Detection of Corrupted Schema Mappings in XML Data Integration Systems

DARIO COLAZZO
LRI
and
CARLO SARTIANI
Università di Pisa

In modern data integration scenarios, many remote data sources are located on the Web and are accessible only through forms or web services, and no guarantee is given about their stability. In these contexts the detection of corrupted mappings, as a consequence of a change in the source or in the target schema, is a key problem. A corrupted mapping fails in matching the target or the source schema, hence it is not able to transform data conforming to a schema \( S \) into data conforming to a schema \( T \), nor it can be used for effective query reformulation.

This paper describes a novel technique for maintaining schema mappings in XML data integration systems, based on a notion of mapping correctness relying on the denotational semantics of mappings.

Categories and Subject Descriptors: H. Information Systems [H.2 DATABASE MANAGEMENT]; H.2.5 Heterogeneous Databases

General Terms: Data Integration
Additional Key Words and Phrases: XML, data integration, data exchange, p2p systems, type systems, type inference, mapping correctness

1. INTRODUCTION

The astonishing growth of available information, on both the Web and local computers, has significantly redefined the requirements and the needs of modern data management: indeed, the problem of finding the right piece of information at the right time has evolved and shifted from a relatively centralized context to an highly decentralized and distributed one. While traditional data management has focused on data hosted on a single data source (e.g., a relational or an object-oriented database), the fragmentation of information on multiple heterogeneous sources, often autonomous and accessible only through forms or low-level web services, implies the definition and development of new techniques and tools for their access.

This scenario shift significantly impacts on current data integration solutions, which address the problem of providing a uniform, declarative interface to access data dispersed on multiple source, possibly heterogeneous and autonomous. Indeed, integrating remote data sources, which offer no guarantees about the stability of their schemas and/or query capabilities, requires the adoption of new techniques for solving the most important integration tasks.
A main task in data integration is the maintenance of schema mappings, and, in particular, the detection of corrupted mappings. A schema mapping from a source schema $S$ to a target schema $T$ describes how to translate data conforming to $S$ into data conforming to $T$, and it can be used to reformulate queries on $S$ into queries over $T$, and vice versa, according to the Global-As-View (GAV), Local As View (LAV), and Global-And-Local-As-View (GLAV) paradigms [Ullman 1988; 1989; Friedman et al. 1999]. Schema mappings are used during query answering for reformulating queries or, as in data exchange systems [Arenas and Libkin 2005], for generating canonical solutions; schema mappings, hence, allow the system to retrieve data that are semantically similar but described by different schemas.

Given the role played by schema mappings, a corrupted mapping, which fails in matching the source or the target schema, can significantly affect query processing, as it may make no more accessible the corresponding remote data source or may produce meaningless query results.

Mapping maintenance is a time-consuming, complex, and expensive activity, and is usually performed by the system/site administrator, who manually inspects schemas and mappings in order to find errors in mappings definitions; as a consequence, quick responses to sudden mapping corruptions are not possible. To aid and accelerate errors detection, several tools and techniques for assisting the administrator in maintaining schema mappings have been described in the recent past (see [McCann et al. 2005], for instance). These techniques are usually based on the monitoring of some arbitrary parameter, like, for instance, the “quality” of samples of transformed data instances or transformed queries, and usually do not offer guarantees about the completeness of the approach.

More in details, the above mentioned techniques have two main drawbacks. First, they are not complete since wrong rules that are not used for reformulating a query or for transforming a sample data instance cannot be discovered. Second, they usually require an interaction with query answering or data transformation algorithms; this implies that these techniques cannot directly check for mapping correctness, but, instead, check for the correctness of a mapping wrt a given reformulation or transformation algorithm. Hence, a significant weakness of these techniques is that they refer to a notion of mapping correctness which is strongly related to the properties of a particular reformulation and/or transformation algorithm.

Our Contribution. This paper describes a novel technique for maintaining schema mappings in XML data integration systems, where mappings are specified by means of XQuery clauses [Boag et al. 2007].

Differently from other approaches, our approach relies on a notion of mapping correctness which is semantic, in the sense that only the pure denotational semantics of the mapping is taken into account in its characterization, thus ensuring a full independence wrt specific implementation issues of the integration system (e.g., ad-hoc query reformulation algorithms). In other words, if new algorithms for, let’s say, query reformulation are adopted, correctness of mappings remains unchanged. Hence, correctness is clearly and formally defined in terms of the formal properties of declarative schema mappings.

More in details, given a schema mapping $m$ from $S$ to $T$, we assume that a schema $S_m$ describing the structure of the image $m(S)$ is available (through an inference process), and then we compare $S_m$ wrt the target schema $T$ according to a type projection notion, which generalizes the notion of relational projection, and captures and formalizes the intended semantics of mapping correctness of typical data integration systems [Halevy et al. 2003]. If this comparison for projection
succeeds, we are sure that the mapping rules describe data that are “compatible” with the target schema; moreover, as an important consequence, if the mapping is deemed as correct, then reformulated queries will always be consistent with the target schema.

The above depicted framework requires the existence of an output type $S_m$ for $m$; in order to show that this assumption is not restrictive, we will provide a quite efficient type inference system, able to infer such upper-bound $S_m$ at static time, starting from $m$ and its input schema $S$; we will also show that the inferred schema $S_m$ is quite close to $m(S)$, thus entailing an high degree of precision in the corruption checking process.

To summarize, the main contributions of the present work are the following:

— we formally define and characterize a relation over XML data, called XML data projection; this relation, together with the dynamic semantics of mappings, is used to characterize a notion of schema mapping correctness;

— to move toward a static correctness checking algorithm, we transpose the notion of data projection to XML types, thus obtaining a notion of XML type projection relation;

— we prove that type projection is decidable by showing its equivalence to a restricted form of subtyping; we also study the relationships between projection and standard subtyping;

— we prove that type projection is inherently hard to check, by showing a NP-hardness lower bound;

— we provide an algorithm for checking type projection, and prove that the algorithm is polynomial in most cases;

— we provide a type inference system to statically infer an output type $S_m$ for $m$, starting from its input schema $S$; we will prove that, for a wide class of cases, the inferred type $S$ is equivalent to $m(S)$ up to type projection; this will directly entail the absence of false negatives in correctness checking, which is performed by checking that $S_m$ is a projection of the $m$ target schema $T$.

As already mentioned, we believe that one the main strengths of this approach lies on the fact that the combination of type projection and type inference results in a technique that is independent from queries posed against the integrated database, does not rely on query reformulation algorithms/techniques, and it is complete, i.e., any incorrect mapping will be detected. As a final remark, the solution proposed in this paper can be used in both traditional and decentralized data integration systems, as well as in data exchange systems ([Fuxman et al. 2005]).

**Paper Outline.** The paper is structured as follows. Section 2 describes a reference scenario for our technique, while Section 3 introduces our query/mapping and type languages, and then defines our notions of mapping validity (no wrong rules wrt the source schema) and mapping correctness (no wrong rules wrt the target schema). Section 4, next, explores the formal properties of type projection. Section 5, then, delimits the kinds of schema changes that can corrupt a mapping. Section 6 proves the decidability of type projection and shows the NP-hardness complexity lower bound for type projection checking. Section 7, then, illustrates the type simulation relation, discusses its properties, and shows its equivalence to type projection. Section 8, then, describes the type projection checking algorithm and discusses its complexity. Section 9 describes the type system we use for inferring query output types. Section 10, next, presents an experimental evaluation of our approach and
discusses its precision and scalability properties. Section 11, then, discusses some related work. In Section 12, finally, we draw our conclusions.

2. MOTIVATING SCENARIO

We motivate our technique by referring to a decentralized data integration scenario, where multiple data sources are connected by means of one-to-one mappings. This scenario is a generalization of centralized approaches, where each data source is mapped into a (single) global schema. For the sake of simplicity, we assume a minimal configuration, comprising two data sources only (p₁ and p₂), so to focus on mapping correctness rather than on query reformulation or routing issues.

Each data source hosts a bunch of XML data, that are described by a schema (S for p₁ and T for p₂); these schemas are connected through a schema mapping (in the following we will use the expression “schema mapping” to denote any mapping between types). The mapping can be defined according to the Global-As-View (GAV) approach, or to the Local-As-View (LAV) approach. Our approach is based on LAV mappings, where the target (local) schema is described in terms of the source (global) schema; nevertheless, this approach applies to GAV mappings too, since, as noted in [Tatarinov 2004], a LAV mapping from p₁ to p₂ can be interpreted as a GAV mapping from p₂ to p₁.

In our framework, a mapping m from S to T is a set of queries that specify how to translate data belonging to S into data conforming to a projection of T. A mapping, hence, can be regarded as a specification rather than an actual transformation from S to T, as it is not forced to detail the construction of all target elements.

Mapping queries are expressed in the same query language used for posing general queries: this language, called µXQ, is roughly equivalent to the FLWR core of XQuery, and will be described in Section 3.

Data integration scenarios like this are usually managed with mediation approaches, where queries are reformulated by means of schema mappings and no centralized warehouse is used. The correctness of the query answering process for a given query depends on the properties of the reformulation algorithm as well as on the correctness of the mappings involved in the transformation: indeed, if the mapping fails in matching the target schema, the transformed query will probably fail as well.

The evolution of the integrated database, namely the changes in data source schemas, can dramatically affect the quality of schema mappings and, in particular, lead to the corruption of existing mappings. This will reflect on query answering and on existing optimization techniques for decentralized and centralized systems, such as the mapping composition approach described in [Tatarinov and Halevy 2004].

The following Example illustrates the basic concepts of the query language, provides an intuition of the projection-based mapping correctness notion (formally characterized in Section 3), and shows how mapping incorrectness can reflect on query answering and data transformation.

Example 2.1. Consider a decentralized data sharing system for music information. The system allows users to share data about their (legally owned) music files, so to discover information about their preferred songs and singers. Each user publishes, on a voluntary basis, the description of all the songs she is storing on her computer or iPod.

Assume that a user in Cupertino publishes her music database according to the
following schema\(^1\).

\[
\text{CupMDB} = \text{mySongs}[(\text{Song})^*]
\]
\[
\text{Song} = \text{song}[\text{Title}, \text{Artist}, \text{Album}, \text{MyRating}]
\]
\[
\text{Title} = \text{title}[\text{String}]
\]
\[
\text{Artist} = \text{artist}[\text{String}]
\]
\[
\text{Album} = \text{album}[\text{String}]
\]
\[
\text{MyRating} = \text{myRating}[\text{Integer}]
\]

This schema groups data by song, and, for each song, represents the title, the artist name (a singer or a band), the album title, as well as a personal rating information.

Suppose now that another user in Seattle publishes her database according to the following (different) schema.

\[
\text{SeattleMDB} = \text{musicDB}[\text{Artist}^*]
\]
\[
\text{Artist} = \text{artist}[\text{Name}, \text{Provenance}, \text{Track}^*]
\]
\[
\text{Name} = \text{name}[\text{String}]
\]
\[
\text{Provenance} = \text{provenance}[\text{Continent}, \text{Country}]
\]
\[
\text{Continent} = \text{continent}[\text{String}]
\]
\[
\text{Country} = \text{country}[\text{String}]
\]
\[
\text{Track} = \text{track}[\text{Title}, \text{Year}, \text{Genre}]
\]
\[
\text{Title} = \text{title}[\text{String}]
\]
\[
\text{Year} = \text{year}[\text{Integer}]
\]
\[
\text{Genre} = \text{genre}[\text{String}]
\]

This schema groups data by artist and, for each artist, details her name and provenance, as well as the list of corresponding tracks.

To make these databases interact together, a proper schema mapping is required, as schemas nest data in very different ways. Assume that the user in Cupertino employs the following mapping (a set of XQuery-like queries) to map her schema into the Seattle-based schema\(^2\).

\[
\text{SeattleMDB} \leftarrow 
\]
\[
Q_1(\text{input}): \text{for } \$a \text{ in } \text{input/song/artist}, \\
\text{return } \text{artist}[
\text{name}[\$a/data()], \\
\text{for } \$s \text{ in } \text{input/song}, \\
\text{\$art in } \$s/\text{artist} \\
\text{where } \$art/data() = \$a/data() \\
\text{return } \text{track}[
\text{for } \$t \text{ in } \$s/\text{title}, \\
\text{return } \$t]
\]
\[
Q_2(\text{input}): \text{for } \$db \text{ in } /\text{mySongs} \\
\text{return } \text{musicDB}[Q_1(\$db)]
\]

This mapping specifies how data conforming to a fragment of the Cupertino schema (\text{album} and \text{myRating} elements are discarded) can be transformed into data conforming to a fraction of the Seattle-based schema (for instance, \text{provenance} elements are discarded). In other words, the schema mapping takes into account a projection of the two schemas. This is a very common situation in data integration systems, as usually only a fraction of semantically related heterogeneous schemas can be reconciled. In particular, as the Seattle user schema does not support \text{album} and \text{myRating} elements, they must be ignored in the mapping. Furthermore, since the Cupertino schema does not provide information about song genre, corresponding elements are not generated, hence any transformed data instance must be regarded as

\(^1\)For the sake of simplicity, we are using here the XDuce [Hosoya and Pierce 2003] type language.

\(^2\)Even though this mapping can be expressed by a single query, by nesting \(Q_1\) into \(Q_2\), we prefer to keep them separated for the sake of clarity.
as a projection of a Seattle-compliant data instance. Being this mapping only a specification, the actual transformation can be derived by applying, for instance, a chase-like approach à la Clio.

Assume now that Seattle slightly changes its schema and, in particular, the way artist names are represented: instead of a simple name element, information about artist’s first name and second name is inserted into the name element: Name = \text{name[first[String],second[String]]}.

This change in the target schema makes the Cupertino → Seattle mapping incorrect. Indeed, this mapping specifies the construction of simple content name elements, which are now no more allowed in the target schema. It should be observed that chasing the mapping cannot fix this problem, as incorrect mappings generate incorrect transformations.

Incorrectness of the mapping from Cupertino to Seattle has two main consequences. First of all, the actual transformation that we can derive from the mapping fails in creating instances of SeattleMDB from instances of CupMDB: indeed, the transformation still generates simple name elements in the target instance, which can no longer be accepted and validated against the target schema. A second consequence is that, just as for data transformation, even query reformulation fails, in the sense that any query involving name elements is incorrectly reformulated. To illustrate this point, consider the query shown in Figure 1 (a). This query, submitted by a user in Cupertino, asks for the titles of all songs published by Burt Bacharach. The query is first executed locally in Cupertino. Then, the system reformulates the query so to match Seattle schema; this reformulation is performed by using standard LAV query rewriting algorithms (one can think of CupMDB as the global schema and SeattleMDB as the local schema) [Madhavan and Halevy 2003; Tatarinov and Halevy 2004]3.

At the end of the reformulation process, the reformulated query, shown in Figure 1 (b), is then sent to the Seattle site. Unfortunately, the transformed query does not match the new schema of Seattle users, so the Cupertino user cannot gather results from the Seattle site.

This example clearly pinpoints the maintenance issues that arise in data integration systems and, in particular, in those systems reconciling autonomous, web-based data sources. Furthermore, this example highlights that the relationship between a schema mapping and its target schema cannot be modeled through a standard subtyping relation à la XQuery/XDuce/CDuce, as this form of subtyping is based on set inclusion. Indeed, the output type of the mapping is not a subtype of the

\footnote{3We show a minimal transformed query, obtained by minimizing the original transformed query and by deleting all redundant subqueries.}
target type, as the target schema prescribes the presence of a genre element, which, instead, is not generated by the mapping.

The nature of schema mappings imposes a more flexible and general way of comparing types than a subtyping-based comparison. This is the main motivation for the introduction of type projection, which captures the Piazza [Halevy et al. 2003] intuition of mappings as “transformation + projection” (i.e., non-functional transformation). We quote a part of this work:

At the core, the semantics of mappings can be defined as follows. Given an XML instance, $I_S$, for the source node $S$ and the mapping to the target $T$, the mapping defines a subset of an instance, $I_T$, for the target node. The reason that $I_S$ is a subset of the target instance is that some elements of the target may not exist in the source (e.g., the publisher element in the examples). In fact, it may even be the case that required elements of the target are not present in the source. In relational terms, $I_T$ is a projection of some complete instance $I_T'$ of $T$ on a subset of its elements and attributes.

This characterization is, indeed, common to most data integration and data exchange systems, and points out that a schema mapping specifies a non-functional transformation from a source schema $S$ to a target schema $T$.

In the following sections we will provide a formalization of type projection for an even wider class of schema mappings: we will regard a mapping as a set of rules that specify how to transform a source data instance $I_S : S$ into a fragment of one or more data instances $I_T$ conforming to $T$, the actual transformation from $S$ to $T$ being obtained through a chase-like process.

3. MAPPING VALIDITY AND CORRECTNESS

In this section we describe the notions of mapping validity (no wrong rules wrt the source schema) and mapping correctness (no wrong rules wrt the target schema). These notions are central to our approach, and allow for the definition of an operational checking technique, as shown in Sections 4 and 7.

To define mapping properties, we have to formally present the data model, the query language used for expressing both user queries and mapping rules, as well as the type language used to represent schemas.

3.1 Data Model and Query Language

We represent an XML database as a forest of unranked, unordered, node-labeled trees, as shown by the following grammar.

$$f ::= () | b | l[f] | f, f$$

A forest $f$ essentially denotes an XML fragment. It may be either an empty sequence ($()$), a base value $b$ (without loss of generality, we only consider string base values), an element node $l[f']$ labeled as $l$ and whose children form a forest $f'$, or a concatenation $f', f''$.

Since our study started from a decentralized perspective, we drop ordering from the model, as no global order can be enforced on data coming from multiple sources. Hence, forest concatenation $f, f'$ is commutative, associative, and has $()$ as neutral element.

\[\text{For the sake of simplicity, we do not explicitly represent attributes, as they can be easily encoded as element nodes.}\]
We will consider user queries and mapping rules defined according to the $\mu$XQ query language [Colazzo et al. 2004], whose grammar is shown in Table 3.1. $\mu$XQ is a minimal core language for XML data, roughly equivalent to the FLWR core of XQuery; we chose a slightly different syntax wrt XQuery only to ease the formal treatment. We impose two further restrictions wrt this grammar: first, we forbid, inside an outer query, the navigation over elements newly created by an inner query; second, we restrict the predicate language to the conjunction, disjunction, or negation of variable comparisons. These restrictions, also present in Piazza, allow for a better handling of errors at the price of a modest decrease in the expressive power of the language. Note that as in XQuery, the semantics of comparison $\chi_1 \delta \chi_2$ is existential.

As pointed out in the grammar, in $\mu$XQ we distinguish between for-variables ($x$) and let-variables ($x$). Indeed, to simplify the formal treatment, we assume that navigational operations (child and dos) always start from a single node; to meet this assumption it is necessary to distinguish between variables bound to singletons (for-variables) and variables that may be bound to collections (let-variables). Typical XPath-like clauses $Q/l$ and $Q//l$ (sometimes used in the following examples) can be expressed in $\mu$XQ as follows:

$$Q/l \triangleq \text{for } x \text{ in } Q \text{ return } x \text{ child :: } l$$

$$Q//l \triangleq \text{for } x \text{ in } Q \text{ return (for } y \text{ in (} x \text{ child :: node()} \text{ return } y \text{ dos :: } l)$$

While being a core language, $\mu$XQ is still very expressive. This implies that the techniques we are presenting here can be easily applied to other formalisms, like, for instance, the mapping language of Piazza or the source-to-target dependencies of Clio [Popa et al. 2002].

The semantics of the language and the required auxiliary functions are shown in Tables 3.2 and 3.3. There, $\rho$ is a substitution assigning a forest to each free variable in the query; we make the assumption that each $\rho$ is well-formed, meaning that it always associates a tree to a for-variable it defines; also, dos is a shortcut for descendant-or-self. The semantics of for queries is defined via the operator $\prod_{t \in \text{trees}(f)} A(t)$, yielding the forest $A(t_1), \ldots, A(t_n)$, when $f = t_1, \ldots, t_n$, and () when $f = ()$. In Table 3.3 the notation $P(\rho)$ indicates the truth value obtained by evaluating the predicate $P$ under a variable assignment environment $\rho$, as indicated in Table 3.4. All the rest is self explicative.

It is worth noting that, differently from XQuery, and in order to simplify the formal treatment, $\mu$XQ semantics does not take into account elements id’s. This difference is not influential in this setting, as in our framework queries are used to infer query output types, and the process of type inference is independent from elements id’s. Also, it can be easily proved that our inferred types are sound, in the sense that they denote upper bounds for query semantics, even if XQuery standard semantics is adopted.

### 3.2 Type Language

Our type language, based on XDuce [Hosoya and Pierce 2003] and XQuery [Draper et al. 2007] type languages, is shown in Figure 2, where () is the type for the empty sequence value, $B$ denotes the type for base values (without loss of generality, we only consider string base values), types $T, U$ and $T \mid U$ are, respectively, product and union types, and, finally, $T^*$ is the type for repetition. Types are unordered, as no global order on XML data dispersed on multiple sources can be established: this aspect significantly increases the hardness of comparing two XML types, as
Table 3.1. μXQ grammar

\[ Q ::= () | b | l(Q) | Q, Q | \mathbf{X} \text{child} :: \text{NodeTest} | \mathbf{X} \text{dos} :: \text{NodeTest} \]

| for in Q return Q | let x ::= Q return Q |
| for in Q where P return Q | let x ::= Q where P return Q |

\[
\begin{align*}
\text{NodeTest} & ::= 1 | \text{node()} | \text{text()} \\
P & ::= \text{true} | \chi \delta \chi | \text{empty}(\chi) | P \text{ or } P | \text{not } P | (P) \\
\chi & ::= \mathbf{X} | x \\
\delta & ::= = | <
\end{align*}
\]

Table 3.2. μXQ semantics

\[
\begin{align*}
[0]_\rho & ::= b \\
[x]_\rho & ::= \rho(x) \\
[\mathbf{X}]_\rho & ::= \rho(\mathbf{X}) \\
[l(Q)]_\rho & ::= l([Q])_\rho \\
[Q_1, Q_2]_\rho & ::= [Q_1]_\rho, [Q_2]_\rho \\
\mathbf{X} \text{child} :: \text{NodeTest} & ::= \text{childr}([\mathbf{X}]_\rho) :: \text{NodeTest} \\
\mathbf{X} \text{dos} :: \text{NodeTest} & ::= \text{dos}([\mathbf{X}]_\rho) :: \text{NodeTest} \\
\text{let } x ::= Q_1 \text{ return } Q_2 & ::= [Q_2]_\rho x \rightarrow [Q_1]_\rho \{Q_2\}_\rho \\
\text{let } x ::= Q_1 \text{ where } P \text{ return } Q_2 & ::= \text{if } P(\rho, x) \rightarrow [Q_1]_\rho \text{ then } [Q_2]_\rho x \rightarrow [Q_1]_\rho \text{ else } () \\
\text{let } x ::= Q_1 \text{ where } P \text{ return } Q_2 & ::= \prod_{\mathbf{X} \in \text{trees}([Q_1]_\rho)} [Q_2]_\rho x \rightarrow [Q_1]_\rho \\
\text{let } x ::= Q_1 \text{ return } Q_2 & ::= \text{if } P(\rho, x) \rightarrow [Q_1]_\rho \text{ then } [Q_2]_\rho x \rightarrow [Q_1]_\rho \text{ else } ()
\end{align*}
\]

Table 3.3. Auxiliary functions

\[
\begin{align*}
\text{dos}(b) & ::= b \\
\text{childr}(b) & ::= () \\
\text{dos}(l(f)) & ::= l([f]) \text{ dos}(f) \\text{childr}([l(f)]) \\
\text{dos}() & ::= () \\
\text{dos}(f, f') & ::= \text{dos}(f), \text{dos}(f') \\
\text{b::l} & ::= l([f]) :: l \\
\text{f::node}() & ::= f \\
\text{b::text}() & ::= b \\
\text{m}[f]::\text{text}() & ::= m \neq l \\
\text{f::node}() & ::= f \\
\text{b::text}() & ::= b \\
\text{m}[f]::\text{text}() & ::= (f, f') :: \text{text}() \\
\end{align*}
\]

usual heuristics and optimizations based on type ordering cannot be applied in this context.

Since types are unordered, in the following we will consider a product type \( T_1, \ldots, T_n \) as identical to all its possible permutations \( T_{\pi(1)}, \ldots, T_{\pi(n)} \). Moreover, as our types actually are XDuce unordered types, we also have that \( T() \) is identical to \( T \), and that \( (T, T'), (T'', T''') \) is identical to \( (T', T'') \). This conforms to the corresponding laws over the data model.

Furthermore, our type language includes horizontal recursive types (allowed by types \( T^* \)) but does not include vertical recursive types, like the one defined by this recursive definition \( \text{Part} = \text{partname}[\text{Description}, \text{Part}^*] \). This is motivated by the fact that most mapping languages are not powerful enough to transform trees with arbitrary depth, whose structure can only be defined by vertical recursive types. Hence we can restrict the type language to types that describe trees with finite and upper-bounded depth. Also, it should be observed that many mapping
Table 3.4. Predicates evaluation

\begin{tabular}{|c|c|}
\hline
true(\rho) & $\triangleq$ true \\
(P or P')(\rho) & $\triangleq$ P(\rho) or P'(\rho) \\
(not P)(\rho) & $\triangleq$ not P(\rho) \\
\hline
\end{tabular}

\[(\chi \delta \chi)(\rho) = \exists t \in \text{trees}(\rho(\chi)), t' \in \text{trees}(\rho(\chi')). t \delta t'\]

\[\text{empty}(\chi)(\rho) = \text{if } \rho(\chi) = () \text{ then true else false}\]

Types

\[T ::= () \text{ empty sequence} \]

| $| B \text{ base type} $ |
|-------------------------|
| $| l[T] \text{ element type} $ |
| $| T,T \text{ sequence type} $ |
| $| T | T \text{ union type} $ |
| $| T^* \text{ repetition type} $ |

Base Type

\[B ::= \text{String}\]

tools like Clio do not support recursive types, as chasing (i.e., the closure of a mapping against a schema) may not terminate on recursive types. For these reasons we believe that discarding vertical recursive types is not restrictive in the study of schema mapping languages.

The semantics of types is standard: as usual, $[\cdot]$ is the minimal function from types to sets of forests that satisfies the following monotone equations:

\[
\begin{align*}
[()] & \triangleq \{()\} \\
[B] & \triangleq \{b \mid b \text{ is a base value}\} \\
[l[T]] & \triangleq \{l[f] \mid f \in [T]\} \\
[T_1 | T_2] & \triangleq [T_1] \cup [T_2] \\
[T_1, T_2] & \triangleq \{f_1, f_2 \mid f_i \in [T_i]\} \\
[T^*] & \triangleq [T]^* 
\end{align*}
\]

Subtyping, which is used in Section 4 to prove decidability of type projection, is defined via type semantics, as shown below.

**Definition 3.1 Semantic subtyping.** Given two types $T$ and $U$, $T$ is a subtype of $U$ if and only if the semantics of $T$ is contained into the semantics of $U$:

\[T < U \iff [T] \subseteq [U]\]

The following lemma describes an interesting property of unordered, non-recursive regular expression types, that will be used later on in the paper.

**Lemma 3.2. Given two types $T$ and $U$:**

\[[T^*, U^*] = [[T | U]^*]\]

We model a schema as a set of type equations of the form $X = T$, where $X$ is a type variable and $T$ is a type defined according to our type language.
3.3 Schema Mappings

In this section we introduce and formalize our notion of schema mappings. In our vision, a schema mapping is the specification of a transformation from a source schema $S$ to a target schema $T$.

**Definition 3.3 Mapping.** Let $S$ be a source schema and let $T$ be a target schema. A schema mapping $m$ from $S$ to $T$ is an assertion from $S$ to $T$ of the form $(Q, \{q_i\})$, where $q_i$ is a query from $S$ to $T$, and $Q$ is an outer query referring each $q_i$.

The previous definition states that a mapping is composed by a primary query $(Q)$, which defines the overall structure of the mapping, and by a set of secondary assertions $\{q_i\}$, which correlate fragments of $S$ with fragments of $T$. The intuition behind this distinction is to provide more modularity to a schema mapping by separating the way assertions are assembled from their specification.

In our specific setting the primary query of a mapping $m$ is a $\mu$XQ query. Secondary assertions are modeled as $\mu$XQ queries too. For instance, in Example 2.1 $Q_2$ is the primary query, while $Q_1$ is a secondary assertion.

For the sake of simplicity, we will denote the combination of a primary query and a set of secondary assertions as $Q[\{q_i\}]$. For the same reason, we will often refer to secondary assertions just as secondary queries.

By the previous definition, a schema mapping is a specification of the actual transformations between source data and target data. Indeed, both the primary query and secondary assertions can be incomplete, in the sense that they do not cover all the elements of the target schema. Of course, our mappings can be enriched and modified to represent a complete transformation by using, for instance, a chasing strategy. We prefer to focus on mappings as a specification tool because this notion captures the essence of schema mappings; furthermore, once a mapping has been deemed as correct, the actual transformation can be easily and automatically generated by existing tools.

3.4 Correctness of Schema Mappings

In this section we will introduce the notion of mapping validity and mapping correctness. Validity is characterized by the following definition.

**Definition 3.4 Mapping validity.** A mapping $m = (Q, \{q_i\})$ from $S$ to $T$ is valid if and only if the combination of the primary query and the secondary queries is correct wrt $S$, in the sense that, for each non-empty subquery $q$ of $Q[\{q_i\}]$, there exists a data instance $d$ of $S$ such that, when evaluated on $d$, $q$ will return a non-empty result.

**Example 3.5.** Consider the schemas and the mapping of Example 2.1. This mapping is valid wrt the source schema, as each subquery (path expression, in particular) returns no empty results for some valid input.

Assume now that the source schema is modified as follows:

```
Song = song[EnglishTitle, Artist, Album, MyRating]
EnglishTitle = englishTitle[String]
```

The mapping now becomes invalid wrt the new source schema. Indeed, assertion $Q_1$ contains a nested query accessing title elements, which are no longer present in the schema.
Mapping validity implies that a valid mapping must be correct wrt the source schema, i.e., it must match the structure of the source schema. We adopt the query correctness notion described in [Colazzo 2004; Colazzo et al. 2004] and [Colazzo and Sartiani 2005]. Mapping validity allows for identifying mappings that are obsolete, i.e., that contain rules referring to fragments of the source schema that have been changed or deleted. From now on, we will assume that each mapping is valid, and focus on mapping correctness, and therefore on the detection of errors wrt the target schema.

Our notion of mapping correctness is based on the following notion of data projection. Intuitively, $f_1$ is a projection of $f_2$, noted as $f_1 \lesssim f_2$, if there exists a subterm $f_3$ in $f_2$ such that $f_3$ matches $f_1$; this is very close (up to simulation) to the relational projection, where $r_1 = \pi_{A}\!r_2$ if $r_1$ is equal to the fragment of $r_2$ obtained by discarding non-$A$ attributes.

**Definition 3.6 Value projection.** The value projection relation $\lesssim$ is the minimal relation such that:

- $(\varepsilon) \lesssim f$
- $b_1 \lesssim b_2$ if $f_1 \lesssim f_2$
- $f_1 \lesssim f_2$
- $l[f_1] \lesssim l[f_2]$ if $f_1 \lesssim f_2$

Note that in data projection we require structural matching, while exact matching on leaf base values $b$ is not required.

**Definition 3.7 Mapping correctness.** A mapping $m = (Q; \{q_i\}_i)$ from $S$ to $T$ is correct if and only if, for each $S$ data instance $d_S$ there exists a $T$ data instance $d_T$ such that $Q[\{q_i\}_i](d_S) \lesssim d_T$.

The above definitions state that a mapping from $S$ to $T$ is correct if and only the result of $Q[\{q_i\}_i]$ on a value conforming to $S$ is mapped, according to the $\lesssim$ relation, into a value conforming to $T$. $\lesssim$ is an injective simulation relation among values, inspired by the projection operator of the relation data model.

**Example 3.8.** Consider again the schemas and the mapping of Example 2.1. This mapping returns values of the following type:

```
OutputType = musicDB[A*]
A = artist[N, T1*]
N = name[String]
T1 = track[T2]
T2 = title[String]
```

As it can be easily seen, each value of this type is actually a projection of a value conforming to the target schema, hence the mapping is correct.

Our notion of mapping correctness relies on the comparison between the semantics of the mapping, i.e., the set of its results when applied to instances of the source schema, and the semantics of the target schema. In this sense, we can say that our notion is semantic, as it only depends on the semantics of the source and target schemas, as well as of the mapping. This does not imply that our notion is able to

---

5 Validity can be checked by using the algorithms proposed in [Colazzo 2004][Colazzo et al. 2004][Colazzo and Sartiani 2005]; these algorithms are polynomial in most practical cases.

6 Our notion of validity is different from the notion of semantically valid mappings of [Velegrakis et al. 2004]. Indeed, in our framework, validity applies only to the source schema.
capture the intended semantics of a mapping: this problem, indeed, is AI-complete and cannot be completely solved by an automatic tool.

Before concluding this section, some final remarks are needed. The notion of XML projection we adopt is a generalization of that introduced by Marian et al. in [Marian and Siméon 2003], where leaf values are taken into account too. Also, our notion of correctness is independent from the mapping specification language, since it is defined in terms of query (mapping) outputs, hence it is applicable in other data integration scenarios where mappings are inferred by semi-automatic tools; for instance, our approach can be easily applied to mappings described in terms of source-to-target dependencies. More generally, our approach can be applied to any mapping language for which suitable notions of type inference and type projection can be defined.  

4. CORRECTNESS CHECKING

Definitions 3.7 and 3.6 describe our notion of mapping correctness, but they cannot directly be used to check whether a mapping is correct or not. To obtain a constructive definition, we need to switch from values to types.

Definition 4.1 Type projection. Given two types $T_1$ and $T_2$, we say that $T_1$ is a projection of $T_2$ ($T_1 \preceq T_2$) if and only if: $\forall d_1 : T_1 \exists d_2 : T_2. d_1 \preceq d_2$.

As for the value projection relation, the type projection relation is semantic, and states that a type $T_1$ is a projection of a type $T_2$ if, for each data instance $d_1$ conforming to $T_1$, there exists a data instance $d_2$ conforming to $T_2$ such that $d_1$ is a projection of $d_2$.

Type projection is quite different from standard subtyping, since it is based on the idea that $T_1 \preceq T_2$ if $T_1$ matches a fragment of $T_2$, while $T_1 < T_2$ implies that $T_1$ is more specific than $T_2$.

Example 4.2. Consider the target schema of Example 2.1. Consider now the following type:

```
Tiny MDB = musicDB[Artist*]
Artist = artist[Name, Provenance]
Name = name[String]
Provenance = provenance[Continent, Country]
Continent = continent[String]
Country = country[String]
```

This type is actually a projection of SeattleMDB, as each TinyMDB instance is a projection of a SeattleMDB instance. However, TinyMDB is not a subtype of SeattleMDB, as TinyMDB instances lack track elements, which are mandatory for SeattleMDB instances.

To use type projection in mapping correctness checking, we must correlate type projection and mapping correctness. To this aim, if we assume that for each query we can infer a type containing all the query results, we can use the inferred type to check mapping correctness, as indicated in the following proposition.

Proposition 4.3 Mapping correctness via type projection. If $m = (Q, \{q_i\}_i)$ is a schema mapping from $S$ to $T$, and $U$ an upper-bound for $Q_{\{q_k\}_k}$ (for each $f \in [S]$, $Q_{\{q_k\}_k}(f) \in [U]$), then $m$ is correct if $U \preceq T$.

\footnote{We stress that we see a mapping as a specification rather than an actual transformation. Hence, our claim of language independence applies to mapping specification languages rather than mapping implementation languages.}
As we will see in Section 9, quite precise query upper bound types can be systematically inferred by means of a type-inference algorithm able to prove judgments of the form $\Gamma \vdash Q : U$, where $\Gamma$ is an environment containing information about the source schema $S$, and $U$ is the inferred upper bound type for the mapping $Q$.

**Example 4.4.** Referring to Example 2.1, the output type of the mapping is the following:

```
OutputType = musicDB[A*]
A = artist[N, T1*]
N = name[String]
T1 = track[T2]
T2 = title[String]
```

This type is a projection of the target schema, hence the mapping can be deemed as correct (as we already know from Example 3.8).

5. SCHEMA CHANGES AND MAPPING CORRECTNESS

In the previous sections we presented a notion of mapping correctness that addresses changes in the structure of a schema. As there are several kinds of updates that can be applied to a schema, it is worth to explore the various forms of schema changes, so to understand to what extent our notion is effective.

In its most common interpretation, a schema consists of a type, describing the structure of the instances of the schema, and of a set of constraints over data instances. As a consequence, a schema change may affect the type, the set of constraints, or both.

In our work we focus our attention on the type component of a schema, hence any change in the set of constraints is not supported. This choice is motivated by the fact that, as previously said, we assume that data sources are autonomous, hence it is unlikely that a data source makes constraints externally visible.

We can consider five main kinds of structural changes that can be applied to a schema: the removal of existing type definitions (e.g. the removal of an element type); the change of a datatype inside an element content type (e.g., the switch from `String` to `Int`), the relocation of a fragment of a schema to a new location; the renaming of an element type; and the insertion of new type definitions inside the schema. In the following paragraphs we will explore the applicability of our approach to these kinds of updates on both the source and the target schema.

**Source schema.** Changes in the source schema of a mapping may affect its validity. When a type definition is removed from the source schema, the validity of a mapping is affected only if the definition was used and referred in the mapping (we assume, of course, that the new schema is well-formed). For instance, assume that the definition $\text{Artist} = \text{artist}[\text{String}]$ is removed from the source schema (CupMDB) of Example 2.1. The mapping becomes invalid, as it tries to access a no longer existing fragment of the schema. The query correctness notion we described in [Colazzo 2004; Colazzo et al. 2004] can easily capture all errors implied by a type definition removal.

The same considerations apply to the relocation of a fragment of the schema to a new location, and to the renaming of element type definitions. These changes may induce errors in a mapping (remember that we see a mapping as a specification, hence it can be incomplete) only if they affect the portion of the schema that is visited by the mapping; again, the technique we developed in [Colazzo et al. 2004] is able to identify and notify all errors induced by these schema updates.
Example 5.1. Consider the schemas and the mapping of Example 2.1 and assume that the source schema is modified as follows.

\[
\text{CupMDB} = (\text{Song})*
\]

\[
\begin{align*}
\text{Name} &= \text{name}[\text{String}] \\
\text{Artist} &= \text{artist}[\text{String}] \\
\text{Album} &= \text{album}[\text{String}] \\
\text{MyRating} &= \text{myRating}[\text{Integer}]
\end{align*}
\]

The new source schema contains a \text{name} element in place of the old \text{title} element (element renaming); furthermore, the collection of \text{song} elements has been relocated to the outermost position in the schema (type relocation). Both changes make the mapping no longer valid and induce errors that are easily identified by the approach we described in [Colazzo. et al. 2006], since the mapping tries to access schema fragments no longer existing.

The switch from a datatype to another one is not directly supported in the approach we described in [Colazzo. et al. 2006], as our type system uses a single base type. However, it can be easily seen that an extension to multiple datatypes is trivial and that all corruptions induced by this kind of changes can be identified.

The enrichment of the source schema with new type definitions never alter the validity of a mapping. Indeed, all type definitions accessed by the mapping are still present, so there is no error that a type-checking algorithm can detect. For instance, if we modify the source schema of Example 2.1 as follows:

\[
\begin{align*}
\text{Song} &= \text{song}[\text{Title}, \text{Artist}, \text{Album}, \text{MyRating}, \text{ChartPosition}] \\
\text{ChartPosition} &= \text{chartPosition}[\text{String}]
\end{align*}
\]

all type definitions used by the mapping are still accessible.

Target schema. The considerations we did for changes in the source schema apply also to the changes on the target schema. In particular, all corruptions induced by the removal of a type definitions, by the renaming of an element type definition, and by the relocation of a fragment of the schema are identified and notified. Furthermore, value projection and type projection can be easily extended to support multiple datatypes.

Example 5.2. Consider again the schemas and the mapping of Example 2.1. Assume that the target schema is modified so that the information about music tracks is removed, as shown below.

\[
\begin{align*}
\text{SeattleMDB} &= \text{musicDB}[\text{Artist}*] \\
\text{Artist} &= \text{artist}[\text{Name}, \text{Provenance}] \\
\text{Name} &= \text{name}[\text{String}] \\
\text{Provenance} &= \text{provenance}[\text{Continent}, \text{Country}] \\
\text{Continent} &= \text{continent}[\text{String}] \\
\text{Country} &= \text{country}[\text{String}]
\end{align*}
\]

The mapping generates \text{track} elements, which are no longer defined in the new target schema. The mapping, hence, is now incorrect and, as expected, the error is detected by our approach (a type projection check would return \text{false}).

Assume now that the original target schema of Example 2.1 is modified so that \text{title} elements are renamed as \text{songTitle} elements. As in the previous case, the mapping generates \text{title} elements that cannot be matched by any element in the new target schema; the mapping corruption is successfully detected by our approach.

Finally, if we assume that the collection of \text{track} elements is relocated under a \text{tracks} element, we can see that, as in the previous cases, the mapping is no longer able to match the new target schema and is deemed as incorrect.
As for mapping validity, mapping correctness is not affected by the insertions of new type definitions in the target schema. This is fully reasonable, as our mappings may be incomplete.

As we have seen, our approach is able to capture all errors induced by a structural update on the target schema. However, when a structural update is coupled with a modification in the intended semantics of the schema, things change. Referring to our previous example, assume that SeattleMDB is modified as follows:

\[
\text{Artist} = \text{artist}[\text{Name}, \text{WName}, \text{Provenance}, \text{Track}^*] \\
\text{WName} = \text{wName}[\text{String}] 
\]

where \text{WName} models the working name of an artist and \text{Name} its actual name. Observe that CupMDB does not distinguish between the actual and the working name of an artist, so all CupMDB name elements are interpreted as working names.

By adding \text{WName}, we do not violate the correctness notion of the previous section. However, the intended semantics of the schema has now been modified, as \text{Name} now represents actual names. Such a change makes the mapping no more adequate, as it maps working names from CupMDB into SeattleMDB actual names.

Any change in the intended semantics of a schema can make the mapping no more adequate. However, checking that a mapping is “semantically” adequate to its source and target schema cannot be automatically performed by a maintenance tool, as this problem is definitely AI-complete. It should be observed that even detecting a change in the intended semantics of the schema of an autonomous data source is, in the general case, not feasible and requires the intervention of a developer.

Dealing with this kind of corruptions, hence, it is definitely not easy. However, our approach can be easily extended to provide useful support to the developer. Indeed, type projection checking can be used to identify the fragments of the target schema that are covered by the mapping; in the same way, validity checking relates the mapping queries with the fragments of the source schema visited by the mapping. By exploiting this information, we can analyze the behavior of the mapping on both the source and the target schema (a preliminary implementation of this approach can be found in the current version of our maintenance tool, available at http://www.di.unipi.it/~sartiani/projects/gamma.html).

By observing that a change in the intended semantics of a schema is more likely to corrupt a mapping if it affects a fragment that is close to those touched by the mapping, we can notify to the developer any schema change that is sufficiently close (according to some proper metric) to the portions of the schema involved in the mapping. As an example, in the case of working names and actual names, this extension will pinpoint the schema change as potentially harmful, as it is very close to name elements, which are touched (covered) by the mapping. Of course, no formal properties can be stated and proved for this extension; however, it provides useful information at a very low extra cost.

6. DECIDABILITY AND COMPLEXITY OF TYPE PROJECTION

6.1 Decidability

As we have seen in the previous sections, and in Proposition 4.3 in particular, if one can establish a projection relation between the inferred type and the target schema of a mapping, the correctness of the mapping is proved. In order to move towards a practical correctness checking technique, we first need to prove decidability of the type-projection relation.
To prove that type projection is decidable we rely on a particular notion of type approximation. Type approximation weakens types by enriching base and element types with a union with the empty sequence type; this allows one to relate type projection to standard subtyping for commutative types, whose decidability has been proved by Huynh in [Huynh 1985].

Type approximation is based on the idea of weakening types by introducing unions with the empty sequence type.

**Definition 6.1** Type approximation. Given a type $U$, we indicate with $U^\triangle$ the type obtained by just by replacing each subexpression $U'$, corresponding to a tree type $l[\_]$ or $B$, with $U'\rightarrow$ (that is $(U'\mid\_)$). Formally:

- $()^\triangle = ()$
- $T \mid U^\triangle \triangleq T^\triangle | U^\triangle$
- $l[T]^\triangle \triangleq l[T^\triangle]$
- $B^\triangle \triangleq B\rightarrow$
- $T, U^\triangle \triangleq T^\triangle, U^\triangle$
- $T^\star \triangle \triangleq T^\triangle$

It is easy to prove that $T < T^\triangle$. To prove the main results about type approximation, we have to introduce the notion of contexts, whose grammar is shown below.

**Contexts** $C ::= x \mid () \mid C, C \mid l[C] \mid b$

A context is a partially specified forest, where variables indicate arbitrary forests. Variables are always assumed to be unique, and context instantiation is indicated as $C$, where $\rho$ is a set of variable assignments $x \mapsto \rho$. We indicate with $C()$ the forest obtained by $C$ by replacing each variable with the empty sequence.

If we indicate with $f \simeq f'$ the fact that the two forests are equal up to leaf values, we can state the following lemma.

**Lemma 6.2.** Given two forests $f_1$ and $f_2$, the following relation holds:

$$f_1 \preceq f_2 \iff \exists C, \exists \rho. f_1 \simeq C() \land f_2 \simeq C\rho$$

The following theorem correlates $T^\triangle$ with $T$.

**Theorem 6.3.** For each type $T$ we have $T^\triangle \preceq T$.

**Proof.** See the Appendix.

The following lemma states that the subtype relation includes the projection relation. We omit the proof as it is immediate.

**Lemma 6.4.** For each type $T$ and $U$ we have $T < U \Rightarrow T^\triangle < U^\triangle$

In order to prove the projection-as-subtyping characterization, we need the following lemma.

**Lemma 6.5.** For each type $T$:

1. $\forall f : U^\triangle. (f \neq \bot) \Rightarrow \exists C, \rho. f' : U. C() \simeq f \land f' \simeq C\rho$
2. $\forall f : U, C, \rho. (f \simeq C\rho \Rightarrow C() : U^\triangle)$
3. $\forall C. (C() \neq \bot) \land C() : U^\triangle \Rightarrow \exists f : U. \exists \rho. f \simeq C\rho$

**Proof.** Property 1 follows by Lemma 6.2 and Theorem 6.3. Property 2 follows by a simple induction on $U$. Property 3 follows from Property 1.

We can now provide one of the main theorems of this work.
Theorem 6.6 Type projection as sub-typing.

\[ T \triangleleft U \iff T < U^\triangleleft \]

Proof. \(\Rightarrow\)

By hypothesis we have

\[ \forall f : T. \exists f' : U. f \triangleleft f' \]

By Lemma 6.2 this implies that

\[ \exists C, \exists \rho. C_1 \simeq f \land f' \simeq C_\rho \]

and by \(f' : U, f' \simeq C_\rho\) and Lemma 6.5 (Property 2) we can conclude that \(C_1 : U^\triangleleft\), that is \(f : U^\triangleleft\), as \(C_1 \simeq f\).

\(\Leftarrow\)

We have that for each \(f:\)

\[ f : T \Rightarrow f : U^\triangleleft \]

We distinguish the case \(f = ()\), which trivially follows since \(() : U^\triangleleft\) for each \(U^\triangleleft\). When \(f \neq ()\), from \(f : U^\triangleleft\) and Lemma 6.5 (Property 1) we have that there exists a context \(C\) such that \(f = C_1\), and this means that, by Lemma 6.5 (Property 3), there exists \(f' : U\) and \(\rho\) such that \(f' = C_\rho\).

This means that \(f \triangleleft f'\) (Lemma 6.2), and the case is proved. \(\Box\)

The previous theorem states that type projection between \(T\) and \(U\) can be checked by weakening \(U\) and, then, by checking for the existence of a subtyping relation between \(T\) and \(U^\triangleleft\). This result proves decidability of type projection, since Huynh proved that inclusion for commutative regular grammars is \(\Pi_2^P\)-hard and is in \(\text{CoNEXPTIME}\) [Huynh 1985]. This result also identifies an upper bound for the complexity of type projection; however, in the following we will show that a better upper bound for complexity of projection can be found by relying on a type simulation approach.

The previous theorem, stating that type projection is equivalent to subtyping, once the right hand-side of the comparison has been approximated, gives rise to a fundamental question, i.e., whether type projection can be replaced, in the context of data integration and data exchange, by subtyping. A strong reason for discarding subtyping, in favor of type projection, is its algorithmic complexity.

So in order to efficiently check type projection, we propose in the following an alternative characterization of type projection which is not based on subtyping and which is a first step towards an efficient algorithm.

6.2 Complexity of Type Projection

Before illustrating our technique for checking type projection, it is worth to analyze the computational complexity of type projection. From Theorem 6.6 we know that type projection is equivalent to subtype-checking when the right hand-side of the comparison has been weakened, i.e., \(T \triangleleft U \iff T < U^\triangleleft\). Since \(U\) can be transformed into \(U^\triangleleft\) in polynomial time and space, this theorem also states that type projection cannot be more expensive than subtype-checking. Inclusion among commutative type is known to be in \(\text{CoNEXPTIME}\) [Huynh 1985], hence \(\text{CoNEXPTIME}\) is an upper bound for type projection too. In Sections 7.1 and 8 we will see that this upper bound can be refined to \(\text{EXPTIME}\).

For what concerns the complexity lower bound, we first introduce a supplementary operation called \(\triangleleft\)-membership.
Definition 6.7 \( \preceq \)-membership. Given a data model instance \( f \) and a type \( T \), we say that \( f \preceq_{\text{mem}} T \) if and only if \( \exists f' \in [T].f \preceq f' \).

The relation \( \preceq_{\text{mem}} \) is here called \( \preceq \)-membership as it is the counterpart of membership for inclusion and equivalence problems. It should be observed that \( f \preceq f' \) can be decided in polynomial time in the size of \( f \) and \( f' \).

The following theorem shows that \( \preceq \)-membership is NP-complete.

Theorem 6.8. \( \preceq \)-membership is NP-complete.

Proof. (Lower Bound)
To prove the NP-hardness of \( \preceq \)-membership, we show a reduction from the NP-complete Minimum Cover problem, described as follows.

Given a finite set \( S \), a collection \( C = \{C_1, \ldots, C_m\} \) of subsets of \( S \), and a positive integer \( k \leq m \), does \( C \) contain a cover for \( S \) comprising at most \( k \) subsets, i.e., a collection \( D = \{D_1, \ldots, D_t\} \), where \( t \leq k \), each \( D_i \) is a set in \( C \), and such that every element in \( S \) belongs to at least one set in \( D \)?

Without loss of generality, we can assume that \( S = \{s_1, \ldots, s_n\} \). Let \( d \) be a symbol not in \( S \): \( d \not\in S \). We map \( S \) in the following data model instance:

\[
\pi = s_1[b], \ldots, s_n[b], d^{m-k}[b]
\]

Assuming \( C_i = \{c_{i1}, \ldots, c_{ip}\} \), for each \( C_i \) we build the following type:

\[
T_i = (c_{i1}[B], \ldots, c_{ip}[B]) \mid d[B]
\]

Finally, we create a type \( T = T_1, \ldots, T_m \).
If \( \pi \preceq_{\text{mem}} T \), then \( S \) is covered by \( C \) and at least \( m - k \) \( T_i \)'s have given no contribution to the cover. It is easy to see that, if \( C \) contains a minimum cover for \( S \), then \( \pi \preceq_{\text{mem}} T \).

(Upper Bound)
To prove that \( \preceq \)-membership is in NP, it suffices to observe that \( x \preceq_{\text{mem}} T \iff x \in T^\preceq \) (by Theorem 6.6). As membership for regular expression types with commutative product has been proved to be in NP [Mayer and Stockmeyer 1994], we can conclude that \( \preceq \)-membership is in NP too.

□

The complexity of \( \preceq \)-membership provides a lower bound for the complexity of type projection, as shown by the following corollary.

Corollary 6.9. Type projection is NP-hard.

Proof. It is easy to see that \( x \preceq_{\text{mem}} T \) if and only if \( T_x \preceq T \), where \( T_x \) is the type containing only \( x \) in its semantics (the construction of \( T_x \) is trivial). Hence, type projection cannot be easier than \( \preceq \)-membership, which proves the thesis. □

7. TYPE SIMULATION
This section introduces type simulation, discusses its main properties, and shows its equivalence to the type projection relation we described in Section 4.
7.1 Definition

Type simulation is a symbolic relation among types, whose main aim is to provide a convenient way to characterize and check for type projection. Indeed, Section 4 showed the “semantic” nature of type projection, and, even though it proved projection decidability, still it did not provide an efficient way to check for projection among XML types.

Type simulation is defined among types in disjunctive normal form, i.e., types where products are distributed across unions. A type $T$ can be normalized by applying the normalization function $\text{norm}(T)$, defined as shown in Table 7.1. The following lemma proves that the evaluation of $\text{norm}(T)$ always terminates.

**Lemma 7.1 Termination of $\text{norm}(T)$.** For each type $T$, $\text{norm}(T)$ is computed in finite time.

**Proof.** See the Appendix.

$\text{norm}()$ works by transforming types, while preserving their semantics (Lemma 7.5), so that the transformed types can be easily compared by the simulation relation (and by the corresponding algorithm). For instance, $\text{norm}(T'*, U'^*, U)$ transforms a product of repetition types, which is hard to formalize in the simulation rules, into a $*$-guarded union, for which much easier simulation rules exist (correctness of this transformation is entailed by Lemma 3.2).

To eliminate some ambiguity, the rules of the $\text{norm}()$ must be applied in the order in which they are defined. $\text{norm}()$ can be applied to any type, and its relevance resides in the proof of equivalence between simulation and projection, as it will be clear in the rest of the paper.

The core of normalization is the transformation of type expressions in conjunctive normal form into equivalent ones in disjunctive normal form (see the third and the sixth rule in Table 7.1). As a consequence, $\text{norm}()$ has an EXPTIME worst case time complexity, and the normalized type may have an exponential size wrt the original type. Despite this, for a vast class of types $\text{norm}()$ can be computed in PTIME. This class contains types where unions are always guarded by a $*$-operator ($*$-guarded types), as shown by the following definition.

**Definition 7.2 SGT.** A type $T$ is in SGT (star-guarded types) if it can be gen-

<table>
<thead>
<tr>
<th>$\text{norm}()$</th>
<th>$\triangleq$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{norm}([])$</td>
<td>$()$</td>
</tr>
<tr>
<td>$\text{norm}(B)$</td>
<td>$B$</td>
</tr>
<tr>
<td>$\text{norm}(T)$</td>
<td>$\begin{cases} \bigcup \cdots [A_i] &amp; \text{if } \text{norm}(T) = A_1 \mid \cdots \mid A_n \ U[\text{norm}(T)] &amp; \text{otherwise} \end{cases}$</td>
</tr>
<tr>
<td>$\text{norm}(T \mid U)$</td>
<td>$\text{norm}(T) \mid \text{norm}(U)$</td>
</tr>
<tr>
<td>$\text{norm}(T'<em>, U'^</em>, U)$</td>
<td>$\text{norm}(T' \mid U'^* \mid U)$</td>
</tr>
<tr>
<td>$\text{norm}(T, A)$</td>
<td>$\begin{cases} \text{norm}(T, A_1) \mid \text{norm}(T, A_2) &amp; \text{if } \text{norm}(U) = (A_1 \mid A_2) \ \text{norm}(T), \text{norm}(U) &amp; \text{otherwise} \end{cases}$</td>
</tr>
<tr>
<td>$\text{norm}(T*)$</td>
<td>$\text{norm}(T)*$</td>
</tr>
</tbody>
</table>
erated by the following grammar:

\[
\begin{align*}
\text{\texttt{*-Types}} & : \quad T ::= () \mid B \mid l[T] \mid T \cdot T \mid U^* \\
\text{\texttt{Union Types}} & : \quad U ::= T \mid T \mid U
\end{align*}
\]

Proving that for \texttt{-guarded} types \texttt{norm()} is polynomial is straightforward. \texttt{-guardedness} is a property enjoyed by a large number of commonly used DTDs and XSDs. For instance, the reader can refer to [Choi 2002] and to [Bex et al. 2004] for a detailed classification of real world DTDs: this classification shows that non-\texttt{-guarded} unions are quite infrequent. In any case, in order for \texttt{norm()} to blow up, the \texttt{-guarded} union restriction must be systematically violated, so a few occurrences of \texttt{-unguarded} are harmless.

It is worth to notice that optional types of the form \( A \mid () \), even though representing a relatively frequent kind of non-\texttt{-guarded}ness, does not affect the complexity of \texttt{norm()} since they can be rewritten into \( A \) by preserving projection. We prove this fact in Lemma 7.4, where the function \texttt{DropOpt()} is used:

\[
\begin{align*}
\texttt{DropOpt}(() & \triangleq () \\
\texttt{DropOpt}(T^*) & \triangleq \texttt{DropOpt}(T)^* \\
\texttt{DropOpt}([T]) & \triangleq \texttt{DropOpt}(T) \\
\texttt{DropOpt}(B) & \triangleq B \\
\texttt{DropOpt}(T, U) & \triangleq \texttt{DropOpt}(T), \texttt{DropOpt}(U)
\end{align*}
\]

The following property is immediate, so we omit its proof.

**Lemma 7.3.** For each type \( T \): \texttt{DropOpt}(\( T \)) \( \prec \) \( T \) and \( T \ prec \texttt{DropOpt}(T) \).

From the above two facts the following lemma follows.

**Lemma 7.4.** For each \( T \) and \( U \):

\[ T \prec U \Leftrightarrow \texttt{DropOpt}(T) \prec \texttt{DropOpt}(U) \]

**Proof.** In the whole proof we will repeatedly use facts stated in Lemma 7.3.

\((\Rightarrow)\) For each \( f : \texttt{DropOpt}(T) \) we have \( f : T \) as well (by \( \texttt{DropOpt}(T) \prec T \)). Then, by the hypothesis \( T \prec U \) we have \( \exists f' : U. f \prec f' \). Now, since \( f' : U \) implies \( \exists f'' : \texttt{DropOpt}(U) \) such that \( f' \prec f'' \) (by \( U \prec \texttt{DropOpt}(U) \)) and since \( \prec \) is transitive, we can conclude \( f \prec f'' \).

\((\Leftarrow)\) For each \( f : T \) we have \( \exists f' : \texttt{DropOpt}(T) \). \( f \prec f' \) (by \( T \prec \texttt{DropOpt}(T) \)). Since \( \texttt{DropOpt}(T) \prec \texttt{DropOpt}(U) \) we have that \( \exists f'' : \texttt{DropOpt}(U) \). \( f' \prec f'' \). Hence, by transitivity of \( \prec \) and \( \texttt{DropOpt}(U) \prec U \) the case is proved.

So, from now on we can make the assumption that types do not contain optional types \( A \mid () \). This is assumption is non-restrictive since, as suggested by Lemma 7.4, they can be safely eliminated.

The following lemma proves that \texttt{norm()} preserves the semantics of types.

**Lemma 7.5.** For each type \( T \), \( \llbracket T \rrbracket = \llbracket \texttt{norm}(T) \rrbracket \).

**Definition 7.6.** A type \( T \) is prime if and only if \( \texttt{norm}(T) = T \) and \( T \neq A \mid B \).

Prime types will play a crucial role in the following. Since prime types are invariant under normalization and they cannot be union types, their semantics never contain mutually exclusive tree structures. For this reason, as we will see, a prime type can be considered as a whole during projection checking, without the
need of any kind of further transformation. This is ensured by the following lemma, formalizing the main property enjoyed by prime types.

**Lemma 7.7 Upward closure.** If $T$ is prime, then $\forall f_1, f_2 \in [T], \exists f \in [T]. f_i \preccurlyeq f$

**Proof.** See the Appendix.

In the rest of the paper, we will need the following lemma that deals with projection among $*$-types. Essentially, as far as prime types are concerned, this lemma states that a type $T^*$ is in the projection relation only wrt to types $U$ containing a $*$-type at the top level, that is $U = U_1^*, A$ with $A$ not containing $*$-types at the top level; moreover, only the $*$-type contributes to the projection, the proof being based on the cardinality of sequences.°

**Lemma 7.8.** If $T^*$ and $U$ are prime, then $T^* \preccurlyeq U \iff U = (U_1)^*, A$ and $T^* \preccurlyeq (U_1)^*$.

We can now state the definition of type simulation.

**Definition 7.9 Type simulation.** The type simulation relation $\preccurlyeq$ among normalized types is defined as follows.

1. $B \preccurlyeq B$
2. $(\emptyset) \preccurlyeq U$
3. $l[T] \preccurlyeq l[U]$ if $T \preccurlyeq U$
4. $T_1 \preccurlyeq U_2, U_3$ if $T_1 \preccurlyeq U_2 \lor T_1 \preccurlyeq U_3$
5. $T_1, T_2 \preccurlyeq U_3, U_4$ if $(T_1 \preccurlyeq U_3 \land T_2 \preccurlyeq U_4)$
6. $T_1 \preccurlyeq U_2 \vert U_3$ if $T_1 \preccurlyeq U_2 \lor T_1 \preccurlyeq U_3$ and $T_1 \neq V_1 \vert V_2$
7. $T_1 \mid T_2 \preccurlyeq U$ if $T_1 \preccurlyeq U \land T_2 \preccurlyeq U$
8. $T \preccurlyeq U^*$ if $T \preccurlyeq U$
9. $T^* \preccurlyeq U^*$ if $T \preccurlyeq U^*$
10. $T_1, T_2 \preccurlyeq U^*$ if $T_1 \preccurlyeq U^* \land T_2 \preccurlyeq U^*$

Rules 1-3 are straightforward as well as Rule 8. Rules 4-5 describe the simulation among product types, while Rules 6-7 illustrate the simulation among union types. Rules 8-10, finally, are dedicated to repetition types.

Rules for product types are of special interest. In particular, Rule 5 shows that simulation between product types is injective, hence capturing the injective nature of projection: for instance, $T = Album, Album$ cannot be projected into $U = Album$, as data conforming to $T$ have two distinct album elements, while data conforming to $U$ have only one album element. Injectivity may be broken by repetition types, or when sequence types are in the immediate scope of a repetition type.

Rules 6-7 describe the simulation for union types. These rules pinpoint the commutative and non-injective nature of union types.

The following lemmas are necessary to prove completeness of the above characterization wrt projection; they also show the behavior of the projection relation on normalized types.

**Lemma 7.10 Accumulation.** If $T \preccurlyeq T_1 \mid T_2$ and $T$ is prime, then $T \preccurlyeq T_1$ or $T \preccurlyeq T_2$.

**Proof.** By aiming to a contradiction, suppose that the thesis does not hold. Under this assumption, if we define

°Recall that each prime type can have at most one $*$-type at the top level.
we have

\[ P(U, C) = \{ f : U \mid \exists f' : C. f \preceq f' \} \]

This means that there exist two different forests \( f_1 \) and \( f_2 \) such that \( f_1 \in P(T, T_1) \setminus P(T, T_2) \) and \( f_2 \in P(T, T_2) \setminus P(T, T_1) \). By the previous inclusions and Lemma 7.7 we have that there exists \( f : T \) such that \( f_1 \preceq f \) and \( f_2 \preceq f \). Now, either \( f \in P(T, T_1) \) or \( f \in P(T, T_2) \), but in both cases we have a contradiction, as by transitivity of \( \preceq \) and by definition of \( P(-,-) \), at least one of the two derived properties \( f_1 \in P(T, T_1) \setminus P(T, T_2) \) and \( f_2 \in P(T, T_2) \setminus P(T, T_1) \) is contradicted. For instance, assume that \( f \in P(T, T_1) \), that is: there exists \( f'' : T_1 \) such that \( f \preceq f'' \). According to what obtained before, we have \( f_2 \preceq f'' \), that is \( f_2 \in P(T, T_1) \), and, as a consequence, \( f_2 \notin P(T, T_2) \setminus P(T, T_1) \), which is a contradiction. □

A similar lemma (Lemma 7.12) deals with the product of prime types; to prove this lemma, the following corollary is necessary.

**Corollary 7.11.** If \( T \) is a prime tree type such that \( T \preceq U_1, U_2 \), then \( T \preceq U_1 \) or \( T \preceq U_2 \).

**Proof.** It suffices to notice that since \( T \) is a prime tree type, we have

\[ T \preceq U_1, U_2 \Leftrightarrow T \preceq U_1 \cup U_2 \]

and then apply Lemma 7.10. □

**Lemma 7.12.** If \( T \) and \( U \) are normalized and prime, if both \( T \) and \( U \) are product types, and \( T \preceq U \), then \( T = T_1, T_2 \) and \( U = U_1, U_2 \) with

\[ (T_1 \preceq U_1 \text{ and } T_2 \preceq U_2) \text{ or } T \preceq U_1 \]

**Proof.** See the Appendix. □

### 7.2 Equivalence Between Type Projection and Type Simulation

The proof of equivalence consists of two distinct implications (**projection-to-simulation** and **simulation-to-projection**). Both the implications are proved by induction on the structure of types with the help of Lemmas 3.2, 7.8, 7.7, 7.10, and 7.12.

**Theorem 7.13.** Given two normalized types \( T \) and \( U \):

\[ T \preceq U \Leftrightarrow T \preceq U \]

**Proof.** \( (\Rightarrow) \) By nested induction on the structure of \( T \) and \( U \). We proceed by case distinction over the shape of \( T \).

1. \( T = () \). Obvious.

2. \( T = B \). Assume that \( U = U_1 \mid \ldots \mid U_n \), with \( U_i \) prime for \( i = 1 \ldots n \). Thanks to \( B \preceq U \), there exists \( j \in 1 \ldots n \) such that \( B \preceq U_j \) (Lemma 7.10); hence the thesis follows by induction on the structure of \( U \).

Assume now that \( U \) is not a union; then, the only possible case is that \( U = U_1, \ldots, U_n \). Therefore, it must exist \( j \in 1, \ldots, n \) such that \( B \preceq U_j \), otherwise the hypothesis is contradicted, hence the thesis follows by induction on the structure of \( U \).
Similar to the previous case.

\( T = l[T_1] \). In this case, we have both \( T_1 \preceq U \) and \( T_2 \preceq U \), as a direct consequence of the hypothesis. So the thesis follows by induction on the structure of \( T \) and by Rule 9 of the simulation definition.

\( T = T_1 \mid T_2 \). In this case, we must distinguish over the form of \( U \). Without loss of generality, assuming that both \( T_1 \neq () \) and \( T_2 \neq () \), we can exclude that \( U = B \) and that \( U = l[V] \). Hence, only three cases are possible. If \( U = U_1 \mid \ldots \mid U_n \), then we can observe that \( T \) is prime, hence Lemma 7.10 implies that there exist \( j \in 1, \ldots, n \) such that \( T \preceq U_j \). By induction on the structure of \( U_j \) we obtain the thesis.

If \( U \) is a product type, then it is prime as it has been normalized, so we can apply Lemma 7.12 so to obtain that \( U = C, D \) and \( T_1 \preceq C \) and \( T_2 \preceq D \). Hence the thesis follows by induction on \( U \) and by Rule 5 of the type simulation relation.

If \( U \) is a \( * \)-type, then it is easy to prove that both \( T_1 \preceq U \) and \( T_2 \preceq U \). Therefore, the thesis follows by induction and by Rule 4 of the type simulation relation definition.

\( T = T^* \). Not difficult thanks to Lemma 7.8. Indeed, by this lemma we have that \( U = U^*, U'' \) and \( T^* \preceq U^* \). At this point it is sufficient to observe that

\[ T^* \preceq U^* \iff T' \preceq U' \]

So the thesis follows by induction on the structure of both left and right hand-side types. Indeed, by applying Rule 4 to the initial pair, we have that \( T' \preceq U^* \) holds if \( T^* \preceq U^* \) does. But this can be proved by Rule 9, thanks to induction, by using \( T' \preceq U' \) which provides \( T' \preceq U^* \), and the case is proved.

\((\Leftarrow)\) By induction on the axiomatization of simulation: as the proof is rather simple, we omit it. \( \Box \)

8. TYPE PROJECTION CHECKING

In the previous section we showed the equivalence between type projection and type simulation. This allows for the construction of an efficient, simulation-based projection-checking algorithm. The algorithm is actually a not-so-naive implementation of the type simulation rules. Indeed, a naive implementation of these rules would lead to a super-exponential algorithm.

The super-exponential complexity of a naive algorithm comes from two key factors. First of all, a recursive comparison of two types \( T_1 \) and \( T_2 \), as suggested by the simulation rules, would lead to many backtracking operations, in particular when comparing union or product types: for instance, when comparing \( l[m[B]] \) with \( l[m[B]], l[m[T]] \) (where \( T \neq B \)), a naive algorithm would (i) apply Rule 4 for product types and choose \( l[m[T]] \) and \( l[m[B]] \) as types to be compared, (ii) start the comparison of the chosen types, and (iii) go back to Rule 4 and step (i) when the comparison fails. This problem can be solved by flattening \( T_1 \) and \( T_2 \), and by constructing a type matrix (\( \text{simT} \)ypes in our algorithm), whose rows and columns are associated, respectively, to type terms in \( T_1 \) and in \( T_2 \). The type matrix is then used to compare each type term in \( T_1 \) with each type term in \( T_2 \) according to the hierarchy of type terms (hence, terms occurring in very distant fragments of \( T_1 \) and \( T_2 \) are not compared); by doing so, the algorithm does not perform backtracking, nor it performs comparisons among types that are “incompatible” according to the type term hierarchy.

The second key factor that makes naive algorithms super-exponential is the comparison among product types. The type simulation definition states that, if outside the immediate scope of a repetition type, \( Z_1, \ldots, Z_n \preceq V_1, \ldots, V_m \) if and only if
each type \( Z_i \) can be mapped into a distinct type in \( V_1, \ldots, V_n \), so that there do not exist \( Z_i \) and \( Z_j \) (\( i \neq j \)) such that \( Z_i \) and \( Z_j \) are mapped into the same type term \( V_k \). This problem is also present in many subtype-checking algorithms, and can be naively solved by generating all possible assignments of \( Z_i \) to \( V_j \) types and by choosing an injective one: this can be done in super-exponential time, as the possible assignments are \( O\left(\binom{m}{n}\right)\).

An alternative solution for the comparison of product types can be obtained by observing that this problem is equivalent to a 0-1 maximum flow problem on bipartite graphs. Indeed, one can build a bipartite graph \( G \), whose first partition \( P_1 \) contains one node per each \( Z_i \) type, and whose second partition \( P_2 \) contains one node per each \( V_j \) type; nodes in \( P_1 \) are connected to a source \( s \), while nodes in \( P_2 \) are connected to a sink \( t \). \( P_1 \) and \( P_2 \) are connected together through edges satisfying the simulation relation, i.e., an edge from \( Z_i \) to \( V_j \) is inserted in \( G \) if \( Z_i \not\preceq V_j \). Each edge has two possible values for its flow: 0 and 1. The source \( s \) emits a flow of \( n \) units, so \( Z_1, \ldots, Z_n \) simulates \( V_1, \ldots, V_m \) if and only if a flow of \( n \) units reaches the sink \( t \). This can be determined by using a quite standard 0-1 maximum flow algorithm on bipartite graph, whose complexity is \( O(n+m)^3 \) [Goldberg 1998]. A similar technique is also used in [Cosmo et al. 2005] for subtype-checking of product types in a rather restrictive type language.

**Remark 8.1.** As stated before, the type simulation relation identifies types modulo commutativity, associativity, and ()-neutrality, hence a type symbol \( T \) actually denotes a class of types. The algorithm for type projection checking drops this assumption, as it treats types in their syntactical form; indeed, it can explicitly derive that two type symbols denote types in the same equivalence class. Hence, in the following \( T = U \) if and only if the two types have the same syntax.

### 8.1 Type Simulation Algorithm

Our algorithm for type simulation checking (\( \text{Sim}(T_1, T_2) \)) is shown in Figures 3 and 4. It consists of three main phases. During Phase 1, the algorithm creates and populates a type matrix \( \text{simTypes} \) with boolean values or symbolic references, both obtained by repeatedly calling the SIMPLESIM algorithm of Figure 3. The matrix has as many rows as the type terms in \( T_1 \), and as many columns as type terms in \( T_2 \). To ensure the proper behavior of the algorithm, type terms from both \( T_1 \) and \( T_2 \) have to be extracted with a pre-order visit, so that, if \( Z_i \) and \( Z_j \) are type terms in \( T_1 \) and \( i < j \), then \( Z_i \) precedes \( Z_j \) in the pre-order visit of \( T_1 \). The SIMPLESIM algorithm is called once per each cell in the type matrix, with the notable exception of those cells that correspond to types whose match is useless as they occupy incompatible positions in the type term hierarchy (this task is performed by the boolean function \( \text{COMPARABLE}^9 \)). For instance, given \( T_1 = l[B, m[B]] \) and \( T_2 = l[[B, p[B]], [B, m[B]]] \), comparing the first \( B \) inside \( T_1 \) with the \( B \) inside \( p[B] \) in \( T_2 \) is a waste of time, as their different positions in the hierarchies of \( T_1 \) and \( T_2 \), respectively, prevent their matching.

SIMPLESIM returns a boolean value for any comparison that does not require any further type comparisons. If, instead, the comparison between \( T \) and \( U \) requires a further comparison, as in the type simulation rule for \( l[Z] \preceq l[W] \) (Rule 3), then the algorithm returns a simple symbolic reference \( \text{ref} \{U_1, V_2\} \), a logical combination of simple references (e.g., \( \bigwedge_{i=1}^n \text{ref} \{U_i, V_{ij}\} \)), or a product reference \( \text{ref} \{U_1, \ldots, U_n \} \otimes \{V_1, \ldots, V_m \} \)). A simple reference \( \text{ref} \{U_2, V_3\} \) is a symbolic

---

9\text{COMPARABLE} is a function based on a parallel scan of the parse trees being compared and is computed at parsing time; its definition is straightforward.
pointer to the content of the cell simTypes[x][y], while a product reference indicates that the maximum flow algorithm must be executed on top of a bipartite graph built from the partitions \{U_1, \ldots, U_n\} and \{V_1, \ldots, V_m\}. References are left unevaluated, and they will be solved during Phase 2.

The output of Phase 1, thus, is a partially instantiated type matrix. This matrix is the input for Phase 2, whose objective is to solve symbolic references. By visiting the matrix in reverse order, starting from the bottom right and going right to left, the algorithm proceeds by replacing references of the form \text{ref}(U_x, V_y) with the content of simTypes[x][y], by evaluating logical combiners, and by applying the maximum flow algorithm for \otimes references; in the latter case, an auxiliary (and trivial) function \text{GraphConstr} is invoked with the aim of building the bipartite graph \mathcal{G}, while the maximum flow is computed with a standard maximum flow algorithm \text{MaximumFlow}. The result of this phase is a fully instantiated type matrix, since, as shown by Lemma 8.7, when a cell simTypes[i][j] has been reached, all the cells that can be referenced by its content have already been visited and instantiated. It should be observed that, as prescribed by Rule 10 of the simulation relation, nodes in the second partition of \mathcal{G}, corresponding to \ast-types in the right hand-side of the comparison, are marked as \text{special}, since they can be used to map more than one term of the left hand-side; from a flow point of view, this means the edges connecting these nodes with the sink of the graph have unbounded capacity.

Phase 3 is very simple and consists in returning a boolean value describing the result of the whole simulation. Since the types being compared correspond to the first row and to the first column of the type matrix, the algorithm just returns the content of simTypes[1][1].

The following example illustrates the behavior of the algorithm on two sample types.

**Example 8.2.** Consider the following two types \(T_1 = l[B, m[B]^*]\) and \(T_2 = l[B, (m[B] | n[B])^*, p[B]]\). As it can observed from the definition of type simulation, \(T_1 \preceq T_2\), since \(B, m[B]^* \preceq B, (m[B] | n[B])^*, p[B]\). Assuming a type term decomposition of \(T_1\) and \(T_2\) obtained by means of a pre-order visit, the algorithm \text{Sim}, when applied to \(T_1\) and \(T_2\), starts with the type matrix of Figure 5. For the sake of simplicity, we insert a \(\bot\) symbol on those cells relating non-comparable types.

During Phase 2, the algorithm solves symbolic references. As it can be seen in the previous matrix, the dependencies among references can be resolved by means of a reverse order visit of the matrix. The output of this phase, hence, is the following (fully instantiated) type matrix:

<table>
<thead>
<tr>
<th></th>
<th>V_1</th>
<th>V_2</th>
<th>V_3</th>
<th>V_4</th>
<th>V_5</th>
<th>V_6</th>
<th>V_7</th>
<th>V_8</th>
<th>V_9</th>
<th>V_{10}</th>
<th>V_{11}</th>
</tr>
</thead>
<tbody>
<tr>
<td>Z_1</td>
<td>\text{true}</td>
<td>\bot</td>
<td>\bot</td>
<td>\bot</td>
<td>\bot</td>
<td>\bot</td>
<td>\bot</td>
<td>\bot</td>
<td>\bot</td>
<td>\bot</td>
<td></td>
</tr>
<tr>
<td>Z_2</td>
<td>\bot</td>
<td>\text{true}</td>
<td>false</td>
<td>false</td>
<td>\bot</td>
<td>\bot</td>
<td>\bot</td>
<td>\bot</td>
<td>\bot</td>
<td>\bot</td>
<td></td>
</tr>
<tr>
<td>Z_3</td>
<td>\bot</td>
<td>\text{false}</td>
<td>\text{true}</td>
<td>false</td>
<td>false</td>
<td>\bot</td>
<td>\bot</td>
<td>\bot</td>
<td>\bot</td>
<td>\bot</td>
<td></td>
</tr>
<tr>
<td>Z_4</td>
<td>\bot</td>
<td>\text{false}</td>
<td>\text{false}</td>
<td>\text{true}</td>
<td>false</td>
<td>false</td>
<td>\bot</td>
<td>\bot</td>
<td>\bot</td>
<td>\bot</td>
<td></td>
</tr>
<tr>
<td>Z_5</td>
<td>\bot</td>
<td>\bot</td>
<td>\bot</td>
<td>\bot</td>
<td>\bot</td>
<td>\bot</td>
<td>\bot</td>
<td>\bot</td>
<td>\bot</td>
<td>\bot</td>
<td></td>
</tr>
<tr>
<td>Z_6</td>
<td>\bot</td>
<td>\bot</td>
<td>\bot</td>
<td>\bot</td>
<td>\bot</td>
<td>\bot</td>
<td>\bot</td>
<td>\bot</td>
<td>\bot</td>
<td>\bot</td>
<td></td>
</tr>
</tbody>
</table>

Since simTypes[1][1] = \text{true}, the algorithm correctly predicts that \(T_1 \preceq T_2\).
SimpleSim($\text{Type } T_1, \text{Type } T_2$)
1 switch
2     case $T_1 = T_2$ :
3         return true
4     case $T_1 = ()$ :
5         return true
6     case $T_1 = a[U_1] \land T_2 = b[U_2]$ :
7         return $ref(U_1, U_2)$
8     case $T_1 = a, \ldots , a \land T_2 = b, \ldots , b$ :
9         return $ref(\{a, \ldots , a\} \otimes \{b, \ldots , b\})$
10    case $T_1 = a, \ldots , a \land T_2 = b, \ldots , b$ :
11         return $v^m_{j=1} ref(T_j, V_j)$
12     case $T_1 = a | \ldots | a \land T_2 = b | \ldots | b$ :
13         return $\bigvee_{j=1}^m ref(T_j, V_j)$
14    case $T_1 = a \ast \land T_2 = b \ast$ :
15         return $ref(U, T_2)$
16     case $T_1 = a, \ldots , a \land T_2 = b \ast$ :
17         return $\bigwedge_{i=1}^n ref(U_i, T_2)$
18     case $T_1 = a \ast \land T_2 = b, \ldots , b$ :
19         return $ref(U, V)$
20     case default :
21         return false

Fig. 3. SimpleSim algorithm.

8.2 Correctness and Completeness
To prove correctness and completeness of the Sim algorithm wrt type simulation, we need the following lemmas.

**Lemma 8.3.** The SimpleSim algorithm satisfies the reflexivity, commutativity, associativity, and ()-neutrality properties.

**Proof.** See the Appendix.

**Lemma 8.4.** The SimpleSim algorithm is compatible with the simulation relation, in the sense that it implements the simulation relation.

**Proof.** See the Appendix.

**Lemma 8.5.** Given $i \in [1, \ldots , n]$, given $j \in [1, \ldots , m]$, if $\text{simTypes}[i][j]$ contains a reference $ref(Z_x, V_y)$, then $x \geq i$ and $y \geq j$.

**Proof.** By a simple inspection of the SimpleSim algorithm.

**Lemma 8.6.** Given $i \in [1, \ldots , n]$, given $j \in [1, \ldots , m]$, if $\text{simTypes}[i][j]$ contains a reference of the form $ref(\{Z_f, \ldots , Z_h\} \otimes \{V_p, \ldots , V_q\})$, then $i \leq f, \ldots , i \leq h$ and $j \leq p, \ldots , j \leq q$.

**Proof.** By a simple inspection of the SimpleSim algorithm.
Sim$(\text{Type } T_1, \text{Type } T_2)$
1 // we assume $T_1$ to be composed by $n$ terms and $T_2$ by $m$ terms
2 // phase 1: type matrix construction
3 Array$[n][m]$ simTypes
4 for each $U_i$ in $T_1$
5   do for each $V_j$ in $T_2$
6     do if Comparable$(U_i, V_j)$
7        then simType$[i][j] = \text{SimpleSim}(U_i, V_j)$
8 // phase 2: reference resolution
9 for $i ← n$ to 1
10 do for $j ← m$ to 1
11   do if simTypes$[i][j] = \text{ref}(Z_p, U_q) \land \text{simTypes}[p][q] ∈ \{\text{true, false}\}$
12      then simTypes$[i][j] = \text{simTypes}[p][q]$
13      else if simTypes$[i][j]$ contains references different from $*$
14          then for each ref$(Z_x, V_y)$ in simTypes$[i][j]$
15            do replace ref$(U_x, V_y)$ with simTypes$[x][y]$
16          evaluate the logical expression in simTypes$[i][j]$
17      else if simTypes$[i][j] = \text{ref}\{U_f, \ldots, U_p\} \otimes \{V_g, \ldots, V_q\}$
18        then $\mathcal{P}_1 = \{U_f, \ldots, U_p\}$
19        $\mathcal{P}_2 = \{V_g, \ldots, V_q\}$
20        mark as special $\mathcal{P}_2$ nodes corresponding to $*$-types
21        $G = \text{GraphConstr}(\mathcal{P}_1, \mathcal{P}_2)$
22        simTypes$[i][j] = \text{MaximumFlow}(G)$
23 // phase 3: result discovery
24 return simTypes$[1][1]$

Fig. 4. Type simulation algorithm.

These lemmas allow us to prove the following lemma about the second phase of the Sim algorithm.

**Lemma 8.7.** Phase 2 of the Sim algorithm produces a fully instantiated type matrix.

**Proof.** Phase 2 consists of a reverse order visit of the type matrix. When a cell simTypes$[i][j]$ is examined, by Lemmas 8.5 and 8.6, it may contain only forward references, i.e., references to already visited cells. These references have already been solved, hence the cell simTypes$[i][j]$ can be fully instantiated.

We can now state the correctness and completeness of Sim wrt type simulation.

**Theorem 8.8.** The Sim algorithm is correct and complete wrt the type simulation relation.

**Proof.** We first observe that the algorithm terminates, as it does not make recursive calls nor it uses unbounded iterations.

We can now prove the thesis by analyzing each phase of the algorithm.

**Phase 1**
Assuming that $T_1$ is formed by $n$ type terms (from $Z_1$ to $Z_n$), and that $T_2$ is formed by $m$ type terms (from $V_1$ to $V_m$), Phase 1 creates a matrix simTypes of $n \times m$ entries, where each entry contains a boolean value or a reference to other
entries: \(\text{simTypes}[i][j]\) indicates whether \(Z_i\) is similar to \(V_j\) (true or false); if the simulation cannot be directly computed, a reference to entries of nested terms is inserted. For each entry, Phase 1 calls the SIMPLESIM algorithm, which, as shown by Lemma 8.4, implements the rules of the definition of type simulation.

**Phase 2**

Phase 2 solves symbolic references in the matrix entries. The only potential source of incompleteness is the comparison among product types. However, we have already shown, even if informally, that this comparison is equivalent to a 0 \(-\) 1 maximum flow problem on bipartite graphs, and that our algorithm is able to capture all matching among product types.

**Phase 3**

Phase 3 just outputs the result of the algorithm.
8.3 Complexity Analysis

To study the complexity of the Sim algorithm we must first analyze the complexity of the auxiliary algorithms SIMPLESIM.

**Lemma 8.9.** The SIMPLESIM algorithm has $O(1)$ worst case complexity.

**Proof.** We assume here that matching a type with the guard of the corresponding case (e.g., `case T1 = ...`) in the Sim algorithm has $O(1)$ complexity.

Each case in the SIMPLESIM algorithm performs no recursive calls nor iterations; furthermore, each reference returned by the return clause is just a symbolic reference and does not involve any operation to be performed. As a consequence, each case requires just $O(1)$ operations.

**Theorem 8.10.** The worst case complexity of the Sim algorithm, while comparing $T$ and $U$, is $O(nm(n + m)^3)$, where $n$ is the number of terms in $T$, and $m$ is the number of terms in $U$.

**Proof.** We prove the thesis by analyzing each phase of the algorithm.

**Phase 1**
Phase 1 creates a matrix of $n \times m$ entries, where each entry contains the projection relation between two types. For each entry, phase 1 calls the SIMPLESIM algorithm, which, as shown in Lemma 8.9, has $O(1)$ complexity. Moreover, each calls to the function COMPARABLE has $O(1)$ complexity, as it involves a single lookup in a compatibility matrix that can be built in $O(nm)$ time during type parsing. Phase 1, hence, performs $O(nm)$ operations.

**Phase 2**
Phase 2 performs the resolution of symbolic references by traversing the type matrix in reverse order. Simple references of the form $ref(Z_x, V_y)$ can be solved with a single access to the matrix, while each logical expression of the form $\bigwedge_{i=1}^{n} \bigvee_{j=1}^{m} ref(U_i, V_j)$ can be evaluated in at most $nm$ operations.

The resolution of $\otimes$-references requires the construction of the bipartite graph $G$ ($O(nm)$ operations), and the execution of the $0-1$ maximum flow algorithm. The best $0-1$ maximum flow algorithm on bipartite graphs is a variant of the Ford-Fulkerson algorithm and has $O((n + m)^3)$ complexity [Goldberg 1998], hence this phase has $O(nm(n + m)^3)$ worst case time complexity.

**Phase 3**
Phase 3 requires just one operation.

In Section 6.2 we showed that type projection is NP-hard. The PTIME complexity of Sim does not conflict with that result, as Sim works on normalized types only, and normalization has exponential complexity.

9. STATIC TYPE INFERENCE

As stated in the Introduction, our approach can be used to verify that a mapping $m$ from $T$ to $U$ is correct only if we are able to infer a type $U'$ describing the structure of the output of $m$, that is a type $U'$ such that: for each $f : T$, $m(f) : U'$. So, $m$ is correct if $U \lesssim U'$ holds.

Depending on the “precision” of the inferred type $U'$, it may happen that $m$ is correct while $U' \lesssim U$ does not hold. Such a false-negative is due to the fact that the type system has not been clever enough to infer a very precise type for the mapping, that is, a type whose semantics is quite close to the set of the mapping co-domain.
Of course, in the presence of false negatives we bother the user with error warnings without any real motivation; furthermore, in this case it is very likely that the user makes unneeded (the mapping is correct!) efforts to change the mapping. So, it is crucial to develop inference techniques that return quite precise inferred types and decrease false negatives, so to improve the effectiveness of our approach.

In this section we illustrate how a quite precise output type can be inferred for a mapping expressed in the \( \mu \)XQ language (defined in Section 3.1): since \( \mu \)XQ is rather expressive, we are quite confident that the proposed inference technique can be generalized to different mapping approaches. To this end we provide opportune query typing rules, prove soundness of the resulting type system, and finally show that the resulting typing algorithm is precise and efficient at the same time.

9.1 Methodology

Our type inference system aims at gaining a good level of precision, so to decrease false negatives. To this end we leverage on the following fact:

1. If \( U' \preceq U \), then \( U' \ast \preceq U \) may not hold.

This fact suggests that the generation of star types during type inference should be avoided as much as possible, as they are a potential source of false negatives. To this purpose, when typing for-iterations or self-or-descendants selectors, we drop the over-approximation approach described in XQuery type system [Draper et al. 2007], and adopt the more precise case-analysis technique [Colazzo. et al. 2006], as well as a new and quite natural technique for typing self-or-descendants selectors.

9.2 Preliminaries

Our type inference system is able to prove judgments of the form \( \Gamma \vdash Q : T \), where \( Q \) is a query, \( \Gamma \) is an environment providing type information about the free variables of \( Q \), and \( T \) is an upper bound for all possible values returned by \( Q \). In particular, both \( \Gamma \) and \( Q \) are intended to be the input arguments of the type inference process, while \( T \) is the output to be inferred.

**Definition 9.1 Variable environment.** A variable environment \( \Gamma \) is a list of pairs \( \chi : T \), where \( \chi \) is a for-variable or a let-variable, and \( T \) is a type. Variable environments meet the following grammar:

\[
\text{Variable Environments} \quad \Gamma ::= () \mid x : T, \Gamma \mid \bar{x} : T, \Gamma
\]

A variable environment \( \Gamma \) is well-formed if no variable is defined twice, and if every for-variable \( \bar{x} \) (i.e., a variable bound by a for clause) is associated to a tree type (\( l[\bar{T}] \) or \( B \)). Moreover, in the following each time we consider a variable environment \( \Gamma \) for a query \( Q \), we will assume that \( \Gamma \) provides definitions for all free-variables of \( Q \). As a consequence, we will assume that \( Q \) can be evaluated only under substitutions \( \rho \) that are valid with respect to \( \Gamma \).

**Definition 9.2 Valid substitutions** \( \mathcal{R}(\Gamma) \). For any well-formed environment \( \Gamma \), we define the set \( \mathcal{R}(\Gamma) \) of valid substitutions wrt \( \Gamma \) as follows:

\[
\mathcal{R}(\Gamma) = \{ \rho \mid \chi \mapsto f \in \rho \Rightarrow (\chi : T \in \Gamma \wedge f \in [T]) \}
\]

9.3 Judgments

Beyond \( \Gamma \vdash Q : T \), several auxiliary judgments are employed in our type rules. These judgments serve the purpose of keeping rules relatively simple, while allowing for a good level of precision of type inference.
For each type $T$, the type rules of our type system are shown in Tables 9.1 and 9.2. More in details, case analysis for for-iterations is performed by the (TypeInTree) rule family, which uses the auxiliary judgments $\Gamma \vdash \varphi \in T \rightarrow Q_2 : T_2$. In particular, termination is ensured by rule (TypeInTree), where we use a particular operator Split($T$), defined in Figure 7. Rule (TypeInTree) stops the case-analysis since a tree type $T = B$ or $T = m[T']$ is reached, inserts the assumption $\varphi : T$ in $\Gamma$, and falls back to standard type-checking.

Splitting preserves type semantics.

**Lemma 9.4.** For each type $T$:

$$[T] = \bigcup_{A \in \text{Split}(T)} [A]$$

**Proof.** By induction on the cardinality of \text{Split}($T$) and by case distinction on the shape of $T$. □

The Split($\cdot$) operator allows the type system to improve the precision of the inferred type for for clauses wrt the technique described in [Draper et al. 2007]. Indeed, consider the input type $X = \text{data}[\text{mbl}[B]+ | \text{phn}[B]+]'$, and the sequence query (x/mbl, x/phn). When $x$ has type $X$, this query yields either a sequence of elements mbl[B] or a sequence of elements phn[B]. Instead, as for the W3C-like type rules for XQuery, without type-splitting (that is by assuming Split($T$)) = \{T\} in the type rules) our type system would infer the type (mbl[B]+, phn[B]+), which also contains sequences with both mbl[B] and phn[B] elements. Now, if this inferred type

\[ Split(\cdot) \triangleq \{()\} \quad \text{Split}(T | U) \triangleq \text{Split}(T) \cup \text{Split}(U) \]
\[ \text{Split}(B) \triangleq \{B\} \quad \text{Split}([T]) \triangleq \{[A] | A \in \text{Split}(T)\} \]
\[ \text{Split}(U^*) \triangleq \{U^*\} \quad \text{Split}(T, U) \triangleq \{(A, B) | A \in \text{Split}(T) \land B \in \text{Split}(U)\} \]
is compared for projection with respect to an expected type $(mb|B|* \mid phn|B|*)$, in order to check whether the query output conforms to this expected type, the checking will fail, thus producing a false negative. By relying on the $Split(\cdot)$ function (as defined in Figure 7), our type system splits the input type in the two types $data[mb|B|+]$ and $data[phn|B|+]$, which are separately analyzed; as a consequence, we obtain the types $data[mb|B|*]$ and $data[phn|B|*]$. The query type, then, is the union of these two types, and thus a projection (and subtype) of the previous expected type, hence avoiding a false negative.

It should be observed that type splitting stops when a *-type is met, and this is one of the main differences wrt the function $norm(\cdot)$, used in previous sections. As shown in [Colazzo et al. 2004], stopping the splitting when a *-type is met ensures acceptable complexity for a very wide class of schemas, while ensuring good precision at the same time, as in most schemas union types have the form $(T \mid U)*$, which are guarded by repetition types and, then, not split.

Type rules in Tables 9.1 and 9.2 deal with *-guarded environments, and *-guardedness is preserved by backward rule applications. In order to generalize their use, we need to split any environment $\Gamma$ before the typing process starts. This is made possible by the following definition.

**Definition 9.5** Query Variables Environment Splitting. For each $\Gamma$ well-formed, we extend splitting to $\Gamma$ in order to decompose it into a finite set of *-guarded environments, which we call $SplitVEnv(\Gamma)$:

\[
\begin{align*}
SplitVEnv(()) & \triangleq \emptyset \\
SplitVEnv((\Gamma, \chi : T)) & \triangleq \{ \Gamma', \chi : A \mid \Gamma' \in SplitVEnv(\Gamma) \land A \in Split(T) \}
\end{align*}
\]

This kind of splitting preserves variable environment semantics (Lemma 9.6), and is needed in Definition 9.7 to extend type splitting to judgements with a generic variable environment $\Gamma$.

**Lemma 9.6.** For each *-guarded type environment $E$ and $\Gamma$ well-formed in $E$:

\[
\bigcup_{\Gamma' \in SplitVEnv(\Gamma)} R(\Gamma') = R(\Gamma)
\]

**Proof**. By induction on the length of $\Gamma$ and by Lemma 9.4.  

**Definition 9.7** Typing by Splitting. For any well-formed query $Q$, any well-formed $\Gamma$, we indicate with $\Gamma \vdash Q : U$ the following facts

\[
\begin{align*}
(1) & \quad SplitVEnv(\Gamma) = \{ \Gamma_1, \ldots, \Gamma_n \} \\
(2) & \quad \Gamma_i \vdash Q : U_i, \quad i = 1 \ldots n \\
(3) & \quad U = U_1 \mid \ldots \mid U_n
\end{align*}
\]

Rule (TypeLetSplitting) is quite standard, and makes use of $Split(\cdot)$ too.

In both type rules (TypeChild) and (TypeDos), as well as in other rules, the statement $WF(\Gamma \vdash Q : U)$ means that the judgement $\Gamma \vdash Q : U$ is well-formed, that is: in $\Gamma$ no variable occurs twice, and each free-variable in $Q$ occurs (is defined) in $\Gamma$.

Rule (TypeChild) requires the type of $\tau$ to be a tree type $m[T']$, and uses the judgment $\vdash T' :: NodeTest \Rightarrow U$.

Rule (TypeDos) is another significant deviation from the standard techniques used in XQuery. Indeed, the sample query $\$x/label$ would be typed by the XQuery type system by collecting all the node types $\{U_1, \ldots, U_n\}$ that are reachable.
from $\mathcal{T}$ type, by building an auxiliary type $U' = (U_1 \mid \ldots \mid U_n)^*$, and by restricting $U'$ to the tree types with structure satisfying the node test $\text{label}$. This technique is probably the best compromise between simplicity and precision when also vertical recursive types are allowed. However, for non-recursive types, it yields some inferred types with very low precision. As an example, consider the type $T = ([l[m[c[B]]]^*], \text{the typed variable } x: T$ and the query $\text{for } y \text{ in } x \text{ return } b[y \text{ dos : c}].$ With the current W3C-like approach, for this query we would infer the type $U_Q = b[c[B]^*]^*$, while a precise upper bound would be $b[c[B]]^*$, which is exactly the type we are able to infer with Rule (TypeDos) of Table 9.2. In other words, this more precise type rule is able to minimize the number of $^*$-types in the inferred type, as requested by fact (1) in Section 9.1.

Rule (TypeDos) uses the auxiliary function $\text{dos}(\cdot)$, which ensures more precision of type inference for self-or-descendants selectors by simulating the execution of the selector on the input type, in a quite accurate way. The function $\text{dos}(\cdot)$ is well defined due to the absence of vertical recursion in the schemas we deal with.

**Definition 9.8.**

$$
\begin{align*}
\text{dos}(()) & = () \\
\text{dos}(B) & = B \\
\text{dos}(l[T]) & = l[T], \text{dos}(T) \\
\text{dos}(T^*) & = \text{dos}(T)^* \\
\text{dos}(T, U) & = \text{dos}(T), \text{dos}(U) \\
\text{dos}(T | U) & = \text{dos}(T | \text{dos}(U))
\end{align*}
$$

It is straightforward to prove the following lemma.

**Lemma 9.9 Soundness of $\text{dos}(\cdot)$**. For any $T$:

$$
\text{dos}(T) = U \Rightarrow \forall f \in [T]. \text{dos}(f) \in [U]
$$

<table>
<thead>
<tr>
<th>Table 9.1. Query Type Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>(TypeEmpty) $WF(\Gamma \vdash () : ()$</td>
</tr>
<tr>
<td>$\Gamma \vdash () : ()$</td>
</tr>
<tr>
<td>(TypeAtomic) $WF(\Gamma \vdash b : B$</td>
</tr>
<tr>
<td>$\Gamma \vdash b : B$</td>
</tr>
<tr>
<td>(TypeVar $\chi : T \in \Gamma \ W F(\Gamma \vdash \chi : T$</td>
</tr>
<tr>
<td>$\Gamma \vdash \chi : T$</td>
</tr>
<tr>
<td>(TypeElem) $\Gamma \vdash Q : T$</td>
</tr>
<tr>
<td>$\Gamma \vdash l[Q] : l[T]$</td>
</tr>
<tr>
<td>(TypeForest $\Gamma \vdash Q_i : T_i \ i = 1, 2$</td>
</tr>
<tr>
<td>$\Gamma \vdash Q_1, Q_2 : T_1, T_2$</td>
</tr>
<tr>
<td>(TypeLetWhere) $\Gamma \vdash Q_1 : T_1 \text{ Split}(T_1) = {A_1, \ldots, A_n}$</td>
</tr>
<tr>
<td>$\Gamma, x : A_i \vdash Q_2 : U_i \ i = 1, \ldots, n$</td>
</tr>
<tr>
<td>$\Gamma \vdash \text{let } x ::= Q_1 \text{ where } P \text{ return } Q_2 : U_1 \mid \ldots \mid U_n \mid ()$</td>
</tr>
<tr>
<td>(TypeForWhere) $\Gamma \vdash Q_1 : T_1$</td>
</tr>
<tr>
<td>$\Gamma \vdash \exists \text{ in } T_1 \rightarrow Q_2 : T_2$</td>
</tr>
<tr>
<td>$\Gamma \vdash \text{for } \exists \text{ in } Q_1 \text{ where } P \text{ return } Q_2 : T_2 \mid ()$</td>
</tr>
</tbody>
</table>
Theorem 9.10 Upper Bound. For any well-formed \( \Gamma \) and query \( Q \):

\[
\Gamma \vdash Q : U \land \rho \in \mathcal{R}(\Gamma) \Rightarrow [Q]_\rho \in [U]
\]

The proof of this theorem is essentially the same as the one given in [Colazzo et al. 2006], since considered XPath-like paths do not match the horizontal structure of sequences, so their typing does not depend on ordering.

This theorem is crucial to guarantee soundness of mapping correctness checking. Indeed, if \( m = (Q, \{ q_k \}_k) \) is a mapping from \( S_i \) to \( S_j \), and \( \Gamma \vdash Q_{\{ q_k \}_k} : U \), then, thanks to the above theorem, we can compare \( U \) wrt \( S_j \) in order to verify whether the semantics of \( Q_{\{ q_k \}_k} \) conforms to \( S_j \).

The previous theorem, therefore, guarantees that, whenever a mapping is deemed as correct, it is really correct. This, in turn, implies that our approach is able to detect all errors in a mapping. Unfortunately, we cannot give a theorem stating that, if a mapping is flagged as incorrect, the mapping contains an error, i.e., we cannot avoid false negatives. Indeed, there is no type inference system able to infer precise types for XQuery-inspired languages like \( \mu XQ \), as there are some perfectly legal queries that produce output languages that cannot be captured by regular expression types; in these cases the inferred types may be so imprecise that a false negative is generated. Consider, for example, the query \( \mathcal{T}/b, \mathcal{T}/b \) with \( \mathcal{T} \) of type \( T = a[b[B]*] \). This query produces sequences with an even number of \( b \) elements, and such a set of sequences cannot be precisely described by a regular expression type.

While we cannot avoid false negatives in the general case, still there is a large class of \( \mu XQ \) mappings for which our inference system can infer a precise type wrt type projection, as shown in the following theorem.

Theorem 9.11 Lower Bound wrt Projection. For any well-formed \( \Gamma \) and query \( Q \) without where clauses and such that the only navigational mechanism used
in $Q$ has the form $\text{child} :: \text{NodeTest}$, with NodeTest being an element label $l$, it holds that:

$$\Gamma \vdash Q : U \Rightarrow \forall f \in [U]. \exists \rho \in \mathcal{R}(\Gamma). \ f \preccurlyeq [Q]_{\rho}.$$

**Proof.** See the Appendix. □

It is worth observing that a large and commonly used class of mappings (capturing most of Piazza and Clio mappings) falls in the restrictions stated in the above theorem, with the exception of the restriction about the absence of *where* clauses.

**Theorem 9.12 Completeness of mapping correctness checking.** Given a mapping $m = (Q, \{q_k\}_k)$ from $S_i$ to $S_j$, without *where* clauses and where the only used navigational selector has the form $\text{child} :: \text{NodeTest}$ with NodeTest being an element label $l$, $m$ is correct if and only if $\Gamma \vdash Q[[q_k]_k] : T$ and $T \preccurlyeq S_j$, where $\Gamma$ is an environment opportunely obtained from $S_i$.

**Proof.** The *if* direction follows from Proposition 4.3 and Theorem 9.10. For the *only if* direction, by Theorem 9.11 and transitivity of $\preccurlyeq$, we have that, for each forest $f_{S_j}$, there exists a forest $f$ produced by $m$ (when applied on a $S_i$ forest) such that $f \preccurlyeq f_{S_j}$, which entails that the mapping $m$ is correct.

□

We suspect that the above theorem can be extended to queries containing the descendant-or-self axes as well, when the input type meets some particular requirement, but we did not investigate this issue, and leave this for further developments of this work.

10. EXPERIMENTAL EVALUATION

In the previous sections we analyzed the theoretical properties of a projection-based approach for capturing errors in schema mappings. This approach has been proved to capture all errors in a mapping. Furthermore, we proved that this approach has exponential complexity in the general case, and it is polynomial on many practical cases.

In this section we want to experimentally validate the usefulness of the proposed approach. To this end, we focus our experimental evaluation on two key aspects: precision and scalability.

An experimental analysis of the precision properties of our approach is important as type inference may generate over approximations of the output type of a given schema mapping, which, in turn, may lead to *false negatives*, i.e., mappings that are deemed as incorrect even though they are perfectly legal. Hence, our approach can be considered practically useful only if it keeps false negatives very low (or does not generate false negatives at all in most common practical scenarios). As a consequence, our first battery of experiments analyzes the precision of our approach on the data integration benchmark described in [Alexe et al. 2008]. This benchmark comprises most transformations used in practical scenarios and represents an interesting way to validate our approach on “real world” scenarios.

The second battery of experiments focus on scalability. As our approach has a (single) exponential worst case complexity, its practical usefulness lies in its ability to beautifully scale and perform even on large schemas or complex mappings: a very precise maintenance algorithm is useless if it is slow on complex schemas and/or mappings. As a consequence, we analyze the behaviour of the algorithms when the number of rules in the mapping or the size of the source and/or target
schemas change: in particular, we evaluate algorithms’ performance on an increasing number of mapping rules when (i) the source schema size increases, (ii) the number of addenda\textsuperscript{12} in non-*-guarded unions increases, and (iii) the number of factors in product types increases. Settings (ii) and (iii) correspond to potentially critical issues for the type projection algorithm, while setting (i) describes a critical situation for the type inference algorithm. We also evaluate the performance of type projection on *-guarded unions, to understand how non-*-guarded unions slow down the performance of our algorithm. We prefer to study the effect of unions and products in isolation to understand their individual contribution to the final performance.

We perform our scalability tests on the schema of the XML-encoded version of DBLP (available at http://dblp.uni-trier.de/xml/). This dataset has already been used for experimental evaluations of data integration and exchange techniques, and reflects the features of commonly used schemas [Hernández et al. 2007; Bonifati et al. 2008].

10.1 Experimental Setup

Our algorithm has been implemented in Java 1.5 and all experiments were performed on a 2.16 Ghz Intel Core 2 Duo machine (3 GB main memory) running Mac OSX 10.5.5. To avoid issues related to independent system activities, we ran each experiments ten times, discarded both the highest (worst) and the lowest (best) processing times, and reported the average processing time of the remaining runs.

10.2 Precision Experiments

To verify the precision of our approach, we rely on the mapping system benchmark described in [Alexe et al. 2008]. This benchmark comprises the most widely used transformations, which can be further expanded or composed to form very complex mappings.

In our tests we validate our approach against the sample XQuery benchmark listed in the Appendix of [Alexe et al. 2008] (see also http://www.stbenchmark.org/). The purpose of these tests is to verify whether our approach generates false negatives in practical scenarios. As shown in Table I, we compare the expected projection result with the actual output of our algorithm; for the sake of completeness we also measured the inference and projection average time for each scenario.

As a first observation, mapping scenario 10 has not been tested as it contains a predicate (exists()) which is not supported by our predicate language. In all the remaining cases, our system generates no false negatives and, as expected, deemed the mappings as correct. It should be observed that, even in the presence of mappings with deeply nested subqueries (composed scenario) or schemas with deeply nested repetition types (expanded scenario), our inference system generates a precise output type.

10.3 Scalability Experiments

We based our scalability experiments on an XML-encoded version of the DBLP schema, which is relatively flat and regular. The source and target schemas we used are shown in Figures 8 and 9: while the source schema is just a translation of the DBLP DTD into our schema language, the target schema has a significantly different element nesting. A fragment of the source and target schemas is graphically represented in Figure 10.

\textsuperscript{12}Given a union type $T_1 \mid T_2$, $T_1$ and $T_2$ are the addenda of the union.
<table>
<thead>
<tr>
<th>Scenario</th>
<th>Inf. Avg Time (ms)</th>
<th>Proj. Avg Time (ms)</th>
<th>Expected Result</th>
<th>Actual Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>1</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>1</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>1</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>1</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>&lt; 1</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>1</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>1</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>9</td>
<td>3</td>
<td>&lt; 1</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>&lt; 1</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>Expanded</td>
<td>2</td>
<td>&lt; 1</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>Composed</td>
<td>108</td>
<td>23</td>
<td>true</td>
<td>true</td>
</tr>
</tbody>
</table>

Table I. Benchmark evaluation (times in ms).

DBLP = dblp[(Article | InProceedings | Proceedings | Book | InCollection | PhDThesis | MasterThesis | WWW)*];
Article = :article[Content*];
InProceedings = :inproceedings[Content*];
Proceedings = :proceedings[Content*];
Book = :book[Content*];
InCollection = :incollection[Content*];
PhDThesis = :phdthesis[Content*];
MasterThesis = :masterthesis[Content*];
WWW = :www[Content*];
Content = Author | Editor | Title | BookTitle | Pages | Year | Address | Journal | Volume | Number | Month | Url | EE | Cdrom | Cite | Publisher | Note | CrossRef | Isbn | Series | School | Chapter;
Author = author[B];
Editor = editor[B];
Title = title[B];
BookTitle = booktitle[B];
Pages = pages[B];
Year = year[B];
Address = address[B];
Journal = journal[B];
Volume = volume[B];
Number = number[B];
Month = month[B];
Url = uri[B];
EE = ee[B];
Cdrom = cdrom[B];
Cite = cite[B];
Publisher = publisher[B];
Note = note[B];
CrossRef = crossref[B];
Isbn = isbn[B];
Series = series[B];
School = school[B];
Chapter = chapter[B];

Fig. 8. DBLP source schema.
We used six schema mappings of increasing complexity, as shown below. These mappings contain self-or-descendants selectors as well as nested queries, hence they significantly stress the performance of our type inference algorithm. Our mappings essentially reverse the nesting of schema elements and are representative of a widely used class of mappings.

**Query 1:**
```xml
authorNested[ for $x in $DBLP//author
    return entry[ name[ $x/text() ],
        articles[ for $y in $DBLP/article
            where $y/author = $x
            return $y/title ]] ]
```

**Query 2:**
```xml
authorNested[ for $x in $DBLP//author
    return entry[ name[ $x/text() ],
        articles[ for $y in $DBLP/article
            where $y/author = $x
            return $y/title ],
        books[ for $y in $DBLP/book
            where $y/author = $x
            return $y/title ]] ]
```

**Query 3:**
```xml
authorNested[ for $x in $DBLP//author
    return entry[ name[ $x/text() ],
        articles[ for $y in $DBLP/article
            where $y/author = $x
            return $y/title ]] ]
```
Query 4: authorNested[ for $x in $DBLP//author
   return entry[ name[ $x/text() ],
       articles[ for $y in $DBLP/article
                   where $y/author = $x
                   return $y/title ],
       books[ for $y in $DBLP/book
                   where $y/author = $x
                   return $y/title ],
       phdthesis[ for $y in $DBLP/phdthesis
                   where $y/author = $x
                   return $y/title ],
       masterthesis[ for $y in $DBLP/masterthesis
                     where $y/author = $x
                     return $y/title ]]
]

Query 5: authorNested[ for $x in $DBLP//author
   return entry[ name[ $x/text() ],
       articles[ for $y in $DBLP/article
                   where $y/author = $x
                   return $y/title ],
       books[ for $y in $DBLP/book
                   where $y/author = $x
                   return $y/title ],
       phdthesis[ for $y in $DBLP/phdthesis
                   where $y/author = $x
                   return $y/title ],
       masterthesis[ for $y in $DBLP/masterthesis
                     where $y/author = $x
                     return $y/title ],
       inproceedings[ for $y in $DBLP/inproceedings
                      where $y/author = $x
                      return $y/title ]]
]

Query 6: authorNested[ for $x in $DBLP//author
   return entry[ name[ $x/text() ],
       articles[ for $y in $DBLP/article
                   where $y/author = $x
                   return $y/title ],
       books[ for $y in $DBLP/book
                   where $y/author = $x
                   return $y/title ],
       phdthesis[ for $y in $DBLP/phdthesis
                   where $y/author = $x
                   return $y/title ],
       masterthesis[ for $y in $DBLP/masterthesis
                     where $y/author = $x
                     return $y/title ],
       inproceedings[ for $y in $DBLP/inproceedings
                      where $y/author = $x
                      return $y/title ],
       proceedings[ for $y in $DBLP/proceedings
                    where $y/author = $x
                    return $y/title ]]
]
Inference Tests. We evaluated the performance of the inference algorithm when the number of addenda of union types in the source schema increases from 10 to 100. Experimental results are shown in Table II and, graphically, in Figure 11 (times are expressed in ms). As it can be seen, our algorithm is very fast and its performance scales beautifully. This is almost due to our specialized inference technique for self-or-descendants selectors, which allows one to avoid useless traversals of the type tree. The performance of the inference algorithm is very good even on a schema with 100-addendum union types, which is a relatively large schema.

<table>
<thead>
<tr>
<th>No. of addenda</th>
<th>Query1</th>
<th>Query2</th>
<th>Query3</th>
<th>Query4</th>
<th>Query5</th>
<th>Query6</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>5</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>20</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>5</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>30</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>40</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>50</td>
<td>3</td>
<td>5</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>60</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>70</td>
<td>4</td>
<td>5</td>
<td>8</td>
<td>8</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>80</td>
<td>6</td>
<td>7</td>
<td>6</td>
<td>8</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>90</td>
<td>5</td>
<td>6</td>
<td>8</td>
<td>9</td>
<td>12</td>
<td>11</td>
</tr>
<tr>
<td>100</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>13</td>
<td>11</td>
</tr>
</tbody>
</table>

Table II. Type inference performance in the DBLP scenario (times in ms).

Projection Tests. To evaluate the properties of the projection algorithm, we revert to the source schema of Figure 8, and altered the target schema. In the first experiment we modified the target schema and inserted three non-*-guarded unions with a number of addenda ranging from 1 to 10 (the total number of addenda is between 3 and 30). These unions are put side by side in the same product type; this
Fig. 12. Projection performance in the DBLP scenario: non-*-guarded union types (times in ms).

configuration describes a worst case situation that only rarely happens in practical scenarios. The results of our evaluation are shown Table 10.3 and, graphically, in Figure 12. These results tell us that, as expected, the performance of type projection degrades when the number of addenda increases. However, the algorithm does not expose an exponential behavior; this is actually a good news, as it implies that our solution can be practical useful.

<table>
<thead>
<tr>
<th>No. of addenda</th>
<th>Query1</th>
<th>Query2</th>
<th>Query3</th>
<th>Query4</th>
<th>Query5</th>
<th>Query6</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>&lt; 1</td>
<td>&lt; 1</td>
<td>&lt; 1</td>
<td>&lt; 1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>4</td>
<td>13</td>
<td>16</td>
<td>19</td>
<td>22</td>
<td>24</td>
<td>27</td>
</tr>
<tr>
<td>5</td>
<td>29</td>
<td>39</td>
<td>45</td>
<td>46</td>
<td>52</td>
<td>56</td>
</tr>
<tr>
<td>6</td>
<td>50</td>
<td>60</td>
<td>69</td>
<td>79</td>
<td>90</td>
<td>99</td>
</tr>
<tr>
<td>7</td>
<td>122</td>
<td>146</td>
<td>172</td>
<td>161</td>
<td>184</td>
<td>198</td>
</tr>
<tr>
<td>8</td>
<td>157</td>
<td>180</td>
<td>202</td>
<td>226</td>
<td>250</td>
<td>272</td>
</tr>
<tr>
<td>9</td>
<td>366</td>
<td>358</td>
<td>390</td>
<td>424</td>
<td>456</td>
<td>526</td>
</tr>
<tr>
<td>10</td>
<td>400</td>
<td>444</td>
<td>489</td>
<td>534</td>
<td>635</td>
<td>675</td>
</tr>
</tbody>
</table>

Table III. Projection performance in the DBLP scenario: non-*-guarded union types.

As a comparison, we performed a similar experiment by adding to the target schema a *-guarded union type with a number of addenda ranging from 10 to 100. The results of our evaluation are shown in Table 10.3 and Figure 13. It can be observed that the performance of type projection is definitely better than in the non-*-guarded scenario. This is consistent with our complexity analysis, as *-guarded unions do not generate the normalization exponential blow up. It is
interesting to note that scalability is very good and that projection time is nearly constant.

<table>
<thead>
<tr>
<th>No. of addenda</th>
<th>Query1</th>
<th>Query2</th>
<th>Query3</th>
<th>Query4</th>
<th>Query5</th>
<th>Query6</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>20</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>30</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>40</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>50</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>60</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>70</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>80</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>5</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>90</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>100</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
</tbody>
</table>

Table IV. Projection performance in the DBLP scenario: *-guarded union types.

In our final experiment, we modify the target schema by increasing the number of factors of the product type `Name, ..., Proceedings` from 10 to 100. The results of our evaluation are shown in Table 10.3 and Figure 14. These results are consistent with the expected polynomial behavior (thanks to the variant of the Ford-Fulkerson algorithm we use in our system). It is interesting to observe that scalability is very good, even not as in the case of *-guarded unions.

11. RELATED WORK

Only a few works deal with the problem of mapping maintenance in data integration systems. In [Velegrakis et al. 2004] Velegrakis et al. present a framework and a semi-automatic tool (called ToMAS) for the incremental maintenance of Clio-like
mappings. The key objectives of the paper are to preserve as much as possible the semantics of the mappings to be adapted and to avoid the reformulation of the whole mapping system, so to decrease the efforts required for the maintenance. To achieve these goals, the framework adopts an incremental maintenance strategy, based on the knowledge of a detailed list of the basic update steps applied to the schemas (either the source or the target schema). Of course, this strategy can be applied only when this information is known to the mapping designer, which is unlikely in the case of autonomous data sources. This approach, hence, is best suited for integration contexts where all data sources are controlled by cooperating organizations (or the same organization at all); in this sense, this approach is complimentary to our one, which assumes that the data sources are fully autonomous and can be applied even in the absence of a detailed list of incremental updates.

It should be observed that, unlike our approach, ToMAS supports changes in both the structure and constraints of a schema; however, the correctness notion of

<table>
<thead>
<tr>
<th>No. of factors</th>
<th>Query1</th>
<th>Query2</th>
<th>Query3</th>
<th>Query4</th>
<th>Query5</th>
<th>Query6</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>20</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>30</td>
<td>&lt; 1</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>40</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>50</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>60</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>70</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>6</td>
<td>9</td>
</tr>
<tr>
<td>80</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>90</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td>10</td>
<td>14</td>
</tr>
<tr>
<td>100</td>
<td>7</td>
<td>5</td>
<td>11</td>
<td>10</td>
<td>10</td>
<td>13</td>
</tr>
</tbody>
</table>

Table V. Projection performance in the DBLP scenario: product types (times in ms).

Fig. 14. Projection performance in the DBLP scenario: product types (times in ms).
ToMAS has a coarser grain than our one, as a mapping is deemed as incorrect (and adapted) when it just works on the same fragment of the schema that has been modified.

The same correctness notion of [Velegakis et al. 2004] is used in [Yu and Popa 2005], where Hu and Popa propose an *ex-post* adaptation technique. The idea, mutated from [Melnik et al. 2003], is to create a mapping from the old version of the schema to the new version of the schema, and to compose this mapping with the existing mapping. Of course, these approaches are subject to well-known non-closure issues related to mapping composition.

Both the approaches of [Velegakis et al. 2004] and [Yu and Popa 2005] are based on the theoretical framework of Clio. Adapting our approach to support this framework is relatively easy, as mappings are expressed through correspondences and logical associations, whose output type can be easily inferred. Furthermore, this framework uses a nested-relational type system, where only labelled union types are supported; this kind of union types is less powerful than non-labelled unions (used in our type system as well as in XML Schema), and we strongly suspect that type projection can be decided in polynomial time when unions are labelled (see [Buneman and Pierce 1999] for a detailed discussion about the reasons why labeled union types are not adequate for semistructured data and XML).

An alternative technique for detecting corrupted mappings in XML data integration systems is the one described in [Colazzo and Sartiani 2005]. This technique is based on the use of a type system capable of checking the correctness of a query, in a XQuery-like language [Colazzo et al. 2004], wrt a schema, i.e., if the structural requirements of the query are matched by the schema. By relying on this type system, a distributed type-checking algorithm verifies that, at each reformulation step, the transformed query matches the target schema, and, if an error is raised, informs the source of the target peers that there is an error in the mapping.

The technique described in [Colazzo and Sartiani 2005] has two main drawbacks. First, it is not *complete*, since wrong rules that are not used for reformulating a given query cannot be discovered. Second, the algorithm requires that a query were reformulated by the system before detecting a possible error in the mapping; this implies that the algorithm cannot directly check for mapping correctness, but, instead, it checks for the correctness of a mapping wrt a given reformulation algorithm. Hence, mapping correctness is not a query-independent, semantics-based property, but is strongly related to the properties of the reformulation algorithm.

Our type system is a variation of the type systems of [Colazzo et al. 2004] and [Colazzo and Sartiani 2005], obtained by dropping error-checking in favor of a better precision in type inference. In these works we have already outlined advantages of these type systems wrt to the W3C XQuery type system [Draper et al. 2007].

Most works on mapping maintenance, in the context of data integration or data exchange systems, focus on the problem of detecting corrupted data sources *wrappers*. These approaches [Kushmerick 2000; Lerman et al. 2003] are based on checkers that learn the most prominent syntactical features of data sources, and warn the administrator when newly probed data fail in matching these features. Since they focus on syntactical changes only, these approaches are quite limited and unsatisfactory. Essentially the same approach forms the basis for the Maveric system [McCann et al. 2005], which systematically monitors the characteristics of wrappers and mappings in data integration systems. The novelty of Maveric is its improved accuracy and efficiency, but it still does not offer any correctness or completeness properties for error discovering. These approaches can be integrated with our maintenance technique, as checkers can be instructed to periodically infer the schema of
external data sources, hence allowing for a type projection checking.

Our system bears some resemblance with Spider [Chiticariu and Tan 2006]. Spider is a debugger for schema mappings based on the logical dependencies framework of Clio. Spider works by analyzing the correspondences between a source data instance and a target data instance, so to help the mapping designer in understanding the behaviour of a mapping. These correspondences are expressed by a forest of minimal routes, which link target elements with source elements (and other target elements too). Our system is not an alternative to Spider, but, instead, can be regarded as a complementary tool that can be used after a mapping has been created and successfully deployed: indeed, our tool comes into play when the integration system is running, while Spider is used before setting up the system.

There exist some similarities between our notion of type projection and the subsumption relation described in [Kuper and Siméon 2001], but these similarities are quite vague, as subtyping implies projection while the same does not hold for subsumption.

12. CONCLUSIONS

This paper presented a novel technique for detecting corrupted mappings in XML data integration systems. This technique can be used in any context where a schema mapping approach is used, and it is based on a semantic notion of mapping correctness, unrelated to the query transformation algorithms being used. This form of correctness works on the ability of a mapping to satisfy the target schema, and it is independent from queries.

We proved that mapping correctness can be reduced to type projection between the inferred result type of the mapping and the target schema, and showed that our approach is complete, i.e., all errors will be detected. To decrease false negatives, we augment the precision of type inference through type splitting.

By reducing type projection to standard subtyping among weakened types, we proved that type projection is decidable [Huynh 1985]. We characterized type projection in terms of type simulation, and, then, used the type simulation rules to define a checking algorithm. The algorithm employs a novel technique for comparing product types, based on the use of a $0-1$ maximum flow algorithm.

The equivalence between type projection and type simulation allowed us to discover some interesting properties of type projection, such as the injective nature of product types and the behavior of product and union types inside repetition types.

The use of a maximum flow algorithm for the projection of product types allowed for designing a correct and complete projection-checking algorithm with polynomial time complexity on normalized types. Since type projection can be efficiently checked, it can also be used in other contexts, like, for instance, the minimization of the amount of data loaded in main memory during XML query processing (see, for instance, [Benzaken et al. 2006]).

REFERENCES


A. PROOFS

A.1 Proofs of Section 4

**Theorem 6.3.**

\[ T^\preceq \preceq T \]

**Proof.** By induction on the structure of \( T \).

We prove the main cases only.

**Case** \( T = \llbracket T' \rrbracket \) (\( T = B \) is similar)

The case \( (\cdot) : \llbracket T^\preceq \rrbracket \) is obvious. Take \( l[f] : \llbracket T^\preceq \rrbracket \). By induction we have that there exists \( f' : T^\preceq \) such that \( f \preceq f' \), and from this the thesis easily follows.

**Case** \( T = T_1, T_2 \) (\( T = T_1 \mid T_2 \) is similar)

Take \( f_1, f_2 : T_1, T_2 \) with \( f_i : T_i \). We have \( T^\preceq = T_1^\preceq \cup T_2^\preceq \), hence by induction we have \( f_i' : T_i^\preceq \) such that \( f_i \preceq f_i' \), which implies the thesis thanks to invariance of \( \preceq \) by forest composition.

**Case** \( T = T^* \) Take \( f : T^* \). We have \( f = f_1, \ldots, f_n \), with \( f_i : T_i \), and \( T^\preceq = T^\preceq^* \). So, by induction, for each \( f_i \) there exists \( f_i' : T_i^\preceq \) such that \( f_i \preceq f_i' \), which implies the thesis thanks to invariance of \( \preceq \) by forest composition, and thanks to \( f_1', \ldots, f_n' : T^\preceq \).

\[ \Box \]

A.2 Proofs of Section 7

**Lemma 7.1** Termination of \( \text{norm}(T) \). For each type \( T \), \( \text{norm}(T) \) is computed in finite time.

**Proof.** Consider the measure \( |T| \) defined as follows:

\[
\begin{align*}
|()| &= 1 \\
|B| &= 1 \\
|T_*| &= |T| + 1 \\
|l[T]| &= |T| + 1 \\
|T, U| &= |T| + |U| + 1
\end{align*}
\]

\(|T|\) denotes the size of \( T \), i.e., the number of type terms inside \( T \). To prove termination of \( \text{norm}(T) \), it suffices to observe that: (i) if \( \text{norm}(T) = (A_1 \mid A_2) \) then \( |A_i| < |T| \), and, therefore, that (ii) when passing from the left to the right hand-side of equations in A.1, the measure \(||\) of the argument of \( \text{norm}(\cdot) \) always decreases. This implies termination, as \(|T| > 0 \) for all \( T \).

**Lemma 7.7** Upward closure. If \( T \) is prime, then \( \forall f_1, f_2 \in [T]. \exists f \in [T]. f_i \preceq f \)

**Proof.** We first observe that, thanks to the hypothesis, we can define a measure \( d^*(A) \), over types obtained by splitting, as follows:

\[
\begin{align*}
d^*(()) &= 0 \\
d^*(B) &= 0 \\
d^*(T_\ast) &= 0 \\
d^*(l[T\ast]) &= 1 + d^*(T\ast) \\
d^*(T_\ast, U\ast) &= 1 + d^*(T\ast) + d^*(U\ast)
\end{align*}
\]
Observe that $d^*(A)$ is not defined over union types, since $A$ cannot be a union type.

We then proceed by induction on $d^*(A)$.

If $d^*(A) = 0$, the cases $A = B$ and $A = ()$ are obvious. Here, the only interesting case is $A = T^*$. For this case, given $f_1$ and $f_2$ in $[A]$, we observe that their composition $f_1, f_2$ is still in $[A]$ and that $f_1 \lesssim f_1, f_2$ and $f_2 \lesssim f_1, f_2$.

If $d^*(A) > 0$ the only interesting case is $A = T', U'$. Consider $f_1$ and $f_2$ in $[T', U']$. We have

$$f_1 = f_1^1, f_1^2 \land f_1^1 \in [T'] \land f_1^2 \in [U']$$
$$f_2 = f_2^1, f_2^2 \land f_2^1 \in [T'] \land f_2^2 \in [U']$$

By induction we have that there exists $f' \in [T']$ and $f'' \in [U']$ such that

$$f_1^1, f_2^1 \lesssim f'$$
$$f_1^2, f_2^2 \lesssim f''$$

hence $f_1^1, f_2^1, f_1^2, f_2^2 \lesssim f', f''$. Since $f_1, f_2 \lesssim f_1^1, f_1^2, f_2^1, f_2^2$ by transitivity of $\lesssim$ we have that $f_1, f_2 \lesssim f', f''$. □

**Lemma 7.12.** If $T$ and $U$ are normalized and prime, if both $T$ and $U$ are product types, and $T \lesssim U$, then $T = T_1, T_2$ and $U = U_1, U_2$ with

$$(T_1 \lesssim U_1 \text{ and } T_2 \lesssim U_2) \text{ or } T \lesssim U_1$$

**Proof.** By case analysis over the form of types.

The first case concerns the situation in which both types do not contain *-types at the top level. Then, two sub-cases are possible. Assume that $T$ contains, at the top level, the base type $B$ with multiplicity $m > 0$, that is

$$T = T_1, T_2, \quad T_1 = B, \ldots, B$$

and $B \not\lesssim T_2$. Thanks to $T \lesssim U$, we have that the multiplicity of $B$ in $U$ is $n$ with $m \geq n$. So, $U = U_1, U_2$ with $T_1 \lesssim U_1$ and $B \not\lesssim U_2$: then, the proof of the sub-case is complete, since it must be $T_2 \lesssim U_2$. The second sub-case, where $T$ contains, at the top level, an element type with tag $l$ and multiplicity $n > 1$, is almost identical.

The second case we consider is the following:

$$T = T_1*, T_2 \quad U = U_1, U_2$$

where both $T_2$ and $U_2$ do not contain *-types at the top level (they are sequence of tree types)$^{11}$. For cardinality reasons, it must be $T_1* \lesssim U_1*$, otherwise the hypothesis is contradicted. Then, it is easy to prove that, for each tree type $l[A]$ (the base case is trivial), either $l[A] \lesssim U_1*$ or $l[A] \lesssim U_2$ (here the proof is identical to lemma 7.10, by observing that the tree type is prime). This automatically entails the thesis, as it is sufficient to assign to $U_2$ the tree types that are in the projection relation wrt it (in the case that each type is a projection of $U_1$, we have the second part of the thesis: $T_1*, T_2 \lesssim U_1*$).

The remaining case, where only $U$ does not contain *-types, is not possible for cardinality reasons. □

$^{11}$Recall that, by normalization, we cannot have multiple *-types at the top level.
A.3 Proofs of Section 8

Lemma 8.3. The *SimpleSim* algorithm satisfies the reflexivity, commutativity, associativity, and ()-neutrality properties.

Proof. The lemma is proved by analyzing the behavior of the algorithm.

Reflexivity:
Reflexivity is directly proved by case 2/3;

()-neutrality:
Consider $T = T_1,()$. By applying case 8/9, *SimpleSim*($T,T_1$) returns a $\otimes$ reference of the form $\text{ref}([T_1,()] \otimes \{T_1\})$, which instructs the *Sim* to build a bipartite graph $\mathcal{G}$, where $\mathcal{P}_1 = \{T_1,()\}$ and $\mathcal{P}_2 = \{T_1\}$. The maximum flow algorithm automatically discards () types during maximum flow analysis.

The case of *SimpleSim*($T_1,T$) is trivial.

Commutativity:
Consider types $T_1 = T_1,T_2$ and $U = T_2,T_1$. *SimpleSim*($T,U$) returns a $\otimes$ reference $\text{ref}([T_1,T_2] \otimes \{T_2,T_1\})$, which instructs *Sim* to build a bipartite graph $\mathcal{G}$, whose partitions are equal.

Associativity:
Consider types $T = (T_1,T_2),T_3$ and $U = T_1,(T_2,T_3)$. The algorithm just drops the parentheses from $T$ and $U$, hence the proof is trivial.

Lemma 8.4. The *SimpleSim* algorithm is compatible with the simulation relation, in the sense that it implements the simulation relation.

Proof. The lemma is proved by analyzing the correspondence between the simulation rules and the algorithm cases. A preliminary observation is that side conditions of the form $T \preceq U$ are encoded by a simple reference of the form $\text{ref}(T,U)$.

Completeness
We want to prove that each simulation rule can be encoded in the algorithm. We proceed by induction on the simulation rules.

$B \preceq B$: By reflexivity.

$(T) \preceq U$: By case 4/5.


$T_1 \preceq U_2,U_3$: Case 8/9 is applied; the algorithm returns a $\otimes$ reference of the form $\text{ref}([T_1] \otimes \{U_2,U_3\})$, which instructs *Sim* to build a bipartite graph $\mathcal{G}$ where $\mathcal{P}_1 = \{T_1\}$ and $\mathcal{P}_2 = \{U_2,U_3\}$. The thesis follows from induction on the rule side conditions.

$T_1,T_2 \preceq U_3,U_4$: As for the previous rule, case 8/9 is applied and the thesis follows from induction on the rule side conditions.

$T_1 \preceq U_2 \mid U_3$: Case 10/11 is applied, hence the algorithm returns $\text{ref}(T_1,U_2) \lor \text{ref}(T_1,U_3)$. The thesis follows from induction on the rule side conditions.

$T_1 \mid T_2 \preceq U$: Case 12/13 is applied, hence the algorithm returns $\text{ref}(T_1,U) \land \text{ref}(T_2,U)$. The thesis follows from induction on the rule side conditions.

$T \preceq U^*$: Case 18/19 is applied.

$T^* \preceq U^*$: Case 14/15 is applied.

$T_1,T_2 \preceq U^*$: Case 16/17 is applied.

Correctness
We want to prove that each algorithm case is backed by the application of the simulation rules. We proceed by induction on the algorithm cases.

Case 2/3: By reflexivity.
**Case 4/5.** By (-)-neutrality.

**Case 6/7.** By Rule 3.

**Case 8/9.** If \( n = 1 \), Rule 4 is applied. If \( n > 1 \), Rule 5, combined with the type equivalence modulo commutativity, is applied.

**Case 10/11.** If \( n = 1 \), it suffices to apply Rule 6. If \( n > 1 \), Rule 6 is applied to decompose the right hand-side of the comparison; then, Rule 5 is applied.

**Case 12/13.** Rule 7 is applied to decompose the left hand-side of the comparison; Rules 4 and 5 are then applied to the resulting terms.

**Case 14/15.** It suffices to apply Rule 9.

**Case 16/17.** Rule 10 is iteratively applied.

**Case 18/19.** Rule 8 is applied.

To conclude the proof, it should be observed that the algorithm does not contain any other case.

### A.4 Proofs of Section 9

The proof of Theorem 9.11 requires a series of other auxiliary properties that we state and prove in the following.

**Lemma A.1** Upward closure for split types. If \( \text{Split}(T) = \{T\} \), then \( \forall f_1, f_2 \in [T]. \exists f \in [T]. f_i \lesssim f \)

**Proof.** The proof is exactly the same as in Lemma 7.7.

Type-splitting type rules assume environments \( \Gamma \) to be \(*\)-guarded.

**Definition A.2** \(*\)-guarded \( \Gamma \). \( \Gamma \) is \(*\)-guarded if and only if, for every \( \chi : T \) in \( \Gamma \), \( \text{Split}(T) = \{T\} \).

We extend the \( \lesssim \) relation to substitution environments in the obvious way:

**Definition A.3.** We say that \( \rho \lesssim \rho' \) if and only if for each \( \chi \mapsto f \in \rho \) there exists \( \chi \mapsto f' \in \rho' \) such that \( f \lesssim f' \)

The \(*\)-guarded variable environments enjoy the property stated in Lemma A.4, which generalizes Lemma A.1 from typed values to typed substitutions, and is one of the crucial lemmas needed to prove the lower bound property in Theorem 9.11.

**Lemma A.4.** For any \(*\)-guarded and well-formed \( \Gamma \) and \( \rho_1, \ldots, \rho_n \in \mathcal{R}(\Gamma) \), there exists \( \rho \in \mathcal{R}(\Gamma) \) such that \( \rho_i \lesssim \rho \) for \( i = 1 \ldots n \).

**Proof.** By induction on the length of \( \Gamma \) and by Lemma A.1.

**Lemma A.5** Query Monotonicity. For any \( Q \) without \texttt{where} clauses and such that the only navigational mechanism in \( Q \) has the form \texttt{π child :: NodeTest}, it holds that:

\( \forall Q, \rho, \rho'. \rho \lesssim \rho' \Rightarrow [Q]_\rho \lesssim [Q]_{\rho'} \)

**Proof.** By case distinction and induction on the structure of \( Q \). We consider only the main cases.

**Q = for π in Q \_ return Q2.** For substitutions \( \rho \) and \( \rho' \) such that \( \rho \lesssim \rho' \), we want to prove

\( [\text{for } \pi \text{ in } Q_1 \text{ return } Q_2]_\rho \lesssim [\text{for } \pi \text{ in } Q_1 \text{ return } Q_2]_{\rho'} \)
By induction, we assume
\[ \llbracket Q_1 \rrbracket_\rho \preceq \llbracket Q_1 \rrbracket_{\rho'} \] (**) 
This means that if
\[ \llbracket Q_1 \rrbracket_\rho = t_1, \ldots, t_n \]
then
\[ \llbracket Q_1 \rrbracket_{\rho'} = t'_1, \ldots, t'_n, f \]
with \( t_i \preceq t'_i \) for \( i = 1 \ldots n \) (**).

By definition of query semantics, the property to prove can be rewritten as:
\[ \prod_{i=1}^{n} \llbracket Q_2 \rrbracket_{\rho, \pi \mapsto t_i} \preceq \prod_{i=1}^{n} \llbracket Q_2 \rrbracket_{\rho', \pi \mapsto t'_i} \]

This property easily follows by observing that by (**) we have \( (\rho, \pi \mapsto t_i) \preceq (\rho', \pi \mapsto t'_i) \), and therefore, by induction, we have:
\[ \prod_{i=1}^{n} \llbracket Q_2 \rrbracket_{\rho, \pi \mapsto t_i} \preceq \prod_{i=1}^{n} \llbracket Q_2 \rrbracket_{\rho', \pi \mapsto t'_i} \]
which entails the thesis by definition of \( \preceq \) itself.

Query monotonicity has the following corollary, which implies monotonicity of substitution extension.

**Corollary A.6.** Given a well-formed query \( Q \) and a substitution \( \rho \) such that \( \text{FV}(Q) \subseteq \text{dom}(\rho) \cup \{\chi\} \):
\[ f_1 \preceq f_2 \Rightarrow \prod_{t \in \text{trees}(f_1)} \llbracket Q \rrbracket_{\rho, \pi \mapsto t} \preceq \prod_{t \in \text{trees}(f_2)} \llbracket Q \rrbracket_{\rho, \pi \mapsto t \chi} \]

**Proof.** By Lemma A.5.

**Lemma A.7 Invariance of Well-Formation.** For any well-formed judgement \( \Gamma \vdash Q : U \) with \( \Gamma \) *-guarded, the backward application of the rules produces judgements that are well-formed as well, and containing *-guarded environments.

**Proof.** It directly follows by the way rules \((\text{TypeInElSplit})\) and \((\text{TypeLetSplit})\) are defined.

As anticipated, thanks to the *-guardedness restriction, the type-splitting system is complete up to forest simulation \( \preceq \). Completeness is proved in Theorem 9.11 and follows from the next lemma, where we first consider the special case where \( \Gamma \) is *-guarded.

**Lemma A.8.** In the type-splitting system, for each \( \Gamma \) *-guarded and well-formed, and query \( Q \) without where clauses and such that the only navigational mechanism in it has the form \( \pi \text{ child :: NodeTest} \) with NodeTest a tag label \( l \), it holds that:
\[ \Gamma \vdash Q : U \Rightarrow \forall f \in [U]. \exists \rho \in \mathcal{R}(E, \Gamma). f \preceq \llbracket Q \rrbracket_\rho \]
Proof. We prove the following statements:

- \( \forall f \in [U]. \exists \rho \in \mathcal{R}(E, \Gamma). \Gamma \vdash Q : U \Rightarrow f \preceq [Q]_{\rho} \)

- \( \forall f \in [U]. \exists \rho \in \mathcal{R}(E, \Gamma). \exists f' \in [T]. \Gamma \vdash \text{\texttt{in}} T \rightarrow Q : U \Rightarrow f \preceq \prod_{t \in \text{trees}(f')} [Q]_{\rho, x \mapsto t} \)

We proceed by induction on the proof tree of the proved judgement and by cases on the last rule applied. We only prove main cases.

(typeletsplitting). We have \( \Gamma \vdash \text{\texttt{let}} x := Q_1 \text{\texttt{return}} Q_2 : U \) and

\[
\begin{align*}
\Gamma & \vdash Q_1 : T_1 \quad (1) \\
& \text{Split}(T_1) = \{A_1, \ldots, A_n\} \quad (2) \\
& \Gamma, x : A_i \vdash Q_2 : U_i \quad i = 1 \ldots n \quad (3) \\
& U = U_1 \ldots U_n \quad (4) \\
& \forall f' \in [T_1]. \exists \rho \in \mathcal{R}(E, \Gamma). f \preceq [Q_1]_{\rho} \quad (5) \\
& \forall f' \in [U_1]. \exists \rho \in \mathcal{R}(E, (\Gamma, x : A_i)). f \preceq [Q_2]_{\rho} \quad i = 1 \ldots n \quad (6)
\end{align*}
\]

We want to prove

\( \forall f \in [U]. \exists \rho \in \mathcal{R}(E, \Gamma). f \preceq [\text{\texttt{let}} x := Q_1 \text{\texttt{return}} Q_2]_{\rho} \)

For any \( f \in [U] \), by (4) we have that \( f \in [U_i] \) for some \( i = 1 \ldots n \). Moreover, by (6):

\( (\exists \rho^2 \in \mathcal{R}(E, (\Gamma, x : A_i)). f \preceq [Q_2]_{\rho^2}) \) (7)

Since \( \rho^2 \in \mathcal{R}(E, (\Gamma, x : A_i)) \), we have

\( \rho^2 = \overline{\rho}'_{\rho, x \mapsto f'} \) (8)

with \( f' \in [A_i] \) and \( \overline{\rho}' \in \mathcal{R}(E, \Gamma) \).

Now, since \( f' \in [A_i] \Rightarrow f' \in [T] \) (Lemma 9.4), and by (5) we have that:

\( \exists \rho^1 \in \mathcal{R}(E, \Gamma). f' \preceq [Q_1]_{\rho^1} \) (9)

Hence, (7) and (8) imply that

\( f \preceq [Q_2]_{\rho^2} = [Q_2]_{\overline{\rho}', x \mapsto f'} \) (10)

while (9) and (10) and Lemma A.5 imply that

\( f \preceq [Q_2]_{\overline{\rho}', x \mapsto [Q_1]_{\rho^1}} \)

Now, by Lemma A.4 there exists \( \rho \in \mathcal{R}(E, \Gamma) \) such that \( \rho^1 \preceq \rho \) and \( \overline{\rho}^2 \preceq \rho \), hence, by Lemma A.5, we have:

\( f \preceq [Q_2]_{\overline{\rho}', x \mapsto [Q_1]_{\rho}} \preceq [Q_2]_{\rho, x \mapsto [Q_1]_{\rho}} \)

By \( \text{\texttt{let}} x := Q_1 \text{\texttt{return}} Q_2 \) \( \rho = [Q_2]_{\rho, x \mapsto [Q_1]_{\rho}} \), the case is proved.

(typefor). We have \( \Gamma \vdash \text{\texttt{for}} \Gamma \in \text{\texttt{Q1 return Q2 : T2}} \) and the following hypothesis:

\[
\begin{align*}
\Gamma & \vdash Q_1 : T_1 \quad (1) \\
\Gamma & \vdash \text{\texttt{for}} \text{\texttt{in}} T_1 \rightarrow Q_2 : T_2 \quad (2) \\
& \forall f \in [T_1]. \exists \rho \in \mathcal{R}(E, \Gamma). f \preceq [Q_1]_{\rho} \quad (3) \\
& \forall f \in [T_2]. \exists \rho \in \mathcal{R}(E, \Gamma). \exists f' \in [T_1]. f \preceq \prod_{t \in \text{trees}(f')} [Q_2]_{\rho, x \mapsto t} \quad (4)
\end{align*}
\]
We want to prove that
\[
\forall f \in [T_2]. \exists \rho \in \mathcal{R}(E, \Gamma). \\
    f \preceq [\text{for } \mathfrak{F} \text{ in } Q_1 \text{ return } Q_2]_\rho
\]
For any \( f \in [T_2] \), by (4) we have
\[
\exists \rho^2 \in \mathcal{R}(E, \Gamma). \exists f' \in [T_1]. f \preceq \prod_{t \in \text{trees}(f')} [Q_2]_{\rho^2, \mathfrak{F}} \tag{5}
\]
Since \( f' \in [T_1] \), by (3) we have:
\[
(\exists \rho^1 \in \mathcal{R}(E, \Gamma). f' \preceq [Q_1]_{\rho^1}) \tag{6}
\]
From (5) and (6) and Corollary A.6 it follows
\[
f \preceq \prod_{t \in \text{trees}(f')} [Q_2]_{\rho^2, \mathfrak{F}} \preceq \prod_{t \in \text{trees}([Q_1]_{\rho^1})} [Q_2]_{\rho^2, \mathfrak{F}}
\]
As in the previous case, by Lemma A.4 there exists \( \rho \in \mathcal{R}(E, \Gamma) \) such that \( \rho^1 \preceq \rho \), \( \rho^2 \preceq \rho \). Therefore, by Lemma A.5 and Corollary A.6, we have:
\[
f \preceq \prod_{t \in \text{trees}([Q_1]_{\rho^1})} [Q_2]_{\rho^2, \mathfrak{F}} \preceq \prod_{t \in \text{trees}([Q_1]_{\rho^1})} [Q_2]_{\rho, \mathfrak{F}}
\]
By \( [\text{for } \mathfrak{F} \text{ in } Q_1 \text{ return } Q_2]_\rho = \prod_{t \in \text{trees}([Q_1]_{\rho^1})} [Q_2]_{\rho, \mathfrak{F}} \) the case is proved.

TYPEINSTAR. We have \( \Gamma \vdash \mathfrak{F} \text{ in } T^* \rightarrow Q : U^* \) and the following hypothesis:
\[
\Gamma \vdash \mathfrak{F} \text{ in } T \rightarrow Q : U
\]
We want to prove that:
\[
(\forall f \in [U^*]. \exists \rho \in \mathcal{R}(E, \Gamma). \exists f' \in [T^*]. f \preceq \prod_{t \in \text{trees}(f')} [Q]_{\rho, \mathfrak{F}})
\]
Consider \( f \in [U^*] \); this entails that \( f = f_1, \ldots, f_n \) with \( f_i \in [U] \), for \( i = 1, \ldots, n \). For each \( f_i \), by induction on \( \Gamma \vdash \mathfrak{F} \text{ in } T \rightarrow Q : U \), we have
\[
\exists \rho^i \in \mathcal{R}(E, \Gamma). \exists f'_i \in [T]. f_i \preceq \prod_{t \in \text{trees}(f'_i)} [Q]_{\rho^i, \mathfrak{F}}
\]
By Lemma A.4, there exists \( \rho \in \mathcal{R}(E, \Gamma) \) such that \( \rho^i \preceq \rho \), for \( i = 1, \ldots, n \). Hence, by Lemma A.5:
\[
f_i \preceq \prod_{t \in \text{trees}(f'_i)} [Q]_{\rho^i, \mathfrak{F}} \preceq \prod_{t \in \text{trees}(f'_i)} [Q]_{\rho, \mathfrak{F}}
\]
Therefore we have:
\[
f = f_1, \ldots, f_n \preceq \prod_{t \in \text{trees}(f'_1)} [Q]_{\rho, \mathfrak{F}} \times \cdots \times \prod_{t \in \text{trees}(f'_n)} [Q]_{\rho, \mathfrak{F}}
\]
and the case is proved by observing that \( f'_1, \ldots, f'_n \in [T^*] \) and that
\[
\prod_{t \in \text{trees}(f'_1)} [Q]_{\rho, \mathfrak{F}} \times \cdots \times \prod_{t \in \text{trees}(f'_n)} [Q]_{\rho, \mathfrak{F}} = \prod_{t \in \text{trees}(f'_1, \ldots, f'_n)} [Q]_{\rho, \mathfrak{F}}
\]
(TypeChild). It follows by Lemma 9.3.

\[\square\]

**Theorem 9.11 [Lower Bound wrt Projection].** For any well-formed \( \Gamma \) and query \( Q \) without \texttt{where} clauses and such that the only navigational mechanism used in \( Q \) has the form \( \exists \text{child} :: \text{NodeTest} \), with \text{NodeTest} being an element label \( l \), it holds that:

\[
\Gamma \vdash Q : U \Rightarrow \forall f \in \llbracket U \rrbracket. \exists \rho \in \mathcal{R}(\Gamma). f \preccurlyeq \llbracket Q \rrbracket_\rho
\]

**Proof.** By hypothesis we have \( \Gamma \vdash Q : U \), that is

1. \( \text{SplitVEnv}(\Gamma) = \{ \Gamma_1, \ldots, \Gamma_n \} \)
2. \( \Gamma_i \vdash Q : U_i \quad i = 1 \ldots n \)
3. \( U = U_1 \mid \ldots \mid U_n \)

Therefore, for each \( f \in \llbracket U \rrbracket \), there exists \( U_i \) and \( \Gamma_i \)-guarded such that \( f \in \llbracket U_i \rrbracket \) and \( \Gamma_i \vdash Q : U_i \). Hence, by Lemma A.8 we have

\[
\exists \rho \in \mathcal{R}(\Gamma_i). f \preccurlyeq \llbracket Q \rrbracket_\rho
\]

Now, the thesis follows from \( \mathcal{R}(E, \Gamma_i) \subseteq \mathcal{R}(E, \Gamma) \) (Lemma 9.6). \( \square \)

...