Extending XML Query Language Xcerpt by Ontology Queries*

Włodzimierz Drabent
Dept. of Computer and Information Science,
Linköping University, S 581 83 Linköping, Sweden
Institute of Computer Science, Polish Academy of Sciences,
ul. Ordona 21, Pl – 01-237 Warszawa, Poland
wdr@ida.liu.se

Artur Wilk
Dept. of Computer and Information Science,
Linköping University, S 581 83 Linköping, Sweden
artwi@ida.liu.se

September 28, 2007

Abstract

The paper addresses a problem of combining XML querying with ontology reasoning. We present an extension of a rule-based XML query and transformation language Xcerpt. The extension allows to interface an ontology reasoner from Xcerpt programs. In this way querying can employ the ontology information, for instance to filter out semantically irrelevant answers. The approach employs an existing Xcerpt engine and ontology reasoner; no modifications are required. We present the semantics of extended Xcerpt and an implementation algorithm. Communication between Xcerpt programs and ontology reasoner is based on DIG interface.

1 Introduction

XML, which is increasingly used for representing semistructured data on the Web, is supported by query languages such as XQuery [10]. Querying of XML data in such languages relies on the structure of the data, thus it is based on its syntax. However XML data may be given semantics by referring to concepts defined by ontologies. XML query languages do not provide ontology reasoning capabilities.

The objective of this paper is to show how structure-based querying of XML data can be combined with ontology reasoning. Thus we would like to query XML data using ontological information. For instance, we may want to filter XML data returned by a structural query by reasoning on an ontology to which the data is related. This can be illustrated by the following example. Assume that an XML database of culinary recipes is given. Each recipe indicates ingredients (like flour, salt, sugar etc.). We assume that

*This report is an extended version of the paper with the same title which will be presented at the conference 'Web Intelligence 2007'.

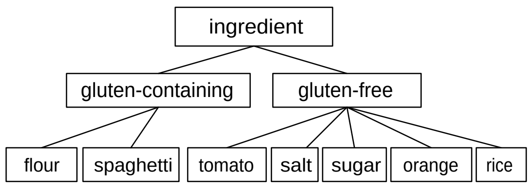


Figure 1: Ingredient ontology graph

the names of the ingredients are defined by a standard ontology, accessible separately on the Web and providing also some classification. For example, the ontology may specify disjoint classes of gluten-containing and gluten-free ingredients (see Figure 1). To find a gluten-free recipe we would query the XML database for recipes, and query the ontology to check if the ingredients are gluten-free.

Thus, the problem outlined above can be seen as the problem of interfacing an XML query language with an ontology reasoner. We propose a solution for XML query language Xcerpt [8, 7] and any ontology reasoner supporting a DIG interface. Xcerpt is being developed by the EU Network of Excellence REWERSE[1] in the 6th Framework Programme. It differs from most other XML query languages in that it is rule based and uses pattern matching instead of path navigation for locating and extracting data. DIG is a standard interface to ontology reasoners, supported by e.g. RacerPro[2] and Pellet[3]. Our approach augments Xcerpt rules with ontology queries. The extended language, called DigXcerpt, is easy to implement on the top of an Xcerpt implementation and a reasoner with DIG interface, without any need of modifying them.

Related work. The problem of combination of XML queries with ontology queries seems to be important for the Semantic Web. There exist many approaches combining a Description Logic and a language with logical semantics, like Datalog. For references see e.g. the articles of Eiter et al. and of Rosati in [2]. In contrast to these approaches, we add ontology queries to a language whose semantics is operational. Moreover we re-use an existing ontology reasoning system and a query language implementation, which is impossible for most of the approaches mentioned above.

An intermediate approach is presented in [1]. Ontology queries which cannot be solved are accumulated during rule computation. The fixed point semantics specifies the formulae (built out of delayed queries) with which the reasoner is eventually queried. This makes possible reasoning by cases. In contrast, our operational semantics makes the programmer decide when an ontology query is evaluated. The approach of [1] allows only Boolean ontology queries. It is applicable to a certain (negation-free) subset of Xcerpt. Our approach imposes no restriction on ontology queries and is applicable to full Xcerpt.

A different approach is that of [6], where XQuery is used both to query data and to perform (restricted kinds of) reasoning.

This work substantially differs from its previous versions. The method of [9] required modifying the Xcerpt engine and was restricted to Boolean ontology queries. An unpublished paper [5] presented a preliminary and low level version of the current approach, in particular without the extended rules (cf. Section 3).

The rest of the paper is organized as follows. Section 2 briefly introduces the query

[1]`http://www.rewerse.net/`

[2]`http://www.racer-systems.com/`

[3]`http://www.mindswap.org/2003/pellet/`

language Xcerpt and gives some background information on the DIG interface. Section 3 presents the syntax and semantics of DigXcerpt, the proposed extension of Xcerpt, including example programs. Section 4 describes how DigXcerpt can be implemented. Sections 5 and 6 provide a discussion and a summary. In the appendix we present a formal semantics of DigXcerpt and a formal proof of soundness of our implementation approach.

2 Preliminaries

2.1 Xcerpt

An Xcerpt program is a set of rules consisting of a body and of a head. The body of a rule is a query intended to match data terms. If the query contains variables such matching results in answer substitutions for variables. The head uses the results of matching to construct new data terms. The queried data is either specified in the body or is produced by rules of the program. There are two kinds of rules: goal rules produce the final output of the program, while construct rules produce intermediate data, which can be further queried by other rules. Their syntax is as follows:

```
GOAL            CONSTRUCT
  head            head
FROM            FROM
  body            body
END             END
```

Sometimes, we will denote the rules as *head* ← *body* neglecting distinction between goal and construct rules.

XML data is represented in Xcerpt as **data terms**. Data terms are built from basic constants and labels using two kinds of parentheses: brackets [ ] and braces { }. Basic constants represent basic values such as attribute values and character data (like #PCDATA in XML). A label represents an XML element name. The parentheses following a label include a sequence of data terms (its direct subterms). Brackets are used to indicate that the direct subterms are ordered (in the order of their occurrence in the sequence), while braces indicate that the direct subterms are unordered. The latter alternative is used to encode attributes of an XML element.

Example 1 *This is an XML element and the corresponding data term.*

```
<CD price="9.90">                CD[ attr{price["9.90"]},
  <title>Empire</title>              title["Empire"],
  <artist>Bob Dylan</artist>         artist["Bob Dylan"]
</CD>                            ]
```
□

There are two other kinds of terms in Xcerpt: query terms and construct terms.

Query terms are (possibly incomplete) patterns which are used in a rule body (query) to match data terms. In particular, every data term is a query term. Generally query terms may include variables so that a successful matching binds variables of a query term to data terms. Such bindings are called answer substitutions. A result of a query term matching a data term is a set of answer substitutions. For example, a query term $a[\text{"}b\text{"}, \texttt{var}\, X\,]$ matches a data term $a[\text{"}b\text{"}, \text{"}c\text{"}\,]$ resulting in the answer substitution set $\{\{X/\text{"}c\text{"}\}\}$. Query terms can be ordered or unordered patterns, denoted, respectively, by brackets and braces. For example, a query term $a[\,"c", "b"\,]$ is an ordered pattern and it does not match a data term $a[\,"b",\ "c"\,]$ but a query term $a\{\,"c",\ "b"\,\}$, which is an unordered pattern, matches $a[\,"b",\ "c"\,]$. Query terms with double brackets or braces are incomplete patterns. For example a query term $a[[\,"b", "d"\,]]$ is an incomplete pattern which matches a data term $a[\,"b",\ "c",\ "d"\,]$. As the query term uses brackets the matching subterms of the data term must occur in the same order as in the pattern. Thus the query term $a[[\,"b", "d"\,]]$ does not match a data term $a[\,"d",\ "b",\ "c"\,]$. In contrast a query term $a\{\{"b",\ "d"\}\}$ does. To specify subterms at arbitrary depth a keyword `desc` is used e.g. a query term `desc` $"d"$ matches a data term $a[\,b[\,"d"\,], "c"\,]$.

A query term q in a rule body may be associated with a resource r storing XML data or data terms. This is done by a construct of the form $\mathtt{in}[r, q]$, called targeted query term. Its meaning is that q is to be matched against data in r. Query terms in the body of a rule which have no associated resource are matched against data generated by rules of the Xcerpt program.

Queries are constructed from (targeted) query terms using logical connectives such as `or`, `and`, and `not`. A rule body is a query.

Construct terms are used in rule heads to construct new data terms. They are similar to data terms, but may contain variables. Data terms are constructed out of construct terms by applying answer substitutions obtained from a rule body. Construct terms may also use grouping constructs `all` and `some` which are used to collect all or, respectively, some instances that result from different variable bindings.

Example 2 *Consider an XML document* recipes.xml*, which is a collection of culinary recipes. The document is represented by the data term:*

```
recipes[
  recipe[ name["Recipe1"],
    ingredient[ name["sugar"], amount[attr{unit["tbsp"]},3] ],
    ingredient[ name["orange"], amount[ ... ] ],
  recipe[ name["Recipe2"],
    ingredient[ name["flour"], amount[ ... ] ],
    ingredient[ name["salt"], amount[ ... ] ],
  recipe[ name ["Recipe3"],
    ingredient[ name["spaghetti"], amount[ ... ] ],
    ingredient[ name["tomato"], amount[ ... ] ] ]
```

The Xcerpt rule queries the document and extracts the names of the recipes:

```
   GOAL
     recipe-names[ all var R ]
   FROM
     in[ "file:recipes.xml",
       recipes[[ recipe[[ name[ var R ] ]] ]] ]
   END
```

The result returned by the rule is:

```
   recipe-names[ "Recipe1", "Recipe2", "Recipe3" ]
```
□

See [4] for a concise formal semantics of single Xcerpt rules and [7] for a full description of Xcerpt.

2.2 DIG interface

To communicate with an ontology reasoner we have chosen DIG interface [3]. The DIG interface is an API for description logic systems. It is capable of expressing class and property expressions common to most description logics. Using DIG, clients can communicate with a reasoner through the use of HTTP POST requests. A *request* is an XML encoded message of one of the following types: management, ask or tell. Management requests are used e.g. to identify the reasoner along with its capabilities or to allocate a new knowledge base and return its unique identifier. *Tell requests*, containing *tell statements*, are used to make assertions into the reasoner's knowledge base. *Ask requests*, containing *ask statements*, are used to query the knowledge base. *Responses* to ask requests contain *response statements*. Tell, ask and response statements are built out of *concept statements* which are used to denote classes, properties, individuals etc. Here we present an extract of DIG statements used in our examples ($C, C_1, C_2, \ldots$ are concept statements):

- Concept statements:
  - `<catom name="`*CN*`"/>` – a concept (class) *CN*
  - `<ratom val="`*RN*`"/>` – a role (property) *RN*
  - `<some>` ***R C*** `</some>` – a concept whose objects are in relation R with some objects of a concept C (like $\exists R.C$ in description logics)
- Ask statements:
  - `<subsumes>`C_1 C_2`</subsumes>` – a Boolean query, asks whether a concept C_2 is subsumed by a concept C_1
  - `<descendants>`C`</descendants>` – asks for the list of subclasses of a concept C
- Response statements:
  - `<true/>` – if a statement is a logical consequence of the axioms in the knowledge base
  - `<false/>` – if a statement is not a logical consequence of the axioms in the knowledge base
  - `<error/>` – if, for instance, a concept queried about is not defined in the knowledge base
  - `<conceptSet>`
    `<synonyms>` $C_{11} \ldots C_{1n_1}$ `</synonyms>`
    `...`
    `<synonyms>` $C_{m1} \ldots C_{mn_m}$ `</synonyms>`
    `</conceptSet>`

DIG requests and responses are XML documents, some of their elements contain attributes. For instance, the attribute *id* is used to associate the obtained answers with the submitted queries.

Example 3 *This is an example of a query request to be sent to an ontology reasoner. It contains three DIG ask statements. The first two ask whether concepts* sugar *and* potato *are subclasses of the concept* gluten-containing. *The third one asks for direct subclasses of the class* gluten-containing. *(We skip namespace declarations in the elements* asks *and* responses.*)*

```
<?xml version="1.0"?>
<asks uri="uri_of_knowledge-base" ... >
  <subsumes id="q1">
    <catom name="gluten-containing"/>
    <catom name="sugar"/>
  </subsumes>
  <subsumes id="q2">
    <catom name="gluten-containing"/>
    <catom name="potato"/>
  </subsumes>
  <descendants id="q3">
    <catom name="gluten-containing"/>
  </descendants>
</asks>
```

This is a possible response to the query:

```
<?xml version="1.0"?>
<responses ... >
  <false id="q1"/>
  <error id="q2" message="Undefined concept name potato in TBox DEFAULT"/>
  <conceptSet id="q3">
    <synonyms><catom name="flour"/></synonyms>
    <synonyms><catom name="spaghetti"/></synonyms>
  </conceptSet>
</responses>
```

□

3 DigXcerpt: ontology queries in Xcerpt

This section presents an extension of Xcerpt, called DigXcerpt, allowing attaching ontology queries to Xcerpt rules. A DigXcerpt program is a set of Xcerpt rules and extended rules. The syntax of an extended construct rule is

```
CONSTRUCT
  head
WHERE
  dig [ digResponseQuery, digAskConstruct ]
FROM
  body
END
```

Analogical syntax can be used for extended goal rules (with the keyword GOAL instead of CONSTRUCT). Sometimes the rule will be denoted as

$$head \leftarrow (digResponseQuery, digAskConstruct), body$$

(without distinguishing between a construct and goal rule). *digAskConstruct* is a construct term intended to produce DIG ask statements which are sent to the reasoner. *digResponseQuery* is a query term that is applied to the response statements returned by the reasoner.

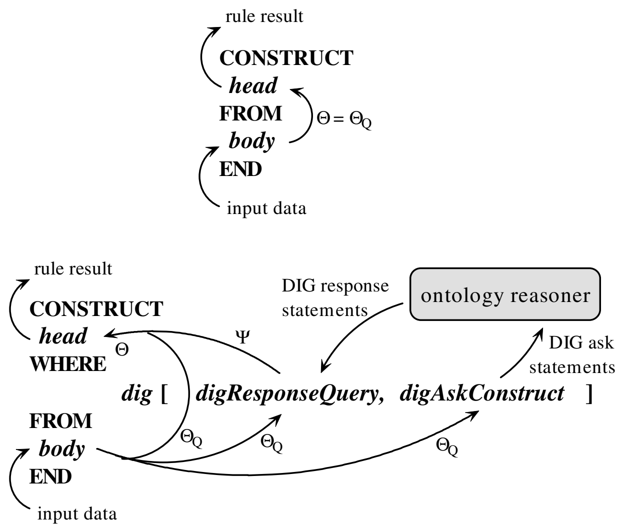


Figure 2: Data flow in an Xcerpt rule and in an extended rule. $\Theta, \Theta_Q, \Psi_\theta$ are sets of substitutions as described below and $\Psi = \bigcup_{\theta \in \Theta_Q} \Psi_\theta$.

As in an Xcerpt rule, *head* is a construct term and *body* is a query to the results of other rules and/or to external resources.

It is required that each variable occurring in *digAskConstruct* occurs in *body*, and each variable occurring in *head* occurs in *body* or in *digResponseQuery*. Additionally, a variable which occurs in *digAskConstruct* under the scope of a grouping construct (i.e. `all` or `some`) cannot occur in *digResponseQuery*.

Now we describe the semantics of an extended construct rule $c \leftarrow (q, c'), Q$. We assume that the rule does not contain grouping constructs. Let Θ_Q be the set of answer substitutions obtained by evaluation of the body Q of the rule. Each $\theta \in \Theta_Q$ is applied to the construct term c'; this produces a DIG ask statement $c'\theta$ to be sent to the reasoner. For each $c'\theta$ the reasoner returns a DIG response statement d_θ. To each d_θ the query term $q\theta$ is applied, producing a set Ψ_θ of substitutions. (The domain of the substitutions are those variables that occur in q and do not occur in Q.) A set of substitutions $\Theta = \{\, \theta \cup \sigma \mid \theta \in \Theta_Q,\ \sigma \in \Psi_\theta \,\}$ is constructed. (Informally: the substitutions bind the rule variables according to the results of Q and of DIG querying.) Now the set of results of the whole rule is $\{\, c\theta \mid \theta \in \Theta \,\}$ (the substitutions from Θ are applied to the head of the rule). Figure 2 presents the data flow in an extended rule.

The semantics above has to be generalized for the case where a construct term (c or c') of the extended construct rule $c \leftarrow (q, c'), Q$ contains a grouping construct. Applying substitutions to a construct term has to be replaced by a more sophisticated operation. We skip the technical details by referring to the semantics of Xcerpt. Two modifications of the definition above are needed. 1. The set of DIG ask statements sent to the reasoner is the set of results of the Xcerpt construct rule $c' \leftarrow Q$ (where evaluating Q results in the set Θ_Q, as previously). 2. The results of the extended construct rule are the results of an Xcerpt construct rule $c \leftarrow Q'$ under an assumption that the results of Q' are Θ (defined as previously).

The semantics of an extended goal rule $c \leftarrow (q, c'), Q$ is similar to that of the extended construct rule $c \leftarrow (q, c'), Q$. The difference is — as in Xcerpt — that the goal rule produces only one answer (from the set of answers of the construct rule).

The new *WHERE* part in the extended rule allows to ask an ontology arbitrary queries expressible in DIG. One category of such queries are Boolean queries for which answer *true* or *false* (or *error*) can be obtained. This kind of queries can be used to filter out some data from the XML document based on the ontological information. For example, an extended rule can be used to filter out recipes which are gluten-free. In such case, the rule would have a query term $true[[\,]]$ as the *digResponseQuery*, thus it would filter out those answer substitutions for the variables in the *body* for which the corresponding reasoner answer was not *true*. It seems that a need for such filtering is relatively common. Hence, to simplify syntax, we assume that the *digResponseQuery* in the *WHERE* part is optional and by default it is a query term $true[[\,]]$.

We explained syntax and semantics of an extended rule of DigXcerpt. The semantics of a standard Xcerpt rule is as in Xcerpt; see Section 2.1 for an informal explanation, and [4] or [7] for a formal definition. The semantics of a DigXcerpt program is defined in terms of the semantics of rules as in Xcerpt [7]. (In particular, for programs with recursion the stratifiability restrictions from [7, Section 6.4] apply.)

Example 4 (Boolean ontology query) *Consider the XML document* recipes.xml *from Example 2 and the culinary ingredients ontology from Figure 1. We assume that the ontology is loaded into an ontology reasoner with which we can communicate using DIG. We also assume that the names of the ingredients used in the XML document are defined by the ontology. We want to find all the recipes in the XML document which are not gluten-free. This can be achieved using a rule:*

```
CONSTRUCT
 bad-recipes[ all name[ var R ] ]
WHERE
 dig[ subsumes[
        catom[ attr{ name[ "gluten-containing" ]}],
        catom[ attr{ name[ var I ] } ] ] ]
FROM
 in[ resource[ "file:recipes.xml" ],
     desc recipe[[
       name[ var R ], ingredient[[ name[ var I ] ]] ]]  ]
END
```

The body of the rule (the FROM part) extracts the names of recipes together with their ingredients and assigns respective substitutions to the variables R, I. This results in an answer substitution set Θ_Q. Based on Θ_Q the digAskConstruct `subsumes[...]` *constructs DIG ask statements asking whether particular ingredients (values of the variable I) are* gluten-containing. digResponseQuery *is omitted in the* WHERE *part which means that its default value* `true[[ ]]` *is used. The final set Θ of answer substitutions which are applied to the head of the rule contains those substitutions from Θ_Q for which the reasoner answer for the corresponding DIG ask statement was* true *i.e. the substitutions where the variable I is bound to data terms representing* gluten-containing *ingredients:* flour *and* spaghetti. *As these ingredients occur in* Recipe2 *and* Recipe3 *the final result of the rule is*

```
bad-recipes[ name[ "Recipe2" ], name[ "Recipe3 " ] ]
```
□

Example 5 (Non Boolean ontology query) *Consider the ingredients ontology (Figure 1) extended with a class* vitamin*, its three subclasses:* A,B,C *and a property* contained_in.

The extended ontology contains also axioms which indicate in which ingredients a particular vitamin is contained. The axioms state that the vitamin A is contained in tomato, vitamin B in tomato, orange, flour and spaghetti, and vitamin C in orange and tomato. For example, using description logics syntax, one of the axioms can be expressed as $A \sqsubseteq \exists \mathit{contained\_in}.\mathit{tomato}$.

The following DigXcerpt rule queries the document recipes.xml *and the ontology to provide a list of vitamins for each recipe in the document. The* WHERE *part of the rule contains a construct term* `descendants[...]` *producing ontology queries which ask about vitamins included in a particular ingredient. The reasoner answers are queried by the query term* `conceptSet[[...]]` *from the* WHERE *part.*

```
CONSTRUCT
 vit-recipes[ all recipe[ var R, all var V ] ]
WHERE
 dig[
   conceptSet [[ synonyms[[ catom[ attr{ name[var V] } ] ]] ]],
   descendants[
     some[ ratom[ attr{ name["contained_in"] } ],
     catom[ attr{ name[var I] } ] ]   ]   ]
FROM
 in[ resource[ "file:recipes.xml" ],
     desc recipe[[
       name[var R], ingredient[[ name[var I] ]] ]]  ]
END
```

The result of the rule is:

```
vit-recipes[ recipe["Recipe1", "B","C" ],
             recipe["Recipe2", "B"],
             recipe["Recipe3", "A","B","C" ] ]
```
□

4 Implementation of DigXcerpt

This section presents a way DigXcerpt can be implemented on the top of Xcerpt engine i.e. without any modification of Xcerpt implementation. Evaluation of a DigXcerpt program can be organized as a sequence of executions of Xcerpt programs and ontology queries. This can be implemented in a rather simple way; an implementation iteratively invokes an Xcerpt system and an ontology reasoner with a DIG interface.

We begin with discussing implementation of DigXcerpt programs where there is no recursion over extended rules i.e. no extended rule depends on itself (directly or indirectly)[4]. We call such programs non *DIG recursive programs*.

Let P be a non DIG recursive DigXcerpt program and $e_1, \ldots, e_n$ be the extended rules from P such that each rule e_i does not depend on any rule $e_{i+1}, \ldots, e_n$. The ordering $e_1, \ldots, e_n$ can be obtained by topological sorting of the dependency graph for the extended rules in P.

The first step is to compile the extended Xcerpt rules of P into pairs of rules. Each

[4]For a definition of rule dependency in Xcerpt see [7]. If a rule p does not depend on a rule p' then p does not use the data produced by p'.

extended construct rule e_i of P

```
CONSTRUCT
   head
WHERE
   dig [ digResponseQuery, digAskConstruct ]
FROM
   body
END
```

is translated into an Xcerpt rule dr_i, called *DIG response rule*:

```
CONSTRUCT
   head
FROM
   idᵢ[ digResponseQuery, context[var X₁, ..., var Xₗ] ]
END
```

and an Xcerpt goal dg_i, called *DIG ask goal*:

```
GOAL
 idᵢ [ all dig[ digAskConstruct, context[var X₁, ..., var Xₗ] ] ]
FROM
 body
END
```

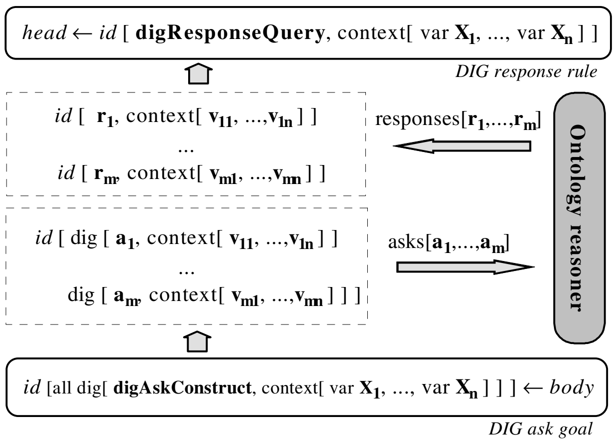
If the extended rule e_i is a goal rule then the translation is analogical, just the first keyword is GOAL instead of CONSTRUCT. Hence the obtained DIG response rule is a goal rule.

The DIG ask goal is used to produce DIG ask statements to be sent to the reasoner and the DIG response rule is used to capture the reasoner responses. $X_1, \ldots, X_l$ are the variables occurring in the query *body*. The term *context*[...] is used here to pass the values of the variables from the *body* of the ask rule to the *head* of the response rule. The construct `all` in the ask goal is added to collect all the results of the query $body$.

The purpose of the labels $id_1, \ldots, id_n$ is 1. to associate DIG ask goals with the corresponding DIG response rules, and 2. to distinguish the data produced by the rules of P from the data related to the implementation of extended rules. So it is required that $id_1, \ldots, id_n$ are distinct and that no id_i occurs in P as the label of the head of a non goal rule of P. (Moreover, if the head of some non goal rule of P is a variable then no id_i can occur in the data to which P is applied.) Figure 3 shows how DIG ask and response rules are evaluated.

Out of the DigXcerpt program P we construct an Xcerpt program P', which is P with each extended rule e_i replaced by the corresponding DIG response rule dr_i. Let P'' be P' with all the goal rules removed. Then a sequence of Xcerpt programs $P_0, \ldots, P_n$ is constructed, where P_0 is $P'' \cup \{dg_1\}$ (and $P_1, \ldots, P_n$ are described later on). For $i = 1, \ldots, n$ we proceed as follows. (We do not distinguish between XML elements and their representation as data terms.)

- Program P_{i-1} is executed by Xcerpt. A data term $id_i[dig[a_1, c_1], \ldots, dig[a_m, c_m]\,]$ is obtained (it is produced by a goal rule dg_i), where $a_1, \ldots, a_m$ are DIG ask statements. Out of $a_1, \ldots, a_m$ a DIG ask request is built. (The DIG ask request is an XML document additionally containing a header with DIG namespace declarations, and unique identifiers for the elements corresponding to $a_1, \ldots, a_m$.)



head ← *id* [**digResponseQuery**, context[var **X₁**, ..., var **Xₙ**]]
DIG response rule
id [**r₁**, context[**v₁₁**, ...,**v₁ₙ**]]
...
id [**rₘ**, context[**vₘ₁**, ...,**vₘₙ**]]
responses[**r₁**,...,**rₘ**]
id [dig [**a₁**, context[**v₁₁**, ...,**v₁ₙ**]]
...
dig [**aₘ**, context[**vₘ₁**, ...,**vₘₙ**]]]
asks[**a₁**,...,**aₘ**]
Ontology reasoner
id [all dig[**digAskConstruct**, context[var **X₁**, ..., var **Xₙ**]]] ← *body*
DIG ask goal

Figure 3: Evaluation of DIG rules: DIG ask goal produces DIG ask statements which are sent to a reasoner; DIG response rule queries data terms constructed out of the reasoner's responses.

- The DIG ask request is sent to the DIG reasoner. The reasoner replies with a response that (after removing its attributes) is $\textit{responses}[r_1, \ldots, r_m]$, where each r_i is an answer for a_i. A set of Xcerpt facts (rules with empty bodies) $R_i = \{\, id_i[r_1, c_1], \ldots, id_i[r_m, c_m] \,\}$ is constructed. (The set contains the results from the reasoner together with the corresponding context information, to be queried by the DIG response rule dr_i. The results of executing the rule dr_i in an Xcerpt program $\{dr_i\} \cup R_i$ are the same as the results of executing e_i in program P according the the semantics described in the previous section.)

- If $1 \leq i < n$ then $P_i = P'' \cup \bigcup_{j=1}^{i} R_j \cup \{dg_{i+1}\}$. (Program P_i contains the reasoner results obtained up to now. They are to be queried by the DIG response rules $dr_1, \ldots, dr_i$. The goal of P_i is dg_{i+1} in order to produce the next query to the reasoner.)

  If $i = n$ then P_n is the Xcerpt program $P'' \cup \bigcup_{j=1}^{n} R_j$ augmented with those goal rules from P' which are not DIG ask goals.

As the last step, P_n is executed by Xcerpt, producing the final results of P.

The results are the same as those described by the DigXcerpt semantics of Section 3. (We skip a formal justification of this fact.) As an additional consequence we obtain that the results do not depend of the ordering of $e_1, \ldots, e_n$ (which may be not unique).

Example 6 *Here we illustrate an evaluation of a simple DigXcerpt program. Consider a program P consisting of the extended rule from Example 4, changed into a goal rule:*

```
GOAL
 bad-recipes[ all name[ var R ] ]
WHERE
 dig[ subsumes[
        catom[ attr{ name["gluten-containing"] } ],
        catom[ attr{ name[ var I ] } ]   ]   ]
FROM
 in[ resource[ "file:recipes.xml" ],
     desc recipe[[
       name[ var R ], ingredient[[ name[ var I ] ]] ]] ]
END
```

First, a program P' is constructed, which consists of a DIG response goal and a DIG ask goal:

```
GOAL
 bad-recipes[ all name[ var R ] ]
FROM
 #g1[ true[[ ]], context[ var I, var R ] ]
END

GOAL
#g1[ all dig[
       subsumes[
        catom[ attr{ name["gluten-containing"] } ],
        catom[ attr{ name[ var I ] } ]  ],
       context[ var I, var R ]  ]  ]
FROM
in[ resource[ "file:recipes.xml" ],
  desc recipe[[ name[ var R ],
                 ingredient[[ name[var I] ]] ]] ]
END
```

As P' contains only goal rules $P'' = \emptyset$. Now a sequence of programs P_0, P_1 is constructed. The program P_0 contains only the DIG ask goal which produces the following data term:

```
#g1[
 dig[ subsumes[
       catom[ attr{ name["gluten-containing"] } ],
       catom[ attr{ name["sugar"] } ]  ],
      context[ "sugar", "Recipe1" ]   ],
 dig[ subsumes[
       catom[ attr{ name["gluten-containing"] } ],
       catom[ attr{ name["orange"] } ]  ],
      context[ "orange", "Recipe1" ]   ],
 dig[ subsumes[
       catom[ attr{ name["gluten-containing"] } ],
       catom[ attr{ name["flour"] } ]  ],
      context[ "flour","Recipe2" ]   ],
 dig[ subsumes[
       catom[ attr{ name["gluten-containing"] } ],
       catom[ attr{ name["salt"] } ]  ],
      context[ "salt","Recipe2" ]   ],
 dig[ subsumes[
       catom[ attr{ name["gluten-containing"] } ],
       catom[ attr{ name["spaghetti"] } ]  ],
      context[ "spaghetti","Recipe3" ]   ],
 dig[ subsumes[
       catom[ attr{ name["gluten-containing"] } ],
       catom[ attr{ name["tomato"] } ]  ],
      context[ "tomato","Recipe3" ]   ]   ]
```

The data term contains DIG ask statements asking whether particular ingredients are gluten-containing. *The additional information attached to each ask statement (its context) is the name of the ingredient queried about and the corresponding name of the recipe.*

Out of these data terms a DIG ask request (which is an XML document) is built. The request contains six DIG ask statements which according to the DIG syntax are augmented by unique identifiers, here $1, \ldots, 6$; for instance the first of the ask statements is

```
<subsumes id="1">
  <catom name="gluten-containing"/>
  <catom name="sugar"/>
</subsumes>
```

The DIG ask request is sent to the ontology reasoner. Its XML answer represented by a data term is (the attributes of the element responses *are removed):*

```
responses[
  false[ attr{id["1"]} ], false[ attr{id["2"]} ],
  true [ attr{id["3"]} ], false[ attr{id["4"]} ],
  true [ attr{id["5"]} ], false[ attr{id["6"]} ] ]
```

Based on the answer the following set R_1 of data terms is constructed:

```
#g1[ false[ attr{ id["1"] } ], context[ "sugar", "Recipe1" ] ]
#g1[ false[ attr{ id["2"] } ], context[ "orange", "Recipe1" ] ]
#g1[ true [ attr{ id["3"] } ], context[ "flour", "Recipe2" ] ]
#g1[ false[ attr{ id["4"] } ], context[ "salt", "Recipe2" ] ]
#g1[ true [ attr{ id["5"] } ], context[ "spaghetti", "Recipe3" ] ]
#g1[ false[ attr{ id["6"] } ], context[ "tomato", "Recipe3" ] ]
```

The final program P_1 to be evaluated by Xcerpt consists of the DIG response goal from P' and the set of data terms R_1. The result of P_1 is the result of the initial program P: `bad-recipes[name["Recipe2"],name["Recipe3"]].` □

DIG recursive programs. In the presented algorithm we assumed that P is a non DIG recursive program. DIG recursive programs, without grouping constructs and negation [7], can be dealt with as follows. Let P be an arbitrary DigXcerpt program and $e_1, \ldots, e_n$ be the extended rules from P not necessarily satisfying the previous condition on their mutual dependencies. Let $E = \{dg_1, \ldots, dg_n\}$ be the set of DIG ask goals corresponding to $e_1, \ldots, e_n$, and P', P'' be as defined earlier. We construct a sequence of Xcerpt programs $P^0, P^1, P^2, \ldots$, where $P^0 = P'' \cup E$, $P^j = P'' \cup E \cup R^j$ for $j > 0$, and R^j is defined below. (R^j represents the reasoner responses for the ask statements produced by the program P^{j-1}.)

Each goal dg_i of P^{j-1} produces a result $id_i[dig[a_1, c_1], \ldots, dig[a_m, c_m]]$. As in the previous approach, a DIG ask request is constructed out of the result. The corresponding response of the reasoner is represented, as previously, by a set of Xcerpt facts $R_i^j = \{\, id_i[r_1, c_1], \ldots, id_i[r_m, c_m] \,\}$. Now $R^j = R_1^j \cup \ldots \cup R_n^j$. It holds that $R^j \subseteq R^{j+1}$ for $j = 1, 2, \ldots$.

The programs $P^0, P^1, \ldots$ are executed (by Xcerpt) until the results of the program P^k are the same as the results of P^{k-1}. Finally, a program Q is $P'' \cup R^k$ augmented with those goal rules from P' that are not DIG ask goals. The results of Q are the same as the results of P described by the semantics of DigXcerpt. See the appendix for a formal justification.

Notice that for non DIG recursive programs this method may be less efficient than the previous one, as it sends more ask requests to the reasoner.

Applying this method to a program with grouping constructs or/and negation may lead to incorrect results. This is because an evaluation of an Xcerpt program with such constructs is a sequence of evaluations of its *strata* [7], in a particular order. In the non DIG recursive case the stratification did not pose any problems. The order of extended rules $e_1, \ldots, e_n$ coincides with the order of strata. Thus the results produced by a goal dg_i added to a program P_j, $j > i$, are the same as the results of dg_i in P_{i-1}. This is not the case for a program that both is DIG recursive and requires stratification. For such programs, the method for handling DIG recursive programs, described above, applies to each stratum separately. The division of a DigXcerpt program into strata and the way of sequential evaluation of strata is the same as in Xcerpt [7]. We omit the details of the algorithm for this case.

The presented algorithm for evaluation of DigXcerpt programs may not terminate for recursive programs. However this is also the case in standard Xcerpt. It is up to the programmer to make sure that a recursive program will terminate.

5 Discussion

We believe that the examples we presented illustrate practical usability of the proposed approach. The examples use arbitrary ontology queries, not only Boolean ones. We put no restriction on usage of DIG. For instance with a query term $error[[\,]]$ used as a *digResponseQuery* a DigXcerpt program can check for which data the reasoner returns an error. Ability to modify ontologies could be added to DigXcerpt, by using *DigAskConstruct* that sends DIG tell statements to the reasoner.

Our approach abstracts from a way XML data is related to an ontology. It is left to a programmer. In our examples XML data is associated with ontology concepts by using common names i.e. XML element names are the same as class names. However, our approach is not restricted to this way of associating. For example, the association may be defined through element attributes.

The semantics of DigXcerpt imposes certain implicit type requirements on programs. The data terms produced by a *digAskConstruct* should be DIG ask statements (represented as data terms). *digResponseQuery* should match data terms being DIG response statements. It is better to check such conditions statically, instead of facing run-time errors. For this purpose the descriptive type system for Xcerpt [4, 11] can be used.

DigXcerpt in its current form requires the programmer to use the verbose syntax of DIG ask and response statements. On the other hand, the tedious details of constructing DIG requests out of DIG ask statements and extracting DIG response statements out of DIG responses are done automatically. Still, one has to specify a construct term *digAskConstruct* for construction of a DIG ask statement, and a query term *digResponseQuery* to match DIG response statements. Dealing with details of DIG syntax may be considered cumbersome and too low level. It may be useful to introduce simpler and more concise syntax for both *digAskConstruct* and *digResponseQuery*. For example, the WHERE part from the rule in Example 5 could be abbreviated as `WHERE V in descendants[ some["contained_in", var I] ]`.

We expect that the approach of this paper can be applied to composing some other XML query languages (such as XQuery) with ontology querying.

The work on implementing DigXcerpt is in progress. A prototype implementation of the Xcerpt extension from our previous work [5] is available on-line[5]. Implementation of DigXcerpt requires only slight modification of that prototype.

6 Summary

The paper addresses the problem of how to use ontological information in the context of querying XML data. The proposed solution extends the XML query language Xcerpt by adding a possibility of querying ontologies. Programs communicate with an ontology reasoner using the DIG interface. No restrictions are imposed on the Xcerpt language and on the DIG ask statements used. In particular, ontologies can be queried with arbitrary, not only Boolean, queries. Data obtained from ontology querying can be used in XML querying, and vice versa. An implementation of DigXcerpt can employ an existing Xcerpt implementation and an existing ontology reasoner; they are treated as "black boxes" (no modifications to the Xcerpt system or the reasoner are needed).

A DigXcerpt Formal Semantics

Now we present a formal semantics with respect to which our implementation approach will be proved sound. We define the semantics of DigXcerpt programs given a semantics of single query rules. We employ the semantics of single Xcerpt rules from [4, 11]. As this semantics deals with a fragment of Xcerpt, in what follows we apply the same restrictions for Xcerpt rules as in [4, 11]. In particular we assume that there is no negation in Xcerpt rules. The semantics of extended rules is as described in Section 3. To avoid complications with stratification, we present semantics for DigXcerpt programs without grouping constructs in non goal rules. We prove our implementation approach to be sound only for such programs.

In what follows, the term *query rule* refers to any rule of Xcerpt or DigXcerpt.

Definition 1 (DigXcerpt program) *A DigXcerpt program $\mathcal{P}$ is a pair (P, G) where P and G are sets of query rules such that $G \subseteq P$ and $|G| > 0$. The query rules from G are called goals.*

In what follows we assume that there exists a fixed ontology loaded to a reasoner to which ontology queries from extended rules and those produced by DIG ask goals refer. Also we assume that for each URI r_i of an external resource occurring in the considered programs, there is a data term $d(r_i)$ associated with r_i. Thus for each query rule p, it queries data terms $d(r_i)$ associated with the URIs occurring in its body and a set of data terms Z produced by the query rules of a program. The set of results of the query rule p and the set of data terms Z is denoted as $res(p, Z)$. For p being an Xcerpt rule the set is defined in [11] by Definition 10 and for p being an extended rule the set is defined in Section 3. For a set of rules P, $res(P, Z)$ is defined as $\bigcup_{p \in P} res(p, Z)$.

Now we are ready to describe the effect of applying a set of rules to a set of data terms, and then the semantics of a program.

[5] `http://www.ida.liu.se/digxcerpt/`

Definition 2 (Immediate consequence operator for rule results) *Let P be a set of DigXcerpt query rules. R_P is a function on sets of data terms such that $R_P(Z) = Z \cup res(P, Z)$.*

We will use the fact that $res(p, \cdot)$ is monotone for any rule p without negation and grouping constructs: if $Z \subseteq Z'$ then $res(p, Z) \subseteq res(p, Z')$. As its consequence we obtain:

Lemma 1 *Let P be a set of query rules without grouping constructs and let Z, Z' be sets of data terms. If $Z \subseteq Z'$ then $R_P(Z) \subseteq R_P(Z')$, $R_P^j(Z) \subseteq R_P^j(Z')$, and $R_P^j(Z) \subseteq R_P^k(Z)$ for $0 \leq j \leq k$.*

Definition 3 (Rule result, no grouping constructs in non goal rules) *Let $\mathcal{P} = (P, G)$ be a DigXcerpt program, such that there is no grouping construct in P', where $P' = P \backslash G$. A data term d is a* **result of a rule** *p in P' if $d \in res(p, R_{P'}^i(\emptyset))$ for some $i \geq 0$. Let $k \geq 0$ be a number such that $R_{P'}^k(\emptyset) = R_{P'}^{k+1}(\emptyset)$. A data term d is a* **result of a goal rule** *g in G and d is a* **result of the program** *$\mathcal{P}$ if $d \in res(g, R_{P'}^k(\emptyset))$.*

The set $R_{P'}^k(\emptyset)$ is denoted as $R_{P'}^\infty(\emptyset)$ and the set $res(G, R_{P'}^\infty(\emptyset))$ of results of the program is denoted as $res(\mathcal{P})$.

Notice that the definition defines results of programs without infinite loops.

Example 7 *Let $P' = \{ p \}$, where $p = c[X] \leftarrow \texttt{or}[X, \texttt{in}[r, b[X]]$ and $d(r) = b[\mathsf{a}]$. $R_{P'}^i(\emptyset) = \{ c[\mathsf{a}], c[c[\mathsf{a}]], \dots, c^i[\mathsf{a}] \}$, for $i > 0$, and $res(p, R_{P'}^i(\emptyset)) = \{ c[\mathsf{a}], \dots, c^{i+1}[\mathsf{a}] \}$.* □

B Soundness Proof of the Implementation Algorithm

Here we present a soundness proof of the implementation algorithm for recursive programs described in Section 4. Soundness of the algorithm is expressed by Theorem 1.

First we introduce notation used in this section. Let d be a data term of the form $id[dig[a_1, c_1], \dots, dig[a_m, c_m]]$, where *id* is one of the unique labels introduced by translation of a DigXcerpt program into an Xcerpt program. The data term d contains DIG ask statements $a_1, \dots, a_m$. The corresponding set of reasoner responses, denoted as $RR(d)$, is $\{ id[r_1, c_1], \dots, id[r_m, c_m] \}$, where $r_1, \dots, r_n$ are the reasoner responses (DIG response statements) for $a_1, \dots, a_n$, respectively. The data terms of the form $id[r_i, c_i]$ are called DIG response terms. For a set of data terms Z, $RR(Z) = \bigcup_{d \in Z} RR(d)$. For a set of data terms R, $r(R)$ denotes rules with empty bodies representing the data terms from R.

For a set of rules P a (length m) *computation* of P is a sequence: $Z_0, p_1, Z_1, \dots, Z_{m-1}, p_m, Z_m$, where $Z_0 = \emptyset$, $Z_j = Z_{j-1} \cup res(p_j, Z_{j-1})$ and $p_j \in P$, for $j = 1, \dots, m$. Notice that the sets $Z_0, \dots, Z_m$ are finite, and that $Z_j \subseteq R_P^j(\emptyset)$ for $j = 0, 1, \dots$ and a P without grouping constructs. (The latter is due to $res(p, Z) \subseteq R_P(Z) \subseteq R_P(Z')$ for any $p \in P$ and $Z \subseteq Z'$, by Lemma 1.) Sometimes the computation will be abbreviated as $Z_0, P_1, Z_1, \dots, Z_{m'-1}, P_{m'}, Z_{m'}$, where each $P_i \subseteq P$ is a set of rules not pairwise dependent (thus the order of execution of rules from P_i is irrelevant) and $Z_0 = \emptyset$, $Z_i = res(P_i, Z_{i-1})$, for $i = 1, \dots, m'$.

Given a computation $\emptyset, \ldots, Z$, the set Z is called the result of the computation. A computation $\emptyset, \ldots, Z$ of P is called final if $Z = R_P^\infty(\emptyset)$. Thus an existence of a final computation of P guarantees that there is no infinite loop in P and that $R_P^\infty(\emptyset)$ exists. Also existence of $R_P^\infty(\emptyset)$ guarantees that a final computation of P exists.

Proposition 1 *Let P be a set of query rules without grouping constructs and $\emptyset, \ldots, Z$, be a computation of P. If $res(p, Z) \subseteq Z$ for each $p \in P$ then $Z = R_P^\infty(\emptyset)$.*

Proof As $res(p, Z) \subseteq Z$ for each $p \in P$, $\bigcup_{p \in P} res(p, Z) \subseteq Z$. Thus $R_P(Z) = Z \cup \bigcup_{p \in P} res(p, Z) \subseteq Z$. Hence by monotonicity of R_P^i (Lemma 1), $R_P^{i+1}(Z) \subseteq R_P^i(Z)$ for $i \geq 0$ and $R_P^l(Z) \subseteq Z$ for any $l \geq 0$. As Z is finite, $R_P^\infty(\emptyset)$ exists. Thus $R_P^\infty(\emptyset) \subseteq Z$ and by Lemma 1, $Z \subseteq R_P^\infty(\emptyset)$. Hence $Z = R_P^\infty(\emptyset)$.

□

In what follows we will use the following Lemmata, whose rather obvious proofs we skip.

Lemma 2 *Let Z be a set of data terms, e be an extended DigXcerpt rule and dg, dr be the corresponding DIG ask goal and DIG response rules. Then $res(e, Z) = res(dr, RR(res(dg, Z)))$.*

Lemma 3 *Let p be a DigXcerpt rule and Z, Z' be sets of data terms such that $Z \subseteq Z'$. If there is no grouping construct and no negation in p then $res(p, Z) \subseteq res(p, Z')$.*

Lemma 4 *Let dg be a DIG goal rule without grouping constructs in* digAskConstruct *and Z, Z' be sets of data terms. If $Z \subseteq Z'$ then $RR(res(dg, Z)) \subseteq RR(res(dg, Z'))$.*

Lemma 5 *Let $\mathcal{P} = (P, G)$ be a DigXcerpt program and $\mathcal{P}' = (P', G')$ be $\mathcal{P}$ with each extended rule e_i replaced by the corresponding DIG response rule dr_i. Let $E = \{dg_1, \ldots, dg_n\}$ be the set of DIG ask goals corresponding to $e_1, \ldots, e_n$ and Z be a set of data terms. Then $res(dr_i, RR(res(E, Z))) = res(dr_i, RR(res(dg_i, Z)))$.*

Lemma 6 *Let $\mathcal{P} = (P, G)$ be a DigXcerpt program and $\mathcal{P}' = (P', G')$ be $\mathcal{P}$ with each extended rule e_i replaced by the corresponding DIG response rule dr_i. Let $E = \{dg_1, \ldots, dg_n\}$ be the set of DIG ask goals corresponding to $e_1, \ldots, e_n$ and Z, Z' be sets of data terms. Let $R = RR(res(E, Z)))$ and $p \in P$. Then $res(p, Z' \cup R) = res(p, Z')$ and $res(E, Z' \cup R) = res(E, Z')$. For Z'' being a set of data terms without DIG response terms $res(dr_i, Z'' \cup R) = res(dr_i, R)$.*

Lemma 7 *Let $\mathcal{P} = (P, G)$ be a DigXcerpt program without grouping constructs. Let $\mathcal{P}' = (P', G')$ be $\mathcal{P}$ with each extended rule e_i replaced by the corresponding DIG response rule dr_i. Let $E = \{dg_1, \ldots, dg_n\}$ be the set of DIG ask goals corresponding to $e_1, \ldots, e_n$ and P'' be $P' \backslash G'$. Let $\mathcal{P}^0, \ldots, \mathcal{P}^{k+1}$ be a sequence of Xcerpt programs such that for $j = 0, \ldots, k+1$,*

- *$\mathcal{P}^j = (P^j \cup E, E)$, $P^j = P'' \cup r(R^j)$, $R^0 = \emptyset$, if $j \leq k$ then there exists a final computation for P^j, and $R^j = RR(res(\mathcal{P}^{j-1}))$ if $j > 0$,*
- *$R^{k+1} = R^k$.*

Let $S = \emptyset, p_1, Z_1, p_2, Z_2, \ldots, p_m, Z_m$ be a computation of $P \backslash G$. There exists a computation $S' = \emptyset, q_1, W_1, q_2, W_2, \ldots, q_{m'}, W_{m'}$ of P^k such that $Z_m \subseteq W_{m'}$.

Proof Proof by induction on m. For $m = 0$, $S = S' = \emptyset$, thus $Z_m = W_{m'} = \emptyset$.

Induction step. Let $S = S_-, p_m, Z_m$ be a length m computation of $P\backslash G$, where $S_- = \emptyset, \ldots, Z_{m-1}$ is a computation of length $m-1$. By the inductive assumption, there exists a computation $S'_- = \emptyset, \ldots, W_l$ of P^k, such that $Z_{m-1} \subseteq W_l \subseteq R^{\infty}_{P^k}(\emptyset)$.

Let p_m be an Xcerpt rule p. Then $S' = S'_-, p, W_{m'}$. By Lemma 3, as $Z_{m-1} \subseteq W_l$, $Z_m = Z_{m-1} \cup res(p, Z_{m-1}) \subseteq W_l \cup res(p, W_l) = W_{m'}$.

Let p_m be an extended rule e and dr and dg be a DIG response rule and DIG ask goal, respectively, corresponding to e.

$$\begin{array}{ll}
res(e, Z_{m-1}) & \\
= res(dr, RR(res(dg, Z_{m-1}))) & \text{by Lemma 2} \\
\subseteq res(dr, RR(res(dg, R^{\infty}_{P^k}(\emptyset)))) & \text{by Lemmata 3, 4, as } Z_{m-1} \subseteq R^{\infty}_{P^k}(\emptyset) \\
\subseteq res(dr, RR(res(E, R^{\infty}_{P^k}(\emptyset)))) & \text{by Lemma 3 and the definition of } res(E, Z) \\
= res(dr, R^{k+1}) & \text{by the definition of } R^{k+1} \\
= res(dr, R^k) & \text{as } R^{k+1} = R^k \\
\subseteq res(dr, W_l \cup R^k) & \text{by Lemma 3}
\end{array}$$

We construct $S' = S'_-, r(R^k), W_l \cup R^k, dr, W_{m'}$, where $W_{m'} = W_l \cup R^k \cup res(dr, R^k \cup W_l)$.

The result of S is $Z_m = Z_{m-1} \cup res(e, Z_{m-1})$. As $Z_{m-1} \subseteq W_l$ and $res(e, Z_{m-1}) \subseteq res(dr, W_l \cup R^k)$, we have $Z_m \subseteq W_l \cup res(dr, W_l \cup R^k) \subseteq W_l \cup R^k \cup res(dr, W_l \cup R^k) = W_{m'}$.

□

Corollary 1 $R^{\infty}_{P\backslash G}(\emptyset) \subseteq R^{\infty}_{P^k}(\emptyset)\backslash R^k$.

Proof Let S be a computation such that $Z_m = R^{\infty}_{P\backslash G}(\emptyset)$. By Lemma 7, we have $Z_m \subseteq W_{m'} \subseteq R^{\infty}_{P^k}(\emptyset)$. Thus $R^{\infty}_{P\backslash G}(\emptyset) \subseteq R^{\infty}_{P^k}(\emptyset)$. As the labels from R^k are unique identifiers $R^{\infty}_{P\backslash G}(\emptyset) = R^{\infty}_{P\backslash G}(\emptyset)\backslash R^k$. Hence $R^{\infty}_{P\backslash G}(\emptyset) \subseteq R^{\infty}_{P^k}(\emptyset)\backslash R^k$.

□

Lemma 8 *Let $\mathcal{P} = (P, G)$ be a DigXcerpt program without grouping constructs and let $\mathcal{P}' = (P', G')$ be $\mathcal{P}$ with each extended rule e_i replaced by the corresponding DIG response rule dr_i. Let $E = \{dg_1, \ldots, dg_n\}$ be the set of DIG ask goals corresponding to $e_1, \ldots, e_n$ and P'' be $P'\backslash G'$. Let $\mathcal{P}^0, \ldots, \mathcal{P}^k$ ($k \geq 0$), be a sequence of Xcerpt programs such that for $j = 0, \ldots, k$,*

- *$\mathcal{P}^j = (P^j \cup E, E)$, $P^j = P'' \cup r(R^j)$, $R^0 = \emptyset$, if $j < k$ then there exists a final computation for P^j, and $R^j = RR(res(\mathcal{P}^{j-1}))$ if $j > 0$.*

Let $S = \emptyset, r(R^j), R^j, p_1, W_1, \ldots, p_m, W_m$ be a computation of P^j. There exists a computation $S' = \emptyset, \ldots, Z_{m'}$ of $P\backslash G$ such that $W_m\backslash R^k \subseteq Z_{m'}$.

Proof Proof by induction on j.

$j = 0$.

Let $S = \emptyset, p_1, W_1, \ldots, p_m, W_m$ be a computation of P^0. As none of $W_0, \ldots, W_m$ contain DIG response terms, by Lemma 6, if p_i ($i = 1, \ldots, m$) is a DIG response rule then $W_{i-1} = W_i$. Thus, by removing DIG response rules from S we obtain a computation

$S'' = \emptyset, \ldots, W_m$ of P^0 and of $P\backslash G$ with the same result as S. Thus $S' = S''$ and $W_m = Z_{m'}$. Hence $W_m \backslash R^k \subseteq Z_{m'}$.

$j > 0$

Let $S_{j-1} = \emptyset, r(R^{j-1}), R^{j-1}, p_1, W'_1, \ldots, p_s, W'_s$ be a computation of P^{j-1} such that $W'_s = R^{\infty}_{Pj-1}(\emptyset)$. By the inductive assumption there exists a computation $S'_{j-1} = \emptyset, \ldots, Z'_{s'}$ of $P\backslash G$ such that $W'_s \backslash R^k \subseteq Z'_{s'}$.

Proof by induction on the length m of S. We construct a computation $S' = \emptyset, \ldots, Z_{m'}$ such that $W_m \backslash R^k \subseteq Z_{m'}$ and $R^{\infty}_{Pj-1}(\emptyset) \backslash R^k \subseteq Z_{m'}$. For $m = 0$, $S' = S_{j-1}$.

By the inductive assumption there exists a computation $S'_{-} = \emptyset, \ldots, Z_{m''}$ of $P\backslash G$ such that $W_{m-1} \backslash R^k \subseteq Z_{m''}$ and $R^{\infty}_{Pj-1}(\emptyset) \backslash R^k \subseteq Z_{m''}$.

Let p_m be not a DIG response rule. Then $W_m = W_{m-1} \cup res(p_m, W_{m-1}) = W_{m-1} \cup res(p_m, W_{m-1} \backslash R^k)$. Now $S' = S'_{-}, p_m, Z_{m'}$, where $Z_{m'} = Z_{m''} \cup res(p_m, Z_{m''})$. As $W_{m-1} \backslash R^k \subseteq Z_{m''}$, by Lemma 3, $W_m \backslash R^k \subseteq Z_{m'}$.

Let p_m be a DIG response rule dr and e be the corresponding extended rule from $P\backslash G$. Then $W_m = W_{m-1} \cup res(dr, W_{m-1})$. $S' = S'_{-}, e, Z_{m'}$ where $Z_{m'} = Z_{m''} \cup res(e, Z_{m''})$.

$$\begin{array}{ll}
res(dr, W_{m-1}) & \\
= res(dr, R^j) & \text{by Lemma 6} \\
= res(dr, RR(res(E, R^{\infty}_{Pj-1}(\emptyset)))) & \text{by the definition of } R^j \\
= res(dr, RR(res(dg, R^{\infty}_{Pj-1}(\emptyset)))) & \text{by Lemma 5} \\
= res(dr, RR(res(dg, R^{\infty}_{Pj-1}(\emptyset) \backslash R^k))) & \text{by Lemma 6} \\
\subseteq res(dr, RR(res(dg, Z_{m''}))) & \text{by Lemmata 3, 4, as } R^{\infty}_{Pj-1}(\emptyset) \backslash R^k \subseteq Z_{m''} \\
= res(e, Z_{m''}) & \text{by Lemma 2}
\end{array}$$

Thus, by the inductive assumption, $W_m \backslash R^k \subseteq Z_{m'}$. □

Corollary 2 $R^{\infty}_{P\backslash G}(\emptyset) \supseteq R^{\infty}_{Pk}(\emptyset) \backslash R^k$.

Proof Let S be a computation such that $W_m = R^{\infty}_{Pk}(\emptyset)$. As $W_m \backslash R^k \subseteq Z_{m'} \subseteq R^{\infty}_{P\backslash G}(\emptyset)$, $R^{\infty}_{Pk}(\emptyset) \backslash R^k \subseteq R^{\infty}_{P\backslash G}(\emptyset)$. □

Lemma 9 *Let $\mathcal{P} = (P, G)$ be a DigXcerpt program without grouping constructs such that there exists a final computation for $P\backslash G$. Let $\mathcal{P}' = (P', G')$ be $\mathcal{P}$ with each extended rule e_i replaced by the corresponding DIG response rule dr_i. Let $E = \{dg_1, \ldots, dg_n\}$ be the set of DIG ask goals corresponding to $e_1, \ldots, e_n$ and P'' be $P'\backslash G'$. Then*

1. *there exists a sequence of Xcerpt programs $\mathcal{P}^0, \mathcal{P}^1, \ldots$, such that, for $j \geq 0$, $\mathcal{P}^j = (P^j \cup E, E)$, $P^j = P'' \cup r(R^j)$, $R^0 = \emptyset$, $R^j = RR(res(\mathcal{P}^{j-1}))$ if $j > 0$, and there exists a final computation for P^j,*

2. *$R^i \subseteq R^{i+1}$ for $i \geq 0$,*

3. *there exists $k \geq 0$ such that $R^k = R^{k+1}$.*

Proof Let $S = \emptyset \ldots, W$ be a final computation of $P \backslash G$.

1. Notice that, for any $m > 0$, a sequence $\mathcal{P}^0, \mathcal{P}^1, \ldots, \mathcal{P}^m$ of programs exists iff there exist final computations for programs $\mathcal{P}^0, \mathcal{P}^1, \ldots, \mathcal{P}^{m-1}$. Proof by contradiction. Assume that there is no final computation for some P^j and take the first P^j for which there is no final computation. Thus, there exist final computations for the sets $P^0, P^1, \ldots, P^{j-1}$. As there is no final computation for P^j we can construct an infinite computation $\emptyset, \ldots, W'_t, \ldots, W'_{t+1}, \ldots$ such that $W'_t \subset W'_{t+1} \subset \ldots$. However, by Lemma 8, for any computation $S' = \emptyset, \ldots, W'$ of P^j, $W' \backslash R^j \subseteq W$. As W is finite we get a contradiction.

2. By induction: $\emptyset = R^0 \subseteq R^1$. If $R^j \subseteq R^{j+1}$ then $P^j \subseteq P^{j+1}$, hence $res(\mathcal{P}^j) \subseteq res(\mathcal{P}^{j+1})$ and $R^{j+1} \subseteq R^{j+2}$.

3. Proof by contradiction. Assume that $R^k \neq R^{k+1}$ for any $k \geq 0$. Thus $RR(res(\mathcal{P}^{k-1})) \neq RR(res(\mathcal{P}^k))$ for any $k \geq 1$ and then $res(\mathcal{P}^{k-1}) \neq res(\mathcal{P}^k)$. By the definition of $res(\mathcal{P}^k)$, $res(E, R^\infty_{P^{k-1}}(\emptyset)) \neq res(E, R^\infty_{P^k}(\emptyset))$. By Lemma 6, $res(E, R^\infty_{P^{k-1}}(\emptyset)) = res(E, R^\infty_{P^{k-1}}(\emptyset) \backslash R^k)$ and $res(E, R^\infty_{P^k}(\emptyset)) = res(E, R^\infty_{P^k}(\emptyset) \backslash R^k)$. Thus $R^\infty_{P^{k-1}}(\emptyset) \backslash R^k \neq R^\infty_{P^k}(\emptyset) \backslash R^k$.

   As $R^{k-1} \subseteq R^k$, $P^{k-1} \subseteq P^k$ and then $R^\infty_{P^{k-1}}(\emptyset) \subseteq R^\infty_{P^k}(\emptyset)$ (from Lemma 1 and the definitions of R_P and of $res(P, Z)$, by induction). Hence $R^\infty_{P^{k-1}}(\emptyset) \backslash R^k \subseteq R^\infty_{P^k}(\emptyset) \backslash R^k$ and, as $R^\infty_{P^{k-1}} \backslash R^k \neq R^\infty_{P^k}(\emptyset) \backslash R^k$, we have $R^\infty_{P^{k-1}}(\emptyset) \backslash R^k \subset R^\infty_{P^k}(\emptyset) \backslash R^k$. However, by Lemma 8, $R^\infty_{P^k}(\emptyset) \backslash R^k \subseteq W$ for any $k \geq 1$. So W is infinite. Contradiction. Hence there exists k such that $R^k = R^{k+1}$.

□

Theorem 1 *Let $\mathcal{P} = (P, G)$ be a DigXcerpt program without negation in rule bodies and without grouping constructs and such that there exists a final computation of $P \backslash G$. Let $\mathcal{P}' = (P', G')$ be $\mathcal{P}$ with each extended rule e_i replaced by the corresponding DIG response rule dr_i. Let $E = \{dg_1, \ldots, dg_n\}$ be the set of DIG ask goals corresponding to $e_1, \ldots, e_n$ and P'' be $P' \backslash G'$. Then there exists $k \geq 0$, a sequence of Xcerpt programs $\mathcal{P}^0, \ldots, \mathcal{P}^{k+1}$, and a sequence of sets $R^0, \ldots, R^{k+1}$ such that*

- *for $j = 0, \ldots, k+1$: $\mathcal{P}^j = (P^j \cup E, E)$, $P^j = P'' \cup r(R^j)$, $R^0 = \emptyset$, there exists a final computation for P^j, and $R^j = RR(res(\mathcal{P}^{j-1}))$ if $j > 0$,*
- *$R^k = R^{k+1}$.*

Let $\mathcal{Q} = (P^k \cup G', G')$. Then $R^\infty_{P \backslash G}(\emptyset) = R^\infty_{P^k}(\emptyset) \backslash R^k$ and $res(\mathcal{P}) = res(\mathcal{Q})$.

Proof By Lemma 9, there exist a sequence $P^0, \ldots, P^k$ and a sequence $R^0, \ldots, R^{k+1}$ such that $R^k = R^{k+1}$. By Lemma 9, there exist final computations for $P^0, \ldots, P^k$. By Corollary 1, $R^\infty_{P \backslash G}(\emptyset) \subseteq R^\infty_{P^k}(\emptyset) \backslash R^k$. By Corollary 2, $R^\infty_{P \backslash G}(\emptyset) \supseteq R^\infty_{P^k}(\emptyset) \backslash R^k$. Hence $R^\infty_{P \backslash G}(\emptyset) = R^\infty_{P^k}(\emptyset) \backslash R^k$.

Let $Z = R^\infty_{P^k}(\emptyset) \backslash R^k = R^\infty_{P \backslash G}(\emptyset)$. As $R^k \subseteq R^\infty_{P^k}(\emptyset)$, $R^\infty_{P^k}(\emptyset) = Z \cup R^k$.

Let $A = \{eg_1, \ldots, eg_f\} \subseteq G$ be the set of those goals of G which are extended rules and let $B = G \backslash A$. Let $A' = \{dr_1, \ldots, dr_f\}$ be the set of DIG response rules (goals) corresponding to the rules from A and $A'' = \{dg'_1, \ldots, dg'_f\}$ be the set of DIG ask goals corresponding to the rules from A. We have $G = A \cup B$ and $G' = A' \cup B$.

$$
\begin{array}{ll}
res(\mathcal{Q}) & \\
= res(G', R^{\infty}_{\mathcal{P}k}(\emptyset)) & \text{by the definition of } res(\mathcal{Q}) \\
= res(A' \cup B, Z \cup R^k) & \text{as } R^{\infty}_{\mathcal{P}k}(\emptyset) = Z \cup R^k \text{ and } G' = A' \cup B \\
= res(A', Z \cup R^k) \cup res(B, Z \cup R^k) & \text{by the definition of } res(P, Z) \\
= res(A', R^k) \cup res(B, Z) & \text{by Lemma 6} \\
= res(A', R^{k+1}) \cup res(B, Z) & \text{as } R^{k+1} = R^k \\
= res(A', RR(res(\mathcal{P}^k))) \cup res(B, Z) & \text{by the definition of } R^{k+1} \\
= res(A', RR(res(E, Z \cup R^k))) \cup res(B, Z) & \text{by the definition of } res(\mathcal{P}^k) \\
= res(A', RR(res(E, Z))) \cup res(B, Z) & \text{by Lemma 6} \\
= \bigcup_{i=1}^{f} res(dr_i, RR(res(dg'_i, Z)) \cup res(B, Z) & \text{by the def. of } res(P, Z) \text{ and Lemma 5} \\
= \bigcup_{e \in A} res(e, Z) \cup res(B, Z) & \text{by Lemma 2} \\
= res(A, Z) \cup res(B, Z) & \text{by the definition of } res(P, Z) \\
= res(A \cup B, Z) & \text{by the definition of } res(P, Z) \\
= res(G, Z) & \text{as } G = A \cup B \\
= res(\mathcal{P}) & \text{by the definition of } res(\mathcal{P})
\end{array}
$$

□

References

[1] U. Aßmann, J. Henriksson, and J. Małuszyński. Combining Safe Rules and Ontologies by Interfacing of Reasoners. In *PPSWR 2006*, number 4187 in LNCS, pages 31–43.

[2] P. Barahona, F. Bry, E. Franconi, N. Henze, and U. Sattler. *Reasoning Web 2006. Second International Summer School. Tutorial Lectures.* Springer.

[3] S. Bechhofer. The DIG Description Logic Interface: DIG/1.1. In *Proceedings of DL2003 Workshop*, Rome, 2003.

[4] S. Berger, E. Coquery, W. Drabent, and A. Wilk. Descriptive typing rules for Xcerpt. In *PPSWR 2005*, number 3703 in LNCS, pages 85–100. Springer Verlag.

[5] W. Drabent and A. Wilk. Combining XML querying with ontology reasoning: Xcerpt and DIG, 2006. RuleML-2006 Workshop. Unpublished proceedings `http://2006.ruleml.org/group3.html#3`.

[6] F. Patel-Schneider and J. Siméon. The Yin/Yang web: A unified model for XML syntax and RDF semantics. *IEEE Transactions on Knowledge and Data Engineering*, 15(4):797–812, 2003.

[7] S. Schaffert. *Xcerpt: A Rule-Based Query and Transformation Language for the Web.* PhD thesis, University of Munich, Germany, 2004. `http://www.wastl.net/download/dissertation/dissertation_schaffert.pdf`.

[8] S. Schaffert and F. Bry. Querying the Web Reconsidered: A Practical Introduction to Xcerpt. In *Extreme Markup Languages*, 2004.

[9] E. Svensson and A. Wilk. XML Querying Using Ontological Information. In *PPSWR 2006*, number 4187 in LNCS.

[10] W3 Consortium. XQuery 1.0: An XML Query Language. `http://www.w3.org/TR/xquery/`.

[11] A. Wilk. Descriptive Types for XML Query Language Xcerpt, 2006. Licentiate Thesis. Linkoping University. `http://www.ida.liu.se/~artwi/lic.pdf`.