Science of Computer Programming 76 (2011) 516–539

Contents lists available at ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico

# Transformation of structure-shy programs with application to XPath queries and strategic functions

Alcino Cunha [a,*], Joost Visser [b,c]

[a] DI-CCTC, Universidade do Minho, Portugal
[b] Software Improvement Group, The Netherlands
[c] Centrum Wiskunde & Informatica, The Netherlands

ARTICLE INFO

*Article history:*
Received 28 November 2007
Received in revised form 23 December 2009
Accepted 6 January 2010
Available online 15 January 2010

*Keywords:*
Algebraic program transformation
Strategic functional programming
XML query languages
Point-free program calculation
Type specialization
Type generalization

ABSTRACT

Various programming languages allow the construction of structure-shy programs. Such programs are defined generically for many different datatypes and only specify specific behavior for a few relevant subtypes. Typical examples are XML query languages that allow selection of subdocuments without exhaustively specifying intermediate element tags. Other examples are languages and libraries for polytypic or strategic functional programming and for adaptive object-oriented programming.

In this paper, we present an algebraic approach to transformation of declarative structure-shy programs, in particular for strategic functions and XML queries. We formulate a rich set of algebraic laws, not just for transformation of structure-shy programs, but also for their conversion into structure-sensitive programs and *vice versa*. We show how subsets of these laws can be used to construct effective rewrite systems for specialization, generalization, and optimization of structure-shy programs. We present a type-safe encoding of these rewrite systems in Haskell which itself uses strategic functional programming techniques. We discuss the application of these rewrite systems for XPath query optimization and for query migration in the context of schema evolution.



## 1. Introduction

Structure-shy programming techniques have been introduced for dealing with highly structured data such as terms, semi-structured documents, and object graphs in a largely generic manner. A structure-shy program specifies type-specific behaviour for a selected set of data constructors only. For the remaining structure, generic behaviour is provided. Prominent flavours of structure-shy programming are adaptive programming [31], strategic programming [38,28,29,40], polytypic or type-indexed programming [17,19], and several XML programming languages and APIs [44,47].

Structure-shy programming offers various clear benefits [27,41]. The elimination of boilerplate code makes a structure-shy program significantly more concise, focusing on the essence of the algorithm. This reduces development time and improves understandability. Also, structure-shy programs are only loosely bound to the data structures on which they operate. As a result, they do not necessarily need adaptation when those data structures evolve, and they may be reusable for different data structures.

The flip side of these benefits is that structure-shy programs have potentially worse space and time behaviour than equivalent structure-sensitive programs. A source of performance loss, generally by a factor linear in the input size, is the dynamic checks employed in the execution of structure-shy programs to determine at each data node whether to apply

* Corresponding author.
*E-mail addresses:* alcino@di.uminho.pt (A. Cunha), j.visser@sig.nl (J. Visser).

0167-6423/$ – see front matter © 2010 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2010.01.003

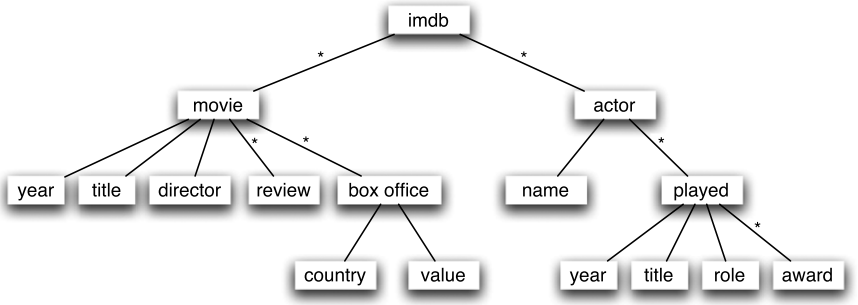



**Fig. 1.** Representation of a movie database schema, inspired by www.imdb.com.

specific or generic behaviour. Another source of inefficiency is that algorithmic optimizations, such as cutting off traversal into certain substructures, cannot be expressed without to some extent sacrificing structure-shyness and its benefits. In fact, manual optimization of structure-shy programs typically involves such sacrifice.

For adaptive programming and polytypic programming, substantial effort has been invested in the development of optimizing compilers. Compilation schemes for Generic Haskell and Generic Clean specialize and optimize polytypic input programs for specific types [43,1,2]. Adaptive programs are compiled to plain object-oriented programs with optimized navigation behaviour [30].

In this paper, we present an approach to transformation of structure-shy programs that encompasses typed strategic programming and XML programming. Our approach builds on the pioneering work of Backus [5] and the ensuing tradition of algebraic transformation of point-free functional programs [16,12]. Falling back to this tradition is natural, since both structure-shy strategic programs and path-based XML processors are basically point-free: they are composed from basic combinators and the terms and document elements they manipulate are not bound to intermediate variables.

Algebraic program transformation laws can be formulated for structure-shy strategic programs and XML processors, just as they have been formulated for structure-sensitive point-free functional programs. Further laws can be formulated that mediate between structure-shy and structure-sensitive programs by type-specialization and generalization. Such laws can be leveraged into useful rewrite systems, not only for optimization of structure-shy programs, but also conversely for increasing a program's degree of structure-shyness, which may have its use in program understanding, refactoring, or re-engineering. We will show how such rewrite systems can be implemented with strong type-safeness guarantees using the functional programming language Haskell.

Our approach to transformation of structure-shy programs has been demonstrated to be useful for optimization of XPath queries [15] and for query and constraint migration in the context of coupled transformation of data models, data instances, queries, and constraints [42,14]. In general, our approach may find applications in compilation, static checking, refactoring, reverse engineering, and migration of programs that employ XPath expressions and/or rewriting strategies.

In Section 2, we briefly motivate our work with some basic examples. In Section 3, we provide an overview of the structure-sensitive point-free style of functional programming and the associated algebraic laws. Section 4 presents the laws for reasoning about structure-shy programs and for mediation between these and structure-sensitive ones. In Section 5, we explain the Haskell encoding of rewrite systems that harnesses the algebraic laws. In Section 6, we revisit our motivating examples, and in Section 7 we discuss various application scenarios. Section 8 discusses related work, and Section 9 concludes.

## 2. Motivating examples

Structure-shy programming allows concise formulation of queries and transformations on rich data formats. Consider as an example the XML schema represented in Fig. 1 for documents that hold information about movies.

Let us consider some queries and transformations for this schema of varying degrees of structure-shyness.

### 2.1. XPath

Suppose one wants to retrieve all movie directors from a document. In the XPath query language, this query can be formulated as follows:

```
//movie/director
```

Specifically, it asks to retrieve *director* elements that are direct children of a *movie* element, where the *movie* element can appear at any depth in the document structure.

We assume some basic knowledge of XPath. Fig. 2 describes the core fragment of XPath syntax used throughout the paper. The full syntax is described in the XPath language reference [44]. Abbreviated syntax is available and heavily used: for instance, `//` expands to `/descendant-or-self::node()/` and an element name without preceding axis modifier expands to `/child::`*name*.

```
location := '/' ? (step ('/' step)*)
step     := axis '::' test pred *
axis     := 'child' | 'descendant' | 'self' | 'descendant-or-self'
test     := name | '*' | 'text()' | 'node()'
pred     := '[' expr ']'
name     := any document tag
```

**Fig. 2.** Core XPath syntax.

The above query is structure-shy in the sense that it does not explicitly specify the structural elements that occur between the document root and the *movie* element. This structure-shyness is desirable from the perspective of understandability, maintainability, and conciseness. But the execution time of the query may suffer from its structure-shyness, since it will look for *movie* elements throughout the document. Using knowledge of the schema, we would like to apply optimizing transformations to the query to obtain

```
imdb/movie/director
```

This query would not need to traverse into any children of the *imdb* element except those that are *movie* elements.

On the other hand, knowledge of the schema could be used to increase the structure-shyness of the query, transforming it into

```
//director
```

On documents conforming to the given schema, this query of increased structure-shyness would produce the same result as the original. But if, during the course of application evolution, the schema were to be changed such that directors no longer (only) appear as direct children of movies, then the original query would need to be adapted while the new query could remain untouched.

### 2.2. Strategic functional programming

Strategic programming was first supported in a non-typed setting in the Stratego language [38]. A strongly-typed combinator suite was introduced as a Haskell library by the Strafunski system [28,29]. This suite was generalized into the so-called 'Scrap-Your-Boilerplate' (SYB) approach to generic functional programming [25]. In this paper, we focus on a limited set of combinators that convey the essence of strategic programming [26]. Namely, for specifying type-preserving generic functions (transformations) we will use the following combinators:

$$
\begin{array}{lll}
\mathit{nop} & :: \mathsf{T} & \text{-- identity} \\
(\triangleright) & :: \mathsf{T} \to \mathsf{T} \to \mathsf{T} & \text{-- sequential composition} \\
\mathit{mapT} & :: \mathsf{T} \to \mathsf{T} & \text{-- map over children} \\
\mathit{mkT}_A & :: (A \to A) \to \mathsf{T} & \text{-- creation} \\
\mathit{apT}_A & :: \mathsf{T} \to (A \to A) & \text{-- application}
\end{array}
$$

The concrete definition of $\mathsf{T}$, the type of generic transformations, will be presented later in Section 5 (for readability we put single-letter type constants in sans serif font). The identity transformation is denoted by *nop*, $\triangleright$ sequences transformations, and *mapT* maps a transformation to all children of a node. Given a type-specific function $f : A \to A$, $\mathit{mkT}_A\ f$ is a generic transformation that applies $f$ to its argument if it is of type $A$, or otherwise returns it unchanged. Given a generic transformation, $\mathit{apT}_A$ applies it to a specific type.

For specifying type-unifying generic functions (queries) we have similar combinators:

$$
\begin{array}{lll}
\emptyset & :: \mathsf{Q}\ R & \text{-- empty result} \\
(\cup) & :: \mathsf{Q}\ R \to \mathsf{Q}\ R \to \mathsf{Q}\ R & \text{-- union of results} \\
\mathit{mapQ} & :: \mathsf{Q}\ R \to \mathsf{Q}\ R & \text{-- fold over children} \\
\mathit{mkQ}_A & :: (A \to R) \to \mathsf{Q}\ R & \text{-- creation} \\
\mathit{apQ}_A & :: \mathsf{Q}\ R \to (A \to R) & \text{-- application}
\end{array}
$$

The type of a generic query with result $R$ is denoted by $\mathsf{Q}\ R$. The result type $R$ is assumed to be a monoid, with a *zero* element and associative *plus* operator. These operations are used, for example, in *mapQ* in order to combine in a single value all the results of mapping a generic query to the children of a node (returning *zero* if there are no children).

Some highly useful derived combinators are

$$
\begin{array}{l}
\mathit{everywhere} :: \mathsf{T} \to \mathsf{T} \\
\mathit{everywhere}\ f = f \triangleright \mathit{mapT}\ (\mathit{everywhere}\ f) \\
\mathit{everything} :: \mathsf{Q}\ R \to \mathsf{Q}\ R \\
\mathit{everything}\ f = f \cup \mathit{mapQ}\ (\mathit{everything}\ f)
\end{array}
$$

```
data Imdb      = Imdb [Movie] [Actor]
data Movie     = Movie Year Title Director [Review] [BoxOffice]
data BoxOffice = BoxOffice Country Value
data Actor     = Actor Name [Played]
data Played    = Played Year Title Role [Award]
data Director  = Director String
data Year  = Year Int          data Review  = Review String
data Title  = Title String     data Country = Country String
data Value = Value Int         data Name    = Name String
data Role  = Role String       data Award   = Award String
```

**Fig. 3.** Haskell datatypes for the schema represented in Fig. 1.

Suppose one wants to truncate all reviews to 100 characters. Using strategic functional programming, and assuming that the IMDb XML schema is encoded in Haskell using the datatype presented in Fig. 3, this transformation can be expressed as follows:

```
trunc = everywhere (mkT_Review take100)
  where
    take100 (Review r) = Review (take 100 r)
```

The *everywhere* combinator applies its generic argument function in topdown fashion to every node in a term. The transformation $mkT_{Review}$ *take100* will only produce an effect in nodes of type *Review*, leaving all others unchanged.

To give an example of a generic query, consider the following function that counts the total size of all reviews stored in the IMDb:

```
count = everything (mkQ_Review size)
  where
    size (Review r) = length r
```

These structure-shy definitions suffer from performance problems just like the structure-shy XPath query above. They traverse into parts of the document where no *Review* occurs, and perform dynamic type tests, even though the data schema provides static information about where these tests would succeed.

For optimization, we would like to transform these strategic functions into definitions that do not employ strategic combinators:

```
trunc' = imdb (map (movie id id id (map take100) id)) id
count' = sum ∘ map (sum ∘ map size ∘ reviews) ∘ movies
```

Here we employ congruence and selector functions (also known as maps and projections, respectively) such as

```
imdb f g (Imdb m a) = Imdb (f m) (g a)
movies (Imdb m a) = m
```

The elimination of strategic functions in favour of ordinary functions enables subsequent optimizations by a regular compiler, and performance gains can be quite substantial.

To give a more precise idea of the possible gains, we have measured the space and time consumption for these examples. In order to quantify precisely the benefits obtained by type-specializing structure-shy strategic functions into ordinary definitions, we have not used the standard type-class based implementation of strategic combinators [25], but our own, based on explicit type parameterization and *Generalized Algebraic Datatypes* (GADTs) [21]. The SYB library is well known for its poor performance [32], most likely due to its heavy use of rank-2 polymorphism and run-time type-safe cast. For the running examples, our GADT version was roughly 14 times faster, and consumed 13 times less space.

Fig. 4 shows the time and space behaviour of the strategic and type-specific programs mapped against the size of the movie database—generated in memory with equal numbers of movies and actors. We analyze three program combinations: *count* ∘ *trunc*, *count* ∘ *trunc'*, and *count'* ∘ *trunc'*. We compiled each program using GHC 6.4.1 with optimization flag O1. In this case, type-specialization implies an improvement in space and time by factors of 2.6 and 4.8. Optimization of the *trunc* transformation alone implies an improvement by a factor of roughly 1.3 in both space and time. Additional type-specialization of the *count* query accounts for the remaining factors of 2.0 and 3.7.

As in the XPath example, increasing the structure-shyness of a function definition is also extremely useful. The introduction of strategic combinators into programs that do not employ them would allow us to synthesize structure-shy programs from structure-sensitive programs. Code that has been developed before the advent of strategic programming, or that has been initially conceived for particular data structures could be made more concise, understandable, and reusable.

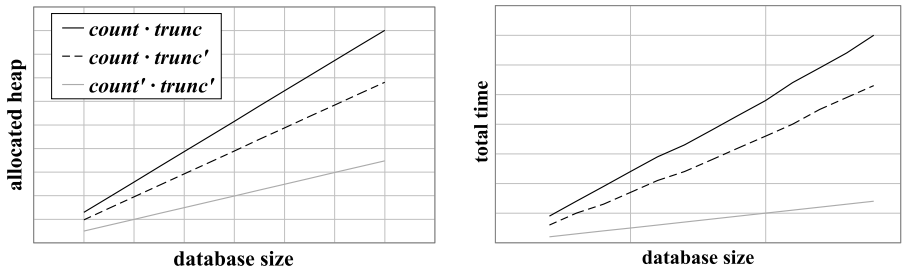



**Fig. 4.** Performance analysis.

## 3. Point-free functional programming

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 [5]. After Backus, others have adopted, complemented, and extended his work. This section briefly overviews this so-called point-free style of programming; for a more detailed presentation we recommend the textbook by Bird and de Moor [6] or the survey by Gibbons [16].

Point-free programs are constructed using a standard set of primitive functions and function combinators. This set of combinators was chosen based on the power of the associated algebraic laws. The set of laws that was used in the calculations performed for this paper is presented in Fig. 5. Although we present some examples in Haskell, we remark that the domain subject to our calculations is that of sets and total functions: in particular some of the presented laws are not valid over the counterpart Haskell types (additional strictness side-conditions would be necessary).

The most fundamental primitive and combinator are, respectively, the identity function and function composition.

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

For defining functions over products we have projections and the *split* combinator that combines results of two functions in a pair. The function product combinator updates a pair using different functions for each element.

$$fst :: A \times B \to A$$
$$snd :: A \times B \to B$$
$$(\vartriangle) :: (A \to B) \to (A \to C) \to (A \to B \times C)$$
$$(\times) :: (A \to B) \to (C \to D) \to (A \times C \to B \times D)$$

Inhabitants of a sum type can be constructed using one of the injection combinators. To process sums we have the *either* combinator, that performs case analysis on a value to decide which of its argument functions should be used. Mapping over sums can be done using the function sum combinator.

$$inl :: A \to A + B$$
$$inr :: B \to A + B$$
$$(\triangledown) :: (A \to C) \to (B \to C) \to (A + B \to C)$$
$$(+) :: (A \to B) \to (C \to D) \to (A + C \to B + D)$$

User-defined datatypes are manipulated by first exposing their isomorphic sum-of-products representation. To be more precise, since a datatype can be recursive, it can be defined as the fixpoint of a regular functor that captures the signature of its constructors. The functor of a datatype $A$ will be denoted as $\mathcal{F}_A$. When the type is clear from the context the subscript will be dropped. If the type is non-recursive the functor just ignores its argument. For example, type *Movie* of Fig. 3 is characterized by the following isomorphism:

$$Movie \cong \mu\ (\mathcal{F}_{Movie})$$
$$\textbf{where}\ \mathcal{F}_{Movie}\ x = Year \times Title \times Director \times [Review] \times [BoxOffice]$$

Here, $\mu$ is the fixpoint operator. For recursive datatypes the argument marks the occurrence of recursive elements. For example, suppose that the datatype *Name* was instead defined as a sequence of given names ended by a family name:

$$\textbf{data}\ Name = Family\ String \mid Given\ String\ Name$$

In this case we have the following isomorphism:

$$Name \cong \mu\ (\mathcal{F}_{Name})$$
$$\textbf{where}\ \mathcal{F}_{Name}\ x = String + String \times x$$

Associated with each datatype $A$ we also have two unique functions, $in_A :: \mathcal{F}\ A \to A$ and $out_A :: A \to \mathcal{F}\ A$, that are each other's inverse. They allow us to, respectively, construct and inspect values of the given type. To further clarify its meaning,

| | |
|---|---|
| $f \circ id = f \ \wedge \ id \circ f = f$ | $\circ$-ID |
| $f \circ (g \circ h) = (f \circ g) \circ h$ | $\circ$-ASSOC |
| $in_A \circ out_A = id$ | *datamap*-ID |
| $f \times g = (f \circ fst) \vartriangle (g \circ snd)$ | $\times$-DEF |
| $fst \circ (f \vartriangle g) = f \ \wedge \ snd \circ (f \vartriangle g) = g$ | $\times$-CANCEL |
| $(f \vartriangle g) \circ h = (f \circ h) \vartriangle (g \circ h)$ | $\times$-FUSION |
| $fst \vartriangle snd = id$ | $\times$-REFLEX |
| $f + g = (inl \circ f) \triangledown (inr \circ g)$ | $+$-DEF |
| $(f \triangledown g) \circ inl = f \ \wedge \ (f \triangledown g) \circ inr = g$ | $+$-CANCEL |
| $f \circ (g \triangledown h) = (f \circ g) \triangledown (f \circ h)$ | $+$-FUSION |
| $inl \triangledown inr = id$ | $+$-REFLEX |
| $map\ id = id$ | *map*-ID |
| $map\ f \circ nil = nil$ | *map*-NIL |
| $map\ f \circ map\ g = map\ (f \circ g)$ | *map*-FUSION |
| $map\ f \circ wrap = wrap \circ f$ | *map*-WRAP |
| $map\ f \circ nil = nil$ | *map*-NIL |
| $map\ f \circ cat = cat \circ (map\ f \times map\ f)$ | *map*-CAT |
| $map\ f \circ concat = concat \circ map\ (map\ f)$ | *map*-CONCAT |
| $concat \circ map\ wrap = id$ | *concat*-MAPWRAP |
| $filter\ true = id$ | *filter*-TRUE |
| $filter\ false = nil$ | *filter*-FALSE |
| $filter\ f \circ nil = nil$ | *filter*-NIL |
| $filter\ f \circ cat = cat \circ (filter\ f \times filter\ f)$ | *filter*-CAT |
| $filter\ f \circ map\ g = map\ g \circ filter\ (f \circ g)$ | *filter*-MAP |
| $filter\ f \circ concat = concat \circ map\ (filter\ f)$ | *filter*-CONCAT |
| $filter\ f \circ wrap = cond\ f\ wrap\ nil$ | *filter*-WRAP |
| $true \circ f = true$ | *true*-FUSION |
| $cond\ true\ f\ g = f$ | *cond*-TRUE |
| $cond\ false\ f\ g = g$ | *cond*-FALSE |
| $(cond\ f\ l\ r) \circ g = cond\ (f \circ g)\ (l \circ g)\ (r \circ g)$ | *cond*-FUSION |
| $plus \circ (zero \vartriangle f) = f \ \wedge \ plus \circ (f \vartriangle zero) = f$ | *plus*-ZERO |
| $zero \circ f = zero$ | *zero*-FUSION |
| $zero \triangledown zero = zero$ | *zero*-EITHER |
| $fold \circ nil = zero$ | *fold*-NIL |
| $fold \circ wrap = id$ | *fold*-WRAP |
| $fold \circ cat = plus \circ (fold \times fold)$ | *fold*-CAT |
| $fold \circ concat = fold \circ map\ fold$ | *fold*-CONCAT |
| $fold \circ map\ zero = zero$ | *fold*-MAPZERO |

**Fig. 5.** Some laws for point-free program calculation.

throughout the paper we will usually denote the inspector function $out_A$ as *unA*. For example, the following point-free function allows us to retrieve the title of a movie (notice that binary operators like $\times$ associate to the right):

$$getTitle :: Movie \to Title$$
$$getTitle = fst \circ snd \circ unMovie$$

We will also make extensive use of congruence functions such as

$$title :: (String \to String) \to Title \to Title$$
$$title\ f = in_{Title} \circ f \circ out_{Title}$$

In general, the congruence function for a given datatype has type $(\mathcal{F}\ A \to \mathcal{F}\ A) \to (A \to A)$ and will be denoted with the same name, but with the first letter in lowercase. For example, given the function $toUpper :: Char \to Char$, that converts a single character to uppercase, we can define a function to change the title of a movie to uppercase as follows:

$$bigTitle :: Movie \to Movie$$
$$bigTitle = movie\ (id \times title\ (map\ toUpper) \times id)$$

Although they could be added as normal user-defined datatypes, we will give a special treatment to lists: they occur frequently when encoding XML schemas as Haskell datatypes, and to make calculation easier it is convenient to encode typical list functions as primitives in our calculus. Namely, we will use the *map* combinator that applies a given function to all elements of a list, the *wrap* function that builds singleton lists, and *filter* that filters list elements with a predicate.

$$map :: (A \to B) \to ([A] \to [B])$$
$$wrap :: A \to [A]$$
$$filter :: (A \to Bool) \to ([A] \to [A])$$

In this paper we also use some overloaded functions for processing monoid types: *zero* returns the zero element of monoid *A*, *plus* sums two elements, and *fold* sums all elements in a list. For example, if the monoid is a list, *zero* returns the empty list, *plus* concatenates two lists, and *fold* flattens a list of lists.

$$zero :: B \to A$$
$$plus :: A \times A \to A$$
$$fold :: [A] \to A$$

Here we follow the standard practice in point-free programming of defining constants as functions that ignore their input parameter (allowing them to be combined with composition or any other point-free combinator). To make the presentation more clear, when defining functions and laws that are specific to a given monoid we will use specific functions instead of the overloaded ones. Namely, for the list monoid we will use $nil :: B \to [A]$, $cat :: [A] \times [A] \to [A]$ (an uncurried version of $(+\!\!+) :: [A] \to [A] \to [A]$), and $concat :: [[A]] \to [A]$, instead of *zero*, *plus*, and *fold*, respectively. For integers we will use $sum :: [Int] \to Int$ instead of *fold*. For example, in Fig. 5, we only use the overloaded functions when presenting laws that are valid for all monoids. Otherwise, we use the specific ones.

We also define (point-free) versions of if-then-else and constant *true*.

$$cond :: (A \to Bool) \to (A \to B) \to (A \to B) \to (A \to B)$$
$$true :: A \to Bool$$

Notice that we use a boolean monoid where *zero* is false and *plus* stands for disjunction. Again, we will use $false :: B \to Bool$ instead of *zero* when defining functions and laws specific to the boolean monoid.

In the next section we will show how the point-free style can be used as the solution space for transformation of structure-shy programs. Structure-shy programs will be converted into structure-sensitive point-free ones, so that we can use the laws presented in this chapter to perform optimizations by calculation. Due to the equational nature of the point-free calculus it is rather straightforward to implement a rewrite system to mechanize calculations, as will be described in Section 5.

In the past, a similar approach has been proposed to reason about pointwise programs: it is possible to define a systematic way of turning functions with variables and pattern matching into equivalent point-free forms, so that calculations can be performed straightforwardly in the point-free counterparts [13]. For example, the tools presented in the cited paper can convert the Haskell definition

$$assocr :: ((a, b), c) \to (a, (b, c))$$
$$assocr\ ((x, y), z) = (x, (y, z))$$

into the standard point-free definition of this combinator:

$$assocr = (fst \circ fst) \vartriangle (snd \times id)$$

This approach of reasoning about programs in the point-free style has been likened to Laplace or Fourier transforms, where one transforms a problem from one mathematical space into another, solves the problem in that space, and transforms the solution back to the original space [34]. In the second space, the solution can be found with a straightforward algorithm, while the original space resists such mechanized reasoning. Likewise, point-free programs can be used as the solution space to reason both about pointwise and structure-shy programs.

## 4. Algebraic laws of structure-shy programs

Just as for point-free functional programs, algebraic laws exist for structure-shy programs. Moreover, laws can be provided for mediation between structure-shy programs and structure-sensitive point-free programs. By virtue of these mediation laws, point-free program transformation can be used as the solution space for transformation of structure-shy programs, as we will demonstrate below.

### 4.1. Strategic programming laws

Fig. 6 provides an overview of equational laws that govern the strategic programming combinators. For example, the $\triangleright$-Id law states that the generic identity function is a left and right zero for generic sequential composition. The *mapT*-Fusion law states that generic maps distribute over generic sequential composition. The dual *mapQ*-Fusion law is only valid for commutative monoids. Some of these laws were formulated earlier [26,25], and can be easily proved by induction on the representation of types. Note that the reasoning power of these strategic programming laws is rather limited, precisely because they do not take type information into account. For example, there are no counterparts of the $\times$-Cancel rule, which enable the elimination of redundant computations.

Further laws can be formulated that mediate between structure-shy and structure-sensitive programs by type-specialization and generalization. Fig. 7 provides an overview of laws that mediate between strategic and point-free combinators. Essentially, these laws correspond to the operational semantics of generic combinators expressed in the

| | |
|---|---|
| $f \triangleright nop = f \wedge nop \triangleright f = f$ | $\triangleright$-ID |
| $f \triangleright (g \triangleright h) = (f \triangleright g) \triangleright h$ | $\triangleright$-ASSOC |
| $mapT\ nop = nop$ | $mapT$-NOP |
| $mapT\ f \triangleright mapT\ g = mapT\ (f \triangleright g)$ | $mapT$-FUSION |
| $f \cup \emptyset = f \wedge \emptyset \cup f = f$ | $\cup$-EMPTYL |
| $f \cup (g \cup h) = (f \cup g) \cup h$ | $\cup$-ASSOC |
| $mapQ\ \emptyset = \emptyset$ | $mapQ$-EMPTY |
| $mapQ\ f \cup mapQ\ g = mapQ\ (f \cup g)$ | $mapQ$-FUSION |

**Fig. 6.** Laws for strategic program combinators.

| | |
|---|---|
| $apT_A\ nop = id$ | $nop$-APPLY |
| $apT_A\ (f \triangleright g) = apT_A\ g \circ apT_A\ f$ | $\triangleright$-APPLY |
| $apT_A\ (mkT_A\ f) = f$<br>$apT_A\ (mkT_B\ f) = id$, **if** $A \not\equiv B$ | $mkT$-APPLY |
| $apT_A\ (everywhere\ f) = apT_A\ (f \triangleright mapT\ (everywhere\ f))$ | $everyw$-APPLY |
| $apT_A\ (mapT\ f) = id$, **if** $A$ base<br>$apT_{(A\times B)}\ (mapT\ f) = apT_A\ f \times apT_B\ f$<br>$apT_{(A+B)}\ (mapT\ f) = apT_A\ f + apT_B\ f$<br>$apT_{[A]}\ (mapT\ f) = map\ (apT_A\ f)$<br>$apT_A\ (mapT\ f) = in_A \circ apT'_{(\mathcal{F}\ A)}\ f \circ out_A$, **if** $A$ data | $mapT$-APPLY |
| $mkT_A\ id = nop$ | $id$-PULLT |
| $mkT_A\ (f \circ g) = mkT_A\ g \triangleright mkT_A\ f$ | $\circ$-PULLT |
| $apQ_A\ \emptyset = zero$ | $\emptyset$-APPLY |
| $apQ_A\ (f \cup g) = plus \circ (apQ_A\ f\ \vartriangle\ apQ_A\ g)$ | $\cup$-APPLY |
| $apQ_A\ (mkQ_A\ f) = f$<br>$apQ_A\ (mkQ_B\ f) = zero$, **if** $A \not\equiv B$ | $mkQ$-APPLY |
| $apQ_A\ (everything\ f) = apQ_A\ (f \cup mapQ\ (everything\ f))$ | $everyt$-APPLY |
| $apQ_A\ (mapQ\ f) = zero$, **if** $A$ base<br>$apQ_{(A\times B)}\ (mapQ\ f) = plus \circ ((apQ_A\ f) \times (apQ_B\ f))$<br>$apQ_{(A+B)}\ (mapQ\ f) = (apQ_A\ f) \triangledown (apQ_B\ f)$<br>$apQ_{[A]}\ (mapQ\ f) = fold \circ map\ (apQ_A\ f)$<br>$apQ_A\ (mapQ\ f) = apQ'_{(\mathcal{F}\ A)}\ f \circ out_A$, **if** $A$ data | $mapQ$-APPLY |
| $mkQ_A\ zero = \emptyset$ | $\emptyset$-PULLQ |
| $mkQ_A\ (plus \circ (f\ \vartriangle\ g)) = mkQ_A\ f \cup mkQ_A\ g$ | $plus$-PULLQ |

**Fig. 7.** Laws for mediating between strategic and point-free programs.

point-free style. However, we believe they were not formulated earlier, nor used for the purpose of type-specialization and generalization of structure-shy programs.

Specialization of structure-shy transformations proceeds by pushing down the *apT* combinator until it gets consumed by the *mkT*-APPLY law. The *mapT*-APPLY law states how a generic map should be specialized. When applied to a base type, the identity function is returned. For products, sums, and lists all the children are transformed using the respective map operations. When applied to a user-defined datatype, its sum-of-products representation is first exposed, and then the argument function is applied to all its content. Since we only have binary products and sums, the representation can be nested and thus it does not suffice to map over direct children: it is necessary to descend down the representation until base types or other user-defined datatypes are reached. This is accomplished by the $apT'$ function, whose concrete definition will be given when discussing the Haskell encoding of this law in Section 5.3. Similar laws are defined for the specialization of structure-shy queries.

Notice that *everyw*-APPLY uses the recursive definition of this combinator using *mapT* and $\triangleright$. As such, it cannot be used for specialization to recursive datatypes, since it would lead to an infinite expansion of the definition (due to successive expansions of *everywhere* in recursive occurrences of the type). A similar problem occurs with *everyt*-APPLY. The examples presented in this paper involve only non-recursive datatypes, and thus, such laws can be safely applied. Recently, we have shown how to extend this specialization mechanism to recursive datatypes [11]; see Section 8 for a brief presentation of that work.

To give an example of applying these laws to the specialization of a generic query, recall the definition of *count* presented in Section 2.2:

```
count = everything (mkQ_Review size)
  where
    size (Review r) = length r
```

$$
\begin{aligned}
& apQ_{Review}\ count \\
= & \{everyt\_apply\} \\
& apQ_{Review}\ (mkQ_{Review}\ size \cup mapQ\ count) \\
= & \{union\_apply\} \\
& plus \circ (apQ_{Review}\ (mkQ_{Review}\ size) \vartriangle apQ_{Review}\ (mapQ\ count)) \\
= & \{mkQ\_apply\} \\
& plus \circ (size \vartriangle apQ_{Review}\ (mapQ\ count)) \\
= & \{mapQ\_apply\} \\
& plus \circ (size \vartriangle apQ_{String}\ count \circ unReview) \\
= & \{everyt\_apply\} \\
& plus \circ (size \vartriangle apQ_{String}\ (mkQ_{Review}\ size \cup mapQ\ count) \circ unReview) \\
= & \{union\_apply, mkQ\_apply\} \\
& plus \circ (size \vartriangle plus \circ (zero \vartriangle apQ_{String}\ (mapQ\ count)) \circ unReview) \\
= & \{mapQ\_apply\} \\
& plus \circ (size \vartriangle plus \circ (zero \vartriangle zero) \circ unReview) \\
= & \{plus\_zero\} \\
& plus \circ (size \vartriangle zero \circ unReview) \\
= & \{zero\_fusion\} \\
& plus \circ (size \vartriangle zero) \\
= & \{plus\_zero\} \\
& size
\end{aligned}
$$

**Fig. 8.** Specialization of $apQ_{Review}\ count$.

Fig. 8 presents a derivation of the type-specific definition that results from applying *count* to a single review. This derivation uses the laws from Fig. 7 to derive a type-specific point-free definition. Notice how all different possibilities of applying the generic query are explored: most of them result in empty queries, which are later eliminated using the point-free calculation laws from Fig. 5. Section 5 presents a Haskell implementation of a rewrite system that can perform this specialization automatically. In fact, this particular calculation can be reproduced by such a system. Section 6 presents two more automatic derivation examples, namely a specialization of a generic transformation and a generalization of a type-specific query.

To increase a program's degree of structure-shyness we can use laws like ∘-PullT, that states how sequential composition can be pulled up through *mkT* to obtain generic sequential composition. However, to successfully accomplish this task some additional heuristic laws are needed, which will be presented in Section 5.4.

### 4.2. XML programming laws

Many XPath constructs can be expressed directly as strategic combinators of type $\mathsf{Q}\ [\star]$, where $\star$ represents a universal node type (a similar encoding was developed by Lämmel [24]):

```
self       :: Q [⋆]  -- self::node()
child      :: Q [⋆]  -- child::node()
desc       :: Q [⋆]  -- descendant::node()
descself   :: Q [⋆]  -- descendant-or-self::node()
name       :: String → Q [⋆]          -- self::name
(/)        :: Q [⋆] → Q R → Q R       -- /
(?)        :: Q [⋆] → Q Bool → Q [⋆]  -- q[p]
nonempty   :: Q Bool
```

The first four combinators model steps with various axes, each with test `node()`. The combinator *name* `"foo"` corresponds to the XPath step `self::foo`. When presenting queries we will just write ⟨*foo*⟩, which should not be confused with the XPath abbreviated syntax for `child::foo`.

Consider the abbreviated XPath query `//movie[//review]`, that uses a predicate to select all movies that have reviews. This query expands to

```
descendant-or-self::node()/child::movie[
          descendant-or-self::node()/child::review]
```

and is encoded using the above combinators as

*descself* / *child* / *name* `"movie"` ? *descself* / *child* / *name* `"review"` / *nonempty*

| | |
|---|---|
| $(f \cup g)\ /\ h = (f\ /\ h) \cup (g\ /\ h)$ | ∪-Dist |
| $\emptyset\ /\ f = \emptyset$ | /-Empty |
| $self\ /\ f = f \wedge f\ /\ self = f$ | /-Self |
| $name\ n\ /\ name\ n = name\ n$ | /-Name |
| $(f\ /\ g)\ /\ h = f\ /\ (g\ /\ h)$ | /-Assoc |
| $\emptyset\ ?\ p = \emptyset$ | ?-Empty |
| $f\ ?\ nonempty = f$ | ?-Nonempty |
| $(f\ ?\ p)\ ?\ q = (f\ ?\ q)\ ?\ p$ | ?-Comut |
| $f\ ?\ (name\ n\ /\ nonempty) = f\ /\ name\ n$ | ?-Name |

**Fig. 9.** Laws for XPath combinators.

| | |
|---|---|
| $child = mapQ\ self$ | *child*-Def |
| $desc = everything\ child$ | *desc*-Def |
| $descself = self \cup desc$ | *descself*-Def |
| $mapQ\ f = child\ /\ f$ | *mapQ*-Def |
| $apQ_A\ (f\ /\ g) = fold \circ map\ (apQ_\star\ g) \circ apQ_A\ f$ | /-Apply |
| $apQ_A\ (f\ ?\ p) = filter\ (apQ_\star\ p) \circ apQ_A\ f$ | ?-Apply |
| $apQ_A\ nonempty = true$ | *nempt*-Apply |
| $apQ_A\ self = wrap \circ mkAny_A$ | *self*-Apply |
| $apQ_A\ (name\ n) = apQ_A\ self$, **if** $A$ has name $n$<br>$apQ_A\ (name\ n) = zero$, **otherwise** | *name*-Apply |
| $apQ_\star\ f \circ mkAny_A = apQ_A\ f$ | ⋆-Apply |
| $mkQ_A\ (wrap \circ mkAny_A) = name\ n$, **if** $A$ has name $n$<br>$mkQ_A\ (wrap \circ mkAny_A) = self$, **otherwise** | ⋆-PullQ |
| $mkAny_\star = id$ | *mkAny*-Id |

**Fig. 10.** Laws for mediating between XPath and strategic/point-free programs.

This, in turn, we write using our abbreviated notation as

$descself\ /\ child\ /\ \langle movie \rangle\ ?\ descself\ /\ child\ /\ \langle review \rangle\ /\ nonempty$

As expressed by the ⋆ result type, the XPath combinators enjoy a very relaxed typing. The list of results returned by a query can contain nodes of any number of different types. As we will explain below, this poses additional challenges for transformation of XPath queries. The function $mkAny_A :: A \to \star$ is used to inject any type $A$ into the universal type. We assume this function is idempotent, i.e. $mkAny_\star = id$. The behaviour of combinators like *mapQ*, *mkQ*, and *apQ* on ⋆ is defined by their behaviour on the injected type.

Some algebraic laws for XPath combinators are presented in Fig. 9. For instance, the ∪-Dist and /-Assoc combinators state the distributivity and associativity properties of XPath combinators. Fig. 10 presents laws for the conversion and type-specialization of XPath expression. The *child*, *desc*, and *descself* axes are convertible to strategic combinators, as stated by various Def laws [24]. After converting them, the previously presented specialization laws for strategic queries can take effect. The remaining combinators can be converted directly to point-free expressions, using Apply rules. Note that a datatype $A$ has name $n$ if it encodes an XML element named $n$. The ⋆ laws allow elimination of the *mkAny* function. Below we will demonstrate how these rules together mediate between XPath and point-free expressions.

## 5. Encoding in Haskell

The various algebraic laws presented above can be harnessed into type-safe, type-directed rewriting systems for generalization, specialization, and optimization of structure-shy programs. In this section, we explain how the functional language Haskell can be used for this purpose.

### 5.1. Type-safe representation of types

Some of our algebraic laws, especially those of Figs. 7 and 10, make explicit reference to types. Some expose the structure of types (e.g. *mapT*-Apply). Others include type equality tests (e.g. *mkT*-Apply). To encode these laws, we will need type representations at the value level, which can be provided with the following GADT:

```
data Type a where
  Int    :: Type Int
  Bool   :: Type Bool
  String :: Type String
```

```
Any    :: Type ⋆
List   :: Type a → Type [a]
Prod   :: Type a → Type b → Type (a, b)
Either :: Type a → Type b → Type (Either a b)
Func   :: Type a → Type b → Type (a → b)
...
```

Note that the type $a$ that parameterizes the type representation *Type a* is instantiated differently in each constructor. This is precisely the difference between a GADT and a common parameterized datatype, where the parameters in the result type are unrestricted in all constructors. In the definition of *Type a*, the parameter $a$ of each constructor is restricted exactly to the type that the constructor represents, which makes our type representation type-safe. For example, the constructor *Int* represents the type *Int*, and *List* (*Prod Int Bool*) represents the type [(*Int*, *Bool*)]. This kind of type representation was first suggested independently by both Baars and Swierstra [4] and Cheney and Hinze [9] to extend a statically typed language to include some form of dynamic typing and generic programming.

The universal node type will be encoded in Haskell using dynamic values. A dynamic value can be encoded as a value paired with the representation of its type:

```
data ⋆ where Any :: Type a → a → ⋆
```

Function *mkAny* can be defined as follows:

```
mkAny :: Type a → a → ⋆
mkAny ⋆ x = x
mkAny a x = Any a x
```

It is possible to define a class with all representable types.

```
class Typeable a where typeof :: Type a
```

Most instances of this class are trivial to define. For example, for integers and functions, we have

```
instance Typeable Int where typeof = Int
instance (Typeable a, Typeable b) ⇒ Typeable (a → b) where
  typeof = Func typeof typeof
```

The definition of *Type a* presented above allows the representation of some basic types, products, sums, functions, and lists. To represent arbitrary user-defined datatypes, we extend it as follows (inspired by a trick previously introduced by Weirich [46]):

```
data Type a where
  ...
  Data :: String → EP a b → Type b → Type a
data EP a b = EP{to :: a → b, from :: b → a}
```

Here, *EP* is an embedding-projection pair that converts values of a user-defined datatype $a$ into its sum-of-products representation. The first parameter stores the name of the datatype. The type $b$ is expected to be equal to $\mathcal{F}\ a$.

Our movie database schema of Fig. 1 can be represented in Haskell by the user-defined datatypes shown in Fig. 3. Representations of these datatypes are constructed with *Data*. For example, the *Imdb* datatype is represented as follows:

```
instance Typeable Imdb where
  typeof = Data "Imdb" (EP to from) rep
    where
      rep = Prod (List typeof) (List typeof)
      to (Imdb ms as) = (ms, as)
      from (ms, as) = Imdb ms as
```

Here, *Typeable* instances are assumed for *Movie* and *Actor*.

Type equality can be defined by induction on type representations [4]:

```
teq :: Type a → Type b → Maybe (Equal a b)
teq Int Int = Just Eq
teq (List a) (List b) =
  case teq a b of Just Eq → Just Eq; _ → Nothing
...
teq _ _ = Nothing
data Equal a b where Eq :: Equal a a
```

The constructor *Eq* of the *Equal* GADT can be seen as a proof token of the equality of types $a$ and $b$.

### 5.2. Type-safe representation of functions

Apart from types, we need to represent functions in a type-safe manner. For this purpose we define a GADT with a constructor for each point-free program combinator:

```
data F f where
  Id      :: F (a→a)
  Comp    :: Type b → F (b→c) → F (a→b) → F (a→c)
  Fst     :: F ((a, b)→a)
  Snd     :: F ((a, b)→b)
  (△)     :: F (a→b) → F (a→c) → F (a→(b, c))
  (×)     :: F (a→b) → F (c→d) → F ((a, c)→(b, d))
  Plus    :: Monoid a → F ((a, a)→a)
  Datamap :: Type b → F (b→b) → F (a→a)
  UnData  :: F (a→b)
  MkAny   :: Type a → F (a→⋆)
  Fun     :: String → (a→b) → F (a→b)
  ...
```

Here we have elided many similar constructors. An inhabitant of type $\mathsf{F}\ (a \to b)$ is a representation of a function of type $a \to b$. The *Datamap* and *UnData* constructors represent congruence and inspector functions for user-defined data types. Again the type $b$ is expected to be the sum-of-products representation of datatype $a$. The *Fun* constructor allows us to include (pointwise) functions in point-free expressions without converting them to point-free shape; it can be used for functions over which no reasoning is performed. Constructors with an (implicitly) existentially quantified variable, such as *Comp* and *Datamap*, take a corresponding type representation as an additional argument. This allows one to reconstruct the type of the argument functions from the result function. Some functions, such as *Plus*, take as argument an explicit dictionary that provides the semantics of the respective monoid operations:

```
data Monoid r = Monoid{zero :: r, plus :: r → r → r}
```

Of course, this dictionary gives no guarantees that $r$ is indeed a monoid: any proof that the operations satisfy the required laws (associativity and identity) is left to the programmer. These operations are not relevant to calculations and are only used in the definition of an evaluation function for our typed abstract syntax.

To represent strategic functions, we must first define their types:

```
type T   = ∀a . Type a → a → a
type Q r = ∀a . Type a → a → r
```

Then we can add further constructors to $\mathsf{F}\ f$ to represent them:

```
data F f where
  ...
  Nop    :: F T
  Seq    :: F T → F T → F T
  ApT    :: Type a → F T → F (a→a)
  MkT    :: Type a → F (a→a) → F T
  MkQ    :: Monoid r → Type a → F (a→r) → F (Q r)
  Empty  :: Monoid r → F (Q r)
  ...
```

Similar constructors have again been elided. These constructors represent the combinators introduced in Section 2.2. Note that the query combinators take an additional argument *Monoid r* because the result type is expected to be a monoid.

Finally, the XPath combinators introduced in Section 4.2 are represented by constructors such as the following:

```
data F f where
  ...
  Self      :: F (Q [⋆])
  Name      :: String → F (Q [⋆])
  (:/:)     :: F (Q [⋆]) → F (Q r) → F (Q r)
  (:?:)     :: F (Q [⋆]) → F (Q Bool) → F (Q [⋆])
  Nonempty  :: F (Q Bool)
```

Lists are monoids; hence there is no need for *Monoid* arguments.

### 5.3. Rewrite rules

Now that type and function representations are in place, we proceed to the encoding of rewrite rules and systems. Individual rewrite rules, as well as the rewrite systems composed from them, are represented by monadic Haskell functions of the following type:

```
type Rule = ∀f . Type f → F f → RewriteM (F f)
```

Thus, a rule takes a function of type $f$ into a new function of the same type. The type representation passed as first argument allows rules to make type-based rewriting decisions; the importance of this will become clear below.

The *RewriteM* monad was designed both to allow partiality of rewrite rules, and to offer the capability of generating rewrite traces. To allow partiality, any instance of the *MonadPlus* class would suffice:

```
class Monad m ⇒ MonadPlus m where
  mzero :: m a
  mplus :: m a → m a → m a
```

Failure in the application of a rule should be signaled by invoking *mzero*. Among the possible instances of this class we will use the *Maybe* monad, where failure is denote by *Nothing* and the *mplus* is a left-biased choice.

The implementation of trace generation is quite more complex, because when rewriting a particular sub-expression we want to show the result of rewriting the whole expression that contains it. To allow this, the *Maybe* monad was extended using *RWST*, a standard monad transformer from the GHC library that adds reader, writer, and state functionalities to a given monad (*cf* [20]):

```
type RewriteM = RWST Location [(String, String)] ⋆ Maybe
```

The state monad allows us to thread a state through the computation. In this case, the state contains the whole point-free expression being rewritten. Since we are using a typed representation, the type of the state varies according to input expression: that is the reason why a dynamic value is used to represent the state. In order to know which sub-expression is currently being rewritten, a *Location* is propagated using a reader monad: a location is just a sequence of integers encoding the path from the root to the current sub-expression. Both the location and the state are updated in the rule combinators presented in Section 5.4.

A writer monad can be used to log information while computing values. In this case, the type of the log is [(*String*, *String*)]: a list of rewrite steps, each represented by a tuple with the name of the applied rule and the result of rewriting. Logging of a particular rule must be explicitly required by the user, by using the *success* function instead of *return*:

```
success :: String → (F f) → Rewrite (F f)
```

The first argument is supposed to be the rule's name.

Here is an encoding of the ∘-Id law from Fig. 5, applied in the left-to-right direction:

```
comp_id :: Rule
comp_id _ (Comp _ Id f) = return f
comp_id _ (Comp _ f Id) = return f
comp_id _ _             = mzero   -- catch all
```

This simple rule does not involve type information, so the first argument is ignored (indicated by _). Pattern matching is performed on a function representation and, on successful match, a resulting function representation is returned. Otherwise, failure of the rule is indicated by *mzero*. We omit this catch-all case in the rules below.

An example of a rule that involves type information is the left-to-right encoding of law ×-Def:

```
prod_def :: Rule
prod_def (Func (Prod a b) _) (f × g)
  = success "prod_def" ((Comp a f Fst) △ (Comp b g Snd))
```

Pattern matching on the type representation is performed to determine the intermediate types of compositions in the returned function. Notice the use of *success* to request the logging this rule in the rewrite traces.

Of course, laws can be applied in the right-to-left direction as well. For example, the inverse of the *prod_def* rule introduces rather than eliminates product maps:

```
prod_def_inv :: Rule
prod_def_inv _ ((Comp _ f Fst) △ (Comp _ g Snd)) =
  success "prod_def_inv" (f × g)
prod_def_inv _ ((Comp _ f Fst) △ Snd) =
  success "prod_def_inv" (f × Id)
prod_def_inv _ (Fst △ (Comp _ g Snd)) =
  success "prod_def_inv" (Id × g)
```

In this case, it does not suffice to encode the $\times$-Def law as presented in Fig. 5. Suppose we have the point-free expression $fst \circ fst \vartriangle snd$. In order to rewrite this expression into the equivalent $fst \times id$, we first need to apply $\circ$-Id (in the right-to-left direction) to $snd$ in order to obtain $fst \circ fst \vartriangle id \circ snd$. Only then could the $\times$-Def be directly applicable. If we included in a rewrite system $\circ$-Id encoded as a right-to-left rewrite rule, complex measures would be needed in order to avoid infinite expansions (including some backtracking technique). To avoid such complexities, some rules need some extra equations to handle cases where $\circ$-Id needs to be first applied in the right-to-left direction. That is the case of the last two equations of *prod_def_inv*.

Type-equality tests play a role in rules such as *mkT*-Apply:

```
mkT_apply :: Rule
mkT_apply _ (ApT a (MkT b f))
   = case teq a b of Just Eq  → success "mkT_apply" f
                     Nothing → success "mkT_apply" Id
```

Thus, if the type of the *ApT* and the type of the *MkT* are equal, the function $f$ is returned. Otherwise, the identity function *Id* is returned. The law *mapT*-Apply is encoded as follows:

```
mapT_apply _ (ApT Int (MapT f)) =
  success "mapT_apply" Id
...
mapT_apply _ (ApT (List a) (MapT f)) =
  success "mapT_apply" (Listmap (ApT a f))
mapT_apply _ (ApT (Data _ _ b) (MapT f)) =
  success "mapT_apply" (Datamap b (apT' b))
    where apT' (Prod a b)   = (apT' a) × (apT' b)
          apT' (Either a b) = (apT' a) + (apT' b)
          apT' a            = ApT a f
```

Notice how the $apT'$ auxiliary function applies the transformation $f$ to all the content of a user-defined datatype: representations consist of nested sums of products that must be traversed until base types or other user-defined datatypes are reached.

### 5.4. Combining rules into transformation systems

Rewrite rules are possibly partial, type-preserving transformations on function representations. Thus, to combine rewrite rules into rewrite systems, we define a new suite of strategic function combinators, similar to those presented in Section 2.2:

```
nop :: Rule                    -- identity rule
(⊰) :: Rule→Rule→Rule          -- sequential composition
(⊘) :: Rule→Rule→Rule          -- choice
all :: Rule → Rule             -- map on all children
one :: Rule → Rule             -- map on one child
rewrite :: Rule → F f → F f    -- top-level application
```

Rewriting rules can also be seen as generic transformations on values of type $\mathsf{F}\ f$ (type *Rule* is basically a restriction of type $\mathsf{T}$ to point-free expressions). We cannot reuse exactly the same strategic combinators of Section 2.2 to define rewrite systems because the result of applying rules is inside the *RewriteM* monad. Since this monad supports partiality we also have some extra combinators: the choice combinator $\oslash$ attempts to apply the left rule, and, if it fails, resorts to the right one; and *one* applies a rule to just one child. The top-level application function *rewrite* takes the result of rewriting a term out of the *RewriteM* monad; in the case of failure it returns the original function representation.

Using the primitive combinators presented above, we can define a useful set of derived combinators:

```
many :: Rule → Rule
many r = (r ⊰ (many r)) ⊘ nop

once :: Rule → Rule
once r = r ⊘ one (once r)

innermost :: Rule → Rule
innermost r = all (innermost r) ⊰ ((r ⊰ innermost r) ⊘ nop)
```

The combinator *many* applies a rule repeatedly until it fails; *once* attempts to apply a rule somewhere inside a point-free expression; and *innermost* performs exhaustive rewrite rule application.

*Optimization of point-free programs.* Using these strategic rewrite rule combinators, we can compose our one-step rewrite rules into complete transformation strategies. For example:

```
optimize_pf = innermost opt ⊰ innermost inv
  where
```

| | |
|---|---|
| $mapT\ (everywhere\ f) \stackrel{?}{=} everywhere\ f$ | $mapT$-ELIM |
| $mkT_A\ f \stackrel{?}{=} everywhere\ (mkT_A\ f)$ | $everyw$-INTRO |
| $mkT_{[A]}\ (map\ f) \stackrel{?}{=} mapT\ (mkT_A\ f)$ | $map$-PULLT |
| $mkT_{(A\times A)}\ (f\times f) \stackrel{?}{=} mapT\ (mkT_A\ f)$<br>$mkT_{(A\times B)}\ (f\times g) \stackrel{?}{=} mapT\ (mkT_A\ f \triangleright mkT_B\ g)$, **if** $A \not\equiv B$ | $\times$-PULLT |
| $mapQ\ (everything\ f) \stackrel{?}{=} everything\ f$ | $mapQ$-ELIM |
| $mkQ_A\ f \stackrel{?}{=} everything\ (mkQ_A\ f)$ | $everyt$-INTRO |
| $mkQ_{[A]}\ (fold \circ map\ f) \stackrel{?}{=} mapQ\ (mkQ_A\ f)$<br>$mkQ_{[A]}\ (map\ f) \stackrel{?}{=} mapQ\ (mkQ_A\ (wrap \circ f))$ | $map$-PULLQ |
| $mkQ_{(A\times B)}\ (f \circ fst) \stackrel{?}{=} mapQ\ (mkQ_A\ f)$, **if** $A \not\equiv B$<br>$mkQ_{(A\times B)}\ (f \circ snd) \stackrel{?}{=} mapQ\ (mkQ_B\ f)$, **if** $A \not\equiv B$ | $\times$-PULLQ |
| $self \stackrel{?}{=} descself$ | $self$-ELIM |
| $child\ /\ descself \stackrel{?}{=} descself$ | $child$-ELIM |

**Fig. 11.** Heuristic laws for strategic and XPath combinators.

$$opt = comp\_id \oslash prod\_def \oslash prod\_cancel \oslash \ldots$$
$$inv = prod\_def\_inv \oslash prod\_fusion\_inv \oslash \ldots$$

The *optimize_pf* strategy first performs optimization of point-free functions by exhaustive application of the laws in Fig. 5, oriented as rewrite rules from left to right. After that, some inverse rules are applied to make the resulting function more concise. The main objective of this second phase is not optimization, but to increase readability of the resulting expressions: most of its rules are right-to-left instances of non-primitive combinator definitions (like $\times$-DEF) and fusion laws (like $\times$-FUSION). For example, the rules in the second phase allow the following derivation:

$$\begin{aligned}
&fst \circ snd \vartriangle fst \circ snd \circ snd \\
&= \{prod_fusion_inv\} \\
&(fst \vartriangle fst \circ snd) \circ snd \\
&= \{prod_def_inv\} \\
&(id \times fst) \circ snd
\end{aligned}$$

Notice how the fusion law factored out a common sub-expression and enabled the application of the product combinator definition.

*Specialization of structure-shy programs*. The specialization of type-preserving strategic programs into point-free form is achieved by systematic application of the APPLY rules of Fig. 7, followed by the point-free optimization strategy:

$$optimize\_t = t2pf \circledcirc optimize\_pf$$
$$t2pf = innermost\ (mapT\_apply \oslash mkT\_apply \oslash \ldots)$$

For generic queries we have similar optimization strategies.

*Increasing structure-shyness*. To increase structure-shyness, we complement the laws presented in Section 4 with additional rules that are *not* valid in general, but are rather *heuristic*. Fig. 11 provides a list. To prevent application of these heuristic laws when they are not valid, they must be *guarded*: the result of applying a heuristic rule must preserve the semantics of the original expression. For type-preserving functions, the application of a heuristic rule can be guarded with the following combinator:

$$\begin{aligned}
&guardT :: Rule \to Rule \\
&guardT\ r\ t\ f = \mathbf{do} \\
&\quad g \leftarrow r\ t\ f \\
&\quad f' \leftarrow optimize_t\ t\ f \\
&\quad g' \leftarrow optimize_t\ t\ g \\
&\quad \mathbf{if}\ (f' \equiv g')\ \mathbf{then}\ return\ g\ \mathbf{else}\ mzero
\end{aligned}$$

Our approximation to checking semantic equivalence consists in comparing (using the syntactic equality operator $\equiv$) the results of specializing to type $t$ (using *optimize_t*) both the argument expression $f$ and the result $g$ of applying the heuristic rule $r$ to $f$. For queries, we have a similar function, *guardQ*.

We have devised a three-phase strategy for increasing the structure-shyness of generic programs. First, we specialize the program to an optimized point-free form, using the strategies presented above. The resulting program will not contain redundant transformations or redundant queries. Secondly, we exhaustively apply PULL laws of Figs. 7, 10 and 11, which result in a program where as many point-free combinators as possible have been replaced by structure-shy counterparts.

In the last phase, we further increase the structure-shyness by application of rules for structure-shy combinators only, presented in Figs. 6, 9 and 10, combined with the ELIM and INTRO laws of Fig. 11. Thus, for type-preserving generic functions, we have

$$\begin{aligned} generalize\_t = {} & optimize\_t \gg mkT\_apply\_inv \gg \\ & many\ (once\ (id\_pullT \oslash comp\_pullT \oslash \ldots) \\ & \qquad \oslash guardT\ (once\ (map\_pullT \oslash \ldots))) \gg \\ & many\ (once\ (seq\_id \oslash mapT\_fusion \oslash \ldots) \oslash \ldots) \end{aligned}$$

The *mkT_apply_inv* rule inserts the combinators $apT_A \circ mkT_A$ at the top level to seed the pull process of the second phase. Notice the use of *guardT* to protect the application of the heuristic rules. The strategies for strategic and XPath queries are similar.

## 6. Revisiting the motivating examples

Now that we have encoded several rewrite systems for structure-shy program transformation, we return to our examples of Section 2. We demonstrate several scenarios, such as generalization, specialization, and optimization of transformations and queries. All examples were run in the GHCi 6.4.1 Haskell interpreter (> denotes the interpreter prompt).

### 6.1. Transformations

Recall the example transformation for truncating reviews:

$$> \textbf{let}\ trunc = everywhere\ (mkT_{Review}\ take100)$$

We can apply our *optimize_t* strategy to specialize this structure-shy transformation to a structure-sensitive one, for a specific type. Let us try this first for the type *Imdb*:

$$\begin{aligned} & > rewrite\ optimize\_t\ (apT_{Imdb}\ trunc) \\ & imdb\ (map\ (movie\ (id \times id \times id \times map\ take100 \times id)) \times id) \end{aligned}$$

Note the use of *apT* to select the type for which we want to specialize. So, indeed, our strategy is able to perform the type-specialization that we expected; the difference with respect to the result presented in Section 2.2 is due to the fact that these datatypes are now internally represented as (nested) products. We get different results when we perform specialization for different types:

$$\begin{aligned} & > rewrite\ optimize\_t\ (apT_{Actor}\ trunc) \\ & id \\ & > rewrite\ optimize\_t\ (apT_{Review}\ trunc) \\ & take100 \end{aligned}$$

Thus, when specialized for the type *Actor*, inside which no reviews can occur, the transformation reduces to the identity function. When specialized for the type *Review*, the transformation reduces to the truncation function itself. The rewrite trace of this last derivation is presented in Fig. 12.

Rather than eliminating the structure-shyness of a transformation by type-specialization, we can attempt to increase structure-shyness with our strategy *generalize_t*. Consider the following function that converts to uppercase all the awards of an actor.

$$\begin{aligned} & > \textbf{let}\ up = apT_{Actor}\ (everywhere\ (mkT_{Award}\ upper)) \\ & \quad \textbf{where}\ upper\ (Award\ t) = Award\ (map\ toUpper\ t) \end{aligned}$$

A programmer who is not fully aware of the schema could try to convert all of the awards in a movie database by applying the *up* query restricted to *Actor* elements.

$$> \textbf{let}\ bigawards = everywhere\ (mkT_{Actor}\ up)$$

However, generalization of this query for *Imdb* yields the following result:

$$\begin{aligned} & > rewrite\ generalize\_t\ (apT_{Imdb}\ bigawards) \\ & apT_{Imdb}\ (everywhere\ (mkT_{Award}\ upper)) \end{aligned}$$

In fact, the check for *Actor* is not needed, because in the *Imdb* schema the *Award* element only occurs under *Actor*.

### 6.2. Queries

The following query computes the total length of reviews:

$$> \textbf{let}\ count = everything\ (mkQ_{Review}\ size)$$

$$
\begin{aligned}
& apT_{Review}\ trunc \\
= & \{everyw\_apply\} \\
& apT_{Review}\ (mkT_{Review}\ take100 \triangleright mapT\ trunc) \\
= & \{seq\_apply\} \\
& apT_{Review}\ (mapT\ trunc) \circ apT_{Review}\ (mkT_{Review}\ take100) \\
= & \{mkT\_apply\} \\
& apT_{Review}\ (mapT\ trunc) \circ take100 \\
= & \{mapT\_apply\} \\
& review\ (apT_{String}\ trunc) \circ take100 \\
= & \{everyw\_apply\} \\
& review\ (apT_{String}\ (mkT_{Review}\ take100 \triangleright mapT\ trunc)) \circ take100 \\
= & \{seq\_apply\} \\
& review\ (apT_{String}\ (mapT\ trunc) \circ apT_{String}\ (mkT_{Review}\ take100)) \circ take100 \\
= & \{mkT\_apply\} \\
& review\ (apT_{String}\ (mapT\ trunc) \circ id) \circ take100 \\
= & \{mapT\_apply\} \\
& review\ id \circ take100 \\
= & \{datamap\_id\} \\
& take100
\end{aligned}
$$

**Fig. 12.** Optimization of $apT_{Review}\ trunc$.

Consider the type-specializations obtained when applied to types *Imdb* and *Actor*:

$$
\begin{aligned}
& > rewrite\ optimize\_q\ (apQ_{Imdb}\ count) \\
& sum \circ map\ (sum \circ map\ size \circ reviews) \circ movies \\
& \quad \textbf{where}\ movies\ = fst \circ unImdb \\
& \qquad\quad reviews = fst \circ snd \circ snd \circ snd \circ unMovie \\
& > rewrite\ optimize\_q\ (apQ_{Actor}\ count) \\
& zero
\end{aligned}
$$

Again we get a similar result to the one in Section 2.2; in this case the difference is that the selector functions are expressed as compositions of *fst* and *snd* due to internal representation of these datatypes as nested products. As expected, the application of *count* to a branch of the schema where reviews do not occur specializes to the constant *zero* function, which always returns 0. If we apply *generalize_q* to the above result of specializing *count*, we obtain the original function *count* again.

In order to increase readability we use a special show function that prints specific monoid functions instead of the overloaded ones (it can do so because it receives a representation of the type of the expression to be printed). The factorization of the selector functions into **where** clauses was manually introduced, but we also intend to automate such functionality in the future.

### 6.3. XPath

Recall the XPath queries presented in Section 2:

```
imdb/movie/director
//movie/director
//director
```

They all represent the same query, expressed at increasing levels of structure-shyness. The specialization of the last query for the [*Imdb*] type produces the following result (we specialize to a list to allow for XML documents with several top-level elements):

$$
\begin{aligned}
& > \textbf{let}\ directors = descself\ /\ child\ /\ \langle director \rangle \\
& > rewrite\ optimize\_xp\ (apQ_{[Imdb]}\ directors) \\
& concat \circ map\ (map\ (mkAny_{Director} \circ director) \circ movies) \\
& \quad \textbf{where} \\
& \qquad movies\ = fst \circ unImdb \\
& \qquad director = fst \circ snd \circ snd \circ unMovie
\end{aligned}
$$

The retrieved director elements are wrapped into the *mkAny* constructor, since the return type of the overall query is still [⋆]. The same result is obtained when we specialize the remaining queries.

The rules involved in the specialization of the query `//director` are shown in tables in Fig. 13, together with how often they are applied. The table excludes applications of the trivial rules ∘-Id and ∘-Assoc. Of the 1177 non-trivial rules that are applied we can see that the initial specialization of the query to a non-optimized point-free expression accounts

| | |
|---|---|
| *mapT*-Apply | 84 |
| ⋆-Apply | 55 |
| *self*-Apply | 55 |
| ∪-Apply | 29 |
| *everyt*-Def | 28 |
| *name*-Apply | 27 |
| /-Apply | 4 |
| *child*-Def | 2 |
| ∪-Dist | 2 |
| *desc*-Def | 1 |
| *descself*-Def | 1 |
| Specialization | 288 |

| | |
|---|---|
| ×-Cancel | 204 |
| ×-Fusion | 159 |
| ×-Def | 132 |
| *plus*-Zero | 59 |
| *zero*-Fusion | 58 |
| *fold*-Wrap | 56 |
| *fold*-Cat | 51 |
| *map*-Cat | 51 |
| *map*-Wrap | 37 |
| *map*-Fusion | 33 |
| *fold*-MapZero | 19 |
| *fold*-Concat | 15 |
| *map*-Concat | 15 |
| Optimization | 889 |

**Fig. 13.** Rules involved in the specialization of the `//director` query.

$$
\begin{aligned}
& apQ_{[Imdb]}\ (mkQ_{[Imdb]}\ (concat \circ map\ (map\ (mkAny_{\_} \circ director) \circ movies))) \\
= & \{map\_pullq\} \\
& apQ_{[Imdb]}\ (mapQ\ (mkQ_{Imdb}\ (map\ (mkAny_{\_} \circ director) \circ movies))) \\
= & \{data\_pullq\} \\
& apQ_{[Imdb]}\ (mapQ\ (mapQ\ (mkQ_{Movie}\ (wrap \circ mkAny_{\_} \circ director)))) \\
= & \{data\_pullq\} \\
& apQ_{[Imdb]}\ (mapQ\ (mapQ\ (mapQ\ (mkQ_{Director}\ (wrap \circ mkAny_{\_}))))) \\
= & \{any\_pullq\} \\
& apQ_{[Imdb]}\ (mapQ\ (mapQ\ (mapQ\ \langle director \rangle))) \\
= & \{mapQ\_def\} \\
& apQ_{[Imdb]}\ (mapQ\ (mapQ\ (child\ /\ \langle director \rangle))) \\
= & \{self\_elim, child\_elim\} \\
& apQ_{[Imdb]}\ (mapQ\ (mapQ\ (descself\ /\ \langle director \rangle))) \\
= & \{mapQ\_def\} \\
& apQ_{[Imdb]}\ (mapQ\ (child\ /\ (descself\ /\ \langle director \rangle))) \\
= & \{child\_elim\} \\
& apQ_{[Imdb]}\ (mapQ\ (descself\ /\ \langle director \rangle)) \\
= & \{mapQ\_def\} \\
& apQ_{[Imdb]}\ (child\ /\ (descself\ /\ \langle director \rangle)) \\
= & \{child\_elim\} \\
& apQ_{[Imdb]}\ (descself\ /\ \langle director \rangle)
\end{aligned}
$$

**Fig. 14.** Generalization of `//directors`.

for about a quarter of the steps. The subsequent optimization of this huge intermediate point-free expression into the final, concise point-free expression accounts for the bulk of the work (three quarters). In general, specialization of highly structure-shy queries involves a high number of rewrite steps, because larger portions of the schema must be searched to find opportunities for optimization.

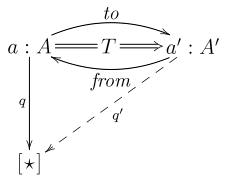
By application of the *generalize_xp* strategy to the optimized query, we can maximize its structure-shyness. As can be seen in the derivation of Fig. 14, the result of this reconstruction is the most structure-shy of the three original XPath queries, i.e. `//director`. Recall that the application of a heuristic rule involves a call to the optimization strategy. The derivation trace presented in the figure only shows one step for each heuristic rule: as such it does not give a clear measure of the effort involved in the generalization.

More challenging is the specialization of queries with predicates, such as retrieving all movies with an actor descendant (`//movie[//actor]`):

$$
\begin{aligned}
& > \textbf{let}\ movactors = descself\ /\ \langle movie \rangle\ ?\ descself\ /\ \langle actor \rangle\ /\ nonempty \\
& > rewrite\ optimize\_xp\ (apQ_{[Imdb]}\ empty) \\
& nil
\end{aligned}
$$

Or, retrieving all elements with a director child (`//*[/director]`):

$$
\begin{aligned}
& > \textbf{let}\ dirparents = descself\ ?\ child\ /\ \langle director \rangle\ /\ nonempty \\
& > rewrite\ optimize\_xp\ (apQ_{[Imdb]}\ dirparents) \\
& concat \circ map\ (map\ mkAny_{Movie} \circ fst \circ unImdb)
\end{aligned}
$$




| | |
|---|---|
| $A$, $A'$ | datatype and transformed datatype |
| $a$,$a'$ | instances of $A$ and $A'$, resp. |
| $q$, $q'$ | query and transformed query |
| $T$ | transformation of type $A$ into type $A'$ |
| *to* | migration function of type $A \to A'$ |
| *from* | partial migration function of type $A' \to A$ |

**Fig. 15.** Coupled transformation of datatypes, data instances, and queries.

Because movies cannot have actors inside, the first query specializes to the constant function *nil*, which always yields the empty list. Its generalization, with *generalize_xp*, yields *empty*. The second query specializes to a point-free function that retrieves movies, as these are the only possible parents of directors. Indeed, generalization of this query with *generalize_xp* produces `//movie`.

## 7. Applications

The algebraic laws and ensuing rewrite systems presented above have found applications in optimizing compilation of XPath queries, and in query and constraint migration in the context of coupled transformation of schemas, documents, queries, and constraints. We will briefly discuss these applications.

### 7.1. XPTO: A schema-aware XPath compiler

Under the supervision of the authors, Ferreira and Pacheco incorporated the rewrite system presented here into a schema-aware XPath compiler called XPTO (*XPaTh Optimizer*) [15]. XPTO is schema aware not only in the traditional sense that it validates XML files against a given schema. It also uses the schema information to optimize the XPath queries using the techniques described in this paper.

XPTO receives as input an XML schema and an XPath query. As output it produces an executable file that can be used to run the query against multiple XML documents conforming to the schema. XPTO can also be used to directly interpret the query over an XML document, but in this case the performance penalty incurred in the optimization process cannot be amortized, and most conventional interpreters can easily achieve better performance.

The compilation process is staged in two phases:

1. The schema and the query are parsed into the respective type-safe GADT representations. The query is then optimized using the strategy presented in Section 5.4. The resulting point-free expression is written to an intermediate Haskell file, together with datatype declarations to represent all XML elements (similar to those presented in Fig. 3). The main function of this file parses an XML document using the HaXml library [45], converts it to the respective datatype, applies the optimized query and pretty-prints the results.
2. The intermediate Haskell file generated in the first phase is then compiled using GHC in order to obtain the desired executable.

Although still far away from the complete XPath 2.0 specification, XPTO already supports more language features then those allowed by the combinators introduced in Section 4.2. In particular, a predicate that indexes by number inside a result set is accepted. For example, the compiler supports the query `//movie/title[1]`, which selects the first movie title in the database. Of course, to optimize this kind of query, new algebraic laws were added to the system. For the particular case of indexes we have, for example, the following laws:

$$\begin{aligned} \mathit{index}\ n \circ \mathit{zero} &= \mathit{zero} && \mathit{idx}\text{-Zero} \\ \mathit{index}\ 1 \circ \mathit{wrap} &= \mathit{wrap} && \mathit{idx}\text{-Wrap} \\ \mathit{index}\ n \circ \mathit{map}\ f &= f \circ \mathit{index}\ n && \mathit{idx}\text{-Map} \end{aligned}$$

The *idx*-Map law used as a rewrite rule from left to right is particularly useful for optimization: to map a function $f$ over all elements in a set of results followed by selection of a single result is obviously more costly than first selecting the element and subsequently applying $f$ only once.

### 7.2. Two-level transformations

Coupled transformations [23] occur in software evolution when multiple artifacts must be modified in such a way that they remain consistent with each other. A particularly challenging instance of coupled transformation involves the joint transformation of a data type, its instances, and the programs that consume or produce it. This problem occurs for example when the schema of a set of documents needs to be adapted. The adaptation of the document schema must then be coupled with migration of the documents and with updates of the queries that operate on these documents.

Fig. 15 provides a schematic summary of the problem of coupled transformation. A type-level transformation $T$ of a source type $A$ into a target type $A'$ is witnessed by associated instance migration functions *to* and *from*. Given a query $q$ that consumes values of the original type $A$, we obtain a query on the transformed type $A'$ by simply composing $q$ with the

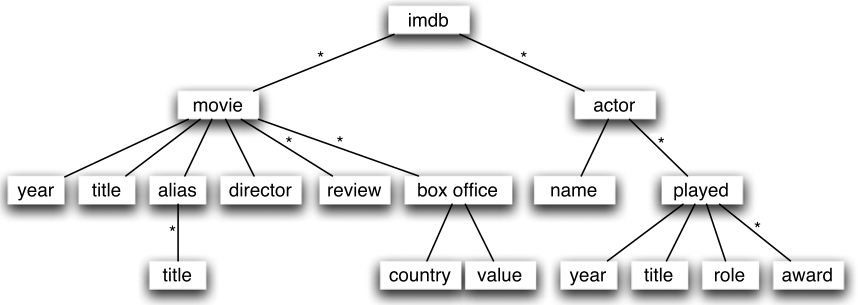



**Fig. 16.** Representation of an evolved version of the schema of Fig. 1.

migration function *from*. This amounts to a *wrapper* approach to query migration where the original type and the original query are still explicitly present. The challenge of query migration is to calculate processor $q'$ from the wrapper composition in such a way that it no longer involves the original type and query.

Previously, we have shown that coupled transformation of schemas and instances can be formalized by refinement theory and can be harnessed in strategic rewrite systems for two-level transformations [10]. By refinement, we mean a lossless transformation where the representation function *to* is injective and the (possible) partial abstraction function *from* is surjective: they must satisfy the law $from \circ to = id$. Furthermore, we have shown that such two-level strategic rewrite systems can be combined with type-preserving rewrite systems for point-free program transformation to support the coupled transformation of schemas, instances, processors (queries, as well as producers) and constraints [14,3,42].

In the current paper, we have generalized those rewrite systems for point-free program transformations to structure-shy programs. As an immediate consequence, our approach to coupled transformation now also encompasses migration and mapping of structure-shy queries and constraints. In particular, we can use the rewrite systems for structure-shy programs to:

1. determine whether query $q$ on the original schema $A$ can be re-used as is on the transformed schema $A'$ without change in meaning;
2. migrate a structure-shy query $q$ on schema $A$ to a new structure-shy query $q'$ on an evolved schema $A'$

We will explain these two scenarios by example.

Consider the *Imdb* schema of our running example (Fig. 1). Suppose that we want to evolve that schema so that, as well as the original title, alternative titles can be listed. Fig. 16 represents the evolved schema, where the *movie* element has an additional *alias* child with a list of alternative titles. This evolution step can be captured by a two-level rewrite step, which is composed of (i) the type transformation from *Imdb* to *Imdb*′, and (ii) the pair of functions $to : Imdb \to Imdb'$ and $from : Imdb' \to Imdb$ which allow migration of documents between the two schemas.

To determine whether a query $q$ can be re-used as is, without change of meaning, we need to check two equivalences, depending on what notion of meaning preservation we want to enforce:

$$apQ_{[Imdb]}\ q \equiv apQ_{[Imdb']} \circ to$$
$$(apQ_{[Imdb]}\ q) \bullet from \equiv apQ_{[Imdb']}$$

Here, $\cdot \bullet \cdot$ is a variation of function composition that takes a partial function as second argument. The first states that applying the query to an old-style document should be equivalent to applying the query after the document has been migrated (with *to*) to the new schema. The second equivalence states that applying the query to a new-style document should be equivalent to applying the query after the document has been migrated backwards, i.e. stripped of any alternative titles it may store. The second equivalence implies the first, because we have $from \circ to = id$ for any well-formed refinement. Thus, the first equivalence incorporates a weaker notion of semantics preservation than the second. The difference lies in what is demanded for information that can be stored in the evolved schema, but not in the original schema. The weaker notion makes no demands regarding alternative titles, while the stronger notion demands that the alternative titles, if present, do not influence the query result.

To see how these two equivalences play out, let us consider the following query, which we assume has been defined on the original schema:

```
//movie/title
```

This query retrieves movie titles from the database. When we instantiate both sides of the first equation with this query, and then apply our specialization strategy, both sides normalize to the same point-free expression:

$$\begin{aligned} &concat \circ map\ (map\ (mkAny_{Title} \circ title) \circ movies) \\ &\quad \textbf{where}\ movies = fst \circ unImdb \\ &\qquad\quad title\ \ = fst \circ snd \circ unMovie \end{aligned}$$

This normal form is reached also when we instantiate and specialize both sides of the second equation. This means that the meaning of the query is preserved, both in the weaker sense and in the stronger sense.

For a very similar query, different results emerge. Consider

```
//movie//title
```

This query is slightly more structure-shy than the previous one. If we instantiate the first equivalence with this query, and apply specialization, both sides reduce to the same point-free expression as before. Thus, like the previous example, under the schema evolution this query preserves its meaning in the weaker sense. However, when instantiating and specializing the second equivalence, a different normal form is obtained for the right-hand side:

$$
\begin{aligned}
&\mathit{concat} \circ \mathit{map}\ (\mathit{concat} \circ \mathit{map}\ (\mathit{cat} \circ (\mathit{title} \vartriangle \mathit{alias})) \circ \mathit{movies}) \\
&\quad \textbf{where}\ \mathit{movies} = \mathit{fst} \circ \mathit{unImdb} \\
&\qquad\qquad \mathit{title} \quad = \mathit{wrap} \circ \mathit{mkAny}_{\mathit{Title}} \circ \mathit{fst} \circ \mathit{snd} \circ \mathit{unMovie} \\
&\qquad\qquad \mathit{alias} \quad = \mathit{map}\ \mathit{mkAny}_{\mathit{Title}} \circ \mathit{unAlias} \circ \mathit{fst} \circ \mathit{snd} \circ \mathit{snd} \circ \mathit{unMovie}
\end{aligned}
$$

The left-hand side still reduces to the previous normal form. Thus, meaning is not preserved for this query in the stronger sense. Indeed, since the addition of alternative titles implies the presence of additional titles at a deeper level (below *alias*), this alters the results of the more structure-shy query.

Since the meaning of `//movie//title` is not preserved in the stronger sense under evolution, we are interested in computing a new query that does have the same meaning. We can do this by applying our generalization strategy to the composition of the query with the *from* function, with the following result:

$$\mathit{child} \,/\, \mathit{child} \,/\, \mathit{child} \,/\, <\mathit{Title}>$$

In sugared notation, this query reads `/*/*/title`. Interestingly, a structure-shy query is obtained with the correct meaning: it retrieves original titles, but not alternative titles. However, other structure-shy queries would have been possible. In particular, our first query `//movie/title` would have been a correct outcome. Currently, the heuristics in our generalization strategy do not lead to that result. Heuristic rules would be needed that replace chains of child axes into combinations of name and descendant axes.

## 8. Related work

*PAT-algebra*. Che et al. [8] perform XML query optimization with a transformation system based on algebraic equivalences of so-called PAT-algebra expressions. PAT-algebra expressions are meant to represent XPath queries, though they return node sets of a single static type. Numerous equivalences and corresponding rules are presented, among which are rules that exploit schema information and pre-existing indices to obtain expressions with better performance. The test-bed for performance measurement relies on translation of PAT-algebra expressions to relational database queries. Optimizations are mostly acquired by making queries *more* structure-shy, and introducing structure indices to short-cut navigation.

Our model of XPath, using strategy combinators and dynamic types, is more faithful: PAT-algebra does not offer the child, self, or descendant-or-self axes; also, only string matching predicates are modeled, while we allow boolean functions. More importantly, our approach is not limited to XPath queries. It encompasses both queries and transformations on any hierarchical data structure, and it facilitates conversion, not only among structure-shy programs, but also to and via structure-sensitive programs.

*Strategic XPath*. Lämmel [24] sketches an encoding of XPath-like combinators using strategic function combinators in the scrap-your-boilerplate style. This style uses Haskell's overloading mechanism, as provided by type classes. The XPath encoding uses dynamic typing with $\star$ for query result types. Not only are downward axes modeled, but also upward axes (parent, ancestor), and sideways axes (siblings). Node selection by name is modeled as selection by type. An indication is given how type-level programming with type classes could be used to statically exclude non-optimal queries.

The most salient difference with the strategic XPath model presented by us in Section 4.2 is the use of type classes, rather than generalized algebraic datatypes, to enable type-dependent behaviour. As far as representing and executing queries is concerned, this difference is fairly insignificant. To enable strategic behaviour, type constraints ($\mathit{Data}\ a \Rightarrow \ldots$) are used to pass implicit dictionaries, rather than additional arguments ($\mathit{Type}\ a \rightarrow \ldots$) to pass type representations. The type-class based approach is more extensible than the GADT approach, since new class instances can be added without modifying the class or existing instances. However, when it comes to query transformation, the type-class approach seems less appropriate, since it would require the encoding of our rules as type-level functions, to be executed statically by the instance resolution mechanism of the type-checker.

*Strategic programming laws*. Some algebraic laws of typed strategic program combinators have been formulated earlier, such as the $\triangleright$-Id laws, the type-preserving *mapT*-Nop law, and several laws for combinators we have not mentioned [26,25]. The type-preserving *mapT*-Fusion law was stated before [25] and has been proved by Reig [35]. We are not aware of earlier formulations of the laws for conversion between strategic and point-free programs, but they are easily derived from the reduction rules of their operational semantics provided in several other sources [39,22,28,26]. Such laws were not used earlier for the construction of transformation systems for the generalization, specialization, and optimization of typed strategic programs. The optimizations performed by the compiler of the untyped strategic term rewriting language

Stratego [38] are likely to correspond to some of the zero and cancellation laws we listed, but probably not to the specialization laws.

*Polytypic program compilation*. Polytypic, or type-indexed programming is supported by the Generic Haskell and Generic Clean languages. The standard compilation technique for these languages inserts conversion functions between user-defined datatypes and their sum-of-product representations. To optimize the resulting, often quite inefficient, code, partial evaluation techniques have been proposed [2]. Generic Haskell now supports other views of datatypes besides the standard sum-of-product representation [18]. In particular, they have an SYB view that allows encoding of strategic program combinators. It is unclear how the mentioned partial evaluation optimization techniques can be applied to these recent extensions.

*Adaptive programming*. Lämmel et al. [27] make a general comparison between strategic programming, both functional and object oriented, and adaptive programming. Adaptive programming is an extension of object-oriented programming where structure-shy traversal specifications are used to create a loose coupling between data and methods [31]. Lieberherr et al. [30] have proposed an approach to the compilation of such traversal specifications into plain object-oriented code. Compilation involves reachability analysis on the class graph and produces a dynamic roadmap to guide run-time traversal without redundant navigation.

Our query optimization approach resembles the compilation of adaptive object graph traversal specification. Both are aimed at avoiding redundant traversal and at normalization to a structure-sensitive underlying programming paradigm, i.e. point-free functional programming and object-oriented programming, respectively. The differences between these paradigms (declarative versus imperative, value-semantics versus reference semantics, object graphs versus algebraic datatypes) explain to a large extent the differences in approach (algebraic laws and compositional term rewriting systems versus global graph reachability).

*Specialization to recursive datatypes*. Recently, we have shown how this specialization mechanism can be extended to handle recursive datatypes [11]. Instead of relying on the recursive definitions of *everywhere* and *everything*, we define these combinators in terms of well-known recursion patterns such as folds and paramorphisms [33]. These recursion patterns are characterized by a rich set of equational laws, likewise to all other point-free combinators, thus enabling smooth integration in our rewrite system for specialization of structure-shy queries and transformations. We also used type-indexed type families [7,36], a new extension to the Haskell type system already supported in GHC, to bind a user-defined datatype with its functor. This technique enforces that the type representation that parameterizes the congruence and inspector of a datatype is exactly the same that parameterizes the respective embedding-projection pair, thus increasing the type-safeness of the rewrite system.

## 9. Concluding remarks

### 9.1. Contributions

We have presented an algebraic approach to transform declarative structure-shy programs. In particular, we have made the following contributions:

1. We have formulated sets of algebraic equivalences for strategic programs, of which only some had been formulated earlier, and for the conversion between strategic and point-free programs.
2. We have modeled the core of the XPath language in terms of strategic program combinators, augmented with a universal node type and associated operations. Our model relies on generalized algebraic datatypes, rather than type classes.
3. We have formulated sets of algebraic equivalences for XPath queries, and for their conversion into strategic and point-free programs. These equivalences allow derivation of static types for dynamically typed queries.
4. We have shown that the algebraic laws can be harnessed in type-safe strategic rewrite systems, encoded in Haskell, for specialization, generalization, and optimization.
5. Our approach offers a unified framework for point-free, strategic, and XPath transformations, where structure-sensitive, point-free programs are used as the solution space for transformation of structure-shy programs.

Though we have only discussed core fragments of strategic programming and XPath, we trust the reader is convinced that richer languages and rules sets can be handled in basically the same way.

### 9.2. Future work

Various aspects of the ideas presented in this paper deserve further elaboration.

*Proofs*. We have stated algebraic laws without proof. Though the validity of many simple laws is immediately evident, proofs should be constructed for some more complex laws. Also, the transformation strategies that we composed from these laws should be better characterized in terms of the normal forms to which they lead, and in terms of their complexity and termination behaviour.

*Further combinators and languages*. We intend to expand our coverage of the XPath language and strategic programming paradigm by representing and transforming more of their constructs. We also intend to address similar query and transformation languages, such as XQuery, and Stratego, and not so similar ones, such as SQL. Like XPath, Stratego does not assign static types to its programs. It may be possible to extend our approach for specializing dynamically typed XPath

queries to Stratego. The objective would be not only to infer static types for Stratego programs but to also exploit them for optimization. The addition of SQL to the mix would allow transformation of structure-shy and dynamically typed queries into relational database queries, again via intermediate structure-sensitive, statically typed point-free expressions. Assignment of strong types to SQL queries [37] could prove instrumental in these transformations.

## Acknowledgements

Thanks to Flávio Ferreira and Hugo Pacheco for inspiring discussions about the representation and transformation of XPath.

## References

[1] A. Alimarine, S. Smetsers, Optimizing generic functions, in: D. Kozen (Ed.), Proceedings of the 7th International Conference on Mathematics of Program Construction, in: LNCS, vol. 3125, Springer, 2004, pp. 16–31.
[2] A. Alimarine, S. Smetsers, Improved fusion for optimizing generics, in: M.V. Hermenegildo, D. Cabeza (Eds.), Proceedings of the 7th International Symposium on Practical Aspects of Declarative Languages, in: LNCS, vol. 3350, Springer, 2005, pp. 203–218.
[3] Tiago Alves, Paulo F. Silva, Joost Visser, Constraint-aware schema transformation, in: Proceedings of the 9th International Workshop on Rule-based Programming, 2008.
[4] Arthur I. Baars, S. Doaitse Swierstra, Typing dynamic typing, in: Proceedings of the 7th ACM SIGPLAN International Conference on Functional Programming, ACM Press, 2002, pp. 157–166.
[5] J.W. Backus, Can programming be liberated from the von Neumann style? A functional style and its algebra of programs, Communications of the ACM 21 (8) (1978) 613–641.
[6] Richard Bird, Oege de Moor, Algebra of Programming, Prentice Hall, 1997.
[7] Manuel M.T. Chakravarty, Gabriele Keller, Simon Peyton Jones, Associated type synonyms, in: Proceedings of the 10th ACM SIGPLAN International Conference on Functional Programming, ACM Press, 2005, pp. 241–253.
[8] Dunren Che, Karl Aberer, Tamer Özsu, Query optimization in XML structured-document databases, The VLDB Journal 15 (3) (2006) 263–289.
[9] J. Cheney, R. Hinze, A lightweight implementation of generics and dynamics, in: Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell, ACM Press, 2002, pp. 90–104.
[10] Alcino Cunha, José Nuno Oliveira, Joost Visser, Type-safe two-level data transformation, in: J. Misra, T. Nipkow, E. Sekerinski (Eds.), Proceedings of the 14th International Symposium on Formal Methods, in: LNCS, vol. 4085, Springer, 2006, pp. 284–299.
[11] Alcino Cunha, Hugo Pacheco, Algebraic specialization of generic functions for recursive types, in: Proceedings of the 2nd Workshop on Mathematically Structured Functional Programming, 2008.
[12] Alcino Cunha, Jorge Sousa Pinto, Point-free program transformation, Fundamenta Informaticae 66 (4) (2005) 315–352.
[13] Alcino Cunha, Jorge Sousa Pinto, José Proença, A framework for point-free program transformation, in: A. Butterfield, C. Grelck, F. Huch (Eds.), Selected Papers of the 17th International Workshop on Implementation and Application of Functional Languages, in: LNCS, vol. 4015, Springer, 2006, pp. 1–18.
[14] Alcino Cunha, Joost Visser, Strongly typed rewriting for coupled software transformation, in: Proceedings of 7th International Workshop on Rule-Based Programming, ENTCS 174 (1) (2007) 17–34.
[15] Flávio Ferreira, Hugo Pacheco, XPTO: An XPath preprocessor with type-aware optimization, Technical report, Universidade do Minho, 2007.
[16] Jeremy Gibbons, Calculating functional programs, in: R. Backhouse, et al. (Eds.), Algebraic and Coalgebraic Methods in the Mathematics of Program Construction, in: LNCS, vol. 2297, Springer, 2002, pp. 148–203.
[17] Ralf Hinze, A new approach to generic functional programming, in: Proceedings of the 27th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, ACM Press, 2000, pp. 119–132.
[18] S. Holdermans, J. Jeuring, A. Löh, A. Rodriguez, Generic views on data types, in: Tarmo Uustalu (Ed.), Proceedings of the 8th International Conference on Mathematics of Program Construction, in: LNCS, vol. 4014, Springer, 2006, pp. 209–234.
[19] Patrik Jansson, Johan Jeuring, Polyp—A polytypic programming language extension, in: Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, ACM Press, 1997, pp. 470–482.
[20] Mark Jones, Functional programming with overloading and higher-order polymorphism, in: Advanced Functional Programming: First International Spring School on Advanced Functional Programming Techniques, in: LNCS, vol. 925, Springer, 1995, pp. 97–136.
[21] Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, Geoffrey Washburn, Simple unification-based type inference for GADTs, in: Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming, ACM Press, 2006, pp. 50–61.
[22] Ralf Lämmel, Typed generic traversal with term rewriting strategies, Journal of Logic and Algebraic Programming 54 (2003).
[23] Ralf Lämmel, Coupled software transformations, in: First International Workshop on Software Evolution Transformations, November 2004 (extended abstract).
[24] Ralf Lämmel, Scrap your boilerplate with XPath-like combinators, in: Proceedings of the 34th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, ACM Press, 2007, pp. 137–142.
[25] Ralf Lämmel, Simon Peyton Jones, Scrap your boilerplate: A practical design pattern for generic programming, in: Proceedings of the ACM SIGPLAN Workshop on Types in Language Design and Implementation, SIGPLAN Notices 38 (3) (2003) 26–37.
[26] Ralf Lämmel, Eelco Visser, Joost Visser, The essence of strategic programming, October 2003. Available at http://www.cwi.nl/~ralf.
[27] Ralf Lämmel, Eelco Visser, Joost Visser, Strategic programming meets adaptive programming, in: Proceedings of the 2nd International Conference on Aspect-oriented Software Development, ACM Press, 2003, pp. 168–177.
[28] Ralf Lämmel, Joost Visser, Typed combinators for generic traversal, in: Proceedings of the 4th International Symposium on Practical Aspects of Declarative Programming, in: LNCS, vol. 2257, Springer, 2002, pp. 137–154.
[29] Ralf Lämmel, Joost Visser, A Strafunski application letter, in: V. Dahl, P. Wadler (Eds.), Proceedings of the 5th International Symposium on Practical Aspects of Declarative Programming, in: LNCS, vol. 2562, Springer, 2003, pp. 357–375.
[30] K. Lieberherr, B. Patt-Shamir, D. Orleans, Traversals of object structures: Specification and efficient implementation, ACM Transactions on Programming Languages and Systems 26 (2) (2004) 370–412.
[31] Karl J. Lieberherr, Adaptive object-oriented software: The demeter method with propagation patterns, PWS Publishing Company, Boston, 1996.
[32] José Pedro Magalhães, Stefan Holdermans, Johan Jeuring, Andres Löh, Optimizing generics is easy!, Technical Report UU-CS-2009-022, Department of Information and Computing Sciences, Utrecht University, 2009.
[33] Erik Meijer, Maarten Fokkinga, Ross Paterson, Functional programming with bananas, lenses, envelopes and barbed wire, in: J. Hughes (Ed.), Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture, in: LNCS, vol. 523, Springer, 1991.
[34] José Nuno Oliveira, Bagatelle in C arranged for VDM SoLo, Journal of Universal Computer Science 7 (8) (2001) 754–781.
[35] F. Reig, Generic proofs for combinator-based generic programs, in: H.-W. Loidl (Ed.), Trends in Functional Programming, vol. 5, Intellect, 2006, pp. 17–32.
[36] Tom Schrijvers, Martin Sulzmann, Simon Peyton Jones, Manuel M.T. Chakravarty, Towards open type functions for Haskell, in: Proceedings of the 19th International Symposium on Implementation and Application of Functional Languages, 2007, pp. 233–251.

[37] Alexandra Silva, Joost Visser, Strong types for relational databases, in: Proceedings of the 2006 ACM SIGPLAN workshop on Haskell, ACM Press, 2006, pp. 25–36.
[38] Eelco Visser, Stratego: A language for program transformation based on rewriting strategies. System description of Stratego 0.5, in: A. Middeldorp (Ed.), Rewriting Techniques and Applications, in: LNCS, vol. 2051, Springer, 2001, pp. 357–361.
[39] Eelco Visser, Zine-el-Abidine Benaissa, A core language for rewriting, ENTCS 15 (1998).
[40] Joost Visser, Visitor combination and traversal control, in: Proceedings of the 16th ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications, ACM Press, 2001, pp. 270–282.
[41] Joost Visser, Generic traversal over typed source code representations, Ph.D. thesis, University of Amsterdam, 2003.
[42] Joost Visser, Coupled transformation of schemas, documents, queries, and constraints, in: Proceedings of the 3rd International Workshop on Automated Specification and Verification of Web Systems, in: ENTCS, vol. 200, 2008, pp. 3–23.
[43] M. de Vries, Specializing type-indexed values by partial evaluation, Master's thesis, Rijksuniversiteit Groningen, 2004.
[44] W3C, XML path language (XPath) 2.0, W3C candidate recommendation, June 8, 2006.
[45] Malcolm Wallace, Colin Runciman, Haskell and XML: Generic combinators or type-based translation? in: Proceedings of the 4th ACM SIGPLAN International Conference on Functional Programming, ACM Press, 1999, pp. 148–159.
[46] Stephanie Weirich, Replib: A library for derivable type classes, in: Proceedings of the 2006 ACM SIGPLAN Workshop on Haskell, ACM Press, 2006, pp. 1–12.
[47] World Wide Web Consortium, Document object model, www.w3.org/DOM.