I4-D15a

Scalable, Space-optimal Implementation of Web Queries Part 1: Data Model, Queries, and Translation

Project title:	Reasoning on the Web with Rules and Semantics
Project acronym:	REWERSE
Project number:	IST-2004-506779
Project instrument:	EU FP6 Network of Excellence (NoE)
Project thematic priority:	Priority 2: Information Society Technologies (IST)
Document type:	D (deliverable)
Nature of document:	R/P (report and prototype)
Dissemination level:	PU (public)
Document number:	IST506779/Munich/I4-D15a/D/PU/a1
Responsible editors:	Tim Furche
Reviewers:	Benedikt Linse and Georg Gottlob
Contributing participants:	Munich
Contributing workpackages:	I4
Contractual date of deliverable:	31 August 2007
Actual submission date:	15 January 2008

Abstract
As we move towards the vision of a next generation Web, be it called Web 2.0 or Semantic Web, we observe that the number of formats used for representing data on the Web increases, so does apparently the languages we are supposed to use to access that data: XPath, XQuery, SPARQL, XSLT, Xcerpt, etc. In this deliverable we propose a formal foundation for most common Web formats and query languages by introducing a formal query language, called CIQLog operating on data graphs and translating Xcerpt, XPath, XQuery, and SPARQL into that language. Not only do we arrive at purely logical semantics for these languages and a better understanding of their differences, we also pave the way to space and time optimal implementations of these languages by means of the CIQCAG algebra introduced in the second part of this deliverable.

Keyword List
algebra,complexity,Xcerpt,XPath,XQuery,SPARQL,queries,data model,semantics

Project co-funded by the European Commission and the Swiss Federal Office for Education and Science within the Sixth Framework Programme.

Scalable, Space-optimal Implementation of Web Queries
Part 1: Data Model, Queries, and Translation

Tim Furche[1], Antonius Weinzierl[1], François Bry[1]

[1] Institute for Informatics, University of Munich, Germany
`http://pms.ifi.lmu.de`

15 January 2008

Abstract
As we move towards the vision of a next generation Web, be it called Web 2.0 or Semantic Web, we observe that the number of formats used for representing data on the Web increases, so does apparently the languages we are supposed to use to access that data: XPath, XQuery, SPARQL, XSLT, Xcerpt, etc. In this deliverable we propose a formal foundation for most common Web formats and query languages by introducing a formal query language, called CIQLog operating on data graphs and translating Xcerpt, XPath, XQuery, and SPARQL into that language. Not only do we arrive at purely logical semantics for these languages and a better understanding of their differences, we also pave the way to space and time optimal implementations of these languages by means of the CIQCAG algebra introduced in the second part of this deliverable.

Keyword List
algebra,complexity,Xcerpt,XPath,XQuery,SPARQL,queries,data model,semantics

Contents

Overview of this Deliverable

In this deliverable, we introduce as a formal foundation for Web queries against any data format, CIQLog, a rule-based query language tailored to semi-structured queries. CIQLog is a slightly modified variant of $\text{datalog}^{\neg}_{new}$, i.e., *datalog extended with negation and value invention*, which is most prominently represented by ILOG [42]. CIQLog is used as a tool to map most major Web query languages, viz. XPath, XQuery, Xcerpt, and SPARQL, into the same formal framework in Chapters 3 to 5.

The common data model and query language CIQLog and the translations from these languages into CIQLog yield

(1) a *purely logical semantics* for XPath, XQuery, Xcerpt, and SPARQL. For XQuery and SPARQL, this is the first purely logical semantics to the best of our knowledge.

(2) a better *understanding of commonalities and differences* between these languages. In particular, the step from XPath to (composition-free) XQuery illustrates how only a small number of additional features dramatically affects the semantics and evaluation of XQuery.

(3) together with the CIQCAG algebra (and the translation from CIQLog to CIQCAG in [34]), we obtain

(a) a space optimal *implementation of navigational XPath* with space complexity $\mathcal{O}(q \cdot d)$ and time complexity $\mathcal{O}(q \cdot n)$ where q is the query, n the data size, and d the depth of the tree data.

(b) the *first implementation for SPARQL* with *polynomial-time complexity for tree queries* on arbitrary graphs and linear complexity on tree and certain graph data (see below).

(c) efficient implementations for Xcerpt and XQuery that scale over different data and query shapes, i.e., that provide on each restricted class time and space complexity rivaling the best known approaches limited to that class.

(4) a foundation for the *integration of queries* from several of these languages (with additional CIQLog interface rules to properly expose data from queries in one language to queries of another language).

Chapter 1

Data Model—Relations over Trees and Graphs

1.1 Introduction

As abstraction of the query cores for most Web query languages we introduce in Chapter 2 CIQLog which is used in Chapters 3 to 5 as target for the translation from XQuery, Xcerpt, and SPARQL and compiled into CIQCAG algebra expressions in [34]. Before we can introduce CIQLog, however, we need to establish the employed data model in this chapter.

We choose binary relational structures as formal data model for CIQLog and CIQCAG. However, to bridge the gap to XML and RDF query languages such as XQuery or SPARQL, we first introduce a common view of Web data as node and edge labeled graphs and mappings from XML and RDF into such graphs. These mappings are, for the most part, simple and intuitive. On these data graphs, we define a set of relations for querying the structure of the graph (Section 1.6.4), the relative position of edges and labels in that graph (Section 1.6.5, the labels of edges and nodes (Section 1.6.3), and edge and nodes equivalent wrt. label, position, or structure (Section 1.6.6. These relations are closely related to XPath's axes and other formalizations of relations on XML data, but exhibit a number of distinct features to properly address arbitrary shapes of the underlying data graphs and the effect of edge labels (see, e.g., the definition of child- and descendant-like traversal relations in Section 1.6.4).

1.2 Data Graphs

From the perspective of their data model, many Web representation formats such as XML, RDF, and Topic Maps have a lot of commonalities: the data is semi-structured, tree- or graph-shaped, and sometimes ordered, sometimes not (XML elements vs. XML attributes, RDF sequence containers vs. bag containers). We choose (finite unranked) **labeled ordered directed graphs** as common data model for Web data:

Definition 1.1 (Data graph). A query is evaluated against a data graph D over finite label alphabets Σ_E

for edges and Σ_N for nodes. D is a 6-tuple

$$(N, E, R, \mathfrak{L}, \mathfrak{O}),$$

where N is the set of nodes of the graph, $E \subset N \times \mathbb{N} \to N$ the set of edges, $R \subset N$ the set of root vertices, $\mathfrak{L} : (N \to \Sigma_N) \cup (E \to \Sigma_E)$ the labeling function on nodes and edges, and the order specification $\mathfrak{O} \subset N \times \Sigma_E$. For simplicity, we assume $N \cap E = \varnothing$ and that $\mathfrak{L}$ is total on edges, but may be partial on nodes. Note, that an edge (n, i, m) maps a pair of a (source) node n and an edge position i to a (sink) node m, thus there are no two distinct edges with the same source and order number. As usual, $\deg n = |\{(n, i, n') \in E\}|$ denotes the degree of a node, i.e., the number of outgoing edges.

D is an *ordered* graph, i.e., the order of the children of a node is significant. Since the order is relative to the parent and a child may occur under several parents (in fact, it may occur several times under the same parent), the order is associated with the edge rather than with the child node. The order specification $\mathfrak{O}_N$ allows both ordered and unordered data (e.g., unordered XML attributes and ordered XML sub-elements, RDF bag and sequence containers) in the same graph: the order among the λ-labeled outgoing edges of a node n is significant only if $(n, \lambda) \in \mathfrak{O}_N$. We choose to record the position of a λ-labeled edge even if $(n, \lambda) \notin \mathfrak{O}_N$. This allows for bag-like data with duplicates represented by multiple edges between same nodes and a consistent signature of edges.

D's nodes may be labeled by virtue of the single (partial) node labeling function $\mathfrak{L}_N$. For sake of conciseness, we choose a single labeling function. In practice, there might be cases where different labeling functions are advisable, e.g., one for element labels and one for string values in XML data, or one for resource URIs and one for literals in RDF data.

Edge labels from Σ_E are used to distinguish different relations among nodes in N. For an edge $e = (p, i, c)$ with $\mathfrak{L}(e) = \lambda$, we call c a λ-child or simply a child of p. In the following, we use the edge labels CHILD and VALUE to model (element) containment and string value in an XML document. The former represent the XML element hierarchy, the latter associate text nodes with element nodes. In case of RDF data, the respective property URI is used as edge label.

The definition allows *multiple root nodes*, e.g., if there are several connected components in the graph. Any node may be a root node, in particular root nodes may, in contrast to usual rooted graph models, have parents. Intuitively, root nodes are simply highlighted "entrance" points into the graph that can be chosen arbitrarily when defining the data graph: If the data graph is a single XML document there will be a single such root node, however this formalization also covers collections of XML documents (as in XQuery) and RDF graphs where, e.g., each subject node can be considered a root node. In the following, we assume that *each node in a data graph is part of a rooted connected component*, i.e., is either a root node itself or reachable from a root node.

Following Codd's surrogate extension [22] for the relational data model, we choose *surrogate or object identity for nodes and edges*, i.e., nodes and edges have identity separate from their "value" or structure. This contrasts with the basic relational data model that uses extensional identity (i.e., the value of a data item defines its identity, and thus two data items with same value necessarily have the same identity). Surrogate identity allows an intuitive and clean semantics for querying cyclic data instances, whereas cyclic data instances under extensional identity lead to infinite regular trees (cf. [24]) which have questionable properties for certain classes of queries, most prominently occurrence queries. Furthermore, [2] shows that graphs with object identity can, up to identity, be seen as finite representations of infinite regular

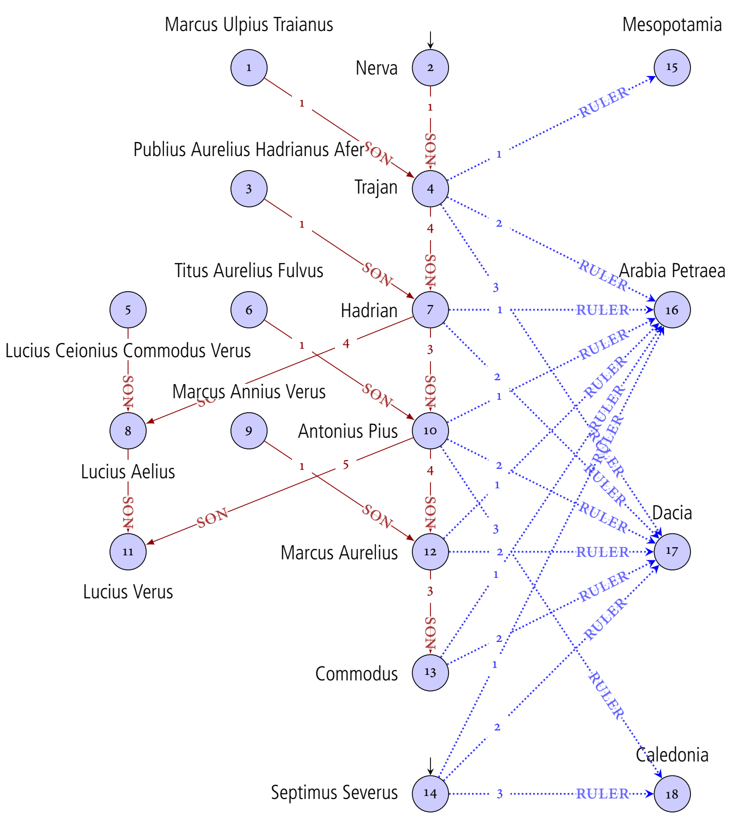


Figure 1. Exemplary Data Graph

trees. In this respect, data graphs are similar to object-oriented data models. XQuery's data model [31] also uses node identities separate from node values, but limits the data to trees.

Figure 1 shows a data graph depicting roman emperors of the Nervan-Antonine dynasty and some of their ruled provinces. Labels are depicted in sans-serif close to the labeled node, edges are decorated with their position index (e.g., $-\,{\scriptstyle 2}\,-$) and label (e.g., $\xrightarrow{\text{SON}}$), a root node is indicated by a sink-only edge (like $\to ②$). The data graph contains two kinds of edges, SON edges (colored in dark red), and RULER edges (colored in blue). For referencing, nodes are numbered and we refer, e.g., to the node with label Nerva as d_2 (for second data graph node).

Formally, Figure 1 depicts the data graph $D = (N, E, R, \mathfrak{L}, \mathfrak{D})$ with

$$
\begin{aligned}
N &= \{d_1, \ldots, d_{18}\} \qquad\qquad R = \{d_2, d_{14}\}\\
E &= \{(d_1,1,d_4),(d_2,1,d_4),(d_3,1,d_7),(d_4,1,d_{15}),(d_4,2,d_{16}),(d_4,3,d_{17}),\\
&\quad (d_4,4,d_7),(d_5,1,d_8),(d_6,1,d_{10}),(d_7,1,d_{16}),(d_7,2,d_{17}),(d_7,3,d_{10}),\\
&\quad (d_7,4,d_8),\ldots\}\\
\mathfrak{L} &= \{d_1 \to \textsf{M. Ulpius Trai.}, d_2 \to \textsf{Nerva}, d_3 \to \textsf{P. Aur. Hadr. Afer}, d_4 \to \textsf{Trajan}, \ldots,\\
&\quad (d_1,1,d_4) \to \text{SON}, (d_2,1,d_4) \to \text{SON}, (d_3,1,d_7) \to \text{SON},\\
&\quad (d_4,1,d_{15}) \to \text{RULER}, (d_4,2,d_{16}) \to \text{RULER}, (d_4,3,d_{17}) \to \text{RULER},\\
&\quad (d_4,4,d_7) \to \text{SON}, \ldots\}\\
\mathfrak{D} &= \varnothing
\end{aligned}
$$

In the following sections, we briefly outline, how XML documents, RDF graphs, and Xcerpt data terms can all be faithfully mapped to data graphs.

1.3 XML: Essentials and Formal Representation

XML [13] is, by now, certainly *the* foremost data representation format for the Web and for semi-structured data in general. It has been adopted in a stupendous number of application domains, ranging from document markup (XHTML, Docbook [72]) over video annotation (MPEG 7 [53]) and music libraries (iTunes[1]) to preference files (Apple's property lists [5]), build scripts (Apache Ant[2]), and XSLT [44] stylesheets. XML is also frequently adopted for serialization of (semantically) richer data representation formats such as RDF or TopicMaps.

The following presentation of and mapping for XML documents is oriented along the XML Infoset [25] which describes the information content of an XML document. The XQuery data model [31] is, for the most parts, closely aligned with this view of XML documents.

Following the XML Infoset, we provide a *graph shaped* view of XML data containing valid ID/IDREF links. This contrasts with the XQuery data model, where such links are not resolved. In the following, ID/IDREF links are distinguished from the parent/child links expressed by the element hierarchy in accordance to the XML Infoset specification. For many applications this separation is unnecessary and even harmful which motivates us to briefly discuss an alternative mapping of ID/IDREF links to data graphs in Section 1.3.3.

[1] `http://www.apple.com/itunes/`
[2] `http://ant.apache.org/`

1.3.1 XML in 500 Words

The core provision of XML is a syntax for representing hierarchical data. Data items are called elements in XML and enclosed in start and end *tags*, both carrying the same tag names or *labels*. `<author>...</author>` is an example of such an element. For ..., we can write other elements or character data as *children* of that element. The following listing shows a small XML fragment that illustrates elements and element nesting:

```
   <conference xmlns:dc="http://purl.org/dc/elements/1.1/"
 2             dc:title="Storage Media">
    <dc:date>44 B.C.</dc:date>
 4  <paper title="Wax Tablets" id="p1" cites="p2">
      <author>Cicero<!-- incomplete! --></author>
 6  </paper>
    <paper id="p2" cites="p1">
 8    <author>Hirtius</author>
    </paper>
10  <pc><member>Cicero</member>
      <member>Atticus</member></pc>
12 </conference>
```

In addition, we can observe *attributes* (name, value pairs associated with start tags) that are essentially like elements but may only contain character data, no other nested attributes or elements. Also, by definition, *element order* is significant, attribute order is not. For instance

```
<pc><member>Atticus</member><member>Cicero</member></pc>
```

represents different information than the pc element in lines 10–11, but

```
<paper cites="p1"id="p2"><author>Hirtius</author></paper>
```

represents the same element information item (inter-element white space is ignored) as lines 7–9.

Elements, attributes, and character data are XML's most common information types. In addition, XML documents may also contain *comments* (line 5), *processing instructions* (name-value pair with specific semantics that can be placed anywhere an element can be placed), *document level information* (such as the XML or the document type declarations), *entities*, and *notations*. The mapping easily provides for these information types and their specifics, but details are omitted here for sake of conciseness.

On top of these information types, two additional facilities relevant to the mapping from XML to data graphs are introduced in XML by subsequent specifications: Namespaces [12] and Base URIs [52]. Namespaces allow the partitioning of element labels used in a document into different namespaces, identified by a URI. Thus, an element is no longer labeled with a single label but with a triple consisting of the *local name*, the *namespace prefix*, and the *namespace URI*. E.g., for the `dc:date` element in line 3, the local name is `date`, the namespace prefix is `dc`, and the namespace URI (called "name" in [25]) is `http://purl.org/dc/elements/1.1/`. The latter can be derived by looking for a *namespace declaration* for the prefix dc. Such a declaration is shown in line 1: `xmlns:dc="http://...` It associates the prefix `dc` with the given URI in the scope of the current element, i.e., for that element and all elements contained within unless there is another nested declaration for `dc`, in which case that declaration takes precedence. Thus, we can associate with each element a set of *in-scope namespaces*, i.e., of pairs namespace prefix and

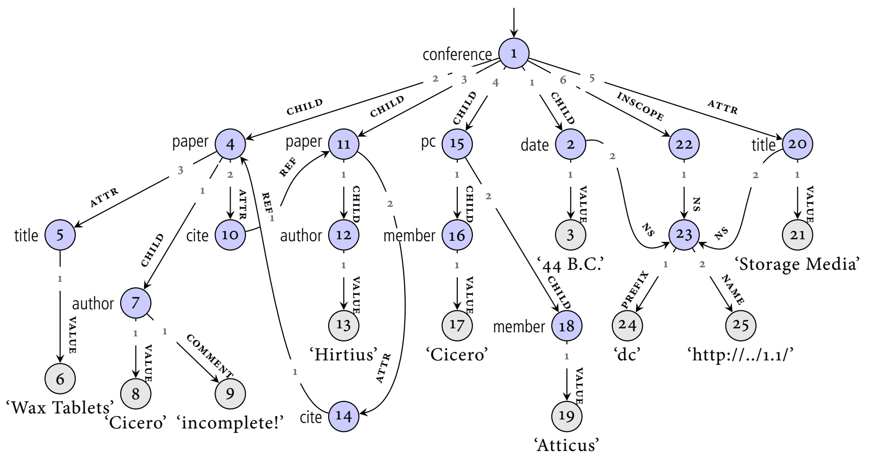


Figure 2. Exemplary Data Graph: XML Conference Data

URI, that are valid in the scope of that element. Base URIs [52] are used to resolve relative URIs in an XML document. They are associated with elements using `xml:base="http://...` and, as namespaces, are inherited to contained elements unless a nested `xml:base` declaration takes precedence.

In the next section, we describe how we map the basic information items as well as derived and heritable information such as namespaces and XML Base URIs to data graphs. It is worth highlighting that the mapping can be easily extended to cover additional information items (such as entities, notations, and processing instructions or information items typed by XML Schema [30] types) as well as other forms of heritable information.

1.3.2 Mapping XML to Data Graphs

In Figure 2, a data graph for the above XML document is shown: As before, edges are decorated with their position index (e.g., $- {\scriptstyle 2} -$) and label (e.g., $\overset{\text{CHILD}}{\longrightarrow}$), a root node is indicated by a sink-only edge (like $\rightarrow\!①$). Nodes representing literal character content are differentiated using gray (like ④) instead of blue color, their labels enclosed in single quotes. The order specification is not presented in the figure. With the order specification $\mathfrak{O} = N \times \{\text{CHILD}, \text{VALUE}, \text{COMMENT}, \text{REF}\}$, i.e., order is significant for all CHILD, VALUE, COMMENT, and REF edges, but not for ATTR, INSCOPE, NS, PREFIX, and NAME edges, the graph faithfully represents the XML information set corresponding with the above fragment. As in [25], the children of an element are ordered and include element, comment, and character data children. Also, for any attribute with type idref or idrefs, the attribute information item contains an ordered list of element information items referenced by that attribute, here expressed using REF edges. This, again, mirrors [25] but deviates from the XQuery data model [31] and many other (purely tree-shaped) views of XML.

In Figure 2, we place ATTR, NS, and INSCOPE edges after CHILD, COMMENT, and VALUE edges, however, this is an arbitrary choice. As long as the order between edges of the latter kind is preserved, the order of the position of the remaining edges is insignificant.

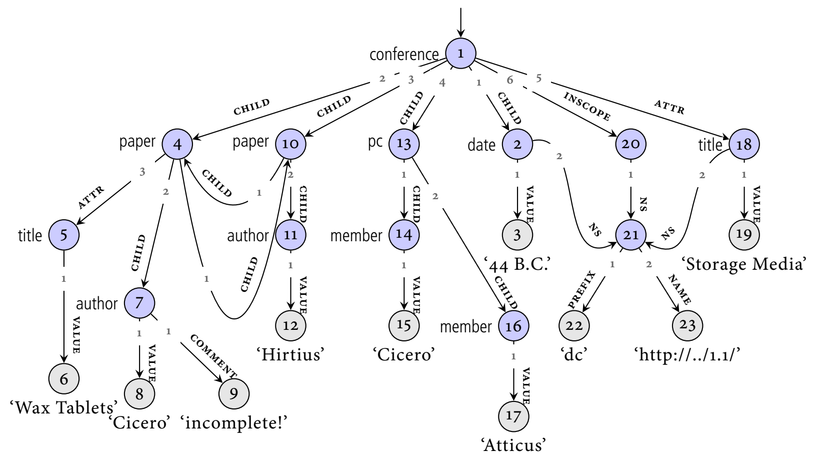


Figure 3. Exemplary Data Graph: XML Conference Data with Transparent id/idref-links

1.3.3 Transparent Links

In the mapping scheme illustrated by Figure 2, we choose to represent id/idref-links as in [25]. However, in many applications, a treatment of such links in the same way as parent/child-relations is preferable. In this case, we place the linked elements, in order, at the beginning of the child list of the element containing the attribute.

If an element contains no other information items (no attributes, elements, comments, etc.), we may also choose to replace the element by the reference. This allows arbitrary placement of id/idref-referenced elements within the child list of a parent element. Furthermore, it allows the transparent resolution of links expressed using reference elements such as XHTML’s a or Docbook’s Link.

1.4 RDF: Essentials and Formal Representation

In addition to RDF, we also demonstrate how to map RDF [51, 45, 39] graphs to datagraphs. RDF is, though much less common than XML, a widespread choice for interchanging (meta-) data together with descriptions of the schema of that data.

Following the recent SPARQL [65] proposal, We choose to support the mapping of RDF graphs under simple entailment as defined in [39]. In contrast to SPARQL, we omit typed literals and named graphs [18], both optional features of RDF (or extensions thereof) for simplicity. Both can be easily added to the described mapping, e.g., each named graph can be represented as a separate connected component, the graph representatives distinguished as root nodes of the data graph.

1.4.1 RDF in 500 Words

RDF graphs contain simple statements about *resources* (or entities, objects, etc.). Statements are triples consisting of subject, predicate, and object, all of which are resources. If we want to refer to a specific resource, we use (supposedly globally unique) URIs, if we want to refer to a resource for which we know that it exists and maybe some of its properties, we use *blank nodes* which play the role of existential quantifiers in logic. However, blank nodes may not occur in predicate position. Finally, for convenience, we can directly use *literal values* as objects.

RDF may be serialized in many formats, such as RDF/XML [7], an XML dialect for representing RDF, or Turtle [6] which is also used in SPARQL. The following Turtle data represents roughly the same data as the XML document discussed in the previous section:

```
   @prefix dc: <http://purl.org/dc/elements/1.1/> .
 2 @prefix dct: <http://purl.org/dc/terms/> .
   @prefix vcard: <http://www.w3.org/2001/vcard-rdf/3.0#> .
 4 @prefix bib: <http://www.edutella.org/bibtex#> .
   @prefix ulp: <http://example.org/roman/libraries/ulpia#> .
 6 ulp:cicero-46-wt a bib:Article ; dc:title "Wax Tablets" ;
        dc:creator [ a rdf:Seq ;
 8          rdf:_1 ulp:cicero ; rdf:_2 ulp:tiro ] ;
        ulp:cites ulp:hirtius-47-bc ;
10      dct:isPartOf ulp:conf-46-mutina .
   ulp:cicero a bib:Person ; vcard:FN "M. T. Cicero" .
12 ulp:tiro a bib:Person ; vcard:FN "M. T. Tiro" .
   ulp:hirtius-47-bc a bib:Article ;
14      ulp:cites ulp:cicero-46-wt ;
        dct:isPartOf ulp:conf-46-mutina .
16 ulp:conf-46-mutina a bib:InProceedings ;
        rdfs:label "Storage Media" .
```

Following the definition of namespace prefixes used in the remainder of the Turtle document (omitting common RDF namespaces), each line contains one or more statements separated by colon or semicolon. If separated by semi-colon, the subject of the previous statement is carried over. E.g., line 6 reads as ulp:cicero-46-wt is a bib:Article and has dc:title "Wax Tablets". Lines 7–9 show a blank node: the creator of the article is neither Cicero nor Tiro, but some unnamed resource that is a sequence of those two authors.

RDF Interpretations are used to provide meaning to the elements of an RDF graph. URIs in subject or object position are interpreted as arbitrary objects, such as people, trains or web pages. An URI in predicate position is interpreted as a set of pairs of objects such as train connections, coauthor relationships or links between webpages. The set of entities that RDF graphs make statements about is called the *domain* of the RDF graph.

Finally blank nodes are used to express existential knowledge or to group information in RDF graphs. Each blank node is interpreted as a domain element but its interpretation is not fixed: An interpretation is a *model* of an RDF graph iff there is an interpretation for the blank nodes such that for every triple, the interpretation of the subject and object is an element of the interpretation of the predicate. An RDF graph g is said to *entail* an RDF graph h if every model of g is also a model of h.

As the definition of an interpretation resembles the definition used in logic, it is possible to view an RDF graph as a formula. This formula has an atom for every triple. URIs and literals are represented by constants, while blank nodes are represented by existential variables.

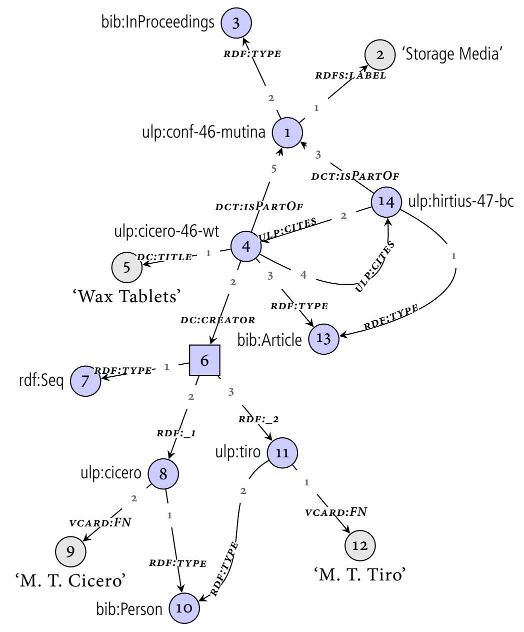


Figure 4. Exemplary Data Graph: RDF Conference Data

1.4.2 Mapping RDF to Data Graphs

This RDF data is mapped to a data graph as shown in Figure 4. In the case of RDF the order of the edges is mostly irrelevant, only for the sequence container ($\boxed{6}$ in Figure 4) we choose to order the RDF:_*i* edges. This allows for a more convenient querying of elements in a sequence, cf. [43]. We choose to consider all named resources, i.e., all resources with URI label (depicted as light blue round nodes ○ in Figure 4), as root nodes of the RDF graph. This choice, however, does not affect the remainder of this article. Recall, that root nodes are nothing more than specifically marked entrance points into the graph. Otherwise they are unrestricted, in particular they may have incoming edges. There is a single node in the graph for each named resource that occurs in the RDF data. Same literals may occur multiple times. Each blank node is depicted as a rectangular node (e.g., $\boxed{6}$). As in Turtle [6] and SPARQL, blank nodes are labeled with local identifiers prefixed by `_:`. There is one node for each blank node in the RDF data.

This mapping faithfully represents the RDF graph (under simple entailment), but does not guarantee that the representation is *minimal* in the sense that there is no other data graph that represents an equivalent RDF graph. For an RDF graph containing blank nodes, there may be a more compact representation that eliminates some of these blank nodes that express redundant information. E.g., if we add a statement "there are two articles that cite each other" using two blank nodes for the involved article, this information is redundant as it already follows from the original data (ulp:cicero-46-wt and ulp:hirtius-47-bc cite each other). The mapping from RDF graphs to data graphs maintains such redundancy. However, the representation is minimal if the input RDF graph is minimal (or *lean* in the sense of [39]). For each

RDF graph, an equivalent lean graph exists and its computation is DP-complete[3].

Representing the actual input graph (and not an equivalent minimal one) is necessary to support queries such as selecting the blank nodes in an RDF graph (as per SPARQL's **`isBlank`** operator). Only if we *disallow queries that can distinguish blank nodes* from named resources, the lean graph returns the same answers for all queries as the original graph.

RDF provides a number of high-level modeling concepts such as collections (list), containers (bag, sequences, and alternatives), reification (the representation of a statement as a resource). There is no need to support such concepts in the mapping to data graphs, however, as all of them are reduced to certain conjunctions of basic RDF triples. E.g., lines 7–8 in the above data show the pattern for sequence containers: A resource (often unnamed, i.e., represented by a blank node) is typed as an rdf:Seq and the elements of the sequence are connected using rdf:_*i* predicates. For details on the triple patterns for other high-level modeling concepts, see [45].

1.5 Xcerpt Data Terms

We conclude our consideration of data graphs and their relation to data representation formats for the Web with a brief look at Xcerpt data terms. The formal description of the mapping is presented as part of Chapter 3. Here, we illustrate the basic ideas and how they relate to the mappings for XML and RDF data.

1.5.1 Xcerpt Data Terms in 500 Words

Recall, the general shape of Xcerpt data terms: they are hierarchical (i.e., tree shaped) representations of *graph-shaped*, semi-structured data. To obtain a hierarchical representation of a graph, *referable term identifiers* and *references* are introduced that allow to express non-hierarchical relations. Term identifiers are like ID attributes in XML (or blank node identifiers in many RDF serializations) identifiers for data items that are unique in the context of a data collection (usually a document). References are similar to IDREF attributes in XML but occur in place of elements (rather than as attributes of a special type) and are *transparently* resolved, i.e., the case of a term containing a reference to another term can not be distinguished from the case where the term contains the other term as a direct child.

The following Xcerpt data term yields the data graph from Figure 3:

```
declare ns-prefix dc = "http://purl.org/dc/elements/1.1/"
conference(dc:title="Storage Media") [
  dc:date [ "44 B.C." ]
  p1 @ paper(title="Wax Tablets") [
    ^p2
    author [ "Cicero"
             xcerpt:comment{ "incomplete!"} ]
  ]
  p2 @ paper [
    ^p1
    author [ "Hirtius" ]
```

[3] Recall, that DP is the class of all decision problems, that can be expressed as the intersection of an NP- and a co-NP-problem.

```
  ]
  pc[ member[ "Cicero" ]
      member[ "Atticus" ] ]
]
```

Note, that namespace declarations enclose the element (or element list) that are in the scope of that declaration, i.e., that may use the defined prefixes. Attributes are associated with term labels using parentheses, whereas the children of a term are contained in brackets or braces. Brackets indicate that the order of the children is significant, braces that it is not. The above fragment only uses brackets to yield the same data graph as an XML document where children are always ordered. However, we could just as well use braces in the above data, if, e.g., the order of papers or the order of members in a program committee is not significant.

1.6 Relations on Data Graphs

As formal basis for the query language CIQLog and the algebra CIQCAG discussed in the following chapters, we adopt (binary) relational structures. In fact, both can be used to query arbitrary binary relational structures. For the formalization and evaluation of Web query languages such as Xcerpt, XQuery, and SPARQL, however, we choose a specific relational schema (defining the arity and names of available relations) and how to obtain instances for this schema (i.e., the actual relations) from data graphs as described above. This bridges between the notion of data graphs that is close to the intuitive shape of semi-structured data on the Web as queried by Xcerpt, XQuery, or SPARQL, and the formal notation of relational structures that is more convenient for the purpose of defining the semantics and evaluation of CIQLog and CIQCAG.

The relations defined in the following are familiar from query languages such as XPath and their formal treatments in, e.g., [57, 61, 37] with three major exceptions: **(1)** As we discuss *graphs* with *labeled* edges, we provide relations for accessing both nodes and edges as well as their connections where most other collections of relations on semi-structured trees or graphs either consider only nodes or only edges as the domain of the relations. **(2)** Since edges are labeled, we extend classical structural relations such as child and descendant in XPath to specify a set of edge labels to be traversed. This allows, e.g., to limit a descendant traversal to only nodes reachable by all child edges in Figure 2, excluding node reachable by a mix of child and ref edges (the latter indicating the traversal of an ID/IDREF link). The same applies to horizontal or positional relations such as following or following-sibling. **(3)** Following many object-oriented query languages as well as Xcerpt and XQuery, we introduce a deep equality on nodes such that two nodes are deep equal if the structures rooted at those nodes are, to some extent, compatible. "Compatible" may be an isomorphism between the substructures or just simulation, for details see Section 1.6.6.

Note, that the domain of the attributes of these relations are either the nodes or the edges of the data graph, or the set of integers. The *active* domains are the same for nodes and edges, for integers, however, the *active* domain is a finite subset of all integers with a size bound by the maximum out-degree d of a node in the data graph (and thus by the number of edges). In all cases, but pos, this set is $\{1, \ldots, d\}$. In the case of pos, the size of the set is also bound by d, but the contained integers are arbitrary.

1.6.1 Binary Relational Structures

First, we briefly recall a (standard) definition, following [1], for relational structures but limited to binary relations:

Both CIQLog and CIQCAG operate on a (slightly extended)[4] *relational structure* D as data. D is defined over a relational schema $\Sigma = (R_1[U_1], \dots, R_k[U_k])$ and a *finite* domain N of nodes (or objects or elements or records) in the data graph. Each $R_i[U_i]$ is a relation schema consisting in a relation name and a nonempty set of attribute names, We assume an equality relation $=$ on the nodes that relates each node to itself only (*identity*). D is a tuple $(R_1^D, \dots, R_k^D, O)$. Each R_i^D is a finite, unary or binary relation over N with name R_i. For a relation R, $\mathsf{ar}(R)$ denotes its arity. We extend D with an order mapping O that associates with each (binary) R_i a total order on N such that all $n \in \operatorname{rng} R_i$ are before all $n' \in N \setminus \operatorname{rng} R_i$. We denote with $O(D) = \{o : \exists R_i \in D : O(R_i) = o\}$ the set of orders to which the relations in D are mapped. These orders serve to represent the image of each node in a relation as one or more continuous intervals over the order associated with that relation. Choosing an appropriate order for a relation is discussed in [34].

1.6.2 A Relational Schema for Data Graphs

CIQLog and CIQCAG operate on arbitrary relational structures, though they profit from relational structures where some or all relations are tree, forest, or CIG shaped. In the following, we outline a particular relational schema containing relations on data graphs (as introduced in Section 1.2) that is used to realize Xcerpt, XQuery, and SPARQL queries in CIQLog (and thus CIQCAG).

Given a data graph $D = (N, E, R, \mathfrak{L}, \mathfrak{O})$ over node labels Σ_N and edge labels Σ_E, we choose as domain $N' = N \cup E \cup \{1, \dots, \max(\{i : \exists n, n' : (n, i, n') \in E\}) \cup \Sigma_N \cup \Sigma_E$, i.e., the union of the disjoint sets of nodes, of edges, of integers from 1 to the maximum edge position, and of node and edge labels. Note, that each of the sets and thus N' is finite. We add node and edge labels as well as (edge position) integers to the set of nodes and edges to keep the relational schema for querying data graphs *independent* of the actual graph.[5]

Table 1 gives a summary of the relations defined in the following together with their relation schema. It is worth emphasizing that not all of these relations are used in the translation of each language. Rather, for the most part we limit ourselves to specific label sets S (mostly, the singleton sets) and only consider a small number of path relations. Furthermore, not all relations must be extensional. In fact, in Chapter 2, we briefly discuss a set of minimal set of extensional relations and show how to specify the remaining relations as intensional rules on top of this minimal set in CIQLog.

In the following, let $\Sigma = \Sigma_N \cup \Sigma_E$ be the set of node and edge labels and $\mathcal{P}(\Sigma)$ be the power set over Σ. Furthermore on a domain D, we denote with $R_1 \circ R_2 = \{(n, m) \in D^2 : \exists n' \in D : (n', m) \in R_1 \wedge (n, n') \in R_2\}$ the composition of two binary relations R_1 and R_2 and generalise this definition to $R^k = \underbrace{R \circ R \circ \dots \circ R}_{k \text{ times}}$. For a binary relations R let $R(n) = \{m \in N \cup E : (n, m) \in R\}$ be the "images" of n under R.

[4]The deviation lies in the addition of order for each relation. Furthermore, we restrict ourselves to binary relations.

[5]Alternatively, we can use one (unary) position relation for each edge position and one (unary) label test relation for each edge or node label in the data. Since both are finite sets, the resulting relations still form a relational schema.

	node	edge	node-node	node-edge	edge-edge
identity			$\doteq$	$\doteq$	$\doteq$
structure			$\mathsf{path}^S_{i,j}$, $\overset{\circ}{=}$	$\circ\!\!\rightarrow$, $\rightarrow\!\!\circ$	
position	$@^S$	$@^S$, pos	$\ll^S, \ll^S_+, \ll^S_*, \blacktriangleleft^S, \blacktriangleleft^S_+, \blacktriangleleft^S_*$	$\prec^S, \prec^S_+, \prec^S_* {=}^S_@$	
label	$\mathfrak{L}$, Lab^S	$\mathfrak{L}$, Lab^S	$\cong$	$\cong$	$\cong$
arity	indeg^S, outdeg^S				
ordered	$\mathfrak{O}$, $\mathfrak{O}^S$				
root	root				

Table 1. Summary of query relations (S is a set of labels from $\Sigma_N \cup \Sigma_E$)

1.6.3 Properties of Nodes and Edges: Labels and Positions

Property relations test a certain "local" property of a node or edge without relating it to other nodes and edges. Instead, they associate an edge or node, e.g., with its label or its position among siblings or, resp., edges with the same source.

Label relation. To obtain the label of a node or edge, we make the labeling function of a data graph accessible as a unary *label relation* that identifies all edges and nodes with their label:

$$\mathfrak{L} = \{(t,\sigma) \in (N \cup E) \times \Sigma : \mathfrak{L}(t) = \sigma\}$$

For any finite set of labels $S \subset \mathcal{P}(\Sigma)$, we provide as convenience the *label test relation*

$$\mathsf{Lab}^S = \{t \in N \cup E : \mathfrak{L}(t) \in S\} = \bigcup_{\sigma \in S} \{t \in N \cup E : \mathsf{Lab}(t,\sigma)\}$$

For brevity, we write for unary sets $S = \{\lambda\}$ just Lab^λ and omit the index for $S = \Sigma$. Note, that $\mathsf{Lab} = \mathsf{Lab}^\Sigma$ is not necessarily $N \cup E$ since there may be nodes (though no edges) without label, i.e., $\mathsf{Lab}^{\Sigma_E} = E$, but Lab^{Σ_N} is not necessarily N. For the translations discussed in following chapters, we only use label relations with singleton label sets S. In this case, the number of label relations is bound by the size of $\Sigma_E \cup \Sigma_N$.

Position relation. To test for the position of an edge among the edges with the same source or of a node among its siblings, the *position relation* $@^S$ associates an edge e with 1+ the number of edges with the same source that (1) are ordered (i.e., have a label in $\mathfrak{O}(n)$ where n is the common source), (2) have a label in $S \in \mathcal{P}(\Sigma)$, and (3) have an edge position preceding the edge position of e. We increment the number of edges by 1 to achieve node positions between 1 and $\deg n$ where n is the source of an edge (rather than 0 and $\deg n - 1$). For nodes, $@^S$ associates a node n with whatever any edge with n as sink is associated with under $@^S$:

$$\begin{aligned}@^S = \{(e, i+1) \in E \times \mathbb{N} : e = (n, k, m) \wedge \mathfrak{L}(e) \in \mathfrak{O}(n) \\ \wedge |\{e' = (n, l, m') \in E : \mathfrak{L}(e') \in S \cap \mathfrak{O}(n) \wedge l < k\}| = i\} \\ \cup \{(n, i) \in N \times \mathbb{N} : \exists e \in E : e = (n', k, n) \wedge @^S(e, i)\}\end{aligned}$$

Notice, that an edge in $@^S$ is not itself required to be labeled with a label in S. Rather, this can be achieved by an additional label relation on the edge. Note, that $@^S$ is a function (for each S) on edges, but for a node there may, in general, be several positions associated with it: This is the case for any node with multiple incoming edges. On tree data, $@^S$ is a function also on nodes. Again we use only λ for unary sets $S = \{\lambda\}$ and omit the index S for $S = \Sigma$. For the translations discussed in the following chapters, we only use position relations with singleton label sets S. In this case, the number of position relations is bound by the size of $\Sigma_E \cup \Sigma_N$.

For nodes, we also provide a relation to retrieve the actual edge position:

$$\mathsf{pos} = \{(e, i) \in E \times \mathbb{N} : \exists n, n' \in N : e = (n, i, n')\}$$

Degree relations. To test for the number of in- or out-edges of a node, there are, for each $S \in \mathcal{P}(\Sigma)$, *in- and out-degree relations* indeg^S and outdeg^S that associate each node with its in- resp. out-degree, counting only edges labeled from S:

$$\begin{aligned}\mathsf{indeg}^S &= \{(c, i) \in N \times \mathbb{N}\} : \left|\{e = (p, j, c) \in E : \mathfrak{L}(e) \in S\}\right| = i\} \\ \mathsf{outdeg}^S &= \{(p, i) \in N \times \mathbb{N}\} : \left|\{e = (p, j, c) \in E : \mathfrak{L}(e) \in S\}\right| = i\}\end{aligned}$$

Notice, that the degree is defined over the number of edges labeled from S, not the number of child nodes reached over edges labeled from S. If the graph is simple, i.e., contains no multi-edges, both definitions are equivalent. But in the presence of multi-edges the given definition leads to more intuitive (and higher) degrees than a definition based on child nodes. The same abbreviations as above for unary sets and $S = \Sigma$ are used.

Ordered relation. For any finite set of labels $S \subset \mathcal{P}(\Sigma)$, we provide as convenience the *order specification test relation*

$$\mathfrak{O}^S = \{n \in N : \forall \sigma \in S : \mathfrak{O}(n, s)\}$$

The order specification $\mathfrak{O}$ is exposed directly.

Root relation. Finally, there is a *root node relation* $\mathsf{root} = R \subset N$ that identifies all root nodes in the data graph.

1.6.4 Structural Relations

The primitives for traversing the structure of the graph data are relations that connect nodes with incident edges or nodes with other nodes reachable via (arbitrary or fixed length) paths.

Source and sink relations. Traversal from edge to node and vice versa is achieved using the *source* and *sink relations*.

$$\begin{aligned}
\circ\!\!\rightarrow &= \{(n,e) \in N \times E : \exists i \in \mathbb{N}, n' \in N : (n,n',i) = e\} \\
\rightarrow\!\!\circ &= \{(n,e) \in N \times E : \exists i \in \mathbb{N}, n' \in N : (n',n,i) = e\}
\end{aligned}$$

$\rightarrow\!\!\circ$ and $\circ\!\!\rightarrow$ relate a node to *all* its in- resp. out-edges. To retrieve only one particular edge, e.g., the i-th out-edge, a combination of $\circ\!\!\rightarrow$ and @ can be used. To retrieve edges with a specific edge label $\circ\!\!\rightarrow$ can be combined with Lab.

Path relations. Structural node-node relations relate nodes that are connected by fixed length or arbitrary length paths (i.e., sequences of edges). For any $S \in \mathcal{P}(\Sigma)$, $i, j \in \mathbb{N} \cup \{\infty\}$ we define the *path relation*

$$\begin{aligned}
\mathsf{path}^S_{i,j} = \{(n,m) \in N^2 : \exists i \le k \le j, e_1, \dots, e_k \in E, n_1, \dots, n_{k-1} \in N :& \\
\circ\!\!\rightarrow^S(n,e_1) \wedge \rightarrow\!\!\circ^S(m,e_k) \wedge \mathsf{Lab}^S(e_k) \wedge& \\
\bigwedge_{l=1}^{k-1} (\rightarrow\!\!\circ^S(n_l,e_l) \wedge \circ\!\!\rightarrow^S(n_l,e_{l+1}) \wedge \mathsf{Lab}^S(e_l))\}&
\end{aligned}$$

Intuitively, two nodes are in $\mathsf{path}^S_{i,j}$ relation, if there is a path with length between i and j that connects the two nodes. For $i = j = 1$, this renders the direct edge (or child) relation between nodes. On XML data represented as in Figure 2 this renders, for $i = 0$, $j = \infty$ and $S = \{\textsc{child}, \textsc{comment}, \textsc{value}\}$, XPath's descendant-or-self, for $i = 1$, $j = \infty$, XPath's descendant or Xcerpt's desc. For the translations discussed in the following chapters, we only use those three types of path (child, descendant, and descendant-or-self).

If $i = j = 1$, we omit the interval index, if $i = 0$, $j = \infty$ we use $*$, if $i = 1$, $j = \infty$ we use $+$ as index. If $S = \Sigma_E$ we omit the label index, if $S = \{\lambda\}$ we write $\lambda_{i,j}$ for $\mathsf{path}^{\{\lambda\}}_{i,j}$.

1.6.5 Order Relations

The following relations are successor and order relations on edges and nodes based on the relative position of the edges or nodes under a common parent or ancestor. For each type, there is a (non-transitive) successor relation and a transitive order relation. The order relations are proper *(strict partial) orders* only for edges, for nodes (due to multi-edges, i.e., several edges with same source and sink but different edge position) the relations are strict *preorders*, i.e., not anti-symmetric. E.g., if a is connected by the first and sixth edge to b and by the third edge to c, c is both a following sibling ($b \ll_+ c$) of b and a preceding sibling ($c \ll_+ b$).

Order relations on edges. Since graphs may carry order, we can compare two edges wrt. their relative position within the out-edges of a common parent. For all $S \in \mathcal{P}(\Sigma)$, we define the *direct* $\prec^S$, the *transitive*

$\prec^S_+$, and the *transitive reflexive edge sibling relation* $\prec^S_*$:

$$\begin{aligned}
\prec^S = \{&(e,e') \in E^2 : \circ\!\!\rightarrow(n,e) \wedge \circ\!\!\rightarrow(n,e') \wedge \mathsf{Lab}^S(e) \wedge \mathsf{Lab}^S(e') \wedge \\
&@^S(e,i) \wedge @^S(e', i+1)\} \\
\prec^S_+ = \{&(e,e') \in E^2 : \circ\!\!\rightarrow(n,e) \wedge \circ\!\!\rightarrow(n,e') \wedge \mathsf{Lab}^S(e) \wedge \mathsf{Lab}^S(e') \wedge \\
&@^S(e,i) \wedge @^S(e', j) \wedge i < j\} \\
\prec^S_* = \{&(e,e') \in E^2 : \circ\!\!\rightarrow(n,e) \wedge \circ\!\!\rightarrow(n,e') \wedge \mathsf{Lab}^S(e) \wedge \mathsf{Lab}^S(e') \wedge \\
&@^S(e,i) \wedge @^S(e', j) \wedge i \le j\}
\end{aligned}$$

Intuitively, $\prec^S$ relates (under the common parent node) each edge to its immediate successors among the edges with label in $S \cap \mathfrak{O}(n)$. It is an injective function even in presence of multi-edges. $\prec^S_+$ relates (under a common parent node p) each node to all following edges outgoing from p with label in $S \cap \mathfrak{O}(n)$. As in the case of $@^S$, the labels S are only used to limit the considered edges.

Order relations on nodes. There are two types of positional relations on nodes: those relating siblings under a common parent and their generalization to the entire graph, i.e., positional relations relating arbitrary nodes in a given graph.

For any $S \in \mathcal{P}(\Sigma)$, there is a *direct* $\ll^S$, a *transitive* $\ll^S_+$, and a *transitive reflexive node sibling relation* $\ll^S_*$.

$$\begin{aligned}
\ll^S &= \{(n,m) \in N^2 : \rightarrow\!\!\circ(n,e) \wedge \rightarrow\!\!\circ(m,e') \wedge e \prec^S e'\} \\
\ll^S_+ &= \{(n,m) \in N^2 : \rightarrow\!\!\circ(n,e) \wedge \rightarrow\!\!\circ(m,e') \wedge e \prec^S_+ e'\} \\
\ll^S_* &= \{(n,m) \in N^2 : \rightarrow\!\!\circ(n,e) \wedge \rightarrow\!\!\circ(m,e') \wedge e \prec^S_* e'\}
\end{aligned}$$

It follows from the definition of $\prec^S$, that two nodes in $\ll^S$ must have a common parent. On nodes, $\ll^S$ is a function only on trees. Again the same abbreviations as above for unary sets and $S = \Sigma$ are used.

For all $S \in \mathcal{P}(\Sigma)$, there is also a *transitive* $\blacktriangleleft^S_+$, a *transitive reflexive* $\blacktriangleleft^S_*$, and a *direct following relation* $\blacktriangleleft^S$ on nodes.

$$\begin{aligned}
\blacktriangleleft^S_+ &= \{(n,m) \in N^2 : n' \ll^S_+ m' \wedge \mathsf{path}^S_*(n',n) \wedge \mathsf{path}^S_*(m',m)\} \\
\blacktriangleleft^S_* &= \{(n,m) \in N^2 : n' \ll^S_* m' \wedge \mathsf{path}^S_*(n',n) \wedge \mathsf{path}^S_*(m',m)\} \\
\blacktriangleleft^S &= \{(n,m) \in N^2 : n \blacktriangleleft^S_+ m \wedge \nexists n' \in N : n \blacktriangleleft^S_+ n' \wedge n' \blacktriangleleft^S_+ m \vee \mathsf{path}^S_*(n',n)\}
\end{aligned}$$

Intuitively, $\blacktriangleleft^S_+$ relates n to all nodes m that follow n within the subgraph induced by all edges with label in S. Notice that, $\blacktriangleleft^S_+(x,y) \implies \exists y' \blacktriangleleft^S (x,y')$ only if $\blacktriangleleft^S_+$ is acyclic and thus irreflexive. If $\blacktriangleleft^S_+$ is cyclic, there is no direct following for all nodes on the cycle.

On XML data represented as in Figure 2, $\ll^{\text{CHILD}}_+$ represents XPath's following-sibling, $\blacktriangleleft^{\text{CHILD}}_+$ XPath's following axis.

1.6.6 Equivalence Relations

The previous binary relations relate nodes and edges based on how they are connected in the data graph. Equally important is the ability to relate two nodes based on their local properties, e.g., their label, structure, or arity. In this section, we introduce several *equivalence* relations based on such properties.

Label equivalence relation. The *label equivalence* relation $\cong$ relates all nodes or edges that have the same label.

$$\cong = \{(t_1, t_2) \in (N \cup E)^2 : \exists \lambda \in \Sigma : \mathfrak{L}(t_1) = \lambda = \mathfrak{L}(t_2)\}$$

Identity equivalence relation. The *identity equivalence* relation $\doteq$ relates each node or edge to itself and itself only.

$$\cong = \{(t, t) \in (N \cup E)^2\}$$

Position equivalence relation. For any $S \in \mathcal{P}(\Sigma)$, the *position equivalence* relation $=_@^S$ relates all edges that occur at the same position among their respective source's out-edges.

$$=_@^S = \{(e, e') \in E^2 : \exists i \in \mathbb{N} : @^S(e, i) \wedge @^S(e', i)\}$$

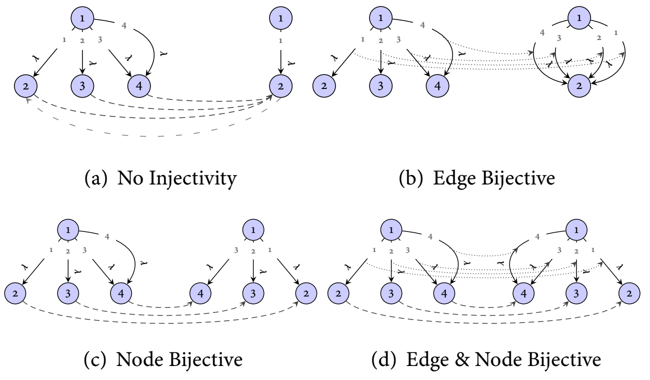
Again we use only λ for unary sets $S = \{\lambda\}$ and omit the index S for $S = \Sigma$.

Structural equivalence relation. As the label equivalence relation stand to the label relation, structural equivalence relations stand to edge and position relations. They relate nodes not based on the equivalence of their local properties, but on the equivalence of their structure, i.e., of the subgraph rooted at the respective node. Unfortunately, since they are dealing in equivalence not between atomic values like label equivalence relations but between structured values, their semantics and evaluation is considerably more complex. In the following, we introduce a flexible structural equivalence relation, referred to following common notation as deep equal. The introduced relation is flexible enough to cover a large set of existing or desirable specific structural equivalence relations:

XQuery, e.g., provides the deep-equal function that identifies pairs of nodes with a structure that is equivalent w.r.t. the XQuery data model (thus, e.g., disregarding attribute ordering and in-scope namespaces). [47] shows that the presence of deep-equal does not affect the complexity of composition-free XQuery. Recall, that composition-free XQuery is a restriction of XQuery similar to the algebra discussed in this work, where the domain of all query operators is limited to the input document. However, XQuery operates only on ordered tree data, where deep-equal (i.e., ordered tree isomorphism) is linear [4]. The generalization of XQuery's deep-equal to general unordered graphs, however, subsumes to graph isomorphism which is believed not to be in P and which exhibits, for the general case, only exponential-time deterministic algorithms [46]. However, there exist efficient algorithm for rather large classes of graphs, e.g., planar graphs [41].

Moreover different queries might require different notions of deep join: order may be significant or insignificant, certain edges or nodes (e.g., representing comments in XML) may be entirely ignored, and non-injective mappings may be acceptable to establish equivalence. From what occurs in practical XML and RDF query languages, these variances can be classified in three dimensions:

(1) What should be mapped bijective? In some cases, two nodes are considered equal already if all the structural information from one node occurs in some form in the other and vice versa. It is not required that multiple occurrences of same information is carried over. This roughly corresponds to simulation [59] as equivalence relation. On the other extreme, one might consider two nodes equivalent only when they are fully isomorph. This is, e.g., the semantics of XQuery's deep-equal.


(a) No Injectivity (b) Edge Bijective

(c) Node Bijective (d) Edge & Node Bijective

Figure 5. deep equal: effect of injectivity

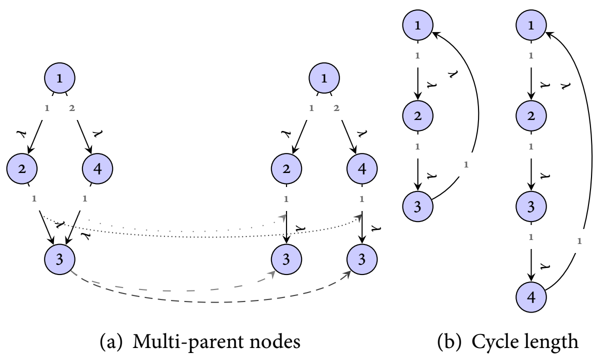

(a) Multi-parent nodes (b) Cycle length

Figure 6. deep equal: effect of cover for equivalence mapping

What makes these cases different is whether the mapping between the two nodes and their respective substructures is bijective or not. More precisely, one can distinguish deep equals by the "degree" of bijectivity required:

(a) *Type cover:* The first choice lies in whether we demand that the mapping is (i) not bijective at all, (ii) bijective only on edges, (iii) bijective only on nodes, or (iv) bijective on both. On trees, case (ii) to (iv) are obviously equivalent, but in presence of multi-edges or cycles differences emerge. Figure 5 illustrates the differences between the different forms.

(b) *Structural cover:* Aside from the question which types are covered by the mapping, a further variance lies in the structural extent of the cover: either (i) the entire (reachable) subgraphs rooted at the two nodes, (ii) only adjacent edges and nodes, or (iii) only outgoing edges and children. In trees all three forms are equivalent, but in graphs the latter two are less restrictive than the first: Figure (a) shows an example of two graphs that are equivalent under (iii), but not under (i) or (ii). In general, (iii) can not distinguish DAGs from trees in all cases. Figure (b) shows a case where (ii) considers the

$$
\begin{aligned}
\mathsf{nds}^{L} &: N \to \wp(\mathbb{N}) = \left\{(n, N_n) \,\middle|\, n \in N \wedge N_n = \{n' \in N : \exists \lambda \in L : \lambda(n, n')\} \cup \{n\}\right\}\\
\mathsf{nds}^{L}_{*} &: N \to \wp(\mathbb{N}) = \left\{(n, N_n) \,\middle|\, n \in N \wedge N_n = \{n' \in N : \exists \lambda \in L : \lambda_{+}(n, n')\} \cup \{n\}\right\}\\
\mathsf{eds}^{L} &: N \to \wp(\mathbb{N}) = \left\{(n, E_n) \,\middle|\, n \in N \wedge E_n = \{(n, i) \in N \times \mathbb{N} : \exists n' \in \mathsf{nds} : (n, n', i) \in E\}\right\}\\
\mathsf{eds}^{L}_{*} &: N \to \wp(\mathbb{N}) = \left\{(n, E_n) \,\middle|\, n \in N \wedge E_n = \{(n', i) \in N \times \mathbb{N} : \exists n', n'' \in \mathsf{nds}^{*} : (n', n'', i) \in E\}\right\}
\end{aligned}
$$

$$
\begin{aligned}
\sim^{L,L_O} = &\left\{(n, m, f, g, \circeq) \,\middle|\, \forall c_n \in N, i \in \mathbb{N} : e = (n, c_n, i) \in E \wedge \mathfrak{L}(e) \in L \implies\right.\\
&\qquad \exists i' \in \mathbb{N} : g(n, i) = (m, i') \wedge e' = (m, f(c_n), i') \in E \wedge \cong (e, e') \wedge \circeq^{L,L_O} (c_n, f(c_n))\\
&\quad \wedge (\lambda \in L_O \cap \mathfrak{O}_N(n) \implies \forall \lambda' \in L_O \cap \mathfrak{O}_N(n), k \in \mathbb{N} :\\
&\qquad \left. e'' = (n, f(c_n), k) \in E \wedge \mathfrak{L}(e'') = \lambda' \wedge k \leq i \implies \exists k' : g(n, k) = (m, k') \wedge k' \leq i' \right\}
\end{aligned}
$$

$$
\begin{aligned}
\circeq^{L,L_O} = &\left\{(n, m) \in N^2 \,\middle|\, =_{\mathsf{label}} (n, m) \wedge \mathfrak{O}_N(n) = \mathfrak{O}_N(m) \wedge \left(\exists f : \mathsf{nds}^{L}(n) \to \mathsf{nds}^{L}(m), f' : \mathsf{nds}^{L}(m) \to \mathsf{nds}^{L}(n),\right.\right.\\
&\left.\left. g : \mathsf{eds}^{L}(n) \to \mathsf{eds}^{L}(m), g' : \mathsf{eds}^{L}(m) \to \mathsf{eds}^{L}(n) : \wedge \sim^{L,L_O} (n, m, f, g, \circeq) \wedge \sim^{L,L_O} (m, n, f', g', \circeq)\right)\right\}\\
\circeq^{L,L_O}_{bij} = &\left\{(n, m) \in N^2 \,\middle|\, =_{\mathsf{label}} (n, m) \wedge \mathfrak{O}_N(n) = \mathfrak{O}_N(m) \wedge \left(\exists f : \mathsf{nds}^{L}(n) \to \mathsf{nds}^{L}(m), g : \mathsf{eds}^{L}(n) \to \mathsf{eds}^{L}(m) :\right.\right.\\
&\left.\left. f, g \text{ bijective} \wedge \sim^{L,L_O} (n, m, f, g, \circeq_{bij}) \wedge \sim^{L,L_O} (m, n, f^{-1}, g^{-1}, \circeq_{bij})\right)\right\}\\
\circeq^{L,L_O}_{*} = &\left\{(n, m) \in N^2 \,\middle|\, \cong (n, m) \wedge \mathfrak{O}_N(n) = \mathfrak{O}_N(m) \wedge \left(\exists f : \mathsf{nds}^{L}_{*}(n) \to \mathsf{nds}^{L}_{*}(m), g : \mathsf{eds}^{L}_{*}(n) \to \mathsf{eds}^{L}_{*}(m) :\right.\right.\\
&\left.\left. \wedge \sim^{L,L_O} (n, m, f, g, \circeq_{*}) \wedge \sim^{L,L_O} (m, n, f^{-1}, g^{-1}, \circeq_{*})\right)\right\}\\
\circeq^{L,L_O}_{*,bij} = &\left\{(n, m) \in N^2 \,\middle|\, \cong (n, m) \wedge \mathfrak{O}_N(n) = \mathfrak{O}_N(m) \wedge \left(\exists f : \mathsf{nds}^{L}_{*}(n) \to \mathsf{nds}^{L}_{*}(m), g : \mathsf{eds}^{L}_{*}(n) \to \mathsf{eds}^{L}_{*}(m) :\right.\right.\\
&\left.\left. f, g \text{ bijective} \wedge \sim^{L,L_O} (n, m, f, g, \circeq_{*,bij}) \wedge \sim^{L,L_O} (m, n, f^{-1}, g^{-1}, \circeq_{*,bij})\right)\right\}
\end{aligned}
$$

Figure 7. DEEP EQUAL ($L_O \subset L \subset \Sigma_E$)

two graphs equivalent, but (i) does not. In general, (ii) fails to distinguish graphs with cycles differing only in cycle length.

(2) WHAT SHOULD BE MAPPED AT ALL? Some parts of the structure may be excluded, e.g., comments in XML data or annotation properties (such as rdfs:seeAlso or rdfs:label) in RD data. The deep equal presented below allows to limit the mapping to certain given edge labels L.

(3) WHAT SHOULD BE MAPPED IN ORDER? In some cases, one might be interested in preserving the order of the data (if it is ordered at all). However, in other cases (in particular for non-bijective mappings) the order may be irrelevant. As in the previous case, the deep equal presented here allows to limit the order-respecting edge labels $L_O \subset L$.

Figure 7 shows the formal definition of the generalized deep equal relation $\circeq$ (omitting analogous variant 2b): We define four variants of $\circeq$ that differ, as explained above, in what nodes and edges are mapped and whether the mapping is bijective. The basic case, $\circeq$, maps only adjacent nodes and edges (more precisely, eds are edge positions in the context of each node). Its bijective variant, $\circeq_{bij}$, also only maps adjacent nodes and edges, but does so using a bijective mapping. Analog variants, $\circeq_{*}$ and $\circeq_{*,bij}$, exist for the case where all nodes and edges in the subgraph are mapped. Again, if L and L_O are Σ_E we omit the superscript.

All definitions use $\sim$ to express first that each L-child of the first node must be mapped to a corresponding child of the second node such that the two children are deep equal. Second, the definition ensures that the order of edges, where they are ordered in the first place and covered in L_O, is preserved by the mapping. The definition takes care that the order specification of the data takes precedence over

L_O, i.e., membership in L_O preserves order of respective edges only if the edges are considered ordered in the data.

1.6.7 Inverse and Complement

We conclude the set of relations on data graphs by inverse and complement relations for each of the previously defined ones.

Complement Relations. For each of the basic binary relations R on nodes and/or edges (but not on integers), we introduce the *complement relation*

$$\complement R = \begin{cases} (N \cup E)^2 \setminus R & \text{if } R \subset N \cup E \text{ label or identity equivalence relation} \\ N^2 \setminus R & \text{if } R \subset N^2 \text{ structural equiv., path, or node order rel.} \\ E^2 \setminus R & \text{if } R \subset E^2 \text{ edge order} \\ (N \times E) \setminus R & \text{if } R \subset N \times E \text{ source or sink relation} \\ N \setminus R & \text{if } R \subset N \text{ root relation} \\ N \cup E \setminus R & \text{if } R \subset N \cup E \text{ and not } R \subset N \text{ label relation} \end{cases}$$

In CIQLog and the CIQCAG algebra these relations are not strictly needed and can be simulated by a negation of the original relation.

Inverse Relations. For each of the basic binary relations R, we also introduce the *inverse relation* $\overline{R} = \{(x,y) \in (N \cup E)^2 : R(y,x)\}$. Though these relations do not add to the expressiveness of conjunctive queries as defined here (cf., e.g., [61]) they can be exploited to rewrite certain classes of graph queries to tree queries as described in [60]. Consider, e.g., the graph query $Q(m) \to \textsc{child}(v_1, v_2) \wedge \textsc{child}(v_3, v_2) \wedge \textsc{child}(v_2, v_4)$. This can be rewritten to $Q'(m) \to \textsc{child}(v_1, v_2) \wedge \overline{\textsc{child}}(v_2, v_3) \wedge \textsc{child}(v_2, v_4)$, where the relation between v_3 and v_2 is inverted making the query tree-shaped.

1.6.8 Example relations

We conclude the discussion of relations on data graphs by looking back to the data from Figure 1. For that data, Table 2 gives some of the relations derived from the data graph in accordance to the above definitions.

1.7 Conclusion

The data model, an ordered, semi-structure data graph with *node and edge* labels, for CIQLog and CIQCAG is an abstraction of data models for common Web query languages. As such, it provides a rich set of relations on data graphs, discussed in Section 1.6, for CIQLog enabling CIQLog to be the target of for the translate of large fragments of Xcerpt, XQuery, and SPARQL, see Chapters 3 to 5. The data model also proves to be sufficiently expressive to capture both XML and RDF data, as well as Xcerpt data terms

$\mathsf{Lab}^{\textsc{attr}} = \{(d_4, 3, d_5), (d_4, 2, d_{10}), (d_{11}, 2, d_{14}), (d_1, 5, d_{20})\}$	
$\mathsf{Lab}^{\textsc{paper}} = \{d_4, d_{11}\}$	$\mathsf{Lab}^{\text{‘Cicero’}} = \{d_8, d_{17}\}$
$\cong = \{(d_7, d_{12}), (d_4, d_{11}), (d_8, d_{17}), (d_{16}, d_{18}), \ldots\}$	
$\doteq = \{(d_1, d_1), (d_2, d_2), \ldots, ((d_1, 1, d_2), (d_1, 1, d_2)), \ldots\}$	
$\overset{\circ}{=} = \{(d_8, d_{17})\}$	$\overset{\circ}{=}^{\{\textsc{child},\textsc{attr}\}} = \{(d_8, d_{17}), (d_4, d_{11}), (d_{16}, d_{18})\}$
$\circ\!\!\rightarrow = \{(d_1, (d_1, 1, d_2)), (d_1, (d_1, 2, d_4)), \ldots\}$	$\rightarrow\!\!\circ = \{(d_2, (d_1, 1, d_2)), (d_4, (d_1, 2, d_4)), \ldots\}$
$\textsc{child} = \{(d_1, d_2), (d_1, d_4), (d_1, d_{11}), (d_1, d_{15}), (d_4, d_7), \ldots\}$	
$\textsc{child}_* = \{(d_1, d_2), (d_1, d_4), (d_1, d_7), (d_1, d_{11}), (d_1, d_{12}), \ldots\}$	
$\mathsf{path}_*^{\textsc{child},\textsc{comment},\textsc{value}} = \{(d_1, d_2), (d_1, d_3), (d_1, d_4), (d_1, d_7), (d_1, d_8), (d_1, d_9), (d_1, d_{11}), \ldots\}$	
$@ = \{(d_{11}, 3), (d_{11}, 1), ((d_1, 3, d_{11}), 3), ((d_{10}, 1, d_{11}), 1), \ldots\}$	
$\ll = \{(d_2, d_4), (d_4, d_{11}), (d_{11}, d_{15}), \ldots\}$	$\ll_+ = \{(d_2, d_4), (d_2, d_{11}), (d_2, d_{15}), \ldots\}$
$\blacktriangleleft = \{(d_3, d_4), \ldots, (d_{17}, d_{18}), \ldots\}$	$\ll_+ = \{(d_3, d_4), (d_3, d_5), (d_3, d_6), \ldots\}$

Table 2. (Partial) instances for data graph relations on Figure 1

(which have, as XML, many different types of data items structured rather freely into arbitrary graphs, as in RDF). Before turning to the translation from XQuery, Xcerpt, and SPARQL, we outline CIQLog, our formal query language on query data graphs in the following chapter.

Chapter 2

Queries—CIQLog: Datalog¬ with Complex Heads

2.1 Introduction

As formal foundation for Web queries, we introduce CIQLog, a rule-based query language tailored to semi-structured queries. CIQLog is a slightly modified variant of $\text{datalog}^{\neg}_{new}$, i.e., *datalog extended with negation and value invention*, which is most prominently represented by ILOG [42]. We only consider binary relations, but extend $\text{datalog}^{\neg}_{new}$ with a partial order on edges (in the spirit of IDLOG [69]) and some (syntactic) conveniences such as conjunction in rule heads and disjunction in rule bodies. For most of the translations in the following chapters and the translation to CIQCAG, we focus on the weakly recursive fragment of CIQLog, denoted as $\text{CIQLog}^{\text{wR}}$. Following [42], a CIQLog program is weakly recursive, if there is no recursion *through value invention*.

Note, that CIQLog's value invention is, as that of ILOG [42] or RDF query languages such as RDFLog [15], based on the essential observation that the actual invented value (in our case node or edge) carries no information whatsoever. Only its membership in relations is material. The same applies for edge positions: We do not care how they are represented (though we usually choose integers) as long as they carry the correct order relations. In particular, there is no requirement that the edge positions are consecutive and thus no need for a successor relation.

2.2 CIQLog Syntax

In the following, we briefly summarize syntax and semantics of CIQLog: A CIQLog rule R consists of a query *head* and a query *body*. The query body is a (quantifier-free) formula over binary and unary atoms. Each atom is a relation over query variables from the underlying relational structure D. The domain of the query variables is the domain of D. The query head is mostly a conjunction of atoms over answer variables and outlined in detail below in Section 2.2.1. All answer variables must occur also in the body of the query. All other variables in the query body are existentially quantified [1].

$\langle program\rangle$	::=	$\langle rule\rangle^*$
$\langle rule\rangle$	::=	$\langle head\rangle$ '$\longleftarrow$' $\langle expression\rangle$
$\langle expression\rangle$	::=	$\langle atom\rangle$ | $\langle negation\rangle$ | $\langle conjunction\rangle$ | $\langle disjunction\rangle$
$\langle atom\rangle$	::=	$\langle relation\rangle$ '(' $\langle variable\rangle$ (',' $\langle variable\rangle$)? ')'
$\langle negation\rangle$	::=	$\neg$ $\langle expression\rangle$
$\langle conjunction\rangle$	::=	'('$\langle expression\rangle$ '$\wedge$'$\langle expression\rangle$ ')'
$\langle disjunction\rangle$	::=	'('$\langle expression\rangle$ '$\vee$'$\langle expression\rangle$ ')'

Table 3. CIQLog syntax (without heads)

Each variable must be either a *node, edge, edge position, or label variable*, i.e., the domain of each variable is either node, edge, edge position, or label. If a rule contains a variable that occurs at the position of two or more attributes with different domains, this rule is *invalid*, otherwise it is *valid*.

Answer variables are variables that occur in the head outside of the condition of a conditional expression. The usual safety restrictions [1] for datalog apply to ensure that all rules are *range-restricted*: For each negation, all answer variables must occur also in a positive expression in the rule body. For each disjunction, all nested expressions have the same answer variables. Finally, each answer variable must also occur in the body.

2.2.1 Complex Heads

CIQLog differs notably from standard datalog in the shape of rule heads, as specified in Table 4:

(1) VALUE AND ORDER INVENTION: CIQLog allows terms over answer variables for rule and order invention where datalog$^\neg$ allows only constants and variables. Value invention is the same as in datalog$^\neg_{new}$ though we use a term notation as in datalog$^\neg_{obj}$: A value invention term is a term over the invention variables with functor new parametrized by an equivalence relation and an identifier. The identifier allows multiple value invention statements in the same head each returning a distinct set of new values. The equivalence relation is used to determine when two binding tuples for the invention variables are considered equivalent (and thus yield the same new value). Otherwise, a value invention term is interpreted as any other function term on nodes or edges and maps to either a node, edge, or edge position (but not a label as new labels can not be invented). As for variables in the body, value invention terms may occur either at the position of node-, edge-, *or* edge position-valued attributes. If the same value invention term occurs at the position of two or more attributes with different domains, that rule is *invalid*. In the following, we only consider *valid* CIQLog rules.

Order invention terms are similar but map to integer values (rather than nodes and edges). Furthermore, they are parametrized by order relations rather than equivalence relations. An order term $t_1 = \mathsf{order}_{<_N}(g, o, \vec{x})$ is a term over the order *term* g of the super-group, the offset o within the group, and a set of pairs of order invention variables and their mappings. It maps to an edge position (i.e., an integer) such that, for any order term $t_2 = \mathsf{order}_<(g_2, o_2, \vec{x}_2)$, $t_1 < t_2$ if $g = g_2 \wedge (\vec{x} <_N \vec{x}_2 \vee \vec{x} = \vec{y} \wedge o < o_2))$ or if $t_1' < t_2'$

⟨*head*⟩	::=	⟨*hexpression*⟩
⟨*hexpression*⟩	::=	⟨*hatom*⟩ | ⟨*hconjunction*⟩ | ⟨*conditional*⟩ | ‘()’
⟨*hatom*⟩	::=	⟨*hrelation*⟩ ‘(’ ⟨*hterm*⟩ (‘,’ ⟨*hterm*⟩)? ‘)’
⟨*hterm*⟩	::=	⟨*variable*⟩ | ⟨*invention*⟩ | ⟨*order*⟩ | ⟨*aggregation*⟩
⟨*invention*⟩	::=	$\mathsf{new}^{\langle \mathit{identifier} \rangle}_{\langle \mathit{equivalence\text{-}rel} \rangle}$(‘(’ ⟨*variable*⟩+ ‘)’)?
⟨*aggregation*⟩	::=	(**sum**| **max**| **avg**(’ ⟨*variable*⟩ ‘)’
⟨*order*⟩	::=	$\mathsf{order}_{\langle \mathit{order\text{-}rel} \rangle}$‘(’ ⟨*order*⟩?, ⟨*offset*⟩, ⟨*variable*⟩* ‘)’ | ⟨*integer*⟩
⟨*hconjunction*⟩	::=	‘(’ ⟨*hexpression*⟩ ‘∧’ ⟨*hexpression*⟩ ‘)’
⟨*conditional*⟩	::=	‘**if**’ ⟨*condition*⟩ ‘**then**’ ⟨*hexpression*⟩ ‘**else**’ ⟨*hexpression*⟩
⟨*condition*⟩	::=	⟨*variable*⟩ (‘=’ | ‘≠’) ‘**nil**’
⟨*offset*⟩	::=	⟨*integer*⟩

Table 4. Heads of CIQLog rules

with $t_1' \in t_1$ and $t_2' \in t_2$ and no two $t_1' \in t_1'' \in t_1$ or $t_2' \in t_2'' \in t_2$ exist. Let $\top$ be the "empty" order term that is neither smaller or larger than any other term (but used to complete top-level order terms) and equal only to itself. Order terms contain $\top$ or other order terms as order terms for their super group in a grouping expression with nested but ordered groupings. The nesting is only on the first argument and thus linear. The latter part of the definition makes two ordered terms stand in $<$ relation, if they contain nested order terms (at arbitrary level) that themselves stand in $<$ relation (and thus, eventually, have a common super group and either preceding bindings for the same grouping variables or same bindings and preceding order number). $\vec{x}$ and $\vec{y}$ may be empty. The empty tuple is equal only to itself and stands in $<_N$ relation to no other tuple. Order invention terms may only occur in place of edge position-valued attributes. $<_N$ is an order relation on (named) tuples of nodes or edges. A typical example of $<_N$ is the component-wise lexical order $<_{lex}$ on the label of those nodes or edges: E.g., $\langle a : x_1, b : x_2 \rangle <_{lex} \langle a : y_1, b : y_2 \rangle$ if $\mathfrak{L}(x_1)$ is in lexical order before $\mathfrak{L}(y_1)$ or both labels are the same and $\mathfrak{L}(x_1)$ is in lexical order before $\mathfrak{L}(y_1)$. For further examples of order terms see Chapters 3 and 4.

(2) Conditional construction: For convenience, we allow conditional construction in the head: some part of the head depends on a condition on an answer variable (viz., that variable being **nil** or not). Conditional construction $h \wedge$ **if** X = **nil** **then** hc_1 **else** $hc_2 \longleftarrow b$, can be rewritten to rules without conditional constructions as follows:

```
h ∧ hc₁ ⟵ b ∧ X = nil
h ∧ hc₂ ⟵ b ∧ X ≠ nil
```

(3) Aggregation: As ILOG, CIQLog extends rule heads with aggregation on integers in the spirit of [48]. Aggregation has no further effect on expressiveness and complexity in presence of value invention.

Adapting the notation of [42], we call an *invention atom* an expression containing either new or order terms. The relation name of that atom is called an *invention relation name*. A rule is a *non-invention* *rule*, if it contains no invention atom in the head, otherwise it is an *invention rule*. A *weakly recursive* CIQLog program (or CIQLog$^{\text{wR}}$ program) is a CIQLog program where no invention rule depends (directly or indirectly) on another invention rule. We say that a program P *has cascading value invention*, if there are two invention rules R, R' such that R depends on R'.

Only the first extension has an effect on the expressive power of CIQLog. It makes CIQLog essentially an ILOG [42] variant extended with (partial) order on edges (though no successor relation as in IDLOG).

For value invention terms, we mostly omit the equivalence relation in the following assuming identity $\doteq$.

2.3 CIQLog Semantics

We characterize the semantics of CIQLog in three ways in the following:

(1) The intuitive semantics of a CIQLog program is that of a logic program with aggregates [40] where we replace all occurrences of new and order in the result by unique new nodes, edges, or edge positions. The last operation is similar to un-Skolemization [26, 15].

(2) Together with the observation, that new and order are thus nothing else but Skolem terms with implicit relations on the results of order that can, as well, be expressed by additional CIQLog rules, we notice that the semantics of CIQLog can be defined by reduction of ILOG [42] whose semantics is based on Skolem terms and logic programming with aggregates.

(3) Finally, we give an algebraic semantics based on fixpoint and relational algebra operations that is useful both for the translations in the following chapters and for the equivalence to the CIQCAG algebra.

In the following, we assume that disjunction in the body and conditional construction in the head is removed as outlined above.

Definition 2.1 (Logic-based Semantics of CIQLog). Let P be a valid, range-restricted CIQLog program. Then let S be the semantic of P considered as a logic program (with aggregates). If S is infinite, the semantics of P is undefined. Otherwise, we replace each value invention terms in S with a new node, edge, or edge position (depending on the domain of the attribute it occurs in; recall that if P is valid, it occurs in only one of these three types of attributes). Order invention terms are replaced with edge positions such that the order constraints between order invention terms are preserved[1].

2.3.1 Expressiveness and Complexity

This characterization gives an intuitive and easy semantics for CIQLog. To judge expressiveness, complexity, and completeness properties of CIQLog the second, equivalent, characterization of the semantic of CIQLog programs by means of ILOG is more helpful. The following theorem establishes that CIQLog is essentially a variant of ILOG:

[1]This can be achieved by determining some partial order on the order invention terms and assigning edge positions (integers) in accordance to that partial order.

Theorem 2.1. CIQLog *has the same expressiveness, complexity, and completeness properties as ILOG [42].*

Proof. Each ILOG program containing only binary relations is a CIQLog program if we replace each invention symbol with a new (with new identifier) over all non-invention variables of the invention atom. For ILOG programs with n-ary relations we construct a binary decomposition of the n-ary relations as in RDFLog [15].

On the other hand, each CIQLog without order terms can be transformed into an equivalent ILOG program in the following way:

(1) For each invention term t, introduce a new creation rule containing a new predicate over the invention variables of t and one invention symbol. In each rule using t, add an atom querying that rule in the body and replace t with the variable bound to the attribute at the position of the invention symbol. The resulting program is an ILOG program (contains only datalog rules and ILOG invention rules). Its semantic is the same as that of the CIQLog program since invention symbols are replaced by Skolem terms in the semantics of ILOG.

(2) Each order term can be transformed analogously, but introduce cascading value invention.

(3) Aggregates are also allowed in ILOG.

The result of the ILOG program is up to an isomorphism between new OIDs and node, edge, and edge positions equivalent to the result of the CIQLog program. □

Corollary 2.1. *From the reduction to ILOG, it follows that*

(1) CIQLog *expresses all computable queries modulo copy removal, cf. [42].*

(2) CIQLog *is (list) constructive complete (in the sense of [16]), cf. [17].*

(3) CIQLog *is* not *determinate complete (in the sense of [1]), cf. [16].*

(4) *already (negation) stratified* CIQLog *expresses all computable queries modulo copy removal, cf. [17].*

In particular, if we limit recursion to non-invention rules, we can "postpone" value invention to the very end of query evaluation:

Corollary 2.2. CIQLog^{wR} *has the same data and program complexity as* $\text{datalog}^{\neg}$*: in P, resp., NEXPTIME-complete.*

If recursion is prohibited entirely, value invention again has no effect on complexity:

Corollary 2.3. *Non-recursive* CIQLog*, i.e.,* CIQLog *where recursion is not allowed for any rule, has the same data and program complexity as non-recursive* $\text{datalog}^{\neg}$*: in* AC_0*, resp., PSPACE-complete.*

2.3.2 Deep and Shallow Copies

In presence of value invention, the creation of *shallow* and *deep copies* of a data item are often considered essential facilities. For CIQLog we consider shallow and deep copy for nodes only (since edges and edge positions have no "structure"). The shallow copy of a node is a *new* node with the same label, if any, and the same children as the original node. The deep copy of a node n is a *new* node n' such that n and n' have the same label and for each out-going edge of n to a child c_n there is an outgoing edge for n' with the same label and same edge position to a child $c_{n'}$ such that c_n and $c_{n'}$ are themselves deep copies.

We add shallow and deep clone relations to CIQLog heads, denoted as deep-copy(X,Y) and shallow-copy(X,Y) where Y is a new node and X is the original node. In full CIQLog, shallow and deep copy can be implemented as the following rules:

```
𝔏(Y,L)
⟵  deep-copy(X,Y) ∧ 𝔏(X,L) ∧ ¬○→(X, E).
𝔏(Y,L) ∧ ○→(Y,new₁(X,Z)) ∧ →○(new₂(X,Z),new₁(X,Z)) ∧
    pos(new₁(X,Z),EPos) ∧ deep-copy(Z,new₂(X,Z))
⟵  deep-copy(X, Y) ∧ 𝔏(X, L) ∧ ○→(X, E) ∧ →○(Z, E) ∧ pos(E, EPos).
```

```
𝔏(Y,L)
⟵  shallow-copy(X,Y) ∧ 𝔏(X,L) ∧ ¬○→(X,E).
𝔏(Y,L) ∧ ○→(Y,new₁(X,Z)) ∧ →○(Z,new₁(X,Z)) ∧ pos(new₁(X,Z),EPos)
⟵  shallow-copy(X,Y) ∧ 𝔏(X,L) ∧ ○→(X,E) ∧ →○(Z,E) ∧ pos(E,EPos).
```

However, the resulting program has necessarily cascading value invention. Furthermore, the rule-based realization is, in general, less efficient than a specialized operator. To provide deep- and shallow-copy also to CIQLog$^{\text{wR}}$, we define them as specialized operators with the above semantics. Note, that both deep- and shallow-copy are linear time, constant additional space operations and run in $\mathcal{O}(|N| + |E|)$. They are considered value invention operators and, thus, may in CIQLog$^{\text{wR}}$ not occur in recursive rules.

2.3.3 Algebraic Semantics

A third characterization of the CIQLog semantics is given by a translation of CIQLog rules to relational algebra expressions with value invention. Combined with a fixpoint operator, the resulting expressions yield a semantics of CIQLog. The target language is roughly $\mathbf{while}_{new}$ of [1], but uses invention terms instead of a dedicated invention relation. The advantage of this characterization is that it yields a very compact semantics closely based on relation algebra expressions, in particular, for CIQLog$^{\text{wR}}$ and non-recursive CIQLog.

In the following, we denote, for any sub-formula q of a rule R, with free(q) the variables in q that also occur outside of q. For a set of attributes $A = \{a_1, \dots, a_n\}$ let $t \in D^A$ be an $|A|$-ary tuple $\langle a_1 : v_1, \dots, a_n : v_n\rangle$ over the domain D with $t[a_i]$ the value of t for attribute a_i. We allow for *partial relational structures* D where we denote that a set of tuples T over attributes U is an instance of a relation schema $RN[U]$ by $RN \mapsto T$ and consider a partial relational structure as a set of such mappings. A partial relational structure is *complete* wrt. a relational schema S if it contains mappings for all relation schemas in S. We call $D' = D_1 \uplus D_2$ the *union of partial relational structures* such that $D' = \{RN \mapsto T_1 \cup T_2 : RN \mapsto T_1 \in D_1 \wedge RN \mapsto T_2 \in D_2\}$ where $RN \mapsto \emptyset$ if there is no mapping of RN in a relational structure.

Using these definitions, Table 5 gives the algebraic semantics of a CIQLog rule. Recall, that all rules are range-restricted. We use $[\![\ ]\!]_b$ to define the semantics of the body of a rule, $[\![\ ]\!]_h$ that of a head. The body of a rule results in a single relation over the free variables of the query. The head of a rule is evaluated once for each tuple resulting from the evaluation of the body of the query, each time replacing all the occurrences of all query variables by their bindings. A deep copy operation is replaced by a relational structure D containing y and a copy of all nodes reachable from x as well as their relations. A shallow copy operation is replaced by a relational structure D containing y, the same label relation on y as exists on x, and all nodes reachable from x (excluding x) and their relations as well as edges from y to all children of x. The result of $[\![\ ]\!]_h$ is a *pre-instance* in the sense of [42], i.e., it still contains value and order invention terms.

$\llbracket head \leftarrow expr \rrbracket$	$= \llbracket head \rrbracket_h (\llbracket expr \rrbracket_b)$
$\llbracket rel(x_1, \dots, x_n) \rrbracket_b$	$= \{ t \in D^{\{x_1,\dots,x_n\}} : (t[x_1], \dots, t[x_n]) \in \llbracket rel \rrbracket_b \}$
$\llbracket \neg(expr) \rrbracket_b$	$= \{ t \in D^A : A = \mathsf{free}(expr) \wedge t \in D^A \setminus \pi_A(\llbracket expr \rrbracket_b) \}$
$\llbracket (expr_1 \wedge expr_2) \rrbracket_b$	$= \pi_A(\llbracket expr_1 \rrbracket_b \bowtie \llbracket expr_2 \rrbracket_b), A = \mathsf{free}(expr_1 \wedge expr_2)$
$\llbracket (expr_1 \vee expr_2) \rrbracket_b$	$= \llbracket \pi_A(expr_1 \rrbracket_b) \cup \pi_A(\llbracket expr_2 \rrbracket_b), A = \mathsf{free}(expr_1)$
$\llbracket expr \rrbracket_h (\beta)$	$= \biguplus_{\langle x_1:d_1,\dots,x_k:d_k \rangle \in \beta} (\llbracket expr\{x_1/d_1, \dots, x_k/d_k\} \rrbracket_h)$
$\llbracket (hexpr_1 \wedge hexpr_2) \rrbracket_h$	$= \llbracket hexpr_1 \rrbracket_h \uplus \llbracket hexpr_2 \rrbracket_h (\beta)$
$\llbracket \mathtt{if}\ c\ \mathtt{then}\ hexpr_1\ \mathtt{else}\ hexpr_2 \rrbracket_h$	$= \llbracket hexpr_1 \rrbracket_h$ if $\llbracket c \rrbracket_h = \mathbf{true}$, $\llbracket hexpr_2 \rrbracket_h$ otherwise
$\llbracket () \rrbracket_h$	$= \varnothing$
$\llbracket rel(t_1, \dots, t_n) \rrbracket_h$	$= \{ rel \mapsto \{t_1, \dots, t_n\} \}$
$\llbracket x = \mathtt{nil} \rrbracket_h$	$= \mathbf{true}$ if $x = \mathtt{nil}$, $\mathbf{false}$ otherwise
$\llbracket x \neq \mathtt{nil} \rrbracket_h$	$= \mathbf{true}$ if $x \neq \mathtt{nil}$, $\mathbf{false}$ otherwise
$\llbracket \mathsf{deep\text{-}copy}(x, y) \rrbracket_h$	$= \mathsf{deep\text{-}copy}(x, y)$
$\llbracket \mathsf{shallow\text{-}copy}(x, y) \rrbracket_h$	$= \mathsf{shallow\text{-}copy}(x, y)$

Table 5. Algebraic CIQLog Semantics (D the domain of the relational structure)

These are replaced by new nodes, edges, and edge positions accordingly as described above. Computing an instance from an pre-instance can be done by assembling the order invention terms in a partial order, and then replacing them according to that order. Nested order terms do not pose a challenge, as we can replace the nested terms by first ordering the depth 1 terms, then add the depth 2 (respecting the order among the depth 1 terms contained in those depth 2 terms, etc. Together with the replacement of node invention terms that computation is in $\mathcal{O}(o \cdot \log o)$ time using $\mathcal{O}(\log o)$ additional space where o is the size of the pre-instance, assuming in-place sorting. Thus, the replacement of value and order invention terms does not affect the overall complexity of evaluating a single rule in CIQLog (which is the complexity of relational algebra, i.e., in L wrt. data complexity, PSPACE wrt. query complexity).

To obtain a semantics for full CIQLog we complement the above evaluation of a single rule (given the current set of derived facts represented as a relational structure) by a standard fixpoint operator (or iteration construct such as used in [1] for $\mathtt{while}_{new}$).

Theorem 2.2. *A single application of the fixpoint operator on the algebraic semantics yields an equivalent result as a single application of the fixpoint operator on the logic-based semantics.*

Proof. For simplicity, we assume stratified negation. Recall, from [17] that, in contrast to Datalog, the restriction to negation stratified programs does not limit the expressiveness of ILOG and thus CIQLog. Furthermore, we limit ourselves to rules without conjunction, conditional construction, and deep or shallow copy in the head or disjunction in the body (all these features can be rewritten to equivalent CIQLog programs beforehand).

It is easy to very that the algebraic semantics given in Table 5 on rules without those features represents all facts derivable from the given relational structure (representing the facts derived by the last application of the fixpoint operator) by the rule. Note, that the result is a relational structure containing a non-empty instance for exactly one relation, viz. the one of the head atom. $\square$

This concludes our discussion of the CIQLog semantics. Before we put CIQLog to work in evaluating Xcerpt, XQuery, and SPARQL in the following chapters, we discuss in the following, briefly, the querying of data graphs as introduced in Chapter 1, as well as the fragment of CIQLog used primarily for the translations.

2.4 Data Graphs in CIQLog: Extensional and Intensional Relations

In CIQLog, only a small set of relations on data graphs need to be extensional, i.e., represented as sets of tuples. The remaining relations can be realized by intensional definitions as CIQLog rules on top of this set.

The basic, extensional relations are, unsurprisingly, $\circ\!\!\rightarrow$, $\rightarrow\!\!\circ$, pos, root, $\mathfrak{L}$, $\mathfrak{O}$ which together provide access to all information in a data graph: about an edge, we know source, sink, edge position, and label, about a node, label, whether it is a root, its incoming and outgoing edges.

The remaining relations from Chapter 1 can be realized on top of these five basic relations by a set of CIQLog rules as shown in Figure 8. Rules for inverse and complement relations are defined as usual. Note, that $\complement R$ is only defined for relations R on nodes and/or edges (not on integers) both of which are finite domains that can be enumerated using $\circ\!\!\rightarrow$, $\rightarrow\!\!\circ$, and root. We only show rules for one case of deep equal, the remaining cases are similar, though considerably more involved if bijective mappings are required as we need to track the actual mapping to ensure bijectivity.

In Figure 8, we give rules for arbitrary label sets $\{\lambda_1, \dots, \lambda_n\}$ (line 1 ff.) and path lengths (i, j in line 15–18). Both path lengths and label sets are size limited by the label alphabet, resp. the number of edges in the data. However, we usually assume a much smaller number of such relations for any particular query task. E.g., in Chapters 3 to 5 each of the discussed translation uses no at most ternary label sets and no more than two or three path relations.

We use some abbreviations for common expressions over data graph relations in CIQLog heads: First, instead of $\mathsf{new}^{\mathsf{id}_1}_{\doteq}(x_1, \dots, x_n)$ we write simply $\mathsf{id}_1(x_1, \dots, x_n)$. Second, we use $RN(n, m, i)$ for $\circ\!\!\rightarrow(n, \mathsf{id}) \land \rightarrow\!\!\circ(m, \mathsf{id}) \land \mathfrak{L}(\mathsf{id}, RN) \land \mathsf{pos}(\mathsf{id}, i)$ where id is a new value invention term not used in the rest of the program. Third, we use $RN(n, m)$ for $\circ\!\!\rightarrow(n, \mathsf{id}) \land \rightarrow\!\!\circ(m, \mathsf{id}) \land \mathfrak{L}(\mathsf{id}, RN) \land \mathsf{pos}(\mathsf{id}, \mathsf{id}')$ where id and id' are new value invention terms not used in the rest of the program. In the latter case, we do not care about the actual edge position and thus allow an arbitrary one. It should only be used if $(n, RN) \notin \mathfrak{O}$.

2.5 Non-recursive CIQLog

Given these low numbers of actually used relations, the evaluation of CIQLog queries profits from precomputing of or providing special access operators to most or all of the derived relations discussed before. This allows us to define the final sub-language of CIQLog, non-recursive or single-rule CIQLog. A non-recursive CIQLog program consists in a single, non-recursive CIQLog rule (that may use any of the data graph relations). Thus its semantics can, given special access operators or precomputation of all data graph relations,

```
$\mathsf{Lab}^{\{\lambda_1,\dots,\lambda_n\}}(\mathtt{X})$
$\longleftarrow\ \mathfrak{L}(\mathtt{X},\lambda_1) \lor \mathfrak{L}(\mathtt{X},\lambda_2) \lor \dots \lor \mathfrak{L}(\mathtt{X},\lambda_n)$.

$\mathfrak{O}^{\{\lambda_1,\dots,\lambda_n\}}(\mathtt{X})$
$\longleftarrow\ \mathfrak{O}(\mathtt{X},\lambda_1) \lor \mathfrak{O}(\mathtt{X},\lambda_2) \lor \dots \lor \mathfrak{O}(\mathtt{X},\lambda_n)$.

$@^S$(E, $\textit{count}$(EPos')+1)
$\longleftarrow$ pos(E,EPos) $\land$ ○→(N,E) $\land$ ○→(N,E') $\land$ $\mathsf{Lab}^S$(E') $\land$ $\mathfrak{L}$(E', EL) $\land$
     $\mathfrak{O}$(N, EL) $\land$ pos(E',EPos') $\land$ EPos < EPos'.
$@^S$(N,EPos)
$\longleftarrow$ →○(N,E) $\land$ $@^S$(E,EPos).

$\mathsf{indeg}^S$(N,$\textit{count}$(E)) $\longleftarrow$ →○(N,E) $\land$ $\mathsf{Lab}^S$(E).
$\mathsf{outdeg}^S$(N,$\textit{count}$(E)) $\longleftarrow$ ○→(N,E) $\land$ $\mathsf{Lab}^S$(E).

$\mathsf{path}^S_{i,j}$(N,N')
$\longleftarrow$ $i < j$ $\land$ $\neg$(N $\doteq$ N') $\land$ ○→(N,$\mathtt{E}_1$) $\land$ $\mathsf{Lab}^S$($\mathtt{E}_1$) $\land$
     →○($\mathtt{N}_1$,$\mathtt{E}_1$) $\land$ $\mathsf{path}^S_{\max(0,i-1),j-1}$($\mathtt{N}_1$,N'))
$\mathsf{path}^S_{0,j}$(N,N) $\longleftarrow$ $0 \le j$.

$\prec^S$(E,E') $\longleftarrow$ $@^S$(E,EPos) $\land$ $@^S$(E',EPos+1).
$\prec^S_+$(E,E') $\longleftarrow$ $@^S$(E,EPos) $\land$ $@^S$(E',EPos') $\land$ EPos < EPos'.
$\prec^S_*$(E,E') $\longleftarrow$ $@^S$(E,EPos) $\land$ $@^S$(E',EPos') $\land$ EPos $\le$ EPos'.
$\lll^S$(N,N') $\longleftarrow$ →○(N,E) $\land$ →○(N',E') $\land$ E $\prec^S$ E'.
$\lll^S_+$(N,N') $\longleftarrow$ →○(N,E) $\land$ →○(N',E') $\land$ E $\prec^S_+$ E'.
$\lll^S_*$(N,N') $\longleftarrow$ →○(N,E) $\land$ →○(N',E') $\land$ E $\prec^S_*$ E'.

$\blacktriangleleft^S_+$(N,N') $\longleftarrow$ $\mathsf{path}^S_*$(N,M) $\land$ $\mathsf{path}^S_*$(N',M') $\land$ M $\lll^S_+$ M'.
$\blacktriangleleft^S_*$(N,N') $\longleftarrow$ $\mathsf{path}^S_*$(N,M) $\land$ $\mathsf{path}^S_*$(N',M') $\land$ M $\lll^S_*$ M'.
$\blacktriangleleft^S$(N,N') $\longleftarrow$ N $\blacktriangleleft^S_+$ N' $\land$ $\neg$(N $\blacktriangleleft^S_+$ M $\land$ M $\blacktriangleleft^S_+$ N' $\lor$ $\mathsf{path}^S_*$(M,N')).

$\cong$(X,X') $\longleftarrow$ $\mathfrak{L}$(X,L) $\land$ $\mathfrak{L}$(X',L).
$\doteq$(N,N) $\longleftarrow$ ○→(N,E) $\lor$ →○(N,E) $\lor$ root(N).
$\doteq$(E,E) $\longleftarrow$ ○→(N,E).
$=^S_@$(E,E') $\longleftarrow$ $@^S$(E,EPos) $\land$ $@^S$(E',EPos).

$\triangleq$(N,N') $\longleftarrow$ N $\doteq$ N'.
$\triangleq$(N,N')
$\longleftarrow$ N $\cong$ N' $\land$ $\neg$(○→(N,E) $\land$ →○(M,E) $\land$
     $\neg$(○→(N',E') $\land$ →○(M',E') $\land$ M $\triangleq$ M')).
```

Figure 8. CIQLog rules for intensional data graph relations

be specified without use of a fixpoint or similar recursion construct as shown in Table 5. Yet such programs are equivalent to CIQLog^{wR} programs comprised of the single rule of the original program and the rules from Figure 8 for the derived data graph relations. Note, that for non-recursive CIQLog the restriction to a single-rule has no effect on expressiveness due to disjunction in rule bodies and conditional construction in the head. Single-rule CIQLog considerably simplifies the correctness proofs in Chapters 3 to 5 as well as the transformation to CIQCAG.

We assume in the following that the head of a single-rule CIQLog program is not matched against its body. This can be achieved by adding a root node to the head of that rule with a label that is not matched by an added root node test in the body (and connections between that root node and all nodes and edges in the query). In the following, we omit these parts of a CIQLog rule when talking about non-recursive CIQLog.

Precomputing these relations is acceptable for unary or integer-valued relations such as Lab, @ or indeg. However, for node-node or edge-edge relations such as path relations, order relations or equivalence relations, the space cost may be prohibitive and thus specialized operators might be preferable.

Using the approaches outlined in the following sections, we obtain for all CIQLog relations (except deep equal) constant membership test at a space cost of only $\mathcal{O}(|D|) = \mathcal{O}(|N| + |E|)$. On tree data, this is obviously $\mathcal{O}(|N|)$.

2.5.1 Reachability in Data Graphs

Order and path relations can be seen as variants of reachability on the underlying base relations. For tree data, membership in closure relations can be tested in constant or almost constant time (e.g., using interval encodings [27] or other labeling schemes such as [74]). However, for graph data this is not so obvious. Fortunately, there has been considerable research on reachability or closure relations and their indexing in arbitrary graph data in recent years. Table 7 summarizes the most relevant approaches for our work. Theoretically, we can obtain constant time for the membership test if we store the full transitive closure matrix. However, for large graphs this is clearly infeasible. Therefore, two classes of approaches have been developed that allow with significantly lower space to obtain sub-linear time for membership test.

The first class are based on the idea of a 2-hop cover [23]: Instead of storing a full transitive closure, we allow that reachable nodes are reached via at most one other node (i.e., in two “hops”). More precisely, each node n is labeled with two connection sets, $in(n)$ and $out(n)$. $in(n)$ contains a set of nodes that can reach n, $out(n)$ a set of nodes that are reachable from n. Both sets are assigned in such a way, that a node m is reachable from n iff $out(n) \cup in(m) \neq \emptyset$. Unfortunately, computing the optimal 2-hop cover is NP-hard and even improved approximation algorithms [68] have still rather high complexity.

A different approach [3, 20, 73, 71] is to use interval encoding for labeling a tree core and treating the remaining non-tree edges separately. This allows for sublinear or even constant membership test, though constant membership test incurs lower but still considerable indexing cost, e.g., in Dual Labeling [73] where a full transitive closure over the non-tree edges is build. GRIPP [71] and SSPI [20] use a different trade-off by attaching additional interval labels to non-tree edges. This leads to linear index size and time at the cost of increased query time.

For a specialized reachability test operator in CIQLog we can choose any of the approaches. For the following, we assume constant time membership, since that is easily achieved on trees and feasible with

approach	characteristics	query time	index time	index size
Shortest path [62]	no index	$\mathcal{O}(n+e)$	–	–
Transitive closure	full reachability matrix	$\mathcal{O}(1)$	$\mathcal{O}(n^3)$	$\mathcal{O}(n^2)$
2-Hop [23]	2-hop cover[a]	$\mathcal{O}(\sqrt{e}) \leq \mathcal{O}(n)$	$\mathcal{O}(n^4)$	$\mathcal{O}(n \cdot \sqrt{e})$
HOPI [68]	2-hop cover, improved approximation algorithm	$\mathcal{O}(\sqrt{e}) \leq \mathcal{O}(n)$	$\mathcal{O}(n^3)$	$\mathcal{O}(n \cdot \sqrt{e})$
Graph labeling [3]	interval-based tree labeling and propagation of intervals of non-tree descendants.	$\mathcal{O}(n)$[b]	$\mathcal{O}(n^3)$	$\mathcal{O}(n^2)$[c]
SSPI [20]	interval-based tree labeling and recursive traversal of non-tree edges	$\mathcal{O}(e-n)$	$\mathcal{O}(n+e)$	$\mathcal{O}(n+e)$
Dual labeling [73]	interval-based tree labeling and transitive closure over non-tree edges	$\mathcal{O}(1)$[d]	$\mathcal{O}(n+e+e_g^3)$	$\mathcal{O}(n+e_g^2)$
GRIPP [71]	interval-based tree labeling plus additional interval labels for edges with incoming non-tree edges	$\mathcal{O}(e-n)$	$\mathcal{O}(n+e)$	$\mathcal{O}(n+e)$

[a] Index time for approximation algorithm in [23].

[b] More precisely, the number of intervals per node. E.g., in a bipartite graph this can be up to n, but in most (sparse) graphs this is likely considerably lower than n.

[c] More precisely, the total number of interval labels.

[d] [73] introduces also a variant of dual labeling with $\mathcal{O}(\log e_g)$ query time using a, in practical cases, considerably smaller index. However, worst case index size remains unchanged.

Table 7. Cost of Membership Test for Closure Relations. n, e: number of nodes, edges in the data, e_g: number of non-tree edges, i.e., if $T(D)$ is a spanning tree for D with edges $E_{T(D)}$, then $e_g = |E_D \setminus E_{T(D)}|$.

approaches such as Dual Labeling even for graphs.

2.5.2 Equivalence in Data Graphs

Most of the equivalence relations can, again, be easily computed from unary base relations such as $\mathfrak{L}$. However, this is not the case for deep equality. Indeed, the deep equal comes at a considerable higher cost than, e.g., label equality, in particular if the data is unordered. First, if the data is fully ordered and $L_O = L$ then deep equal is in $O(j(|E|))$ where E are the edges of the data graph. The same applies if either edge or node labels are keys, i.e., partition the child nodes of each node in the data in singleton sets. A straightforward parallel traversal of the two subgraphs suffices to test equivalence. In case of XML data, e.g., the data is ordered except for attributes. Attributes, however, are keys and thus do not need to be ordered for linear time complexity.

If the data is tree shaped but unordered, deep equal reduces to general tree isomorphism which can still be solved in linear time due to [41]. Moreover, for composition-free languages such as non-recursive

CIQLog$^{\text{wR}}$ or single-rule CIQLog where invented nodes are never queried and thus node invention can be seen as a post-processing step, deep equal does not add to the expressiveness of the query language if the data is tree shaped (cf. [47] on composition-free XQuery).

If the deep equal is not injective, but the data graph-shaped the complexity becomes polynomial as the problem essentially reduces to bisimulation [28]. However, on unordered *graphs* testing injective deep join of two nodes subsumes to graph isomorphism and thus is commonly believed to exhibit no polynomial algorithm (regardless of whether we consider the full subgraph or only the children or adjacent nodes for injectivity). However, even for this general case there exist reasonably fast algorithms, e.g., [56, 33].

Thus in presence of deep equal we either have to accept that membership tests for deep equal induce considerable, in most cases exponential additional cost, or we have to reconcile to precomputation of deep equal (which can be done once at a cost of $\mathcal{O}(|N|^2 \cdot |E|^{|E|})$ time). Obviously this relation can be precomputed in $O(n^2 \times n)$ for tree data. Using sub*tree* isomorphism algorithms, we can further reduce the time complexity by a linear factor. Whether there is a better algorithm for precomputing this relation in the general case, remains an open problem.

As a closing remark on deep equals over graphs, notice that the precomputation does *not* subsume to subgraph isomorphism. This is due to the fact, that deep equal considers only two nodes together with all their respective descendants. General subgraph isomorphism considers any node induced subgraph of the target graph (or any connected subgraph for connected subgraph isomorphism).

2.5.3 Examples

To illustrate, that the restriction to non-recursive CIQLog (with all data graph relations) still yields many interesting queries, we discuss a few examples of non-recursive CIQLog queries and, in a preview of Chapters 3 to 5, show counterparts in Xcerpt or XQuery.

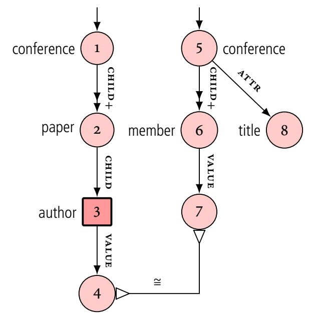
Let us first consider only the bodies of CIQLog rules. Figure 9 shows graphical representations for four such CIQLog rule bodies (or queries) against the data from Figure 2. This common, intuitive representation of queries as graphs is used throughout this paper: Query variables are represented as nodes with labels and values, as well as root nodes represented as in data graphs. Edges annotated with relation names represent atoms connecting query variables. Answer variables are marked by a darker rectangles whereas normal variables are indicated by lighter circles.

The first query (Figure (a)) selects paper authors that are also members (of the pc) at a named conference. Although the visual representation is already graph-, rather than tree-shaped, this query can still be expressed in XPath (using abbreviated syntax for closure axis), which, if we disregard functions and equality, only expresses tree-shaped queries.

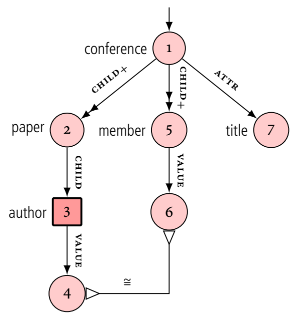
```
/conference//paper/author[text() =
    /conference[@title]/member/text()]
```

The following gives the textual representation of the query as a CIQLog rule body (assuming only v_3 occurs in the head and is thus the only answer variable):

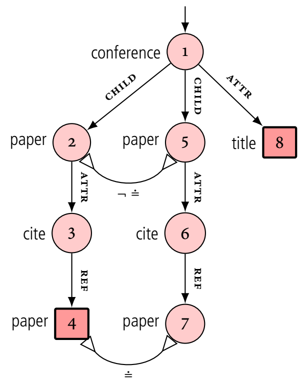
$$
\begin{aligned}
\longleftarrow\ & \mathfrak{L}(v_1,\text{conference}) \land \text{CHILD}_{+}(v_1,v_2) \land \\
&\quad \mathfrak{L}(v_2,\text{paper}) \land \text{CHILD}(v_2,v_3) \land \mathfrak{L}(v_3,\text{author}) \land \text{VALUE}(v_3,v_4) \land \\
& \mathfrak{L}(v_5,\text{conference}) \land \text{ATTR}(v_5,v_8) \land \mathfrak{L}(v_8,\text{title}) \land \text{CHILD}_{+}(v_5,v_6) \land
\end{aligned}
$$



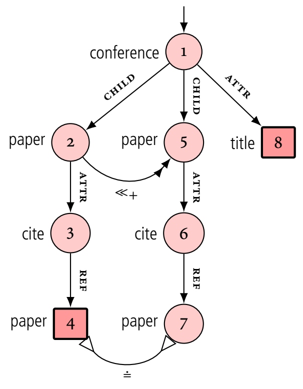
(a) Q_1^a (absolute value join)



(b) Q_1^b (relative value join)



(c) Q_2^a (identity anti-join)



(d) Q_2^b (sibling order)

Figure 9. Exemplary Query Graphs

```
        𝔏(v₆,member) ∧ VALUE(v₆, v₇) ∧ ≅(v₄, v₇).
```

Path relations are abbreviated here since the label sets are singleton in all cases.

On the other hand, the second query (Figure (a)), though still unary, already requires a language with multiple variables such as XQuery or Xcerpt. Intuitively, this is the case so as to be able to express that matched bindings for v_4 and v_6 are connected to the same binding for v_1. In XQuery, e.g., Q_1^b can be expressed as

```
for $c in /conference[@title], $a in $c//paper/author
where $c/member = $a
return $a
```

It selects an author only if he is also a (pc) member at the *same* conference. Where the difference between the two queries is quite large in XQuery, CIQLog uses nearly the same queries omitting the label test for the second conference node and connecting all its children to v_1:

```
 ⟵ 𝔏(v₁,conference) ∧ ATTR(v₁,v₇) ∧ 𝔏(v₇,title) ∧ CHILD₊(v₁,v₂) ∧
        𝔏(v₂,paper) ∧ CHILD(v₂,v₃) ∧ 𝔏(v₃,author) ∧ VALUE(v₃,v₄) ∧
        CHILD₊(v₁,v₅) ∧ 𝔏(v₅,member) ∧ VALUE(v₅, v₆) ∧ ≅(v₄, v₆).
```

Finally, the two queries Q_2^a and Q_2^b show how the same query intent (viz., *to select papers that are cited in two different papers at the same conference together with the name of that conference*) can be expressed differently: Q_2^a uses a negated equality condition between to ensure that its two different papers, Q_2^b uses order to ensure the same. However, the two queries are only equivalent if (a) CHILD edges outgoing from conferences are ordered and (b) if there are no multi-edges between conferences and papers (which is always the case if the data is tree shaped). Otherwise, $\ll_+$ is no longer necessarily anti-reflexive.

Both queries can be expressed only in languages with multiple variables. The following Xcerpt query term is equivalent (up to representation of ID/IDREF links to query Q_2^a:

```
conference{{
  paper{{ cite{{
    var Paper → idvar cited @ paper{{ }} }} }}
  paper{{ cite{{
                  idvar cited @ paper{{ }} }} }}
  name{{ var Name }} }}
```

Notice, that the anti-join is not expressed explicitly, but rather guaranteed by Xcerpt's injective mapping for sibling nodes (cf. [67] and 3). In CIQLog, we obtain:

```
 ⟵ 𝔏(v₁,conference) ∧ CHILD(v₁,v₂) ∧ CHILD(v₁,v₅) ∧
        𝔏(v₂,paper) ∧ ATTR(v₂,v₃) ∧ 𝔏(v₃,cite) ∧ REF(v₃,v₄) ∧ 𝔏(v₄,paper) ∧
        𝔏(v₅,paper) ∧ ATTR(v₅, v₆) ∧ 𝔏(v₆,cite) ∧ REF(v₆,v₇) ∧ 𝔏(v₇,paper) ∧
        ≐(v₄, v₇) ∧ ¬( ≅(v₂,v₅) ) ∧ ATTR(v₁,v₈) ∧ 𝔏(v₈,title).
```

Finally, Q_2^b can be expressed in XQuery as follows:

```
for $c in /conference, $n in $c/name, $p1 in $c/paper,
    $cited = $p1/@cite->paper
where (some $p2 in $c/paper satisfies $p1 << $p2 and
      (some $cited2 in $p2/@cite->paper
           satisfies $cited is $cited2))
return ($cited, $name)
```

For CIQLog, the textual form is mostly as Q_2^a, but uses in line 4 $\ll_+$ instead of a negated identity equivalence. In CIQLog, the child relation between v_1 and v_5 is implied by the order relation between v_2 and v_5 and may be omitted (this can be achieved in XQuery by using the optional following axis).

$$\longleftarrow\ \mathfrak{L}(v_1,\text{conference}) \land \text{CHILD}(v_1,v_2) \land \text{CHILD}(v_1,v_5) \land$$

2 $$\mathfrak{L}(v_2,\text{paper}) \land \text{ATTR}(v_2,v_3) \land \mathfrak{L}(v_3,\text{cite}) \land \text{REF}(v_3,v_4) \land \mathfrak{L}(v_4,\text{paper}) \land$$

$$\mathfrak{L}(v_5,\text{paper}) \land \text{ATTR}(v_5,\ v_6) \land \mathfrak{L}(v_6,\text{cite}) \land \text{REF}(v_6,v_7) \land \mathfrak{L}(v_7,\text{paper}) \land$$

4 $$\doteq(v_4,\ v_7) \land \ll_+(v_2,\ v_5) \land \text{ATTR}(v_1,v_8) \land \mathfrak{L}(v_8,\text{title}).$$

Chapter 3

Translating Xcerpt 2.0

3.1 Introduction

In [35] we have introduced Xcerpt 2.0 as an example of a versatile Web query language and argued that versatility is increasingly becoming an essential requirement for Web queries. With CIQLog we have the formal foundation to demonstrate how to translate queries in different languages and on different data formats into the same formal framework that can be evaluated using the CIQCAG algebra introduced in [34].

The following discussion of the translation of Xcerpt 2.0 starts with a fragment only, viz. non-recursive, single-rule Core Xcerpt which can be translated into a single non-recursive CIQLog rule (and thus be evaluated by a single CIQCAG expression). We first define that fragment and its syntax in Section 3.2, then illustrate its semantics along examples in Section 3.3. The semantics is fully aligned with full Xcerpt. The actual translation of non-recursive, single-rule Xcerpt is covered in Section 3.4 and concluded by an outlook towards the translation of full Xcerpt in Section 3.5.

3.2 Non-recursive, Single-Rule Core Xcerpt

We choose a fragment of the rule-based, Web and Semantic Web query language Xcerpt [67]. Before we characterize the language fragment, this chapter gives a brief recall of some of Xcerpt's most relevant features in the context of this translation. For a proper introduction please see [67].

An Xcerpt program consists of a finite set of Xcerpt *rules*. The rules of a program are used to derive new, or transform existing, data from existing data (i.e. the data being queried). *Construct rules* are used to produce intermediate results while *goal rules* form the output of programs.

While Xcerpt works directly on XML or RDF data, it has its own data format for modeling XML documents or RDF graphs, viz. Xcerpt *data terms*. For example, the XML snippet `<book><title>White Mughals</title></book>` corresponds to the data term `book [ title [ "White Mughals"] ]`. The data term syntax makes it easy to reference XML document structures in queries and extends XML slightly, most notably by allowing unordered data and making references first class citizens (thus moving from a tree to a proper graph data model).

For instance, in the following query the construct rule defines data about books and their authors which is then queried by the goal. Intuitively, the rules can be read as deductive rules (like in, say, Datalog): if the body (after **FROM**) holds, then the head (following **CONSTRUCT** or **GOAL**) holds. A rule with an empty body is interpreted as a fact, i.e., the head always holds.

```
GOAL
 authors [ var X ]
FROM
 book [[ author [ var X ] ]]
END

CONSTRUCT book [ title [ "White Mughals" ],
      author [ "William Dalrymple" ] ] END
```

Xcerpt *query terms* are used for querying data terms and intuitively describe patterns of data terms. Query terms are used with a pattern matching technique[1] to match data terms. Query terms can be configured to take partiality and/or ordering of the underlying data terms into account during matching (indicated by different types of brackets). Query terms may also contain (logic) variables. If so, successful matching with data terms results in variable bindings used by Xcerpt rules for deriving new data terms. Matching, for instance, against the XML snippet above the query term `book [ title [ var X ] ]` with results in the variable binding `{X/"White Mughals"}`. In addition to the query term types discussed in [66], we also consider non-injective ordered and unordered query terms indicated by three braces or brackets, respectively.

Construct terms are essentially data terms with variables. The variable binding produced via query terms in the body of a rule can be applied to the construct term in the head of the rule in order to derive new data terms. For the example above we obtain the data term `authors [ "William Dalrymple"]` as result.

Definition 3.1 (Non-recursive Single-rule Xcerpt)**.** Let P be an Xcerpt program. Then, P is a non-recursive, single-rule Xcerpt program, if it consists of a single **GOAL** rule and arbitrary many data terms.

In other words, there is a single rule whose body is evaluated a given set of data terms. If the body matches, the head is then evaluated given the bindings generated by the body.

3.2.1 Formal Syntax

In the following, we omit some of Xcerpt's more advanced features to allow for a translation that is reasonably compact and easy to follow. The most prominent omitted features are query terms involving **optional** or **except** and construct terms with **some**, **first**, declare blocks, order specifications, and conditions. Most of these limitations are purely for presentation reasons. This is, however, not the case for **except** which is not supported by non-recursive, single-rule CIQLog as it requires composition, i.e., first the sub-terms excluded by **except** are removed from a surrounding sub-graph binding, then that binding is used in the remainder of the query (e.g., for deep equality). This can be realized by multiple CIQLog rules, but not in non-recursive, single-rule CIQLog as defined in Chapter 2.4.

We call the resulting language fragment non-recursive, single-rule *Core* Xcerpt, abbreviated as Xcerpt$^{\text{core,NR,SR}}$ and specify its full grammar in Figure 10 (using common EBNF-like notation).

[1]Called *simulation unification*. For details of this technique, please refer to [66].

```
⟨rule⟩         ::=  ‘GOAL’ ⟨cterm⟩ ‘FROM’ ⟨query⟩ ‘END’
⟨cterm⟩        ::=  (⟨identifier⟩ ‘@’)? ⟨label⟩ ⟨hchildren⟩ | ‘"’ ⟨string⟩ ‘"’
                 |  ⟨reference⟩ | ⟨id-variable⟩ | ⟨variable⟩
                 |  ⟨grouping⟩
⟨grouping⟩     ::=  ‘all’ ⟨cterm⟩ ‘group-by’ ‘(’ ⟨variable⟩+ ‘)’
⟨hchildren⟩    ::=  ‘{’ ⟨cterm⟩* ‘}’  |  ‘[’ ⟨cterm⟩* ‘]’
⟨query⟩        ::=  ‘and’ ‘(’ ⟨query⟩‘,’? ⟨query⟩ ‘)’
                 |  ‘or’ ‘(’ ⟨query⟩‘,’? ⟨query⟩ ‘)’
                 |  ‘not’ ‘(’ ⟨query⟩ ‘)’
                 |  ⟨qterm⟩
⟨qterm⟩        ::=  ⟨term-id⟩? ⟨position⟩? ⟨label⟩ ⟨qchildren⟩ | ‘"’ ⟨string⟩ ‘"’
                 |  ⟨reference⟩
                 |  ⟨variable⟩ (‘→’⟨qterm⟩)?
                 |  (‘desc’ | ‘without’) ⟨qterm⟩
⟨term-id⟩      ::=  (⟨identifier⟩ | ⟨id-variable⟩) ‘@’
⟨reference⟩    ::=  ‘^’ ⟨identifier⟩
⟨position⟩     ::=  ‘position’ (⟨number⟩ | ⟨variable⟩)
⟨children⟩     ::=  ‘{’ ⟨term⟩* ‘}’  |  ‘[’ ⟨term⟩* ‘]’
                 |  ‘{{’ ⟨term⟩* ‘}}’  |  ‘[[’ ⟨term⟩* ‘]]’
                 |  ‘{{{’ ⟨term⟩* ‘}}}’  |  ‘[[[’ ⟨term⟩* ‘]]]’
⟨variable⟩     ::=  ‘var’ ⟨identifier⟩
⟨id-variable⟩  ::=  ‘idvar’ ⟨identifier⟩
⟨label⟩        ::=  ⟨variable⟩ | ⟨identifier⟩ | ‘*’
```

Figure 10. Syntax of non-recursive, single-rule Core Xcerpt

```
conference [
 title [ "Storage Media" ] date [ "44 B.C." ]
 p1 @ paper [ title [ "Wax Tablets" ]
   ^p2 author [ "Cicero" ]
 ]
 p2 @ paper [
   ^p1 author [ "Hirtius" ]
 ]
 pc[ member[ "Cicero" ] member[ "Atticus" ] ]
]
```

Figure 11. Xcerpt$^{\text{core,NR,SR}}$ dataterm on conferences and papers

The usual semantic restriction for Xcerpt rules apply, e.g., range restriction (each variable occurring in the head must also occur positively in each body disjunct where both **not** and **without** are considered as negative expressions); **without** may not occur in total query terms; an identity variable can only occur with idvar, a label variable only in label and term position, but not with →, idvar, or position, a position variable only with position; **without** and **position** may not occur for top-level terms; each referenced term identifier is defined somewhere; no term identifier is defined twice.

In the following, we assume a few additional restrictions to allow for a concise and easy to follow description of the translation: (1) We do not consider nested grouping lists as in

```
all a[var X, all (var Y, d)]
```

(2) We consider node injectivity instead of position injectivity for matching injective term specifications such as `a{{b, var X}}` or `a{b, var X}` where `b` and `var X` must be connected by edges with different edge positions to `a` in standard Xcerpt, where we consider that `var X` binds only to different nodes than `b`. (3) We consider `*` as abbreviation for the Xcerpt regular expression `/.*/` that indicates that the label of a query term may be arbitrary. (4) We omit XML specificities present in full Xcerpt terms for such as comments, processing-instructions, attributes, namespaces. (5) We normalize grouping expressions (`all` … `group-by`) such that all free variables in each grouping expression are listed in the `group-by` clause of that grouping. In full Xcerpt, this is not necessary but possible and yields a grouping expression with the same semantics.

Before we turn to a more detailed examination of the semantics of Xcerpt query and construct terms in the following section, we give a first intuition by the following two examples on a slight simplification of the data introduced in Section 1.5. Figure 11 shows a Core Xcerpt data term where namespaces and comments are removed and attributes are changed to elements, if compared to the data term from Section 1.5.

On that data term, the Xcerpt$^{\text{core,NR,SR}}$ rule in Figure 12 selects papers containing “Cicero” as author and “puts them in a shelf”.

To illustrate the difference between references and copies of query terms consider the final two rules and their query terms shown in Figure 13: the left query term returns all papers such that there is another paper with at least one (structurally) same author, the right hand returns only papers where the identical

```
GOAL
 shelf{ all var X group-by(var X) }
FROM
 conference{{
   var X → paper{{
     desc author{{
       "Cicero"
     }}
   }} }}
END
```

Figure 12. Xcerpt$^{\text{core,NR,SR}}$ rule to extract Cicero's papers to a shelf

```
GOAL
 shelf{ all var X group-by(var X) }
FROM
 conference{{
   var X → paper{{
     author{{
       var Y
   }} }}
   paper{{ author{{ var Y }} }} }}
END
```

```
GOAL
 shelf{ all var X group-by(var X) }
FROM
 conference{{
   var X → paper{{
     idvar A @ author{{
       var Y
   }} }}
   paper{{ idvar A @ author }} }}
END
```

Figure 13. Structural versus identity equivalence in Xcerpt

author term is used in both cases (rather than just a structurally equivalent one).

3.3 Xcerpt Semantics by Example

An Xcerpt term is essentially a labeled list of children. In addition to the label, we also record whether a term is *ordered* and, if it is a query term, if it is *total* or *partial injective* or *partial non-injective*. Recall, that a query term specifies a query by exemplifying (think QBE [75]) the shape of the matched data terms. However, to be effective, we leave out certain parts of that shape and focus only on the parts relevant to the query intent. Leaving out certain parts is achieved by various forms of incompleteness: regarding the structural relations by moving from child to descendant relations (indicated by the `desc` keyword), regarding how complete the list of given children is using the three latter properties above: If a query term is total, there may be no children of a matching query term in addition to the ones matched by each of the query term's children. If partial, there may be additional ones. If the query term is in addition injective, each of its children is mapped to a unique child of a matching data term. We denote ordered query terms with square brackets, otherwise we use curly braces. For total terms we use single such brackets, for partial injective double, for partial non-injective triple.

The data term on conferences in Figure 11 serves as an example for total terms since data (and construct) terms only contain this term type. Query terms also contain partial terms as evidenced by the query examples in Figures 12 and 13.

	Query term	Data terms
T1	`a{ }`	$\preceq$ `a{ }`; `a[ ]` $\not\preceq$ `b{ }`
T2	`a[ ]`	$\preceq$ `a[ ]` $\not\preceq$ `b{ }`; `a{ }`
T3	`a{ b }`	$\preceq$ `a{ b }` $\not\preceq$ `a{ b, b }`
T4	`a{ b, b }`	$\preceq$ `a{ b, b }` $\not\preceq$ `a{ b }`; `a{ b, b, b }`
P1	`a{{ b }}`	$\preceq$ `a{ b }`; `a{ c, b, d }`; `a{ b, b }` $\not\preceq$ `a{ }`;
P2	`a[[ b, c ]]`	$\preceq$ `a[ b, c ]`; `a[ d, b, e, c ]` $\not\preceq$ `a[ c, b ]`; `a{ b, c }`
I1	`a{{{ b, b }}}`	$\preceq$ `a[ b ]`; `a{ c, b, d }`; `a{ b, b }` $\not\preceq$ `a[ ]`; `a{ }`
I2	`a[[[ b, b, d ]]]`	$\preceq$ `a[ b, d ]`; `a[ c, b, d ]`; $\not\preceq$ `a[ d, b ]`; `a{ }`
D1	`a{ desc b }`	$\preceq$ `a[ b ]`; `a[ c{ b, e } ]`; $\not\preceq$ `a{ d, c{ b } }`;
D2	`a{ desc b, desc c }`	$\preceq$ `a[ b, e[ c ] ]`; $\not\preceq$ `a{ b, c, d }`; `a{ e[b, c] }`;
D3	`a{{ desc b, desc c }}`	$\preceq$ `a[ b, e[ c ] ]`; `a{ b, c, d }`; $\not\preceq$ `a{ e[b, c] }`;
D4	`a{{{ desc b, desc c }}}`	$\preceq$ `a[b,e[c]]`; `a{b,c,d}`; `a{e[b,c]}`;
W1	`a{{ b, without( c ), d }}`	$\preceq$ `a[ b, d ]`; $\not\preceq$ `a{ b, c, d }`; `a{ c, b, d }`;
W2	`a[[ b, without( c ), d ]]`	$\preceq$ `a[ b, d ]`; `a[ c, b, d ]`; $\not\preceq$ `a[ b, c, d ]`;

W3	`a[[ b, without( c, d ),`
	`e ]]`
		$\preceq$ `a[ b, e ]`; `a[ b, c, e ]`;
		$\not\preceq$ `a[ b, c, d, e ]`;

W4	`a[[ b, without( c ),`
	`without( d ), e ]]`
		$\preceq$ `a[ b, e ]`;
		$\not\preceq$ `a[ b, c, e ]`; `a[ b, c, d, e ]`;

Table 9. Query terms and matching data (; separates different data terms)

Matching query terms. Table 9 illustrates how these properties affect the matching of query terms against data terms by example. For space reasons, we omit in query terms empty double braces and in data term empty single braces, i.e., c reads `c{{ }}` in a query term and `c{ }` in a data term. We denote matching using $\preccurlyeq$ (simulation unification from in [66], but here we consider only data terms on the right hand), non matching with $\npreccurlyeq$.

The first examples T1–T4 illustrate matching of ordered and unordered total query terms. Note, that unordered query terms match against ordered data terms (since the use of the curly braces indicates only that we do not care about the order). In total query terms both terms have exactly the same number of children in all cases. This is what sets partial query terms (P1–P2, I1–I2) apart from total query terms. Here, we may have additional query terms in the data that are ignored. For partial non-injective query terms (I1–I2), two children of the query may even match to the same data term.

The remaining examples of Table 9 illustrate the two query term modifiers, `desc` and `without`. The former allows matching at any depth (cf. D1–D4). Totality and injectivity are still enforced between the children of a matching data term (observe the difference between D2, D3, and D4). The latter forbids the existence of a data term matching its enclosed query terms, cf. W1–W4. It may even take a list of query terms, in which case (W3) the query fails, if the entire list (and not just some subset of it) fails. This contrasts with the use of multiple `without`s as in W4.

	Query term	Data terms	Bindings
V1	`a{ var X }`	$\preccurlyeq$ `a[ b ]`; $\npreccurlyeq$ `a{ }`; `a[ b, c ]`	$\{X/\texttt{b}\}$
V2	`a{{ var X }}`	$\preccurlyeq$ `a[ b, c ]`; $\npreccurlyeq$ `a{ }`;	$\{X/\texttt{b}, X/\texttt{c}\}$
V3	`a{{ var X, var X }}`	$\preccurlyeq$ `a[ b, b ]`; `a{ c, b, b, d }` $\npreccurlyeq$ `a{ b, c }`; `a{ b }`	$\{X/\texttt{b}_1, X/\texttt{b}_2\}$
V4	`a{{{ var X, var X }}}`	$\preccurlyeq$ `a[ b, b ]`; `a{ c, b, b, d }` $\preccurlyeq$ `a{ b }` $\npreccurlyeq$ `a{ b, c }`;	$\{X/\texttt{b}_1, X/\texttt{b}_2\}$ $\{X/\texttt{b}\}$
V5	`a{ var X{ var X } }`	$\preccurlyeq$ `a[ b{ "b"} ]`; $\npreccurlyeq$ `a{ b, c }`;	$\{X/\text{“b”}\}$
V6	`a{ var X → c, var X }`	$\preccurlyeq$ `a[ c, c ]`; $\npreccurlyeq$ `a{ b, b }`;	$\{X/\texttt{b}\}$
V7	`a{ desc var X }`	$\preccurlyeq$ `a[ c{ b, e[ f ] } ]`; $\npreccurlyeq$ `a{ d, c{ b } }`;	$\{X/\texttt{c\{...\}}, X/\texttt{b}, X/\texttt{e[...]}, X/\texttt{f}\}$
V8	`a{{var X, without(var X)}}`	$\preccurlyeq$ `a[ b ]`; `a[ b, c ]`; $\npreccurlyeq$ `a{ b, b }`; `a[ b, b ]`;	$\{X/\texttt{b}\}$

Table 11. Query terms containing variables and their bindings

The last remaining feature of query terms are variables, the effect of which on term matching is illustrated in Table 11: Essentially, a variable matches any single term (or label, or position, or node, if so placed), but matches are recorded in the bindings of the query. If a variable occurs multiple times (V3),

	Construct term	Result data
C1	`a{ b, c }`	$\xrightarrow{\mathcal{E}}$ `a{ b, c }`
C2	`a{ id @ b, ^id }`	$\xrightarrow{\mathcal{E}}$ `a{ id’ @ b, ^id’ }`
C3	`a{ var X }`	$\xrightarrow{\mathcal{E}}$ a{ b$_1$ }
C4	`a{ var X, var Y }`	$\xrightarrow{\mathcal{E}}$ a{ b$_1$, c$_1$ }
G1	`a{ all var X group-by(var X)}`	$\xrightarrow{\mathcal{E}}$ a{ b$_1$, b$_2$, b$_3$[e, f] }
G2	`all a{ var X group-by(var X)}`	$\xrightarrow{\mathcal{E}}$ a{ b$_1$ }, a{ b$_2$ }, a{ b$_3$[e, f] }

G3	`all a{ all var X group-by(var X),`
	`var Y } group-by(var Y)`
		$\xrightarrow{\mathcal{E}}$ a{ b$_1$, b$_2$, b$_3$[e, f], c$_1$ },
		a{ b$_1$, b$_3$[e, f], c$_2$ }

G4	`all a{ } group-by(var Y)`	$\xrightarrow{\mathcal{E}}$ a{ }, a{ }

Table 13. Construct terms and their instantiation

the matched query terms must be structurally equivalent (deep equal, cf. Section 1.6.6). A variable may occur as a label (V5), in which case it is bound to the value of the label and can only match with other labels or character data (as the "b" in V5). A variable may occur as in a term restriction before → (V6), in which case the right hand query term restricts the matching bindings for X. Finally, it can be combined with `desc` and `without` with the expected result (V7, V8).

Instantiating construct terms. Once the body of a rule is matched against the input data, the bindings for the answer variables can be used to instantiate the construct term of an Xcerpt$^{\text{core,NR,SR}}$ rule. Again, we illustrate the semantics of construct terms along a number of examples, cf. Table 13, using the following binding tuples for answer variables X and Y:

$$\begin{aligned}\mathcal{B} = \{&\{\mathtt{X/b_1, Y/c_1}\}, \{\mathtt{X/b_2, Y/c_1}\}, \{\mathtt{X/b_3[e,\ f], Y/c_1}\},\\ &\{\mathtt{X/b_1, Y/c_2}\}, \{\mathtt{X/b_3[e,\ f], Y/c_2}\}\}\end{aligned}$$

If a construct term contains no variables (C1–C2), the only resulting data term has the exact same shape, possibly renaming local identifiers for references (C2). If it contains variables outside grouping expressions (C3, C4) these are instantiated by some of their bindings (we choose here the first binding). Grouping expressions iterate over the bindings of their grouping variables and instantiate their contained construct term once for each binding tuple of the grouping variables. The scope of the grouping expression defines, in this case which parts of the construct term are repeated (G1, G2). It is not necessary that the contained construct terms actually contain the occurrences of the grouping variables (G4), though that is usually the case. Employing nested grouping terms as in G3, we can create complex nestings of related bindings, here, e.g., for each binding of Y the corresponding bindings of X are grouped as siblings.

3.4 Translating Non-recursive Core Xcerpt

With the syntax and intuitive semantics of Xcerpt$^{\text{core,NR,SR}}$ established, we can turn to the actual translation. The translation is split in three parts, the translation of construct terms to CIQLog rule heads, the translation of query terms to CIQLog rule bodies, and the "glue", the translation of Xcerpt$^{\text{core,NR,SR}}$ rules to CIQLog rules. We start off with the translation of rules and a "grand example" that illustrates the principles of the translations. The details of construct and query term translation are discussed in the remainder of this section.

3.4.1 Rules

Recall, that a Xcerpt$^{\text{core,NR,SR}}$ program consists of a single Core Xcerpt **GOAL** rule. Such a rule is translated by the $\mathrm{tr}_{\mathrm{Xcerpt}}$ translation function: as follows:

$$\mathrm{tr}_{\mathrm{Xcerpt}}\langle \textbf{GOAL}\ \textit{head}\ \textbf{FROM}\ \textit{query}\ \textbf{END}\rangle = C \longleftarrow Q \quad \textbf{where}\ (\mathcal{E}, Q) = \mathrm{tq}\langle \textit{body}\rangle$$
$$C = \mathrm{tc}(\mathcal{E})\langle \textit{head}\rangle$$

It translates the body first, redirecting the resulting environment containing associations of Xcerpt (answer) variables to CIQLog variables to the translation of the head of the input rule. Finally, the translation of the body and head are combined into the translation of the full rule. The translation functions for head and body, tc and tq, are defined below in Sections 3.4.2 and Section 3.4.3. Here and in the following, we denote the set of common edge labels with $L = \{\text{CHILD}, \text{COMMENT}, \text{VALUE}\}$.

Examples. Before turning to the precise definitions of those translation functions, let us return to the example data and rule from Figures 11 and 12.

Xcerpt data terms can be translated to CIQLog (thus giving a formal definition for the mapping described in Section 1.5) by the translation function for Xcerpt rule heads using an empty environment as input. The environment can be empty since it is responsible only for passing Xcerpt to CIQLog variable mappings and data terms contain no Xcerpt variables.

The data term D in Figure 11 is translated by $\mathrm{tc}(\varnothing)\langle D\rangle$ into the following CIQLog rule:

```
root(id_1) ∧ 𝔏(id_1, conference) ∧ 𝔒^L(id_1) ∧ CHILD(id_1, id_2, order(⊤,1)) ∧ CHILD(id_1, id_3, order(⊤,2)) ∧ CHILD(id_1,
    id_4, order(⊤,3)) ∧ CHILD(id_1, id_5, order(⊤,4)) ∧ CHILD(id_1, id_6, order(⊤,5)) ∧
 𝔏(id_2, title) ∧ 𝔒^L(id_2) ∧ VALUE(id_2, id_21, order(⊤,1)) ∧ 𝔏(id_21, "Storage Media") ∧
 𝔏(id_3, date) ∧ 𝔒^L(id_3) ∧ VALUE(id_3, id_31, order(⊤,1)) ∧ 𝔏(id_31, "44 B.C.") ∧
 𝔏(id_4, paper) ∧ 𝔒^L(id_4) ∧ CHILD(id_4, id_41, order(⊤,1)) ∧ CHILD(id_4, id_5, order(⊤,2)) ∧ CHILD(id_4, id_42,
     order(⊤,3)) ∧
   𝔏(id_41, title) ∧ 𝔒^L(id_41) ∧ VALUE(id_41, id_411, order(⊤,1)) ∧ 𝔏(id_411, "Wax Tablets") ∧
   𝔏(id_42, author) ∧ 𝔒^L(id_42) ∧ VALUE(id_42, id_421, order(⊤,1)) ∧ 𝔏(id_421, "Cicero") ∧
 𝔏(id_5, paper) ∧ 𝔒^L(id_5) ∧ CHILD(id_5, id_4, order(⊤,1)) ∧ CHILD(id_5, id_51, order(⊤,2)) ∧
   𝔏(id_51, author) ∧ 𝔒^L(id_51) ∧ VALUE(id_51, id_511, order(⊤,1)) ∧ 𝔏(id_511, "Hirtius") ∧
 𝔏(id_6, pc) ∧ 𝔒^L(id_6) ∧ CHILD(id_6, id_61, order(⊤,1)) ∧ CHILD(id_6, id_62, order(⊤,2)) ∧
   𝔏(id_61, member) ∧ 𝔒^L(id_61) ∧ VALUE(id_61, id_611, order(⊤,1)) ∧ 𝔏(id_611, "Cicero") ∧
   𝔏(id_62, member) ∧ 𝔒^L(id_62) ∧ VALUE(id_62, id_621, order(⊤,1)) ∧ 𝔏(id_621, "Atticus") ⟵ true.
```

The relations are all ordered as the data term is deliberately similar to the XML fragment from Section 1.3 and thus contains only ordered terms. However, in general, Xcerpt data terms may also contain unordered terms.

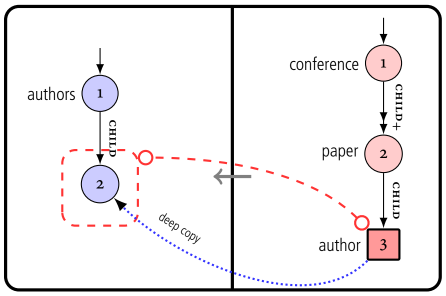


Figure 14. Resulting CIQLog rule

Here, and in the following we use two abbreviations for head formulas:

$$\texttt{child}(\mathsf{id}_1(\vec{x}_1),\mathsf{id}_2(\vec{x}_2),o)$$

where o is some order term and $\vec{x}_1, \vec{x}_2$ are variable lists, abstracts the edge construction necessary in CIQLog and thus is an abbreviation for (with id_N a new identifier):

$$\texttt{source}(\mathsf{id}_1(\vec{x}_1),\mathsf{id}_N(\vec{x}_2)),,\ \texttt{sink}(\mathsf{id}_2(\vec{x}_2),\mathsf{id}_N(\vec{x}_2)),,\ \underline{\texttt{position}}(\mathsf{id}_N(\vec{x}_2),\texttt{order}(\top,0))$$

In the same way, $\texttt{child}(\mathsf{id}_1(\vec{x}_1),\mathsf{id}_2(\vec{x}_2))$ is an abbreviation for ((with $\mathsf{id}_N, \mathsf{id}_M$ new identifiers):

$$\texttt{source}(\mathsf{id}_1(\vec{x}_1),\mathsf{id}_N(\vec{x}_2)),,\ \texttt{sink}(\mathsf{id}_2(\vec{x}_2),\mathsf{id}_N(\vec{x}_2)),,\ \underline{\texttt{position}}(\mathsf{id}_N(\vec{x}_2),\mathsf{id}_M(\vec{x}_2))$$

To illustrate the full translation, first consider the following very simple Xcerpt rule selecting all authors of papers and grouping them under a new root authors:

```
GOAL
  authors{ all var X group-by(var X) }
FROM
  conference{{ desc paper{{ var X → author }} }}
END
```

If we translate this Xcerpt rule to CIQLog we obtain the query visualized in Figure 14 (using the visualization from Chapter 2 and additionally depicting the scope of a grouping variable by red rectangles ⬚ as well as query variables used in the head by directed connections ┈►).

$$\texttt{root}(\mathsf{id}_1) \wedge \mathfrak{L}(\mathsf{id}_1,\ \texttt{authors}) \wedge \textsc{child}(\mathsf{id}_1,\ \mathsf{id}_2(v_3)) \wedge \textsf{deep-copy}(v_3,\ \mathsf{id}_2(v_3))$$
$$\longleftarrow \texttt{root}(\texttt{v}_1) \wedge \mathfrak{L}(\texttt{v}_1,\ \texttt{conference}) \wedge \textsc{child}_{+}(\texttt{v}_1,\ \texttt{v}_2) \wedge \mathfrak{L}(\texttt{v}_2,\ \texttt{paper}) \wedge \textsc{child}(\texttt{v}_2,\ \texttt{v}_3) \wedge \mathfrak{L}(\texttt{v}_3,\ \texttt{author}).$$

Finally, we reconsider the example rule from Figure 12, an Xcerpt rule querying that data and selecting all papers with author "Cicero" and "puts them on a shelf". Applying $\mathsf{tr}_{\mathsf{Xcerpt}}$ to that rule yields the CIQLog program depicted in Figure 15 and given in textual form in the following:

$$\texttt{root}(\mathsf{id}_1) \wedge \mathfrak{L}(\mathsf{id}_1,\ \texttt{shelf}) \wedge \textsc{child}(\mathsf{id}_1,\ \mathsf{id}_2(v_2)) \wedge \textsf{deep-copy}(v_2,\ \mathsf{id}_2(v_2))$$
$$\longleftarrow \texttt{root}(\texttt{v}_1) \wedge \mathfrak{L}(\texttt{v}_1,\ \texttt{conference}) \wedge \textsc{child}(\texttt{v}_1,\ \texttt{v}_2) \wedge \mathfrak{L}(\texttt{v}_2,\ \texttt{paper}) \wedge \textsc{child}_{+}(\texttt{v}_2,\ \texttt{v}_3) \wedge \mathfrak{L}(\texttt{v}_3,\ \texttt{author}) \wedge \textsc{value}(\texttt{v}_3,\ \texttt{v}_4),\ \mathfrak{L}(\texttt{v}_4,\ \texttt{"Cicero"}).$$

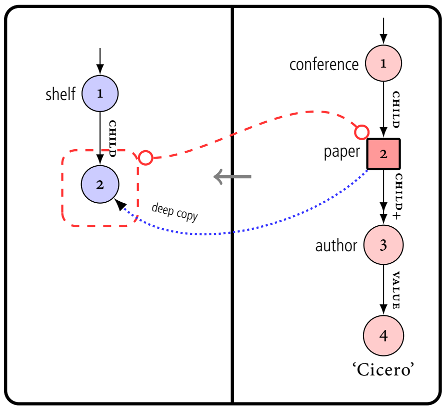


Figure 15. CIQLog rule for Xcerpt program from Figure 12

3.4.2 Construct Terms

With the intuition from the above examples, the translation of construct terms using tc can be discussed in more detail. The full specification of tc and its helper functions tc_{term}, for translating actual construct terms, and tc_{label}, for translating labels of construct terms, is given in Table 16. We normalize construct terms such that there are no variables outside (**all**) grouping terms. If there are such variables we wrap an **all** around the entire construct term with the all variables outside any other **all** as grouping variables. For instance, `a{var X}` becomes `all a{var X} group-by(var X)`.

tc adds a root atom and then delegates the translation to tc_{term}. This allows us to translate all query terms in the same way, yet yields the necessary root atom for top-level query terms. tc_{term} is called with the environment $\mathcal{E}$ as only parameter which is passed to tc by $\text{tr}_{\text{Xcerpt}}$ and contains the mappings form Xcerpt variables to CIQLog query variables in the translation of the rule body. We write $\mathcal{E} + (\mathtt{X}, v)$ to add a mapping from the Xcerpt variable X to the CIQLog variable v into $\mathcal{E}$. In addition to those mappings, we let the environment contain the current grouping (or iteration) variables, a sequence accessed as $\mathcal{E}.\text{iter}$. We use list concatenation ($\circ$) to append variables to $\mathcal{E}.\text{iter}$. tc_{term} returns a pair (C, v) where C is the CIQLog formula resulting from the translation of the passed Xcerpt expression and v the variable associated with the top-level node in the translation of the expression.

For each kind of construct term, there is a matching rule for tc_{term} in Table 16. The translation of most cases is fairly straightforward. The most involved cases are the structure terms (the first two cases) where we distinguish ordered and unordered terms. For both, we translate the child terms and connect their top-level variables to the variable of the current term by either CHILD or VALUE. The label is translated using the helper function tc_{label} (which distinguishes between the translation of plain labels and of variables). Strings are translated like terms with empty, unordered term lists (case 3); references (like the term identifier part of a structured term) by retrieving an existing CIQLog variable for the Xcerpt identifier *tid* or creating a new one and storing that mapping in the environment $\mathcal{E}$ (case 4). Standard variables are translated using deep-copy, id-variables by simply retrieving the mapped query variable (recall that rules are range restricted and thus such a binding always exists), case 5–6. Finally, grouping terms are

functionterm	= clQLog **expression**
$\text{tc}(\mathcal{E})\langle \textit{cterm} \rangle$	$= \texttt{root}(v) \wedge C$ **where** $(\mathcal{E}', C, v) = \text{tc}_{\textit{term}}(\mathcal{E})\langle \textit{cterm} \rangle$
$\text{tc}_{\textit{term}}(\mathcal{E})\langle \textit{tid} \mathbin{@} \textit{label}\{t_1, \ldots, t_k\} \rangle$	$= (\mathcal{E}_k, r_1(v, n_1) \wedge \ldots \wedge r_k(v, n_k) \wedge L \wedge C_1 \wedge \ldots \wedge C_k, v)$ **where** $v = \mathcal{E}(\textit{tid})$ if defined, otherwise $v = \text{id}(\mathcal{E}.\text{iter})$ with id new identifier $L = \text{tc}_{\textit{label}}(\mathcal{E}, v)\langle \textit{label} \rangle$ $r_i = \text{VALUE}$ if t_i is "*string*", $r_i = \text{CHILD}$ otherwise $\mathcal{E}_0 = \mathcal{E} +\!\!+ (\textit{tid}, v) \qquad (\mathcal{E}_i, C_i, n_i) = \text{tc}_{\textit{term}}(\mathcal{E}_{i-1})\langle t_i \rangle$
$\text{tc}_{\textit{term}}(\mathcal{E})\langle \textit{tid} \mathbin{@} \textit{label}[t_1, \ldots, t_k] \rangle$	$= (\mathcal{E}_k, r_1(v, n_1, \text{order}(\top, 1, \mathcal{E}.\text{iter})) \wedge \ldots \wedge r_k(v, n_k, \text{order}(\top, k, \mathcal{E}.\text{iter})) \wedge$ $\mathfrak{O}(v, \text{CHILD}) \wedge L \wedge C_1 \wedge \ldots \wedge C_k, v)$ **where** $v = \mathcal{E}(\textit{tid})$ if defined, otherwise $v = \text{id}(\mathcal{E}.\text{iter})$ with id new identifier $r_i = \text{VALUE}$ if t_i is "*string*", $r_i = \text{CHILD}$ otherwise $L = \text{tc}_{\textit{label}}(\mathcal{E}, v)\langle \textit{label} \rangle$ $\mathcal{E}_0 = \mathcal{E} +\!\!+ (\textit{tid}, v) \qquad (\mathcal{E}_i, C_i, n_i) = \text{tc}_{\textit{term}}(\mathcal{E}_{i-1})\langle t_i \rangle$
$\text{tc}_{\textit{term}}(\mathcal{E})\langle \text{"}\textit{string}\text{"} \rangle$	$= (\mathcal{E}, L, v)$ **where** $v = \text{id}(\mathcal{E}.\text{iter})$ with id new identifier $L = \text{tc}_{\textit{label}}(\mathcal{E}, v)\langle \textit{string} \rangle$
$\text{tc}_{\textit{term}}(\mathcal{E})\langle \hat{}\,\textit{tid} \rangle$	$= (\mathcal{E} +\!\!+ (\textit{tid}, v), \top, v), v)$ **where** $v = \mathcal{E}(\textit{tid})$ if defined, otherwise $v = \text{id}(\mathcal{E}.\text{iter})$ with id new identifier
$\text{tc}_{\textit{term}}(\mathcal{E})\langle \texttt{var X} \rangle$	$= (\mathcal{E}, \text{deep-copy}(\mathcal{E}(X), v), v)$ **where** $v = \text{id}(\mathcal{E}.\text{iter})$ and id new identifier
$\text{tc}_{\textit{term}}(\mathcal{E})\langle \texttt{idvar X} \rangle$	$= (\mathcal{E}, \top, \mathcal{E}(\text{X}))$
$\text{tc}_{\textit{term}}(\mathcal{E})\langle \texttt{all}\ t\ \texttt{group-by}(X_1, \ldots, X_n) \rangle$	$= \text{tc}_{\textit{term}}(\mathcal{E}')\langle t \rangle$ **where** $\mathcal{E}' = \mathcal{E}$ with $\mathcal{E}'.\text{iter} = \mathcal{E}.\text{iter} \circ [\mathcal{E}(X_1), \ldots, \mathcal{E}(X_n)]$
$\text{tc}_{\textit{label}}(\mathcal{E}, v)\langle \textit{label} \rangle$	$= \mathfrak{L}(v, \textit{label})$
$\text{tc}_{\textit{label}}(\mathcal{E}, v)\langle \texttt{var X} \rangle$	$= \cong (v, \mathcal{E}(\text{X}))$

Table 16. Translating Xcerpt construct terms

translated by adding the grouping variables to the sequence of iteration variables used for the translation of all contained construct terms, case 7.

Examples. We conclude the illustration of the translation function for construct terms by a collection of construct terms from Table 13 together with their clQLog translation. Table 18 shows the translation to clQLog for some of these construct terms. For convenience, we use X and Y to denote the query variables mapped to X and Y in a given environment $\mathcal{E}$, i.e., $\mathcal{E}(X)$ and $\mathcal{E}(Y)$. Recall, that the result is an expression containing some query variables that are replaced with their bindings for each binding tuple in turn.

	Construct term	CIQLog expression
C1	`a{ b, c }`	$\xrightarrow{\text{tc}}$ $\texttt{root}(\text{id}_1) \wedge \mathfrak{L}(\text{id}_1,\texttt{a}) \wedge \textsc{child}(\text{id}_1,\text{id}_2) \wedge \mathfrak{L}(\text{id}_2,\texttt{b}) \wedge \textsc{child}(\text{id}_1,\text{id}_3) \wedge \mathfrak{L}(\text{id}_3,\texttt{c})$
C2	`a[ id @ b, ^id ]`	$\xrightarrow{\text{tc}}$ $\texttt{root}(\text{id}_1) \wedge \mathfrak{L}(\text{id}_1,\texttt{a}) \wedge \mathfrak{O}^L(\text{id}_1)$ $\textsc{child}(\text{id}_1,\text{id}_2,\texttt{order}(\top,1)) \wedge \mathfrak{L}(\text{id}_2,\texttt{b}) \wedge$ $\textsc{child}(\text{id}_1,\text{id}_2,\texttt{order}(\top,2))$
C3	`a{ var X }`	$\xrightarrow{\text{tc}}$ normalized to G2.
G1	`a{ all var X group-by(var X)}`	$\xrightarrow{\text{tc}}$ $\texttt{root}(\text{id}_1) \wedge \mathfrak{L}(\text{id}_1,\texttt{a}) \wedge \textsc{child}(\text{id}_1,\text{id}_2(X)) \wedge$ $\text{deep-copy}(X,\ \text{id}_2(X))$
G2	`all a{ var X group-by(var X)}`	$\xrightarrow{\text{tc}}$ $\texttt{root}(\text{id}_1(X)) \wedge \mathfrak{L}(\text{id}_1(X),\texttt{a}) \wedge$ $\textsc{child}(\text{id}_1(X),\text{id}_2(X)) \wedge \text{deep-copy}(X,\ \text{id}_2(X))$
G3	`all a{ all var X group-by(var X), var Y } group-by(var Y)`	$\xrightarrow{\text{tc}}$ $\texttt{root}(\text{id}_1(Y)) \wedge \mathfrak{L}(\text{id}_1(Y),\texttt{a}) \wedge$ $\textsc{child}(\text{id}_1(Y),\text{id}_2(Y,\ X)) \wedge \text{deep-copy}(X,\ \text{id}_2(Y,\ X)) \wedge$ $\textsc{child}(\text{id}_1(Y),\text{id}_3(Y)) \wedge \text{deep-copy}(Y,\ \text{id}_3(Y))$
G4	`all a{ } group-by(var Y)`	$\xrightarrow{\text{tc}}$ $\texttt{root}(\text{id}_1(Y)) \wedge \mathfrak{L}(\text{id}_1(Y),\texttt{a})$

Table 18. Construct terms and their CIQLog translations

3.4.3 Queries and Query Terms

Bodies of Xcerpt rules are translated using tq. Again, we us an environment $\mathcal{E}$ to record already established mappings between Xcerpt and CIQLog variables. For queries we do not record iteration or grouping variables as there are no grouping expression in queries. The environment additionally contains isLabel(X) and isTerm(X, v) terms for Xcerpt variables X and CIQLog variables v. The former indicates that X is known to have occurred as a label variable. The latter that X has occurred in term position and is represented by y in the CIQLog expression. These are used to establish proper variable occurrences in label and term position. As before, we use $+\!\!+$ to add these terms to a given environment $\mathcal{E}$. We do use an additional helper structure for the translation of query terms, viz. $\mathcal{V} = (v_\uparrow, V_\leftarrow, V_\rightarrow, r_\uparrow, r_\leftarrow, r_\rightarrow)$, and denote each component by $\mathcal{V}.V_\leftarrow$, etc. and with $()$ the empty $\mathcal{V}$. $\mathcal{V}$ holds the CIQLog variable for the parent term of the term to be translated ($v_\uparrow$), the variables for left and right siblings of that term ($V_\leftarrow$ and $V_\rightarrow$, respectively) and the relations to these variables ($r_\uparrow$ for parent, $r_\leftarrow$ for left siblings, $r_\rightarrow$ for right siblings).

An Xcerpt rule body is first translated using tq which takes care of all top-level disjunction, conjunction, or negations. Note, that for disjunctions and conjunctions we propagate the environment return by the translation of the first operand ($\mathcal{E}_1$ in case 2 and 3) to the translation of the second operand. Thus Xcerpt variables occurring in both disjuncts are mapped to the same CIQLog variable. This assumes that, as usual, non-answer variables are standardized apart for each disjunct. After translating any top-level

functionterm	= CIQLog **expression**
$\mathsf{tq}\langle query\rangle$	$= \mathsf{tq}(\varnothing)\langle query\rangle$
$\mathsf{tq}(\mathcal{E})\langle \texttt{and(}t_1, t_2\texttt{)}\rangle$	$= (\mathcal{E}_2, (Q_1 \wedge Q_2))$ **where** $(\mathcal{E}_1, Q) = \mathsf{tq}(\mathcal{E})\langle t_1\rangle \quad (\mathcal{E}_2, Q) = \mathsf{tq}(\mathcal{E}_1)\langle t_2\rangle$
$\mathsf{tq}(\mathcal{E})\langle \texttt{or(}t_1, t_2\texttt{)}\rangle$	$= (\mathcal{E}_2, (Q_1 \vee Q_2))$ **where** $(\mathcal{E}_1, Q) = \mathsf{tq}(\mathcal{E})\langle t_1\rangle \quad (\mathcal{E}_2, Q) = \mathsf{tq}(\mathcal{E}_1)\langle t_2\rangle$
$\mathsf{tq}(\mathcal{E})\langle \texttt{not(}t\texttt{)}\rangle$	$= (\mathcal{E}', \neg(Q))$ **where** $(\mathcal{E}', Q) = \mathsf{tq}(\mathcal{E})\langle t\rangle$
$\mathsf{tq}(\mathcal{E})\langle qterm\rangle$	$= (\mathcal{E}', \texttt{root}(r) \wedge Q)$ **where** $(\mathcal{E}', r, Q) = \mathsf{tq}_{term}(\mathcal{E}, \bot, ())\langle qterm\rangle$
$\mathsf{tq}_{term}(\mathcal{E}, v, \mathcal{V})\langle tid \mathbin{@} t\rangle$	$= (\mathcal{E}', \{v\}, Q)$ **where** $v = \mathcal{E}(tid)$ if defined otherwise, if $v = \bot$, v new variable $(\mathcal{E}', V, Q) = \mathsf{tq}_{term}(\mathcal{E} \mathbin{+\!\!+} (tid, v), v, \mathcal{V})\langle t\rangle$
$\mathsf{tq}_{term}(\mathcal{E}, v, \mathcal{V})\langle \texttt{idvar } X \mathbin{@} t\rangle$	$= (\mathcal{E}'', \{v\}, E \wedge Q)$ **where** $v = \mathcal{E}(tid)$ if defined otherwise, if $v = \bot$, v new variable $(\mathcal{E}', E) = \mathsf{tq}_{var}(\mathcal{E}, v, \doteq)\langle X\rangle$ $(\mathcal{E}'', V, Q) = \mathsf{tq}_{term}(\mathcal{E}', v, \mathcal{V})\langle t\rangle$
$\mathsf{tq}_{term}(\mathcal{E}, v, \mathcal{V})\langle \texttt{position } num \mathbin{@} t\rangle$	$= (\mathcal{E}', \{v\}, @^{L}(v, num) \wedge Q$ **where** if $v = \bot$, v new variable $(\mathcal{E}', V, Q) = \mathsf{tq}_{term}(\mathcal{E}, v, \mathcal{V})\langle t\rangle$
$\mathsf{tq}_{term}(\mathcal{E}, v, \mathcal{V})\langle \texttt{position var } X \mathbin{@} t\rangle$	$= (\mathcal{E}'', \{v\}, E \wedge Q$ **where** if $v = \bot$, v new variable $(\mathcal{E}', E) = \mathsf{tq}_{var}(\mathcal{E}, v, =_{@})\langle X\rangle$ $(\mathcal{E}'', V, Q) = \mathsf{tq}_{term}(\mathcal{E}', v, \mathcal{V})\langle t\rangle$
$\mathsf{tq}_{term}(\mathcal{E}, v, \mathcal{V})\langle \texttt{var } X \rightarrow t\rangle$	$= (\mathcal{E}''', \{v\}, E' \wedge Q$ **where** if $v = \bot$, v new variable $(\mathcal{E}', E) = \mathsf{tq}_{var}(\mathcal{E}, v, \circeq)\langle X\rangle$ if $\mathsf{isLabel}(\texttt{X}) \in \mathcal{E}$, $E' = E \wedge \mathsf{outdeg}(v, 0)$ otherwise $E' = E$ and $\mathcal{E}'' = \mathcal{E}' \mathbin{+\!\!+} \{\mathsf{isTerm}(\texttt{X}, v)\}$ $(\mathcal{E}''', V, Q) = \mathsf{tq}_{term}(\mathcal{E}'', \mathcal{V})\langle t\rangle$
$\mathsf{tq}_{term}(\mathcal{E}, v, \mathcal{V})\langle \texttt{desc } t\rangle$	$= (\mathcal{E}'', \{v\}, F_{struct} \wedge Q$ **where** if $v = \bot$, v new variable $F_{struct} = \mathsf{tq}_{struct}(v, \mathcal{V})$ $(\mathcal{E}'', V, Q) = \mathsf{tq}_{term}(\mathcal{E}, \bot, (v, \varnothing, \varnothing, \mathsf{path}^{L}_{*}, \top, \top))\langle t\rangle$
$\mathsf{tq}_{term}(\mathcal{E}, v, \mathcal{V})\langle label\ children\rangle$	$= (\mathcal{E}_2, v, F_1 \wedge F_2 \wedge F_{struct})$ **where** if $v = \bot$, v new variable $(\mathcal{E}_1, F_1) = \mathsf{tq}_{label}(\mathcal{E}, v)\langle label\rangle$ $(\mathcal{E}_2, F_2) = \mathsf{tq}_{child}(\mathcal{E}_1, v)\langle children\rangle$ $F_{struct} = \mathsf{tq}_{struct}(v, \mathcal{V})$
$\mathsf{tq}_{term}(\mathcal{E}, v, \mathcal{V})\langle \texttt{"}string\texttt{"}\rangle$	$= (\mathcal{E}', v, F \wedge F_{struct})$ **where** if $v = \bot$, v new variable $(\mathcal{E}', F) = \mathsf{tq}_{label}(\mathcal{E}, v)\langle string\rangle$ $F_{struct} = \mathsf{tq}_{struct}(v, \mathcal{V})$
$\mathsf{tq}_{term}(\mathcal{E}, v, \mathcal{V})\langle \wedge tid\rangle$	$= (\mathcal{E} \mathbin{+\!\!+} (tid, v), v, F_{struct})$ **where** $v = \mathcal{E}(tid)$ if defined, otherwise v new variable $F_{struct} = \mathsf{tq}_{struct}(v, \mathcal{V})$
$\mathsf{tq}_{term}(\mathcal{E}, v, \mathcal{V})\langle \texttt{var } X\rangle$	$= (\mathcal{E}', v, E \wedge F_{struct})$ **where** if $v = \bot$, v new variable $(\mathcal{E}', E) = \mathsf{tq}_{var}(\mathcal{E}, v, \circeq)\langle X\rangle$ $F_{struct} = \mathsf{tq}_{struct}(v, \mathcal{V})$
$\mathsf{tq}_{term}(\mathcal{E}, v, \mathcal{V})\langle \texttt{without } tlist\rangle$	$= \mathsf{tq}_{tlist}(\mathcal{E}, \mathcal{V})\langle tlist\rangle$

Table 20. Translating Xcerpt query terms: queries and query terms

expressions, we turn to the evaluation of the top-level query terms which we root and translate using tq_{term} (case 5 of $\mathsf{tr}_{\mathsf{Xcerpt}}$). tq_{term} carries, in addition to the Xcerpt term to be translated, three parameters: the environment $\mathcal{E}$, a CIQLog variable for the current term (which may be $\perp$ indicating that no variable has been allocated for the term yet), and the context variables $\mathcal{V}$. Initially, there is no CIQLog variable for the current term yet (and thus the second parameter is $\perp$) and also no context variables $\mathcal{V}$ as a top-level Xcerpt term has no parent or sibling terms.

As for tc_{term}, tq_{term} is specified by one case for each of the possible query terms. Cases 1–6 cover "prefixes" of full query terms, e.g., term identifiers, term restrictions, position specifications, or descendant modifiers. Cases 7–10 cover the four basic query term kinds, structured (case 7), character data (case 8), reference (case 9), and variable (case 10). Finally, case 11 covers `without` with may contain an entire term list and is thus a bit of the "odd man out" in the translation of query terms.

For tq_{term}, we use five helper functions depicted in Table 22. **(1)** Bottom up, tq_{struct} creates a CIQLog expression for all the relations in a given $\mathcal{V}$ with a given current variable v_{curr}. It is used for the translation of each of the base term kinds (case 7–10 of tq_{term}). **(2)** tq_{var} turns a Xcerpt variable into a CIQLog variable using the specified equivalence relation eq. If the Xcerpt variable is already defined in the current environment the stored CIQLog variable is retrieved and an equivalence atom is emitted, otherwise we simply record a new mapping from the given Xcerpt `X` variable to the CIQLog variable v. A special treatment of label variables as in the translation of construct terms is provided. **(3)** tq_{label} translates just the label of a structured term: if it is the wildcard $*$, no relation is added (any node fulfills that restriction), if it is a proper label a corresponding label relation is omitted, if it is variable a similar translation as for tq_{var} takes place but we mark the variable by $\mathsf{isLabel}$ in the environment to ensure that other occurrences of that variable are connected using $\cong$ and no other equivalence relation. **(4)** tq_{child} translates child lists of a structure term selected by the CIQLog variable $v_{\uparrow}$. Basically, it distinguishes the six term list types (total, partial injective, partial non-injective, each combined with order or unorder) and calls for each tq_{tlist} with different parameters. The relation to the parent variable is always path^{L} where $L = \{\text{CHILD}, \text{COMMENT}, \text{VALUE}\}$ as stated above. But between the siblings, the relations vary: In the unordered, partial, non-injective case, there are no such relations at all (indicated by $\top$), in the case of partial, but injective terms we add complemented identity join, in the case of total terms we add a limitation on the out-degree of the parent (case 3). When combined with order, we obtain transitive-reflexive sibling order for partial, non-injective, transitive sibling order for partial, injective, and, again, an additional out-degree constraint for the total case (cases 4–6). **(5)** The actual formula of the relations between the variables is defined in tq_{tlist} which is called whenever translating lists of query terms (i.e., in tq_{child} and in the without case of tq_{term}). tq_{tlist} does not merely translate the given lists in the given order, but delays the translation of negative sub-terms until all positive ones are translated. This is necessary to allow the relations between negative and positive sub-terms (even ones after the negative sub-term) to be contained in the scope of the $\neg$. Otherwise, we demand the existence of such a relation rather than demand its non-existence. Nevertheless we ensure that all relations are enforced by gathering the CIQLog variables returned by the translation of all positive sub-terms to the right of a negative one in that sub-terms V_i^{right}.

To illustrate tq_{tlist} consider the Xcept query term

```
a[[ b, without( c, without( d ) ), without( e ), f ]]
```

It is necessary to ensure that in the resulting CIQLog expression all references to a CIQLog variable for the translation of, e.g., `c` are within a negation. This includes sibling relations to, e.g., the translation of `f`.

Otherwise we would, falsely, require that there has to be a (following) sibling relation to `f`-labeled node, instead of requiring that there is no `c` with such a relation.

Returning to tq_{term} in Table 20, we see that all the "prefixes" of a query term (case 1–6) are translated along the same scheme: if necessary, we take a new variable and add some relations on that variable to the output of the translation of the contained query term. For the identifier case (case 1) we only modify the term environment. Identity variables (case 2), position variables (case 4), and term restrictions (case 5) are translated using appropriate $\doteq$, $=_{@}$, and $\hat{=}$, respectively, between the involved CIQLog variables. Descendant sub-terms are a bit like the translation of a singleton term list using tq_{tlist}: we do not need to establish any relations to left or right siblings (as there are none) but can directly translate the single child, but using the transitive closure path^{L}_{*} instead of path^{L} as relation between parent and child variable. The translation of the basic terms (case 7–10) is fairly straightforward: we translated label parts, if there are any, then children, if there are any, and finally establish structural relations between the parent and sibling variables using tq_{struct}.

Examples. To illustrate the translation of query terms, we once again turn back to the examples from Section 3.3, in this case the example query terms in Tables 9 and 11. Tables 24 and 26 show the translation to CIQLog for some of these query terms as well as the environment passed to the translation of a corresponding rule head. If these query terms occur as top-level query terms, we have to add a root relation on the top-level CIQLog variable that we omit in the following for space reasons.

functionterm	= CIQLog **expression**
$\mathrm{tq}_{\mathit{tlist}}(\mathcal{E},\mathcal{V})\langle t_1,\dots,t_n\rangle$	$= \left(\mathcal{E}_n, \bigcup V_i^+, \bigwedge_{t_i \text{ positive}} F_i \wedge \neg\left(\bigwedge_{t_i \text{ negative}} F_i\right)\right)$ **where** $\mathcal{E}_0 = \mathcal{E}$, $V_0 = \mathcal{V}.V_\leftarrow$ t_i positive: $V_i^+ = V_i$, $V_i^- = \varnothing$, $V_i^{\mathit{left}} = \bigcup_{j<i} V_j^+$, $V_i^{\mathit{right}} = \mathcal{V}.V_\rightarrow$ t_i negative: $V_i^+ = \varnothing$, $V_i^- = V_i$, $V_i^{\mathit{left}} = \bigcup_{j<i} V_j$, $V_i^{\mathit{right}} = \bigcup_{j>i} V_j^+ \cup \mathcal{V}.V_\rightarrow$ $(\mathcal{E}_i, V_i, F_i) = \mathrm{tq}_{\mathit{term}}(\mathcal{E}_{i-1}, \bot, (\mathcal{V}.v_\uparrow, V_i^{\mathit{left}}, V_i^{\mathit{right}}, \mathcal{V}.r_\uparrow, \mathcal{V}.r_\leftarrow, \mathcal{V}.r_\rightarrow))$
$\mathrm{tq}_{\mathit{child}}(\mathcal{E},v_\uparrow)\langle\texttt{\{\{\{}\ \mathit{tlist}\texttt{\}\}\}}\rangle$	$= (\mathcal{E}, F)$ **where** $(\mathcal{E}, V, F) = \mathrm{tq}_{\mathit{tlist}}(\mathcal{E}, (v_\uparrow, \varnothing, \varnothing, \mathrm{path}^L, \top, \top))\langle \mathit{tlist}\rangle$
$\mathrm{tq}_{\mathit{child}}(\mathcal{E},v_\uparrow)\langle\texttt{\{\{}\ \mathit{tlist}\texttt{\}\}}\rangle$	$= (\mathcal{E}, F)$ **where** $(\mathcal{E}, V, F) = \mathrm{tq}_{\mathit{tlist}}(\mathcal{E}, (v_\uparrow, \varnothing, \varnothing, \mathrm{path}^L, \mathsf{C}\doteq, \mathsf{C}\doteq))\langle \mathit{tlist}\rangle$
$\mathrm{tq}_{\mathit{child}}(\mathcal{E},v_\uparrow)\langle\texttt{\{}\ \mathit{tlist}\texttt{\}}\rangle$	$= (\mathcal{E}, \mathrm{outdeg}^L(v_\uparrow, n) \wedge F)$ **where** $(\mathcal{E}, V, F) = \mathrm{tq}_{\mathit{tlist}}(\mathcal{E}, (v_\uparrow, \varnothing, \varnothing, \mathrm{path}^L, \mathsf{C}\doteq, \mathsf{C}\doteq))\langle \mathit{tlist}\rangle$ n number of terms in *tlist*
$\mathrm{tq}_{\mathit{child}}(\mathcal{E},v_\uparrow)\langle\texttt{[[[}\ \mathit{tlist}\texttt{]]]}\rangle$	$= (\mathcal{E}, \mathfrak{O}^L(v_p) \wedge F)$ **where** $(\mathcal{E}, V, F) = \mathrm{tq}_{\mathit{tlist}}(\mathcal{E}, (v_\uparrow, \varnothing, \varnothing, \mathrm{path}^L, \ll^L_*, \ll^L_*))\langle \mathit{tlist}\rangle$
$\mathrm{tq}_{\mathit{child}}(\mathcal{E},v_\uparrow)\langle\texttt{[[}\ \mathit{tlist}\texttt{]]}\rangle$	$= (\mathcal{E}, \mathfrak{O}^L(v_p) \wedge F)$ **where** $(\mathcal{E}, V, F) = \mathrm{tq}_{\mathit{tlist}}(\mathcal{E}, (v_\uparrow, \varnothing, \varnothing, \mathrm{path}^L, \ll^L_+, \ll^L_+))\langle \mathit{tlist}\rangle$
$\mathrm{tq}_{\mathit{child}}(\mathcal{E},v_\uparrow)\langle\texttt{[}\ \mathit{tlist}\texttt{]}\rangle$	$= (\mathcal{E}, \mathrm{outdeg}^L(v_\uparrow, n) \wedge \mathfrak{O}^L(v_p) \wedge F)$ **where** $(\mathcal{E}, V, F) = \mathrm{tq}_{\mathit{tlist}}(\mathcal{E}, (v_\uparrow, \varnothing, \varnothing, \mathrm{path}^L, \ll^L_+, \ll^L_+))\langle \mathit{tlist}\rangle$ n number of terms in *tlist*
$\mathrm{tq}_{\mathit{label}}(\mathcal{E},v)\langle *\rangle$	$= (\mathcal{E}, \top)$
$\mathrm{tq}_{\mathit{label}}(\mathcal{E},v)\langle \mathit{label}\rangle$	$= (\mathcal{E}, \mathfrak{L}(v, \mathit{label}))$
$\mathrm{tq}_{\mathit{label}}(\mathcal{E},v)\langle\texttt{var X}\rangle$	$= \begin{cases} (\mathcal{E} \mathbin{+\mkern-10mu+} \mathrm{isLabel}(\texttt{X}), \cong (v, \mathcal{E}(\texttt{X}) \wedge F)) & \text{if } \exists v' : (\texttt{X}, v') \in \mathcal{E} \text{ and} \\ & F = \bigwedge_{\mathrm{isTerm}(\texttt{X},v)\in\mathcal{E}} \mathrm{outdeg}(v, 0) \\ (\mathcal{E} \mathbin{+\mkern-10mu+} (\texttt{X}, v) \mathbin{+\mkern-10mu+} \mathrm{isLabel}(\texttt{X}), \top) & \text{otherwise} \end{cases}$
$\mathrm{tq}_{\mathit{var}}(\mathcal{E},v,\mathrm{eq})\langle\texttt{X}\rangle$	$= \begin{cases} (\mathcal{E}, v \cong \mathcal{E}(\texttt{X})) & \text{if } \exists v' : \mathrm{isLabel}(\texttt{X}) \in \mathcal{E} \wedge (\texttt{X}, v') \in \mathcal{E} \\ (\mathcal{E}, v \ \mathrm{eq}\ \mathcal{E}(\texttt{X})) & \text{if } \exists v' : \mathrm{isLabel}(\texttt{X}) \notin \mathcal{E} \wedge (\texttt{X}, v') \in \mathcal{E} \\ (\mathcal{E} \mathbin{+\mkern-10mu+} (\texttt{X}, v), \top) & \text{otherwise} \end{cases}$
$\mathrm{tq}_{\mathit{struct}}(v_{\mathit{curr}},\mathcal{V})$	$= r_\uparrow(v_\uparrow, v_{\mathit{curr}}) \wedge \bigwedge_{v_l \in V_\leftarrow} r_\leftarrow(v_l, v_{\mathit{curr}}) \wedge \bigwedge_{v_r \in V_\rightarrow} r_\rightarrow(v_r, v_{\mathit{curr}})$
$\mathrm{tq}_{\mathit{struct}}(v_{\mathit{curr}},\mathcal{V})$	$= r_\uparrow(v_\uparrow, v_{\mathit{curr}})$ if $r_\rightarrow = r_\leftarrow = \top$

Table 22. Translating Xcerpt query terms: term lists, variables, and labels

	Query term	CIQLog expression	$\mathcal{E}$
T1	`a{ }`	$\xrightarrow{\mathsf{tq}_{term}}$ $\mathfrak{L}(v_1,a) \wedge \text{outdeg}^L(v_1,0)$	$\varnothing$
T2	`a[ ]`	$\xrightarrow{\mathsf{tq}_{term}}$ $\mathfrak{L}(v_1,a) \wedge \text{outdeg}^L(v_1,0) \wedge \mathfrak{O}^L(v_1)$	$\varnothing$
T3	`a{ b }`	$\xrightarrow{\mathsf{tq}_{term}}$ $\mathfrak{L}(v_1,a) \wedge \text{outdeg}^L(v_1,1) \wedge \text{CHILD}(v_1,v_2) \wedge \mathfrak{L}(v_2,b)$	$\varnothing$
T4	`a{ b, b }`	$\xrightarrow{\mathsf{tq}_{term}}$ $\mathfrak{L}(v_1,a) \wedge \text{outdeg}^L(v_1,2) \wedge \text{CHILD}(v_1,v_2) \wedge \mathfrak{L}(v_2,b) \wedge$ $\text{CHILD}(v_1,v_3) \wedge \mathfrak{L}(v_3,b) \wedge v_2\ \mathsf{C}\doteq\ v_3$	$\varnothing$
P1	`a{{ b }}`	$\xrightarrow{\mathsf{tq}_{term}}$ $\mathfrak{L}(v_1,a) \wedge \text{CHILD}(v_1,v_2) \wedge \mathfrak{L}(v_2,b)$	$\varnothing$
P2	`a[[ b, c ]]`	$\xrightarrow{\mathsf{tq}_{term}}$ $\mathfrak{L}(v_1,a) \wedge \mathfrak{O}^L(v_1) \wedge \text{CHILD}(v_1,v_2) \wedge \mathfrak{L}(v_2,b) \wedge$ $\text{CHILD}(v_1,v_3) \wedge \mathfrak{L}(v_3,c) \wedge v_2 \ll_{+} v_3$	$\varnothing$
I1	`a{{{ b, b }}}`	$\xrightarrow{\mathsf{tq}_{term}}$ $\mathfrak{L}(v_1,a) \wedge \text{CHILD}(v_1,v_2) \wedge \mathfrak{L}(v_2,b) \wedge$ $\text{CHILD}(v_1,v_3) \wedge \mathfrak{L}(v_3,b)$	$\varnothing$
I2	`a[[[ b, b, d ]]]`	$\xrightarrow{\mathsf{tq}_{term}}$ $\mathfrak{L}(v_1,a) \wedge \mathfrak{O}^L(v_1) \wedge \text{CHILD}(v_1,v_2) \wedge \mathfrak{L}(v_2,b) \wedge$ $\text{CHILD}(v_1,v_3) \wedge \mathfrak{L}(v_3,b) \wedge \text{CHILD}(v_1,v_4) \wedge \mathfrak{L}(v_4,d) \wedge$ $v_2 \ll_{*} v_3 \wedge v_2 \ll_{*} v_4 \wedge v_3 \ll_{*} v_4$	$\varnothing$
D1	`a{ desc b }`	$\xrightarrow{\mathsf{tq}_{term}}$ $\mathfrak{L}(v_1,a) \wedge \text{outdeg}^L(v_1,1) \wedge \text{CHILD}(v_1,v_2) \wedge$ $\text{CHILD}_{*}(v_2,v_3) \wedge \mathfrak{L}(v_3,\ b)$	$\varnothing$
D2	`a{ desc b, desc c }`	$\xrightarrow{\mathsf{tq}_{term}}$ $\mathfrak{L}(v_1,a) \wedge \text{outdeg}^L(v_1,2) \wedge \text{CHILD}(v_1,v_2) \wedge$ $\text{CHILD}(v_1,v_3) \wedge v_2\ \mathsf{C}\doteq\ v_3 \wedge \text{CHILD}_{*}(v_2,v_4) \wedge$ $\mathfrak{L}(v_4,b) \wedge \text{CHILD}_{*}(v_3,v_5) \wedge \mathfrak{L}(v_5,c)$	$\varnothing$
D3	`a{{ desc b, desc c }}`	$\xrightarrow{\mathsf{tq}_{term}}$ $\mathfrak{L}(v_1,a) \wedge \text{CHILD}(v_1,v_2) \wedge$ $\text{CHILD}(v_1,v_3) \wedge v_2\ \mathsf{C}\doteq\ v_3 \wedge \text{CHILD}_{*}(v_2,v_4) \wedge$ $\mathfrak{L}(v_4,b) \wedge \text{CHILD}_{*}(v_3,v_5) \wedge \mathfrak{L}(v_5,c)$	$\varnothing$
D4	`a{{{ desc b, desc c }}}`	$\xrightarrow{\mathsf{tq}_{term}}$ $\mathfrak{L}(v_1,a) \wedge \text{CHILD}(v_1,v_2) \wedge \text{CHILD}(v_1,v_3) \wedge$ $\text{CHILD}_{*}(v_2,v_4) \wedge \mathfrak{L}(v_4,b) \wedge \text{CHILD}_{*}(v_3,v_5) \wedge \mathfrak{L}(v_5,c)$	$\varnothing$
W1	`a{{ b, without( c ), d }}`	$\xrightarrow{\mathsf{tq}_{term}}$ $\mathfrak{L}(v_1,a) \wedge \text{CHILD}(v_1,v_2) \wedge \mathfrak{L}(v_2,b) \wedge \text{CHILD}(v_1,v_3) \wedge$ $\mathfrak{L}(v_3,d) \wedge v_2\ \mathsf{C}\doteq\ v_3 \wedge$ $\neg\big(\text{CHILD}(v_1,v_4) \wedge \mathfrak{L}(v_4,c) \wedge v_2\ \mathsf{C}\doteq\ v_4 \wedge v_3\ \mathsf{C}\doteq\ v_4\big)$	$\varnothing$
W2	`a[[ b, without( c ), d ]]`	$\xrightarrow{\mathsf{tq}_{term}}$ $\mathfrak{L}(v_1,a) \wedge \mathfrak{O}^L(v_1) \wedge \text{CHILD}(v_1,v_2) \wedge \mathfrak{L}(v_2,b) \wedge$ $\text{CHILD}(v_1,v_3) \wedge \mathfrak{L}(v_3,d) \wedge v_2 \ll_{+} v_3 \wedge$ $\neg\big(\text{CHILD}(v_1,v_4) \wedge \mathfrak{L}(v_4,c) \wedge v_2 \ll_{+} v_4 \wedge v_4 \ll_{+} v_3\big)$	$\varnothing$

Table 24. Query terms and their CIQLog translation

Notice in Table 24 in particular the subtle, but essential differences in the translations for total (T1–T2), partial, but injective (P1–P2), and partial and non-injective (I1–I2) query terms. I2 is an example where more than necessary sibling relations are generated. This is the case for all order relations between siblings as the translation above does not exploit their transitivity. This can be easily recognized and removed in a post-processing step. Alternatively, one can adapt the translation to handle ordered and unordered term lists differently. D1, D4 show that when translating `desc` we always generate an intermediate child step to express sibling relations on that child step. However, if there are no such relations (because it is the only sub-term or because we are in a partial, non-injective term) we can avoid the child step and use path^L_+ instead of path^L followed by path^L_*. Again this is a general equivalence for CIQLog queries on data graph relations and can be optimized in a post-processing.

	Query term	CIQLog expression	$\mathcal{E}$
V1	`a{ var X }`	$\xrightarrow{\text{tq}_{term}}$ $\mathfrak{L}(v_1,a) \land \text{outdeg}^L(v_1,1) \land \text{CHILD}(v_1,v_2)$	$\{(X, v_2)\}$
V2	`a{{ var X }}`	$\xrightarrow{\text{tq}_{term}}$ $\mathfrak{L}(v_1,a) \land \text{CHILD}(v_1,v_2)$	$\{(X, v_2)\}$
V3	`a{{ var X, var X }}`	$\xrightarrow{\text{tq}_{term}}$ $\mathfrak{L}(v_1,a) \land \text{CHILD}(v_1,v_2) \land \text{CHILD}(v_1,v_3) \land v_2 \complement\doteq v_3 \land v_2 \circeq v_3$	$\{(X, v_2)\}$
V4	`a{{{ var X, var X }}}`	$\xrightarrow{\text{tq}_{term}}$ $\mathfrak{L}(v_1,a) \land \text{CHILD}(v_1,v_2) \land \text{CHILD}(v_1,v_3) \land v_2 \circeq v_3$	$\{(X, v_2)\}$
V5	`a{ var X{ var X } }`	$\xrightarrow{\text{tq}_{term}}$ $\mathfrak{L}(v_1,a) \land \text{CHILD}(v_1,v_2) \land \text{CHILD}(v_2,v_3) \land \text{outdeg}^L(v_3,0) \land v_2 \cong v_3$	$\{(X, v_2), \text{isLabel}(X)\}$
V6	`a{ var X → c, var X }`	$\xrightarrow{\text{tq}_{term}}$ $\mathfrak{L}(v_1,a) \land \text{CHILD}(v_1,v_2) \land \mathfrak{L}(v_2, c) \land \text{CHILD}(v_1,v_3) \land v_2 \circeq v_3$	$\{(X, v_2)\}$
V7	`a{ desc var X }`	$\xrightarrow{\text{tq}_{term}}$ $\mathfrak{L}(v_1,a) \land \text{outdeg}^L(v_1,1) \land \text{CHILD}(v_1,v_2) \land \text{CHILD}_*(v_2,v_3)$	$\{(X, v_3)\}$
V8	`a{{var X, without(var X)}}`	$\xrightarrow{\text{tq}_{term}}$ $\mathfrak{L}(v_1,a) \land \text{CHILD}(v_1,v_2) \land \neg\big(\text{CHILD}(v_1,v_3) \land v_2 \complement\doteq v_3 \land v_2 \circeq v_3\big)$	$\{(X, v_2)\}$

Table 26. Query terms containing variables and their CIQLog translation

Query terms with variables are considered in Table 26. Notice, in particular V3 and V6 that illustrate multiple occurrences of the same variable in a total or partial, injective query term: here we demand that matches for the two terms are not the same node (i.e., in the complement relation of $\doteq$) but have the same label and structure (i.e., stand in $\circeq$ relation). In V5, the effect of a label occurrence of a variable is illustrated.

3.5 From Non-recursive, single-rule Core Xcerpt to Full Xcerpt

In the previous sections, we focus on $\text{Xcerpt}^{\text{core,NR,SR}}$, i.e., the non-recursive, single-rule fragment of Xcerpt. Notice, however, that the above translation generates a CIQLog rule from a Core Xcerpt rule and is applicable whether there are only one or many rules in a translated Xcerpt program.

Thus, for translating full Core Xcerpt, i.e., possibly recursive, multi-rule Xcerpt but with the restrictions from Section 3.2 wrt. each rule, we can use the above translation unchanged. Some improvements of the resulting CIQLog program can be achieved by exploiting that Xcerpt programs are grouping and negation stratified. Before the translation, we select one such stratification and rewrite the program to introduce root terms with unique labels for each stratum. This allows for easier recognition of said strata after the translation to CIQLog, where we only have to consider the root variable and its label to limit rule chaining to each stratum.

Moving from Core Xcerpt to full Xcerpt is less obvious. Aside from numerous, but essentially easy specificities such as namespaces, attributes, comments, processing-instructions etc. there are a few constructs that merit a closer consideration. Among them are:

(1) `optional` TERMS: Optional QUERY terms can be rewritten to combinations of `without` and `or` but require specific treatment in heads.[2] For that CIQLog provides conditional construction which precisely addresses optional construct terms.

(2) `order-by` CLAUSES: `order-by` clauses in Xcerpt allow for different lists of order variables than grouping variables whereas $\text{Xcerpt}^{\text{core,NR,SR}}$ always assumes that both lists are identical. This is supported by CIQLog (the variable lists in order do not have to coincide with the ones in new) and can be easily added to the translation by a new $\mathcal{E}$.order sequence of variables.

(3) GROUPING OVER TERM LISTS instead of single terms yields results not expressible in $\text{Xcerpt}^{\text{core,NR,SR}}$: `a[b, all(var X, var Y), d]` yields

```
a[b, x₁, y₁, x₂, y₂, ..., xₙ, yₙ, d]
```

where x_i is the binding for `X` in the i-th binding tuple, i.e., bindings for `X` and `Y` are paired wrt. term order. This can be accommodated in the translation by nesting order expressions as in the translation of XQuery (cf. Chapter 4).

(4) ADDITIONAL GROUPING EXPRESSIONS such as `some` and `first` groupings are to some extent expressible using aggregation operators in CIQLog. However, in general, `some` is not expressible in CIQLog as its result is non-deterministic and CIQLog expresses only deterministic queries (up to isomorphisms on invented values).

(5) CONDITIONS IN XCERPT QUERY TERMS can be translated to CIQLog if appropriate relations or functions are available (or added) to CIQLog.

[2] This is the case, as we can not always split the Xcerpt rule in two rules on the Xcerpt level due to grouping expressions.

Chapter 4

Translating XQuery

4.1 Introduction

Among XML query languages, XPath and, increasingly, XQuery play the dominant role to such an extent that motivating their use in this work has become superfluous. XPath's properties, evaluation, complexity, containment, etc. have been studied extensively in recent years, for a survey see [9]. For XQuery, most research still focuses on implementation and evaluation aspects, e.g., [32, 55, 11]. In the following, we present a novel semantics for XPath and XQuery by translation to CIQLog that also serves as foundation for the evaluation of XPath and XQuery with the CIQCAG algebra defined in [34]. Together with the CIQCAG algebra, we achieve the first implementation of XQuery that scales from tree queries to graph queries, from tree data to graph data. Its time and space complexity is as good as or better than the complexities of previous systems limited to, e.g., tree queries on tree data (for details on the complexity see [34]).

The translation of XPath and XQuery to CIQLog is the focus of this chapter. For the most part, we use navigational XPath (introduced in [38]) and composition-free Core XQuery[1] (introduced in [47]), two important and convenient fragments of XPath and XQuery that can be translated to non-recursive CIQLog. The extension to full XQuery is only briefly outlined in Section 4.4. The essential limitation of both fragments compared to the full languages is that all relations in the query are only on nodes of the input tree but not on nodes by the query itself. This limitation is similar to the limitation to Xcerpt$^{\text{core,NR,SR}}$ in Chapter 3.

In addition to providing a path for implementing XPath and XQuery using CIQCAG, the translation also gives a purely logical semantics for both languages where previous semantics for XQuery are functional or algebraic (see also Table 31). This sheds new light on some of the differences between composition-free XQuery on the one hand and XPath on the other hand, in particular, on the effect of nested **for** loops and element construction in XQuery.

[1]In the following, we omit the "Core" where no confusion is possible.

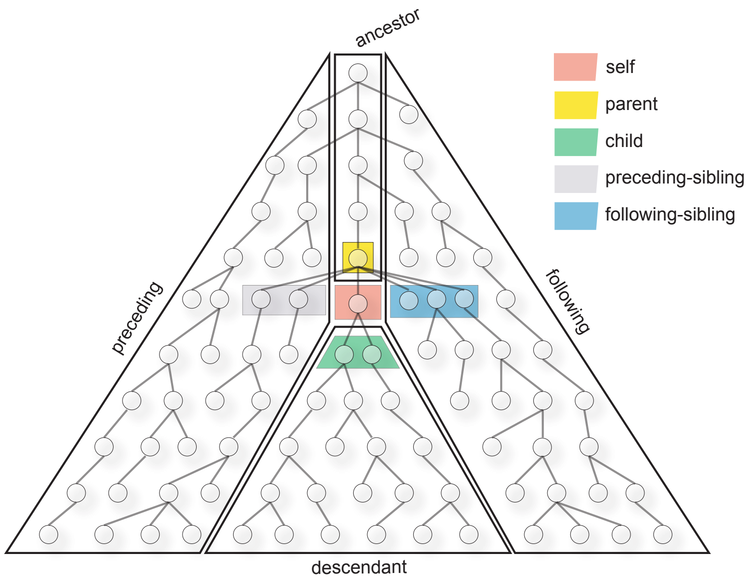


Figure 16. XPath axis (from [36])

4.2 Translating XPath

Unary selection of XML elements is, by now, almost always done using XPath or some variant of XPath (such as XPointer). XPath provides an elegant and compact way of describing "paths" in an XML document (represented similar to Chapter 1.3 but without resolution of ID/IDREF links). Paths are made up of "steps" each specifying a direction, called *axis*, in which to navigate through the document, e.g., child, following, or ancestor, cf. Figure 16 for the full set of axes. Together with the axis, a step contains a restriction on the type or label of the data items to be selected, called *node test*. Node tests may be labels of element or attribute nodes, node kind wildcards such as * (any node with some label), element(), node(), text(), or comment(). Any step may be adorned by one or more *qualifiers* each expressing additional restrictions on the selected nodes. Compared to languages such as XQuery, Xcerpt, or even SPARQL, the most distinctive feature of XPath is the lack of explicit variables. This makes it impossible to express n-ary queries and limits XPath, for the most part, to two-variable logic [54, 10].

For instance, the XPath expression `/descendant::paper/child::author` consists of two steps, the first selecting `paper` elements that are descendants of the root ("of the root" is indicated by the leading slash), the second selecting `author` children of such `paper` elements. More interesting queries can be expressed by exploiting XPath's qualifiers, e.g., the following XPath expression that selects all authors that are also PC members of a conference (more precisely that have node children with the same label):

```
/child::conference/descendant::paper/child::author[child::node() =
     /child::conference/child::member/child::node()]
```

In addition to the strict axis plus node test notation, XPath uses also an abbreviated syntax where child axis may be omitted, descendant is (roughly) abbreviated by // etc. In the following, we only use the full syntax. We also limit ourselves to the core feature of XPath as discussed here and thus present a view of XPath similar to Navigational XPath of [37] and [9]. Due to [61], we also limit ourselves to forward axes such as child and following, rewriting expressions with reverse axes such as parent, ancestor, or preceding where necessary.

4.2.1 Syntax and Semantics

Following [9], we define the semantics of XPath over a relational structure as introduced in Section 1.6.1: An XML-tree is considered a relational structure T over the schema $((\mathsf{Lab}^{\lambda})_{\lambda\in\Sigma}, R_{\text{child}}, R_{\text{nest-sibling}}, relRoot)$. The nodes of this tree are labeled using the symbols from σ which are queried using $\mathfrak{L}^{\lambda}$ (note, that λ is a single label not a label set as in the graph relations of Chapter 1). The parent-child relations are represented by R_{child}. The order between siblings is represented by $R_{\text{nest-sibling}}$. The root node of the tree is identified by root. It is easy to see that this view of XML trees (which is as in [9] or [47]), makes an XML-tree a specific instance of a CIQLog data graph, cf. Chapter 1. There are some additional derived relations, viz. $R_{\text{descendant}}$, the transitive, $R_{\text{descendant-or-self}}$ the transitive reflexive closure of R_{child}, $R_{\text{following-sibling}}$, the transitive closure of $R_{\text{next-sibling}}$, R_{self} relating each node to itself, and $R_{\text{following}}$ the composition of $R^{-1}_{\text{descendant-or-self}} \circ R_{\text{following-sibling}} \circ R_{\text{descendant-or-self}}$. Finally, we can compare nodes based on their label using $\cong$ which contains all pairs of nodes with same label.

The syntax of navigational XPath is defined as follows (again following [37] and [9]):

$$
\begin{array}{lll}
\langle path\rangle & ::= & \langle step\rangle \mid \langle step\rangle \text{ `/' } \langle path\rangle \mid \langle path\rangle \text{ `}\cup\text{'} \langle path\rangle \mid \text{`/' } \langle path\rangle \\
\langle step\rangle & ::= & \langle axis\rangle \text{ `::' } \langle node\text{-}test\rangle \mid \langle step\rangle\text{`['}\langle qualifier\rangle\text{`]'} \\
\langle axis\rangle & ::= & \text{`child'} \mid \text{`descendant'} \mid \text{`descendant-or-self'} \\
 & \mid & \text{`next-sibling'} \mid \text{`following-sibling'} \mid \text{`following'} \\
\langle node\text{-}test\rangle & ::= & \langle label\rangle \mid \text{`node()'} \\
\langle qualifier\rangle & ::= & \langle path\rangle \mid \langle path\rangle \text{ `}\wedge\text{'}\langle path\rangle \mid \langle path\rangle \text{ `}\vee\text{'}\langle path\rangle \mid \text{`}\neg\text{'}\langle path\rangle \\
 & \mid & \text{`lab()' `=' `}\lambda\text{'} \\
 & \mid & \langle path\rangle \text{ `=' } \langle path\rangle
\end{array}
$$

The semantics of a navigational XPath expression over a relational structure T representing an XML tree (as defined above) is defined in Table 27 by means of $[\![\ ]\!]_{\text{Nodes}}(n)$ where n is a node, called *context node*. $[\![\ ]\!]_{\text{Nodes}}(n)$ associates each XPath expression and context node with a set of nodes that constitutes the semantics of that expression if evaluated with the given context node. It uses $[\![\ ]\!]_{\text{Bool}}(n)$ for the semantics of qualifiers under a context node n.

For details on the semantics as well as differences to full XPath see [9].

4.2.2 Translation

Consider again the above examples. The first (`/descendant::paper/child::author`) is translated to the following CIQLog rule:

```
ans(v₃) ⟵ root(v₁) ∧ CHILD₊(v₁,v₂) ∧ 𝔏(v₂,paper) ∧
  CHILD(v₂,v₃) ∧ 𝔏(v₃,author)
```

$[\![axis]\!]_{\mathsf{Nodes}}(n)$	$= \{(n' : R_{axis}(n, n')\}$
$[\![\lambda]\!]_{\mathsf{Nodes}}(n)$	$= \{(n' : \mathsf{Lab}^{\lambda}(n')\}$
$[\![\mathtt{node()}]\!]_{\mathsf{Nodes}}(n)$	$= \mathsf{Nodes}(T)$
$[\![axis\mathtt{::}nt\mathtt{[}qual\mathtt{]}]\!]_{\mathsf{Nodes}}(n)$	$= \{n' : n' \in [\![axis]\!]_{\mathsf{Nodes}} \wedge n' \in [\![nt]\!]_{\mathsf{Nodes}} \wedge [\![qual]\!]_{\mathsf{Bool}}(n')\}$
$[\![step/path]\!]_{\mathsf{Nodes}}(n)$	$= \{n'' : n' \in [\![step]\!]_{\mathsf{Nodes}}(n) \wedge n'' \in [\![path]\!]_{\mathsf{Nodes}}(n')\}$
$[\![path_1 \cup path_2]\!]_{\mathsf{Nodes}}(n)$	$= [\![path_1]\!]_{\mathsf{Nodes}}(n) \cup [\![path_2]\!]_{\mathsf{Nodes}}(n)$
$[\![path]\!]_{\mathsf{Bool}}(n)$	$= [\![path]\!]_{\mathsf{Nodes}}(n) \neq \varnothing$
$[\![path_1 \wedge path_2]\!]_{\mathsf{Bool}}(n)$	$= [\![path_1]\!]_{\mathsf{Bool}}(n) \wedge [\![path_2]\!]_{\mathsf{Bool}}(n)$
$[\![path_1 \vee path_2]\!]_{\mathsf{Bool}}(n)$	$= [\![path_1]\!]_{\mathsf{Bool}}(n) \vee [\![path_2]\!]_{\mathsf{Bool}}(n)$
$[\![\neg path]\!]_{\mathsf{Bool}}(n)$	$= \neg [\![path]\!]_{\mathsf{Bool}}(n)$
$[\![\mathtt{lab()} = \lambda]\!]_{\mathsf{Bool}}(n)$	$= \mathsf{Lab}^{\lambda}(n)$
$[\![path_1 = path_2]\!]_{\mathsf{Bool}}(n)$	$= \exists n', n'' : n' \in [\![path_1]\!]_{\mathsf{Nodes}}(n) \wedge n'' \in [\![path_2]\!]_{\mathsf{Nodes}}(n) \wedge \cong (n', n'')$

Table 27. Semantics for navigational XPath (following [9])

We use ans as the canonical answer predicate containing the (single) answer variable whose bindings represent the results of an XPath expression. Just as the original expression, the body of the CIQLog rule selects descendants of the root with label paper and of those the author children. The latter are propagated to the head of the rule.

The second example is as follows

```
/child::conference/descendant::paper/child::author[child::node() =
    /child::conference/child::member/child::node()]
```

and translated similarly but illustrates qualifiers and nested queries:

```
ans(v₄) ⟵ root(v₁) ∧ CHILD(v₁,v₂) ∧ 𝔏(v₂,conference) ∧
  CHILD₊(v₂,v₃) ∧ 𝔏(v₃,paper) ∧ CHILD(v₃,v₄) ∧ 𝔏(v₄,author) ∧
  CHILD(v₄,v₅) ∧ ≅(v₅,w₄) ∧ root(w₁) ∧ CHILD(w₁,w₂) ∧
  𝔏(w₂,conference) ∧ CHILD(w₂,w₃) ∧ 𝔏(w₃, member) ∧ CHILD(w₃,w₄)
```

In general, we translate a navigational XPath expression using the $\mathsf{tr}_{\mathsf{XPath}}$ specified in Table 29. $\mathsf{tr}_{\mathsf{XPath}}$ returns a CIQLog formula that realizes the given XPath expression as well as an answer variable. We wrap $(Q, w) = \mathsf{tr}_{\mathsf{XPath}}(\perp, \perp)$ where $\perp$ is an arbitrary variable into a CIQLog rule $\mathtt{ans}(w) \longleftarrow Q$.

functionXPath expression	**= CIQLog expression**
$\text{tr}_{\text{XPath}}(v, v')\langle \mathit{axis} \rangle$	$= (\text{relation}(\mathit{axis})(v, v'), v')$
$\text{tr}_{\text{XPath}}(v, v')\langle \lambda \rangle$	$= (\mathfrak{L}(v', \lambda), v')$
$\text{tr}_{\text{XPath}}(v, v')\langle \texttt{node()} \rangle$	$= (\top, v')$
$\text{tr}_{\text{XPath}}(v, v')\langle \mathit{axis}::\mathit{node\text{-}test}[\mathit{qualifier}] \rangle$	$= (\text{tr}_{\text{XPath}}(v, v')\langle \mathit{axis} \rangle \land \text{tr}_{\text{XPath}}(v, v')\langle \mathit{node\text{-}test} \rangle \land \text{tr}_{\text{XPath}}(v', v'')\langle \mathit{qualifier} \rangle, v')$ **where** v'' is a new variable
$\text{tr}_{\text{XPath}}(v, v')\langle \mathit{step}/\mathit{path} \rangle$	$= (\text{tr}_{\text{XPath}}(v, v')\langle \mathit{step} \rangle \land F, w)$ **where** v'' is a new variable $(F, w) = \text{tr}_{\text{XPath}}(v', v'')\langle \mathit{path} \rangle$
$\text{tr}_{\text{XPath}}(v, v')\langle /\mathit{path} \rangle$	$= (\text{root}(v'') \land F, w)$ **where** v'', v''' are new variables $(F, w) = \text{tr}_{\text{XPath}}(v'', v''')\langle \mathit{path} \rangle$
$\text{tr}_{\text{XPath}}(v, v')\langle \mathit{path}_1 \cup \mathit{path}_2 \rangle$	$= ((F_1 \lor F_2) \land (w \doteq w_1 \lor w \doteq w_1), w)$ **where** w is a new variable $(F_1, w_1) = \text{tr}_{\text{XPath}}(v, v')\langle \mathit{path}_1 \rangle$ $(F_2, w_2) = \text{tr}_{\text{XPath}}(v, v')\langle \mathit{path}_2 \rangle$
$\text{tr}_{\text{XPath}}(v, v')\langle \mathit{path}_1 \lor \mathit{path}_2 \rangle$	$= ((F_1 \lor F_2), w_1)$ **where** $(F_1, w_1) = \text{tr}_{\text{XPath}}(v, v')\langle \mathit{path}_1 \rangle$ $(F_2, w_2) = \text{tr}_{\text{XPath}}(v, v')\langle \mathit{path}_2 \rangle$
$\text{tr}_{\text{XPath}}(v, v')\langle \mathit{path}_1 \land \mathit{path}_2 \rangle$	$= (F_1 \land F_2, w_1)$ **where** $(F_1, w_1) = \text{tr}_{\text{XPath}}(v, v')\langle \mathit{path}_1 \rangle$ $(F_2, w_2) = \text{tr}_{\text{XPath}}(v, v')\langle \mathit{path}_2 \rangle$
$\text{tr}_{\text{XPath}}(v, v')\langle \neg \mathit{path} \rangle$	$= (\neg(F), w)$ **where** $(F, w) = \text{tr}_{\text{XPath}}(v, v')\langle \neg \mathit{path} \rangle$
$\text{tr}_{\text{XPath}}(v, v')\langle \texttt{lab()} = \lambda \rangle$	$= (\mathfrak{L}(v, \lambda), v)$
$\text{tr}_{\text{XPath}}(v, v')\langle \mathit{path}_1 = \mathit{path}_2 \rangle$	$= (F_1 \land F_2 \land w_1 \cong w_2, w_1)$ **where** $(F_1, w_1) = \text{tr}_{\text{XPath}}(v, v')\langle \mathit{path}_1 \rangle$ $(F_2, w_2) = \text{tr}_{\text{XPath}}(v, v')\langle \mathit{path}_2 \rangle$

Table 29. Translating navigational XPath

The translation in Table 29 is fairly straightforward. In a slight abuse of notation, we allow the direct use tr_{XPath} in a formula. In this case we ignore the second return value (i.e., the answer variable). The translation is parametrized by the parent variable v and the current variable v'. Axis are mapped to corresponding relations using the helper function $\text{relation}(\mathit{axis})$), as are label node-steps (cases 1–3). $\text{relation}(\mathit{axis})$ is defined in the obvious way:

$$\text{relation}(\text{child}) = \text{CHILD} \qquad \text{relation}(\text{descendant}) = \text{CHILD}_{+}$$
$$\text{relation}(\text{descendant-or-self}) = \text{CHILD}_{*} \qquad \text{relation}(\text{next-sibling}) = \ll^{\text{CHILD}}$$
$$\text{relation}(\text{following-sibling}) = \ll^{\text{CHILD}}_{+} \qquad \text{relation}(\text{following}) = \blacktriangleleft^{\text{CHILD}}_{+}$$

Steps are translated by translating axis, node-test, and qualifier separately. For the translation of the qualifier a new variable is created and the qualifier is translated with the old current variable as parent and the new variable as current variable. Notice, how we ignore the answer variable returned by the qualifier. This small difference to the translation of a path outside a qualifier implements the existential semantics of XPath qualifiers. Together with the path operator / (cases 5–6), the qualifier is the only context-changing expression where we move from one query variable to the next one. The inner path operator (case 5) translates the leftmost step and then continues with the translation of the remainder of the path using a new current variable. Absolute XPath expressions (expressions starting with /) are translated in case 7 where we use *fresh* variables rather than v and v'. Thus, the only possible link between nested absolute expressions is the answer variable (which is used, e.g., for unions or joins). In unions, conjunctions, disjunctions, and joins (cases 7–9, 12) we translate each operand separately but combine the result differently, most notably for conjunction, disjunction, and join (cases 8, 9, 12) we do not care about the answer variable (they only occur in qualifiers where, as described above, we drop answer variables anyway). Finally, label equality (case 11) is translated just like a labeled node test (case 2), but on the parent variable instead of the current variable due to the "context" switch of the qualifier.

Theorem 4.1. *Let P be a navigational XPath expression and $(Q, w) = tr_{XPath}(\bot, \bot)\langle P\rangle$. Then the relation* ans *defined by* $\text{ans}(w) \longleftarrow Q$ *is* $[\![P]\!]_{Nodes}$.

Proof (Sketch). First, consider only path expression without qualifiers. It is easy to see that, given appropriate base relations, the resulting CIQLog expressions express the same query as the XPath expressions. Note, that the answer variable is by translation the variable for the last step, as in XPath. With qualifiers, the same observation holds (as answer variables returned by paths in qualifiers are ignored by definition, see case 4). Note that the join in case 12 is a label join (as in the semantics of navigational XPath). In full XPath = is a join on the string value of a node, which is a notion covered neither in navigational XPath as defined in [9] nor by the CIQLog translation. □

4.3 From XPath to Composition-Free XQuery

Turning from XPath to a larger fragment of XQuery that is amenable to translation into CIQLog, we choose non-compositional XQuery as defined in [47] (though we follow more closely the variant in [8]).

4.3.1 Composition-Free XQuery in 1000 Words

Though not nearly as common as XPath, XQuery has nevertheless achieved the status of predominant XML query language, at least as far as database products and research are concerned (in total, XSLT [21] is probably still more widely supported and used). XQuery is essentially an extension of XPath (though some of its axis are only optional in XQuery), but most of XPath becomes syntactic sugar in XQuery. This is particularly true for XPath qualifiers which can be reduced to **`where`** or **`if`** clauses in XQuery. Indeed, the XQuery standard is accompanied [29] by a normalization of XQuery to a core dialect of the language.

Here, we consider first an important, if somewhat academic fragment of XQuery, viz. composition-free XQuery as defined in [47] and [8]. It is slightly academic as we restrict the syntax far more than necessary to minimize the constructs to be consider for the formal semantics of composition-free XQuery as well as for the translation to CIQLog. However, many of the restrictions to the syntax can be dropped (e.g., we could integrate full navigational XPath as discussed in Section 4.2) without affecting expressiveness and complexity, see also [8]. The only real restriction of composition-free XQuery in comparison to full XQuery is that it disallows any querying of constructed nodes, i.e., the domain of all relations is limited to the input nodes. This limitation clearly does not hold for full XQuery (even if we do not consider user-defined functions) and its effect on expressiveness and complexity is discussed in detail in [47].

(Composition-free) XQuery is built around controlled iterations over nodes of the input tree, expressed using **for** expressions. Controlled iteration is important for XQuery as it founded on sequences of nodes rather than sets of nodes (as XPath 1.0 and CIQLog). In this respect it is more similar to languages such as DAPLEX [70] or OQL [19] than to XPath or Xcerpt. (**For**) loops use XPath expressions for navigation and XML-look-a-likes for element construction all of which can be, essentially, freely nested. The following query gives an example of XQuery expressions. It creates a paperlist containing one author element for each author in the input XML tree (bound here and in the following to the canonical input variable `$inp`). For each such author, the nested **for** loop creates a list of all its papers. The latter expression can be more elegantly expressed in full XQuery using XPath qualifiers or **where** clauses but here it is shown in the "normalized" syntax of composition-free XQuery after [47].

```
<paperlist>
 for $a in $inp/descendant::author return
  <author> for $p in $inp/descendant::paper return
   if some $x in $p/child::author satisfies deep-equal($x, $a)
   then $p
  </author>
</paperlist>
```

4.3.2 Syntax

A full definition of the syntax of composition-free XQuery as used here is given in Table 30. It deviates only marginally from [47] and [8]. In addition to the specification in Table 30, the usual semantic restrictions apply, e.g., the label of the start and end tags must be the same, variables must be defined (using **for**) before use, etc. As stated, there is one exception from the latter, viz. the canonical input variable `$inp` which is always bound to the input XML tree.

In Table 30, we use a general equality. XQuery provides in fact three kinds of equality, viz. node, atomic (or value), and deep equality which correspond roughly to $\doteq$, $\cong$, and $\triangleq_{bij}$ of CIQLog data graphs. For all forms of equality the productions of Table 30 apply.

Again, compared to full XQuery the principle omission is the ability to query constructed nodes or values. In the syntax, this leads most prominently to the restriction of expressions following **in** in a **for**, i.e., expressions that provide bindings for variables, to XPath steps with variables. This way variables are always bound only to nodes from the input tree (anything reachable from `$inp` using XPath expressions). Another important omission is the absence of **let** clauses, which provide set-valued variables to XQuery. Conditional expressions are normalized to **if** clauses, where XQuery offers XPath qualifiers, **where** clauses, and **if** clauses.

Though **order-by** clauses are omitted, the result of an XQuery expression is always an ordered tree and the order of node construction must be precisely preserved (as given by the iteration of the **for** clauses which iterated over their respective node sequences mostly in document order).

⟨*query*⟩	::=	⟨*query*⟩ ⟨*query*⟩ | ⟨*element*⟩ | ⟨*variable*⟩
	|	⟨*step*⟩ | ⟨*iteration*⟩ | ⟨*conditional*⟩
⟨*element*⟩	::=	'<' ⟨*label*⟩ '>' ⟨*query*⟩ '<' ⟨*/label*⟩ '>'
	|	'<' '`lab(`' ⟨*variable*⟩ '`)>`' ⟨*query*⟩ '`</`' '`lab(`' ⟨*variable*⟩ '`)>`'
⟨*step*⟩	::=	⟨*variable*⟩ '/' ⟨*axis*⟩ '`::`' ⟨*node-test*⟩
⟨*iteration*⟩	::=	'`for`' ⟨*variable*⟩ '`in`' ⟨*step*⟩ '`return`' ⟨*query*⟩
⟨*conditional*⟩	::=	'`if`' ⟨*condition*⟩ '`then`' ⟨*query*⟩
⟨*condition*⟩	::=	⟨*variable*⟩ '=' ⟨*variable*⟩ | ⟨*variable*⟩ '=' '<' ⟨*label*⟩ '`/>`' | '`true`'
	|	'`some`' ⟨*variable*⟩ '`in`' ⟨*step*⟩ '`satisfies`' ⟨*condition*⟩
	|	⟨*condition*⟩ '`and`' ⟨*condition*⟩ | ⟨*condition*⟩ '`or`' ⟨*condition*⟩ | '`not`' ⟨*condition*⟩
⟨*axis*⟩	::=	'`child`' | '`descendant`' | '`descendant-or-self`'
	|	'`next-sibling`' | '`following-sibling`' | '`following`'
⟨*node-test*⟩	::=	⟨*label*⟩ | '`node()`'
⟨*variable*⟩	::=	'`$`'⟨*identifier*⟩

Table 30. Syntax of composition-free XQuery

4.3.3 Semantics

The formal semantics for composition-free XQuery is, for the most part, closely aligned with the one for XPath discussed above. Again we considered an XML tree a relational structure T over the schema $((\mathsf{Lab}^{\lambda})_{\lambda\in\Sigma}, R_{\mathsf{child}}, R_{\mathsf{nest\text{-}sibling}}, \mathsf{root})$. The nodes of this tree are labeled using the symbols from σ which are queried using $\mathfrak{L}^{\lambda}$ (note, that λ is a single label not a label set as in the graph relations of Chapter 1). The parent-child relations are represented by R_{child}. The order between siblings is represented by $R_{\mathsf{nest\text{-}sibling}}$. The root node of the tree is identified by root. It is easy to see that this view of XML trees (which is as in [9] or [47]), makes an XML-tree a specific instance of a CIQLog data graph, cf. Chapter 1. There are some additional derived relations, viz. $R_{\mathsf{descendant}}$, the transitive, $R_{\mathsf{descendant\text{-}or\text{-}self}}$ the transitive reflexive closure of R_{child}, $R_{\mathsf{following\text{-}sibling}}$, the transitive closure of $R_{\mathsf{next\text{-}sibling}}$, R_{self} relating each node to itself, and $R_{\mathsf{following}}$ the composition of $R^{-1}_{\mathsf{descendant\text{-}or\text{-}self}} \circ R_{\mathsf{following\text{-}sibling}} \circ R_{\mathsf{descendant\text{-}or\text{-}self}}$. Finally, we can compare nodes based on their label using $\cong$ which contains all pairs of nodes with same label. In addition to the XPath relations, XQuery also considers two more forms of equality: one based on node identity, $=_{\mathsf{nodes}}$ which relates each node to itself, and deep equality $=_{\mathsf{deep}}$ which holds for two nodes if there exists an isomorphism between their respective sub-trees.

For example, the XML document `<a>`$_1$ `<b/>`$_2$ `<c>`$_3$`<c/>`$_4$`</c> </a>` (denoting node id's by integer subscripts) is represented as $T = (\mathsf{Lab}^{a} = \{1\}, \mathsf{Lab}^{b} = \{2\}, \mathsf{Lab}^{c} = \{3,4\}, R_{\mathsf{child}} = \{(1,2),(1,3),(3,4)\}, R_{\mathsf{nest\text{-}sibling}} = \{(2,3)\}, \mathsf{root} = \{1\})$ over the label alphabet $\{a, b, c\}$. All other relations can be derived from this definition, see also Chapter 2.

In the following, we also allow unions of such structures, i.e., XML "forests". The semantics of a composition-free XQuery expression is then defined, following [8], using $[\![\;]\!]$ over a given such forest and a list of nodes from that forest $\vec{e} = [e_1, \dots, e_n]$ that represent bindings for variables $x_1, \dots, x_n$. For that, we assume that all variables are first renamed to x_i such that i is the number of variables in whose scope x_i is declared and assuming that `$inp` is scoped over the entire query. E.g., the query

```
for $x in $inp/child::a return
   for $y in $x/child::b return $x
for $z in $inp/child::c return
   for $v in $inp/child::d return $v
```

becomes

```
for $2 in $1/child::a return
   for $3 in $2/child::b return $2
for $2 in $1/child::c return
   for $3 in $1/child::d return $3
```

In the following, we assume that queries are in the latter form.

Table 31 specifies the semantics of composition-free XQuery on an XML forest F and a binding vector $\vec{e} = [e_1, \dots, e_n]$ which is initially of length 1 containing bindings for `$inp`, i.e., usually one (or more, if querying XML collections) root node(s).

The semantics uses three auxiliary notions. **(1)** $\uplus$ is the union on pairs of XML forests and binding vectors such that $(F_1, \vec{e}_1) \uplus (F_2, \vec{e}_2) = (F_1 \cup F_2, \vec{e}_1 \circ \vec{e}_2)$ where $\circ$ is list (or vector) concatenation and the union of XML forests is defined component by component. **(2)** $\Cap$ is the intersection on pairs of XML forests and binding vectors such that $(F_1, \vec{e}_1) \Cap (F_2, \vec{e}_2) = (F_1, [e_i \in \vec{e}_1 : e_i \in \vec{e}_2])$. Note, that we only preserve F_1 (and thus $\Cap$ is *not* associative). However, for the purpose of the semantics the choice of the XML forest is arbitrary as $\Cap$ is only used for the semantics of conditions for which only the existence or non existence (and not their actual value) of bindings is relevant for the semantics of the full query. **(3)** $\mathsf{construct}(l, (F, [w_1, \dots, w_n]))$ denotes construction of a new tree where l is a label, F is an XML forest and $[w_1, \dots, w_n]$ is a vector of nodes in F. It returns a pair $(F \cup T', [\mathsf{root}(T')])$ where T' is a tree over a new set of nodes whose root $\mathsf{root}(T')$ is labeled with l and with the i-th subtree of $\mathsf{root}(T')$ isomorphic to the sub-tree rooted at w_i in F. Furthermore construct is assumed to return a tree with a distinct set of nodes each time it is called. This corresponds to *value invention* in CIQLog.

Using these definitions, the semantics is fairly straightforward. In [8], Benedikt and Koch point out that most of the condition expressions (cases 10, 12–16) can be reduced to other XQuery expressions and thus do not need to be addressed in the semantics. We choose to give their definitions directly as the resulting expressions are no longer in composition-free XQuery.

The crucial parts of the semantics are cases 2 and 3, that illustrate element construction, case 7 that illustrates iteration, and case 8, the semantics of conditionals. The other cases are very similar to XPath and mostly just return appropriate binding vectors but leave F unchanged. Element construction (case 2 and 3) is achieved using the aforementioned construct function and returns a forest containing the newly constructed tree and bindings pointing to that tree's root node. Iteration using **for** has almost exactly the same semantics as the path separator / in XPath: the **return** expression is evaluated in the context of the **in** part, just like the subordinate path is evaluated in the context of the superordinate one. Indeed, the XQuery normalization transforms path expressions consisting of multiple steps to **for** loops as in composition-free XQuery. The difference is, of course, that the semantics of the **return** may be nodes

$$
\begin{aligned}
&[\![\, ()\,]\!]\,(F,\vec{e}) &&= (F,[\,])\\
&[\![\, \texttt{<}l\texttt{>}\; q\; \texttt{</}l\texttt{>}\,]\!]\,(F,\vec{e}) &&= \mathsf{construct}(l, [\![\, q\,]\!]\,(F,\vec{e}))\\
&[\![\, \texttt{<lab(\$}x_i\texttt{)>}\; q\; \texttt{</lab(\$}x_i\texttt{>}\,]\!]\,(F,[e_1,\dots,e_n]) &&\\
& &&= \mathsf{construct}(\texttt{lab}(e_i), [\![\, q\,]\!]\,(F,[e_1,\dots,e_n]))\\
&[\![\, \$x_i\,]\!]\,(F,[e_1,\dots,e_n]) &&= (F,[e_i])\\
&[\![\, \$x_i/axis\texttt{::}l\,]\!]\,(F,[e_1,\dots,e_n]) &&= (F,[d : R_{axis}(e_i,d) \wedge \mathsf{Lab}^l(d)]\\
&\Big[\!\!\Big[\, q_1\, q_2\, \Big]\!\!\Big]\,(F,\vec{e}) &&= \Big[\!\!\Big[\, query_1\, \Big]\!\!\Big]\,(F,\vec{e}) \uplus \Big[\!\!\Big[\, query_1\, \Big]\!\!\Big]\,(F,\vec{e})\\
&[\![\, \texttt{for}\ \$x_i\ \texttt{in}\ s\ \texttt{return}\ q\,]\!]\,(F,\vec{e}) &&= \biguplus_{l\in\vec{l}} [\![\, q\,]\!]\,(F,\vec{e}\cdot l) \text{ where } (F,\vec{l}) = [\![\, s\,]\!]\,(F,\vec{e})\\
&[\![\, \texttt{if}\ cond\ \texttt{then}\ q\,]\!]\,(F,\vec{e}) &&= \begin{cases} [\![\, query\,]\!]\,(F,\vec{e}) & \text{if } \pi_2([\![\, cond\,]\!]\,(F,\vec{e})) \neq [\,]\\ (F,[\,]) & \text{otherwise}\end{cases}\\
&[\![\, \$x_i/axis\texttt{::node()}\,]\!]\,(F,[e_1,\dots,e_n]) &&= (F,[d : R_{axis}(e_i,d)])\\
&[\![\, \texttt{some}\ \$x_i\ \texttt{in}\ s\ \texttt{satisfies}\ c\,]\!]\,(F,\vec{e}) &&= [\![\, \texttt{for}\ \$x_i\ \texttt{in}\ s\ \texttt{return}\ c\,]\!]\,(F,\vec{e})\\
&\Big[\!\!\Big[\, \$x_i = \$x_j\, \Big]\!\!\Big]\,(F,[e_1,\dots,e_n]) &&= \begin{cases} (F,[e_i]) & \text{if } e_i = e_j\\ (F,[\,]) & \text{otherwise}\end{cases}\\
&[\![\, \$x_i = \texttt{<}l\texttt{/>}\,]\!]\,(F,[e_1,\dots,e_n]) &&= \begin{cases} (F,[e_i]) & \text{if = atomic equal and } \mathsf{Lab}^l(e_i)\\ (F,[e_i]) & \text{if = deep equal, } \mathsf{Lab}^l(e_i),\\ & \text{and } \nexists\, d : R_{\mathsf{child}}(e_i,d)\\ (F,[\,]) & \text{otherwise}\end{cases}\\
&[\![\, c_1\ \texttt{or}\ c_2\,]\!]\,(F,\vec{e}) &&= [\![\, c_1\,]\!]\,(F,\vec{e}) \uplus [\![\, c_2\,]\!]\,(F,\vec{e})\\
&[\![\, c_1\ \texttt{and}\ c_2\,]\!]\,(F,\vec{e}) &&= [\![\, c_1\,]\!]\,(F,\vec{e}) \Cap [\![\, c_2\,]\!]\,(F,\vec{e})\\
&[\![\, \texttt{not}\ c\,]\!]\,(F,\vec{e}) &&= \begin{cases} (F,[\mathsf{root}(F)]) & \text{if } (F',[\,]) = [\![\, c\,]\!]\,(F,\vec{E})\\ (F,[\,]) & \text{otherwise}\end{cases}\\
&[\![\, \texttt{true}\,]\!]\,(F,\vec{e}) &&= (F,[\mathsf{root}(F)])
\end{aligned}
$$

Table 31. Semantics for composition-free XQuery (following [8])

from a newly constructed tree. It is crucial that this is the case *only* for the semantics of the **return** expression, not for that of the **in** expression which never modifies the given XML forest. In full XQuery, this does not hold, the **in** is followed by an arbitrary expression. Finally, conditionals are (again reminiscent of qualifiers in XPath) translated using a non-empty test on the bindings returned by the condition.

Note that the *relations of the input forest are never changed*. We may add new forests, but those do not have any relations to the input forest.

It is worth noting, that the semantics is uniform for boolean-valued conditions and for node-valued expressions (in contrast to the XPath case in Section 4.2). This follows [8] and allows a more compact definition of the semantics, at the cost of slightly surprising definitions for boolean operations and true in the latter part of the semantics. In the translation, we separate boolean-valued conditions from other expressions by a separate translation function as in the XPath case.

4.3.4 Translation

For the translation of composition-free XQuery, the main challenge lies in the "constructive" part of composition-free XQuery not in the selection part. In fact, compared to XPath, the selection part is enriched by only three significant features: the ability to use variables and thus to refer back to previously established bindings, the presence of deep-equal, and the ability to sequence expressions and thus their results. Moreover, the latter is the only feature that is challenging for the translation as it requires somewhat more sophisticated management of sibling order than in the translation for XPath. In a sense, this is already part of the constructive part, i.e., element construction and the translation of the results of queries contained in element construction. This is what ends up in the head of a CIQLog rule (which in the XPath case above is always a single atom over the unary answer variable).

Consider again the XQuery example from above:

```
<paperlist>
 for $a in $inp/descendant::author return
   <author> for $p in $inp/descendant::paper return
     if some $x in $p/child::author satisfies deep-equal($x, $a)
     then $p
   </author>
</paperlist>
```

What is the result of this query, if there is no author in the document? What if there is an author with no paper? In XPath, if any part of a path (disregarding **or** for the moment) has no match the entire query has no match. In XQuery, this is not the case. The above example *always* yields at least a paperlist element. It may be empty, if there are no authors in the document but it may never be absent. The some for the inner loop: The author element is constructed and included in the result for any author in the input even if there is no paper for an author. Of course, we change this behavior by placing additional **if** clauses. But in general, an XQuery expression may always return an empty set of nodes, but never causes other expressions not contained in it to fail.

Translation example. Continuing with this example, how can we express the same query as a CIQLog (in fact, CIQLog^{NR}) rule? The following CIQLog rule shows the answer to that question:

```
1 $\mathrm{root}(\mathrm{id}_1(i)) \land \mathfrak{D}(\mathrm{id}_1(i),\mathrm{CHILD}) \land$
  $\mathrm{CHILD}(\mathrm{id}_1(i),\mathrm{id}_2(i),\underline{\mathrm{order}}(\top,0)) \land \mathfrak{L}(\mathrm{id}_2(i),\mathrm{paperlist}) \land \mathfrak{D}(\mathrm{id}_2(i),\mathrm{CHILD}) \land$
3 if $v_1 \neq$ nil then $\big($
    $\mathrm{CHILD}(\mathrm{id}_2(i),\mathrm{id}_3(i,v_1),\underline{\mathrm{order}}(\underline{\mathrm{order}}(\top,0),0,v_1)) \land \mathfrak{L}(\mathrm{id}_3(i,v_1),\ \mathrm{author}) \land$
5   if $v_2 \neq$ nil then (
      if $v_3 \neq$ nil then (
7       $\mathrm{CHILD}(\mathrm{id}_3(i,v_1),\ \mathrm{id}_4(i,v_1,v_2),\underline{\mathrm{order}}(\underline{\mathrm{order}}(\top,0),0,v_2)) \land$
        $\mathrm{deep\text{-}copy}(\mathrm{id}_4(i,v_1,v_2),v_2)$ ) ) $\big)$
9 $\longleftarrow$ $\mathrm{root}(i) \land \big(\mathrm{CHILD}_+(i,\ v_1) \land \mathfrak{L}(v_1,\ \mathrm{author}) \land$
     $(\mathrm{CHILD}_+(i,\ v_2) \land \mathfrak{L}(v_2,\ \mathrm{paper}) \land$
11     $(\mathrm{CHILD}(v_2,\ v_3) \land \mathfrak{L}(v_3,\ \mathrm{author}) \land v_3 \triangleq_{bij} v_1$
         $\lor\ v_3 =$ nil)
13     $\lor\ v_2 =$ nil)
     $\lor\ v_1 =$ nil$\big)$
```

The same abbreviations of head formulas as in the translation of Xcerpt in Chapter 3 are used:

```
$\mathrm{CHILD}(\mathrm{id}_1(\vec{x}_1),\mathrm{id}_2(\vec{x}_2),o)$
```

abstracts the edge construction necessary in CIQLog and thus is an abbrevation for

```
$\circ\!\!\rightarrow(\mathrm{id}_1(\vec{x}_1),\mathrm{id}_N(\vec{x}_2)) \land \rightarrow\!\!\circ(\mathrm{id}_2(\vec{x}_2),\mathrm{id}_N(\vec{x}_2)) \land \mathrm{pos}(\mathrm{id}_N(\vec{x}_2),\underline{\mathrm{order}}(\top,0))$
```

We also implicitly omit the document order $<$ as parameter for all order terms. That is, two binding vectors for the same query variables are ordered by looking at the bindings of the query variables in sequence and considering their relative position in document order $<$ (which is provided, e.g., by $\mathrm{CHILD}_+(n,n') \lor \blacktriangleleft^{\mathrm{CHILD}}_+(n,n')$, i.e., n is before n' if it is a ancestor of n' or before n' in $\blacktriangleleft^{\mathrm{CHILD}}_+$.

The most notable difference to the translations for XPath or even Xcerpt is the extraordinary amount of conditional construction (and corresponding $\lor v =$ **nil** for some variable v). As discussed above, this is due to the nature of XQuery expressions where non matching sub-expression often do not affect the matching of their superordinate expressions. Each of the conditionals "guards" one sub-expression and ensures that it is omitted from the result if there are no bindings for the guard variable (but without affecting the remainder of the head construction).

The other striking feature of the translation are the unusually (and at first glance, unnecessarily) complex order terms (e.g., line 4 and line 7). They are necessary to allow arbitrary occurrences of **for** loops (and thus arbitrarily long sequences of constructed elements) to occur in arbitrary positions in sequences of XQuery expressions within the same element constructor. First, notice, that the order terms in line 4 and 7 are the same. This is possibly since order is only relevant between edges with the same source and these order terms are on edges with different source. In the translation below this is reflected by "resetting" the nesting of order terms at any element construction (cases 2, 3 in Table 34).

To further illustrate the need for nested order terms, let's add another XQuery expression before the outer **for** loop but within the paperlist element:

```
<paperlist>
  <abc>()</abc>
  for $a in $inp/descendant::author return
    <author> for $p in $inp/descendant::paper return
      if some $x in $p/child::author satisfies deep-equal($x, $a)
      then $p
    </author>
</paperlist>
```

The body of the resulting CIQLog rule is unchanged, but in the head we have to construct the new element, but also to adapt the order terms (omitting the unchanged body):

```
  root(id₁(i)) ∧ 𝔒(id₁(i),CHILD) ∧
2   CHILD(id₁(i),id₂(i),order(⊤,0)) ∧ 𝔏(id₂(i),paperlist) ∧ 𝔒(id₂(i),CHILD) ∧
    CHILD(id₂(i),idₙ(i),order(⊤,0)) ∧ 𝔏(idₙ(i),abc) ∧ 𝔒(idₙ(i),CHILD) ∧
4   if v₁ ≠ nil then (
      CHILD(id₂(i),id₃(i,v₁),order(order(⊤,1),0,v₁)) ∧ 𝔏(id₃(i,v₁), author) ∧
6     if v₂ ≠ nil then (
        if v₃ ≠ nil then (
8         CHILD(id₃(i,v₁), id₄(i,v₁,v₂),order(order(⊤,0),0,v₂)) ∧
          deep-copy(id₄(i,v₁,v₂),v₂) ) ) )
```

In line 5, we no longer use offset 0 but offset 1. That is $t_1 = \text{order}(\top, 0)$, the order term for the new abc, is always smaller (wrt. the order on order invention terms defined in Section 2.2.1) than $t_2 = \text{order}(\text{order}(\top, 1), 0, v_1))$ regardless of the binding for v_1. Here, that is the case as the parent order term of t_2 is the same as t_1 up to the offset which is higher and thus $t_1 < t_2$.

Similar changes occur if the added element is at the end or a **for** loop (in which case the parent order terms of the elements created by the original and by the new **for** loop are the same, except for the offset).

Instead of adding abc before the loop, we might also want to add it within the loop:

```
<paperlist>
  for $a in $inp/descendant::author return
    <abc>()</abc>
    <author> for $p in $inp/descendant::paper return
      if some $x in $p/child::author satisfies deep-equal($x, $a)
      then $p
    </author>
</paperlist>
```

Again, this can be addressed by adapting the order terms, in this case the new element shares the same order term as the author element contained in the loop, but with a smaller offset:

```
root(id₁(i)) ∧ 𝔒(id₁(i),CHILD) ∧
  CHILD(id₁(i),id₂(i),order(⊤,0)) ∧ 𝔏(id₂(i),paperlist) ∧ 𝔒(id₂(i),CHILD) ∧
  if v₁ ≠ nil then (
    CHILD(id₂(i),idₙ(i,v₁),order(order(⊤,0),0,v₁)) ∧ 𝔏(idₙ(i,v₁),abc) ∧
        𝔒(idₙ(i,v₁),CHILD) ∧
    CHILD(id₂(i),id₃(i,v₁),order(order(⊤,0),1,v₁)) ∧ 𝔏(id₃(i,v₁), author) ∧
    if v₂ ≠ nil then (
      if v₃ ≠ nil then (
        CHILD(id₃(i,v₁), id₄(i,v₁,v₂),order(order(⊤,0),0,v₂)) ∧
        deep-copy(id₄(i,v₁,v₂),v₂) ) ) )
```

The nesting level of order terms increases only if **for** loops are contained within each other without intermediate element construction as in the final example (where we delete the author around the inner loop).

```
<paperlist>
 for $a in $inp/descendant::author return
   <abc>()</abc>
   for $p in $inp/descendant::paper return
     if some $x in $p/child::author satisfies deep-equal($x, $a)
     then $p
</paperlist>
```

Now the order terms for the elements created by the inner loop depend on the order terms for the outer loop. It is still ensured, that the order term for the abc is before the order terms of all the elements in the inner loop since it is the same as their parent order term except for the offset which is smaller.

```
root(id₁(i)) ∧ 𝔒(id₁(i),CHILD) ∧
  CHILD(id₁(i),id₂(i),order(⊤,0)) ∧ 𝔏(id₂(i),paperlist) ∧ 𝔒(id₂(i),CHILD) ∧
  if v₁ ≠ nil then (
    CHILD(id₂(i),idₙ(i,v₁),order(order(⊤,0),0,v₁)) ∧ 𝔏(idₙ(i,v₁),abc) ∧ 𝔒(idₙ(i,v₁),CHILD) ∧
    if v₂ ≠ nil then (
      if v₃ ≠ nil then (
        CHILD(id₂(i), id₃(i,v₁,v₂),order(order(order(⊤,0),1,v₁),0,v₂)) ∧
        deep-copy(id₃(i,v₁,v₂),v₂) ) ) )
```

Translation function. The actual translation of composition-free XQuery expressions to CIQLog is specified by $\text{tr}_{\text{XQuery}}$. As for the translation of Xcerpt, we use an environment $\mathcal{E}$ containing mappings from XQuery variables to CIQLog variables and the list of current iteration variables. In addition, we use $\mathcal{O}$ to hold information about the current order term. $\mathcal{O}$ is a triple $\langle g : \textit{term}, o : \textit{offset}, i : \textit{order vars}\rangle$ where g is the current parent order term, o the current offset, and i the list of order variables. We use $\mathcal{O}.g$, $\mathcal{O}.o$, and $\mathcal{O}.i$ to refer to each of the components. If there is no parent order term, we use the canonical empty order term $\top$ from Section 2.2.1. Finally, we write $\text{order}(\mathcal{O})$ as abbreviation for $\text{order}(\mathcal{O}.g, \mathcal{O}.o, \mathcal{O}.i)$.

Given an XQuery expression P, $\text{tr}_{\text{XQuery}}$ computes a corresponding CIQLog expression as follows:

$$\begin{aligned}
\text{tr}_{\text{XQuery}}\langle P\rangle = {} & \text{root}(r_c) \wedge \mathfrak{O}^{\text{CHILD}}(r_c) \wedge C \longleftarrow \text{root}(r_q) \wedge Q \\
& \textbf{where } r_q \text{ be a new variable, } r_c = \text{id}(r_q) \text{ with id a new identifier} \\
& \qquad (C, Q) = \text{tr}_{\text{XQuery}}(\mathcal{E}, (\top, 0, []), r_c)\langle P\rangle \text{ with} \\
& \qquad (\texttt{inp}, r_q) \in \mathcal{E} \text{ and } \mathcal{E}.\text{iter} = [r_q].
\end{aligned}$$

As for Xcerpt, this is mainly a wrapper adding root restrictions and root construction around the result of the actual translation which is computed by $\text{tr}_{\text{XQuery}}$ with a new environment initialized to contain a mapping for the canonical XQuery input variable `$inp` to the root of the XML document and an iteration sequence containing only the CIQLog variable corresponding to `$inp`. Thus, for each root node of the input document (of which there is exactly one, if the input is an XML tree, but may be several if we allow XML forests) a (distinct) result for P is computed. The order environment $\mathcal{O}$ is initialized with $(\top, 0, [])$, i.e., the empty order term, offset 0, and no iteration variables. Finally, we also pass r_c the parent *construct* *variable* to $\mathrm{tr}_{\mathrm{XQuery}}$

functionXQuery	= CIQLog **expression**
$\mathrm{tr}_{\mathrm{XQuery}}(\mathcal{E},\mathcal{O},p)\langle()\rangle$	$= (\top,\top)$
$\mathrm{tr}_{\mathrm{XQuery}}(\mathcal{E},\mathcal{O},p)\langle\texttt{<}l\texttt{>}\ q\ \texttt{</}l\texttt{>}\rangle$	$= (\mathfrak{L}(v,l) \wedge \mathrm{CHILD}(p,v,\mathrm{order}(\mathcal{O})) \wedge \mathfrak{D}^{\mathrm{CHILD}}(v) \wedge C, Q)$ **where** $v = \mathrm{id}(\mathcal{E}.\mathrm{iter})$ and id a new identifier $(C,Q) = \mathrm{tr}_{\mathrm{XQuery}}(\mathcal{E},(\top,0,[]),p)\langle q\rangle$
$\mathrm{tr}_{\mathrm{XQuery}}(\mathcal{E},\mathcal{O},p)\langle\texttt{<lab(\$}x_i\texttt{)>}\ q\ \texttt{</lab(\$}x_i\texttt{>}\rangle$	$= (\cong(\mathcal{E}(x_i),v) \wedge \mathrm{CHILD}(p,v,\mathrm{order}(\mathcal{O})) \wedge \mathfrak{D}^{\mathrm{CHILD}}(v) \wedge C, Q)$ **where** $v = \mathrm{id}(\mathcal{E}.\mathrm{iter})$ and id a new identifier $(C,Q) = \mathrm{tr}_{\mathrm{XQuery}}(\mathcal{E},(\top,0,[]),p)\langle q\rangle$
$\mathrm{tr}_{\mathrm{XQuery}}(\mathcal{E},\mathcal{O},p)\langle\texttt{\$}x_i\rangle$	$= (\text{deep-copy}(\mathcal{E}(x_i),v) \wedge \mathrm{CHILD}(p,v,\mathrm{order}(\mathcal{O})),\top)$ **where** $v = \mathrm{id}(\mathcal{E}.\mathrm{iter})$ and id a new identifier
$\mathrm{tr}_{\mathrm{XQuery}}(\mathcal{E},\mathcal{O},p)\langle\texttt{\$}x_i/\mathit{step}\rangle$	$= (\textbf{if}\ r \neq \textbf{nil}\ \textbf{then}\ \text{deep-copy}(r,v) \wedge \mathrm{CHILD}(p,v,\mathrm{order}(\mathcal{O})),(Q_s \vee r = \textbf{nil}))$ **where** $(Q_s,r) = \mathrm{tqc}(\mathcal{E}')\langle\texttt{\$}x_i/\mathit{step}\rangle$ with $\mathcal{E}' = \mathcal{E}$ but $\mathcal{E}'.\mathrm{iter} = \mathcal{E}.\mathrm{iter} \circ r$ $v = \mathrm{id}(\mathcal{E}.\mathrm{iter} \circ r)$ and id a new identifier
$\mathrm{tr}_{\mathrm{XQuery}}(\mathcal{E},\mathcal{O},p)\langle q_1\ q_2\rangle$	$= (C_1 \wedge C_2, Q_1 \wedge Q_2)$ **where** $(C_1,Q_1) = \mathrm{tr}_{\mathrm{XQuery}}(\mathcal{E},\mathcal{O},p)\langle q_1\rangle$ $(C_2,Q_2) = \mathrm{tr}_{\mathrm{XQuery}}(\mathcal{E},(\mathcal{O}.g,\mathcal{O}.o+1,\mathcal{O}.i),p)\langle q_2\rangle$ query sequence is left-associative, i.e., q_1 no sequence
$\mathrm{tr}_{\mathrm{XQuery}}(\mathcal{E},\mathcal{O},p)\langle\texttt{for \$}x_i\ \texttt{in}\ s\ \texttt{return}\ q\rangle$	$= (\textbf{if}\ r \neq \textbf{nil}\ \textbf{then}\ C,(Q_s \wedge Q \vee r = \textbf{nil}))$ **where** $(Q_s,r) = \mathrm{tqc}(\mathcal{E})\langle s\rangle$ with $\mathcal{E}' = \mathcal{E}$ but $\mathcal{E}'.\mathrm{iter} = \mathcal{E}.\mathrm{iter} \circ r$ $(C,Q) = \mathrm{tr}_{\mathrm{XQuery}}(\mathcal{E}' + (x_i,r),(\mathrm{order}(\mathcal{O}),0,r),p)\langle q\rangle$
$\mathrm{tr}_{\mathrm{XQuery}}(\mathcal{E},\mathcal{O},p)\langle\texttt{if}\ \mathit{cond}\ \texttt{then}\ \mathrm{q}\rangle$	$= (\textbf{if}\ r \neq \textbf{nil}\ \textbf{then}\ C,(Q_c \wedge Q \vee r = \textbf{nil}))$ **where** $(Q_c,r) = \mathrm{tqc}(\mathcal{E})\langle \mathit{cond}\rangle$ $(C,Q) = \mathrm{tr}_{\mathrm{XQuery}}(\mathcal{E},\mathcal{O},p)\langle q\rangle$

Table 34. Translating composition-free XQuery

The $\mathrm{tr}_{\mathrm{XQuery}}(\mathcal{E},\mathcal{O},p)$ is specified in full in Table 34. As stated, it takes an environment $\mathcal{E}$, an order environment $\mathcal{O}$, a parent construct variable, and, of course, the XQuery expression as parameter. It returns a pair (C,Q) of CIQLog formulas, C a conjunction of head atoms, Q a body formula. The intuitive semantics of $\mathrm{tr}_{\mathrm{XQuery}}$ is that the result of C applied to the bindings provided by Q (i.e., $C \longleftarrow Q$) is isomorphic to the result of the XQuery expression under the given $\mathcal{E}$, $\mathcal{O}$, and parent construct variable (resp. its XQuery counterpart).

The translation uses a helper function tqc for the translation of XQuery conditions which is given in Table 36. Before turning to tqc, note the structural similarities between $\mathrm{tr}_{\mathrm{XQuery}}$ and the above semantics for composition-free XQuery (which is due to [47, 8]). In contrast to that semantics, however, the translation has to split up the XQuery expression into a specification of the construction (for the CIQLog rule head) and a CIQLog query expressing relations on the nodes of the input tree. This split makes the nature of XQuery expressions, whether they are mainly about querying the input nodes or about creating output, eminently visible in $\mathsf{tr}_{\mathsf{XQuery}}$. E.g., element construction (cases 2–3) does not affect the query (the Q part), but only the construction. On the other hand, all the conditions (in Table 36) used, e.g., for the **in** expression of **for** loops (case 7) only result in query formula and have influence on the construction. Regarding, the order environment $\mathcal{O}$ and its management, it is worth pointing out, that element construction resets that environment (cases 2–3) by using a $(\top, 0, [])$ for the translation of contained expressions. In contrast, iteration (case 7) adds to the nesting depth of order terms of contained expressions by using $(\mathsf{order}(\mathit{order}(\mathcal{O}), o, r)$ as order environment for the translation of nested expressions. Finally, a sequence of queries (case 6) translates the first query with the given order environment, but increases the offset for each following query. For case 6, we assume that query sequence operator is left-associative, i.e., q_1 does not consist in a sequence of two other queries.

As stated, conditions are translated using tqc specified in Table 36, which also takes an environment $\mathcal{E}$ in addition to the condition to be translated and returns pairs of CIQLog body formulas and query variables. The returned query variable identifies a newly created variable, if there is any. It uses the same helper function $\mathsf{relation}(\mathit{axis})$ as the translation of XPath, see Section 4.2.2.

In one aspect, we deviate slightly from XQuery and the above semantics for composition-free XQuery: we always construct *new* forest whereas XQuery expressions may return simply lists of bindings into the existing data. In other words, we implicitly assume a root element around any XQuery expression to be translated. Without this assumption the result computed by our translation is only correct up to node identity wrt. the above semantics. This can be addressed by distinguishing between expressions within (at least one) element constructors and those outside of any element constructors. For the latter, we use, instead of deep-copy, direct references to variables and their bindings in the head.

Order Example. We conclude with a further illustration of the role of order terms in XQuery. Consider the following XQuery program that generates a list of a, b,c, and d tags under a common root r, but the order and number of those tags depends on the number of bindings for `$inp/descendant::*`:

```
<r>
 for $x in $inp/descendant::* return
   <a>()</a>
   for $y in $inp/descendant::* return
     for $z in $inp/descendant::* return
       <b>()</b> <c>()</c>
   <d>()</d>
</r>
```

Assuming, for instance, that `$inp/descendant::*`: matches exactly two nodes in the input, the resulting XML document looks as follows:

```
<r>
 <a />
   <b /> <c /> <b /> <c /> <b /> <c /> <b /> <c />
 <d />
 <a />
   <b /> <c /> <b /> <c /> <b /> <c /> <b /> <c />
```

functionXQuery	= CIQLog **expression**
$\mathrm{tqc}(\mathcal{E})\langle \texttt{\$}x_i/\mathit{axis}\texttt{::}l\rangle$	$= (\mathrm{relation}(\mathit{axis})(\mathcal{E}(x_i), v) \wedge \mathfrak{L}(v, l), v)$ **where** $v = \mathrm{id}(\mathcal{E}.\mathrm{iter})$ and id new identifier
$\mathrm{tqc}(\mathcal{E})\langle \texttt{\$}x_i/\mathit{axis}\texttt{::node()}\rangle$	$= (\mathrm{relation}(\mathit{axis})(\mathcal{E}(x_i), v)$ **where** $v = \mathrm{id}(\mathcal{E}.\mathrm{iter})$ and id a new identifier
$\mathrm{tqc}(\mathcal{E})\langle \texttt{\$}x_i = \texttt{\$}x_j\rangle$	$= \begin{cases} (\mathcal{E}(x_i) \doteq \mathcal{E}(x_j), \mathcal{E}(x_i)) & \text{if = is node equal} \\ (\mathcal{E}(x_i) \cong \mathcal{E}(x_j), \mathcal{E}(x_i)) & \text{if = is atomic equal} \\ (\mathcal{E}(x_i) \overset{\circ}{=}_{bij} \mathcal{E}(x_j), \mathcal{E}(x_i)) & \text{if = is deep equal} \end{cases}$
$\mathrm{tqc}(\mathcal{E})\langle \texttt{\$}x_i = \texttt{<}l\texttt{/>}\rangle$	$= \begin{cases} (\texttt{false}, r) & \text{if = is node equal and } r \text{ new var.} \\ (\mathfrak{L}(\mathcal{E}(x_i), l), \mathcal{E}(x_i)) & \text{if = is atomic equal} \\ (\mathfrak{L}(\mathcal{E}(x_i), l) \wedge \mathrm{outdeg}(\mathcal{E}(x_i), 0), \mathcal{E}(x_i)) & \text{if = is deep equal} \end{cases}$
$\mathrm{tqc}(\mathcal{E})\langle q_1 \texttt{ and } q_2\rangle$	$= (Q_1 \wedge Q_2 \wedge r = (r_1 \neq \texttt{nil} \wedge r_2 \neq \texttt{nil}), r)$ **where** r is a new variable $(Q_1, r_1) = \mathrm{tqc}(\mathcal{E})\langle q_1\rangle$ $(Q_2, r_2) = \mathrm{tqc}(\mathcal{E})\langle q_2\rangle$
$\mathrm{tqc}(\mathcal{E})\langle q_1 \texttt{ or } q_2\rangle$	$= ((Q_1 \vee Q_2) \wedge r = (r_1 \neq \texttt{nil} \vee r_2 \neq \texttt{nil}), r)$ **where** r is a new variable $(Q_1, r_1) = \mathrm{tqc}(\mathcal{E})\langle q_1\rangle$ $(Q_2, r_2) = \mathrm{tqc}(\mathcal{E})\langle q_2\rangle$
$\mathrm{tqc}(\mathcal{E})\langle \texttt{not } q\rangle$	$= (\neg(Q), r)$ **where** $(Q, r) = \mathrm{tqc}(\mathcal{E})\langle q\rangle$
$\mathrm{tqc}(\mathcal{E})\langle \texttt{true}\rangle$	$= (r = \texttt{true}, r)$ **where** r is a new variable ($\texttt{true}$ arbitrary not-$\texttt{nil}$ value)
$\mathrm{tqc}(\mathcal{E})\langle \texttt{some } \texttt{\$}x_i \texttt{ in } s \texttt{ satisfies } c\rangle$	$= ((Q_s \wedge Q \vee r = \texttt{nil}), r')$ **where** $(Q_s, r) = \mathrm{tqc}(\mathcal{E})\langle s\rangle$ with $\mathcal{E}' = \mathcal{E}$ but $\mathcal{E}'.\mathrm{iter} = \mathcal{E}.\mathrm{iter} \circ r$ $(Q, r') = \mathrm{tqc}(\mathcal{E}' \mathbin{+\!\!+} (x_i, r))\langle c\rangle$

Table 36. Translating composition-free XQuery: conditions

```
  <d />
</r>
```

The query returns as many sequences of a and d elements surrounding sequences of b and c elements as there are matches for `$inp/descendant::*`.The inner sequence of b's and c's is also repeated once for each match. In CIQLog, we use node invention terms with appropriate grouping variables to obtain as many new nodes as there are matches, and order (invention) terms to ensure that their order is correct. Applying $\mathrm{tr}_{\mathrm{XQuery}}$ to the above program yields:

```
root(id₁(i)) ∧ 𝔒(id₁(i),CHILD) ∧
 CHILD(id₁(i),id₂(i),order(⊤,0)) ∧ 𝔏(id₂(i),r) ∧ 𝔒(id₂(i),CHILD) ∧
 if v₁ ≠ nil then (
   CHILD(id₂(i),id₃(i,v₁),order(order(⊤,0),0,v₁)) ∧ 𝔏(id₃(i,v₁), a) ∧
   if v₂ ≠ nil then ( if v₃ ≠ nil then (
     CHILD(id₂(i),id₄(i,v₁,v₂,v₃),order(order(order(⊤,0),1,v₁),0,v₂,v₃)) ∧
     𝔏(id₄(i,v₁,v₂), b) ∧
     CHILD(id₂(i),id₅(i,v₁,v₂,v₃),order(order(order(⊤,0),1,v₁),1,v₂,v₃)) ∧
     𝔏(id₅(i,v₁,v₂), c) ) ) ∧
   CHILD(id₂(i),id₆(i,v₁),order(order(⊤,0),2,v₁)) ∧ 𝔏(id₆(i,v₁), d) )
⟵ root(i) ∧ (CHILD+(i, v₁) ∧
      (CHILD+(i, v₂) ∧
        (CHILD(v₂, v₃) ∧
          ∨ v₃ = nil)
        ∨ v₂ = nil)
      ∨ v₁ = nil)
```

Notice, how the order terms for the construction of b and c elements have the same parent order term which in turn is between the order term of the a and of the d children of r.

With this example, we conclude the illustration of the translation function for composition-free XQuery expressions to CIQLog. Before a brief outlook on an extension of that translation to a larger fragment of XQuery, the next section discusses the equivalence between the translation function and the above semantics for composition-free XQuery from [8].

4.3.5 Equivalence

To establish equivalence between the semantics of composition-free XQuery after [8] given in Table 31 and the semantic of a CIQLog rule generated by $\mathrm{tr}_{\mathrm{XQuery}}$ for a given composition-free XQuery expression P, first consider that XML forests as used in the semantics are merely a special case of CIQLog data graphs (under the mapping from axis to data graph relations given by relation in Section 4.2.2.

Thus, we have to establish that the data graph constructed by the CIQLog rule resulting from a translation of P is isomorphic to the XML forest F returned by Table 31 on P if restricted to the sub-trees rooted at the binding nodes $\vec{e}$.

For simplicity, we assume in the following that P is wrapped in some root element root. This avoids having to consider multiple binding nodes (and thus forests instead of trees). It also addresses that above remark, that the CIQLog semantics is always a *new* tree, whereas Table 31 allows fragments of an original tree to be returned. However, up to node identity the two forms are equivalent.

Theorem 4.2. *The semantics of the* CIQLog *expression returned by* $\mathsf{tr}_{\mathsf{XQuery}}$ *for a given XQuery expression P is equivalent to the semantics of P under Table 31 up to node identity.*

Proof (Sketch). The proof is by structural induction over the shape of an XQuery expression.

For case 1 (`()`), both semantics do not change the bindings of any variables and do not add any construction. In particular, if `<root>()</root>` is the entire program both semantics are obviously equivalent (both return a tree with single, empty node root (cf. the $\mathsf{tr}_{\mathsf{Xcerpt}}$ wrapper above and case 2 in both semantics).

For case 2 and 3, element construction, add a root node with given label around the bindings returned by the evaluation of their child expressions. The iteration is on bindings in $\vec{e}$, resp. iteration variables in $\mathcal{E}.\mathsf{iter}$ which are in all cases unchanged from the call of the semantics function to nested calls except for the iteration case (case 7), where they are extended in both semantics in the same way, see below.

For case 4, observe that both semantics construct a tree isomorphic to the one rooted at a binding e_i for x_i. However, $[\![\ ]\!]$ returns directly $(F, [e_i])$, whereas $\mathsf{tr}_{\mathsf{XQuery}}$ returns a deep-copy of the subtree rooted at e_i. If we disregard node identity, however, the two subtrees are equivalent.

For case 5, the same observation holds. In addition to case 4, both semantics add restrictions to the returned bindings, as expressed by the path expression. In the case of $\mathsf{tr}_{\mathsf{XQuery}}$ this is achieved by the call to tqc. For $[\![\ ]\!]$ we also need to consider case 9.

For case 6, the sequence of two queries, both semantics delegate the translation to recursive calls on the operands and combine the result using union resp. conjunction. Note, that the involved order terms of $\mathsf{tr}_{\mathsf{XQuery}}$ are covered in $[\![\ ]\!]$ simply by concatenating the bindings returned by the second query after the end of the bindings of the first query (see definition of $\uplus$ in Section 4.3.3.

For case 7, we observe that both semantics delegate the translation, though $\mathsf{tr}_{\mathsf{XQuery}}$ uses tqc for the **`in`** expression. This is nevertheless equivalent, as an **`in`** expression in composition-free XQuery may only contain a step. This is not true of XQuery where exactly this case can not always be translated to a single CIQLog rule. The iteration variables are in both cases extended in the same way (by the single variable bound in the **`in`** expression).

Analog observations hold for case 8.

If one remembers that cases 10–15 of $[\![\ ]\!]$ handle expressions that only occur in conditions (but not in general query contexts), it is easy to verify that they are equivalent to the corresponding cases of tqc. □

4.4 Beyond Composition-free XQuery

In the previous sections, we have considered two important fragments of XQuery, viz. navigational XPath and composition-free XQuery, and shown ho to translate them into $\text{CIQLog}^{\text{NR}}$.

We have chosen composition-free XQuery as it limits all relations to nodes of the input tree, just like in $\text{CIQLog}^{\text{NR}}$. However, in full CIQLog we can also query invented nodes resulting from a previous rule application. We can exploit this to translate a larger fragment of query, viz. XQuery without user-defined functions. Intuitively, each expression is partitioned into a sequence of rules such that a following rule depends only on preceding ones. Whenever we construct data, that is then queried, we introduce a new rule.

Incidentally, this approach is used and described in more detail in [49] where a translation from XQuery to Xcerpt is defined. In fact, we can use that translation to first create an Xcerpt program for a given XQuery expression and then translate that Xcerpt program to CIQLog as described in Section 4.

In both cases, the resulting CIQLog program is non-recursive since each rule depends only on rules generated from nested XQuery expressions. This coincides with results from [47] where it is shown that XQuery without user-defined functions and deep-equal has NEXPTIME-complete query complexity, just as non-recursive CIQLog (recall that non-recursive CIQLog has the same complexity as non-recursive logic programming).

Finally, for full XQuery we have to consider user-defined, possibly recursive functions and the full operator library of XQuery [50]. The prior can be translated by the same scheme as above, but now a rule may depend also on the results of itself or rules generated from superordinate expressions. The latter can be provided as predefined relations or, where possible, implemented as CIQLog rules, and does not add to the expressiveness of full CIQLog.

4.5 Conclusion

CIQLog is designed as an abstraction of the core aspects of Web query languages. Its non-recursive fragment CIQLog$^{\text{NR}}$ is well-suited to specify and implement diverse query languages without query composition. In this chapter, we show how navigational XPath and a large fragment of XQuery, composition-free XQuery as defined in [47], can be translated to CIQLog while preserving the standard semantics.

The translation also highlights one of the core differences between XPath (and, to some extent, Xcerpt) and XQuery: Where we can consider navigational XPath without caring about the precise iteration order and consider the semantics of XPath-expressions as sets of nodes, XQuery provides a much greater control over the iteration order on binding nodes and the order of constructed elements in the result. Though this certainly is beneficial in some cases, there are many queries for which such precise control is pointless. This has been recognized in the design of XQuery through the addition of the **unordered** operator which, essentially, switches from sequence- to (multi-) set-based semantics. Xcerpt takes the dual approach: We assume that in most cases the query author is not all that much concerned about the order of result elements. If that is the case, a grouping expression may be adorned by an explicit **order-by** clause. Another reason for the more involved order terms in the translation of XQuery is that we specifically disallow lists of terms in Xcerpt grouping expressions in Chapter 3. If we allow such expressions as in, e.g., `r[a, all(b, c)group-by(var X), d]`, we arrive at similarly involved order terms as in the translation presented here.

Together with the CIQCAG algebra the above translation gives as an efficient and in many cases optimal evaluation of composition-free XQuery (and navigational XPath). For composition-free XQuery on tree, forest, or even CIG data, we obtain $\mathcal{O}(q \cdot n)$ time and space bounds if the query is tree shaped and $\mathcal{O}(n^{q_g} + q \cdot n)$ where q is the size of the query, n the size of the data and q_g the number of answer variables or variables with non-tree edges in the query. On XML data, the space bound is even $\mathcal{O}(q \cdot d)$. The detailed complexities of CIQCAG are discussed in [34].

Chapter 5

Translating SPARQL

5.1 Introduction

Compared with XML, the field of RDF query languages is less mature and has not received as much attention from research. Recently, the W3C has started to derive a standard RDF query language, called SPARQL [65], that is, visibly influenced by languages such as RDQL [58], RQL[43], and SeRQL[14], aiming to create a stable foundation for use, implementation, and research on RDF databases and query languages. Fundamentally, SPARQL is a fairly simple, first-order-style query language in the spirit of SQL or OQL. However, the specifics of RDF have lead to a number of unusual features that, arguably, make SPARQL more suited to RDF querying than previous approaches such as RDQL [58]. However, the price is a more involved semantics complemented by a tendency in [65] to redefine or ignore established notions from relational and XML query languages rather than build upon them.

Nevertheless, SPARQL is expected to become the "lingua franca" of RDF querying and thus well worth further investigation. In the following sections, we first briefly introduce into SPARQL and its semantics (based on [63] and [64] but extended to full SPARQL queries rather than only patterns). From the discussion of the semantics, we turn to the translation from SPARQL to CIQLog$^{\text{NR}}$ which turns out to be closely aligned with the semantics of [63]. The translation is also the first purely logical semantics for SPARQL which illustrates how to integrate SPARQL with rule-based reasoning approaches prolific on the Semantic Web. We briefly discuss along one such approach, viz. RDFLog, the effects of extending SPARQL with rules on the translation to CIQLog.

5.2 SPARQL Syntax and Semantics in 1000 Words

Example. In Section 1.4, we introduce RDF and its data model together with a mapping to CIQLog data graphs. Recall, the sample data used there, which is depicted again in Figure 17 and represents articles in a given conference and their authors.

The following SPARQL query selects from that graph all articles created by someone with the fullname "M. T. Cicero" and returns a new graph where the dc:creator relation of the original graph is inverted

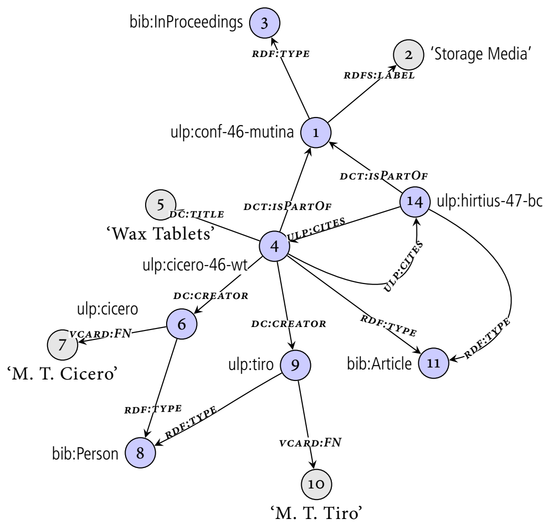


Figure 17. Exemplary Data Graph: RDF Conference Data (simplification of Figure 4 omitting the sequence container and edge positions)

to my:published.[1]

```
CONSTRUCT { ?p my:published ?a }
WHERE { ?a rdf:type bib:Article AND ?a dc:creator ?p
        AND ?p vcard:FN 'M. T. Cicero' }
```

The query illustrates SPARQLs fundamental query construct: a pattern (s, p, o) for RDF triples (whose components are usually thought of as subject, predicate, object). Any RDF triple is also a triple pattern, but triple patterns allow variables for each component. Furthermore, SPARQL also allows literals in subject position, anticipating the same change also in RDF itself. We use the variant syntax for SPARQL discussed in [63] to ease the definition of syntax and semantics of the language. For instance, standard SPARQL, uses `.` instead of AND for triple conjunction. We consider two forms of SPARQL queries, viz. **SELECT** queries that return list of variable bindings and **CONSTRUCT** queries that return new RDF graphs. Triple patterns contained in a **CONSTRUCT** clause (or "template") are instantiated with the variable bindings provided by the evaluation of the triple pattern in the **WHERE** clause. We omit named graphs and assume that all queries are on the single input graph. An extension of the discussion to named graphs is easy (and partially demonstrated in [64]) but only distracts from the salient points of the discussion.

The full grammar of SPARQL queries as considered here (extending [63] by **CONSTRUCT** queries) is as follows:

$$\begin{array}{lll} \langle query \rangle & ::= & \text{`CONSTRUCT'} \ \langle template \rangle \ \text{`WHERE'} \ \langle pattern \rangle \\ & | & \text{`SELECT'} \ \langle variable \rangle + \ \text{`WHERE'} \ \langle pattern \rangle \end{array}$$

[1]Here, and in the following we use namespace prefixes to abbreviate IRIs. The usual IRIs are assumed for rdf, rdfs, dc (dublin core), foaf (friend-of-a-friend), vcard vocabularies. my is a prefix bound to an arbitrary IRI.

```
⟨template⟩     ::= ⟨triple⟩ | ⟨template⟩ 'AND' ⟨template⟩ | '{' template '}'
⟨triple⟩       ::= '(' ⟨resource⟩',' ⟨predicate⟩',' ⟨resource⟩ ')'
⟨resource⟩     ::= ⟨iri⟩ | ⟨variable⟩ | ⟨literal⟩ | ⟨blank⟩
⟨predicate⟩    ::= ⟨iri⟩ | ⟨variable⟩
⟨variable⟩     ::= '?' ⟨identifier⟩
⟨pattern⟩      ::= ⟨triple⟩ | '{' ⟨pattern⟩ '}'
                |  ⟨pattern⟩ 'FILTER' '(' ⟨condition⟩ ')' |
                |  ⟨pattern⟩ 'AND' ⟨pattern⟩ | ⟨pattern⟩ 'UNION' ⟨pattern⟩
                |  ⟨pattern⟩ 'MINUS' ⟨pattern⟩ | ⟨pattern⟩ 'OPT' ⟨pattern⟩
⟨condition⟩    ::= ⟨variable⟩ '=' ⟨variable⟩ | ⟨variable⟩ '=' (⟨literal⟩|⟨iri⟩)
                |  'BOUND(' ⟨variable⟩ ')' | 'isBLANK(' ⟨variable⟩ ')'
                |  'isLITERAL(' ⟨variable⟩ ')' | 'isIRI(' ⟨variable⟩ ')'
                |  ⟨negation⟩ | ⟨conjunction⟩ | ⟨disjunction⟩
⟨negation⟩     ::= '¬'⟨condition⟩
⟨conjunction⟩  ::= ⟨condition⟩ '∧' ⟨condition⟩
⟨disjunction⟩  ::= ⟨condition⟩ '∨' ⟨condition⟩
```

We pose some additional syntactic restrictions: SPARQL queries are *range-restricted*, i.e., all variables in the "head" (**CONSTRUCT** or **SELECT** clause) also occurs in the "body" (**WHERE** clause) of the query. We assume *error-free* SPARQL expressions (in contrast to [63] and [64]), i.e., for each **FILTER** expression all variables occurring in the (right-hand) condition must also occur in the (left-hand) pattern.

Finally, we allow only *valid RDF constructions* in **CONSTRUCT** clauses, i.e., no literal may occur as a subject, all variables occurring in subject position are never bound to literals, and all variables occurring in predicate position are only ever bound to IRIs (but not to literals or blank nodes). The first condition can be enforced statically, the others by adding appropriate `isIRI` or negated isLITERAL filters to the query body.

Following [64], we define the semantics of SPARQL queries based on *substitutions*. A substitution $\theta = \langle v_1, n_1, \ldots, v_k : n_k \rangle$ with $v_i \in \mathsf{Vars}(Q) \wedge n_i \in nodes(D)\}$ for a query Q over a data graph D (as in Section 1.4) maps some variables from Q to nodes in D. For a substitution θ we denote with $\mathrm{dom}(\theta)$ the variables mapped by θ. Given a triple pattern $t = (s, p, o)$, we denote with $t\theta$ the application of θ to t replacing all occurrences of variables mapped in θ by their mapping in t. For a triple (s, p, o) containing no variables, we say $(s, p, o) \in D$ if there is a p labeled edge between s and o labeled nodes in D.

On sets of substitutions the usual relational operations $\bowtie$, $\cup$, and $\setminus$ apply. We define the (left) semi-join $R \ltimes S = (R \bowtie S) \cup (R \setminus S)$.

Finally, given a template t, i.e., a conjunction of triple patterns, $\mathsf{std}(t)$ returns t but replacing each blank node identifier (i.e., strings of the form **_:** *identifier*) with a new blank node identifier not occurring in D and not created by a prior application of std. Intuitively, $\mathsf{std}(t)$ creates a new instance of t such that the blank nodes of two instances (and any instance with the input graph) do not overlap.

Using these definitions, Table 37 gives the semantics of SPARQL **SELECT** and **CONSTRUCT** queries by means of $[\![\ ]\!]^D$. $[\![\ ]\!]^D$ translates the **WHERE** clause using $[\![\ ]\!]^D_{\mathsf{Subst}}$ and a **CONSTRUCT** clause, if present, using

$\llbracket (s, p, o) \rrbracket^D_{\mathsf{Subst}}$	$= \{\theta : \mathrm{dom}(\theta) = \mathsf{Vars}((s, p, o)) \wedge t\theta \in D\}$
$\llbracket \mathit{pattern}_1 \ \texttt{AND}\ \mathit{pattern}_2 \rrbracket^D_{\mathsf{Subst}}$	$= \llbracket \mathit{pattern}_1 \rrbracket^D_{\mathsf{Subst}} \bowtie \llbracket \mathit{pattern}_2 \rrbracket^D_{\mathsf{Subst}}$
$\llbracket \mathit{pattern}_1 \ \texttt{UNION}\ \mathit{pattern}_2 \rrbracket^D_{\mathsf{Subst}}$	$= \llbracket \mathit{pattern}_1 \rrbracket^D_{\mathsf{Subst}} \cup \llbracket \mathit{pattern}_2 \rrbracket^D_{\mathsf{Subst}}$
$\llbracket \mathit{pattern}_1 \ \texttt{MINUS}\ \mathit{pattern}_2 \rrbracket^D_{\mathsf{Subst}}$	$= \llbracket \mathit{pattern}_1 \rrbracket^D_{\mathsf{Subst}} \setminus \llbracket \mathit{pattern}_2 \rrbracket^D_{\mathsf{Subst}}$
$\llbracket \mathit{pattern}_1 \ \texttt{OPT}\ \mathit{pattern}_2 \rrbracket^D_{\mathsf{Subst}}$	$= \llbracket \mathit{pattern}_1 \rrbracket^D_{\mathsf{Subst}} \ltimes \llbracket \mathit{pattern}_2 \rrbracket^D_{\mathsf{Subst}}$
$\llbracket \mathit{pattern}\ \texttt{FILTER}\ \mathit{condition} \rrbracket^D_{\mathsf{Subst}}$	$= \{\theta \in \llbracket \mathit{pattern} \rrbracket^D_{\mathsf{Subst}} : \mathsf{Vars}(\mathit{condition}) \subset \mathrm{dom}(\theta) \wedge \llbracket \mathit{condition} \rrbracket^D_{\mathsf{Bool}}(\theta)\}$
$\llbracket \mathit{condition}_1 \wedge \mathit{condition}_2 \rrbracket^D_{\mathsf{Bool}}(\theta)$	$= \llbracket \mathit{condition}_1 \rrbracket^D_{\mathsf{Bool}}(\theta) \wedge \llbracket \mathit{condition}_2 \rrbracket^D_{\mathsf{Bool}}(\theta)$
$\llbracket \mathit{condition}_1 \vee \mathit{condition}_2 \rrbracket^D_{\mathsf{Bool}}(\theta)$	$= \llbracket \mathit{condition}_1 \rrbracket^D_{\mathsf{Bool}}(\theta) \vee \llbracket \mathit{condition}_2 \rrbracket^D_{\mathsf{Bool}}(\theta)$
$\llbracket \neg \mathit{condition} \rrbracket^D_{\mathsf{Bool}}(\theta)$	$= \neg \llbracket \mathit{condition} \rrbracket^D_{\mathsf{Bool}}(\theta)$
$\llbracket \texttt{BOUND}(?v) \rrbracket^D_{\mathsf{Bool}}(\theta)$	$= v\theta \neq \mathbf{nil}$
$\llbracket \texttt{isLITERAL}(?v) \rrbracket^D_{\mathsf{Bool}}(\theta)$	$= v\theta \in L$
$\llbracket \texttt{isIRI}(?v) \rrbracket^D_{\mathsf{Bool}}(\theta)$	$= v\theta \in I$
$\llbracket \texttt{isBLANK}(?v) \rrbracket^D_{\mathsf{Bool}}(\theta)$	$= v\theta \in B$
$\llbracket ?v = \mathit{literal} \rrbracket^D_{\mathsf{Bool}}(\theta)$	$= v\theta = \mathit{literal}$
$\llbracket ?u = ?v \rrbracket^D_{\mathsf{Bool}}(\theta)$	$= u\theta = v\theta \wedge u\theta \neq \mathbf{nil}$
$\llbracket \mathit{triple} \rrbracket^D_{\mathsf{Graph}}(\theta)$	$= \mathit{triple}\theta$ if $\forall v \in \mathsf{Vars}(\mathit{triple}) : v\theta \neq \mathbf{nil}$, $\top$ otherwise
$\llbracket \mathit{template}_1 \ \texttt{AND}\ \mathit{template}_2 \rrbracket^D_{\mathsf{Graph}}(\theta)$	$= \llbracket \mathit{template}_1 \rrbracket^D_{\mathsf{Graph}}(\theta) \cup \llbracket \mathit{template}_2 \rrbracket^D_{\mathsf{Graph}}(\theta)$
$\llbracket \texttt{CONSTRUCT}\ t\ \texttt{WHERE}\ p \rrbracket^D$	$= \bigcup_{\theta \in \llbracket P \rrbracket^D_{\mathsf{Subst}}} \llbracket \mathsf{std}(t) \rrbracket^D_{\mathsf{Graph}}(\theta)$
$\llbracket \texttt{SELECT}\ V\ \texttt{WHERE}\ p \rrbracket^D$	$= \pi_V(\llbracket P \rrbracket^D_{\mathsf{Subst}})$

Table 37. Semantics for SPARQL

$[\![\ ]\!]^D_{\mathsf{Graph}}$. For a **SELECT** query, we project the set of substitutions returned by $[\![\ ]\!]^D_{\mathsf{Subst}}$ to the set of answer variables V. For a **CONSTRUCT** query we apply each substitution $\theta \in [\![P]\!]^D_{\mathsf{Subst}}$ to a new instance of the template t contained in the **CONSTRUCT** clause created using std. Applying a substitutions is straightforward except that triples containing one or more variables that bound to **nil** by θ are omitted entirely.

The semantics of a SPARQL pattern P contained in the **WHERE** clause is given by $[\![P]\!]^D$ and produces a set of substitutions (or bindings) for variables in P. Triple patterns t (case 1) are evaluated to the set of substitutions θ such that the $t\theta$ contains no more variables and falls in D. Pattern compositions **AND**, **UNION**, **MINUS**, and **OPT** are reduced to the appropriate operations on sets of substitutions (cases 2–4). **FILTER** expressions (case 5) are again evaluated straightforwardly, as restrictions on the substitutions returned by the (left-hand) pattern with the boolean formula that is provided by $[\![\ ]\!]^D_{\mathsf{Bool}}$ for the condition of the filter expression. $\mathsf{Vars}(\mathit{condition}) \subset \mathrm{dom}(\theta)$ is not strictly necessary as it merely restates that we only consider error-free SPARQL queries.

5.3 Translating SPARQL Queries

Translating SPARQL to CIQLog is, for the most part, a direct mirror of the semantics in Table 37. The main difference is when translating **CONSTRUCT** clauses. Here, we create value invention terms for each resource in the template that depend on all variables in the **CONSTRUCT** clause. This implements std in the above semantics, i.e., the instantiation of the blank nodes for each substitution (i.e., binding tuple). This implies, however, that the result is not the data graph representation of an RDF graph in the sense of Section 1.4 since it may contain several nodes with the same (IRI or literal) label. he translation of **WHERE** clauses, though, does not use repeated variable occurrences in the body but separate variables for each subject, predicate, or object and label variables for common occurrences of the same SPARQL variable. Thus it can not distinguish between a graph with several nodes with the same label (each carrying some but not all properties of the named resource) and one where all these nodes are collapsed. Nevertheless, we may want to create a proper data graph representation by collapsing all such nodes. This can be achieved in CIQLog by a simple graph transformation on all nodes with the same label that exploits that value invention in CIQLog can be parametrized with an equivalence relation. Usually, we assume node equality $\doteq$, but in this case we employ label equality $\cong$ and thus create only one node in the transformed graph per unique node or edge label in the input graph.

Translation examples. To illustrate this point and the general translation more closely, consider again the above SPARQL example:

```
CONSTRUCT { ?p my:published ?a }
WHERE { ?a rdf:type bib:Article AND ?a dc:creator ?p
        AND ?p vcard:FN ‘M. T. Cicero’ }
```

In CIQLog, we can express the same query constructing a new graph with two nodes and one edge per binding tuple for p and a. Here and in the following examples we omit conditionals if all variables are non-optional and use prefix abbreviations also for CIQLog labels:

$$\mathfrak{L}(\mathsf{id}_1(\mathcal{A}),v_p) \land \circ\!\!\rightarrow(\mathsf{id}_1(\mathcal{A}),\mathsf{id}_2(\mathcal{A})) \land \mathfrak{L}(\mathsf{id}_2(\mathcal{A}),\mathtt{my{:}published}) \land$$
$$\rightarrow\!\!\circ(\mathsf{id}_3(\mathcal{A}),\mathsf{id}_2(\mathcal{A})) \land \mathfrak{L}(\mathsf{id}_3(\mathcal{A}),v_a)$$
$$\longleftarrow\ \mathfrak{L}(s_1,v_a) \land \circ\!\!\rightarrow(s_1,p_1) \land \mathfrak{L}(p_1,\mathtt{rdf{:}type}) \land$$

```
       →○(o₁, p₁) ∧ 𝔏(o₁,bib:Article) ∧
5    𝔏(s₂,v_a) ∧ ○→(s₂,p₂) ∧ 𝔏(p₂,dc:creator) ∧ →○(o₂, p₂) ∧ 𝔏(o₂,v_p) ∧
     𝔏(s₃,v_p) ∧ ○→(s₃,p₃) ∧ 𝔏(p₃,vcard:FN) ∧ →○(o₃,p₃) ∧
7       𝔏(o₃,‘M. T. Cicero’)
```

The above CIQLog rule illustrates the general translation scheme: each triple pattern in the body is separately translated using new variables for subject, predicate, and object. If any of those is a SPARQL variable, we add a label restriction on the respective variable and a label variable that is the associated CIQLog variable of that SPARQL variable. In the head, triple patterns are also translated independently, but now label restrictions are established with query variables (line 2). All node value expressions in the head are over all variables in the CONSTRUCT clause (abbreviated above as $\mathcal{A} = [v_a, v_p]$). This includes possible optional variables (see next example). As equivalence relation we use here equality on labels, not node identity.

In contrast to Xcerpt, XPath, and XQuery, we do not include root relations though if desired appropriate relations (e.g., to each named resource) can be easily added.

The second example focuses on the effect of **OPT** clauses and blank nodes in the head:

```
1 CONSTRUCT { _:group my:member ?a AND ?a my:otherAuthor ?p2 }
  WHERE { ?a rdf:type bib:Article AND ?a dc:creator ?p
3         AND ?p vcard:FN ‘M. T. Cicero’
          OPT { ?a dc:creator ?p2 FILTER (¬ ?p = ?p2) } }
```

We select in addition any further creators of a paper authored by Cicero. We return a graph over the article and the optional further creator. _:group is a blank node (with local identifier group. In CIQLog we obtain the following rule for this SPARQL query (with $\mathcal{A} = [v_a, v_p, v_{p_2}]$)

```
   𝔏(id₁(𝒜), _:id₁(𝒜)) ∧ ○→(id₁(𝒜),id₂(𝒜)) ∧ 𝔏(id₂(𝒜),my:member) ∧
2      →○(id₃(𝒜),id₂(𝒜)) ∧ 𝔏(id₃(𝒜),v_a) ∧
       if v_p ≠ nil ∧ p₂ ≠ nil then
4         𝔏(id₄(𝒜),v_a) ∧ ○→(id₄(𝒜),id₅(𝒜)) ∧ 𝔏(id₅(𝒜),my:otherAuthor) ∧
          →○(id₆(𝒜),id₅(𝒜)) ∧ 𝔏(id₆(𝒜),v_p₂) ⟵
6   𝔏(s₁,v_a) ∧ ○→(s₁,p₁) ∧ 𝔏(p₁,rdf:type) ∧ →○(o₁, p₁) ∧ 𝔏(o₁,bib:Article) ∧
    𝔏(s₂,v_a) ∧ ○→(s₂,p₂) ∧ 𝔏(p₂,dc:creator) ∧ →○(o₂,p₂) ∧ 𝔏(o₂,v_p) ∧
8   𝔏(s₃,v_p) ∧ ○→(s₃,p₃) ∧ 𝔏(p₃,vcard:FN) ∧ →○(o₃,p₃) ∧ 𝔏(o₃,‘M.T.Cicero’) ∧
   (𝔏(s₄,v_a) ∧ ○→(s₄,p₄) ∧ 𝔏(p₄,dc:creator) ∧ →○(o₄,p₄) ∧
10  𝔏(o₄,v_p₂) ∧ ¬(v_p ≠ v_p₂)) ∨ true
```

First, in the body the optional is realized by an disjunction with **true**, thus ensuring that the query never fails due to the optional part. In the head, we add a conditional around all triple pattern containing optional variables. Notice, how we use value invention to instantiate a new blank node for each binding pair in line 1.

Translation function. Guided by these examples, we can turn to the translation function for SPARQL, $\mathrm{tr}_{\mathrm{SPARQL}}$, as specified in Table 39. It employs two helper functions, tsh for translating **CONSTRUCT** clauses and tsp for translating **WHERE** clauses. As for the translation of Xcerpt and XQuery we use an environment containing mapping from SPARQL variables to those of CIQLog and a list of iteration variables that is always the set of all variables occurring in the **CONSTRUCT** clause (case 1 in Table 39).

functionSPARQL expression	= CIQLog **expression**
$\mathrm{tr}_{\mathrm{SPARQL}}\langle \texttt{CONSTRUCT}\ \mathit{template}\ \texttt{WHERE}\ \mathit{pattern}\rangle$	$= C \longleftarrow Q$ **where** $(C, v) = \mathrm{tsh}(\mathcal{E}')\langle \mathit{template}\rangle$ $(\mathcal{E}, Q) = \mathrm{tsp}(\varnothing)\langle \mathit{pattern}\rangle$ $\mathcal{E}' = \mathcal{E}$ with $\mathcal{E}'.\mathrm{iter} = \mathrm{Vars}(\mathit{template})$
$\mathrm{tr}_{\mathrm{SPARQL}}\langle \texttt{SELECT}\ v_1, \ldots, v_n\ \texttt{WHERE}\ \mathit{pattern}\rangle$	$= \mathrm{ans}(\mathcal{E}(v_1), \ldots, \mathcal{E}(v_n)) \longleftarrow Q$ **where** $(\mathcal{E}, Q) = \mathrm{tsp}(\varnothing)\langle \mathit{pattern}\rangle$
$\mathrm{tsh}(\mathcal{E})\langle \mathit{literal}\rangle$	$= (\mathrm{Lab}(v, \text{'}\mathit{literal}\text{'}), v)$ **where** $v = \mathrm{id}(\mathcal{E}.\mathrm{iter})$ with id new identifier
$\mathrm{tsh}(\mathcal{E})\langle \mathit{iri}\rangle$	$= (\mathrm{Lab}(v, \mathit{iri}), v)$ **where** $v = \mathrm{id}(\mathcal{E}.\mathrm{iter})$ with id new identifier
$\mathrm{tsh}(\mathcal{E})\langle ?\mathit{vid}\rangle$	$= (\mathrm{Lab}(v, \mathcal{E}(\mathit{vid})), v)$ **where** $v = \mathrm{id}(\mathcal{E}.\mathrm{iter})$ with id new identifier
$\mathrm{tsh}(\mathcal{E})\langle (s, p, o)\rangle$	$= (\ \mathbf{if}\ C\ \mathbf{then}\ F_s \wedge F_p \wedge F_o \wedge \circ\!\!\rightarrow(v_s, v_p) \wedge \rightarrow\!\!\circ(v_o s, v_p), v_s)$ **where** $C = (v_1 \neq \mathbf{nil} \wedge \ldots \wedge v_k \neq \mathbf{nil})$ where $\{v_1, \ldots, v_k\} = \mathrm{Vars}((s, p, o))$ $(F_s, v_s) = \mathrm{tsh}(\mathcal{E})\langle s\rangle \qquad (F_p, v_p) = \mathrm{tsh}(\mathcal{E}_s)\langle p\rangle$ $(F_o, v_o) = \mathrm{tsh}(\mathcal{E}_p)\langle o\rangle$
$\mathrm{tsh}(\mathcal{E})\langle t_1\ \texttt{AND}\ t_2\rangle$	$= (F_1 \wedge F_2, v_2)$ **where** $(F_1, v_1) = \mathrm{tsp}(\mathcal{E})\langle t_1\rangle$ $(F_2, v_2) = \mathrm{tsp}(\mathcal{E})\langle t_2\rangle$

Table 39. Translating SPARQL queries and **CONSTRUCT** clauses

functionSPARQL expression	= CIQLog **expression**
$\mathsf{tsr}(\mathcal{E})\langle \mathit{literal}\rangle$	$= (\mathcal{E}, \mathsf{Lab}(v, \text{'}\mathit{literal}\text{'}), v)$ **where** v is a new variable
$\mathsf{tsr}(\mathcal{E})\langle \mathit{iri}\rangle$	$= (\mathcal{E}, \mathsf{Lab}(v, \mathit{iri}), v)$ **where** v is a new variable
$\mathsf{tsr}(\mathcal{E})\langle ?\mathit{vid}\rangle$	$= (\mathcal{E} \mathbin{+\!\!\!+} \{(\mathit{vid}, l)\}, \mathsf{Lab}(v, l), v)$ **where** v is a new variable and $l = \mathcal{E}(\mathit{vid})$ if defined, otherwise a new variable
$\mathsf{tsp}(\mathcal{E})\langle (s, p, o)\rangle$	$= (\mathcal{E}_o, F_s \wedge F_p \wedge F_o \wedge \circ\!\!\rightarrow(v_s, v_p) \wedge \rightarrow\!\!\circ(v_o s, v_p))$ **where** $(\mathcal{E}_s, F_s, v_s) = \mathsf{tsr}(\mathcal{E})\langle s\rangle$ $(\mathcal{E}_p, F_p, v_p) = \mathsf{tsr}(\mathcal{E}_s)\langle p\rangle$ $(\mathcal{E}_o, F_o, v_o) = \mathsf{tsr}(\mathcal{E}_p)\langle o\rangle$
$\mathsf{tsp}(\mathcal{E})\langle p_1\ \texttt{AND}\ p_2\rangle$	$= (\mathcal{E}_2, F_1 \wedge F_2)$ **where** $(\mathcal{E}_1, F_1) = \mathsf{tsp}(\mathcal{E})\langle p_1\rangle$ $(\mathcal{E}_2, F_2) = \mathsf{tsp}(\mathcal{E}_1)\langle p_2\rangle$
$\mathsf{tsp}(\mathcal{E})\langle p_1\ \texttt{UNION}\ p_2\rangle$	$= (\mathcal{E}_2, F_1 \vee F_2)$ **where** $(\mathcal{E}_1, F_1) = \mathsf{tsp}(\mathcal{E})\langle p_1\rangle$ $(\mathcal{E}_2, F_2) = \mathsf{tsp}(\mathcal{E}_1)\langle p_2\rangle$
$\mathsf{tsp}(\mathcal{E})\langle p_1\ \texttt{MINUS}\ p_2\rangle$	$= (\mathcal{E}_2, F_1 \wedge \neg(F_2))$ **where** $(\mathcal{E}_1, F_1) = \mathsf{tsp}(\mathcal{E})\langle p_1\rangle$ $(\mathcal{E}_2, F_2) = \mathsf{tsp}(\mathcal{E}_1)\langle p_2\rangle$
$\mathsf{tsp}(\mathcal{E})\langle p_1\ \texttt{OPT}\ p_2\rangle$	$= (\mathcal{E}_2\}, F_1 \wedge (F_2 \vee \mathbf{true})$ **where** $(\mathcal{E}_1, F_1) = \mathsf{tsp}(\mathcal{E})\langle p_1\rangle$ $(\mathcal{E}_2, F_2) = \mathsf{tsp}(\mathcal{E}_1)\langle p_2\rangle$
$\mathsf{tsp}(\mathcal{E})\langle p_1\ \texttt{FILTER}\ p_2\rangle$	$= (\mathcal{E}_1, F_1 \wedge \mathsf{tsc}(\mathcal{E}_1)\langle p_2\rangle)$ **where** $(\mathcal{E}_1, F_1) = \mathsf{tsp}(\mathcal{E})\langle p_1\rangle$
$\mathsf{tsc}(\mathcal{E})\langle c_1 \wedge c_2\rangle$	$= \mathsf{tsc}(\mathcal{E})\langle c_1\rangle \wedge \mathsf{tsc}(\mathcal{E})\langle c_2\rangle$
$\mathsf{tsc}(\mathcal{E})\langle c_1 \vee c_2\rangle$	$= \mathsf{tsc}(\mathcal{E})\langle c_1\rangle \vee \mathsf{tsc}(\mathcal{E})\langle c_2\rangle$
$\mathsf{tsc}(\mathcal{E})\langle \neg c\rangle$	$= \neg(\mathsf{tsc}(\mathcal{E})\langle c\rangle)$
$\mathsf{tsc}(\mathcal{E})\langle ?\mathit{vid}_1 = ?\mathit{vid}_2\rangle$	$= \mathcal{E}(\mathit{vid}_1) = \mathcal{E}(\mathit{vid}_2)$
$\mathsf{tsc}(\mathcal{E})\langle ?\mathit{vid} = \mathit{literal}\rangle$	$= \mathcal{E}(\mathit{vid}_1) = \text{'}\mathit{literal}\text{'}$
$\mathsf{tsc}(\mathcal{E})\langle ?\mathit{vid} = \mathit{iri}\rangle$	$= \mathcal{E}(\mathit{vid} = \mathit{iri})$
$\mathsf{tsc}(\mathcal{E})\langle \texttt{BOUND}(?\mathit{vid})\rangle$	$= \mathcal{E}(\mathit{vid}) \neq \texttt{nil}$
$\mathsf{tsc}(\mathcal{E})\langle \texttt{isBLANK}(?\mathit{vid})\rangle$	$= \mathcal{E}(\mathit{vid}) = \texttt{\_:}\mathit{identifier}$
$\mathsf{tsc}(\mathcal{E})\langle \texttt{isLITERAL}(?\mathit{vid})\rangle$	$= \mathcal{E}(\mathit{vid}) = \text{'}\mathit{string}\text{'}$
$\mathsf{tsc}(\mathcal{E})\langle \texttt{isIRI}(?\mathit{vid})\rangle$	$= \neg(\mathsf{tsc}(\mathcal{E})\langle \texttt{isLiteral}(?\mathit{vid})\rangle$ or $\mathsf{tsc}(\mathcal{E})\langle \texttt{isLiteral}(?\mathit{vid})\rangle)$

Table 41. Translating SPARQL patterns and conditions

The translation of SPARQL construct patterns is fairly unremarkable. Notice that variables are translated by retrieving the related label variable from the environment.

SPARQL patterns are translated using tsp, specified in Table 41, and its helpers tsr (for resources) and tsc (for conditions). Subjects, predicates, and objects are translated by always creating new variables and placing label restrictions on those variables. When translating SPARQL variables, we establish label restrictions with the associated label variable of CIQLog (l in case 3 of Table 41). Otherwise the translation is fairly straightforward.

We conclude the translation of SPARQL with the conjecture that the graph constructed by $\mathrm{tr}_{\mathrm{SPARQL}} P$ contains (in the sense defined above), up to consistent renaming of blank nodes, the same triples as $[\![P]\!]^D$.

Conjecture 5.1. For a given SPARQL **CONSTRUCT** query P there is a mapping f on literals, IRIs, and blank nodes, that is the identity on literals and IRIs, such a triple $(s, p, o) \in T$ if and only if $(f(s), f(p), f(o)) \in [\![P]\!]^D$ where T is in the graph obtained from the evaluation of $\mathrm{tr}_{\mathrm{SPARQL}} P$.

5.4 From SPARQL to Rules: RDFLog

SPARQL queries can be considered as non-recursive, single-rule expressions. There have been some approaches to extending SPARQL with rules, e.g., [64], or to embed SPARQL queries in a rule-based query language.

The above translation yields a single CIQLog rule for each SPARQL query. Obviously, we can allow a program to contain many such rules which then can provide input to each other.

However, there are a number of challenges when adding rules to an RDF query language that are better addressed in RDFLog [15]. Most notably, SPARQL's heads always group over all answer variables which limits the kind of queries that can be expressed significantly (at no gain in complexity as shown in [15].

5.5 Conclusion

Compared with the translations for XQuery and Xcerpt, SPARQL is an easy target for translation to CIQLog. The main difficulty lies in the graph construction not in the query patterns. We have employed a translation scheme using label variables above. This emphasizes that SPARQLs domain is actually better thought of as IRIs, literals, and blank node identifier rather than as nodes of an RDF graph (as the model theory might suggest), as done also in the semantics for SPARQL above and in [64]. For the most part, we could nevertheless also employ a node-based translation scheme but then the result construction becomes far more involved.

Acknowledgements.

This research has been funded by the European Commission and by the Swiss Federal Office for Education and Science within the 6th Framework Programme project REWERSE number 506779 (cf. `http://rewerse.net`).

Bibliography

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley Publishing Co., Boston, MA, USA, 1995.

[2] S. Abiteboul and P. C. Kanellakis. Object Identity as a Query Language Primitive. *Journal of the ACM*, 45(5):798–842, 1998.

[3] R. Agrawal, A. Borgida, and H. V. Jagadish. Efficient management of transitive relationships in large data and knowledge bases. In *Proc. ACM Symp. on Management of Data (SIGMOD)*, pages 253–262, New York, NY, USA, 1989. ACM.

[4] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1974.

[5] Apple Inc. *plist — Property List Format*, 2003.

[6] D. Backett. Turtle—Terse RDF Triple Language. Technical report, Institute for Learning and Research Technology, University of Bristol, 2007.

[7] D. Beckett and B. McBride. RDF/XML Syntax Specification (Revised). Recommendation, W3C, 2004.

[8] M. Benedikt and C. Koch. Interpreting tree-to-tree queries. In *Proc. Int'l. Symp. on Automata, Languages and Programming (ICALP)*, pages 552–564, 2006.

[9] M. Benedikt and C. Koch. Xpath leashed. In *ACM Computing Surveys*, 2007.

[10] M. Bojańczyk, C. David, A. Muscholl, T. Schwentick, and L. Segoufin. Two-variable logic on data trees and xml reasoning. In *Proc. ACM Symp. on Principles of Database Systems (PODS)*, pages 10–19, New York, NY, USA, 2006. ACM.

[11] P. Boncz, T. Grust, M. van Keulen, S. Manegold, J. Rittinger, and J. Teubner. MonetDB/XQuery: a fast XQuery Processor powered by a Relational Engine. In *Proc. ACM Symp. on Management of Data (SIGMOD)*, pages 479–490, New York, NY, USA, 2006. ACM Press.

[12] T. Bray, D. Hollander, A. Layman, and R. Tobin. Namespaces in XML (2nd Edition). Recommendation, W3C, 2006.

[13] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau. Extensible Markup Language (XML) 1.0 (Third Edition). Recommendation, W3C, 2004.

[14] J. Broekstra and A. Kampman. An rdf query and transformation language. In *Semantic Web and Peer-to-Peer*, chapter 2, pages 23–39. Springer Verlag, 2006.

[15] F. Bry, T. Furche, C. Ley, and B. Linse. Rdflog: Filling in the blanks in rdf querying. Technical Report PMS-FB-2008-01, University of Munich, 2007.

[16] J. V. D. Bussche, D. V. Gucht, M. Andries, and M. Gyssens. On the completeness of object-creating database transformation languages. *Journal of the ACM*, 44(2):272–319, 1997.

[17] L. Cabibbo. The expressive power of stratified logic programs with value invention. *Information and Computation*, 147(1):22–56, 1998.

[18] J. J. Carroll, C. Bizer, P. Hayes, and P. Stickler. Named graphs, provenance and trust. In *Proc. Int'l. World Wide Web Conf. (WWW)*, pages 613–622, New York, NY, USA, 2005. ACM.

[19] R. G. G. Cattell, D. K. Barry, M. Berler, J. Eastman, D. Jordan, C. Russell, O. Schadow, T. Stanienda, and F. Velez, editors. *Object Data Standard: ODMG 3.0*. Morgan Kaufmann, 2000.

[20] L. Chen, A. Gupta, and M. E. Kurul. Stack-based algorithms for pattern matching on dags. In *Proc. Int'l. Conf. on Very Large Data Bases (VLDB)*, pages 493–504. VLDB Endowment, 2005.

[21] J. Clark. XSL Transformations, Version 1.0. Recommendation, W3C, 1999.

[22] E. F. Codd. Extending the Database Relational Model to Capture more Meaning. *ACM Transactions on Database Systems*, 4(4):397–434, 1979.

[23] E. Cohen, E. Halperin, H. Kaplan, and U. Zwick. Reachability and Distance Queries via 2-hop Labels. In *Proc. ACM Symposium on Discrete Algorithms*, pages 937–946, Philadelphia, PA, USA, 2002. Society for Industrial and Applied Mathematics.

[24] B. Courcelle. Fundamental Properties of Infinite Trees. *Theoretical Computer Science*, 25:95–169, 1983.

[25] J. Cowan and R. Tobin. XML Information Set (2nd Ed.). Recommendation, W3C, 2004.

[26] P. T. Cox and T. Pietrzykowski. A complete, nonredundant algorithm for reversed skolemization. In *Proc. Int'l. Conf. on Automated Deduction (CADE)*, pages 374–385, London, UK, 1980. Springer-Verlag.

[27] P. F. Dietz. Maintaining order in a linked list. In *Proc. ACM Symp. on Theory of Computing (STOC)*, pages 122–127, New York, NY, USA, 1982. ACM.

[28] A. Dovier, C. Piazza, and A. Policriti. An efficient Algorithm for Computing Bisimulation Equivalence. *Theoretical Computer Science*, 311(1-3):221–256, 2004.

[29] D. Draper, P. Frankhauser, M. Fernández, A. Malhotra, K. Rose, M. Rys, J. Siméon, and P. Wadler. XQuery 1.0 and XPath 2.0 Formal Semantics. Recommendation, W3C, 2007.

[30] D. C. Fallside and P. Walmsley. XML Schema Part 0: Primer Second Edition. Recommendation, W3C, 2004.

[31] M. Fernández, A. Malhotra, J. Marsh, M. Nagy, and N. Walsh. XQuery 1.0 and XPath 2.0 Data Model. Recommendation, W3C, 2007.

[32] M. Fernández, J. Siméon, B. Choi, A. Marian, and G. Sur. Implementing XQuery 1.0 : The Galax Experience. In *Proc. Int'l. Conf. on Very Large Data Bases (VLDB)*, 2003.

[33] P. Foggia, C. Sansone, and M. Vento. An Improved Algorithm for Matching Large Graphs. In *Proc. Int'l. Workshop on Graph-based Representations*, 2001.

[34] T. Furche and F. Bry. Scalable, space-optimal implementation of xcerpt single rule programs—part 2: Algebra and evaluation. Deliverable I4-D15b, REWERSE, 2007.

[35] T. Furche, F. Bry, and S. Schaffert. Xcerpt 2.0: Specification of the (core) language syntax. Deliverable I4-D12, Network of Excellence REWERSE (Reasoning on the Web with Rules and Semantics), 2007.

[36] P. Genevès and J.-Y. Vion-Dury. XPath formal semantics and beyond: A Coq-based approach. In *Proc. Int'l. Conf. on Theorem Proving in Higher Order Logics (TPHOLs)*, pages 181–198, Salt Lake City, Utah, United States, August 2004. University Of Utah.

[37] G. Gottlob, C. Koch, and R. Pichler. Efficient Algorithms for Processing XPath Queries. *ACM Transactions on Database Systems*, 2005.

[38] G. Gottlob, C. Koch, R. Pichler, and L. Segoufin. The Complexity of XPath Query Evaluation and XML Typing. *Journal of the ACM*, 2005.

[39] P. Hayes and B. McBride. Rdf semantics. Recommendation, W3C, 2004.

[40] L. Hella, L. Libkin, J. Nurmonen, and L. Wong. Logics with aggregate operators. *Journal of the ACM*, 48(4):880–907, 2001.

[41] J. E. Hopcroft and J. K. Wong. Linear time algorithm for isomorphism of planar graphs. In *Proc. ACM Symposium on Theory of Computing*, pages 172–184, New York, NY, USA, 1974. ACM Press.

[42] R. Hull and M. Yoshikawa. Ilog: Declarative creation and manipulation of object identifiers. In *Proc. Int'l. Conf. on Very Large Data Bases (VLDB)*, pages 455–468, San Francisco, CA, USA, 1990. Morgan Kaufmann Publishers Inc.

[43] G. Karvounarakis, S. Alexaki, V. Christophides, D. Plexousakis, and M. Scholl. Rql: a declarative query language for rdf. In *Proc. Int'l. World Wide Web Conf. (WWW)*, pages 592–603, New York, NY, USA, 2002. ACM Press.

[44] M. Kay. XSL Transformations, Version 2.0. Recommendation, W3C, 2007.

[45] G. Klyne, J. J. Carroll, and B. McBride. Resource Description Framework (RDF): Concepts and Abstract Syntax. Recommendation, W3C, 2004.

[46] J. Köbler, U. Schöning, and J. Torán. *The Graph Isomorphism Problem: Its Structural Complexity*. Progress in Theoretical Computer Science. Birkhäuser/Springer-Verlag, 1993.

[47] C. Koch. On the Complexity of Nonrecursive XQuery and Functional Query Languages on Complex Values. *tods*, 31(4), 2006.

[48] L. Libkin and L. Wong. Query Languages for Bags and Aggregate Functions. *J. Comput. Syst. Sci.*, 55(2):241–272, 1997.

[49] B. Linse. Automatic Translation between XQuery and Xcerpt. Diplomarbeit/Master thesis, Institute for Informatics, University of Munich, 2006.

[50] A. Malhotra, J. Melton, and N. Walsh. XQuery 1.0 and XPath 2.0 Functions and Operators. Recommendation, W3C, 2007.

[51] F. Manola, E. Miller, and B. McBride. Rdf primer. Recommendation, W3C, 2004.

[52] J. Marsh. XML Base. Recommendation, W3C, 2001.

[53] J. M. Martínez. Mpeg-7 overview. Technical Report ISO/IEC JTC1/SC29/WG11N6828, INTERNATIONAL ORGANISATION FOR STANDARDISATION (ISO), 2004.

[54] M. Marx. First Order Paths in Ordered Trees. In *Proc. Int'l. Conf. on Database Theory (ICDT)*, pages 114–128, 2005.

[55] N. May, S. Helmer, C.-C. Kanne, and G. Moerkotte. XQuery Processing in Natix with an Emphasis on Join Ordering. In *Proc. of Int. Workshop on XQuery Implementation, Experience and Perspectives*, 2004.

[56] B. D. McKay. Practical Graph Isomorphism. In *Proc. Conf. on Numerical Mathematics and Computing*, 1980.

[57] H. Meuss and K. U. Schulz. Complete Answer Aggregates for Treelike Databases: A Novel Approach to Combine Querying and Navigation. *ACM Transactions on Information Systems*, 19(2):161–215, 2001.

[58] L. Miller, A. Seaborne, and A. Reggiori. Three Implementations of SquishQL, a Simple RDF Query Language. In *Proc. Int'l. Semantic Web Conf. (ISWC)*, June 2002.

[59] R. Milner. An Algebraic Definition of Simulation Between Programs. In *Proc. Intl. Joint Conf. on Artificial Intelligence*, pages 481–489, 1971.

[60] D. Olteanu. *Evaluation of XPath Queries against XML Streams*. Dissertation/Ph.D. thesis, University of Munich, 2005.

[61] D. Olteanu, H. Meuss, T. Furche, and F. Bry. XPath: Looking Forward. In *Proc. EDBT Workshop on XML-Based Data Management*, volume 2490 of *Lecture Notes in Computer Science*. Springer Verlag, 2002.

[62] R. Paige and R. E. Tarjan. Three partition refinement algorithms. *SIAM Journal of Computing*, 16(6):973–989, 1987.

[63] J. Perez, M. Arenas, and C. Gutierrez. Semantics and complexity of sparql. In *Proc. Int'l. Semantic Web Conf. (ISWC)*, 2006.

[64] A. Polleres. From sparql to rules (and back). In *Proc. Int'l. World Wide Web Conf. (WWW)*, pages 787–796, New York, NY, USA, 2007. ACM.

[65] E. Prud'hommeaux and A. Seaborne. SPARQL Query Language for RDF. Proposed recommendation, W3C, 2007.

[66] S. Schaffert. *Xcerpt: A Rule-Based Query and Transformation Language for the Web*. Dissertation/Ph.D. thesis, University of Munich, 2004.

[67] S. Schaffert and F. Bry. Querying the Web Reconsidered: A Practical Introduction to Xcerpt. In *Proc. Extreme Markup Languages (Int'l. Conf. on Markup Theory & Practice)*, 2004.

[68] R. Schenkel, A. Theobald, and G. Weikum. HOPI: An Efficient Connection Index for Complex XML Document Collections. In *Proc. Extending Database Technology*, 2004.

[69] Y.-H. Shen. Idlog: Extending the expressive power of deductive database languages. In *Proc. ACM Symp. on Management of Data (SIGMOD)*, pages 54–63, New York, NY, USA, 1990. ACM.

[70] D. W. Shipman. The Functional Data Model and the Data Languages DAPLEX. *ACM Transactions on Database Systems*, 6(1):140–173, 1981.

[71] S. Trißl and U. Leser. Fast and practical indexing and querying of very large graphs. In *Proc. ACM Symp. on Management of Data (SIGMOD)*, pages 845–856, New York, NY, USA, 2007. ACM.

[72] N. Walsh and L. Muellner. *DocBook: The Definitive Guide*. O?Reilly, 1999.

[73] H. Wang, H. He2, J. Yang, P. S. Yu, and J. X. Yu. Dual labeling: Answering graph reachability queries in constant time. In *Proc. Int'l. Conf. on Data Engineering (ICDE)*, page 75, Washington, DC, USA, 2006. IEEE Computer Society.

[74] F. Weigel, K. U. Schulz, and H. Meuss. The bird numbering scheme for xml and tree databases – deciding and reconstructing tree relations using efficient arithmetic operations. In *Proc. Int'l. XML Database Symposium (XSym)*, volume 3671 of *LNCS*, pages 49–67. Springer-Verlag, 2005.

[75] M. M. Zloof. Query By Example. In *AFIPS National Computer Conference*, 1975.