

Descriptive Typing Rules for Xcerpt and their Soundness

Sacha Berger¹, Emmanuel Coquery², Włodzimierz Drabent^{3,4} and Artur Wilk⁴

¹ Institute for Computer Science, University of Munich, Germany
`sacha.berger@ifi.lmu.de`

² INRIA Rocquencourt and Conservatoire des Arts et Métiers, France
`Emmanuel.Coquery@inria.fr`

³ IPI PAN, Polish Academy of Sciences, Warszawa, Poland

⁴ IDA, Linköping University, 581-83 Linköping, Sweden
`{wdr|artwi}@ida.liu.se`

Abstract. We present typing rules for the Web query language Xcerpt. The rules provide a descriptive type system: the typing of a program is an approximation of its semantics. The rules can also be seen as an abstract form of a type inference algorithm (presented in previous work), and as a stage in a formal soundness proof of the algorithm. The paper considers a substantial fragment of Xcerpt; the main restriction is that we deal with data terms corresponding to trees (instead of general graphs), and we do not deal with Xcerpt rule chaining. We provide a formal semantics for the fragment of Xcerpt and a soundness theorem for the presented type system. The semantics is a basis for a soundness proof of the typing system, the proof is given in a full version of this paper.

1 Introduction

This article presents a type system for the Web and Semantic Web query language Xcerpt [11,5,10], formalized using typing rules in the spirit of [6]. It is an extension and reformulation of the type system presented in the earlier work [12,7]. The type system is *descriptive*, this means a typing approximates the semantics of a program (in an untyped programming language). In descriptive typing, type inference means finding an approximation of the semantics of the given program; type checking means proving program correctness with respect to a specification expressed by means of types. In our case, for a given Xcerpt program and a type of data (i.e. the set of data objects to which the program may be applied) the type system provides a type of the program's results (i.e. a superset of the set of the program's results). This is type inference; if a type of expected results is given then type checking can be performed by checking if the obtained type of results is a subset of the the given one. The main intended application of the proposed type system is discovering errors in Xcerpt programs.

In the previous work [12,7] two descriptive type systems for an **Xcerpt** fragment⁵ have been presented. They are formulated by means of algorithms. This is rather complicated and makes any formal reasoning about the type system difficult. In the present paper we generalize the simpler of these type systems to a bigger fragment of **Xcerpt** (grouping constructs are added). An important difference is that we formulate the type system by means of derivation rules. The rules are similar to proof rules of logic, rules used in operational semantics, and those used in prescriptive typing [6]. Employing rules makes it possible to specify a type system in a formal and concise way. Such approach facilitates formal reasoning; we confirm this by presenting a soundness proof of the type system. The rules may be seen as an abstraction of an algorithm; they abstract from lower level details. Thus – we believe – this formulation of a type system is also easier to understand by humans than the previous one.

To facilitate the soundness proof we provide a formal semantics (based on [12,5,10]) of the fragment of **Xcerpt**. The semantics is substantially simpler than that of a full **Xcerpt** [10] (as it does not use the notion of simulation unification), and may be of separate interest.

Similarly to other work related to **Xcerpt** [11,10] we use *data terms* as an abstraction of semi-structured data [1] of the Web. Data terms generalize the notion of a term: the number of arguments of a symbol is not fixed, moreover a symbol may have an (unordered) set of arguments, instead of an ordered sequence. (This paper does not deal with data terms representing graphs which are not trees). As a formalism to define types we use *type definitions* [12,3]. They are similar to unranked tree automata [2] (and equivalent formalisms), but deal also with the case of unordered children of a tree node. The types defined by type definitions roughly correspond to the sets of documents defined by various schema languages like DTD, XML Schema or Relax NG.[9]

Our descriptive type system uses rules in a similar way as prescriptive type systems [6] do. We expect that this should make possible a formal comparison of the two approaches, and maybe even combining their advantages, thus obtaining a system that can be used for detecting errors, checking program composition and providing a base for documentation. (There is no general agreement about what exactly descriptive and prescriptive typing mean. Roughly speaking, the former deals with an untyped programming language and types approximate program semantics, while in the latter the language is typed and types are an important part of its semantics.)

The article is organized as follows: First, data terms and type definitions are introduced. A short introduction of **Xcerpt** is given afterwards, explaining a substantial fragment of the language and the semantics of the fragment. Then, in Section 4 the type system for the **Xcerpt** fragment is introduced, by (1) motivating the idea of descriptive types for **Xcerpt**, and (2) providing typing rules in the spirit of [6], specifying the type system inductively based on the syntax

⁵ The main **Xcerpt** features excluded are: data terms corresponding to general graphs (which are not trees), grouping constructs (**all**, **some**), negation, and programs consisting of multiple query rules.

of *Xcerpt*. In the Appendix, a soundness proof of the given typing rules with respect to the semantics of *Xcerpt* is presented.

2 Modelling XML Data

We model XML data using a formalism of data terms similar to that defined in [11]. Data terms can be seen as mixed trees which are labelled trees where children of a node are either linearly ordered or unordered. This is related to existence of two basic concepts in XML: *tags* which are nodes of an ordered tree and *attributes* that attach attribute-value mappings to nodes of a tree. These mappings are represented as unordered trees. Unordered children of a node may also be used to abstract from the order of elements, when this order is inessential. We assume that there is no syntactic difference between XML tag names and attribute names and they both are labels of nodes in our mixed trees (and symbols of our data terms). The infinite alphabet of labels will be denoted by $\mathcal{L}$.

A content of an element is a sequence of other elements or **basic constants**. Basic constants are basic values such as attribute values and all “free” data appearing in an XML document – all data that is between start and end tag except XML elements, called PCDATA (short for *parseable character data*) in XML jargon. Basic constants occur as strings in XML documents but they can play a role of data of other types depending on an adequate definition in DTD (or other schema languages) e.g. IDREF, CDATA, The set of basic constants will be denoted by $\mathcal{B}$. In our notation we will enclose all basic constants in quotation marks ” ”.

XML documents are represented as *data terms*.

Definition 1. *A data term is an expression defined inductively as follows:*

- Any basic constant is a data term,
- If l is a label and $t_1, \dots, t_n$ are $n \geq 0$ data terms, then $l[t_1, \dots, t_n]$ and $l\{t_1, \dots, t_n\}$ are data terms.

The linear ordering of children of the node with label l is denoted by enclosing them by brackets $[]$, while unordered children are enclosed by braces $\{\}$.

A **subterm** of a data term t is defined inductively: t is a subterm of t , and any subterm of t_i ($1 \leq i \leq n$) is a subterm of $l'[t_1, \dots, t_n]$ and of $l'\{t_1, \dots, t_n\}$. Data terms $t_1, \dots, t_n$ will be sometimes called the arguments of l' , or the **direct subterms** of $l'[t_1, \dots, t_n]$ (and of $l'\{t_1, \dots, t_n\}$). The **root** of a data term t , denoted $root(t)$, is defined as follows. If t is of the form $l[t_1, \dots, t_n]$ or $l\{t_1, \dots, t_n\}$ then $root(t) = l$; for t being a basic constant we assume that $root(t) = \$$.

2.1 Type Definitions

Here we introduce a formalism for specifying a class of decidable sets of data terms representing XML documents. It is a certain simplification of the formalism of [3]. First we specify a set of **type names** $\mathcal{T} = \mathcal{C} \cup \mathcal{S} \cup \mathcal{V}$ which consist of

- **type constants** from the alphabet $\mathcal{C}$
- **special type names** from the alphabet $\mathcal{S}$
- **type variables** from the alphabet $\mathcal{V}$

We associate each type name T with a set $\llbracket T \rrbracket$ (the *type denoted by T*) of data terms which are allowed values assigned to T . For T being a type constant or a special type name, the elements of $\llbracket T \rrbracket$ are basic constants.

Type constants correspond to an XML schema language base types. The set of type constants is fixed and finite. In our examples we will use a type constant $\#$ assuming that $\llbracket \# \rrbracket$ is the set of non empty strings of characters. This is similar to **#PCDATA** in DTD. In our notation, type constants and special type names are sequences of letters beginning with character $\#$.

Each type variable T is associated with a set of data terms $\llbracket T \rrbracket$ which is specified in a way similar to that of [3] and described below. Similarly, each special type name T is associated with a finite set $\llbracket T \rrbracket$ of basic constants.

First we introduce some auxiliary notions. The empty string will be denoted by ϵ . A *regular expression* over an alphabet Σ is ϵ , ϕ , any $a \in \Sigma$ and any $r_1 r_2$, $r_1 | r_2$ and r_1^* , where r_1, r_2 are regular expressions. A language $L(r)$ of strings over Σ is assigned to each regular expression r in the standard way (see e.g. [8]). In particular, $L(\phi) = \emptyset$, $L(\epsilon) = \{\epsilon\}$ and $L(r_1 | r_2) = L(r_1) \cup L(r_2)$.

Definition 2. A **regular type expression** is a regular expression over the alphabet of type names $\mathcal{T}$. We abbreviate a regular expression $r^n | r^{n+1} | \dots | r^m$, where $n \leq m$, as $r^{(n:m)}$, $r^n r^*$ as $r^{(n:\infty)}$, $r r^*$ as r^+ , and $r^{(0:1)}$ as $r^?$. A regular type expression of the form

$$r_1 \cdots r_k$$

where $k \geq 0$, each r_i is $T_i^{(n_{i,1}:n_{i,2})}$, $0 \leq n_{i,1} \leq n_{i,2} \leq \infty$ for $i = 1, \dots, k$, and $T_1, \dots, T_k$ are distinct type names, will be called a **multiplicity list**.

Multiplicity lists will be used to specify multisets of type names. We use $\text{types}_D(r)$ to denote the set of all type names occurring in the regular expression r .

Definition 3. A **type definition** is a set D of rules of the form

$$T \rightarrow l[r], \quad T \rightarrow l\{s\}, \quad \text{or} \quad T' \rightarrow c_1 | \dots | c_n,$$

where T is a type variable, T' a special type name, l a label, r a regular type expression, s a multiplicity list, and $c_1, \dots, c_n$ are basic constants. A rule $U \rightarrow G$ in D will be called a **rule for U** in D . We require that for any type name $U \in \mathcal{V} \cup \mathcal{S}$ occurring in D there is exactly one rule for U in D .

If the rule for a type variable T in D is as above then l will be called the **label** of T (in D) and denoted $\text{label}_D(T) = l$. For T being a type constant or a special type name we define $\text{label}_D(T) = \$$. The regular expression in a rule for type variable T is called the **content model** of T .

Example 4. Consider type definition D :

$$\begin{aligned}
Cd &\rightarrow cd[Title\ Artist^+ \#Category^?] \\
Title &\rightarrow title[\# \ Subtitle^?] \\
Subtitle &\rightarrow subtitle[\#] \\
Artist &\rightarrow artist[\#] \\
\#Category &\rightarrow pop \mid rock \mid classic
\end{aligned}$$

D contains a rule for each of type variables: Cd , $Title$, $Subtitle$, $Artist$ and a rule for special type name $\#Category$. Labels occurring in D are: cd , $title$, $subtitle$, $artist$, and pop , $rock$, $classic$ are basic constants.

Type definitions are a kind of grammars, they define sets by means of derivations, where a type variable T is replaced by the right hand side of the rule for T and a regular expression r is replaced by a string from $L(r)$; if T is a type constant or a special type name then it is replaced by a basic constant from respectively $\llbracket T \rrbracket$, or from the rule for T . This can be concisely formalized as follows (treating type definitions similarly to tree automata).

Definition 5. Let D be a type definition. We will say that a data term t is **derived** in D from a type name T , iff there exists a mapping ν from the subterms of t to type names such that $\nu(t) = T$ and for each subterm u of t

- if u is a basic constant then $\nu(u) \in \mathcal{C}$ and $u \in \llbracket \nu(u) \rrbracket$ or $\nu(u) \in \mathcal{S}$ and there exists a rule $\nu(u) \rightarrow \dots |u| \dots$ in D .
- otherwise $\nu(u) = U \in \mathcal{V}$ and
 - there is a rule $U \leftarrow l[r] \in D$, $u = l[t_1, \dots, t_n]$, and $\nu(t_1) \dots \nu(t_n) \in L(r)$,
 - or there is a rule $U \leftarrow l\{r\} \in D$, $u = l\{t_1, \dots, t_n\}$, and $\nu(t_1) \dots \nu(t_n)$ is a permutation of a string in $L(r)$.

The set of the data terms derived in D from a type name T will be denoted by $\llbracket T \rrbracket_D$.

Example 6. For the type definition D from the previous example, we have that the data term

$$t = cd[title["Stop"], artist["Sam Brown"], "pop"]$$

is derived from the type variable Cd . The type names assigned to the three arguments of cd are, respectively, $Title$, $Artist$, $\#Category$, and the type constant $\#$ is assigned to the constants "Stop", and "Sam Brown".

Notice that if T is a type constant then $\llbracket T \rrbracket_D = \llbracket T \rrbracket$. If it is clear from the context which type definition is considered, we will often omit the subscript in the notation $\llbracket \cdot \rrbracket_D$ and similar ones. For U being a set of type names $\{T_1, \dots, T_n\}$, we define a set of data terms $\llbracket U \rrbracket = \llbracket T_1 \rrbracket \cup \dots \cup \llbracket T_n \rrbracket$. For a regular type expression r we define $\llbracket r \rrbracket = \{d_1, \dots, d_n \mid d_1 \in \llbracket T_1 \rrbracket, \dots, d_n \in \llbracket T_n \rrbracket \text{ for some } T_1, \dots, T_n \in L(r)\}$. Notice that if $D \subseteq D'$ are type definitions then $\llbracket T \rrbracket_D = \llbracket T \rrbracket_{D'}$ for any type name T occurring in D .

3 Xcerpt– Introduction

Xcerpt is a rule-based query and transformation language for XML (see [10,5,11,4]). It employs patterns instead of paths to query XML and semistructured data. This approach stems from logic programming. A query term is matched against a data term from a database. A successful matching results in binding the variables in the query term to certain subterms of the data term. This operation is called simulation unification.

We consider here a somehow simplified version of Xcerpt. We focused on core Xcerpt features to make our type system simpler and easier to understand. The main difference is that our data terms represent trees while in full Xcerpt terms are used to represent graphs (by adding unique identifiers to some tree nodes and introducing nodes which are references to these identifiers). Other neglected Xcerpt features in respect to the Xcerpt version described in [11,10] are: functions and aggregations, non-pattern conditions, optional subterms, position specifications, negation, regular expressions and label variables. Moreover, we restrict ourselves to Xcerpt programs containing only one query rule.

We provide a formal semantics to the chosen fragment of Xcerpt. The semantics of query terms is from [12], the rest of the semantics is based on [10].

We assume that a database is a data term or a multiset of data terms. There are two other kinds of terms in Xcerpt: query terms and construct terms. The role of query terms is to be matched against a database. Construct terms are used in constructing data terms which are query results.

Definition 7. *Query terms are inductively defined as follows:*

- Any basic constant is a query term.
- A variable X is a query term.
- If q is a query term, then $\text{desc } q$ is a query term.
- If X is a variable and q is a query term, then $X \rightsquigarrow q$ is a query term.
- If l is a label and $q_1, \dots, q_n$ ($n \geq 0$) are query terms, then $l[q_1, \dots, q_n]$, $l\{q_1, \dots, q_n\}$, $l[[q_1, \dots, q_n]]$ and $l\{\{q_1, \dots, q_n\}\}$ are query terms (called rooted query terms).

For a rooted query term $q = l\alpha q_1, \dots, q_n\beta$, where $\alpha\beta$ are parentheses $[], [[]], \{\}$ or $\{\{\}\}$, $\text{root}(q) = l$ and $q_1, \dots, q_n$ are the child subterms of q . If q is a basic constant then $\text{root}(q) = \$$.

To informally explain the role of query terms, consider a query term $q = l\alpha q_1, \dots, q_m\beta$ and a data term $d = l'\alpha'd_1, \dots, d_n\beta'$, where $\alpha, \beta, \alpha', \beta'$ are parentheses. In order to q match d it is necessary that $l = l'$. Moreover the child subterms $q_1, \dots, q_m$ of q should match certain child subterms of d . Single parentheses in d ($[]$ or $\{\}$) mean that $m = n$ and each q_i should match some (distinct) d_j . Double parentheses mean that $m \leq n$ and $q_1, \dots, q_m$ are matched against some m terms out of $d_1, \dots, d_n$. Curly braces ($\{\}$ or $\{\{\}\}$) in q mean that the order of the child subterms in d does not matter; square brackets in q mean that $q_1, \dots, q_m$ should match (a subsequence of) $d_1, \dots, d_n$ in the same order.

A variable matches any data term, $\text{desc } q$ matches a data term d whenever q matches some subterm of d . A query term $X \rightsquigarrow q$ matches any data term matched by q . A side effect of a query term X or $X \rightsquigarrow q$ matching a data term d is that variable X obtains a value d .

Now we formally define which query terms match which data terms and what are the resulting assignments of data terms to variables. We do not follow the original definition of simulation unification. Instead we define a notion of answer substitution for a query term q and a data term d . As usually, by a *substitution* (of data terms for variables) we mean a set $\theta = \{X_1/d_1, \dots, X_n/d_n\}$, where $X_1, \dots, X_n$ are distinct variables and $d_1, \dots, d_n$ are data terms; its domain $\text{dom}(\theta)$ is $\{X_1, \dots, X_n\}$, its application to a (query) term is defined in a standard way.

Definition 8 ([12]). A substitution θ is an answer substitution (shortly, an **answer**) for a query term q and a data term d if q and d are of one of the forms below and the corresponding condition holds. (In what follows $m, n \geq 0$, X is a variable, l is a label, $q, q_1, \dots$ are query terms, and $d, d_1, \dots$ data terms; set notation is used for multisets, for instance $\{d, d\}$ and $\{d\}$ are different multisets).

q	d	condition on q and d
b	b	b is a basic constant
$l[q_1, \dots, q_n]$	$l[d_1, \dots, d_n]$	θ is an answer for q_i and d_i , for each $i = 1, \dots, n$
$l[[q_1, \dots, q_m]]$	$l[d_1, \dots, d_n]$	for some subsequence $d_{i_1}, \dots, d_{i_m}$ of $d_1, \dots, d_n$ (i.e. $0 < i_1 < \dots < i_m \leq n$) θ is an answer for q_j and d_{i_j} , for each $j = 1, \dots, m$,
$l\{q_1, \dots, q_n\}$	$l\{d_1, \dots, d_n\}$ or $l[d_1 \cdots d_n]$	for some permutation $d_{i_1}, \dots, d_{i_n}$ of $d_1, \dots, d_n$ (i.e. $\{d_{i_1}, \dots, d_{i_n}\} = \{d_1, \dots, d_n\}$) θ is an answer for q_j and d_{i_j} for each $j = 1, \dots, m$,
$l\{\{q_1, \dots, q_m\}\}$	$l\{d_1, \dots, d_n\}$ or $l[d_1, \dots, d_n]$	for some $\{d_{i_1}, \dots, d_{i_m}\} \subseteq \{d_1, \dots, d_n\}$ θ is an answer for q_j and d_{i_j} for each $j = 1, \dots, m$,
X	d	$X\theta = d$
$X \rightsquigarrow q$	d	$X\theta = d$ and θ is an answer for q and d
$\text{desc } q$	d	θ is an answer for q and some subterm d' of d

We say that q matches d if there exists an answer for q, d .

Thus if q is a rooted query term (or a basic constant) and $\text{root}(q) \neq \text{root}(d)$ then no answer for q, d exists. If $q = d$ then any θ is an answer for q, d . A query $l\{\{\}\}$ matches any data term with the label l . If θ, θ' are substitutions and $\theta \subseteq \theta'$

then if θ is an answer for q, d then θ' is an answer for q, d . If a variable X occurs in a query term q then queries $X \rightsquigarrow q$ and $X \rightsquigarrow \text{desc } q$ match no data term, provided that $q \neq X$ and q is not of the form $\text{desc } \dots \text{desc } X$.

Example 9. Query term $q_1 = a[c\{\{d[], "e"\}\}, f[g[], h\{"i"\}]]$ matches data terms $a[c\{"e", d[], g[]\}, f[g[], l[], h\{"i"\}]]$ and $a[c[d[], g[], "e"], f[g[], h\{"i"\}]]$. In contrast, data terms $f[h\{"i"\}, g[]]$ and $f\{g[], h\{"i"\}\}$ are not matched by $f[g[], h\{"i"\}]]$. Query term $q_2 = \text{desc } w\{\{\}\}$ matches data terms $a[b\{w[]\}]$ and $w\{"s"\}$. Query term $q_2 = a[[X_1 \rightsquigarrow c[d\{\{\}\}], X_2, "p"]]$ matches $a["s", c[d\{\{\}\}, "r"], h\{j[]\}, "p"]$, with an answer which binds X_1 to $c[d\{\{\}\}, "r"]$ and X_2 to $h\{j[]\}$.

Each answer for a query term q binds all the variables of the query to some data terms. For any such answer θ' (for q and d) there exists an answer $\theta \subseteq \theta'$ (for q and d) binding exactly these variables. We will call such answers *non redundant*. From Definition 8 one can derive an algorithm which produces non redundant answers for a given q and d . Construction of the algorithm is rather simple, we skip the details. Non redundant answers are actually those of interest; we consider a more general class of answers to simplify Definition 8.

A **targeted query term** is a pair $\text{in}(db, q)$, of a URI and a query term. We assume that the URI locates on the Web a data term $d(db)$. An answer substitution for q and $d(db)$ is called an answer substitution for $\text{in}(db, q)$ (and an arbitrary data term).

Definition 10. A query is inductively defined as follows.

- Any query term and any targeted query term is a query.
- If $Q_1, \dots, Q_n$ ($n \geq 0$) are queries then $\text{and}(Q_1, \dots, Q_n)$ and $\text{or}(Q_1, \dots, Q_n)$ are queries.

A substitution θ is an answer substitution for $\text{and}(Q_1, \dots, Q_n)$ (respectively for $\text{or}(Q_1, \dots, Q_n)$) and a data term d iff θ is an answer substitution for each of (some of) $Q_1, \dots, Q_n$ and d .

A query can be transformed into equivalent one in *disjunctive normal form* $\text{or}(Q_1, \dots, Q_n)$, where each Q_i is of the form $\text{and}(Q_{i1}, \dots, Q_{ik_i})$ and each Q_{ij} is a (targeted) query term (cf. [10, Proposition 6.4]).

Definition 11. A **construct term** and the set $FV(c)$ of free variables of a construct term c are defined recursively. If b is a basic constant, X a variable, l a label, $c, c_1, \dots, c_n$ construct terms ($n \geq 0$), and k a natural number then

$$b, \quad X, \quad l[c_1, \dots, c_n], \quad l\{c_1, \dots, c_n\}, \quad \text{all } c, \quad \text{some } k \ c,$$

are construct terms. $FV(b) = \emptyset$, $FV(X) = \{X\}$, $FV(l[c_1, \dots, c_n]) = FV(l\{c_1, \dots, c_n\}) = \bigcup_{i=1}^n FV(c_i)$, $FV(\text{all } c) = FV(\text{some } k \ c) = \emptyset$.

Notice that any data term is a construct term. (Also, a construct term without any **all** and **some** construct is a query term).

Before we define the notion of a query rule and its result we need to provide some auxiliary definitions. By a substitution set we mean a set of substitutions of data terms for variables, e.g. of answers for a query and a data term.

Definition 12. Given a substitution set Θ and a set V of variables, such that $V \subseteq \text{dom}(\theta)$ for each $\theta \in \Theta$, the equivalence relation $\simeq_V \subseteq \Theta \times \Theta$ is defined as: $\theta_1 \simeq \theta_2$ iff $\theta_1(X) = \theta_2(X)$ for all $X \in V$. The set of equivalence classes of $\simeq_V$ is denoted by $\Theta/\simeq_V$.

The concatenation of two sequences S_1, S_2 of data terms will be denoted by $S_1 \circ S_2$. We do not distinguish between a data term d and the one element sequence with the element d .

Definition 13. Let c be a construct term and Θ be a substitution set containing the same assignments for the free variables $FV(c)$ of c (i.e. $\theta_1 \simeq_{FV(c)} \theta_2$ for any $\theta_1, \theta_2 \in \Theta$). The application $\Theta(c)$ of the substitution set Θ to c is a sequence of data terms defined as follows

- $\Theta(b) = b$, where b is a basic constant
- $\Theta(X) = X\theta$, where $\theta \in \Theta$
- $\Theta(l\{c_1, \dots, c_n\}) = l\{\Theta(c_1) \circ \dots \circ \Theta(c_n)\}$
- $\Theta(l[c_1, \dots, c_n]) = l[\Theta(c_1) \circ \dots \circ \Theta(c_n)]$
- $\Theta(\text{all } c') = \Theta_1(c') \circ \dots \circ \Theta_k(c')$, where $\{\Theta_1, \dots, \Theta_k\} = \Theta/\simeq_{FV(c')}$
- $\Theta(\text{some } k \ c') = \Theta_1(c') \circ \dots \circ \Theta_m(c')$, where $\{\Theta_1, \dots, \Theta_m\} \subseteq \Theta/\simeq_{FV(c')}$ and $m = k$ if $|\Theta/\simeq_{FV(c')}| \geq k$ or $m = |\Theta/\simeq_{FV(c')}|$ otherwise.

For a construct term c containing neither **all** nor **some**, $\Theta(c) = c\theta$ for any $\theta \in \Theta$. Notice that $\Theta(c)$ is defined uniquely unless c contains **all** or **some** (and $\Theta(c)$ is defined uniquely up to reordering provided c does not contain **some**). Notice also that $\Theta(c)$ is a one element sequence unless c is of the form **all** c' or **some** $k \ c'$.

Definition 14. A **construct-query rule** (shortly, query rule) is an expression of the form $c \leftarrow Q$, where c is a construct term not of the form **all** c' or **some** $k \ c'$, Q is a query and every variable occurring in c also occurs in Q . Moreover, if $\text{or}(Q_1, \dots, Q_n)$ is a disjunctive normal form of Q then every variable of c occurs in each Q_i , for $i = 1, \dots, n$. The construct term c will be sometimes called the head and Q the body of the rule.

If Θ is the set of all answers for Q and a data term d , and $\Theta' \in \Theta/\simeq_{FV(c)}$ then $\Theta'(c)$ is a **result** for query $c \leftarrow Q$ and d .

Each result of a query rule is a data term, as an answer for a query term binds all the variables of the rule to data terms.

Example 15. Consider a database which is a data term:

```
catalogue[ cd[title["Empire Burlesque"], artist["Bob Dylan"], year["1985"]],
           cd[title["Hide your heart"], artist["Bonnie Tyler"], year["1988"]],
           cd[title["Stop"], artist["Sam Brown"], year["1988"]] ]
```

Here is a rule which extracts titles and artists for the CD's issued in 1988 and presents the results in a changed form (title as name and artist as author). *TITLE* and *ARTIST* are variables.

```
result[ name[TITLE], author[ARTIST] ] ←
      catalogue[ { cd[title[TITLE], artist[ARTIST], year["1988"]} ] }
```

The results returned by the rule are:

```
result[name["Hide your heart"], author["Bonnie Tyler"]]
result[name["Stop"], author["Sam Brown"]]
```

The next query rule is similar. It uses **all** for grouping all the results together and another **all** for grouping together the CD's from the same year.

```
results[all result[ cds[all name[TITLE]], year[YEAR]]] ←
  catalogue{{ cd{{ title[TITLE], year[YEAR] }}}}
```

The rule returns the following result:

```
results[ result[ year["1988"], cds[ title["Hide your heart"], title["Stop"] ]],
  result[ year["1985"], cds[ title["Empire Burlesque"] ] ] ]
```

4 Reasoning about Types of Xcerpt Query Results

4.1 Motivation

In this section we study the relation between types of databases and types of query results. Assume that the only information available about the database is that it is a data term (or a set of data terms) from a given type $\llbracket T_{DB} \rrbracket$ (or from a given union of types $\llbracket T_1 \rrbracket \cup \dots \cup \llbracket T_n \rrbracket$). One may want to know what query results are possible for such database. We show how to compute (a superset of) the set of such results. The set will be expressed as a type, specified by a type definition. We will usually call it the query result type.

Computing the query result type may serve some additional purposes. 1. If this type is empty, then the query will never give an answer for a data term from $\llbracket T_{DB} \rrbracket$. An algorithm checking this property is obtained by combining computing query result type with checking emptiness of a type. 2. If some specification of the intended type of results exists, one may check if the query is correct w.r.t. the specification, by checking whether the computed type of the results is included in the specified one. 3. If we use a data term d as the body of the query, then computing the result type is also a check whether $d \in \llbracket T_{DB} \rrbracket$. Namely $d \in \llbracket T_{DB} \rrbracket$ iff the result type is not empty. 4. The algorithm computing the query result type produces as a side effect the types of the variables of the queries. For each variable from the query it gives a set containing every value that can be assigned to the variable (when querying a data term from type $\llbracket T_{DB} \rrbracket$). This provides additional information about the behaviour of the query. We may consider specifications of the types of the query variables. A query is correct w.r.t. such a specification if for every variable the computed type is a subset of the specified type.

Example 16. Consider the type definition D from Example 4 and a construct-query rule Q :

```
result[name[TITLE], author[ARTIST]] ←
  cd{{ TITLE, ARTIST ~ artist{{}}, "rock" }}
```

The intention of the rule is to collect titles and authors of all the CD's of the rock category. When the query term of the rule is matched against a database of type *Cd*, the variables *TITLE*, *ARTIST* are bound to data terms of types, respectively, *Title*, *Artist* or *Artist*, *Artist*. As the variable *TITLE* is intended to take values only of type *Title*, the query is incorrect w.r.t. our expectations. The type *Result* of the query result can be described by the following type definition $D' = D \cup \{ \text{Result} \rightarrow \text{result}[\text{Name Author}], \text{Name} \rightarrow \text{name}[\text{Title|Artist}], \text{Author} \rightarrow \text{author}[\text{Artist}] \}$.

4.2 Variable-type mappings

In this section we assume a fixed type definition D (describing the type of the database).

To represent a set of answers (for a query term and a set of data terms) we will use a mapping $\Gamma: V \rightarrow \mathcal{E}$ (called a *variable-type mapping*), where V is the set of variables occurring in the considered query rule and $\mathcal{E}$ is a set of expressions. $\mathcal{E}$ contains 0, 1, the type names from D , and expressions of the form $T_1 \cap T_2$, where $T_1, T_2 \in \mathcal{E}$. Each expression E from $\mathcal{E}$ denotes a set $\llbracket E \rrbracket$ of data terms. $\llbracket 1 \rrbracket$ denotes the set of all data terms, $\llbracket 0 \rrbracket = \emptyset$, $\llbracket T \rrbracket = \llbracket T \rrbracket_D$ for any type name T , and $\llbracket T_1 \cap T_2 \rrbracket = \llbracket T_1 \rrbracket \cap \llbracket T_2 \rrbracket$. The set of substitutions corresponding to a mapping $\Gamma: V \rightarrow \mathcal{E}$ is

$$\text{substitutions}_D(\Gamma) = \{ \theta \mid \forall_{X \in V} \theta X \in \llbracket \Gamma(X) \rrbracket \}.$$

(According to our convention, we will often skip the index $_D$.) Notice that if $\theta \in \text{substitutions}(\Gamma)$ then $V \subseteq \text{dom}(\theta)$ and if $\theta \subseteq \theta'$ then $\theta' \in \text{substitutions}(\Gamma)$. For a set Ψ of variable-type mappings we define $\text{substitutions}(\Psi) = \bigcup_{\Gamma \in \Psi} \text{substitutions}(\Gamma)$.

For $Y_1, \dots, Y_k \in V$, $T_1, \dots, T_k \in \mathcal{E}$, mapping $[Y_1 \mapsto T_1, \dots, Y_k \mapsto T_k]: V \rightarrow \mathcal{E}$ is defined as

$$[Y_1 \mapsto T_1, \dots, Y_k \mapsto T_k](X) = \begin{cases} T_i & \text{if } X = Y_i \\ 1 & \text{otherwise.} \end{cases}$$

We will not distinguish between expressions $T \cap 1$ and T , and between $T \cap 0$ and 0 (where $T \in \mathcal{E}$).

Inclusion of types induces a partial order $\sqsubseteq$ on the mappings from $V \rightarrow \mathcal{E}$, as follows. If Γ and Γ' are such mappings then $\Gamma \sqsubseteq \Gamma'$ iff $\llbracket \Gamma(X) \rrbracket \subseteq \llbracket \Gamma'(X) \rrbracket$ for each variable $X \in V$. Notice that $\Gamma \sqsubseteq \Gamma'$ is equivalent to $\text{substitutions}(\Gamma) \subseteq \text{substitutions}(\Gamma')$.

4.3 Typing Rules for Xcerpt

The rules presented in this section provide a descriptive type system for Xcerpt: the typing of a program is an approximation of its semantics. An algorithm computing a type of results for a given Xcerpt query rule can be easily derived from the presented rules as they can be seen as an abstract version of the algorithm. Below we present the rules for query terms, queries, construct terms and query rules. In the Appendix we prove correctness of the typing system.

Query terms. The rules in this subsection provide a way to derive facts of the form $D \vdash q : T \triangleright \Gamma$, where D is a type definition, q a query term, T a type name, and Γ a variable-type mapping. The intention is that if q is applied to a data term $d \in \llbracket T \rrbracket$ then the resulting substitution is in $\text{substitutions}(\Gamma)$ for some Γ such that $D \vdash q : T \triangleright \Gamma$ can be derived.

$$\frac{b \in \llbracket T \rrbracket}{D \vdash b : T \triangleright \Gamma} \quad (\text{CONST})$$

where b is a basic constant.

$$\frac{\Gamma \sqsubseteq [X \mapsto T]}{D \vdash X : T \triangleright \Gamma} \quad (\text{VAR})$$

$$\frac{D \vdash q : T \triangleright \Gamma \quad \Gamma \sqsubseteq [X \mapsto T]}{D \vdash X \rightsquigarrow q : T \triangleright \Gamma} \quad (\text{AS})$$

$$\frac{D \vdash q : T \triangleright \Gamma}{D \vdash \text{desc } q : T \triangleright \Gamma} \quad (\text{DESCENDANT})$$

$$\frac{D \vdash \text{desc } q : T' \triangleright \Gamma}{D \vdash \text{desc } q : T \triangleright \Gamma} \quad (\text{DESCENDANT REC})$$

where $T' \in \text{types}(r)$ and r is the content model of T .

$$\frac{D \vdash q_1 : T_1 \triangleright \Gamma \quad \dots \quad D \vdash q_n : T_n \triangleright \Gamma}{D \vdash l \ \alpha q_1, \dots, q_n \beta : T \triangleright \Gamma} \quad (\text{PATTERN})$$

where the rule for T in D is of the form $T \rightarrow l[r]$
or it is of the form $T \rightarrow l\{r\}$ and $(\alpha\beta = \{\}$ or $\alpha\beta = \{\{\}\})$,
 s is r with every type name U replaced by $U|e$,
 $T_1 \dots T_n \in L(r)$ if $\alpha\beta = []$,
 $T_1 \dots T_n \in L(s)$ if $\alpha\beta = [[]]$,
 $T_1 \dots T_n \in \text{perm}(L(r))$ if $\alpha\beta = \{\}$,
 $T_1 \dots T_n \in \text{perm}(L(s))$ if $\alpha\beta = \{\{\}\}$.

Here $\text{perm}(L)$ stands for the language of permutations of the strings from a language L .

Queries. From the rules below one can derive facts of the form $D \vdash Q : U \triangleright \Gamma$, where Q is a query, U a finite set of type names and Γ a variable-type mapping. If θ is an answer substitution for Q and a data term from $\llbracket U \rrbracket$ then $\theta \in \text{substitutions}(\Gamma)$ for some Γ such that $D \vdash q : T \triangleright \Gamma$ can be derived.

In general a query may be applied to data terms produced by query rules of an Xcerpt program. As their results may be of different types, we consider here a set of types U instead of a single type T .

$$\frac{D \vdash q : T \triangleright \Gamma \quad T \in U}{D \vdash q : U \triangleright \Gamma} \quad (\text{QUERY TERM})$$

$$\frac{D \vdash q : T \triangleright \Gamma}{D \vdash \text{in}(db, q) : U \triangleright \Gamma} \quad (\text{TARGETED QUERY TERM})$$

where $d(db)$ is of type T (formally $d(db) \in \llbracket T \rrbracket$).

$$\frac{D \vdash Q_1 : U \triangleright \Gamma \quad \dots \quad D \vdash Q_n : U \triangleright \Gamma}{D \vdash \text{and}(Q_1, \dots, Q_n) : U \triangleright \Gamma} \quad (\text{AND QUERY})$$

$$\frac{D \vdash Q : U \triangleright \Gamma}{D \vdash \text{or}(\dots, Q, \dots) : U \triangleright \Gamma} \quad (\text{OR QUERY})$$

Construct terms. To formulate typing rules for construct terms we need an equivalence relation on mappings:

Definition 17. *Given a type definition D , a set of variable-type mappings Ψ and a set V of variables, such that $V \subseteq \text{dom}(\Gamma)$ for each $\Gamma \in \Psi$, the relation $\sim_V \subseteq \Psi \times \Psi$ is defined as: $\Gamma_1 \sim_V \Gamma_2$ iff $\llbracket \Gamma_1(X) \rrbracket \cap \llbracket \Gamma_2(X) \rrbracket \neq \emptyset$ for all $X \in V$. The set of equivalence classes of the transitive closure $\sim_V^*$ of $\sim_V$ is denoted by $\Psi / \sim_V^*$.*

The following rules allow to derive facts of the form $D \vdash c : \Psi \triangleright S$, where c is a construct term, Ψ is a set of variable-type mappings (for which the types are defined by D) and S is a regular type expression. The intention is that if applying a substitution set Θ to c results in a data term sequence $\Theta(c) = d_1, \dots, d_n$ and $\text{substitutions}(\Theta) \subseteq \text{substitutions}(\Psi)$ then $D \vdash c : \Psi \triangleright S$ can be derived such that each $d_i \in \llbracket T_i \rrbracket$ and $T_1 \dots T_n \in L(S)$. To derive $D \vdash c : \Psi \triangleright S$ it is necessary that $\Gamma(X) \neq 1$ for any $\Gamma \in \Psi$ and any variable X occurring in c . For correctness of the rules it is required that for any $\Gamma_1, \Gamma_2 \in \Psi$, $\Gamma_1 \sim_{FV(c)}^* \Gamma_2$.

$$\frac{(T_c \rightarrow c) \in D}{D \vdash c : \Psi \triangleright T_c} \quad (\text{CONST})$$

where c is a basic constant.

$$\frac{\llbracket T_1 \rrbracket = \llbracket \Gamma_1(X) \rrbracket \quad \dots \quad \llbracket T_n \rrbracket = \llbracket \Gamma_n(X) \rrbracket}{D \vdash X : \{\Gamma_1, \dots, \Gamma_n\} \triangleright T_1 \mid \dots \mid T_n} \quad (\text{VAR})$$

$$\frac{D \vdash c_1 : \Psi \triangleright S_1 \quad \cdots \quad D \vdash c_n : \Psi \triangleright S_n \quad (T_c \rightarrow l\alpha S_1 \cdots S_n \beta) \in D}{D \vdash l\alpha c_1, \dots, c_n \beta : \Psi \triangleright T_c} \quad (\text{PATTERN})$$

$$\frac{D \vdash c : \Psi_1 \triangleright S_1 \quad \cdots \quad D \vdash c : \Psi_n \triangleright S_n \quad \{\Psi_1, \dots, \Psi_n\} = \Psi / \sim_{FV(c)}^*}{D \vdash \mathbf{all} \ c : \Psi \triangleright (S_1 \mid \cdots \mid S_n)^+} \quad (\text{ALL})$$

$$\frac{D \vdash c : \Psi_1 \triangleright S_1 \quad \cdots \quad D \vdash c : \Psi_n \triangleright S_n \quad \{\Psi_1, \dots, \Psi_n\} = \Psi / \sim_{FV(c)}^*}{D \vdash \mathbf{some} \ k \ c : \Psi \triangleright (S_1 \mid \cdots \mid S_n)^{(1:k)}} \quad (\text{SOME})$$

Xcerpt query rules. For a given type definition D , query Q and a set U of types names, the rules introduced above nondeterministically generate variable type mappings. Now we describe which sets of generated mappings are sufficient for the purpose of approximating the semantics of query-rules.

Definition 18. Let D be a type definition. Let Q be a query term and W a type name, or Q a query and W a set of type names. A set $\{\Gamma_1, \dots, \Gamma_n\}$ of variable-type mappings is **complete** for Q and W wrt. D if

- $D \vdash Q : W \triangleright \Gamma_i$ for $i = 1, \dots, n$, and
- whenever $D \vdash Q : W \triangleright \Gamma$, there exists $i \in \{1, \dots, n\}$ such that $\Gamma \sqsubseteq \Gamma_i$.

From the following rule one can derive facts of the form $D \vdash (c \leftarrow Q) : U \triangleright S_1 \mid \cdots \mid S_n$ where $c \leftarrow Q$ is a query rule, U is a finite set of type names and S_i are regular type expressions. The intention is that if we apply a query rule $c \leftarrow Q$ to a database of a type $\llbracket U \rrbracket$ then we obtain results belonging to the set $\llbracket S_1 \mid \cdots \mid S_n \rrbracket$.

$$\frac{D \vdash c : \Psi_1 \triangleright S_1 \quad \cdots \quad D \vdash c : \Psi_n \triangleright S_n \quad \{\Psi_1, \dots, \Psi_n\} = \Psi / \sim_{FV(c)}^*}{D \vdash (c \leftarrow Q) : U \triangleright S_1 \mid \cdots \mid S_n} \quad (\text{QUERY RULE})$$

where Ψ is complete for Q and U wrt. D .

Example 19. Consider type definition $D = \{T \rightarrow l[A^*BC], A \rightarrow a, B \rightarrow b, C \rightarrow c, R_1 \rightarrow a[A^+A], R_2 \rightarrow a[A^+B], R_3 \rightarrow a[(A \mid B)^+C]\}$ and the query rule

$$a[\mathbf{all} \ X, Y] \leftarrow l[[X, Y]]$$

abbreviated as $c_0 \leftarrow q$. We apply the query rule to a set of types $U = \{T, A, B, C\}$. First we need to find a complete set of mappings Ψ_0 for q and U . If we apply the query term q to the type T using the rules for query terms we can derive facts $D \vdash q : T \triangleright \Gamma_i$ for $i = 1, \dots, 4$, where $\Gamma_1 = [X \mapsto A, Y \mapsto A]$, $\Gamma_2 = [X \mapsto A, Y \mapsto B]$, $\Gamma_3 = [X \mapsto A, Y \mapsto C]$ and $\Gamma_4 = [X \mapsto B, Y \mapsto C]$. If we apply the query term q to the type A, B or C we cannot derive anything using the rules. Hence, the rules for queries allow us to derive $D \vdash q : U \triangleright \Gamma_i$ for $i = 1, \dots, 4$. The set $\Psi_0 = \{\Gamma_1, \Gamma_2, \Gamma_3, \Gamma_4\}$ is complete for q and U . Since $FV(c_0) = \{Y\}$, $\Psi_0 / \sim_{FV(c_0)}^* =$

$\{\Psi_1, \Psi_2, \Psi_3\}$, where $\Psi_1 = \{\Gamma_1\}$, $\Psi_2 = \{\Gamma_2\}$, $\Psi_3 = \{\Gamma_3, \Gamma_4\}$. Now we apply each of Ψ_i to the construct term c_0 . Using the rules for construct terms we can derive the following facts: $D \vdash c_0 : \Psi_1 \triangleright R_1$, $D \vdash c_0 : \Psi_2 \triangleright R_2$ and $D \vdash c_0 : \Psi_3 \triangleright R_3$. Using the rule (QUERY RULE) we can derive $D \vdash c_0 \leftarrow q : U \triangleright R_1 \mid R_2 \mid R_3$. It means that if the rule $c_0 \leftarrow q$ is applied to a data term from $\llbracket U \rrbracket$ all the obtained results are in the set $\llbracket R_1 \mid R_2 \mid R_3 \rrbracket$.

The following theorem expresses the correctness of the typing rules wrt. the semantics given in section 3. More precisely, it expresses the existence of a typing derivation for a rule whenever it has a result for some data term d in the type denoted by a set U of type names. It also expresses that any derivation of a query rule $(c \leftarrow Q)$ wrt. a set of type names U is a correct approximation of the set of results for $(c \leftarrow Q)$ and any data term in the type denoted by U .

Theorem 20. *Let D be a type definition and $(c \leftarrow Q)$ be a query rule, where for each targeted query term $\text{in}(db, q)$ in Q there is a type name T in D such that $d(db) \in \llbracket T \rrbracket$. Let U be a set of type names and d a data term such that $d \in \llbracket U \rrbracket$.*

If a result for $(c \leftarrow Q)$ and d exists then there exist S and D' such that $D' \supseteq D$ and $D' \vdash (c \leftarrow Q) : U \triangleright S$.

If there exists S such that $D \vdash (c \leftarrow Q) : U \triangleright S$ and if d' is a result for $(c \leftarrow Q)$ and d , then $d' \in \llbracket S \rrbracket$.

Proof. See the Appendix.

5 Conclusion

This paper presents a descriptive type system for a substantial fragment of the Web and Semantic Web query language Xcerpt. The type system provides approximation of the semantics of Xcerpt programs. For a given Xcerpt query rule it provides a type of its results (i.e. a superset of the set of the results) under the assumption that the query rule is applied to data of a given type. The main contribution of the paper is a formalization of the type system of [12] by means of typing rules, and a correctness proof of the type system. The employed formal semantics of the Xcerpt fragment may be of separate interest.

A topic for the future work is formalization by means of typing rules of the more precise typing algorithm presented in [7]. The current work should also be generalized to the omitted features of Xcerpt. Dealing with some of them, e.g. negation or terms representing graphs, seems to be difficult and needs further investigation. Moreover the type system should be extended to Xcerpt programs containing more than one query rule.

Ongoing research on Xcerpt is about integration of standard Web and Semantic Web, especially querying of RDF. As RDF data represents graph shaped structures, dealing with graph structured data is the central aspect for typed RDF querying. Additionally, investigation of existing Semantic Web type formalisms found in RDFS and OWL and integration of some of their features (eg. transitive properties) in the Xcerpt type systems is a subject of further work.

Another interesting topic for the future work is a comparison between the descriptive typing approach (to which our work belongs) and the prescriptive approach [6,7]. As our type system is presented by means of typing rules similar to the typing rules of prescriptive type systems, the differences and similarities between two approaches can be better understood.

The work on a prototype implementation of the type system for Xcerpt is in progress. The algorithm corresponding to the presented typing rules has been implemented as an additional module in Xcerpt prototype. We plan to implement the more precise version of the algorithm and also to extend the prototype to be able to handle all constructs used in Xcerpt.

Acknowledgement. This research has been partially 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>).

References

1. Serge Abiteboul, Peter Buneman, and Dan Suciu. *Data on the Web: from relations to semistructured data and XML*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2000.
2. A. Brüggemann-Klein, M. Murata, and D. Wood. Regular tree and regular hedge languages over unranked alphabets. Technical Report HKUST-TCSC-2001-0, The Hongkong University of Science and Technology, April 2001.
3. François Bry, Włodzimierz Drabent, and Jan Maluszynski. On subtyping of tree-structured data: A polynomial approach. In *International Workshop, PPSWR 2004, St. Malo, France, September, 2004, Proceedings*, number 3208 in LNCS, pages 1–18, 2004.
4. François Bry, Tim Furche, Liviu Badea, Christoph Koch, Sebastian Schaffert, and Sacha Berger. Identification of Design Principles. Deliverable I4-D2, REWERSE, 2004. At <http://rewerse.net/publications.html/#REWERSE-DEL-2004-I4-D2>.
5. François Bry and Sebastian Schaffert. An Entailment Relation for Reasoning on the Web. In *Proceedings of Rules and Rule Markup Languages for the Semantic Web, Sanibel Island (Florida), USA (20th October 2003)*, LNCS, 2003.
6. Luca Cardelli. Type Systems. In Allen B. Tucker, editor, *The Handbook of Computer Science and Engineering, Second Edition*, chapter 97-1. CRC Press, 2004.
7. Horatiu Cirstea, Emmanuel Coquery, Włodzimierz Drabent, François Fages, Claude Kirchner, Luigi Liquori, Benjamin Wack, and Artur Wilk. Types for REWERSE reasoning and query languages. Deliverable I3-D4, REWERSE, 2005. Available at <http://rewerse.net/publications.html/#REWERSE-DEL-2005-I3-D4>.
8. J. E. Hopcroft and J. D. Ullmann. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.
9. M. Murata, D. Lee, M. Mani, and K. Kawaguchi. Taxonomy of XML schema languages using formal language theory. Submitted, 2003.
10. Sebastian Schaffert. *Xcerpt: A Rule-Based Query and Transformation Language for the Web*. PhD thesis, University of Munich, 2004.
11. Sebastian Schaffert and François Bry. Querying the Web Reconsidered: A Practical Introduction to Xcerpt. In *Proceedings of Extreme Markup Languages 2004, Montreal, Quebec, Canada (2nd–6th August 2004)*, 2004.

12. A. Wilk and W. Drabent. On types for XML query language Xcerpt. In *International Workshop, PPSWR 2003, Mumbai, India, December 8, 2003, Proceedings*, number 2901 in LNCS, pages 128–145. Springer Verlag, 2003.

A Soundness Proof of the Descriptive Xcerpt Typing Rules

Definition 21. Given a type definition D and a substitution θ , the mapping Γ_θ is defined as:

$$\Gamma_\theta(X) = \begin{cases} T_1 \cap \dots \cap T_n & \text{if } X \in \text{dom}(\theta) \text{ and } \{T_1, \dots, T_n\} = \{T \mid X\theta \in \llbracket T \rrbracket_D\} \\ 1 & \text{otherwise.} \end{cases}$$

One can remark that by definition $\theta \in \text{substitutions}(\Gamma_\theta)$.

Lemma 22. Let D be a type definition, q a query term, d a data term, T a type name and θ an answer for q and d . If $d \in \llbracket T \rrbracket_D$ then $D \vdash q : T \triangleright \Gamma_\theta$.

Proof. By induction on the query term q .

- If q is a basic constant b , then $d = b$. Since $d \in \llbracket T \rrbracket$, by rule (CONST) we obtain $D \vdash q : T \triangleright \Gamma_\theta$.
- If q is of the form $l\alpha q_1, \dots, q_n\beta$ then d is of the form $l\alpha'd_1, \dots, d_m\beta'$. By Definition 5, the rule for T in D is of the form $T \rightarrow l\alpha'r\beta'$. Let $\{S_1, \dots, S_p\} = \text{types}(r)$ and s be the regular expression r with every type name U replaced by $U \mid \epsilon$.
 - If $\alpha\beta = []$ then $\alpha'\beta' = []$ and $m = n$. Moreover, for each $i \in \{1, \dots, n\}$, θ is an answer for q_i and d_i . Since $d \in \llbracket T \rrbracket$ and by Definition 5, there exists $T_1, \dots, T_n$ such that $d_i \in \llbracket T_i \rrbracket$ and $T_1 \cdots T_n \in L(r)$. By induction hypothesis, $D \vdash q_i : T_i \triangleright \Gamma_\theta$. Thus, by rule (PATTERN), we deduce $D \vdash q : T \triangleright \Gamma_\theta$.
 - If $\alpha\beta = [[]]$ then $\alpha'\beta' = []$ and $m \geq n$. Since θ is an answer for q and d , there exists $0 < i_1 < \dots < i_n \leq m$ such that for each $j \in \{1, \dots, n\}$, θ is an answer for q_j and d_{i_j} . Moreover, there exists $T_1, \dots, T_m$ such that $d_i \in \llbracket T_i \rrbracket$ and $T_1 \cdots T_m \in L(r)$. By induction hypothesis, $D \vdash q_j : T_{i_j} \triangleright \Gamma_\theta$. Since $T_1 \cdots T_m \in L(r)$ and $T_{i_1} \cdots T_{i_n}$ is a subsequence of $T_1 \cdots T_m$, we have $T_{i_1} \cdots T_{i_n} \in L(s)$. Thus, by rule (PATTERN), we deduce $D \vdash q : T \triangleright \Gamma_\theta$.
 - If $\alpha\beta = \{\}$ then $m = n$ and there exists a permutation π_Q of $[1, n]$, such that for each j in $\{1, \dots, n\}$, θ is an answer for q_j and $d_{\pi_Q(j)}$. Moreover, since $d \in \llbracket T \rrbracket$, there exists a permutation π_T of $[1, n]$ and a sequence of types $T_1, \dots, T_n$, such that $T_{\pi_T(1)} \cdots T_{\pi_T(n)} \in L(r)$ and for each $i \in \{1, \dots, n\}$, $d_i \in \llbracket T_i \rrbracket$. By induction hypothesis, for each $j \in \{1, \dots, n\}$, $D \vdash q_j : T_{\pi_Q(j)} \triangleright \Gamma_\theta$. Since $T_{\pi_T(1)} \cdots T_{\pi_T(n)} \in L(r)$, $T_1 \cdots T_n \in \text{perms}(L(r))$, and thus $T_{\pi_Q(1)} \cdots T_{\pi_Q(n)} \in \text{perms}(L(r))$. By rule (PATTERN), we deduce $D \vdash q : T \triangleright \Gamma_\theta$.

- If $\alpha\beta = \{\{\}\}$ then $m \geq n$ and there exists an injection $\pi_Q : \{1, \dots, n\} \rightarrow \{1, \dots, m\}$ such that for each $j \in \{1, \dots, n\}$, θ is an answer for q_j and $d_{\pi_Q(j)}$. Since $d \in \llbracket T \rrbracket$, there exists a permutation π_T of $[1, m]$ and a sequence of types $T_1, \dots, T_m$, such that $T_{\pi_T(1)} \cdots T_{\pi_T(m)} \in L(r)$ and for each $i \in \{1, \dots, m\}$, $d_i \in \llbracket T_i \rrbracket$. By induction hypothesis, we have, for each $j \in \{1, \dots, n\}$, $D \vdash q_j : T_{\pi_Q(j)} \triangleright \Gamma_\theta$. Let $k_1, \dots, k_n$ be such that $0 < k_1 < \dots < k_n \leq m$ and for each $p \in \{1, \dots, n\}$, $\exists j$ s.t. $k_p = \pi_T^{-1}(\pi_Q(j))$. Since $T_{\pi_T(1)} \cdots T_{\pi_T(m)} \in L(r)$, $T_{\pi_T(k_1)} \cdots T_{\pi_T(k_n)} \in L(s)$. Thus $T_{\pi_T(\pi_T^{-1}(\pi_Q(1)))} \cdots T_{\pi_T(\pi_T^{-1}(\pi_Q(n)))} \in \text{perms}(L(s))$, that is $T_{\pi_Q(1)} \cdots T_{\pi_Q(n)} \in \text{perms}(L(s))$. By rule (PATTERN), we deduce $D \vdash q : T \triangleright \Gamma_\theta$.
- If q is a variable X then $X\theta = d$. By construction $\Gamma_\theta(X) = T_1 \cap \dots \cap T_n$ with $\{T_1, \dots, T_n\} = \{T' \mid X\theta \in \llbracket T' \rrbracket\}$. Since $d \in \llbracket T \rrbracket$, there exists i such that $T = T_i$. Thus $\llbracket \Gamma_\theta(X) \rrbracket \subseteq \llbracket T \rrbracket$, and we obtain $\Gamma_\theta \sqsubseteq [X \mapsto T]$. By rule (VAR), we obtain $D \vdash X : T \triangleright \Gamma_\theta$.
- If q is of the form $X \rightsquigarrow q'$, then θ is an answer for q' and d . By induction hypothesis, we have $D \vdash q' : T \triangleright \Gamma_\theta$. Similarly to the preceding case, obtain $\Gamma_\theta \sqsubseteq [X \mapsto T]$. By rule (AS), we deduce $D \vdash X \rightsquigarrow q' : T \triangleright \Gamma_\theta$.
- If q is of the form **desc** q' , then θ is an answer for some subterm d' of d . We show the result by induction on d .
 - If $d = d'$ then, by the induction hypothesis for the query term q' , we have $D \vdash q' : T \triangleright \Gamma_\theta$. By rule (DESCENDANT), we obtain $D \vdash \text{desc } q' : T \triangleright \Gamma_\theta$.
 - If $d \neq d'$, then d is of the form $\lambda\alpha d_1, \dots, d_n\beta$ and there exists i , such that d' is a subterm of d_i . We have that θ is an answer for **desc** q' and d_i . Let r be the content model of T . there exists a type $T' \in \text{types}(r)$ such that $d_i \in \llbracket T' \rrbracket$. By induction hypothesis on the data term d_i , we have $D \vdash \text{desc } q' : T' \triangleright \Gamma_\theta$. Thus, by rule (DESCENDANT REC), we obtain $D \vdash \text{desc } q' : T \triangleright \Gamma_\theta$.

Proposition 23. *Let D be a type definition, d a data term and U a set of type names, such that $d \in \llbracket U \rrbracket$. Let Q be a query such that for each targeted query term $\text{in}(db, q)$ in Q there is a type name T in D such that $d(db) \in \llbracket T \rrbracket$. If θ is an answer for Q and d , then $D \vdash Q : U \triangleright \Gamma_\theta$.*

Proof. By induction on the query Q .

- If Q is a query term q , then θ is an answer for q and d , with $d \in \llbracket T_i \rrbracket$ for some $T_i \in U$. By Lemma 22, $D \vdash q : T_i \triangleright \Gamma_\theta$. By rule (QUERY TERM), we obtain $D \vdash Q : U \triangleright \Gamma_\theta$.
- If Q is a targeted query term $\text{in}(db, q)$, then $d(db) \in \llbracket T \rrbracket$ for some type name T in D . Moreover θ is an answer for q and $d(db)$. By Lemma 22, $D \vdash q : T \triangleright \Gamma_\theta$. By rule (TARGETED QUERY TERM), we obtain $D \vdash Q : U \triangleright \Gamma_\theta$.
- If Q is of the form **and**($Q_1, \dots, Q_p$), then for each $i \in \{1, \dots, p\}$, θ is an answer Q_i and d . By induction, we obtain, for each $i \in \{1, \dots, p\}$, $D \vdash Q_i : U \triangleright \Gamma_\theta$. Thus, by rule (AND QUERY), we have $D \vdash Q : U \triangleright \Gamma_\theta$.
- If Q is of the form **or**($Q_1, \dots, Q_p$), then for some $i \in \{1, \dots, p\}$, θ is an answer Q_i and d . By induction, we obtain $D \vdash Q_i : U \triangleright \Gamma_\theta$. Thus, by rule (OR QUERY), we have $D \vdash Q : U \triangleright \Gamma_\theta$.

Lemma 24. *Let D be a type definition, U be a set of type names, Q be a query and Γ be a variable-type mapping such that $D \vdash Q : U \triangleright \Gamma$. Let $Q' = \text{or}(Q_1, \dots, Q_n)$ be a disjunctive normal form of Q . If X is a variable occurring in each Q_i ($1 \leq i \leq n$) then $\Gamma(X) \neq 1$.*

Proof. If $D \vdash Q : U \triangleright \Gamma$ then $D \vdash Q' : U \triangleright \Gamma$ (we skip the details of the proof). Given $D \vdash Q' : U \triangleright \Gamma$ from the rule (OR QUERY) it holds $D \vdash Q_i : U \triangleright \Gamma$ for some $1 \leq i \leq n$. By induction on Q_i : for each variable in Q_i a rule (VAR) or (AS) must be triggered which impose that $\Gamma(X) \neq 1$.

Lemma 25. *Let D be a type definition, U be a set of type names, Q be a query term and Γ and Γ' be variable-type mappings such that $\Gamma(X) = \Gamma'(X)$ for all X occurring in Q . Then $D \vdash Q : U \triangleright \Gamma$ if and only if $D \vdash Q : U \triangleright \Gamma'$.*

Proof. It is sufficient to show, by induction on the derivation, that if $D \vdash Q : U \triangleright \Gamma$ then $D \vdash Q : U \triangleright \Gamma'$. The other direction is simply obtained by exchanging the roles of Γ and Γ' .

Lemma 26. *Let D be a type definition, U be a set of type names, Q be a query term. If there exists a variable-type mapping Γ such that $D \vdash Q : U \triangleright \Gamma$, then there exists a (finite) variable-type mapping set Ψ that is complete for U and Q wrt. D .*

Proof. Let Ψ_1 be the set of variable-type mappings Γ that can be defined in the following way:

$$\Gamma(X) = \begin{cases} T_1 \cap \dots \cap T_n \text{ for some } T_1, \dots, T_n \text{ defined in } D, \text{ if } X \text{ occurs in } Q \\ 1 \text{ otherwise.} \end{cases}$$

Since the set of variables occurring in Q and the set of type names defined in D are finite, Ψ_1 is also finite. Let $\Psi \subseteq \Psi_1$ be the set of variable-type mappings Γ such that $D \vdash Q : U \triangleright \Gamma$.

Let Γ' be a variable-type mapping such that $D \vdash Q : U \triangleright \Gamma'$. Let $\Gamma'' \in \Psi_1$ be a mapping such that for every X occurring in Q , $\Gamma''(X) = \Gamma'(X)$. By lemma 25, we obtain that $D \vdash Q : U \triangleright \Gamma''$, that is $\Gamma'' \in \Psi$. Thus, Ψ is complete for Q and U wrt D .

Lemma 27. *Let D be a type definition, c a construct term and Ψ a nonempty set of variable-type mappings such that $\{\Psi\} = \Psi / \sim_{FV(c)}^*$, and $\Gamma(X) \neq 1$ for any variable X occurring in c and any $\Gamma \in \Psi$. Then there exists a type definition $D' \supseteq D$ and a regular type expression S such that $D' \vdash c : \Psi \triangleright S$.*

Proof. By induction on a construct term c , it is clear that if $D_1 \vdash c : \Psi \triangleright S$, then for any $D_2 \supseteq D_1$, $D_2 \vdash c : \Psi \triangleright S$. Now we show the lemma, by induction on the construct term c .

- If c is a basic constant b , then the rule (CONST) can be applied, with $D' = D \cup \{T_c \rightarrow b\}$.

- Assume that c is a variable X . For each $\Gamma_i \in \Psi$, $\Gamma_i(X)$ is of the form $T_{i1} \cap \dots \cap T_{im_i}$. A type definition D_i (employing new type variables) can be constructed such that $\llbracket T_i \rrbracket_{D_i} = \llbracket T_{i1} \rrbracket_D \cap \dots \cap \llbracket T_{im_i} \rrbracket_D$ (see [7, Section 2.5.2] for details). Thus there exists a type definition $D' \supseteq D$ such that $\llbracket T_i \rrbracket_{D'} = \llbracket \Gamma_i(X) \rrbracket_{D'} = \llbracket \Gamma_i(X) \rrbracket_D$ for each $\Gamma_i \in \Psi$. The rule (VAR) produces $D' \vdash c : \Psi \triangleright T_1 \mid \dots \mid T_n$.
- If c is of the form $l\alpha c_1, \dots, c_n\beta$, then, by induction hypothesis, there exists $D_1, \dots, D_n$ such that for each $i \in \{1, \dots, n\}$, $D_i \vdash c_i : \Psi \triangleright S_i$ and $D_i \supseteq D$. Let $D' = \{T_c \rightarrow l\alpha S_1 \dots S_n \beta\} \cup \bigcup_{i=1}^n D_i$. We have, for each $i \in \{1, \dots, n\}$, $D' \vdash c_i : \Psi \triangleright S_i$. Thus, by applying (PATTERN), we obtain $D' \vdash c : \Psi \triangleright T_c$.
- If c is of the form **all** c' . Let $\{\Psi_1, \dots, \Psi_n\} = \Psi / \sim_{FV(C)}$. For each $i \in \{1, \dots, n\}$, $\{\Psi_i\} = \Psi_i / \sim_{FV(C)}$. Thus, by induction hypothesis, there exists $D_1, \dots, D_n$ such that, for each $i \in \{1, \dots, n\}$, $D_i \vdash c : \Psi_i \triangleright S_i$. Let $D' = \bigcup_{i=1}^n D_i$. Thus, for each $i \in \{1, \dots, n\}$, $D' \vdash c : \Psi_i \triangleright S_i$. By rule (ALL), we obtain $D' \vdash c : \Psi \triangleright T_c$.
- The case where c is of the form **some** $k c'$ is similar to the case **all** c' .

Lemma 28. *Let $D \subseteq D'$ be type definitions. If $D \vdash q : T \triangleright \Gamma$ (respectively $D \vdash Q : U \triangleright \Gamma$) then $D' \vdash q : T \triangleright \Gamma$ ($D' \vdash Q : U \triangleright \Gamma$).*

Proof. In any derivation tree of $D \vdash q : T \triangleright \Gamma$ (or $D \vdash Q : U \triangleright \Gamma$) we can replace D by D' , as $\Gamma \sqsubseteq_D [X \mapsto T]$ implies $\Gamma \sqsubseteq_{D'} [X \mapsto T]$ whenever T and each type name occurring in Γ occurs in D (or is a type constant).

Lemma 29. *Let D, D' be type definitions, Q a query, and U a set of type names. If $D \subseteq D'$ and Ψ is complete for Q and U wrt. D then Ψ is complete for Q and U wrt. D' .*

Proof. The first condition of Definition 18 follows from Lemma 28 (for each $\Gamma \in \Psi$ if $D \vdash Q : U \triangleright \Gamma$ then $D' \vdash Q : U \triangleright \Gamma$).

To show the second condition, assume $D' \vdash Q : U \triangleright \Gamma$ and consider the corresponding derivation tree. Let $\Gamma' = [X_1 \mapsto S_1, \dots, X_n \mapsto S_n]$, where $X_1, \dots, X_n$ are all the variables occurring in the tree, each S_i is of the form $T_{i1} \cap \dots \cap T_{im_i}$ and T_{ij} occurs in S_i iff the condition $\Gamma \sqsubseteq [X_i \mapsto T_{ij}]$ occurs in the tree (in an instance of a rule (VAR) or (AS)). The mapping Γ in the tree can be replaced by Γ' . Hence $D' \vdash Q : U \triangleright \Gamma'$. Then $D \vdash Q : U \triangleright \Gamma'$, as all the type variables occurring in Γ' are in U , and all the type variables from U occur in D . Moreover $\Gamma \sqsubseteq_{D'} \Gamma'$. As Ψ is complete for Q and U wrt. D then $\Gamma' \sqsubseteq_D \Gamma_i$ for some $\Gamma_i \in \Psi$. Hence $\Gamma' \sqsubseteq_{D'} \Gamma_i$ and $\Gamma \sqsubseteq_{D'} \Gamma_i$.

Lemma 30. *Let D be a type definition, Θ be a set of substitutions, and Ψ a set of variable-type mappings such that $\Theta \subseteq \text{substitutions}(\Psi)$. Let V be a set of variables such that $V \subseteq \text{dom}(\theta)$ for each $\theta \in \Theta$. If $\Theta' \in \Theta / \simeq_V$ then there exists $\Psi' \in \Psi / \sim_V$ such that $\Theta' \subseteq \text{substitutions}(\Psi')$.*

Proof. Let us consider $\Theta' \in \Theta / \simeq_V$. Let $\theta' \in \Theta'$. Since $\Theta' \subseteq \text{substitutions}(\Psi)$, there exists $\Gamma' \in \Psi$ such that $\theta' \in \text{substitutions}(\Gamma')$. Let $\Psi' \in \Psi / \sim_V$ be such that

$\Gamma' \in \Psi'$. Let $\theta'' \in \Theta'$. There exists $\Gamma'' \in \Psi$ such that $\theta'' \in \text{substitutions}(\Gamma'')$. Thus for every $X \in V$, $X\theta'' \in \llbracket \Gamma''(X) \rrbracket$. Since $\theta' \simeq_V \theta''$, we have $\forall X \in V, X\theta' = X\theta''$. Since $\theta' \in \llbracket \Gamma' \rrbracket$, we obtain $\forall X \in V, X\theta'' = X\theta'$ and $X\theta' \in \llbracket \Gamma'(X) \rrbracket$. This means that for every $X \in V$, $\llbracket \Gamma'(X) \rrbracket \cap \llbracket \Gamma''(X) \rrbracket \neq \emptyset$, that is $\Gamma' \sim_V^* \Gamma''$. Thus $\theta'' \in \text{substitutions}(\Psi')$.

Proposition 31. *Let D be a type definition, Ψ a set of mappings, c be a construct term and S a regular type expression such that $\{\Psi\} = \Psi / \sim_{FV(c)}$ and $D \vdash c : \Psi \triangleright S$. Let Θ be a set of substitutions such that $\Theta \subseteq \text{substitutions}(\Psi)$ and $\{\Theta\} = \Theta / \sim_{FV(c)}$. Then $\Theta(c) \in \llbracket S \rrbracket$.*

Proof. By induction on the construct term c .

- If $c = b$, where b is a basic constant, then the typing rule used to deduce $D \vdash c : \Psi \triangleright S$ is (CONST). We have $\Theta(c) = b$. Moreover, we have $S = T_c$, and, since the production rule for T_c is $T_c \rightarrow b$, we obtain $\Theta(c) \in \llbracket S \rrbracket$.
- If c is a variable X , the typing rule used to deduce $D \vdash c : \Psi \triangleright S$ is (VAR). We have $\Theta(X) = X\theta$ for some $\theta \in \Theta$. Since $\Theta \subseteq \llbracket \Psi \rrbracket$, $X\theta \in \llbracket \Gamma(X) \rrbracket$, for some $\Gamma \in \Psi$, and thus $X\theta \in \llbracket \Gamma_1(X) \rrbracket \cup \dots \cup \llbracket \Gamma_n(X) \rrbracket$. Thus $\Theta(c) \in \llbracket S \rrbracket$.
- If c is of the form $\lambda\alpha c_1, \dots, c_n\beta$, then the typing rule used to deduce $D \vdash c : \Psi \triangleright S$ is (PATTERN). Thus for each $i \in \{1, \dots, n\}$, $D \vdash c_i : \Psi \triangleright S_i$. By induction hypothesis, for each $i \in \{1, \dots, n\}$, $\Theta(c_i) \in \llbracket S_i \rrbracket$. Therefore $\Theta(c_1) \circ \dots \circ \Theta(c_n) \in \llbracket S_1 \dots S_n \rrbracket$. Since the production rule for T_c is $T_c \rightarrow \lambda\alpha S_1 \dots S_n\beta$ and $\Theta(c) = \lambda\alpha\Theta(c_1) \circ \dots \circ \Theta(c_n)\beta$, we obtain $\Theta(c) \in \llbracket T_c \rrbracket$.
- If c is of the form **all** c' , then the typing rule used to deduce $D \vdash c : \Psi \triangleright S$ is (ALL). Let $\{\Theta_1, \dots, \Theta_p\} = \Theta / \sim_{FV(c')}$. By lemma 30, there exists $\pi : [1, p] \rightarrow [1, n]$ such that for each $i \in \{1, \dots, p\}$, $\Theta_i \subseteq \Psi_{\pi(i)}$. Moreover for each $j \in \{1, \dots, n\}$, we have $D \vdash c' : \Psi_j \triangleright S_j$. By induction hypothesis, $\Theta_i(c') \in \llbracket S_{\pi(i)} \rrbracket \subseteq \llbracket S_1 \mid \dots \mid S_n \rrbracket$. Thus $\Theta(c) = \Theta_1(c') \circ \dots \circ \Theta_p(c') \in \llbracket (S_1 \mid \dots \mid S_n)^+ \rrbracket$.
- If c is of the form **some** k c' , then, similarly to the case for **all** c' , we have $\Theta_i(c') \in \llbracket S_1 \mid \dots \mid S_n \rrbracket$, where $\{\Theta_1, \dots, \Theta_m\} \subseteq \Theta / \sim_{FV(c')}$ with $1 \leq m \leq k$. Thus $\Theta(c) = \Theta_1(c') \circ \dots \circ \Theta_m(c') \in \llbracket (S_1 \mid \dots \mid S_n)^{(1:k)} \rrbracket$.

Now, we recall Theorem 20 and show its proof.

Theorem 20. *Let D be a type definition and $(c \leftarrow Q)$ be a query rule, where for each targeted query term $\text{in}(db, q)$ in Q there is a type name T in D such that $d(db) \in \llbracket T \rrbracket$. Let U be a set of type names and d a data term such that $d \in \llbracket U \rrbracket$.*

If a result for $(c \leftarrow Q)$ and d exists then there exist S and D' such that $D' \supseteq D$ and $D' \vdash (c \leftarrow Q) : U \triangleright S$.

If there exists S such that $D \vdash (c \leftarrow Q) : U \triangleright S$ and if d' is a result for $(c \leftarrow Q)$ and d , then $d' \in \llbracket S \rrbracket$.

Proof. Let us assume that there exists a result for $(c \leftarrow Q)$ and d . By Proposition 23, $D \vdash Q : U \triangleright \Gamma_\theta$. Thus, by Lemma 26, there is a set Ψ of mappings that is complete for Q and U and D .

Let V_Q be the set of variables occurring in Q and V_c the set of variables occurring in c . Let $\text{or}(Q'_1, \dots, Q'_n)$ be the disjunctive normal form of Q . Since

Xcerpt requires that, for $i \in \{1, \dots, n\}$, all variables in c must occur Q'_i , by Lemma 24, we have that for all variables $X \in V_c$ and all mappings $\Gamma \in \Psi$, $\Gamma(X) \neq 1$.

Let $\{\Psi_1, \dots, \Psi_n\} = \Psi / \sim_{FV(c)}^*$. By induction from Lemma 27 we obtain that there exist $D_n \supseteq \dots \supseteq D_1 \supseteq D$ and $S_1, \dots, S_n$ such that $D_i \vdash c : \Psi_i \triangleright S_i$ (and $D_n \vdash c : \Psi_i \triangleright S_i$) for each $i \in \{1, \dots, n\}$. By Lemma 29, Ψ is also complete for Q and U wrt. D_n . By rule (QUERY RULE), we obtain $D_n \vdash (c \leftarrow Q) : U \triangleright S_1 \mid \dots \mid S_n$.

Now assume that there exists S such that $D \vdash (c \leftarrow Q) : U \triangleright S$. Let Θ be the set of all answers for Q and d and let $\theta \in \Theta$. By Proposition 23, $D \vdash Q : U \triangleright \Gamma_\theta$. Since Ψ used in the rule (QUERY RULE) is complete for Q and U wrt. D , there exists $\Gamma \in \Psi$ such that $\Gamma_\theta \sqsubseteq \Gamma$. Since $\theta \in \text{substitutions}(\Gamma_\theta)$, we obtain $\theta \in \text{substitutions}(\Psi)$. Thus $\Theta \subseteq \text{substitutions}(\Psi)$.

We have $d' = \Theta'(c)$ for some $\Theta' \in \Theta / \sim_{FV(c)}$. By Lemma 30, there exists $\Psi' \in \Psi / \sim_{FV(c)}^*$ such that $\Theta' \subseteq \Psi'$. Since $D \vdash (c \leftarrow Q) : U \triangleright S_1 \mid \dots \mid S_n$, then for some $i \in \{1, \dots, n\}$, we have $\Psi_i = \Psi'$ and $D \vdash c : \Psi_i \triangleright S_i$. By Proposition 31, $d' = \Theta'(c) \in \llbracket S_i \rrbracket \subseteq \llbracket S_1 \mid \dots \mid S_n \rrbracket$.