Available online at www.sciencedirect.com

SciVerse ScienceDirect

Electronic Notes in Theoretical Computer Science 290 (2012) 3–18

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Constraint-aware Schema Transformation

Tiago L. Alves[1]

Universidade do Minho, Portugal
Software Improvement Group, The Netherlands

Paulo F. Silva[2]

Universidade do Minho, Portugal

Joost Visser[3]

Software Improvement Group, The Netherlands

Abstract

Data schema transformations occur in the context of software evolution, refactoring, and cross-paradigm data mappings. When constraints exist on the initial schema, these need to be transformed into constraints on the target schema. Moreover, when high-level data types are refined to lower level structures, additional target schema constraints must be introduced to balance the loss of structure and preserve semantics.
We introduce an algebraic approach to schema transformation that is constraint-aware in the sense that constraints are preserved from source to target schemas and that new constraints are introduced where needed. Our approach is based on refinement theory and point-free program transformation. Data refinements are modeled as rewrite rules on types that carry point-free predicates as constraints. At each rewrite step, the predicate on the reduct is computed from the predicate on the redex. An additional rewrite system on point-free functions is used to normalize the predicates that are built up along rewrite chains.
We implemented our rewrite systems in a type-safe way in the functional programming language Haskell. We demonstrate their application to constraint-aware hierarchical-relational mappings.

Keywords: Schema transformation, Constraints, Invariants, Data refinement, Strategic rewriting, Point-free program transformation, Haskell.

1 Introduction

Data schemas lie at the heart of software systems. Examples are relational database schemas, XML document schemas, grammars, and algebraic datatypes in formal specification. Data schemas prescribe not only the formats to which data instances

[1] Email: t.alves@sig.nl
[2] Email: paufil@di.uminho.pt
[3] Email: j.visser@sig.nl

doi:10.1016/j.entcs.2012.11.008

$$\begin{tikzcd}[column sep=large]
A \arrow[rr, Rightarrow, "T"] \arrow[rr, bend left, "to"] \arrow[dr, bend right, "\phi"'] & & A' \arrow[ll, bend left, "from"] \arrow[dl, dashed, "\phi'"'] \arrow[dl, bend left, "\psi"] \\
& \mathbb{B} &
\end{tikzcd}$$

A, A'	datatype and transformed datatype
T	transformation of type A into type A'
to	migration function of type $A \to A'$ (injective)
from	migration function of type $A' \to A$ (surjective)
ϕ, ϕ'	constraint and transformed constraint
ψ	newly introduced constraint

$$\phi' = \phi \circ \mathit{from} \wedge \psi$$

Fig. 1. Constraint-aware transformation of datatype A with constraint ϕ into datatype A' with constraint ϕ'. The constraint on the target type is the logical conjunction of (i) the constraint on the source type post–composed with the migration function *from*, and (ii) any new constraint ψ introduced by the type-change. When ϕ' is normalized it works on A' directly rather than via A.

must conform, but they also dictate the well-formedness of data queries and update functions. Generally, schema definitions consist of a structural description augmented with constraints that capture additional semantic restrictions, e.g. SQL and XSD schemas may declare referential integrity constraints, grammars include operator precedences, VDM specifications contain datatype invariants.

Data schema transformations occur in a variety of contexts. For example, software maintenance commonly involves enhancement of the data formats employed for storing or exporting application data. Likewise, evolution of programming languages brings along modification of their grammars between versions. More complex schema transformations are involved in data mappings between programming paradigms [14], such as between XML and SQL.

When a data schema is transformed, the corresponding data instances, queries, and constraints must also be adapted. For example, when mapping an XML schema to an SQL schema, data conversion functions between schemas are required. When an XML schema is augmented with new document elements, the queries developed for that schema may need to be adapted to take these elements into account. When a datatype in a formal specification is adapted, so must its invariant and update functions.

In previous work, we and others have addressed the problem of transforming schemas together with the data instances and queries that are coupled with them. We have shown that data refinement theory can be employed to formalize schema transformation [1] as well as the transformation of the corresponding data instances [9]. In combination with point-free program transformation, this formalization extends to migration of data processors [11] including structure-shy queries and update functions [12]. We have harnessed this theoretical treatment in various type-safe rewrite systems and applied these to VDM-SL specifications [1], XML schemas and queries [11,5], and SQL databases [1,5].

We have also addressed the problem of propagation and introduction of constraints [5]. However, this approach was not theoretically supported, did not achieve type-safeness, and was limited to referential integrity constraints only.

In this paper, we propose an improved approach to constraint-aware schema transformation. Figure 1 concisely represents the new approach. Rather than la-

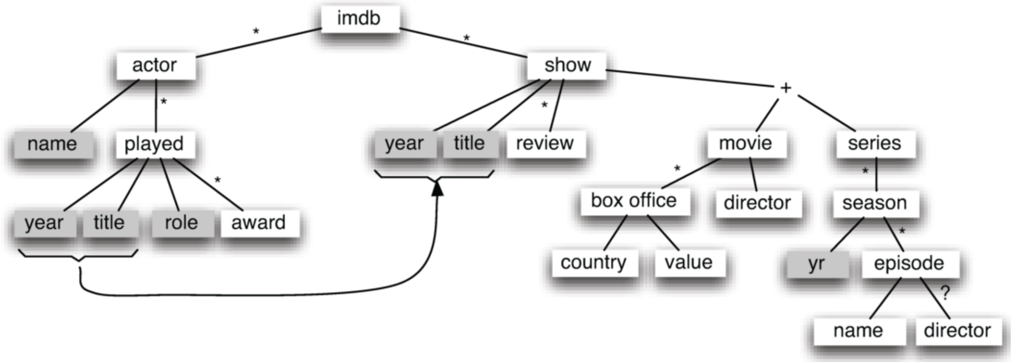

Fig. 2. Schema for an XML database of movies and TV series, inspired by IMDb (http://www.imdb.com/). The shaded elements indicate unique keys in the respective collection elements. In addition, the year and title of the played element are a foreign key into the show collection.

beling the types being transformed with cross-reference information as in [5], we augment them with general constraints represented by strongly-typed function representations. Constraint-propagation is achieved by composing a constraint ϕ on a source data type with a backward conversion function *from* between target and source type. Constraint-introduction is achieved by logical conjunction of a new constraint ψ to the propagated constraint. Finally, point-free program transformation is applied to fuse the various ingredients of the synthesized constraint into a simplified form.

The paper is structured as follows. We introduce the problem of constraint-aware schema transformation with a motivating example in Section 2. We provide background about refinement theory and point-free program calculation in Section 3. Theoretical support about constraints representation and rewriting is provided in Section 4. In Section 5, we explain how this theory can be made operational in the form of strongly-typed rewriting systems implemented in the functional programming language Haskell. We return to the motivating example in Section 6 to demonstrate the application of our rewriting system to schema-aware hierarchical-relational mapping. We discuss related work in Section 7 and conclude in Section 8.

2 Motivating Example

To illustrate the objectives of our approach, we will pick up the motivating example from [5]. The diagram in Figure 2 represents an XML schema for a database of movies and TV series. The schema indicates that the database contains two main collections: one for shows (movies or TV series), and one for actors that play in those shows. Apart from the structure of the database, the following uniqueness constraints are present:

(i) A show is identified by its year and title.

(ii) An actor is identified by his/her name.

(iii) A season is identified by its yr.

(iv) A played element is identified by its year, title, and role.

Also the following referential integrity constraint is present:

(v) The year and title of a played element refer to the year and title of a show.

In XML Schema, such uniqueness and referential integrity constraints are defined by so-called *identity constraints*, using the `key`, `keyref`, and `unique` elements. More constraints could exist, such as that value is always non-zero, or that the name of an episode is different from the title of the corresponding series. Such constraints could be expressed by general queries, e.g. using XPath.

When an XML-to-SQL data mapping is applied to our XML schema, an SQL database schema should result where the various constraints are propagated appropriately. In addition, new constraints would need to exist on the SQL schema that balance the loss of structure due to the flattening to relational form. The schema we need is the following:

```
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)
series(year,title)
  foreign key (year,title) references shows(year,title)
seasons(year,title,yr)
  foreign key (year,title) references series(year,title)
episodes(id,year,title,yr,name,director?)
  foreign key (year,title,yr) references seasons(year,title,yr)
actors(name)
playeds(name,year,title,role)
  foreign key (year,title) references shows(year,title)
  foreign key (name) references actors(name)
awards(name,year,title,role,id,award)
  foreign key (name,year,title,role) references playeds(name,year,title,role)
```

In this pseudo-SQL notation, primary keys are indicated by underlining. The first foreign key constraint is an example of a newly introduced constraint. It arises from the fact that reviews were nested inside shows in the XML schema, but appear in a separate top-level table in the SQL schema. The first foreign key on the `playeds` table is an example of a constraint that was present in the original XML schema and was propagated through the data mapping.

In the remainder of this paper, we will demonstrate how schema transformations such as this XML-to-SQL data mapping can be constructed from strongly-typed algebraic combinators. The propagation of initial constraints and the introduction of new constraints will come for free.

3 Background

In this section, we will explain how schema transformation can be formalized by data refinement theory and point-free program transformation. We start in Section 3.1 by providing background on data refinement theory and its application to two-level transformation. In Section 3.2, we recapitulate point-free program transformation and show how it can be combined with data refinement to model query migration driven by schema transformation.

Sequential and structural composition

$$\text{if } \begin{tikzcd} A \arrow[r, bend left, "to"] \arrow[r, phantom, "\leqslant"] & B \arrow[l, bend left, "from"] \end{tikzcd} \text{ and } \begin{tikzcd} B \arrow[r, bend left, "to'"] \arrow[r, phantom, "\leqslant"] & C \arrow[l, bend left, "from'"] \end{tikzcd} \text{ then } \begin{tikzcd} A \arrow[r, bend left, "to' \cdot to"] \arrow[r, phantom, "\leqslant"] & C \arrow[l, bend left, "from \cdot from'"] \end{tikzcd}$$

$$\text{if } \begin{tikzcd} A \arrow[r, bend left, "to"] \arrow[r, phantom, "\leqslant"] & B \arrow[l, bend left, "from"] \end{tikzcd} \text{ then } \begin{tikzcd} \mathsf{F}\,A \arrow[r, bend left, "\mathsf{F}\,to"] \arrow[r, phantom, "\leqslant"] & \mathsf{F}\,B \arrow[l, bend left, "\mathsf{F}\,from"] \end{tikzcd}$$

Hierarchical-relational data mapping

$A^\star \leqslant \mathbb{N} \rightharpoonup A$	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

Fig. 3. Summary of data refinement theory. For a complete account, the reader is referred to Oliveira [20]. Note that $\cdot \rightharpoonup \cdot$ denotes a simple relation, of which finite maps are a special case.

3.1 Two-level transformation as data refinement

Data refinement theory provides an algebraic framework for calculating with datatypes [20,16,18]. The following inequation captures the essence of refining a datatype A to a datatype B:

$$\begin{tikzcd} A \arrow[r, bend left, "to"] \arrow[r, phantom, "\leqslant"] & B \arrow[l, bend left, "from"] \end{tikzcd} \quad \text{where} \quad \begin{cases} to : A \to B \text{ injective and total} \\ from : B \to A \text{ surjective} \\ from \cdot to = id_A \end{cases}$$

Here, id_A is the identity function on datatype A. Thus, the inequation $A \leqslant B$ expresses that B is a refinement of A, which is witnessed by the conversion functions *to* and *from*. (In fact, *to* can be any injective and total relation, not necessarily a function.) In the special case where the refinement works in both ways we have an isomorphism $A \cong B$. On the basis of this formalization of data refinement, an algebraic theory for calculation with datatypes has been constructed. This theory is summarized in Figure 3.

Data refinement theory can be used to formalize coupled transformation of schemas and their instances [9]. Such two-level transformations can be captured by sequential and structural compositions of data refinement rules. In particular, hierarchical-relational data mappings can be modeled by repeated application of elimination, distribution, and flattening rules, until a fixpoint is reached [1].

3.2 Point-free program transformation

In his 1977 Turing Award lecture, Backus advocated a variable-free style of functional programming, on the basis of the ease of formulating and reasoning with algebraic laws over such programs [3]. After Backus, others have adopted, complemented, and extended his work; an overview of this point-free style of programming is found in [10]. Some function combinators and associated laws that are used in the current paper are shown in Figure 4.

Primitive combinators	
$id : A \to A$	$(\circ) : (B \to C) \to (A \to B) \to (A \to C)$
$\pi_1 : A \times B \to A$	$(\triangle) : (A \to B) \to (A \to C) \to (A \to B \times C)$
$\pi_2 : A \times B \to B$	$(\times) : (A \to B) \to (C \to D) \to (A \times C \to B \times D)$
$\delta : (A \rightharpoonup B) \to Set\ A$	$list : (A \to B) \to ([A] \to [B])$
$\rho : (A \rightharpoonup B) \to Set\ B$	$set : (A \to B) \to (Set\ A \to Set\ B)$
$\bowtie_n :((A \rightharpoonup B) \times ((A \times C) \rightharpoonup D)) \to (A \rightharpoonup (B \times (C \rightharpoonup D)))$ $\bowtie_n^{-1} :(A \rightharpoonup (B \times (C \rightharpoonup D))) \to ((A \rightharpoonup B) \times ((A \times C) \rightharpoonup D))$	
Laws	
$f \circ id = f \qquad id \circ f = f \qquad F\ id = id$ $f \circ (g \circ h) = (f \circ g) \circ h \qquad F\ f \circ F\ g = F\ (f \circ g)$	
$\pi_1 \circ (f \triangle g) = f$	$\pi_2 \circ (f \triangle g) = g$
$\pi_1 \triangle \pi_2 = id$	$(f \times g) \circ (h \triangle i) = (f \circ h) \triangle (g \circ i)$
$\pi_1 \circ (f \times g) = f \circ \pi_1$	$\pi_2 \circ (f \times g) = g \circ \pi_2$
$id \times id = id$	$(f \times g) \circ (h \times i) = (f \circ h) \times (g \circ i)$
$list\ id = id$	$list\ f \circ list\ g = list\ (f \circ g)$
$set\ id = id$	$set\ f \circ set\ g = set\ (f \circ g)$

Fig. 4. Summary of point-free program transformation. For a complete account, refer to Cunha et al. [10].

Point-free program transformation can be used *after* schema transformation to simplify the calculated conversion functions and to migrate queries from source to target type or vice-versa [11]. In Section 5.4 we will use point-free program transformation to migrate and simplify constraints *during* schema transformation.

4 Refinement of datatypes with constraints

In this section we provide theoretical support about constraints representation and rewriting. The formalization of constraints is presented in Section 4.1. Section 4.2 discusses how constraints can be added to data refinement laws to formalize the propagation and introduction of constraints during schema transformation.

4.1 Data types with constraints

A constraint on a datatype can be modeled as a unary predicate, i.e. a boolean function which distinguishes between legal values and values that violate the constraint. To associate a constraint to a type, we will write it as a subscript: A_ϕ where $\phi : A \to \mathbb{B}$ total and functional. This notation, as well as some of the results below, originates in [19]. We will write constraints as much as possible as point-free expressions, to enable subsequent calculation with them. For example, the following datatype represents two tables with a foreign key constraint:

$$((A \rightharpoonup B) \times (C \rightharpoonup A \times D))_{(set\ \pi_1) \circ \rho \circ \pi_2 \subseteq \delta \circ \pi_1}$$

Here we use projection functions π_1 and π_2 to select the left or right table, we use δ and ρ to select the domain and range of a map, and *set* f to map a function f over the elements of a set. Additionally, we use a variant of the set inclusion operated lifted to point-free functions: $\subseteq : (A \to Set\ B) \to (A \to Set\ B) \to (A \to \mathbb{B})$.

Hence, the defined constraint states that all values of A defined in the left table must be contained in the set of keys of the right table.

When a second constraint is added to a constrained datatype, both constraints can be composed with logical conjunction: $(A_\phi)_\psi \equiv A_{\phi \wedge \psi}$.

When a constraint is present on a datatype under a functor, the constraint can be pulled up through the functor (for a categorical proof, see [19]):

$$\mathsf{F}\,(A_\phi) \equiv (\mathsf{F}\,A)_{(\mathsf{F}\,\phi)} \qquad \text{functorial pull}$$

For example, a constraint on the elements of a list can be pulled up to a constraint on the list: $(A_\phi)^\star \equiv (A^\star)_{list\,\phi}$.

4.2 *Introducing, propagating, and eliminating constraints*

The laws of the data refinement calculus must be enhanced to deal with constrained datatypes. Firstly, if a constrained datatype is refined with a 'classic' law, i.e. a law that does not involve constraints, the constraint must be properly propagated through the refinement:

$$\text{if } \begin{tikzcd} A \arrow[r, bend left, "to"] \arrow[r, phantom, "\leqslant"] & B \arrow[l, bend left, "from"] \end{tikzcd} \text{ then } \begin{tikzcd} A_\phi \arrow[r, bend left, "to"] \arrow[r, phantom, "\leqslant"] & B_{\phi \cdot from} \arrow[l, bend left, "from"] \end{tikzcd}$$

Thus, the constraint of the source datatype is propagated to the target datatype, where it is post-composed with the backward conversion function *from*. Such compositions can give rise to opportunities for point-free program transformation, as we will see further on.

Several refinement laws can be changed from inequations to isomorphisms by adding a constraint to the target type. For example, the laws from Figure 3 for sum elimination, distribution of map over sum in its range, and flattening of nested maps can be enhanced as follows:

$$A + B \cong A? \times B?_{(\epsilon \circ \pi_1) \oplus (\epsilon \circ \pi_2)}$$
$$A \rightharpoonup (B + C) \cong (A \rightharpoonup 1) \times (A \rightharpoonup B) \times (A \rightharpoonup C)_{(\delta \circ \pi_2 \subseteq \delta \circ \pi_1) \wedge (\delta \circ \pi_3 \subseteq \delta \circ \pi_1)}$$
$$A \rightharpoonup (B \times (C \rightharpoonup D)) \cong (A \rightharpoonup B) \times (A \times C \rightharpoonup D)_{(set\ \pi_1) \circ \delta \circ \pi_2 \subseteq \delta \circ \pi_1}$$

Here, we have used point-free variants of exclusive disjunction ($\oplus$) and a test for emptiness of an optional (ϵ).

When applying a law that introduces a constraint to a datatype that already has a constraint, the new and existing constraints must be combined:

$$\text{if } \begin{tikzcd} A \arrow[r, bend left, "to"] \arrow[r, phantom, "\leqslant"] & B_\psi \arrow[l, bend left, "from"] \end{tikzcd} \text{ then } \begin{tikzcd} A_\phi \arrow[r, bend left, "to"] \arrow[r, phantom, "\leqslant"] & (B_\psi)_{\phi \cdot from} \arrow[l, bend left, "from"] \end{tikzcd} \equiv B_{\psi \wedge (\phi \cdot from)}$$

This is the *invariant pulling* theorem of [19]. A more general case arises when not only the target, but also the source is constrained in the law that is applied:

$$\text{if } A_\chi \underset{\mathit{from}}{\overset{\mathit{to}}{\leqslant}} B_\psi \text{ and } \phi \Rightarrow \chi \text{ then } A_\phi \underset{\mathit{from}}{\overset{\mathit{to}}{\leqslant}} B_{\psi \wedge (\phi \cdot \mathit{from})}$$

Here we use a point-free variant on logical implication ($\Rightarrow$) to state that the actual constraint ϕ on A must imply the required constraint χ.

In addition to introduction and propagation, constraints can also be weakened or even eliminated, by virtue of the following: if $\phi \Rightarrow \psi$ then $A_\phi \leqslant A_\psi$.

In the special case that ψ is the constant true predicate, such weakening boils down to elimination of a constraint.

5 Constraint-aware rewriting

In this section, we show how the enhanced data refinement theory of the previous section can be captured in a rewriting system, implemented as a strategic functional program in the functional language Haskell. In Section 5.1 we recall how type-safe representations of types and functions can be constructed using generalized algebraic datatypes (GADTs). In Section 5.2 we extend the type representation to constrained types. In Section 5.3 and 5.4, we explain how rewrite systems can be constructed to transform such constrained types.

5.1 Representation of types and functions

To represent both types and functions in a type-safe manner, we rely on *generalized algebraic data types* (GADTs) [21]. To represent types, we use a GADT:

$$
\begin{array}{lll}
\multicolumn{3}{l}{\textbf{data}\ \mathit{Type}\ t\ \textbf{where}} \\
\quad \mathit{One} & :: \mathit{Type}\ () & \\
\quad \mathit{List} & :: \mathit{Type}\ a \to \mathit{Type}\ [a] & \\
\quad \mathit{Set} & :: \mathit{Type}\ a \to \mathit{Type}\ (\mathit{Set}\ a) & \\
\quad \cdot \rightharpoonup \cdot & :: \mathit{Type}\ a \to \mathit{Type}\ b \to \mathit{Type}\ (a \rightharpoonup b) & \text{-- We use } \texttt{lhs2TeX} \text{ for} \\
\quad \cdot + \cdot & :: \mathit{Type}\ a \to \mathit{Type}\ b \to \mathit{Type}\ (a + b) & \text{-- type-setting various} \\
\quad \cdot \times \cdot & :: \mathit{Type}\ a \to \mathit{Type}\ b \to \mathit{Type}\ (a, b) & \text{-- symbols in Haskell.} \\
\quad \mathit{String} & :: \mathit{Type}\ \mathit{String} & \\
\quad \ldots & &
\end{array}
$$

In the result types of the various constructors of this GADT, the parameter t has been instantiated exactly to the type that is represented by the constructor. Such instantiation is what distinguishes a GADT from a traditional parameterized algebraic datatype. To represent functions, we also use a GADT:

$$
\begin{array}{ll}
\multicolumn{2}{l}{\textbf{data}\ F\ f\ \textbf{where}} \\
\quad \mathit{id} & :: F\ (a \to a) \\
\quad \cdot \circ \cdot & :: F\ (b \to c) \to F\ (a \to b) \to F\ (a \to c) \\
\quad \cdot \wedge \cdot & :: F\ (a \to \mathbb{B}) \to F\ (a \to \mathbb{B}) \to F\ (a \to \mathbb{B}) \\
\quad \cdot \subseteq \cdot & :: F\ (a \to (\mathit{Set}\ b)) \to F\ (a \to (\mathit{Set}\ b)) \to F\ (a \to \mathbb{B})
\end{array}
$$

$$
\begin{array}{ll}
\pi_1 & :: F\ ((a, b) \to a) \\
\pi_2 & :: F\ ((a, b) \to b) \\
\cdot \times \cdot & :: F\ (a \to b) \to F\ (c \to d) \to F\ ((a, c) \to (b, d)) \\
\cdot \vartriangle \cdot & :: F\ (a \to b) \to F\ (a \to c) \to F\ (a \to (b, c)) \\
\ldots &
\end{array}
$$

Note that the parameters in the result types are instantiated exactly to the type of the function being represented. For brevity, only a few constructors are shown.

Function representations can be evaluated to the function that is represented: $eval :: Type\ (a \to b) \to F\ (a \to b) \to a \to b$.

Note that GADTs help us to enforce that the type of the function produced matches the type of the function representation.

5.2 Representation of constrained types

To represent constrained datatypes, the first GADT above needs to be enhanced:

$$
\begin{array}{l}
\mathbf{data}\ Type\ t\ \mathbf{where} \\
\quad \ldots \\
\quad (\cdot). :: Type\ a \to F\ (a \to \mathbb{B}) \to Type\ a
\end{array}
$$

Thus, the $(\cdot).$ constructor has as first argument the type that is being constrained, and as second argument the function that represents the constraint. The use of GADTs pays off here, since it enforces that the function is of the right type. This use of the function representation inside the type representation has as important consequence that the rewriting system for functions will be embedded into the rewrite system for types, as we will see later.

To verify if the constraints hold for a specific value we defined *check*:

$$
\begin{array}{ll}
\multicolumn{2}{l}{check :: Type\ t \to t \to Bool} \\
check\ One\ \_ & = True \\
check\ (t1 \times t2)\ (x, y) & = check\ t1\ x \wedge check\ t2\ y \\
\ldots & \\
check\ (t_\phi)\ x & = eval\ (Func\ t\ Bool)\ \phi\ x \wedge check\ t\ x
\end{array}
$$

This function descends through a type representation and the corresponding value. Each time a constraint is found the *eval* function is applied to check its value.

5.3 Rewriting types and functions

The laws of point-free program transformation can be captured in rewrite rules of the following type:

$$\mathbf{type}\ RuleF = \forall a\ b\ .\ F\ (a \to b) \to Maybe\ (F\ (a \to b))$$

We make use of the *Maybe* monad to deal with partiality. For example, the law stating that *id* is the identity of composition is defined as follows:

$$
\begin{aligned}
&\mathit{idR} :: \mathit{RuleF} \\
&\mathit{idR}\ (f \circ \mathit{id}) = \mathit{return}\ f \\
&\mathit{idR}\ \_ \qquad\quad\ = \mathit{mzero}
\end{aligned}
$$

Single step rules of this kind can be combined into full rewrite systems using combinators like the following:

$$
\begin{array}{ll}
\mathit{nop} :: \mathit{RuleF} & \text{-- identity} \\
(\triangleright) :: \mathit{RuleF} \to \mathit{RuleF} \to \mathit{RuleF} & \text{-- sequential composition} \\
(\oslash) :: \mathit{RuleF} \to \mathit{RuleF} \to \mathit{RuleF} & \text{-- left-biased choice} \\
\mathit{many} :: \mathit{RuleF} \to \mathit{RuleF} & \text{-- repetition} \\
\mathit{once} :: \mathit{RuleF} \to \mathit{RuleF} & \text{-- arbitrary depth rule application}
\end{array}
$$

For the implementation of these and other combinators, we refer elsewhere [12,11], as well as for how they can be combined into rewrite systems such as:

$$
\mathit{simplify} :: \mathit{RuleF} \quad \text{-- exhaustively apply rules until reaching normal form}
$$

To implement type transformations, we need a two-level rewrite system. A two-level rewrite rule can be represented as follows [9]:

$$
\begin{aligned}
&\textbf{type}\ \mathit{Rule} = \forall a\ .\ \mathit{Type}\ a \to \mathit{Maybe}\ (\mathit{View}\ (\mathit{Type}\ a)) \\
&\textbf{data}\ \mathit{View}\ a\ \textbf{where}\ \mathit{View} :: (a{\to}b) \to (b{\to}a) \to \mathit{Type}\ b \to \mathit{View}\ (\mathit{Type}\ a)
\end{aligned}
$$

The *View* constructor expresses that a type a can be refined to a type b if a pair of conversion functions between them exist. Note that only the source type a escapes from the type constructor of *View*. The *Rule* type expresses that, when rewriting a type representation we do not replace it but *augment* it with representation functions to translate between the source and the target types.

To compose two-level rewrite systems out of single rules, strategic rewrite combinators are defined, similar to those for rewriting point-free functions. A strategy *flatten* for hierarchical-relational mappings is defined for example in [9].

5.4 Constructing constraint-aware rewrite rules

The construction of constraint-aware rewrite rules differs from normal rules in three important details. Firstly, the rules need to introduce constraints on the target types. Secondly, they need to take into account the possible existence of a constraint on the source type, which needs to be propagated and combined with the newly introduced constraint. Thirdly, some rules require the existence of a constraint on the source type, which must be checked before rule application. To illustrate the first two issues, consider the rule for flattening nested maps.

$$
\begin{aligned}
&\mathit{flatMap} :: \mathit{Rule} \\
&\mathit{flatMap}\ (a \rightharpoonup (b \times (c \rightharpoonup d))) = \mathit{return}\ (\mathit{View}\ \ \bowtie_n^{-1} \bowtie_n\ \ (t_\phi)) \\
&\quad \textbf{where}\ t = (a \rightharpoonup b) \times ((a \times c) \rightharpoonup d)
\end{aligned}
$$

$$\phi = ((set\ \pi_1) \circ (\delta \circ \pi_2)) \subseteq (\delta \circ \pi_1)$$
$$flatMap\ t = propagate\ flatMap\ t$$

The first equation takes care of invariant introduction, where constraint ϕ is attached to the result type t. The issue of constraint propagation is dealt with by the helper function *propagate*, defined as follows:

$$\begin{array}{l} propagate\ rule\ (a_\phi) = \mathbf{do} \\ \quad (View\ to\ from\ b) \leftarrow rule\ a \\ \quad \mathbf{let}\ \psi = \phi \circ from \\ \quad return\ \$\ View\ to\ from\ (b_\psi) \\ propagate\ \_\ \_ = mzero \end{array}$$

The *propagate* function applies its argument rewrite rule to a constrained datatype a to obtain a new datatype b and the conversion functions *to* and *from*. It post-composes constraint ϕ with the *from* to obtain the new constraint ψ. Finally, that constraint is attached to the result type b.

The third issue, of checking the existence of a required constraint, comes into play in the construction of the reciprocal rule, which nests one map into another:

$$\begin{array}{l} nestMap :: Rule \\ nestMap\ (((a \rightharpoonup b) \times ((\_ \times c) \rightharpoonup d))_{(((set\ \pi_1)\circ(\delta\circ\pi_2))\subseteq(\delta\circ\pi_1))}) \\ \quad = return\ \$\ View\ \bowtie_n\ \bowtie_n^{-1}\ (a \rightharpoonup (b \times (c \rightharpoonup d))) \\ nestMap\ \_ = mzero \end{array}$$

Here, pattern matching is performed on the type as well as the constraint. Only if this constraint is equal to the required constraint, the rule succeeds. This does not take into account the possibility that the actual constraint implies the required constraint, but is not equal to it. In that case, some satisfiability proof is needed, which falls outside the scope of this paper (but see [17]).

The function compositions and nested constraints that are created during the application of rewrite steps can be simplified with the following rules:

$$\begin{array}{ll} compose\_constraint :: Rule & \\ compose\_constraint\ ((a_\phi)_\psi) & = return\ (View\ id\ id\ (a_{(\phi\wedge\psi)})) \\ compose\_constraint\ \_ & = mzero \end{array}$$

$$\begin{array}{l} fuse\_constraint :: Rule \\ fuse\_constraint\ (a_\phi) = \mathbf{do} \\ \quad \psi \leftarrow simplify\ \phi \\ \quad return\ (View\ id\ id\ (a_\psi)) \\ fuse\_constraint\ \_ = mzero \end{array}$$

In the latter rewrite rule, the rewrite system *simplify* for point-free functions is invoked, which means that our first function rewrite system will be embedded in our type rewrite system.

6 Application to hierarchical-relational mapping

We will now revisit the example of Section 2. The schema of Figure 2 can be captured by the following type representation:

$$
\begin{array}{ll}
\mathit{imdb} & = (\mathit{actors} \times \mathit{show})_{\mathit{imdb\_inv}} \\
\quad \textbf{where } \mathit{imdb\_inv} = (\mathit{set}\ \pi_1) \circ \mathit{fuse} \circ (\mathit{set}\ \delta) \circ \rho \circ \pi_1 \subseteq \delta \circ \pi_2 & \\
\mathit{show} & = (\texttt{"Year"} \times \texttt{"Title"}) \rightharpoonup ((\mathit{List}\ \texttt{"Review"}) \times (\mathit{movie} + \mathit{series})) \\
\mathit{movie} & = (\mathit{List}\ (\texttt{"Country"} \times \texttt{"Value"})) \times \texttt{"Director"} \\
\mathit{series} & = \texttt{"Yr"} \rightharpoonup (\mathit{List}\ \mathit{episode}) \\
\mathit{episode} & = \texttt{"Name"} \times (\mathit{Maybe}\ \texttt{"Director"}) \\
\mathit{actors} & = \texttt{"Name"} \rightharpoonup \mathit{played} \\
\mathit{played} & = ((\texttt{"Year"} \times \texttt{"Title"}) \times \texttt{"Role"}) \rightharpoonup (\mathit{List}\ \texttt{"Award"})
\end{array}
$$

The primary key constraints of the original schema are captured structurally, by the employment of finite maps. The foreign key constraint is captured by *imdb_inv*, which specifies that the values in the domain of *played* are contained in the domain of *show*, i.e. the year and title (defined in the domain) of *played* are references to the year and title defined in (the domain of) *show*. This constraint is expected to be propagated through the schema transformation process.

The result of the transformation from hierarchical to its relation equivalent, using the *flatten* strategy of [9], followed by constraint simplification, is as follows:

$$
\begin{array}{l}
(\mathit{playeds} \times \mathit{awards} \times \mathit{actors} \times \mathit{shows} \times \mathit{reviews} \times \mathit{seasons} \\
\quad \times\ \mathit{episodes} \times \mathit{series} \times \mathit{movies} \times \mathit{boxoffices})_{\mathit{inv}} \\
\quad \textbf{where}
\end{array}
$$

$$
\begin{array}{ll}
\mathit{playeds} & = (\texttt{"Name"} \times \texttt{"Year"} \times \texttt{"Title"} \times \texttt{"Role"}) \rightharpoonup \mathit{One} \\
\mathit{awards} & = ((\texttt{"Name"} \times \texttt{"Year"} \times \texttt{"Title"} \times \texttt{"Role"}) \times \mathit{Int}) \rightharpoonup \texttt{"Award"} \\
\mathit{actors} & = \texttt{"Name"} \rightharpoonup \mathit{One} \\
\mathit{shows} & = (\texttt{"Year"} \times \texttt{"Title"}) \rightharpoonup \mathit{One} \\
\mathit{reviews} & = ((\texttt{"Year"} \times \texttt{"Title"}) \times \mathit{Int}) \rightharpoonup \texttt{"Review"} \\
\mathit{seasons} & = ((\texttt{"Year"} \times \texttt{"Title"}) \times \texttt{"Yr"}) \rightharpoonup \mathit{One} \\
\mathit{episodes} & = (((\texttt{"Year"} \times \texttt{"Title"}) \times \texttt{"Yr"}) \times \mathit{Int}) \\
 & \quad \rightharpoonup (\texttt{"Name"} \times (\mathit{Maybe}\ \texttt{"Director"})) \\
\mathit{series} & = (\texttt{"Year"} \times \texttt{"Title"}) \rightharpoonup \mathit{One} \\
\mathit{movies} & = (\texttt{"Year"} \times \texttt{"Title"}) \rightharpoonup \texttt{"Director"} \\
\mathit{boxoffices} & = ((\texttt{"Year"} \times \texttt{"Title"}) \times \mathit{Int}) \rightharpoonup (\texttt{"Country"} \times \texttt{"Value"}) \\
\mathit{inv} & = \mathit{fk1} \wedge \mathit{fk2} \wedge \mathit{fk3} \wedge \mathit{fk4} \wedge \mathit{fk5} \wedge \mathit{fk6} \wedge \mathit{fk7} \wedge \mathit{fk8} \wedge \mathit{fk9} \\
\mathit{fk1} & = (\mathit{set}\ \pi_1) \circ \delta \circ \pi_{\mathit{boxoffices}} \subseteq \delta \circ \pi_{\mathit{movies}} \\
\mathit{fk2} & = (\mathit{set}\ \pi_1) \circ \delta \circ \pi_{\mathit{episodes}} \subseteq \delta \circ \pi_{\mathit{seasons}} \\
\mathit{fk3} & = (\mathit{set}\ \pi_1) \circ \delta \circ \pi_{\mathit{seasons}} \subseteq \delta \circ \pi_{\mathit{series}} \\
\mathit{fk4} & = (\mathit{set}\ \pi_2) \circ \delta \circ \pi_{\mathit{reviews}} \subseteq \delta \circ \pi_{\mathit{shows}} \\
\mathit{fk5} & = \delta \circ \pi_{\mathit{movies}} \subseteq \delta \circ \pi_{\mathit{shows}} \\
\mathit{fk6} & = \delta \circ \pi_{\mathit{series}} \subseteq \delta \circ \pi_{\mathit{shows}} \\
\mathit{fk7} & = (\mathit{set}\ \pi_1) \circ \delta \circ \pi_{\mathit{awards}} \subseteq \delta \circ \pi_{\mathit{playeds}} \\
\mathit{fk8} & = (\mathit{set}\ \pi_1 \circ \pi_1 \circ \pi_1) \circ \pi_{\mathit{playeds}} \subseteq \delta \circ \pi_{\mathit{actors}} \\
\mathit{fk9} & = (\mathit{set}\ (\pi_2 \times \mathit{id}) \circ \pi_1) \circ \delta \circ \pi_{\mathit{playeds}} \subseteq \delta \circ \pi_{\mathit{shows}}
\end{array}
$$

Here, we have introduced table names for readability and for comparison to the expected result shown in pseudo-SQL in Section 2. The result consists of 10 tables: 3 derived from the *actor* subschema and 7 from *show*. Additionally, 9 constraints are obtained from which 8 were introduced during transformation. Constraint *fk9* results from the propagation of the original constraint *imdb_inv*. Note that without invocation of the *simplify* rewrite system, the synthesized constraints would not be so concise. For example, without rewrite, an initial fragment of the *fk2* constraint would be:

$$
\begin{aligned}
fk2 = {} & (((set\ \pi_1) \circ \delta \circ \pi_2) \subseteq \delta \circ \pi_1) \circ ((assocl \rightharpoonup id) \times id) \\
& \circ (id \times (assocl \rightharpoonup id)) \circ (id \times (assocl \rightharpoonup id)) \circ ((assocr \rightharpoonup id) \times id) \\
& \circ (id \times (assocr \rightharpoonup id)) \circ (id \times (assocr \rightharpoonup id))) \circ (id \times \pi_1) \circ \pi_2 \circ \pi_2 \circ \ldots
\end{aligned}
$$

Note that in general, simplification can not be postponed until after rewriting, since rules that match on constraints expect them to be in simplified form.

To validate the result we can insert information into the database and observe the constraint checking result. For example, we can add information about the role of an actor in a movie:

$$
\begin{aligned}
& > db' \leftarrow addActorsPlayed\ db\ ((\texttt{"Jet Li"}, (2001, \texttt{"The One"})), \texttt{"Lawless"}) \\
& > check\ imdbResult\ db' \\
& False
\end{aligned}
$$

The constraint correctly fails since neither the name of the actor nor the show exist. We should add that information first:

$$
\begin{aligned}
& > db' \leftarrow addShow\ db\ (2001, \texttt{"The One"}) \\
& > db'' \leftarrow addActor\ db'\ \texttt{"Jet Li"} \\
& > db''' \leftarrow addActorsPlayed\ db''\ ((\texttt{"Jet Li"}, (2001, \texttt{"The One"})), \texttt{"Lawless"}) \\
& > check\ imdbResult\ db''' \\
& True
\end{aligned}
$$

Now the constraint check succeeds.

7 Related work

A large number of approaches has been proposed for mapping XML to relational databases [7,6,2,4], but usually without taking constraints into account. Lee *et al* [15] first addressed the issue of constraint preservation. Their CPI algorithm deals with referential integrity and some cardinality constraints, which are stored in an annotated DTD graph. When the graph is serialized to an SQL schema, various SQL constraints are generated along with the tables. In contrast to our approach, this graph-based algorithm does not deal with arbitrary constraints, it is specific for hierarchical-relational mapping, and it lacks type-safety and formal justification.

A notion of *XML Functional Dependency* (XFD) was introduced by Chen *et al* [8], based on path expressions. 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* [13] present an alternative constraint-preserving approach, also using path expressions. In contrast, our constraints are not restricted to relational integrity constraints. We have expressed constraints as point-free functions, which can be converted automatically to and from structure-shy programs including path expressions [12].

Barbosa *et al* [4] discuss generation of constraints on relational schemas that make XML-relational mappings information preserving, i.e. isomorphic. Non-structural constraints on the initial XML schema are not taken into account. Constraints and conversion functions are expressed in (variations on) Datalog, which can be (manually) rewritten to normal form in a mechanical way.

Berdaguer et al. [5] employ a type annotation mechanism to capture constraints. As a result, a smaller class of possible constraints is covered. Nevertheless, the annotation mechanism allows for a compositional treatment of constraint-aware schema transformation. Rather than path expressions or labels, our approach employs strongly-typed boolean functions to capture constraints. This has the advantage of being more expressive, and allowing a fully compositional treatment. Also note that our approach is not limited to hierarchical-relational mappings, as it can be used for schema transformation in general.

8 Concluding remarks

Contributions We have contributed a treatment of constraint-aware schema transformation to a line of research on the application of data refinement and point-free program transformation to problems of coupled transformation of data schemas, data instances, and queries [20,1,9,11,12,5]. In particular:

- we have shown how data refinement theory [20] can be enhanced to include types constrained by boolean predicates, which amounts to extending the work of [19];
- we have enhanced refinement rules for hierarchical-relational mapping [1,5] such that appropriate constraints are introduced on target types, turning some refinements into isomorphisms;
- we have extended rewrite systems for two-level transformation [9] and coupled transformation [11] to include the propagation, introduction, and simplification of constraints. Moreover, we have shown that value-level rewriting needs to be done *during* type-level rewriting, because type-level rewrite rules may trigger on types with normalized constraints only;
- we have demonstrated the use of the extended rewriting systems for mapping XML schemas to SQL schemas where referential integrity constraints are generated automatically for the target schema. The approach taken in this paper is an alternative to the approach of [5], which was limited to constraints representable with a particular labeling trick while we deal with constraints in general.

Several directions of future work are envisioned.

Constraints as co-reflexive relations We have modeled constraints as boolean functions. Another approach is to model constraints as co-reflexive relations. One advantage of the alternative approach is that it would allow us to use a relational proof system [17] during rewriting to check whether the actual constraint of a redex implies the constraint required by a rule with constrained source datatype.

Integration We want to integrate the treatment of constraints presented here with the front-ends and name-preservation developed in the context of the label-based treatment [5]. Likewise, we want to integrate a rewrite system for structure-shy queries of [12] such that we can deal with structure-shy constraints.

Acknowledgement

Thanks to José Nuno Oliveira and Alcino Cunha for inspiring discussions. The first two authors are supported by the *Fundação para a Ciência e a Tecnologia*, grants `SFRH/BD/30215/2006` and `SFRH/BD/19195/2004`, respectively.

References

[1] Alves, T., P. Silva, J. Visser and J. Oliveira, *Strategic term rewriting and its application to a VDM-SL to SQL conversion.*, in: J. Fitzgerald, I. Hayes and A. Tarlecki, editors, *Formal Methods*, LNCS **3582** (2005), pp. 399–414.

[2] Amer-Yahia, S., 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* (2004), pp. 31–38.

[3] Backus, J., *Can programming be liberated from the von Neumann style? A functional style and its algebra of programs.*, Commun. ACM **21** (1978), pp. 613–641.

[4] Barbosa, D. et al., *Designing information-preserving mapping schemes for XML*, in: *VLDB'05: Int. Conf. Very Large Data Bases* (2005), pp. 109–120.

[5] Berdaguer, P., A. Cunha, H. Pacheco and J. Visser, *Coupled schema transformation and data conversion for XML and SQL*, in: M. Hanus, editor, *Proc. of Practical Aspects of Declarative Programming (PADL'07)*, LNCS **4354** (2007), pp. 290–304.

[6] Bohannon, P. et al., *From XML schema to relations: A cost-based approach to XML storage*, in: *ICDE '02: Proc. 18th Int. Conf. on Data Engineering* (2002), pp. 64–.

[7] Bohannon, P. et al., *LegoDB: Customizing relational storage for XML documents.*, in: *Proc. 28th Int. Conf. on Very Large Data Bases* (2002), pp. 1091–1094.

[8] Chen, Y. et al., *Constraints preserving schema mapping from XML to relations.*, in: *Proc. 5th Int. Workshop Web and Databases (WebDB)*, 2002, pp. 7–12.

[9] Cunha, A., J. Oliveira and J. Visser, *Type-safe two-level data transformation*, in: J. Misra et al., editors, *Proc. Formal Methods, 14th Int. Symp. Formal Methods Europe*, LNCS **4085** (2006), pp. 284–299.

[10] Cunha, A. and J. S. Pinto, *Point-free program transformation.*, Fundam. Inform. **66** (2005), pp. 315–352.

[11] Cunha, A. and J. Visser, *Strongly typed rewriting for coupled software transformation*, ENTCS **174** (2007), pp. 17–34, proc. 7th Int. Workshop on Rule-Based Programming (RULE 2006).

[12] Cunha, A. and J. Visser, *Transformation of structure-shy programs: applied to XPath queries and strategic functions*, in: G. Ramalingam and E. Visser, editors, *PEPM* (2007), pp. 11–20.

[13] Davidson, S. et al., *Propagating XML constraints to relations*, in: *Proc. 19th Int. Conf. on Data Engineering* (2003), pp. 543–.

[14] Lämmel, R. and E. Meijer, *Mappings make data processing go 'round*, in: R. Lämmel, J. Saraiva and J. Visser, editors, *Generative and Transformational Techniques in Software Engineering*, LNCS **4143** (2006).

[15] Lee, D. and W. W. Chu, *Cpi: Constraints-preserving inlining algorithm for mapping xml dtd to relational schema*, Data Knowl. Eng. **39** (2001), pp. 3–25.

[16] Morgan, C. and P. Gardiner, *Data refinement by calculation*, Acta Informatica **27** (1990), pp. 481–503.

[17] Necco, C., J. Oliveira and J. Visser, *Extended static checking by strategic rewriting of pointfree relational expressions* Draft of February 3, 2007.

[18] Oliveira, J., *A reification calculus for model-oriented software specification*, Formal Aspects of Computing**2** (1990), pp. 1–23.

[19] Oliveira, J., *'Fractal' Types: an Attempt to Generalize Hash Table Calculation*, in: *Workshop on Generic Programming (WGP'98), Marstrand, Sweden*, 1998.

[20] Oliveira, J., *Data transformation by calculation*, in: R. Lämmel et al., editors, *Generative and Transformational Techniques in Software Engineering*, LNCS **to appear** (2008).

[21] Peyton Jones, S., G. Washburn and S. Weirich, *Wobbly types: type inference for generalised algebraic data types*, Technical Report MS-CIS-05-26, Univ. of Pennsylvania (2004).