

# An Efficient Algorithm for XML Type Projection

Dario Colazzo

Laboratoire de Recherche en Informatique (LRI)  
Bat 490 - Université Paris Sud - 91405 Orsay Cedex  
France  
dario.colazzo@lri.fr

Carlo Sartiani

Dipartimento di Informatica - Università di Pisa  
Largo B. Pontecorvo 3 - 56127 - Pisa - Italy  
sartiani@di.unipi.it

## Abstract

In the contexts of data integration and data exchange, *schema mappings* are primarily used for query answering. As a consequence, their maintenance and, in particular, the detection of *corrupted* mappings, i.e., mappings that fail in matching the source and/or the target schema, is crucial.

Corruption checking can be automatically performed by relying on an operation called *type projection*. This work describes an efficient algorithm for checking XML type projection, based on a characterization of type projection in terms of *type simulation*.

**Categories and Subject Descriptors** H.2.3 [Languages]: Database (persistent) programming languages

**General Terms** Languages

## 1. Introduction

XML is an universal data format that can be used to represent any kind of data sources, from strongly structured data (e.g., relational data) to *semistructured* or even *unstructured* data. This property made XML a natural medium for integrating heterogeneous data sources, both in a centralized fashion [16] and in a *decentralized* way (e.g., p2p data management systems like Piazza [14, 22, 13]).

One of the most important problems in data integration systems (both centralized and decentralized) is the *maintenance* of *mappings*. Mappings are dependencies among schemas, that are used during query answering for reformulating queries or, as in data exchange systems [1], for generating canonical solutions. Since a mapping  $m$  from  $\mathcal{S}_i$  to  $\mathcal{S}_j$  exploits the structural properties of both  $\mathcal{S}_i$  and  $\mathcal{S}_j$ , a sudden change in one of the schemas, let's say  $\mathcal{S}_j$ , may corrupt the mapping  $m$ , so that the mapping rules of  $m$  are no longer true. Mapping corruption has a deep impact on query answering, and essentially prevents the system from generating (useful) query results.

Mapping maintenance is a time-consuming and expensive activity, and is usually performed by the system/site administrator, which manually inspects schemas and mappings. In [6] we proposed a mapping maintenance technique based on two key ideas: checking for correctness on both the source and the target schema, and using a type projection notion to make correctness checking on the target schema operational. By checking for correctness on both the source and the target schema, a system based on that technique can capture rules referring to non-existing (or no more existing) fragments of the source schema as well as capture rules that do not *match* the target schema. Type projection refines this *match* in

terms of projection, so to capture the intuition of mapping as transformation + projection.

A system based on the technique of [6] periodically performs, in the background, maintenance checks on the source and target schemas of mappings<sup>1</sup>. Since each check on the target schema implies a type projection check, a crucial and key issue of this technique is the availability of an efficient algorithm for type projection checking.

**Our Contribution** In this paper we study a type projection operation for *unordered* XML types. In particular, we provide:

- a formal definition of type projection;
- a characterization of type projection in terms of *type simulation*;
- a comparison between type projection and subtyping among unordered types;
- and, an efficient algorithm for checking type projection.

All these contributions are relevant. From a theoretical point of view, the characterization of type projection in terms of type simulation allows for a better understanding of the properties of type projection, and, in particular, of its relationship with subtyping; this characterization, hence, is very important since type projection can be regarded, in the data integration and data exchange contexts, as the counterpart of subtyping in transformation analysis. Furthermore, this study provides a first hint toward the analysis of the computational complexity of subtype-checking among unordered XML types, which has not yet been extensively studied.

From a practical point of view, the algorithm makes the technique described in [6] a viable option for automatic mapping maintenance in p2p Piazza-like database systems. Furthermore, most of the techniques used in this algorithm can be extended to subtype-checking algorithms, so to extend the class of tractable cases.

As final remark, the solution proposed in this paper can be used in both p2p and centralized data integration systems, as well as in data exchange systems ([11]). Data integration and data exchange are the main application fields of XML type projection. Still, this notion is very general and it can be used in other contexts, as for instance, the minimization of the amount of XML data loaded into main memory during XML query processing.

**Paper Outline** The paper is structured as follows. Sections 2 and 3 describe the background of our research as well as the data model and type language being used. Section 4, then, introduces the type projection notion. Section 5, next, illustrates the type simulation relation, discusses its properties, and shows its equivalence to type projection. Section 6, then, describes the type projection checking algorithm and discusses its complexity. Finally, Section 7 analyzes some related work, while Section 8 concludes.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PPDP'06 July 10–12, 2006, Venice, Italy.  
Copyright © 2006 ACM 1-59593-388-3/06/0007...\$5.00.

<sup>1</sup>A quite efficient and precise type inference algorithm for the core of XQuery used for Piazza-like p2p mappings is defined in [5] [4].

## 2. Motivation

As stated in the Introduction, we defined type projection to formalize the (very practical) problem of checking whether a mapping from a schema  $S_i$  into a schema  $S_j$  respects the structural properties of  $S_j$ . To this aim, we embraced the Piazza [14] characterization of schema mappings as “transformation + projection”. 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, given in the context of the Piazza system, is common to most data integration and data exchange systems, and points out that a schema mapping is a *non-functional* transformation from a source schema  $S$  to a target schema  $T$ . Indeed, as a transformed data instance  $I_T$  may not contain all attributes/elements specified by the target schema  $T$ ,  $I_T$  may not be an instance of  $T$ . The following example clarifies this issue.

**Example 2.1** Consider a p2p 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<sup>2</sup>.

```
CupMDB = mySongs[(Song)*]
Song = song[Title, Artist, Album, MyRating]
Title = title[String]
Artist = artist[String]
Album = album[String]
MyRating = myRating[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.

```
SeattleMDB = musicDB[Artist*]
Artist = artist[Name, Provenance, Track*]
Name = name[String]
Provenance = provenance[Continent, Country]
Continent = continent[String]
Country = country[String]
Track = track[Title, Year, Genre]
Title = title[String]
Year = year[Integer]
Genre = genre[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.

```
SeattleMDB <-
Q1($input): for $t in $input/title
return $t
Q2($input): for $a in $input//artist,
return artist[
```

```
name[$a/data()],
for $s in $input//song,
$aart in $s/artist
where $aart/data() = $a/data()
return track[Q1($s)]
```

```
Q3($input): for $db in /mySongs
return musicDB[Q2($db)]
```

This mapping transforms data conforming to a fragment of the Cupertino schema (album and myRating elements are discarded) into data conforming to a fraction of the Seattle-based schema. 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 album and 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 a projection of a Seattle-compliant data instance. ■

This example highlights that the relationship between a schema mapping and its target schema cannot be modeled through a standard subtyping relation *ala* 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 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 impose a more flexible and general way of comparing types than a *subtyping-based* comparison. This is main motivation for the introduction of *type projection*, which captures the Piazza intuition of mappings as “transformation + projection” (i.e., non-functional transformation). 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 transform a source data instance  $I_S : S_i$  into a *fragment* of one or more data instances  $I_T$  conforming to  $S_j$ .

While in [7] we grounded on an XQuery-like mapping language, in the spirit of Piazza, in this work we will extend our approach to any mapping language, like that of [1], provided that an output type for any given mapping can be inferred.

## 3. Data Model and Type Language

We represent an XML document as an *unranked, unordered*, node-labeled tree, as shown by the following grammar.

$$f ::= () \mid b \mid l[f] \mid f, \dots, f$$

$f$  is a forest that may comprise the empty sequence  $()$ , base values  $(b)$ , element nodes  $(l[f])$ , as well as the concatenation of other forests  $(f, \dots, f)$ .

Since our study started from a p2p perspective, we drop ordering from the model, as no global order can be enforced on data coming from multiple sources.<sup>3</sup> Hence, concatenation  $(f, \dots, f)$  is commutative, associative, and has  $()$  as neutral element.

On data model instances we define the following *value projection* relation.

**Definition 3.1 (Value projection)** *The value projection relation  $\lesssim$  is the minimal relation such that:*

$$\begin{array}{l} () \lesssim f \\ f_1, f_2 \lesssim f_3, f_4 \quad \text{if } (f_1 \lesssim f_3 \wedge f_2 \lesssim f_4) \\ b_1 \lesssim b_2 \\ f_1 \lesssim f_3 \quad \text{if } \exists f_2 : f_1 \lesssim f_2 \wedge f_2 \lesssim f_3 \\ f \lesssim f, () \\ l[f_1] \lesssim l[f_2] \quad \text{if } f_1 \lesssim f_2 \\ f_1, f_2 \lesssim f_2, f_1 \end{array}$$

$\lesssim$  is an *injective* simulation relation among values, inspired by the projection operator of the relation data model. Intuitively,

<sup>2</sup>For the sake of simplicity, we are using here the XDuce [15] type language.

<sup>3</sup>For the same reason, we do not explicitly represent attributes, as they can be easily encoded as element nodes.

<b>Types</b>	$T$	::=	$()$	empty sequence
			$B$	base type
			$l[T]$	element type
			$T, T$	sequence type
			$T \mid T$	union type
		$T^*$	repetition type	
<b>Base Type</b>	$B$	::=	String	

Figure 1. Type language

$d_1 \lesssim d_2$  if there exists a subterm  $d_3$  in  $d_2$  such that  $d_3$  matches  $d_1$ ; 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. This notion of projection for XML trees is a generalization of that introduced in [20], where leaf values are taken into account too.

Our type language, based on XDuce [15] and XQuery [10] type languages, is shown in Figure 1, 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 usual heuristics and optimizations based on type ordering cannot be applied in this context. Furthermore, our type language does not include vertical recursive types; this is motivated by the fact that most mapping languages are not powerful enough to transform trees with arbitrary depth, hence we can restrict the type language to types that describe trees with limited and finite depth. This is a significant difference from XDuce and XQuery type languages, where types are ordered and vertical recursion is allowed.

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

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

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

Subtyping is defined via type semantics:  $T < U \iff \llbracket T \rrbracket \subseteq \llbracket U \rrbracket$ .

The following 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  $U$ :

$$\llbracket T^*, U^* \rrbracket = \llbracket (T \mid U)^* \rrbracket$$

## 4. Type Projection

Type projection is a generalization to types of the value projection relation, as shown by Definition 4.1.

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

As for the value projection relation, the type projection relation is semantics, 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  $d_2$ .

Notice that this formalization perfectly fits the informal description from [14] that we quoted in Section 2.

The decidability of type projection comes from the main result proved in [6], that relates type projection checking and subtype-checking over *unordered* types, whose decidability has been proved in [9]. For the sake of convenience, we report here this result.

**Theorem 4.2 (Type projection as sub-typing)**

$$T \lesssim U \iff T < U^\triangleleft$$

where  $U^\triangleleft$  is defined as shown in Definition 4.3.

**Definition 4.3 (Type approximation)** Given a type  $U$ , we indicate with  $U^\triangleleft$  the type obtained by  $U$  just by replacing each subexpression  $U'$ , corresponding to a tree type  $l[\_]$  or  $B$ , with  $U'?$  (that is  $(U' \mid ())$ ). Formally

$$\begin{aligned}
()^\triangleleft &\triangleq () & T \mid U^\triangleleft &\triangleq T^\triangleleft \mid U^\triangleleft \\
l[T]^\triangleleft &\triangleq l[T^\triangleleft]? & B^\triangleleft &\triangleq B? \\
T, U^\triangleleft &\triangleq T^\triangleleft, U^\triangleleft & T^*{}^\triangleleft &\triangleq T^*{}^\triangleleft
\end{aligned}$$

Theorem 4.2 states that type projection is equivalent to subtyping, once the right hand-side of the comparison has been approximated. Hence, a subtype-checking algorithm, combined with a type approximation algorithm, can be used to check for type projection. This equivalence gives rise to a fundamental question, i.e., whether type projection can be replaced, in the contexts of data integration and data exchange, by subtyping. A strong reason for discarding subtyping, in the contexts of data integration and data exchange, in favor of type projection, is its algorithmic complexity. In [9], authors gave a *super-exponential* upper bound for the complexity of subtyping for a superset of our language, while some optimizations have been implemented in XDuce, CDuce and XQuery only for ordered types [15, 2, 10]. On the contrary, as shown in Section 6.3, type projection has an exponential upper-bound, but it can be checked, in most cases, in polynomial time, hence it can be adopted as the core of systematic way of checking for mapping correctness.

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

## 5. Type Simulation

This Section introduces type simulation, discusses its main properties, and shows its equivalence to the type projection relation we described in the previous Section.

### 5.1 Definition

*Type simulation* is a *symbolic* relation among type equivalence classes (i.e., types are identified modulo commutativity, associativity, and  $()$ -neutrality), whose main aim is to provide a convenient way to characterize and check for type projection. Indeed, the previous Section 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  $norm(T)$ , defined as shown in Table 5.1. The following lemma proves that the evaluation of  $norm(T)$  always terminates.

**Lemma 5.1 (Termination of  $norm(T)$ )** For each type  $T$ ,  $norm(T)$  is computed in finite time.

**Proof.** Consider the measure  $|T|$  defined as follows:

$$\begin{aligned} |()| &= 1 \\ |B| &= 1 \\ |T^*| &= |T| + 1 \\ |l[T]| &= |T| + 1 \\ |T \mid U| &= |T| + |U| + 1 \\ |T, U| &= |T| + |U| + 1 \end{aligned}$$

$|T|$  denotes the *size* of  $T$ , i.e., the number of type terms inside  $T$ . To prove termination of  $norm(T)$ , it suffices to observe that: (i) if  $norm(T) = (A_1 \mid A_2)$  then  $|A_i| < |T|$ , and, therefore, that (ii) when passing from the left to the right hand-side of equations in 5.1, the measure  $| \cdot |$  of the argument of  $norm(\cdot)$  always decreases. This implies termination, as  $|T| > 0$  for all  $T$ . ■

$norm()$  works by transforming types, while preserving their semantics (Lemma 5.3), so that the transformed types can be easily compared by the simulation relation (and by the corresponding algorithm). For instance,  $norm(T^*, U^*, U)$  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  $norm()$  must be applied in the order in which they are defined.  $norm()$  can be applied to any type to obtain a corresponding normalized 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.

$norm()$  is essentially equivalent to the distribution of  $\wedge/\vee$ , hence its computational complexity is in EXPTIME. Despite this upper bound, for a vast class of types  $norm()$  can be computed in PTIME. This class contains types where unions are always guarded by a \*-operator (\*-guarded types).

**Definition 5.2 (SGT)** A type  $T$  is in SGT (star-guarded types) if it can be generated by the following grammar:

$$\begin{aligned} \text{*Types} \quad T &::= () \mid B \mid l[T] \mid T, T \mid U^* \\ \text{Union Types} \quad U &::= T \mid T \mid U \end{aligned}$$

Proving that for \*-guarded types  $norm()$  is polynomial is straightforward. \*-guardedness is a property enjoyed by a large number of commonly used DTDs and XML schemas. For instance, the reader can refer to [3] for a detailed classification of real world DTDs: this classification shows that \*-unguarded unions are quite infrequent. Furthermore, as in [5], we also verified, by looking at repositories of schemas over the Web, that product and nested types generally contains very few \*-unguarded unions, so that the third and sixth line of the  $norm()$  definition in 5.1 do not compromise the  $norm()$  evaluation cost.

The following lemma proves that  $norm()$  preserves the semantics of types.

**Lemma 5.3** For each type  $T$ ,  $\llbracket T \rrbracket = \llbracket norm(T) \rrbracket$ .

In the following, we will say that a type  $T$  is *prime* if and only if  $norm(T) = T$  and  $T \neq A \mid B$ . Moreover, in the following 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.<sup>4</sup>

**Lemma 5.4** If  $T^*$  and  $U$  are prime, then  $T^* \lesssim U \Leftrightarrow U = (U_1)^*, A$  and  $T^* \lesssim (U_1)^*$ .

<sup>4</sup> Recall that each prime type can have at most one \*-type at the top level.

**Definition 5.5 (Type simulation)** The type simulation relation  $\preceq$  among normalized types is defined as follows.

$$\begin{aligned} 1) \quad B &\preceq B \\ 2) \quad () &\preceq U \\ 3) \quad l[T] &\preceq l[U] \quad \text{if } T \preceq U \\ 4) \quad T_1 &\preceq U_2, U_3 \quad \text{if } T_1 \preceq U_2 \vee T_1 \preceq U_3 \\ 5) \quad T_1, T_2 &\preceq U_3, U_4 \quad \text{if } (T_1 \preceq U_3 \wedge T_2 \preceq U_4) \\ 6) \quad T_1 &\preceq U_2 \mid U_3 \quad \text{if } T_1 \preceq U_2 \vee T_1 \preceq U_3 \\ &\quad \text{and } T_1 \neq V_1 \mid V_2 \\ 7) \quad T_1 \mid T_2 &\preceq U \quad \text{if } T_1 \preceq U \wedge T_2 \preceq U \\ 8) \quad T &\preceq U^* \quad \text{if } T \preceq U \\ 9) \quad T^* &\preceq U^* \quad \text{if } T \preceq U^* \\ 10) \quad T_1, T_2 &\preceq U^* \quad \text{if } T_1 \preceq U^* \wedge T_2 \preceq U^* \end{aligned}$$

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 5.6 (Upward closure)** If  $T$  is prime then  $\forall f_1, f_2 \in \llbracket T \rrbracket, \exists f. f_i \lesssim f$

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

$$\begin{aligned} d^*(()) &= 0 \\ d^*(B) &= 0 \\ d^*(T^*) &= 0 \\ d^*(l[T']) &= 1 + d^*(T') \\ d^*(T', U') &= 1 + d^*(T') + d^*(U') \end{aligned}$$

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  $\llbracket A \rrbracket$ , we observe that their composition  $f_1, f_2$  still is in  $\llbracket A \rrbracket$  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  $\llbracket T', U' \rrbracket$ . We have

$$\begin{aligned} f_1 &= f_1^1, f_1^2 \wedge f_1^1 \in \llbracket T' \rrbracket \wedge f_1^2 \in \llbracket U' \rrbracket \\ f_2 &= f_2^1, f_2^2 \wedge f_2^1 \in \llbracket T' \rrbracket \wedge f_2^2 \in \llbracket U' \rrbracket \end{aligned}$$

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

$$\begin{aligned} f_1^1, f_1^2 &\lesssim f' \\ f_2^1, f_2^2 &\lesssim f'' \end{aligned}$$

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

**Lemma 5.7 (Accumulation)** If  $T \lesssim T_1 \mid T_2$  and  $T$  is prime, then  $T \lesssim T_1$  or  $T \lesssim T_2$ .

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

$$P(U, C) = \{f : U \mid \exists f' : C. f \lesssim f'\}$$

Table 5.1.  $norm()$  function.

$norm()$	$\triangleq$	$()$
$norm(B)$	$\triangleq$	$B$
$norm(l[T])$	$\triangleq$	$\begin{cases} \cup l[A_i] & \text{if } norm(T) = A_1 \mid \dots \mid A_n \\ l[norm(T)] & \text{otherwise} \end{cases}$
$norm(T \mid U)$	$\triangleq$	$norm(T) \mid norm(U)$
$norm(T'*, U'*, U)$	$\triangleq$	$norm((T' \mid U')*, U)$
$norm(T, U)$	$\triangleq$	$\begin{cases} norm(A_1, U) \mid norm(A_2, U) & \text{if } norm(T) = (A_1 \mid A_2) \\ norm(T, A_1) \mid norm(T, A_2) & \text{if } norm(U) = (A_1 \mid A_2) \\ norm(T), norm(U) & \text{otherwise} \end{cases}$
$norm(T*)$	$\triangleq$	$norm(T)*$

we have

$$P(T, T_1) \subset \llbracket T \rrbracket, P(T, T_2) \subset \llbracket T \rrbracket, P(T, T_1) \cup P(T, T_2) = \llbracket T \rrbracket$$

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 5.6 we have that there exists  $f : T$  such that  $f_1 \lesssim f$  and  $f_2 \lesssim f$ . 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  $\lesssim$  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  $f \lesssim f''$ . According to what obtained before, we have  $f_2 \lesssim f''$ , that is  $f_2 \in P(T, T_1)$ , and, as a consequence,  $f_2 \notin P(T, T_1) \setminus P(T, T_2)$ , which is a contradiction. ■

A similar lemma (Lemma 5.9) deals with the product of prime types; to prove the lemma, the following corollary is necessary.

**Corollary 5.8** *If  $T$  is a prime tree type such that  $T \lesssim U_1, U_2$ , then  $T \lesssim U_1$  or  $T \lesssim U_2$ .*

**Proof.** It suffice to notice that since  $T$  is a prime tree type, we have

$$T \lesssim U_1, U_2 \Leftrightarrow T \lesssim U_1 \mid U_2$$

and then apply Lemma 5.7. ■

**Lemma 5.9** *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  $n > 0$ , that is

$$T = T_1, T_2, T_1 = B, \dots, B$$

and  $B \not\lesssim T_2$ . Thanks to  $T \lesssim U$ , we have that the multiplicity of  $B$  in  $U$  is  $m \geq n$ . 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)<sup>5</sup>. 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 type case is trivial), either  $l[A] \lesssim U_1*$  or  $l[A] \lesssim U_2$  (here the proof is identical to lemma 5.7, 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 third case is

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

where both  $T_1$  and  $U_1$  does not contain  $*$ -types at the top level. Here it is sufficient to proceed as before.

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

## 5.2 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, 5.4, 5.6, 5.7, and 5.9.

**Theorem 5.10** *Given two normalized types  $T$  and  $U$ :*

$$T \lesssim U \Leftrightarrow T \preceq U$$

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

$T = ()$  Obvious.

$T = B$  Assume that  $U = U_1 \mid \dots \mid U_n$ , with  $U_i$  prime for  $i = 1 \dots n$ . Thanks to  $B \lesssim U$ , there exists  $j \in 1 \dots n$  such that  $B \lesssim U_j$  (Lemma 5.7); 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  $U = U_1, \dots, U_n$ . Therefore, it must exist  $j \in 1, \dots, n$  such that  $B \lesssim U_j$ , otherwise the hypothesis is contradicted, hence the thesis follows by induction on the structure of  $U$ .

$T = l[T_1]$  Similar to the previous case.

$T = T_1 \mid T_2$  In this case, we have both  $T_1 \lesssim U$  and  $T_2 \lesssim U$ , 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  $\preceq$  definition.

$T = T_1, T_2$  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  $U = U_1 \mid \dots \mid U_n$ ,

<sup>5</sup> Recall that, by normalization, we cannot have multiple  $*$ -types at the top level.

then we can observe that  $T$  is prime, hence Lemma 5.7 implies that  $\exists j \in 1, \dots, n$  such that  $T \lesssim U_j$ . 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 5.9 so to obtain that  $U = C, D$  and  $T_1 \lesssim C$  and  $T_2 \lesssim D$ . 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  $T_1 \lesssim U$  and  $T_2 \lesssim U$ . Therefore, the thesis follows by induction and by Rule 4 of the type simulation relation definition.

$T = T'*$  Not difficult thanks to Lemma 5.4. Indeed, by this lemma we have that  $U = U'*, U''$  and  $T' * \lesssim U'*$ . At this point it is sufficient to observe that

$$T' * \lesssim U' * \Leftrightarrow T' \lesssim U' *$$

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  $T' * \preceq U'*, U''$  holds if  $T' * \lesssim U'*$  does. But this can be proved by Rule 9, thanks to induction, by using  $T' \lesssim U'*$  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.  $\blacksquare$

## 6. 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  $l[m[T]]$  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* (*simTypes* 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,  $Z_1, \dots, Z_n \preceq V_1, \dots, V_m$  if and only if each type  $Z_i$  can be mapped into a distinct type in  $V_1, \dots, V_m$ , 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  $V_h$ . 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  $O(\binom{m}{n})$ .

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  $\mathcal{G}$ , whose first partition  $\mathcal{P}_1$  contains one node per each  $Z_i$  type, and whose second partition  $\mathcal{P}_2$  contains one node per each  $V_j$  type; nodes in  $\mathcal{P}_1$  are connected to a source  $s$ , while nodes in  $\mathcal{P}_2$  are connected to a sink  $t$ .  $\mathcal{P}_1$  and  $\mathcal{P}_2$  are connected together through edges satisfying the simulation relation, i.e., an

edge from  $Z_i$  to  $V_j$  is inserted in  $\mathcal{G}$  if  $Z_i \preceq V_j$ . Each edge has two possible values for its flow: 0 and 1. The source  $s$  emits a flows of  $n$  units, so  $Z_1, \dots, Z_n$  simulates  $V_1, \dots, V_m$  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  $O((n + m)^3)$  [12]. A similar technique is also used in [8] for subtype-checking of product types in a rather restrictive type language.

**Remark 6.1** 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.  $\blacksquare$

### 6.1 Type Simulation Algorithm

Our algorithm for type simulation checking ( $\text{SIM}(T_1, T_2)$ ) is shown in Figures 2 and 3. It consists of three main phases. During Phase 1, the algorithm creates and populates a type matrix *simTypes* with boolean values or symbolic references, both obtained by repeatedly calling the *SIMPLESIM* algorithm of Figure 2. 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 *COMPARABLE*). For instance, given  $T_1 = l[B, m[B]]$  and  $T_2 = l[(B, p[B]) \mid (B, m[B])]$ , 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 ( $\text{ref}(U_x, V_y)$ ), a logical combination of simple references (e.g.,  $\bigwedge_{i=1}^n \text{ref}(U_i, T_2)$ ), or a *product* reference ( $\text{ref}(\{U_1, \dots, U_n\} \otimes \{V_1, \dots, V_m\})$ ). A simple reference  $\text{ref}(U_x, V_y)$  is a symbolic 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  $\{U_1, \dots, U_n\}$  and  $\{V_1, \dots, V_m\}$ . 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 *simType*[ $x$ ][ $y$ ], by evaluating logical combinators, and by applying the maximum flow algorithm for  $\otimes$  references; in the latter case, an auxiliary (and trivial) function *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 *MAXIMUMFLOW*. The result of this phase is a fully instantiated type matrix, since, as shown by Lemma 6.7, 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  $*$ -types in the right hand-side of the comparison, are marked as *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  $simTypes[i][j]$ .

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

**Example 6.2** Consider two types  $T_1 = l[B, m[B]]$  and  $T_2 = l[B, m[B], p[B]]$ . As it can be observed from the definition of type simulation,  $T_1 \preceq T_2$ , since  $B, m[B] \preceq B, m[B], p[B]$ . Assuming a type term decomposition of  $T_1$  and  $T_2$  obtained by means of a pre-order visit, the algorithm SIM, when applied to  $T_1$  and  $T_2$ , starts with the following type matrix:

	$V_1$	$V_2$	$V_3$	$V_4$	$V_5$	$V_6$	$V_7$
$Z_1$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$
$Z_2$	$\perp$				$\perp$		$\perp$
$Z_3$	$\perp$				$\perp$		$\perp$
$Z_4$	$\perp$				$\perp$		$\perp$
$Z_5$	$\perp$	$\perp$	$\perp$	$\perp$		$\perp$	

where:

$Z_1 = l[B, m[B]]$	$V_1 = l[B, m[B], p[B]]$
$Z_2 = B \mid m[B]$	$V_2 = B, m[B], p[B]$
$Z_3 = B$	$V_3 = B$
$Z_4 = m[B]$	$V_4 = m[B]$
$Z_5 = B$	$V_5 = B$
	$V_6 = p[B]$
	$V_7 = B$

For the sake of simplicity, we insert a  $\perp$  symbol on those cells relating non-comparable types.

The algorithm enters its first phase, so, by comparing  $Z_i$  types with  $V_j$  types, it generates the type matrix of Figure 6.2 (for lack of space, we indicate only the presence of a reference).

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:

	$V_1$	$V_2$	$V_3$	$V_4$	$V_5$	$V_6$	$V_7$
$Z_1$	true	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$
$Z_2$	$\perp$	true	false	false	$\perp$	false	$\perp$
$Z_3$	$\perp$	true	true	false	$\perp$	false	$\perp$
$Z_4$	$\perp$	true	false	true	$\perp$	false	$\perp$
$Z_5$	$\perp$	$\perp$	$\perp$	$\perp$	true	$\perp$	true

Since  $simTypes[1][1] = \text{true}$ , the algorithm correctly predicts that  $T_1 \preceq T_2$ . ■

## 6.2 Correctness and Completeness

To prove correctness and completeness of the SIM algorithm wrt type simulation, we need the following Lemmas.

**Lemma 6.3** *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  $ref(\{T_1, ()\} \otimes \{T_1\})$ , 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  $T = T_1, T_2$  and  $U = T_2, T_1$ . SIMPLESIM( $T, U$ ) returns a  $\otimes$  reference  $ref(\{T_1, T_2\} \otimes \{T_2, T_1\})$ , 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 6.4** *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  $ref(T, U)$ .

**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.

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

$l[T] \preceq l[U]$ : By case 5/6.

$T_1 \preceq U_2, U_3$ : Case 8/9 is applied; the algorithm returns a  $\otimes$  reference of the form  $ref(\{T_1\}, \{U_2, U_3\})$ , which instructs the SIM algorithm 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  $ref(T_1, V_1) \vee ref(T_1, V_2)$ . 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  $ref(T_1, U) \wedge ref(T_2, U)$ . 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 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. ■

**Lemma 6.5** *Given  $i \in [1, \dots, n]$ , given  $j \in [1, \dots, m]$ , if  $simTypes[i][j]$  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. ■

```

SIMPLESIM(Type  $T_1$ , Type  $T_2$ )
1  switch
2    case  $T_1 = T_2$  :
3      return true
4    case  $T_1 = ()$  :
5      return true
6    case  $T_1 = l[U_1] \wedge T_2 = l[U_2]$  :
7      return  $ref(U_1, U_2)$ 
8    case  $T_1 = U_1, \dots, U_n \wedge T_2 = V_1, \dots, V_m$  :
9      return  $ref(\{U_1, \dots, U_n\} \otimes \{V_1, \dots, V_m\})$ 
10   case  $T_1 = U_1, \dots, U_n \wedge T_2 = V_1 \mid \dots \mid V_m$  :
11     return  $(\bigvee_{j=1}^m ref(T_1, V_j))$ 
12   case  $T_1 = U_1 \mid \dots \mid U_n \wedge T_2 = V_1, \dots, V_m$  :
13     return  $(\bigwedge_{i=1}^n ref(U_i, T_2))$ 
14   case  $T_1 = U * \wedge T_2 = V*$  :
15     return  $ref(U, T_2)$ 
16   case  $T_1 = U_1, \dots, U_n \wedge T_2 = V*$  :
17     return  $(\bigwedge_{i=1}^n ref(U_i, T_2))$ 
18   case  $T_1 = U \wedge T_2 = V*$  :
19     return  $ref(U, V)$ 
20   case default :
21     return false

```

Figure 2. SimpleSim algorithm.

```

SIM(Type  $T_1$ , 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  $simTypes[i][j] = SIMPLESIM(U_i, V_j)$ 
8  // phase 2: reference resolution
9  for  $i \leftarrow n$  to 1
10 do for  $j \leftarrow m$  to 1
11   do if  $simTypes[i][j] = ref(Z_p, U_q) \wedge simTypes[p][q] \in \{\mathbf{true}, \mathbf{false}\}$ 
12     then  $simTypes[i][j] = simTypes[p][q]$ 
13   else if  $simTypes[i][j]$  contains references different from  $\otimes$ 
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  $simTypes[i][j] = ref(\{U_f, \dots, U_p\} \otimes \{V_g, \dots, V_q\})$ 
18     then  $\mathcal{P}_1 = \{U_f, \dots, U_p\}$ 
19            $\mathcal{P}_2 = \{V_g, \dots, V_q\}$ 
20     mark as special  $\mathcal{P}_2$  nodes corresponding to *-types
21      $G = GRAPHCONSTR(\mathcal{P}_1, \mathcal{P}_2)$ 
22      $simTypes[i][j] = MAXIMUMFLOW(G)$ 
23 // phase 3: result discovery
24 return  $simTypes[1][1]$ 

```

Figure 3. Type simulation algorithm.

	$V_1$	$V_2$	$V_3$	$V_4$	$V_5$	$V_6$	$V_7$
$Z_1$	$ref(Z_2, V_2)$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$
$Z_2$	$\perp$	$ref(\{Z_3, Z_4\} \otimes \{V_3, V_4, V_6\})$	$ref(\{Z_3, Z_4\} \otimes \{V_3\})$	$ref(\{Z_3, Z_4\} \otimes \{V_4\})$	$\perp$	$ref(\{Z_3, Z_4\} \otimes \{V_6\})$	$\perp$
$Z_3$	$\perp$	$ref(\{Z_3\} \otimes \{V_3, V_4, V_6\})$	<b>true</b>	<b>false</b>	$\perp$	<b>false</b>	$\perp$
$Z_4$	$\perp$	$ref(\{Z_4\} \otimes \{V_3, V_4, V_6\})$	<b>false</b>	$ref(Z_5, V_5)$	$\perp$	<b>false</b>	$\perp$
$Z_5$	$\perp$	$\perp$	$\perp$	$\perp$	<b>true</b>	$\perp$	<b>true</b>

Figure 4. First phase type matrix.

**Lemma 6.6** Given  $i \in [1, \dots, n]$ , given  $j \in [1, \dots, m]$ , if  $simTypes[i][j]$  contains a reference of the form  $ref(\{Z_f, \dots, Z_h\} \otimes \{V_p, \dots, V_q\})$ , then  $i \leq f, \dots, i \leq h$  and  $j \leq p, \dots, j \leq q$

**Proof.** By a simple inspection of the SIMPLESIM algorithm. ■

These lemmas allows us to prove the following Lemma about the second phase of the SIM algorithm.

**Lemma 6.7** 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 6.5 and 6.6, 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 6.8** 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:  $simTypes[i][j]$  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 6.4, 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 showed, 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. ■

### 6.3 Complexity Analysis

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

**Lemma 6.9** The SIMPLESIM algorithm has  $O(1)$  worst case complexity.

**Proof.** We assume the pattern matching on types (e.g., **case**  $T_1 = \dots$ ) 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 6.10** 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 6.9, has  $O(1)$  complexity. Moreover, each call 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  $\mathcal{G}$  ( $O(nm)$  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 [12], hence this phase has  $O(nm(n + m)^3)$  worst case time complexity.

#### Phase 3

Phase 3 requires just one operation. ■

It should be noted that, while SIM has polynomial complexity, the overall approach has exponential complexity, as types must be normalized (via  $norm()$ ) before the application of SIM.

## 7. Related Work

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 [18, 19] 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 [21], 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.

We have already discussed the mapping maintenance technique of [6], which forms the starting point for this work. An alternative technique for detecting corrupted mappings in XML p2p systems has been presented in [7]. This technique is based on the use of a type system capable of checking the correctness of a query, in a XQuery-like language [5], 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 [7] 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.

There exist some similarities between our notion of type projection and the subsumption relation described in [17], but these similarities are quite vague, as subtyping implies projection while the same does not hold for subsumption.

## 8. Conclusions and Future Work

In this paper we presented an efficient algorithm for projection-checking among XML types. We first 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. The relevance of this algorithm is very significant, as type projection forms the basis of the mapping maintenance technique for p2p data integration systems presented in [6]. 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.

Our future work moves along three lines. First, we want to finalize the implementation of our algorithm and test it on various testbeds, so to validate its efficiency: we believe that there is still space for some improvements, in particular for relatively frequent special cases.

Second, we want to study an extension of our approach to unordered recursive types. This is a challenging task, as recursive types prevent the structure of the type matrix to be set before type unfolding.

Finally, we are adapting the techniques used in the SIM algorithm to existing subtype-checking algorithms for XML types, in particular for what concerns product and union type matching. Theorem 4.2 shows that subtype-checking algorithms, combined with type approximation, can be used to check for type projection: while the contrary is yet to be proved, we are quite confident that most of the techniques of the algorithm presented in Section 6 can be exploited to significantly improve existing subtype-checking algorithms.

## 9. Acknowledgments

Authors would like to thank the anonymous reviewers of PDP for their useful and insightful comments.

## References

- [1] M. Arenas and L. Libkin. Xml data exchange: Consistency and query answering. In *PODS*, 2005.
- [2] V. Benzaken, G. Castagna, and A. Frisch. Cduce: an xml-centric general-purpose language. In *ICFP*, pages 51–63, 2003.
- [3] B. Choi. What are real dtts like? In *WebDB*, pages 43–48, 2002.
- [4] D. Colazzo. *Path Correctness for XML Queries: Characterization and Static Type Checking*. PhD thesis, Dipartimento di Informatica, Università di Pisa, 2004.
- [5] D. Colazzo, G. Ghelli, P. Manghi, and C. Sartiani. Types for Path Correctness of XML Queries. In *Proceedings of the 2004 International Conference on Functional Programming (ICFP), Snowbird, Utah, September 19-22, 2004*, 2004.
- [6] D. Colazzo and C. Sartiani. Mapping Maintenance in XML P2P Databases. In *Proceedings of the 10th International Symposium on Data Bases and Programming Languages - DBPL 2005, Trondheim, Norway, August 28-29, 2005*, 2005.
- [7] D. Colazzo and C. Sartiani. Typechecking Queries for Maintaining Schema Mappings in XML P2P Databases. In *Proceedings of the 3th Workshop on Programming Language Technologies for XML (Plan-X), in conjunction with POPL 2005*, 2005.
- [8] R. D. Cosmo, F. Pottier, and D. Rémy. Subtyping recursive types modulo associative commutative products. In P. Urzyczyn, editor, *TLCA*, volume 3461 of *Lecture Notes in Computer Science*, pages 179–193. Springer, 2005.
- [9] S. Dal-Zilio, D. Lugiez, and C. Meyssonier. A logic you can count on. In N. D. Jones and X. Leroy, editors, *POPL*, pages 135–146. ACM, 2004.
- [10] D. Draper, P. Fankhauser, M. Fernandez, A. Malhotra, K. Rose, M. Rys, J. Siméon, and P. Wadler. XQuery 1.0 and XPath 2.0 Formal Semantics. Technical report, World Wide Web Consortium, Sept. 2005. W3C Working Draft.
- [11] A. Fuxman, P. G. Kolaitis, R. J. Miller, and W. C. Tan. Peer data exchange. In *PODS*, 2005.
- [12] A. V. Goldberg. Recent developments in maximum flow algorithms. In *SWAT: Scandinavian Workshop on Algorithm Theory*, 1998.
- [13] A. Y. Halevy, Z. G. Ives, J. Madhavan, P. Mork, D. Suciu, and I. Tatarinov. The piazza peer data management system. *IEEE Trans. Knowl. Data Eng.*, 16(7):787–798, 2004.
- [14] A. Y. Halevy, Z. G. Ives, P. Mork, and I. Tatarinov. Piazza: data management infrastructure for semantic web applications. In *Proceedings of the Twelfth International World Wide Web Conference, WWW2003, Budapest, Hungary, 20-24 May 2003*, pages 556–567. ACM, 2003.
- [15] H. Hosoya and B. C. Pierce. Xduce: A statically typed xml processing language. *ACM Trans. Internet Techn.*, 3(2):117–148, 2003.
- [16] Z. G. Ives, A. Y. Halevy, and D. S. Weld. Integrating network-bound xml data. *IEEE Data Eng. Bull.*, 24(2):20–26, 2001.
- [17] G. M. Kuper and J. Siméon. Subsumption for xml types. In J. V. den Bussche and V. Vianu, editors, *ICDT*, volume 1973 of *Lecture Notes in Computer Science*, pages 331–345. Springer, 2001.
- [18] N. Kushmerick. Wrapper verification. *World Wide Web*, 3(2):79–94, 2000.
- [19] K. Lerman, S. Minton, and C. A. Knoblock. Wrapper maintenance: A machine learning approach. *J. Artif. Intell. Res. (JAIR)*, 18:149–181, 2003.
- [20] A. Marian and J. Siméon. Projecting xml documents. In *VLDB*, pages 213–224, 2003.
- [21] R. McCann, B. K. AlShebli, Q. Le, H. Nguyen, L. Vu, and A. Doan. Mapping maintenance for data integration systems. In *VLDB*, pages 1018–1030, 2005.
- [22] I. Tatarinov and A. Y. Halevy. Efficient query reformulation in peer-data management systems. In *SIGMOD Conference*, pages 539–550, 2004.