$$

Here f_i, g_j are the initial structural functions of f and g respectively. Note that neither $f_{i \sqcup j}$ nor $\tilde{f}_i, \tilde{g}_j$ are called by any of the structural functions of $\widetilde{f \sqcup g}$ and $\tilde{f} \sqcap \tilde{g}$, hence they can be deleted. Afterwards, apart from the initial structural functions of $\widetilde{f \sqcup g}$ and $\tilde{f} \sqcap \tilde{g}$ the rest of the structural functions will be the same.

Let a be an arbitrary symbol in $\Sigma \cup \{*\}$ s.t. both $\gamma^f_{i,a}, \gamma^g_{j,a}$ exist. (We may assume that if $\gamma^f_{i,a}$ is given, then $\gamma^g_{j,a}$ is also given and vice versa. Otherwise define $\gamma^g_{j,a}$ to be $\gamma^g_{j,*}$, which must exist since g is in CCNF.)

$Form(\tilde{\gamma}_{i \sqcup j,a}) = \neg(Form(\gamma^f_{i,a}) \vee Form(\gamma^g_{j,a})) = \neg Form(\gamma^f_{i,a}) \wedge \neg Form(\gamma^g_{j,a}) = Form(\tilde{\gamma}^f_{i,a}) \wedge Form(\tilde{\gamma}^g_{j,a}) = Form(\gamma_{\tilde{i} \sqcap \tilde{j}})$.

In other words the formulae of the initial structural functions are the same.

In the last step it should be proven that $Frst(\tilde{\gamma}_{i \sqcup j,a})$ is the same as $Frst(\gamma_{\tilde{i} \sqcap \tilde{j},a})$. Since by definition $\widetilde{f \sqcup g}$ and $\tilde{f} \sqcap \tilde{g}$ are in CCNF the aforementioned forest are either consist of a single edge ψ or they are the empty graph. This depends on the form of the transformation rules $\gamma^f_{i,a}$ and $\gamma^g_{j,a}$. Since we have supposed that f and g are in CCNF, for both transformation rules three different cases should be considered. Namely, no condition has

been given and either an edge ψ or nothing is to be constructed, or there is a condition (in this case an edge ψ is surely constructed). All together nine different cases should be considered. We will examine only a single possibility here, the rest of the cases can be handled in a similar, straightforward manner.

If no condition is given in any of the two transformation rules and in $\gamma^f_{i,a}$ an edge ψ, while in $\gamma^g_{j,a}$ nothing is constructed, then

$$Frst(\gamma_{i\sqcup j,a}) = Frst(\gamma^f_{i,a}) \sqcup Form(\gamma^g_{j,a})$$

is an edge ψ, $\gamma_{i\sqcup j,a}$ does not have a condition, hence $Frst(\tilde{\gamma}_{i\sqcup j,a})$ should be the empty graph. On the other hand, $Frst(\tilde{\gamma}^f_{i,a})$ is the empty graph, whereas $Frst(\tilde{\gamma}^g_{j,a})$ is an edge ψ thus

$$Frst(\gamma_{\tilde{i}\sqcap\tilde{j},a}) = Frst(\tilde{\gamma}^f_{i,a}) \sqcap Frst(\tilde{\gamma}^g_{j,a})$$

is the empty graph.

To sum up we have shown that both the conditions and the then-branches of the initial structural functions of $\widetilde{f \sqcup g}$ and $\tilde{f} \sqcap \tilde{g}$ are the same syntactically (by definition neither of them may have an else-branch), which concludes the proof. ■

A 6.1.1.

Proposition 6.2. For given structural recursion in $SR()$ f and instance I, $f(I)$ is not empty $\Leftrightarrow$ I contains a path pa as a pregraph s.t. A_f accepts w_{pa}.

Lemma 9.3. *Let $f = (\{f_1, \ldots, f_n\}, \Sigma, F_I, \Gamma)$ be a structural recursion in $SR()$ and $pa = (u_1, a_1, v_1) \ldots (u_m, a_m, v_m)$ a path. Then, $((f_i, u_k), a_k, (\vartheta, v_k))$ $(\vartheta \in \{f_1 \ldots, f_n, w_{end}\})$ is an edge in $U_f \sqcap pa \Leftrightarrow$ there exists a run $q_{i_0}, \ldots, q_{i_k}$ of A_f on w_{pa} s.t. $q_{i_{k-1}} = q_i$ and*

(i) $q_{i_k} = q_j$, if $\vartheta = f_j$ $(1 \leq j \leq n)$,

(ii) $q_{i_k} = q_{accept}$, if $\vartheta = w_{end}$ or f_j and there is a construction in $frst$, where $frst$ is the tree constructed γ_{i,a_k},

(iii) $q_{i_k} = q_{deny}$, *if* $\vartheta = w_{end}$ *and there is no construction in* $frst$ $(1 \leq i \leq n, 1 \leq k \leq m)$.

Proof. We prove the statement by using induction on the number of edges of pa. Suppose first that this number is equal to 1. Clearly, in this case $f_i \in F_I$, consequently $q_i \in Q_I^f$. Assume now that (f_i, a, ϑ) is in $U_f \sqcap pa$. (i) If $\vartheta = f_j$, then according to the construction of transition rules of A_f $a_1(q_i) \to q_j$ is in Φ^f. In this case the corresponding run is $q_i q_j$. Similarly, (ii) if there is a construction in $frst$, then $a_1(q_i) \to q_{acc} \in \Phi^f$, hence the corresponding run is $q_i q_{acc}$. Otherwise (iii) if there is no construction in $frst$ and no structural function is called, then $a_1(q_i) \to q_{deny} \in \Phi^f$. Here, the corresponding run is $q_i q_{deny}$. The reverse direction of the base case can be proven in a similar and straightforward manner. All together this proves our statement for the basic case. Again, the general step of the induction can be proven in the same way, thus we omit the details. ■

Proposition 6.2. is a straightforward consequence of Lemma 4.18. and 9.3. Keeping the notations of the proposition, by Lemma 4.18. $f(I)$ is not empty iff I contains a path pa as a pregaph s.t. $f(pa)$ is also not empty. On the other hand, Lemma 9.3. implies that $f(pa)$ is not empty iff A_f accepts w_{pa}.

A 6.2.1.

Lemma 6.8. For an arbitrary structural recursion f $\tilde{A}_f$ is the complement of A_f.

Proof. Let t be a node-labelled tree and λ a run of A_f on t. We prove the statement by using induction on the length k of the longest path from the root to a leaf of λ. Here, the length of a path is the number of its edges. Recall that in the definition of A_f, $\tilde{A}_f$ each transitional rule $(q_i, a) \to \psi$ has a counterpart $(\tilde{q}_i, a) \to \neg\psi$. Thus, to each run λ of A_f on tree t a complement run $\tilde{\lambda}$ of $\tilde{A}_f$ on t can be assigned in a straightforward manner. Formally,

a one-to-one mapping ν can be given from $V.\lambda$ to $V.\tilde{\lambda}$ s.t. (i) if $lab^{\lambda}(u) = (q_j, w)$ $(lab^{\lambda}(u) = (\tilde{q}_j, w))$, then $lab^{\tilde{\lambda}}(\nu(u)) = (\tilde{q}_j, w)$ $(lab^{\tilde{\lambda}}(\nu(u)) = (q_j, w))$ $(w \in \mathbb{N}^*)$. (ii) If $lab^{\lambda}(u) = true$ $(lab^{\lambda}(u) = false)$, then $lab^{\tilde{\lambda}}(\nu(u)) = false$ $(lab^{\tilde{\lambda}}(\nu(u)) = true)$. (iii) If there is an edge from u to v in λ, there is also an edge from $\nu(u)$ to $\nu(v)$ in $\tilde{\lambda}$. (iv) If there is an edge from $\tilde{u}$ to $\tilde{v}$ in $\tilde{\lambda}$, then there is also an edge from $\nu^{-1}(\tilde{u})$ to $\nu^{-1}(\tilde{v})$ in λ.

Assume now that $k = 1$. Suppose first that A_f accepts tree t $(t \in \mathcal{T}^{\Upsilon_f})$. We have to show that $\tilde{A}_f$ refuses t. Since A_f has only one initial state, $lab^{\lambda}(\epsilon)$ is (q_i, ϵ) (consider page 11 for the definition of a run of an alternating tree automaton). Since λ stops in one step and accepts t, $lab^{\lambda}(1)$ has to be $true$. Remember that 1 is the child of the root of λ here. Consequently, t has to be a single node with label a_{acc}. Clearly, $\tilde{A}_f$ refuses t. On the other hand, if $\tilde{A}_f$ accepts t, then with a similar reasoning it can be shown that t is a single node with label a_{deny}, which is refused by A_f.

Assume now that the statement holds for $k \leq m$ and $k := m+1$. Suppose that A_f accepts t, and transitional rule $(q_i, a) \rightarrow \phi$ has been used to get the children of the root of λ. Recall that ϕ can have three different forms: $Form_{aut}(\gamma_{i,a})$, $(q_i, m_f + 1)$ and $(q_i, m_f + 2)$. If ϕ is $(q_i, m_f + 1)$, then in $\tilde{A}_f$ transitional rule $(\tilde{q}_i, a) \rightarrow (\tilde{q}_i, m_f + 1)$ is used. The $(m_f + 1)^{th}$ child of the root of t cannot be labelled with a_{deny}, because in this case A_f would refuse t. On the other hand, if it is labelled with a_{acc}, then A_f clearly accepts, while $\tilde{A}_f$ refuses t. If the label is different from both a_{acc} and a_{deny}, then the same reasoning can be applied as that we use in the next paragraph. Besides, if $\phi = (q_i, m_f + 2)$ the same train of thoughts can be applied as that of used for the previous case when ϕ was $(q_i, m_f + 1)$.

Thus we may assume that ϕ is $Form_{aut}(\gamma_{i,a})$. This means that in $\tilde{\lambda}$ transitional rule $(\tilde{q}_i, a) \rightarrow \neg Form_{aut}(\gamma_{i,a})$ is used to get the children of the root. Suppose that $\Lambda(j) = true$, i.e., when the truth values are assigned to the nodes of λ, the j^{th} child of the root gets value $true$ (consider page 12). Moreover, suppose that $lab^{\lambda}(j) = (q_r, s)$, which means that state q_r is called on the s^{th} child of the root of t. Clearly, $A_f^r := (Q_f, \Upsilon_f, \{q_r\}, \Psi_f)$ accepts t^s,

where t^s is the same as t, only the aforementioned s^{th} child is taken to be the root. The accepting run can be obtained from λ by taking the j^{th} child in question as the root. In this run the length of the longest path is less than equal to m, hence the induction hypothesis can be applied. It is easy to see that the label of the j^{th} child of $\tilde{\lambda}$ is $(\tilde{q}_r, s)$, and from the induction hypothesis it follows that a $false$ value is assigned to this child. Remember that (q_r, s) stands in a chain of disjunctions $(q_r, 1) \vee \ldots \vee (q_r, m_f)$ in $Form_{aut}(\gamma_{i,a})$, while $(\tilde{q}_r, s)$ is in a conjunction chain $(\tilde{q}_r, 1) \wedge \ldots \wedge (\tilde{q}_r, m_f)$ in $\neg Form_{aut}(\gamma_{i,a})$. The previous reasoning has just showed that, if $\Lambda(j)$ is $true$ ($lab^{\lambda}(j) = (q_r, s)$), then the former disjunction chain becomes $true$ and the conjunction chain becomes $false$.

Suppose that $lab^{\lambda}(j) = (\tilde{q}_r, s)$ (we still assume that $\Lambda(j) = true$). Then in the same way as before it can be shown that the j^{th} child of the root of $\tilde{\lambda}$ is labelled with (q_r, s) and Λ assigns a $false$ value to this node. From this train of thoughts it also follows that if a conjunction chain $(\tilde{q}_r, 1) \wedge \ldots \wedge (\tilde{q}_r, m_f)$ in $Form_{aut}(\gamma_{i,a})$ becomes $true$, then the corresponding disjunction chain $(q_r, 1) \vee \ldots \vee (q_r, m_f)$ in $\neg Form_{aut}(\gamma_{i,a})$ becomes $false$ and vice versa. If in an interpretation variables $X_1, \ldots, X_l$ of propositional formula P become $true$ and this makes P $true$, then the $false$ value of these variables implies that $\neg P$ becomes $false$. From this observation it follows that $Form_{aut}(\gamma_{i,a})$ is satisfied iff $\neg Form_{aut}(\gamma_{i,a})$ becomes $false$. Furthermore, if $Form_{aut}(\gamma_{i,a})$ is satisfied, then in Λ a $true$ value is assigned to the root of λ. On the other hand, if $\neg Form_{aut}(\gamma_{i,a})$ becomes $false$, then a $false$ value is assigned to this root. To sum up in this case A_f accepts t iff $\tilde{A}_f$ refuses it. In the other case, when $\Lambda(j)$ is $false$, the same reasoning can be applied. This concludes the proof. ■

Lemma 6.9. Let $f = (\{f_1, \ldots, f_n\}, \Sigma, \{f_i\}, \Gamma)$ be a structural recursion and t an f simulation tree, then $f(t)$ is not empty $\Leftrightarrow$ A_f accepts $t^{node} = \phi_{edge \to node}(t)$.

Proof. Assume that $t = \{a : t_1 \cup \ldots \cup t_{m_f+2}\}$ ($a \in \Sigma_f \cup \{\S\}$). We prove the lemma by using induction on the number of steps k of the condition

evaluation in $U_f \sqcap t$. Suppose first that $k = 1$. If $f(t)$ is not empty, then in $\gamma_{i,a}$ a ψ edge is constructed without condition. Thus the corresponding transitional rule is of the form $(q_i, a) \to (q_i, m_f+1)$. Clearly, A_f accepts t^{node} in this case, since the label of the $(m_f+1)^{th}$ child is a_{acc} and $(q_i, a_{acc}) \to true$ is in Φ_f. On the other hand, if $f(t)$ is empty, then there is no condition in $\gamma_{i,a}$ and nothing is constructed, hence in A_f transitional rule $(q_i, a) \to (q_i, m_f+2)$ is to be applied, which entails the refusal of t^{node}.

Suppose now that the statement holds for $k \leq m$ and $k := m + 1$. This means that $\gamma_{i,a}$ has a condition and transition rule $(q_i, a) \to Form_{aut}(\gamma_{i,a})$ was used to construct the children of the root in the run of A_f on t^{node}. Denote λ this run. Assume first that $f(t)$ is not empty. This entails that $Form(\gamma_{i,a})$ is satisfied by $t_1 \cup \ldots \cup t_{m_f+2}$. Suppose that n.i.$(f_j(t))$ occurs in $Form(\gamma_{i,a})$ and it is satisfied by t_o $(1 \leq j \leq n, 1 \leq o \leq m_f)$. Recall that $\gamma_{j,a_{acc}}, \gamma_{j,a_{deny}}$ are without any condition and construction, thus $f_j(t_{m_f+1})$ and $f_j(t_{m_f+2})$ are surely empty (which explains that why o should be less than $m_f + 1$). Recall also that n.i.$(f_j(t))$ is substituted with $(q_j, 1) \vee \ldots \vee (q_j, m_f)$ in $Form_{aut}(\gamma_{i,a})$. For $f^j = (\{f_1, \ldots, f_n\}, \Sigma, \{f_j\}, \Gamma)$, $f^j(t_o)$ is obviously not empty. Since t_o is an f simulation tree and in $U_{f^j} \sqcap t_o$ the condition evaluation can be accomplished in at most m steps, the induction hypothesis can be applied, which means that A_{f^j} accepts $\phi_{edge \to node}(t_o)$. Here A_{f^j} is the same as A_f, only the set of initial states is changed to $\{q_j\}$. Consequently, (q_j, o) and hence $(q_j, 1) \vee \ldots \vee (q_j, m_f)$ becomes $true$ in $Form_{aut}(\gamma_{i,a})$.

On the other hand assume that i.$(f_j(t))$ is in $Form(\gamma_{i,a})$ and it becomes $true$. This entails that $f_j(t_o)$ is empty for all o $(1 \leq i \leq m_f)$. From the induction hypothesis it follows that A_{f^j} refuses t_o in this case. However, according to Lemma 6.8. $\tilde{A}_{f^j}$ accepts t_o. Remember that in $Form_{aut}(\gamma_{i,a})$ i.$(f_j(t))$ is substituted with $(\tilde{q}_j, 1) \wedge \ldots \wedge (\tilde{q}_j, m_f)$. Also remember that $\tilde{A}_{f^j}$ is the same as A_{f^j}, only the set of initial states is changed to $\{\tilde{q}_j\}$. Since $\tilde{A}_{f^j}$ accepts t_o, $(\tilde{q}_j, o)$ becomes $true$ in $Form_{aut}(\gamma_{i,a})$. Since o has been chosen arbitrarily and $1 \leq o \leq m_f$, from this observation it follows that the whole conjuction chain $(\tilde{q}_j, 1) \wedge \ldots \wedge (\tilde{q}_j, m_f)$ becomes $true$. All in all, it has been

shown that if $Form(\gamma_{i,a})$ is satisfied by $t_1 \cup \ldots \cup t_{m_f+2}$, then $Form_{aut}(\gamma_{i,a})$ becomes $true$ in λ, which means that $true$ is assigned to the root of λ, hence by definition A_f accepts t.

In the same way it can be proven that if n.i.$(f_j(t))$ (i.$(f_j(t))$) is not satisfied, then $(q_j, 1) \vee \ldots \vee (q_j, m_f)$ $((\tilde{q}_j, 1) \wedge \ldots \wedge (\tilde{q}_j, m_f))$ also becomes $false$. Consequently, if $Form(\gamma_{i,a})$ is not satisfied by $t_1 \cup \ldots \cup t_{m_f+2}$, then $Form_{aut}(\gamma_{i,a})$ becomes $false$ in λ. This proves that the emptiness of $f(t)$ implies the refusal of t^{node} by A_f.

The reverse direction can be proven in a similar way. ■