Coupled Schema Transformation and Data Conversion for XML and SQL

Pablo Berdaguer, Alcino Cunha⋆, Hugo Pacheco, and Joost Visser⋆

DI-CCTC, Universidade do Minho, Portugal
joost.visser@di.uminho.pt

Abstract. A two-level data transformation consists of a type-level transformation of a data format coupled with value-level transformations of data instances corresponding to that format. We have implemented a system for performing two-level transformations on XML schemas and their corresponding documents, and on SQL schemas and the databases that they describe. The core of the system consists of a combinator library for composing type-changing rewrite rules that preserve structural information and referential constraints. We discuss the implementation of the system's core library, and of its SQL and XML front-ends in the functional language Haskell. We show how the system can be used to tackle various two-level transformation scenarios, such as XML schema evolution coupled with document migration, and hierarchical-relational data mappings that convert between XML documents and SQL databases.

Key words: Haskell, Transformation, SQL, XML

1 Introduction

Coupled software transformation involves the modification of multiple software artifacts such that they remain consistent with each other [12,8]. Two-level data transformation is a particular instance of coupled transformation, where the coupled artifacts are a data format on the one hand, and the data instances that conform to that format on the other hand [7]. In this paper we will focus on the transformation of data formats described in the XML Schema or in the SQL language, coupled with the conversion of the corresponding data captured in XML documents or stored in SQL databases.

The phenomenon of two-level data transformation occurs in a variety of contexts. For example, software maintenance commonly involves enhancement of the data formats employed for storing or exporting an application's data. Typically such enhancements are fairly conservative, such as adding new fields to the format. When the enhanced format only serves internal data storage, a one-off conversion of old data into new data may be sufficient to restore conformance. When the format concerns data exported to other applications, or shared with older versions of the same application, old-to-new as well as new-to-old data conversions may be needed on a repetitive or continuous basis.

⋆ Work funded by Fundação para a Ciência e a Tecnologia, POSI/ICHS/44304/2002.

Two-level data transformation also encompasses less conservative format changes, such as data mappings between programming paradigms. For example, the logic of an application may be programmed against an XML schema, while for efficient storage of its persistent data a relational database is employed. The required data mapping involves a format transformation from an XML schema to an SQL schema, as well as forward and backward data conversions between XML documents and an SQL database. Unlike format enhancements in the maintenance context, data mappings typically involve profound structural modifications.

Other contexts in which two-level data transformations may play a role include: system integration, where data needs to be exchanged between independently developed applications; evolution of programming languages, where grammar modifications between versions spark the need for migration of source programs; and model-driven engineering where high-level (meta-)model transformations give rise to conversion of their instances.

Previously, we have shown how data refinement theory can be employed to formalize two-level data transformation, and how the functional programming language Haskell can be employed to capture this formalization in a type-safe manner [7]. We also provided suites of rule combinators as well as basic rules for format evolution and hierarchical-relational data mappings from which two-level data transformation pipelines are built in compositional fashion.

In the present paper, we discuss practical application of our Haskell-based two-level transformation support. In particular, we make these contributions:

1. We elaborate the rule combinators and basic rules to take into account not only structural information, but also *constraint information*, such as primary keys and foreign keys (Section 4).
2. We embed the general transformation kernel into a language-specific transformation framework, including front-ends for SQL (schemas and data) and for XML Schema and XML documents (Section 5).
3. We illustrate by example how the XML/SQL transformation framework is used to handle various two-level transformation scenarios, including XML-to-SQL data mappings, XML schema evolution, and SQL database migration (Section 6).

Before discussing these contributions, we will present a motivating example (Section 2) and briefly recapitulate our previous work on two-level data transformation (Section 3). We end with a discussion of related work (Section 7) and concluding remarks (Section 8).

2 Motivating example

The tree in Figure 1 represents an XML movie database schema, before and after evolution. Before evolution, the database holds information for movies and actors only. The evolution steps aims to add information for TV series to the database. This is done through the following changes:

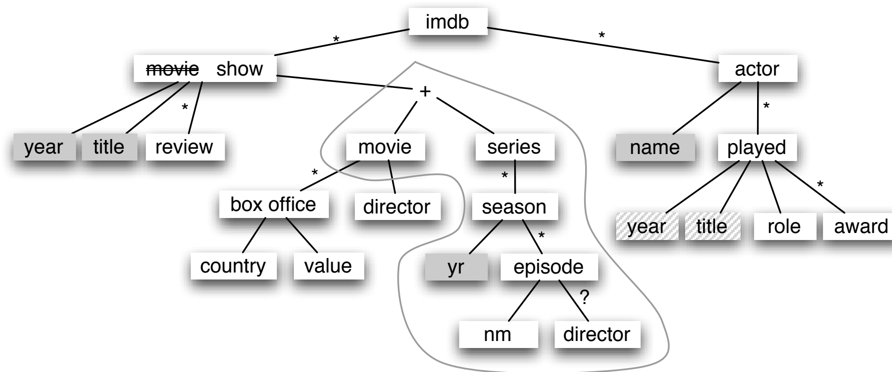

Fig. 1. Evolution of a movie database schema, inspired by IMDb (`http://www.imdb.com/`). The circled area points out the introduced structure.

1. The `movie` element is renamed to `show`.
2. Some information specific to movies is factored out into a new `movie` element.
3. An element `series` with information specific to TV series is introduced as an alternative to the `movie` element.

In the original schema, the following constraints should hold:

1. A `movie` is identified by its `year` and `title`.
2. An `actor` is identified by his/her `name`.
3. The `year` and `title` of a `played` element refers to the `year` and `title` of a `movie`.

The evolution step introduces the following additional constraint:

4. A `season` is identified by its `yr`.

When an XML-to-SQL data mapping is applied to the original and the evolved schema, different SQL databases with different constraints will result. For example, the original schema is mapped to the following database (this example will be revisted and continued in Section 6):

```
movies(year,title,director)
reviews(id,year,title,review)
  foreign key (year,title) references movies(year,title)
boxoffices(id,year,title,country,value)
  foreign key (year,title) references movies(year,title)
actors(name)
playeds(id,name,year,title,role)
  foreign key (year,title) references movies(year,title)
  foreign key (name) references actors(name)
awards(id,name,playedid,award)
  foreign key (playedid,name) references playeds(id,name)
```

In the sequel we will show how both evolution and mapping can be specified by composing library combinators. The backward and forward data conversions induced by these schema transformations will come for free. The properties of the combinators guarantee that the conversions are invertible, i.e. that no data gets lost. The propagation and generation of constraints support the preservation of not only structural, but also semantic information.

3 Two-level data transformation

Two-level data transformation can be formalized in terms of data refinement theory, and can be modeled in Haskell as systems of type-changing rewrite rules [7]. These rewrite rules operate on Haskell types. In Section 5, we will discuss how XML and SQL schemas are represented by such types.

Data refinements A datatype A can be refined to a datatype B, usually denoted by the inequation $A \leqslant B$, if there is an injective, total function $\mathit{to} : A \to B$ (the *representation function*) and a surjective, possibly partial function $\mathit{from} : B \to A$ (the *abstraction function*) such that $\mathit{from} \cdot \mathit{to} = \mathit{id}_A$, where id_A is the identity function on datatype A.

The inequations of data refinement theory can be used as rewrite rules that replace one datatype by another. When applied left-to-right, an inequation $A \leqslant B$ will preserve or enrich information content, while applied right-to-left it will preserve or restrict information content. The (potential) partiality of the *from* function implies that left-to-right application is only valid if the invariant $\mathit{to} \cdot \mathit{from} = \mathit{id}_B$ can be shown to hold.

In fact, when used as a left-to-right rewrite rule, a data refinement inequation $A \leqslant B$, witnessed by functions *to* and *from*, can be interpreted as a two-level data transformation step that takes its input datatype A into the triple $(B, \mathit{to}, \mathit{from})$.

Representation of types and rules The core of the model of two-level data transformations in Haskell are the following declarations:

```
type Rule = ∀a . Type a → Maybe (View (Type a))
data Type a where
  Int :: Type Int
  Prod :: Type a → Type b → Type (a, b)
  Either :: Type a → Type b → Type (Either a b)
  Map :: Type a → Type b → Type (Map a b)
  ...
data View a where View :: Rep a b → Type b → View (Type a)
data Rep a b = Rep{ to :: a → b, from :: b → a }
```

Note that *Type* and *View* are *generalized* algebraic data types (GADTs) [19], an extension to the Haskell type system that allows (partially) instantiated type parameters in the result type of data constructors.

The *Rule* type expresses that a two-level transformation step is a partial function that takes a type into a view of that type. Here we use a value-level

representation of datatypes [11], where a value of $Type\ a$ is the representation of type a. For instance, the value $Prod\ Int\ Int$ represents type (Int, Int).

The *View* constructor expresses that a type a can be transformed into a type b, if there are functions $to :: a \rightarrow b$ and $from :: b \rightarrow a$, bundled in the *Rep* constructor, that allow data conversion between a and b. Note that only the source type a escapes from the *View* constructor, while the target type b remains encapsulated — it is implicitly existentially quantified.

Two-level transformation combinators To construct complex two-level transformations from basic ones, combinators are defined for identity, sequential composition, left-biased choice, repetition, and generic traversal:

$$
\begin{array}{l}
nop :: Rule \\
nop\ x = Just\ (View\ (Rep\ id\ id)\ x) \\
(\triangleright) :: Rule \rightarrow Rule \rightarrow Rule \\
(f \triangleright g)\ a = \mathbf{do}\ View\ (Rep\ t1\ f1)\ b \leftarrow f\ a \\
\qquad\qquad\qquad View\ (Rep\ t2\ f2)\ c \leftarrow g\ b \\
\qquad\qquad\qquad return\ (View\ (Rep\ (t2 \cdot t1)\ (f1 \cdot f2))\ c)
\end{array}
$$

$$
\begin{array}{ll}
(\oslash) :: Rule \rightarrow Rule \rightarrow Rule & everywhere :: Rule \rightarrow Rule \\
many :: Rule \rightarrow Rule & somewhere :: Rule \rightarrow Rule
\end{array}
$$

These combinators are common for typed strategic rewriting libraries [16,15]. For conciseness, we show definitions of the first two only. These combinators allow us to combine local, single-step transformations into a single global transformation.

Several local, single-step transformation rules are shown in Figure 2. These rules are implemented in Haskell in a straightforward way. For example, the rule for adding alternatives is implemented as follows:

$$
\begin{array}{l}
addalt :: Type\ b \rightarrow Rule \\
addalt\ b\ a = Just\ (View\ (Rep\ Left\ (\lambda(Left\ x) \rightarrow x))\ (Either\ a\ b))
\end{array}
$$

Using these basic rules and the rule combinators, we can compose sophisticated strategies for two-level transformation. For example, a hierarchical-relational mapping can be defined along the following lines (details in [7]):

$$
\begin{array}{l}
toRDB :: Rule \\
toRDB = many\ (somewhere\ (listelim \oslash setelim \oslash ... \oslash flatmap))
\end{array}
$$

Such compositions are guaranteed to be refinements again, i.e. they induce invertible data conversion function. The combinators give full control over the order and conditions under which rules are applied.

4 Constraint Preserving Transformation

The type representation and the two-level transformation rules from [7], recapitulated above, fail to take into account constraint information. In particular, foreign key relationships play an important role in relational database modeling and querying. A similar concept is present in XML Schema, though its usage is limited [13]. In this section we discuss how the type representation and transformation rules can be augmented to take constraint information into account.

Hierarchical-to-relational data mapping	
$[A] \leqslant \mathbb{N} \rightharpoonup A$	List elimination
$2^A \cong A \rightharpoonup 1$	Set elimination
$A? \cong 1 \rightharpoonup A$	Optional elimination
$A + B \leqslant A? \times B?$	Sum elimination
$A \times (B + C) \cong (A \times B) + (A \times C)$	Distribute product over sum
$A \rightharpoonup (B + C) \leqslant (A \rightharpoonup B) \times (A \rightharpoonup C)$	Distribute map over sum (range)
$(B + C) \rightharpoonup A \cong (B \rightharpoonup A) \times (C \rightharpoonup A)$	Distribute map over sum (domain)
$A \rightharpoonup (B \times (C \rightharpoonup D)) \leqslant (A \rightharpoonup B) \times (A \times C \rightharpoonup D)$	Flatten nested map

Format evolution			
$A \leqslant A \times B$	Add field	$A^+ \leqslant [A]$	Allow empty list
$A \leqslant A + B$	Add alternative	$A? \leqslant [A]$	Allow repetition
$A \leqslant A?$	Make optional	$A \leqslant A^+$	Allow non-empty repetition

Fig. 2. One-step rules for two-level transformation systems. More details can be found elsewhere [7].

Representation of field names and referential constraints To represent field names and references, we introduce an annotation mechanism on data types. We will write ${}_kA_r^n$ to denote a datatype A with name n, key k, and key references r.

- The name annotation n is either empty, or contains a single name.
- The key annotation k is either empty, or contains a globally unique identifier.
- The key references annotation r is a list of zero or more identifiers.

With such annotations, we can represent the first two tables of our example as:

$$\begin{array}{l}({}_{\mathbf{1}}(Int^{\mathtt{year}} \times Str^{\mathtt{title}}) \rightharpoonup Str^{\mathtt{director}})^{\mathtt{movies}} \quad \times \\ ((Int^{\mathtt{id}} \times (Int^{\mathtt{year}} \times Str^{\mathtt{title}})_{\mathbf{1}}) \rightharpoonup Str^{\mathtt{review}})^{\mathtt{reviews}}\end{array}$$

Note that we represent tables with finite maps, where the map's domain is the primary key of the table. The compound foreign key relationship is represented by the annotation $\mathbf{1}$ on the year-title pair inside each map.

Constraint-preserving transformation rules Using our datatype annotation mechanism, we can enhance some of our two-level transformation rules to manipulate constraint and name information in addition to structural information. Concatenation of reference lists is denoted by juxtaposition.

For example, the introduction of a new key reference when flattening nested maps is captured by the following:

$$({}_kA_r \rightharpoonup (B \times (C \rightharpoonup D)^o))^m \quad \cong \quad ({}_kA_r \rightharpoonup B)^m \times ({}_{\emptyset}A_{kr} \times C \rightharpoonup D)^o$$

Here we use $\emptyset$ to denote absence of keys. Where annotations on types are omitted, we assume that the annotations get copied over from left to right without modifications. The first map on the right-hand side inherits its key k from the outer map on the left-hand side. If no key is present on A, a new key is generated. The second map on the right-hand side contains a datatype A that is

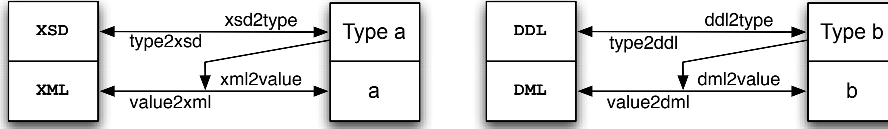

Fig. 3. Overview of the XML and SQL front ends.

annotated with a reference to that key k. Note also that the rule is no longer an inequation, but an isomorphism, because the referential constraint ensures that the flat maps can always be nested again.

The presence of annotations may also *invalidate* the applicability of a rule. For example, the distribution of a map over a sum may only be performed when the domain of the map is not a key (name annotations omitted for brevity):

$$_{\emptyset}A \rightharpoonup B + C \quad \leqslant \quad _{\emptyset}A \rightharpoonup B \times {_{\emptyset}A} \rightharpoonup C$$

The $\emptyset$ indicates that the key annotation of A is required to be empty. This prevents that the target of a reference gets distributed over two different tables, which would break referential integrity. Our system of rules handles types of the form $_{k}A \rightharpoonup B + C$, where k is *not* empty by first applying the sum elimination rule, followed by the optional elimination rule (name annotations omitted again):

$$_{k}A \rightharpoonup B + C \quad \leqslant \quad _{k}A \rightharpoonup B? \times C? \quad \cong \quad _{k}A \rightharpoonup (1 \rightharpoonup B) \times (1 \rightharpoonup C)$$

After this, the rule for flattening nested maps, given above, can be applied twice to obtain a relational representation.

We have adapted the datatype *Type* to accomodate annotations on type representations, and we have augmented all implementations of two-level rewrite rules with appropriate annotation handling.

5 XML and SQL front-ends

In order to embed the general transformation kernel presented above into a language-specific transformation framework, we developed front ends for the relational database language SQL, and the document markup language XML. The essential operations offered by these front ends are shown in Figure 3.

Both front-ends perform their work in two phases (first schema conversion, then value conversion) and in two directions (from external to internal representation and *vice versa*). In the case of XML, schema information and values are stored separately, using separate languages (XML Schema and XML itself), while in the case of SQL type and value information are stored together (`CREATE` and `INSERT` statements).

The functions for the first phase of the XML front end have the following type signatures:

```
type2xsd :: Type a → Maybe XSD
xsd2type :: XSD → Maybe DynType

data DynType where DynType :: Type a → DynType
```

The *type2xsd* function converts a type representation into the abstract syntax of an XML Schema file, if possible. The *xsd2type* function performs the opposite conversion, but it returns the computed type representation wrapped in the *DynType* constructor. Note that the type variable a does not escape from the *DynType*, which means that it is implicitly existentially quantified. This is essential since the *xsd2type* function is to be applied without knowing the type it will produce. The *Maybe* monad indicates the partiality of the conversions.

The second-phase functions of the XML front end have the following type signatures:

```
xml2value :: Type a → XML → Maybe a
value2xml :: Type a → a → Maybe XML
```

The first argument of both functions is the type representation from the first phase. Using this type representation, a string representation of an XML document gets converted into a value of the represented type, or *vice versa*. These functions are partial, since parsing may fail (*xml2value*) or the type may not have the appropriate form (*value2xml*).

These four XML front-end functions are combined with parsers and pretty-printers for the *XSD* and *XML* abstract syntax trees. For *XML* we use the HaXml parser and printer [22]. For *XSD* we use XML Schema support from the XsdMetz tool [21] which in turn again uses HaXml (schemas in XML Schema are themselves XML files).

The functions of the SQL front end have very similar signatures:

```
create2type :: DDL → Maybe DynType
type2create :: Type a → Maybe DDL

insert2value :: Type a → DML → Maybe a
value2insert :: Type a → a → Maybe DML
```

Here, *DDL* is an abstract syntax for the data definition sublanguage of SQL (`CREATE` statements), and *DML* is an abstract syntax for the data manipulation sublanguage (`INSERT` statments). These functions are combined with an SQL parser that we generated with the Happy parser generator [17], and a hand-crafted pretty-printer.

The pattern shared by the two front ends is captured in the following class and corresponding instances:

```
class FrontEnd t v | t → v, v → t where
  parsetype :: t → Maybe DynType
  printtype :: Type a → Maybe t
  parsevalue :: Type a → v → Maybe a
  printvalue :: Type a → a → Maybe v
instance FrontEnd XSD XML where ...
instance FrontEnd DDL DML where ...
```

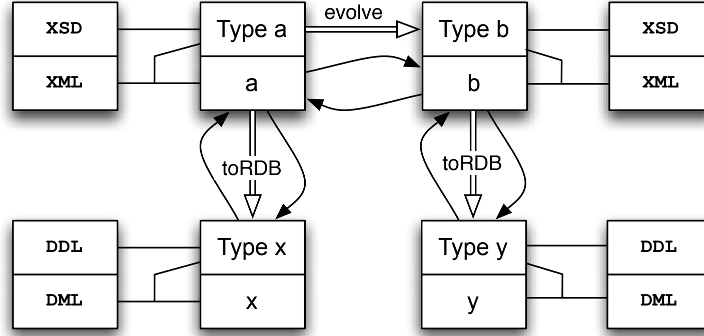

Fig. 4. Overview of the application scenarios.

For brevity, the straightforward instance bodies are not shown. Against the interface of the *FrontEnd* class, we can program an overloaded function that lifts a *Rule* on our internal type representation to a two-level transformation on external abstract syntaxes:

$$
\begin{array}{l}
\mathit{transform} :: (\mathit{FrontEnd}\ t\ v, \mathit{FrontEnd}\ t'\ v') \\
\qquad\qquad \Rightarrow \mathit{Rule} \to t \to \mathit{Maybe}\ (t', v \to \mathit{Maybe}\ v', v' \to \mathit{Maybe}\ v) \\
\mathit{transform}\ r\ t = \mathbf{do} \\
\quad \mathit{DynT}\ a \leftarrow \mathit{parsetype}\ t \\
\quad \mathit{View}\ (\mathit{Rep}\ \mathit{to}\ \mathit{from})\ a' \leftarrow r\ a \\
\quad t' \leftarrow \mathit{printtype}\ a' \\
\quad \mathbf{let}\ \mathit{to}'\ v = \mathbf{do}\ \{x \leftarrow \mathit{parsevalue}\ a\ v; \mathit{printvalue}\ a'\ (\mathit{to}\ x)\} \\
\quad \mathbf{let}\ \mathit{from}'\ v' = \mathbf{do}\ \{x \leftarrow \mathit{parsevalue}\ a'\ v'; \mathit{printvalue}\ a\ (\mathit{from}\ x)\} \\
\quad \mathit{return}\ (t', \mathit{to}', \mathit{from}')
\end{array}
$$

Note that the result type is a triple, where t' is the transformed type, and the partial functions convert v to v' and *vice versa*. In the upcoming sections, we resolve the overloading of the *transform* function in different ways to obtain various concrete two-level transformations for XML and SQL.

6 Application scenarios

We now illustrate by example how the two-level transformation rules can be combined with the XML and SQL front ends to handle various two-level transformation scenarios. See Figure 4 for an overview.

XML evolution The evolution of Section 2, where TV series are added as an alternative to movies, can be encoded as follows:

$$
\begin{array}{l}
\mathit{evolve} :: \mathit{Rule} \\
\mathit{evolve} = \mathit{somewhere}\ (\mathit{changeName}\ \texttt{"movie"}\ \texttt{"show"}) \triangleright \\
\qquad\qquad \mathit{somewhere}\ (\mathit{when}\ \mathit{isMovie}\ (\mathit{putName}\ \texttt{"movie"} \triangleright \mathit{addalt}\ \mathit{series}))
\end{array}
$$

```
  where
    isMovie :: Type a → Bool
    isMovie (Prod (List a) b) = getName a ≡ Just "boxoffice" ∧
                                getName b ≡ Just "director"
    isMovie _ = False
    series = setName "series" (Map year episodes)
    year = setName "yr" Int
    episodes = ...
when :: (∀a . Type a → Bool) → Rule → Rule
getName :: Type a → Maybe String    changeName :: String→String→Rule
putName :: String → Rule            setName :: String → Type a → Type a
```

Thus, the `movie` name is changed into `show` in a single traversal, using *somewhere*. Then, in a second traversal, the schema fragment to be factored out is located with the *isMovie* predicate. This predicate tests for the presence of `boxoffice` and `director`. If the predicate is satisfied, at that point in the schema the `movie` name is reintroduced, and the *addalt* rule is triggered to insert the `series` fragment. Note that this latter fragment is defined by a *Map*, which encodes that a season is uniquely identified by its year.

We can now feed the *evolve* rule to our *transform* function to perform a data mapping:

```
> xsd ← parseXsdFile "imdb.xsd"
> let Just (xsd', to, from) = transform evolve xsd
> xml ← parseXmlFile "imdb.xml"
> let Just xml' = to xml
> show xml'
<imdb>
  <show><title>Pulp Fiction</title><year>1994</year>
    <movie><director>Quentin Tarantino</director></movie>
  </show>
  <actor><name>John Travolta</name>
    <played><title>Pulp Fiction</title><year>1994</year>
      <character>Vincent Vega</character>
    </played>
  </actor>
</imdb>
```

Thus, we use the resulting *to* function and apply it to an input document, to obtain a converted document. Note that the `show` tag appears in the original place of the `movie` tag, which now tags nested information specific to movies.

XML to SQL data mapping We map the original schema to SQL as follows:

```
> xsd ← parseXsdFile "imdb.xsd"
> let Just (ddl, tosql, fromsql) = transform toRDB xsd
> xml ← parseXmlFile "imdb.xml"
> let Just dml = tosql xml
> show dml
```

```
insert into movies (year,title,director)
  values (1994,'Pulp Fiction','Quentin Tarantino');
insert into actors (name)
  values ('John Travolta');
insert into playeds (id,name,year,title,role)
  values (0,'John Travolta',1994,'Pulp Fiction','Vincent Vega');
```

Here we have supplied the *toRDB* strategy to the *transform* function. The resulting *ddl* corresponds to the pseudo-SQL that we showed in Section 2. Note that the *tosql* function would return *Nothing* if this document does not conform to the original XML schema. Multiple documents can be converted into SQL insert statements and loaded into a relational database:

$>$ *createDB* `"imdb"` *ddl*
$>$ *loadDB* `"imdb"` *dml*
$>$ *xml* $\leftarrow$ *parseXmlFile* `"imdb2.xml"`
$>$ **let** *Just dml* $=$ *tosql xml*
$>$ *loadDB* `"imdb"` *dml*

With *createDB* and *loadDB* we connect to an external DBMS. If the combination of documents violates the propagated constraints, the DBMS will refuse to load the data. An XML view of the complete database can be obtained as follows:

$>$ (*ddl*, *dml*) $\leftarrow$ *dumpDB* `"imdb"`
$>$ **let** (*Just xml*) $=$ *fromsql dml*
$>$ *show xml*

```
<imdb>
  <movie><title>Pulp Fiction</title><year>1994</year>
    <director>Quentin Tarantino</director>
  </movie>
  <movie><title>Videodrome</title><year>1983</year>
    <director>David Cronenberg</director>
  </movie>
  <actor><name>John Travolta</name>
    ...
  </actor>
  ...
</imdb>
```

Note that we use the *fromsql* function to do backward conversion.

Data mapping after evolution Like the original XML schema, the evolved schema can be mapped to a relational database:

$>$ **let** *Just* $(ddl', tosql', fromsql') = $ *transform toRDB* xsd'

In the pseudo-SQL notation, the relational schema ddl' looks as follows:

```
shows(year,title)
reviews(id,year,title,review)
  foreign key (year,title) references shows(year,title)
movies(year,title,director)
  foreign key (year,title) references shows(year,title)
```

```
boxoffices(id,year,title,country,value)
  foreign key (year,title) references movies(year,title)
seriess(year,title)
  foreign key (year,title) references shows(year,title)
seasons(year,title,yr)
  foreign key (year,title) references seriess(year,title)
episodes(id,year,title,yr,nm,director?)
  foreign key (year,title,yr) references seasons(year,title,yr)
actors(name)
playeds(id,name,year,title,role)
  foreign key (year,title) references shows(year,title)
  foreign key (name) references actors(name)
awards(id,name,playedid,award)
  foreign key (playedid,name) references playeds(id,name)
```

Note that the `shows` table was called `movies` before, and that the `director` field has moved to the new `movies` table. New tables for series, seasons, and episodes have appeared. The generated referential constraints enforce that all movies and series also appear in the `shows` table.

Database migration With the composition $tosql' \cdot to \cdot fromsql$ of various conversion functions, we can migrate the relational database `imdb` to an evolved relational database. However, this pipeline performs various superflous pretty-print and parse steps, since the intermediate types are XML ASTs. To avoid this, we can use a dedicated function for migrations:

$$
\begin{array}{l}
migrate :: Rule \rightarrow XSD \rightarrow Maybe\ (DML \rightarrow Maybe\ DML, DML \rightarrow Maybe\ DML) \\
migrate\ r\ t = \mathbf{do} \\
\quad DynT\ a \leftarrow parsetype\ t \\
\quad View\ (Rep\ to\ from)\ b \leftarrow toRDB\ a \\
\quad View\ (Rep\ to'\ from')\ b' \leftarrow (r \triangleright toRDB)\ a \\
\quad \mathbf{let}\ to'\ v = \mathbf{do}\ \{x \leftarrow parsevalue\ b\ v; printvalue\ b'\ (to'\ (from\ x))\} \\
\quad \mathbf{let}\ from'\ v' = \mathbf{do}\ \{x \leftarrow parsevalue\ b'\ v'; printvalue\ b\ (to\ (from'\ x))\} \\
\quad return\ (to', from')
\end{array}
$$

The *migrate* function takes an evolution rule and an initial XML schema, and produces forward and backward conversion functions between the relational databases corresponding to the initial and the evolved schema. For example:

$$
\begin{array}{l}
> \mathbf{let}\ Just\ (migrateto, migratefrom) = migrate\ evolve\ xsd \\
> \mathbf{let}\ Just\ dml' = migrateto\ dml \\
> createDB\ \texttt{"evolvedimdb"}\ ddl' \\
> loadDB\ \texttt{"evolvedimdb"}\ dml'
\end{array}
$$

After this, a second movie database has been created and filled with the data from the old database.

7 Related work

XML-to-relational mappings A large number of approaches has been proposed for mapping XML to relational databases [1]. Most approaches offer a fixed

mapping strategy, but some allow manual intervention [3] or automatic cost-based selection of an optimal target schema [4]. Many approaches only offer forward data conversion, though some offer backward conversion as well [2]. Our approach is fully compositional, and allows various mappings known from the literature to be recomposed in a purely declarative way from basic rules.

XML-to-relational mappings are expected to be information-preserving in some sense, but few approaches come with a precise definition or formal guarantees of such preservation properties. An exception is the use of the notion of *invertibility* by Barbosa *et al* [2], which in turn is based on the classic notion of relative information capacity in the database context. The same property of invertibility is satisfied by our two-level data transformation rules, as expressed by the law $from \cdot to = id_A$. Data refinement theory shows that structural and sequential composition of our rules maintain invertibility.

Constraint preservation Few XML-to-relational mapping approaches take constraint information into account. A notion of *XML Functional Dependency* (XFD) is introduced by Chen *et al* [5,6], based on path expression, and mapping algorithms are provided that propagate XFDs to the target relational schema, and exploit XFDs to arrive at a schema with less redundancy. Davidson *et al* [9] and Barbosa *et al* [2] present alternative constraint-preserving approaches, also involving constraints based on path expressions.

Our approach, by contrast, employs a type annotation mechanism to capture constraints, rather than path expressions. As a result, we capture a smaller class of possible XML constraints. The advantage, however, is that our annotation mechanism allows a compositional treatment of constraints, which fits better with our rule-based mapping approach.

XML format evolution Lämmel *et al* [14] propose a systematic approach to evolution of XML-based formats, where DTDs are transformed in a well-defined, step-wise fashion, and migration of corresponding documents can largely be induced from the DTD-level transformations. They discuss properties of transformations and identify categories of transformation steps, such as renaming, introduction and elimination, folding and unfolding, generalization and restriction, enrichment and removal, taking into account many XML-specific issues, but they stop short of formalization and implementation of two-level transformations. In fact, they identify the following ‘challenge’: “We have examined typeful functional XML transformation languages, term rewriting systems, combinator libraries, and logic programming. However, the coupled treatment of DTD transformations and induced XML transformations in a typeful and generic manner, poses a challenge for formal reasoning, type systems, and language design.” We have now met this challenge, albeit for XML Schema rather than DTDs.

Bi-directional programming Foster *et al* tackle the classical *view-update problem* for databases with *lenses*: combinators for bi-directional programming [10]. Each lens connects a concrete representation C with an abstract view A on it by means of two functions $get : C {\rightarrow} A$ and $put : A {\times} C {\rightarrow} C$. Thus, *get* and *put* are similar

to our *from* and *to*, except for *put*'s additional argument of type C. Also, an additional law on these functions guarantees that *put* can be used to reconstruct an updated C from an updated A. Hu *et al* take a smilar approach [20].

We believe that our techniques for coupled transformations can equally be beneficial for bi-directional programming with lenses. In particular, we are currently designing an embedding of bi-directional programs in Haskell that provides strong, inferable types, as well as strategic rewrite systems for lens composition.

8 Concluding remarks

We have shown how XML format evolution, XML-to-SQL mappings, and SQL migrations can be given a unified declarative treatment as instances of two-level data transformations. Schema-level transformations produce new schemas, as well as bi-directional conversion functions between old and new. Name information and constraint information can be preserved through transformation steps. The approach is compositional, in the sense that full transformations are composed from basic transformation rules and rule combinators, and properties such as invertibility are preserved under composition. The approach can be extended to cover other hierarchical and relational data languages, by providing more implementations of the *FrontEnd* class. Source code and examples are available from the homepages of the authors under the name 2LT.

Future work Though already useful in practise, our approach suffers from various limitations that we intend to overcome.

In [8] we have shown that two-level data transformation systems can be supplemented with type-directed program transformation systems to perform optimization of the induced conversion functions. Moreover, such combined rewriting systems can be used to perform migration of queries through evolution. We would like to extend our XML and SQL front-ends to leverage such program transformations for corresponding query languages.

So far, all our transformations *on the type level* are performed in the refinement direction, i.e. from abstract to more concrete types. Constraint handling opens the door to performing these steps in the opposite direction, i.e. to perform reverse engineering from low-level data schemas to higher-level ones [18].

Our annotation mechanism is sufficient to capture a large class of common XML and SQL constraints. We would like to enlarge this class further.

Acknowledgements We thank Flávio Ferreira and Diogo Lapa for their work on the front ends, and José Nuno Oliveira for inspiring discussions.

References

1. S. Amer-Yahia, F. Du, and J. Freire. A comprehensive solution to the XML-to-relational mapping problem. In *WIDM '04: Proc. 6th annual ACM Int workshop on Web Information and Data Management*, pages 31–38. ACM Press, 2004.

2. D. Barbosa, J. Freire, and A.O. Mendelzon. Designing information-preserving mapping schemes for XML. In *VLDB'05: Proc. 31st Int. Conf. Very Large Data Bases*, pages 109–120. VLDB Endowment, 2005.
3. P. Bohannon et al. LegoDB: Customizing relational storage for XML documents. In *Proc. 28th Int. Conf. on Very Large Data Bases*, pages 1091–1094, 2002.
4. P. Bohannon, J. Freire, P. Roy, and J. Siméon. From XML schema to relations: A cost-based approach to XML storage. In *ICDE '02: Proc. 18th Int. Conf. on Data Engineering*, pages 64–. IEEE Computer Society, 2002.
5. Y. Chen, S.B. Davidson, C.S. Hara, and Y. Zheng. RRXS: Redundancy reducing XML storage in relations. In *Proc. 29th VLDB Conference*, pages 189–200, 2003.
6. Y. Chen et al. Constraints preserving schema mapping from XML to relations. In *Proc. 5th Int. Workshop Web and Databases (WebDB)*, pages 7–12, 2002.
7. A. Cunha, J.N. Oliveira, and J. Visser. Type-safe two-level data transformation. In J. Misra et al., editors, *Proc. Int. Symp. of Formal Methods Europe*, volume 4085 of *LNCS*. Springer, 2006.
8. A. Cunha and J. Visser. Strongly typed rewriting for coupled software transformation. In M. Fernandez and R Lämmel, editors, *Proc. 7th Int. Workshop on Rule-Based Programming (RULE 2006)*, ENTCS. Elsevier, 2006. To appear.
9. S.B. Davidson et al. Propagating XML constraints to relations. In *Proc. 19th Int. Conf. on Data Engineering*, pages 543–. IEEE Computer Society, 2003.
10. J.N. Foster et al. Combinators for bi-directional tree transformations: a linguistic approach to the view update problem. In *Proc. 32nd ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, pages 233–246. ACM Press, 2005.
11. R. Hinze, A. Löh, and B.C.d.S. Oliveira. "Scrap your boilerplate" reloaded. In *Proc. 8th Int. Symp. on Functional and Logic Programming*, volume 3945 of *Lecture Notes in Computer Science*, pages 13–29. Springer, 2006.
12. R. Lämmel. Coupled Software Transformations (Extended Abstract). In *First International Workshop on Software Evolution Transformations*, November 2004.
13. R. Lämmel, S. Kitsis, and D. Remy. Analysis of XML schema usage. In *Conference Proceedings XML 2005*, November 2005.
14. R. Lämmel and W. Lohmann. Format Evolution. In *Proc. 7th Int. Conf. on Reverse Engineering for Information Systems*, volume 155 of *books@ocg.at*, pages 113–134. OCG, 2001.
15. R. Lämmel and S. Peyton Jones. Scrap your boilerplate: a practical design pattern for generic programming. *ACM SIGPLAN Notices*, 38(3):26–37, March 2003.
16. R. Lämmel and J. Visser. Typed Combinators for Generic Traversal. In *Proc. Practical Aspects of Declarative Programming PADL 2002*, volume 2257 of *LNCS*, pages 137–154. Springer, January 2002.
17. S. Marlow. *Happy User Guide*. Glasgow University, December 1997.
18. F.L. Neves, J.C. Silva, and J.N. Oliveira. Converting informal meta-data to VDM-SL: A reverse calculation approach. In *VDM in Practice!*, September 1999.
19. S. Peyton Jones, G. Washburn, and S. Weirich. Wobbly types: type inference for generalised algebraic data types. Technical Report MS-CIS-05-26, Univ. of Pennsylvania, July 2004.
20. M. Takeichi S.-C. Mu, Z. Hu. Bidirectionalizing tree transformation languages: A case study. *JSSST Computer Software*, 23(2):129–141, 2006.
21. J. Visser. Structure metrics for XML Schema. In J.C. Ramalho et al., editors, *XATA2006, XML: Aplicações e Tecnologias Associadas*. Univ. of Minho, 2006.
22. M. Wallace and C. Runciman. Haskell and XML: generic combinators or type-based translation? In *Proc. 4th ACM SIGPLAN Int. Conf. on Functional Programming*, pages 148–159. ACM Press, 1999.