Multifocal: A Strategic Bidirectional Transformation Language for XML Schemas

Hugo Pacheco and Alcino Cunha

HASLab / INESC TEC & Universidade do Minho, Braga, Portugal
{hpacheco,alcino}@di.uminho.pt

Abstract. Lenses are one of the most popular approaches to define bidirectional transformations between data models. However, writing a lens transformation typically implies describing the concrete steps that convert values in a source schema to values in a target schema. In contrast, many XML-based languages allow writing structure-shy programs that manipulate only specific parts of XML documents without having to specify the behavior for the remaining structure. In this paper, we propose a structure-shy bidirectional two-level transformation language for XML Schemas, that describes generic type-level transformations over schema representations coupled with value-level bidirectional lenses for document migration. When applying these two-level programs to particular schemas, we employ an existing algebraic rewrite system to optimize the automatically-generated lens transformations, and compile them into Haskell bidirectional executables. We discuss particular examples involving the generic evolution of recursive XML Schemas, and compare their performance gains over non-optimized definitions.

Keywords: coupled transformations, bidirectional transformations, two-level transformations, strategic programming, XML

1 Introduction

Data transformations are often coupled [16], encompassing software transformation scenarios that involve the modification of multiple artifacts such that changes to one of the artifacts induce the reconciliation of the remaining ones in order to maintain global consistency. A particularly interesting instance of this class are *two-level transformations* [18, 5], that concern the type-level transformation of schemas coupled with the value-level transformation of documents that conform to those schemas. A typical example of two-level transformations are format evolution scenarios [18, 11], such as schema changes occurring during maintenance operations or imposed by the natural evolution of the applications. These schema evolutions call for the coupled evolution of the underlying documents and related artifacts so that they remain consistent with the new schema.

Most existing XML transformation and querying languages, such as XSLT, XQuery or XPath, allow writing structure-shy programs that provide specific behavior only for the interesting bits of a (possibly huge) XML document without

having to specify how to traverse the remaining structure. Due to their generic form, such programs are easier to write and can be applied to documents satisfying different schemas. Nevertheless, they are not two-level. For example, using XSLT we can separately specify a transformation between XML schemas (since these can be represented as regular XML documents) and between XML documents, but the second is not a byproduct of the first. Thus, consistency between both levels must be manually verified, while a two-level transformation provides both transformations such that they are consistent by construction.

Another prominent instance of coupling are *bidirectional transformations*, as a "mechanism for maintaining the consistency of two (or more) related sources of information" [9]. For example, after a format evolution both old and new documents may co-exist and evolve independently. In a bidirectional transformation, the coupling occurs between forward and backward value transformations such that changes made to one of the data instances can be propagated to its connected pair in order to recover consistency.

Similarly to two-level transformations, a good approach is to design intrinsic bidirectional languages in which a program can be read both as a forward or a backward transformation, so that these are correct for the respective semantic space. Following this notion, many bidirectional languages have emerged in the most diverse computing domains, including many focused on the transformation of tree-structured data and with a particular application to XML documents [15, 3, 10, 21, 20, 14]. Among these, one of the most successful approaches are the so-called lenses, introduced by Foster *et al* [10] to solve the classical view-update problem: if a source model is abstracted into a view, how can updates made to the view be propagated back to the original model? They propose the *Focal* tree transformation language that allows users to build lenses with sophisticated synchronization behavior in a compositional way.

Still, the aforementioned bidirectional languages are at best typed but not two-level. On top of that, the programming style that grants them bidirectionality is usually more biased towards structure-sensitive constructs, to be able to identify precisely the concrete steps required to translate between source and target documents.

In this paper, we propose *Multifocal*, a generic structure-shy two-level transformation language for XML Schema evolution whose underlying value-level functions are bidirectional lens transformations that translate XML documents conforming to the old and new schemas. In comparison to a *Focal* lens transformation, that describes a bidirectional view between two particular tree structures, a *Multifocal* transformation describes a general type-level transformation (over XML Schemas) that provides multiple focus points, in the sense that it produces a different view schema and a corresponding bidirectional lens for each XML Schema to which it is applied successfully.

To describe such two-level transformations, we will use a generic style familiar of strategic rewriting languages [24, 19, 17], where the combination of a standard set of basic rules allows the design of flexible rewrite strategies in a compositional

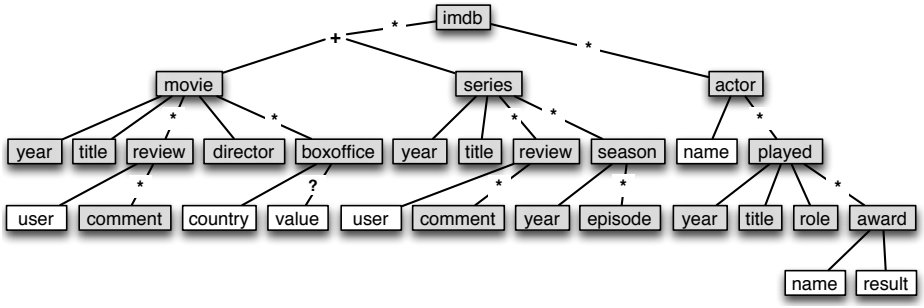

Fig. 1: Representation of a movie database schema inspired by IMDb (http://www.imdb.com). Grey boxes denote elements and white ones model attributes.

way, such as generic traversals that apply type-level transformations at arbitrary levels inside schema representations.

A known disadvantage of generic programs is their worse performance in comparison to analogous non-generic ones, since they must undergo runtime checks and blindly traverse whole input structures. In our framework, we mitigate this issue by encoding the underlying bidirectional lens transformations in a point-free[1] language with powerful algebraic laws and allowing automatic optimization by calculation [23], so that the optimized lens programs are able to efficiently propagate updates on XML documents.

In Section 2 we motivate our framework with an example. Section 3 presents the *Multifocal* language and discusses the design of our framework for the specification, optimization and execution of *Multifocal* transformations. The implementation of the framework (using the functional programming language Haskell) is shown in Section 4. Section 5 illustrates by example how our framework can tackle various application scenarios involving the generic evolution of recursive XML schemas, and compares the speedups achieved by an automatic optimization phase. In Section 6 we survey related work and Section 7 concludes the paper with a synthesis of the main contributions and directions for future work.

2 Motivating Example

Consider the XML Schema from Figure 1 representing an IMDb-like database for storing information about movies and actors. Imagine that we want to summarize this schema according to the following steps:

1. Delete all `series` elements.
2. For each `movie`, replace its `reviews` by a `popularity` attribute counting the number of `comments` and replace its `boxoffice` elements with a `profit` attribute summing the total `value` elements.
3. For each `actor`, keep its `name` and a list of `award` `names` renamed to `awname`.

[1] The point-free style is characterized by the lack of explicit "points" or variables.

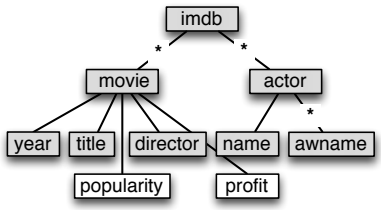

Fig. 2: A view of the movie database schema from Figure 1.

The resulting schema is shown in Figure 2. However, not only do we want to transform the XML schema, but also to migrate conforming XML documents and propagate updates in both directions: if a source document is modified, then a new view document must be computed; and if a view document is modified, then those changes shall be translated back into a modified source document. Consider, for example, the source XML document from Figure 3(a) containing one movie, one series and one actor. The forward transformation of our running example would produce the XML view shown in Figure 3(b). If we modify the view by correcting Uma Thurman's name and award information, and add an entry for the new Sherlock Holmes movie before the single `actor` element (Figure 3(c)), then the backward transformation shall correct the actress' name at the appropriate location and insert a new `movie` element with default `review` and `boxoffice` elements that are consistent with the modified view (Figure 3(d)).

In the remainder of this paper, we will propose a generic XML transformation language for XML schemas in which we can express the above transformation in a concise way close to its informal definition. Plus, the transformations for XML documents will come for free as conforming bidirectional lenses, satisfying strong round-tripping properties and supporting automatic optimization.

3 The Multifocal Framework

We now provide an overview of the *Multifocal* language and the respective framework for strategic two-level bidirectional transformation. We start with a formal definition of the bidirectional lenses at the core of our framework:

Definition 1 (Lens [10]). *A lens* $l{:}S \trianglerighteq V$ *comprises two total transformations* $get : S \to V$ *and* $put : V \times S \to S$*, satisfying the following properties:*

$$get\ (put\ (v, s)) = v \qquad \textsc{PutGet} \qquad put\ (get\ (s), s) = s \qquad \textsc{GetPut}$$

To give an idea of the bidirectional programs we are considering, these round-tripping properties guarantee that a lens is indeed an abstraction, i.e., the source schema S contains more information than the view schema V, and that backward propagation without modifications preserves the original documents.

Our two-level language over XML schemas is defined by instantiating a well-known suite of combinators for strategic programming [24, 19], together with specific XML transformers. The full syntax of *Multifocal* is defined as follows:

strat ::= `nop` | *strat* `>>` *strat* | *strat* `||` *strat* | `many` *strat* | `try` *strat*
| `all` *strat* | `once` *strat* | `everywhere` *strat* | `outermost` *strat*

```
<imdb>
 <movie>
  <year>2003</year>
  <title>Kill Bill: Vol. 1</title>
  <review user="emma">
   <comment>Gorgeous!</comment></review>
  <director>Quentin Tarantino</director>
  <boxoffice country="USA" value="22089322"/>
  <boxoffice country="Japan" value="3521628"/>
 </movie>
 <series><year>2011</year>
  <title>Game of Thrones</title>
  <season><year>2011</year>
   <episode>Winter is Coming</episode>
 </season></series>
 <actor name="Umma Thurman">
  <played><year>2003</year>
   <title>Kill Bill: Vol. 1</title>
   <role>The Bride</role>
   <award name="Saturn" result="Won"/>
 </played></actor>
</imdb>
```

(a) Source XML document

```
<imdb>
 <movie popularity="1" profit="25610950">
  <year>2003</year>
  <title>Kill Bill: Vol. 1</title>
  <director>Quentin Tarantino</director>
 </movie>
 <actor name="Umma Thurman">
  <awname>Saturn</awname>
 </actor>
</imdb>
```

(b) View XML document

```
<imdb>
 <movie> ... </movie>
 <series> ... </series>
 <movie><year>2012</year>
  <title>Sherlock Holmes: Game of Shadows</title>
  <review user="" comment=""/>
  <review user="" comment=""/>
  <director>Guy Ritchie</director>
  <boxoffice country="" value="15"/>
 </movie>
 <actor name="Uma Thurman">
  <played><year>2003</year>
   <title>Kill Bill: Vol. 1</title>
   <role>The Bride</role>
   <award name="Saturn Best Actress" result="Won"/>
 </played></actor>
</imdb>
```

(d) Modified source XML document

```
<imdb>
 <movie> ... </movie>
 <movie popularity="2" profit="15">
  <year>2012</year>
  <title>Sherlock Holmes: Game of Shadows</title>
  <director>Guy Ritchie</director>
 </movie>
 <actor name="Uma Thurman">
  <awname>Saturn Best Actress</awname>
 </actor>
</imdb>
```

(c) Modified view XML document

Fig. 3: Example of a bidirectional transformation between XML documents.

| `at` '"' *tag* '"' *strat* | `when` '"' *tag* '"' *strat*
| `hoist` | `plunge` '"' *tag* '"' | `rename` '"' *tag* '"'
| `erase` | `select` '"' *xpath* '"'

The set of strategic combinators allows to apply transformations sequentially (`>>`), alternatively (`||`), repetitively (`many`) or, more challengingly, at arbitrary depths inside schema representations. It also includes combinators for identity (`nop`) and optional rule application (`try`). Likewise [17] and other generic programming languages, instead of defining generic traversals by induction on the structure of types, we define a small set of traversal combinators. The `all` combinator applies a transformation to all immediate children of the current schema element (for the `imdb` element from Figure 1, these would be all `movie`, `series` and `actor` elements). The `once` traversal applies a given transformation exactly once somewhere inside a schema representation at an arbitrary depth, by traversing the schema in a top-down approach. Using `all`, we can define the `everywhere` combinator that traverses a schema representation in a bottom-up fashion and

applies the given transformation to all its descendants. The `outermost` traversal performs top-down exhaustive rule application and can be defined at the cost of `once`.

To control the application of certain rules, it is useful to identify locations inside schemas. The `at` combinator applies a given rule if the name of the current element matches a given XML element tag[2]. On the other hand, `when` takes the name of an XML Schema element and performs the following pattern matching: if the given element name is defined as a top-level element in the source schema, it converts its structure into a type-level predicate; then, if the predicate succeeds when applied to the current top-level element in the input schema (such that its structure matches the structure of the pattern element) it applies the argument rule, otherwise rule application fails. Other local combinators inspired in *Focal* [10] are: `hoist` that untags the current element, `plunge` that names a new XML element and `rename` that renames the current element.

As a language for defining views of schemas, *Multifocal* also supports specific abstraction combinators. To delete part of a schema, we simply call `erase` at the appropriate location. So far, our language builds generic transformations that describe the explicit changes that are performed on the source schema. An alternative way to specify generic programs is to perform queries that traverse arbitrary structures to collect values of a specific type, as in for example the XPath language for selecting particular nodes from XML documents. To apply an XPath query to a schema, we invoke the `select` combinator that attempts to bidirectionalize the XPath expression by converting it into a lens transformation that abstracts the schema into the desired result type.

As an example, the evolution scenario from Section 2 can be encoded as the following *Multifocal* transformation:

```
everywhere (try (at "series" erase))
>> everywhere (try (at "movie" (
 outermost (when "reviews" (
  select "count(//comment)" >> plunge "@popularity"))
 >> outermost (when "boxoffices" (
  select "sum(//@value)" >> plunge "@profit")))))
>> everywhere (try (at "actor" (
 outermost (at "played" (select "award/@name" >> all (rename "awname"))))))
```

This transformation deletes `series` elements by applying an `erase` (constrained by `at`) everywhere in the source schema, and the `popularity` and `profit` attributes are calculated using XPath queries (constrained by `when`) and tagged with `plunge`. The list of award names of an `actor` are selected with another XPath query, and such resulting `name` elements are renamed to `awname` by applying `rename` within the `all` traversal. In this transformation, the `reviews` and `boxoffices` tags used by the `when` combinator denote top-level XML Schema elements (Figure 4) that must be defined in the source XML Schema. They match lists of elements named `review` and `boxoffice` (using the schema representa-

[2] As in XPath, XML node names preceded by an ampersat "@" denote attributes.

```
<xs:group name="reviews"><xs:sequence>
  <xs:element name="review" minOccurs="0" maxOccurs="unbounded"/>
</xs:sequence></xsd:group>
<xs:group name="boxoffices"><xs:sequence>
  <xs:element name="boxoffice" minOccurs="0" maxOccurs="unbounded"/>
</xs:sequence></xs:group>
```

Fig. 4: XML Schema top-level elements modeling specific type patterns.

tions introduced in Section 4, they denote the types $[\mu_{review} F]$ and $[\mu_{boxoffice} G]$, for arbitrary functors F and G), respectively.

The general architecture of our framework is illustrated in Figure 5. A two-level transformation defined as a *Multifocal* expression is executed in two stages: first, it is evaluated as a type-level transformation by applying it to a source XML Schema, producing a target XML Schema and a bidirectional lens; second, the lens is compiled into an executable file that can be used to propagate updates between XML documents conforming to the source and target schemas. In our scenario, optimization is done at the second stage: we optimize the value-level lenses once for each input schema and generate optimized executables that efficiently propagate updates between XML documents.

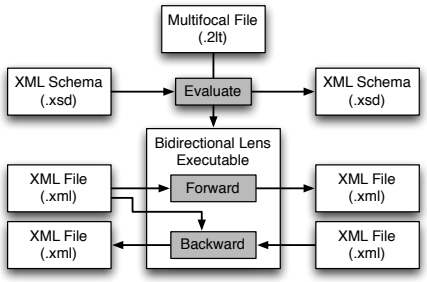

Fig. 5: Architecture of the *Multifocal* framework.

4 Implementation

This section unveils the implementation of the *Multifocal* framework in Haskell. Haskell is a general-purpose functional programming language with strong static typing, where structures are modeled by algebraic data types and programs are written as well-typed functions through pattern matching on their input values. This embedding is supported by front-ends that translate XML Schemas and XML documents into Haskell types and values, and vice-versa. A more technical description of similar XML-Haskell front-ends can be found in previous work [2].

Two-level transformations written in *Multifocal* are translated into a core library of Haskell combinators that operate on Haskell type representations. After translating the source XML Schema into an Haskell type, the framework applies the type-level transformation to produce as output a target type and a lens representation as Haskell values. From these, it generates a target XML Schema and an Haskell executable file containing the lens transformation and the data type declarations that represent all the source and target XML elements.

$\circ : (B \trianglerighteq C) \to (A \trianglerighteq B) \to (A \trianglerighteq C)$ $\quad$ $id : A \trianglerighteq A$

$\times : (A \trianglerighteq C) \to (B \trianglerighteq D) \to (A \times B) \trianglerighteq (C \times D)$ $\quad$ $\pi_1 : A \times B \trianglerighteq A$

$+ : (A \trianglerighteq C) \to (B \trianglerighteq D) \to (A + B) \trianglerighteq (C + D)$ $\quad$ $\pi_2 : A \times B \trianglerighteq A$

$\triangledown : (A \trianglerighteq C) \to (B \trianglerighteq C) \to (A + B) \trianglerighteq C$ $\quad$ $! : A \trianglerighteq 1$

$(\!|g|\!)_F : (F\ A \trianglerighteq A) \to (\mu F \trianglerighteq A)$ $\quad$ $out_F : \mu F \trianglerighteq F\ (\mu F)$

$[\!(g)\!]_F : (A \trianglerighteq F\ A) \to (A \trianglerighteq \mu F)$ $\quad$ $in_F : F\ (\mu F) \trianglerighteq \mu F$

$map : (A \trianglerighteq B) \to ([A] \trianglerighteq [B])$ $\quad$ $concat : [[A]] \trianglerighteq [A]$ $\quad$ $length : [A] \trianglerighteq Int$

$filter\_l : [A + B] \trianglerighteq [A]$ $\quad$ $filter\_r : [A + B] \trianglerighteq [B]$

Fig. 6: Point-free lens combinators.

The main function of this file parses XML documents complying to the schemas, converts them into internal Haskell values, runs the lens transformation either in the forward or backward direction to propagate source-to-target or target-to-source updates, and finishes by pretty-printing an updated XML document.

Encoding of schemas and lenses In Haskell, sums + and products × correspond to `xs:sequence` and `xs:choice` elements in XML Schema notation. Primitives include base types, such as the unit type 1, integers *Int* or strings *String*, and lists $[A]$ of values of type A that model XML sequences. To accommodate recursive schemas, we represent user-defined types (denoting XML elements) as fixpoints $\mu_{tag} F$ of a polynomial functor F, for a given name *tag*[3]. A polynomial functor is either the identity *Id* (for recursive invocation), the constant $\underline{A}$, the lifting of sums $\oplus$, products $\otimes$ and lists $[\,]$ or the composition of functors $\odot$. For example, the top-level element of the non-recursive schema from Figure 1 is represented as $\mu_{imdb}(([\,] \odot (\underline{\mu_{movie} M} \oplus \underline{\mu_{series} S})) \otimes ([\,] \odot \underline{\mu_{actor} A}))$, where M,S and A are the functors of the `movie`, `series` and `actor` elements. Application of a polynomial functor F to a type A yields an isomorphic sum-of-products type $F\ A$.

In our framework, bidirectionality is achieved by defining the value-level semantics of our two-level programs according to the point-free lens language developed in [22] and summarized in Figure 6. Each of these lens combinators possesses a *get* and a *put* function satisfying the bidirectional properties from Definition 1[4]. Fundamental lenses are identity (*id*) and composition ($\circ$). The $!$,π_1 and π_2 combinators project away parts of a source type, while $\triangledown$ applies two lenses alternatively to distinct sides of a sum. The $\times$ and $+$ combinators map two lenses to both sides of a pair or a sum, respectively. The out_F and in_F isomorphisms expose and encapsulate the top-level structure of an inductive type with functor F. The well-known fold $(\!|\cdot|\!)_F$ and unfold $[\!(\cdot)\!]_F$ recursion patterns recursively consume and produce values of an inductive type. In this paper, we treat some typical operations over lists such as mapping, concatenation, length and filtering as primitive lenses. Their recursive definitions can be found in [23].

[3] This is actually one of many possible representations of algebraic data types for use in generic programming. For a detailed discussion see [13].

[4] In [22], some of the lens combinators admit additional parameters to control value generation. In this paper, we substitute such parameters with suitable defaults.

```
nop, erase :: Rule
nop a = return (id, a)
erase a = return (!, 1)

at :: String → Rule → Rule
at name r a@(μtag f) | name ≡ tag = r a
at name r a = mzero

all :: Rule → Rule
all r Int = return (id, Int)
all r [a] = do (l, b) ← r a
               return (map l, [b])
all r μtag f = do (l, g) ← allF r f
                  return ((|in_g ∘ l|)_f, μtag g)
...
allF :: Rule → RuleF
allF r Id = return (id, Id)
allF r [] = return (id, [])
allF r a = do (l, b) ← r a
              return (l, b)
allF r (f ⊗ g) = do (l1, h) ← allF r f
                    (l2, i) ← allF r g
                    return (l1 × l2, h ⊗ i)
...
everywhere r = all (everywhere r) >> r

once :: Rule → Rule
once r Int = r Int
once r [a] =  r [a] `mplus`
              do (l, b) ← once r a
                 return (map l, [b])
once r a@(μtag f) = r a `mplus`
  do (l, g) ← onceF r f
     return ([[l ∘ out_f]]_g, μtag g)
...
type RuleF = Fctr → Maybe (Lens, Fctr)
onceF :: Rule → RuleF
onceF r Id = mzero
onceF r [] = do (l, g •) ← r [•]
                return (l, g)
onceF r (f ⊗ g) =
  do (l, h •) ← r ((f ⊗ g) •)
     return (l, h)
  `mplus` do (l, h) ← onceF r f
             return (l × id, h ⊗ g)
  `mplus` do (l, i) ← onceF r g
             return (id × l, f ⊗ i)
...
outermost r = many (once r)
```

Fig. 7: Encoding of some strategic combinators as Haskell rewrite rules.

Two-level lens transformations Multifocal combinators can be encoded as rewrite rules that, given a source type representation, yield a lens representation and a target type representation:[5]:

type *Rule* = *Type* → *Maybe* (*Lens*, *Type*)

In our implementation, types and lenses are represented as values of type *Type* and *Lens* (a grammar for lenses built using the combinators from Figure 6). The *Maybe* Haskell type models partiality of rule application: *return* denotes successful application, failure is signaled with *mzero* and *mplus* implements left-biased choice. Figure 7 presents the encoding of some combinators, namely the fundamental `all` and `once` that traverse inside the functorial structure of types.

The `all` traversal applies an argument rule to all children of the current type and has the most interesting behavior for user-defined types: it invokes the auxiliary rule *allF* that propagates a rule application down to the constants, where it applies the argument rule, and returns a lens transformation (wrapped

[5] For a clearer presentation, we encode types and transformations with unconstrained data representations. Our actual implementation follows a type-safe encoding inspired in [5], such that the conformity between all the artifacts (schemas,documents and transformations) is enforced by the Haskell type system.

as a rewrite rule *RuleF* on functor representations *Fctr*); then, it constructs a bottom-up lens (fold) that recursively applies the lens transformation to all values of the recursive type. The `once` traversal applies an argument rule exactly once at an arbitrary depth in a top-down approach, and stops as soon as the argument rule can be successfully applied. To be able to apply normal type rules inside a functor, the auxiliary rule *onceF* flattens the functor by applying it to a special type mark $\bullet$. When the argument rule can be successfully applied, it infers a new functor representation using $\bullet$ to remember the recursive invocations[6]. For recursive types, the resulting lens performs a top-down traversal (unfold) that applies the value-level transformations of the argument rule to each recursive.

Other combinators for processing user-defined types are: `hoist` that unpacks an user-defined type by applying *out* at the value-level; `plunge` that constructs a new (non-recursive) user-defined type by applying *in* at the value-level; and `rename` that renames an existing user-defined type and is coupled to the *id* lens. Notice that `rename n` is different from `hoist >> plunge n`, since rename works for all data types, and plunge can only create non-recursive ones.

The `erase` combinator deletes the current top-level type, by replacing it with the unit type and applying ! at the value-level. In order to bundle a XPath query as a two-level transformation, the `select` combinator specializes it for the input type and then tries to lift the specialized expression into a lens. We specialize XPath expressions by translating them into generic point-free programs than can be optimized to non-generic point-free functions using the techniques from [8, 6]. To lift the resulting functions into lenses, we check if their point-free expressions are defined using only the point-free lens combinators from Figure 6, otherwise rule application fails.

Schema normalization To keep a minimal suite of combinators, our language supports abstractions through the `erase` combinator, that deletes elements locally and thus leaves "dangling" unit types in the target schema. However, these empty unnamed types are unintended and may yield XML Schemas that are deemed ambiguous by many XML processors. For example, when applying our running *Multifocal* transformation to the IMDb schema from Figure 1, deleting `series` inside `imdb` elements will result in a list $[\mu_{movie} M + 1]$. Such dangling unit types have no representation in the XML side and must be deleted from the target schema representation. Such deletion is performed by a *normalize* procedure that removes these and other ambiguities, by exhaustively applying the rules from Figure 8[7]. Normalization is silently applied by extending the `all` and `once` traversals so that they apply *normalize* after rewriting.

Lens optimization Although the lens transformations generated by our framework are instantiated for particular source and target schemas, they still contain many redundant computations and traverse the whole structures, as a consequence of being a two-level transformation. To improve their efficiency, we reuse

[6] Unlike in the pseudo-code from Figure 7, in our implementation functor inference must be performed as a separate procedure and not simply via pattern matching.

[7] The exact lens definitions of $id \triangledown nil$ and $nil \triangledown id$ can be found in [23].

$\pi_1 : A \times 1 \trianglerighteq A$	$\pi_2 : 1 \times A \trianglerighteq A$	-- Products
$id \triangledown nil : [A] + 1 \trianglerighteq [A]$	$nil \triangledown id : 1 + [A] \trianglerighteq [A]$	-- Sums
$\mathit{filter\_l} : [A + 1] \trianglerighteq [A]$	$\mathit{filter\_r} : [1 + A] \trianglerighteq [A]$	-- Lists
$id \triangledown id : A + A \trianglerighteq A$	$concat : [[A]] \trianglerighteq [A]$	-- Ambiguous types

Fig. 8: Rules for normalization of XML Schema representations.

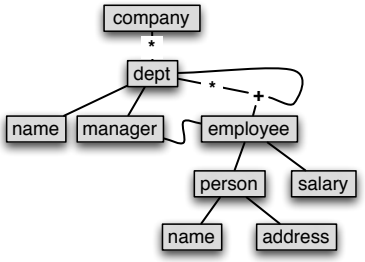

Fig. 9: A company hierarchized payroll XML schema inspired in [17].

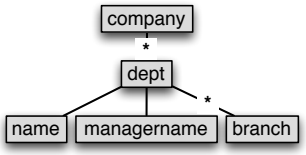

Fig. 10: A view of the company schema.

a rewrite system for the optimization of point-free lenses [23] (similar to the one for the optimization of XPath queries) that employs powerful algebraic point-free laws for fusing and cutting redundant traversals. After rewriting, the resulting transformations work directly between the source and target types and are significantly more efficient, as demonstrated in Section 5. In our framework, we provide users with the option to optimize the generated bidirectional programs at the time of generation of the Haskell bidirectional executable, if they intend to repeatedly propagate updates between XML documents conforming to the same schemas. This could be the case, for example, when the schemas represent the configuration of a live system that replies to frequent requests. In such cases, the once-a-time penalty of an additional optimization phase for a specific schema is amortized by a larger number of executions.

5 Application Scenarios

We now demonstrate two XML evolution scenarios (the IMDb example from Section 3 and another example for the evolution of a recursive XML Schema), and compare the performance of the lenses resulting from the execution of the two-level transformations with their automatically optimized definitions.

A classical schema used to demonstrate strategic programming systems is the so called "paradise benchmark" [17]. Suppose one has a recursive XML Schema to model a company with several departments, each having a name, a manager and a collection of employees or sub-departments, illustrated in Figure 9. Our second evolution example consists in creating a view of this schema according to the following transformation:

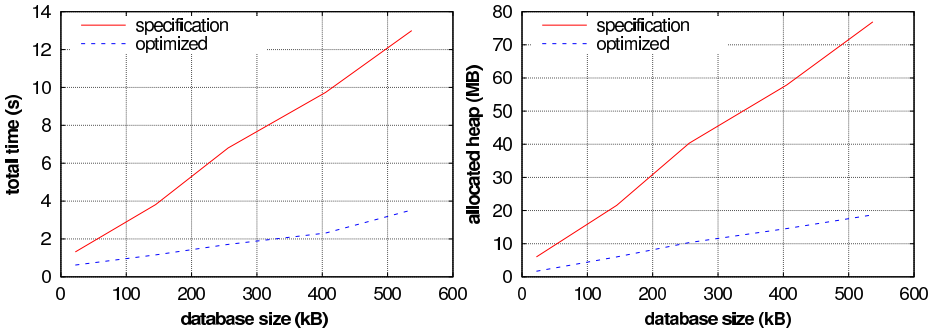

Fig. 11: Benchmark results for the *IMDb* example.

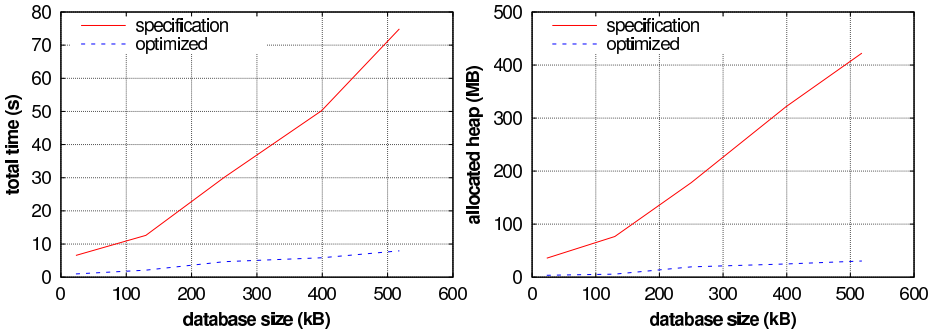

Fig. 12: Benchmark results for the *paradise* example.

```
everywhere (try (at "manager" (
 all (select "(//name)[1]") >> rename "managername")))
>> everywhere (try (at "employee" erase))
>> once (at "dept" (hoist >> outermost (at "dept" (
   select "name" >> rename "branch")) >> plunge "dept"))
```

For each top-level department, this transformations keeps only the `name`s of `manager`s (renamed to `managername`), deletes all `employee`s and collects the names of direct sub-departments renamed to `branch`. The resulting non-recursive schema is depicted in Figure 10. There are some details worth noticing. First, it is easier to keep only manager names using a generic query instead of a transformation that would need to specify how to drop the remaining structure. The XPath filter "[1]" guarantees a sole result if multiple names existed under `manager`s. Second, since `dept` is a recursive type, we unfold its top-level recursive structure once using `hoist` to be able to process sub-branches, and create a new non-recursive `dept` element with `plunge`.

Performance Analysis Unfortunately, the lenses resulting from the above transformations are not very efficient. For instance, in the IMdb example the traversals over `series`' `movie`s and `actor`s are independent and can be done in parallel.

Also, the transformations of `reviews` and `boxoffices` and the extra normalizing step that filters out unit types (resulting from erased `series` elements) can be fused into a single traversal. For the paradise example, all the three steps and the extra normalization step (for erased `employee` elements) can be fused into a single traversal. Also, the first two steps, that traverse all departments recursively due to the semantics of `all` (invoked by `everywhere`) for recursive types, are deemed redundant for sub-departments by the last step.

All these optimizations can be performed by our lens optimization phase. We have measured space and time consumption of the lenses generated by our two examples, and the results are presented in Figures 11 and 12. To quantify the speedup achieved by the optimizations, we have compared the runtime behavior of their backward transformations for non-optimized (*specification*) and optimized lens definitions (*optimized*)[8]. To factor out the cost of parsing and pretty-printing XML documents, we have tested the *put* functions of the lenses with pre-compiled input databases of increasing size (measured in kBytes needed to store their Haskell definitions), randomly generated with the *QuickCheck* testing suite [4]. We compiled each function using `GHC 7.2.2` with optimization flag `O2`. As expected, the original specification performs much worse than the optimized lens, and the loss factor grows with the database size. Considering the biggest sample, the loss factors are of 3.7 in time and 4.1 in space for the IMDb example and of 9.4 in time and 13.4 in space for the paradise example. The more significant results (and the worse overall performance) for the paradise example are justified by the elimination of the recursive traversals over sub-departments.

6 Related Work

In [18], Lämmel *et al* propose a systematic approach to XML schema evolution, where the XML-based formats are transformed in a step-wise fashion and the transformation of coupled XML documents can be largely induced from the schema transformations. They study the properties of such transformations and identify categories of XML schema evolution steps, taking into account many XML-specific issues, but do not propose a formalization or implementation of such a general framework for two-level transformation. The X-Evolution system [11] provides a graphical interface for the evolution of XML Schemas coupled with the adaptation of conforming XML documents. Document migration is automated for the cases when minimal document changes can be inferred from the schema evolution steps, while user intervention through query-based adaptation techniques is required to appropriately handle more complex schema changes.

Two famous bidirectional languages for XML are XSugar [3] and biXid [15], that describe XML-to-ASCII and XML-to-XML mappings, respectively. In both, bidirectional transformations are specified using pairs of intertwined grammars describing the source and target formats, from which a forward transformation is obtained by parsing according to the rules in one grammar and a backward

[8] Note that parsed XPath expressions are already optimized in the non-optimized lens, since their successful "lensification" depends on their specialization.

transformation by pretty printing according to the rules in the other. However, while the emphasis of XSugar is on bijective transformations, biXid admits ambiguity in the transformations and only postulates that translated documents shall be consistent up to the grammar specification.

The *Focal* lens language [10] provides a rich set of lens combinators, from general functional programming features (composition, mapping, recursion) to tree-specific operations (splitting, pruning, merging) for the transformation of tree-structured data. In [20], Liu *et al* propose Bi-X, a functional lens language closely resembling the XQuery Core language that can serve as the host language for the bidirectionalization of XQuery. The main feature of Bi-X is its support for variable binding, allowing lenses that perform implicit duplication. Using variable referencing, structure-shy combinators such as XPath's descendant axis can also be translated into equivalent Bi-X programs. Although their development is done in an untyped setting, they define a type system of regular expressions that is used to refine backward behavior.

In previous work [2], we proposed a two-level bidirectional transformation framework (2LT) implemented in Haskell for the strategic refinement of XML Schemas into SQL databases. Later [7], we showed how point-free program calculation can be used to optimize such bidirectional programs and how these can be combined to provide structure-shy query migration. In this paper, we tackle the dual problem of XML schema abstraction, supporting the execution and optimization of coupled bidirectional lenses. While refinement scenarios are inherent to strategic rewriting techniques, like the automatic mapping of abstract schemas to more concrete ones [2], view definition scenarios typically involve more surgical steps that abstract or preserve specific pieces of information and motivate a different language of primitive evolution steps. Contrarily to the 2LT framework, where two-level transformations are written within Haskell using a combinator library, we propose the *Multifocal* XML transformation language, mixing strategic and specific XML transformers, to write "out of the box" views of XML Schemas. Another new feature of our approach is the specification and optimization of generic two-level transformations over recursive XML Schemas.

7 Conclusion

In this paper we have proposed *Multifocal*, a generic two-level bidirectional transformation language for XML Schema evolution with document-level migrations based on the bidirectional framework of lenses. By using strategic programming techniques, these coupled transformations can be specified in a concise and generic way, mimicking the typical coding pattern of XML transformation languages such as XSLT, that allow to easily specify how to modify only selected nodes via specific templates. When applied to input schemas, our schema-level transformations produce new schemas, as well as bidirectional lens transformations that propagate updates between old and new documents. In our framework, we release such bidirectional transformations as independent programs that can be used to translate updates for particular source and target schemas. We also

provide users with an optional optimization phase that improves the efficiency of the generated lens programs for intensive usage scenarios.

Our framework has been fully implemented in Haskell, and is available through the Hackage package repository (`http://hackage.haskell.org`) under the name `multifocal`. It can be used both as a stand-alone tool for XML Schema evolution and as a combinator library for the two-level bidirectional evolution of arbitrary inductive data type representations.

Although our language already supports combinators in the style of XSLT transformations and XPath queries, the expressiveness of the underlying bidirectional transformations is naturally limited by the language of point-free lenses in use. That said, the translation of some XPath features that are not perfect abstractions, such as value-level filtering, is not considered in our approach. In future work, we plan to extend this language to support more XPath features. This would require, however, to loosen either the round-tripping laws or the requirement that lens functions must be totally defined for documents conforming to the schemas. Other directions for future work that we are investigating are: (i) the processing of annotations in the input XML Schemas such as `xs:key` to identify reorderable chunks in the source document and provide extra alignment information to guide the translation of view updates in the style of [1]; (ii) the leveraging of the underlying bidirectional framework from asymmetric lenses to other symmetric formulations such as [12] that guarantee weaker properties but do not impose a particular abstract-or-refine data flow.

In this work, we propose a way of replacing three unidirectional XML transformations (a schema-level transformation and two transformations between XML documents) with a single two-level bidirectional *Multifocal* transformation. In order to bring *Multifocal* closer to standard XML transformation tools, we plan to develop translations from XSLT-like idioms to *Multifocal*. For a successful integration, a comparative study on the usefulness, expressiveness and efficiency of *Multifocal* transformations would be needed.

Acknowledgments

This work is funded by the ERDF through the programme COMPETE and by the Portuguese Government through FCT (Foundation for Science and Technology), project reference FCOMP-01-0124-FEDER-020532.

References

1. D. M. J. Barbosa, J. Cretin, J. N. Foster, M. Greenberg, and B. C. Pierce. Matching lenses: alignment and view update. In *ICFP'10*, pages 193–204. ACM, 2010.
2. P. Berdaguer, A. Cunha, H. Pacheco, and J. Visser. Coupled schema transformation and data: Conversion for XML and SQL. In *PADL'07*, volume 4354, pages 290–304. Springer, 2007.
3. C. Brabrand, Anders Møller, and M. I. Schwartzbach. Dual syntax for xml languages. *Information Systems*, 33:385–406, 2008.

4. K. Claessen and J. Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *ICFP'00*, pages 268–279. ACM, 2000.
5. A. Cunha, J. N. Oliveira, and J. Visser. Type-safe two-level data transformation. In *FM'06*, volume 4085, pages 284–299. Springer, 2006.
6. A. Cunha and H. Pacheco. Algebraic specialization of generic functions for recursive types. *ENTCS*, 229(5):57–74, 2011.
7. A. Cunha and J. Visser. Strongly typed rewriting for coupled software transformation. *ENTCS*, 174(1):17–34, 2007.
8. A. Cunha and J. Visser. Transformation of structure-shy programs with application to xpath queries and strategic functions. *Science of Computer Programming*, 76(6):512–539, 2011.
9. K. Czarnecki, J. N. Foster, Z. Hu, R. Lämmel, A. Schürr, and J. Terwilliger. Bidirectional transformations: A cross-discipline perspective. In *ICMT'09*, volume 5563 of *LNCS*, pages 260–283. Springer, 2009.
10. J. N. Foster, M. B. Greenwald, J. T. Moore, B. C. Pierce, and A. Schmitt. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *TOPLAS'07*, 29(3):17, 2007.
11. G. Guerrini and M. Mesiti. X-evolution: A comprehensive approach for xml schema evolution. In *DEXA'08*, pages 251–255. IEEE, 2008.
12. M. Hofmann, B. C. Pierce, and D. Wagner. Symmetric lenses. In *POPL'11*, pages 371–384. ACM, 2011.
13. S. Holdermans, J. Jeuring, A. Löh, and A. Rodriguez. Generic views on data types. In *MPC'06*, volume 4014 of *LNCS*, pages 209–234. Springer, 2006.
14. Z. Hu, S.-C. Mu, and M. Takeichi. A programmable editor for developing structured documents based on bidirectional transformations. *Higher Order and Symbolic Computation*, 21(1-2):89–118, 2008.
15. S. Kawanaka and H. Hosoya. bixid: a bidirectional transformation language for xml. In *ICFP'06*, pages 201–214. ACM, 2006.
16. R. Lämmel. Coupled Software Transformations (Extended Abstract). In *1st International Workshop on Software Evolution Transformations*, 2004.
17. R. Lämmel and S. P. Jones. Scrap your boilerplate: A practical design pattern for generic programming. In *TLDI'03*, pages 26–37. ACM, 2003.
18. R. Lämmel and W. Lohmann. Format Evolution. In *RETIS'01*, volume 155, pages 113–134. OCG, 2001.
19. R. Lämmel and J. Visser. A strafunski application letter. In *PADL'03*, volume 2562 of *LNCS*, pages 357–375. Springer, 2003.
20. D. Liu, Z. Hu, and M. Takeichi. Bidirectional interpretation of xquery. In *PEPM'07*, pages 21–30. ACM, 2007.
21. S.-C. Mu, Z. Hu, and M. Takeichi. An algebraic approach to bi-directional updating. In *APLAS'04*, volume 3302 of *LNCS*, pages 2–20. Springer, 2004.
22. H. Pacheco and A. Cunha. Generic Point-free Lenses. In *MPC'10*, volume 6120, pages 331–352. Springer, 2010.
23. H. Pacheco and A. Cunha. Calculating with lenses: optimising bidirectional transformations. In *PEPM'11*, pages 91–100. ACM, 2011.
24. E. Visser. Stratego: A language for program transformation based on rewriting strategies system description of stratego 0.5. In *RTA'01*, volume 2051 of *LNCS*, pages 357–361. Springer, 2001.