

# Constraint-aware Schema Transformation

Tiago L. Alves<sup>1,2</sup>, Paulo F. Silva<sup>1</sup>, and Joost Visser<sup>2</sup>

<sup>1</sup> University of Minho, Braga, Portugal

tiagomlalves@gmail.com and paufil@di.uminho.pt

<sup>2</sup> Software Improvement Group, Amsterdam, The Netherlands  
j.visser@sig.nl

**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, such as referential integrity constraints.

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.

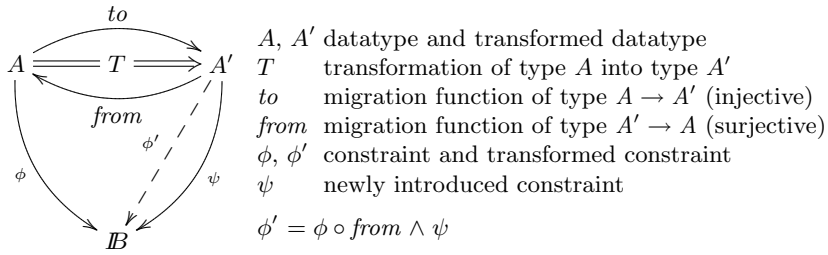
We implemented our rewrite systems in a type-safe way in the functional programming language Haskell. We demonstrate their application to constraint-aware hierarchical-relational mappings.

**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 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. For example, SQL database schemas and XSD document 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.



**Fig. 1.** Constraint-aware transformation of datatype  $A$  with constraint  $\phi$  into datatype  $A'$  with constraint  $\phi'$ . 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  $\psi$  introduced by the type-change. When  $\phi'$  is normalized it works on  $A'$  directly rather than via  $A$ .

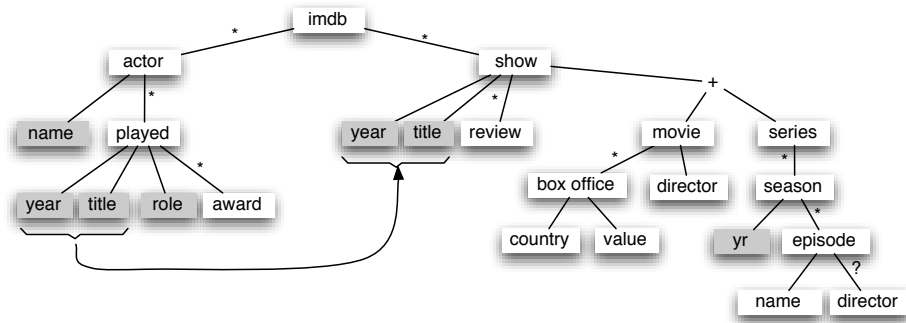
More complex schema transformations are involved in data mappings between programming paradigms [16], such as between XML and SQL.

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 these 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 also extended our approach to deal with propagation and introduction of constraints in a practical way [5]. However, this approach was not theoretically supported, did not achieve type-safeness, and was limited to the specific cases of referential integrity constraints.

In this paper, we propose an improved approach to constraint-aware schema transformation. Figure 1 concisely represents the new approach. Rather than labeling the types being transformed with 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  $\phi$  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



**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.

new constraint  $\psi$  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 the theoretical support of our approach in Section 3. This includes a brief overview of refinement theory and point-free program calculation. In Section 4, 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 5 to demonstrate the application of our rewriting system to schema-aware hierarchical-relational mapping. We discuss related work in Section 6 and conclude in Section 7.

## 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:

1. A `show` is identified by its year and title.
2. An `actor` is identified by his/her name.
3. A `season` is identified by its `yr`.
4. A `played` element is identified by its year, title, and role.

And the following referential integrity constraint:

5. 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,yr) references series(year,title)
episodes(id,year,title,yr,name,director?)
  foreign key (year,title,yr) references seasons(year,title,yr)
actors(name)
played(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 played(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 `played` 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, together with functions for backward and forward data conversion between the source and target schemas.

### 3 Refinement of datatypes with constraints

In this section, we will explain how constraint-aware 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. Based on this background information, we turn to the formalization of constraints in Section 3.3. Finally, Section 3.4 discusses how constraints can be added to data refinement laws to formalize the propagation and introduction of constraints during schema transformation.

### 3.1 Two-level transformation as data refinement

Data refinement theory provides an algebraic framework for calculating with datatypes [22,18,20]. The following inequation captures the essence of refining a datatype  $A$  to a datatype  $B$ :

$$A \begin{array}{c} \xrightarrow{to} \\ \leq \\ \xleftarrow{from} \end{array} B \text{ where } \begin{cases} to : A \rightarrow B \text{ injective and total} \\ from : B \rightarrow A \text{ surjective} \\ from \cdot to = id_A \end{cases}$$

Here,  $id_A$  is the identity function on datatype  $A$ . Thus, the inequation  $A \leq B$  expresses that  $B$  is a refinement of  $A$ , which is witnessed by the conversions 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 so-called point-free style of programming is found in [15,10]. Some function combinators and associated laws that are used in the current paper are shown in Figure 4.

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]. Below, we will use point-free program transformation to migrate and simplify constraints *during* schema transformation.

Sequential and structural composition	
$\text{if } A \begin{array}{c} \xrightarrow{to} \\ \leq \\ \xleftarrow{from} \end{array} B \text{ and } B \begin{array}{c} \xrightarrow{to'} \\ \leq \\ \xleftarrow{from'} \end{array} C \text{ then } A \begin{array}{c} \xrightarrow{to' \cdot to} \\ \leq \\ \xleftarrow{from \cdot from'} \end{array} C$	
$\text{if } A \begin{array}{c} \xrightarrow{to} \\ \leq \\ \xleftarrow{from} \end{array} B \text{ then } FA \begin{array}{c} \xrightarrow{F to} \\ \leq \\ \xleftarrow{F from} \end{array} FB$	
Hierarchical-relational data mapping	
$A^* \leq \mathbb{N} \rightarrow A$	List elimination
$2^A \cong A \rightarrow 1$	Set elimination
$A? \cong 1 \rightarrow A$	Optional elimination
$A + B \leq A? \times B?$	Sum elimination
$A \times (B + C) \cong (A \times B) + (A \times C)$	Distribute product over sum
$A \rightarrow (B + C) \leq (A \rightarrow B) \times (A \rightarrow C)$	Distribute map over sum (range)
$(B + C) \rightarrow A \cong (B \rightarrow A) \times (C \rightarrow A)$	Distribute map over sum (domain)
$A \rightarrow (B \times (C \rightarrow D)) \leq (A \rightarrow B) \times (A \times C \rightarrow D)$	Flatten nested map
Format evolution	
$A \leq A \times B$ Add field	$A^+ \leq A^*$ Allow empty list
$A \leq A + B$ Add alternative	$A? \leq A^*$ Allow repetition
$A \leq A?$ Make optional	$A \leq A^+$ Allow non-empty repetition

**Fig. 3.** Summary of data refinement theory. For a complete account, the reader is referred to Oliveira [22]. Note that  $\cdot \rightarrow \cdot$  denotes a simple relation, of which finite maps are a special case.

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

**Fig. 4.** Summary of point-free program transformation. For a complete account, the reader is referred to Cunha et al. [10].

### 3.3 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:

$$A_\phi \text{ where } \phi : A \rightarrow \mathbb{B} \text{ total and functional}$$

This notation, as well as some of the results below, originates in [21]. 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:

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

Here we use projection functions  $\pi_1$  and  $\pi_2$  to select the left or right table, we use  $\delta$  and  $\rho$  to select the domain and range of a map, and *set f* to map a function *f* over the elements of a set.

When a second constraint is added to a constrained datatype, both constraints can be composed with logical conjunction:

$$(A_\phi)_\psi \equiv A_{\phi \wedge \psi} \quad \text{logical composition}$$

Note that we use a variant of the conjunction operator lifted to point-free predicates:  $\cdot \wedge \cdot : (A \rightarrow \mathbb{B}) \rightarrow (A \rightarrow \mathbb{B}) \rightarrow (A \rightarrow \mathbb{B})$ .

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 [21]):

$$F(A_\phi) \equiv (FA)_{(F \phi)} \quad \text{functorial pull}$$

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

### 3.4 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:

$$\text{if } A \begin{array}{c} \xrightarrow{\text{to}} \\ \leq \\ \xleftarrow{\text{from}} \end{array} B \text{ then } A_\phi \begin{array}{c} \xrightarrow{\text{to}} \\ \leq \\ \xleftarrow{\text{from}} \end{array} B_{\phi \cdot \text{from}}$$

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:

$$A + B \cong A? \times B?_{(\epsilon \circ \pi_1) \oplus (\epsilon \circ \pi_2)}$$

$$A \multimap (B + C) \cong (A \multimap 1) \times (A \multimap B) \times (A \multimap C)_{(\delta \circ \pi_2 \subseteq \delta \circ \pi_1) \wedge (\delta \circ \pi_3 \subseteq \delta \circ \pi_1)}$$

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

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

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

$$\text{if } A \begin{array}{c} \xrightarrow{to} \\ \leq \\ \xleftarrow{from} \end{array} B_\psi \text{ then } A_\phi \begin{array}{c} \xrightarrow{to} \\ \leq \\ \xleftarrow{from} \end{array} (B_\psi)_{\phi \cdot from} \equiv B_{\psi \wedge (\phi \cdot from)}$$

This is the *invariant pulling* theorem of [21]. A more general case arises when not only the target, but also the source is constrained in the law that is applied:

$$\text{if } A_\chi \begin{array}{c} \xrightarrow{to} \\ \leq \\ \xleftarrow{from} \end{array} B_\psi \text{ and } \phi \Rightarrow \chi \text{ then } A_\phi \begin{array}{c} \xrightarrow{to} \\ \leq \\ \xleftarrow{from} \end{array} B_{\psi \wedge (\phi \cdot from)}$$

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

Constraints can not only be introduced and propagated. They can also be weakened or even eliminated, by virtue of the following:

$$\text{if } \phi \Rightarrow \psi \text{ then } A_\phi \leq A_\psi$$

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

## 4 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 4.1 we recall how type-safe representations of types and functions can be constructed using generalized algebraic datatypes (GADTs). In Section 4.2 we extend the type representation to constrained types. In Section 4.3 and 4.4, we explain how rewrite systems can be constructed to transform such constrained types.

### 4.1 Representation of types and functions

To represent both types and functions in a type-safe manner, we rely on *generalized algebraic data types* (GADTs) [23]. To represent types, we use a GADT:

```
data Type t where
  One  :: Type ()
```

```

List    :: Type a → Type [a]
Maybe :: Type a → Type (Maybe a)
Set     :: Type a → Type (Set a)
· → ·   :: Type a → Type b → Type (a → b)  -- We use lhs2TeX for
· + ·   :: Type a → Type b → Type (a + b)  -- type-setting various
· × ·   :: Type a → Type b → Type (a, b)    -- symbols in Haskell.
Func    :: Type a → Type b → Type (a → b)
String  :: Type String
Int     :: Type Int
Bool    :: Type Bool
...

```

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:

```

data F f where
  id    :: F (a → a)
  · ∘ · :: F (b → c) → F (a → b) → F (a → c)
  · ∧ · :: F (a →  $\mathbb{B}$ ) → F (a →  $\mathbb{B}$ ) → F (a →  $\mathbb{B}$ )
  · ⊆ · :: F (a → (Set b)) → F (a → (Set b)) → F (a →  $\mathbb{B}$ )
  π1  :: F ((a, b) → a)
  π2  :: F ((a, b) → b)
  · × · :: F (a → b) → F (c → d) → F ((a, c) → (b, d))
  · Δ · :: F (a → b) → F (a → c) → F (a → (b, c))
  ...

```

Note again 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 → b) → F (a → b) → a → b
```

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

## 4.2 Representation of constrained types

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

```

data Type t where
  ...
  (·) :: Type a → F (a →  $\mathbb{B}$ ) → Type a

```

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*:

```

check :: Type t → t → Bool
check One _      = True
check (t1 × t2) (x, y) = check t1 x ∧ check t2 y
...
check (tφ) x      = eval (Func t Bool) φ x ∧ check t x

```

Thus, this function descends through a type representation and the corresponding value. Each time it finds a constraint in the type it applies the *eval* function to check it on the value.

### 4.3 Rewriting types and functions

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

```

type RuleF = ∀ a b . F (a → b) → Maybe (F (a → 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:

```

idR :: RuleF
idR (f ∘ id) = return f
idR _       = mzero

```

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

```

nop :: RuleF           -- identity
(▷) :: RuleF → RuleF → RuleF -- sequential composition
(⊗) :: RuleF → RuleF → RuleF -- left-biased choice
many :: RuleF → RuleF         -- repetition
once :: RuleF → RuleF         -- arbitrary depth rule application

```

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:

```

simplify :: RuleF -- 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]:

```

type Rule = ∀ a . Type a → Maybe (View (Type a))
data View a where View :: (a→b) → (b→a) → Type b → View (Type a)

```

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].

#### 4.4 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.

```

flatMap :: Rule
flatMap (a → (b × (c → d))) = return (View ↯n-1 ↯n (tϕ))
  where
    t = (a → b) × ((a × c) → d)
    ϕ = ((set π1) ∘ (δ ∘ π2)) ⊆ (δ ∘ π1)
    flatMap t = propagate flatMap t

```

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

```

propagate rule (aϕ) = do
  (View to from b) ← rule a
  let ψ = ϕ ∘ from
  return $ View to from (bψ)
propagate _ _ = mzero

```

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  $\phi$  with the *from* to obtain the new constraint  $\psi$ . 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:

```

nestMap :: Rule
nestMap (((a → b) × ((- × c) → d))((set π1) ∘ (δ ∘ π2)) ⊆ (δ ∘ π1))
  = return $ View ↯n ↯n-1 (a → (b × (c → d)))
nestMap _ _ = mzero

```

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 [19]).

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

```

compose_constraint :: Rule
compose_constraint ((aφ)ψ) = return (View id id (a(φ∧ψ)))
compose_constraint _ = mzero

fuse_constraint :: Rule
fuse_constraint (aφ) = do
  ψ ← simplify φ
  return (View id id (aψ))
fuse_constraint _ = mzero

```

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.

## 5 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:

```

imdb = (actors × show)imdb_inv
  where imdb_inv = (set π1) ∘ fuse ∘ (set δ) ∘ ρ ∘ π1 ⊆ δ ∘ π2
show  = ("Year" × "Title") → ((List "Review") × (movie + series))
movie = (List ("Country" × "Value")) × "Director"
series = "Yr" → (List episode)
episode = "Name" × (Maybe "Director")
actors = "Name" → played
played = (("Year" × "Title") × "Role") → (List "Award")

```

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

```

(played × awards × actors × shows × reviews × seasons
 × episodes × series × movies × boxoffices)inv
  where
    played = ("Name" × "Year" × "Title" × "Role") → One
    awards = (("Name" × "Year" × "Title" × "Role") × Int) → "Award"
    actors = "Name" → One

```

```

shows      = ("Year" × "Title") → One
reviews    = (("Year" × "Title") × Int) → "Review"
seasons    = (("Year" × "Title") × "Yr") → One
episodes   = (((("Year" × "Title") × "Yr") × Int)
              → ("Name" × (Maybe "Director")))
series     = ("Year" × "Title") → One
movies     = ("Year" × "Title") → "Director"
boxoffices = (("Year" × "Title") × Int) → ("Country" × "Value")
inv        = fk1 ∧ fk2 ∧ fk3 ∧ fk4 ∧ fk5 ∧ fk6 ∧ fk7 ∧ fk8 ∧ fk9
fk1        = (set π1) ∘ δ ∘ πboxoffices ⊆ δ ∘ πmovies
fk2        = (set π1) ∘ δ ∘ πepisodes ⊆ δ ∘ πseasons
fk3        = (set π1) ∘ δ ∘ πseasons ⊆ δ ∘ πseries
fk4        = (set π2) ∘ δ ∘ πreviews ⊆ δ ∘ πshows
fk5        = δ ∘ πmovies ⊆ δ ∘ πshows
fk6        = δ ∘ πseries ⊆ δ ∘ πshows
fk7        = (set π1) ∘ δ ∘ πawards ⊆ δ ∘ πplayed
fk8        = (set π1 ∘ π1 ∘ π1) ∘ πplayed ⊆ δ ∘ πactors
fk9        = (set (π2 × id) ∘ π1) ∘ δ ∘ πplayed ⊆ δ ∘ πshows

```

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 *rewrite* rewrite system, the synthesized constraints would not be so concise. For example:

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

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:

```

> db' ← addActorsPlayed db (("Jet Li", (2001, "The One")), "Lawless")
> check imdbResult db'
False

```

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

```

> db' ← addShow db (2001, "The One")
> db'' ← addActor db' "Jet Li"
> db''' ← addActorsPlayed db'' (("Jet Li", (2001, "The One")), "Lawless")
> check imdbResult db'''
True

```

Now the constraint check succeeds.

## 6 Related work

A large number of approaches has been proposed for mapping XML to relational databases [6,7,2,4], but usually without taking constraints into account. Lee *et al* [17] first addressed the issue of constraint preservation. Their CPI algorithm deals with referential integrity constraints and some cardinality constraints, which are stored in an annotated DTD dependency 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.

## 7 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 [22,1,9,11,12,5]. In particular,

- we have shown how data refinement theory [22] can be enhanced to include types constrained by boolean predicates, extending the work of [21];

- we have enhanced refinement rules for hierarchical-relational mapping [1,5] such that appropriate constraints are introduced on target types;
- we have extended rewrite systems for two-level transformation [9] and coupled transformation [11] to include the propagation, introduction, and simplification of constraints.

Several directions of future work are envisioned.

*Bi-directional programming* Lenses are combinators for bi-directional programming [14,24]. They are used to tackle the classical *view-update problem* for databases, where a view is created from a database with a *get* function, and any changes to that view are written back to the database with a *put* function. Lenses and two-level transformations are different, but similar devices, and their interrelationships are an interesting point of study. We believe that our treatment of constraints could also be of value for bi-directional programming with lenses.

*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 [19] 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 with the rewrite system for structure-shy queries of [12] such that we can deal with structure-shy constraints.

**Acknowledgements** 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* under grant numbers SFRH/BD/30215/2006 and SFRH/BD/19195/2004, respectively.

## References

1. T.L. Alves, P.F. Silva, J. Visser, and J.N. Oliveira. Strategic term rewriting and its application to a VDM-SL to SQL conversion. In J. Fitzgerald, I.J. Hayes, and A. Tarlecki, editors, *FM*, volume 3582 of *LNCS*, pages 399–414. Springer, 2005.
2. 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.
3. J.W. Backus. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Commun. ACM*, 21(8):613–641, 1978.
4. 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.

5. P. Berdaguer, A. Cunha, H. Pacheco, and J. Visser. Coupled schema transformation and data conversion for XML and SQL. In Michael Hanus, editor, *PADL*, volume 4354 of *LNCS*, pages 290–304. Springer, 2007.
6. 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.
7. 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.
8. 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.
9. A. Cunha, J.N. 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*, volume 4085 of *LNCS*, pages 284–299. Springer, 2006.
10. A. Cunha and J. Sousa Pinto. Point-free program transformation. *Fundam. Inform.*, 66(4):315–352, 2005.
11. A. Cunha and J. Visser. Strongly typed rewriting for coupled software transformation. *ENTCS*, 174(1):17–34, 2007. Proc. 7th Int. Workshop on Rule-Based Programming (RULE 2006).
12. A. Cunha and J. Visser. Transformation of structure-shy programs: applied to XPath queries and strategic functions. In G. Ramalingam and Eelco Visser, editors, *PEPM*, pages 11–20. ACM, 2007.
13. S.B. Davidson et al. Propagating XML constraints to relations. In *Proc. 19th Int. Conf. on Data Engineering*, pages 543–. IEEE Computer Society, 2003.
14. J.N. Foster et al. Combinators for bi-directional tree transformations: a linguistic approach to the view update problem. In *POPL '05: Proc. 32nd ACM symp. on Principles of Programming Languages*, pages 233–246. ACM Press, 2005.
15. J. Gibbons. Calculating functional programs. In R. Backhouse et al., editors, *Algebraic and Coalgebraic Methods in the Mathematics of Program Construction*, volume 2297 of *LNCS*, chapter 5, pages 148–203. Springer, 2002.
16. R. Lämmel 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*, volume 4143 of *LNCS*. Springer, 2006.
17. Dongwon Lee and Wesley W. Chu. Cpi: Constraints-preserving inlining algorithm for mapping xml dtd to relational schema. *Data Knowl. Eng.*, 39(1):3–25, 2001.
18. C. Morgan and P.H.B. Gardiner. Data refinement by calculation. *Acta Informatica*, 27:481–503, 1990.
19. C.M. Necco, J.N. Oliveira, and J. Visser. Extended static checking by strategic rewriting of pointfree relational expressions. Draft of February 3, 2007.
20. J.N. Oliveira. A reification calculus for model-oriented software specification. *Formal Aspects of Computing*, 2(1):1–23, April 1990.
21. J.N. Oliveira. 'Fractal' Types: an Attempt to Generalize Hash Table Calculation. In *Workshop on Generic Programming (WGP'98), Marstrand, Sweden*, June 1998.
22. J.N. Oliveira. Data transformation by calculation. In R. Lämmel, J. Saraiva, and J. Visser, editors, *Generative and Transformational Techniques in Software Engineering*, volume to appear of *LNCS*. Springer, 2008.
23. 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.
24. M. Takeichi S.-C. Mu, Z. Hu. Bidirectionalizing tree transformation languages: A case study. *JSSST Computer Software*, 23(2):129–141, 2006.