Transformation of Structure-Shy Programs

Applied to XPath Queries and Strategic Functions

Alcino Cunha   Joost Visser

DI-CCTC, Universidade do Minho, Portugal

{alcino,joost.visser}@di.uminho.pt

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.

Categories and Subject Descriptors D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming; F.3 [*Logics and Meanings of Programs*]: Semantics of Programming Languages

General Terms Languages, Performance, Theory

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

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 [20], strategic programming [24, 17, 18, 26], polytypic or type-indexed programming, and several XML programming languages and APIs [29].

Structure-shy programming offers various clear benefits [15, 27]. A structure-shy program can be significantly more concise, focusing on the essence of the algorithm rather than oozing with boilerplate code. 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, are dynamic checks employed in the execution of structure-shy programs to determine at each data node whether to apply 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 [28, 1, 2]. Adaptive programs are compiled to plain object-oriented programs with optimized navigation behaviour [19].

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 [3] and the ensuing tradition of algebraic transformation of point-free functional programs [9, 6]. 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, 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 show that the various algebraic laws involving structure-shy programs can be harnessed in type-safe, type-directed rewriting systems. We employ the functional language Haskell, extended with generalized algebraic datatypes, to implement such systems.

In Section 2, we briefly motivate our work with some basic examples. In Section 3, we recapitulate algebraic laws for structure-sensitive point-free functional programs, and we complement them with laws for structure-shy programs and for mediation between structure-sensitive and structure-shy programs. In Section 4, we explain the encoding in Haskell of rewrite systems that harness the algebraic laws. In Section 5, we discuss various application scenarios of our rewrite systems. Section 6 discusses related work, and Section 7 concludes.

PEPM 2007 January 15-16, 2007, Nice, France.
Copyright ©2007 ACM 978-1-59593-620-2/07/0001 ...$5.00

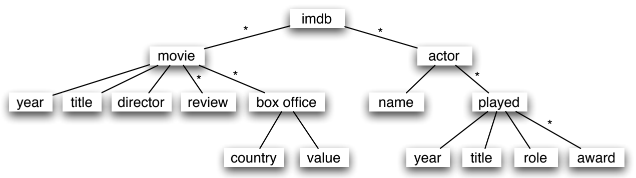


Figure 1. A movie database schema, inspired by IMDb (`http://www.imdb.com/`).

The following syntax describes essential parts of XPath:

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

The full syntax is available in the XPath language reference [29]. Abbreviated syntax is available and heavily used where for instance `//` expands to `/descendant-or-self::node()/` and an element name without preceding axis modifier expands to `/child::`*name*.

Figure 2. Summary of XPath.

2. Motivating examples

Structure-shy programming allows concise formulation of queries and transformations on rich data formats. Consider as an example the XML schema shown in Figure 1 for documents that hold information about movies. Let's consider some queries and transformations for this schema of varying degrees of structure-shyness.

Retrieve all movie directors from a document In the XPath query language, this query can be formulated as follows:

```
//movie/director
```

In particular, this query 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. See Figure 2 for a summary of XPath constructs. The 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.

The structure-shyness of the query 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 would 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.

Strategic programming was first supported in non-typed setting in the Stratego language [24]. A strongly-typed combinator suite was introduced as a Haskell library by the Strafunski system [17, 18]. This suite was generalized into the so-called 'scrap-your-boilerplate' approach to generic functional programming [14]. We focus on a limited set of combinators that convey the essence of strategic programming [16]. Combinators for type-preserving generic functions (transformations):

nop	$:: \mathsf{T}$	-- identity
$(\triangleright)$	$:: \mathsf{T} \to \mathsf{T} \to \mathsf{T}$	-- sequence
$mapT$	$:: \mathsf{T} \to \mathsf{T}$	-- map over children
mkT_A	$:: (A \to A) \to \mathsf{T}$	-- creation
apT_A	$:: \mathsf{T} \to (A \to A)$	-- application

For readability we put single-letter type constants in sans serif font. Combinators for type-unifying generic functions (queries):

$\emptyset$	$:: \mathsf{Q}\ R$	-- empty result
$(\cup)$	$:: \mathsf{Q}\ R \to \mathsf{Q}\ R \to \mathsf{Q}\ R$	-- union of results
$mapQ$	$:: \mathsf{Q}\ R \to \mathsf{Q}\ R$	-- fold over children
mkQ_A	$:: (A \to R) \to \mathsf{Q}\ R$	-- creation
apQ_A	$:: \mathsf{Q}\ R \to (A \to R)$	-- application

The result type R is assumed to be a monoid, with a $zero$ element and associative $plus$ operator. Typical derived combinators:

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

See the running text for examples of usage.

Figure 3. Strategic functional programming.

Truncate reviews to 100 characters Using strategic functional programming, this transformation can be expressed as follows:

$$everywhere\ (mkT_{Review}\ take100)$$
$$\textbf{where}\ take100\ (Review\ r) = Review\ (take\ 100\ r)$$

The *everywhere* combinator applies its generic argument function in topdown fashion to every node in a term. The mkT_A combinator applies its type-specific argument function to a given node if it is of type A, otherwise it returns the node untouched. A summary of strategic functional programming is provided in Figure 3.

This structure-shy transformation suffers from performance problems just like the structure-shy XPath query above. It traverses into parts of the document where no *Review* occurs, and it performs dynamic type tests, even though the data schema provides static information about where these tests succeed.

For optimization, we would like to transform the strategic transformation into one that does not employ strategic combinators:

$$imdb\ (map\ (movie\ id\ id\ id\ (map\ take100)\ id))\ id$$

Here we employ congruence functions such as:

$$imdb\ f\ g\ (Imdb\ m\ a) = Imdb\ (f\ m)\ (g\ a)$$

The elimination of strategic functions in favour of ordinary functions enables subsequent optimizations by a regular compiler. As illustrated in Figure 4, performance gains can be quite substantial.

On the other hand, the introduction of strategic combinators into programs that do not employ them would allow us to synthesise structure-shy from structure-sensitive programs. Code that has been developed before the advent of strategic programming, or that has been conceived for particular data structures could be made more concise, understandable, and reusable.

In Section 5, we revisit these examples and discuss further ones.

We have measured space and time consumption for the following strategic functions:

$$\begin{aligned}&\mathit{trunc} = \mathit{everywhere}\ (\mathit{mkT}_{\mathit{Review}}\ \mathit{take100})\\&\mathit{count} = \mathit{everything}\ (\mathit{mkQ}_{\mathit{Review}}\ \mathit{size})\\&\mathit{take100}\ (\mathit{Review}\ r) = \mathit{Review}\ (\mathit{take}\ 100\ r)\\&\mathit{size}\ (\mathit{Review}\ r) = \mathit{length}\ r\end{aligned}$$

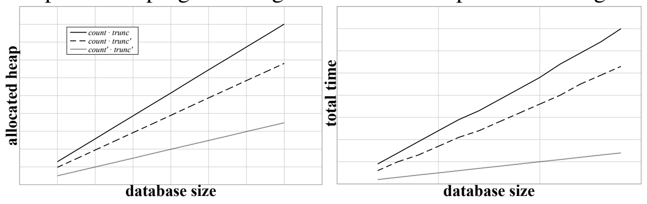
These structure-shy functions can be transformed to the following structure-sensitive functions:

$$\begin{aligned}&\mathit{trunc}' = \mathit{imdb}\ (\mathit{map}\ (\mathit{movie}\ \mathit{id}\ \mathit{id}\ \mathit{id}\ (\mathit{map}\ \mathit{take100})\ \mathit{id}))\ \mathit{id}\\&\mathit{count}' = \mathit{sum} \circ \mathit{map}\ (\mathit{sum} \circ \mathit{map}\ \mathit{size} \circ \mathit{reviews}) \circ \mathit{movies}\end{aligned}$$

Here we employ congruence and selector functions such as:

$$\begin{aligned}&\mathit{imdb}\ f\ g\ (\mathit{Imdb}\ m\ a) = \mathit{Imdb}\ (f\ m)\ (g\ a)\\&\mathit{movies}\ (\mathit{Imdb}\ m\ a) = m\end{aligned}$$

For a fair comparison, we have not used a type-class based implementation of strategic combinators [14], but our own, GADT-based implementation. For this specific example, the GADT version was roughly 14 times faster, and consumed 13 times less space. The following charts show 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: $\mathit{count} \circ \mathit{trunc}$, $\mathit{count} \circ \mathit{trunc}'$, and $\mathit{count}' \circ \mathit{trunc}'$. We compiled each program using GHC 6.4.1 with optimization flag `O1`.



Thus, in this case, type-specialization implies an improvement in space and time by factors of 2.6 and 4.8. Optimization of the transformation alone implies an improvement by a factor of roughly 1.3 in both space and time. Additional type-specialization of the query accounts for the remaining factors of 2.0 and 3.7.

Figure 4. Performance comparison for strategic functions and their type-specializations.

Point-free programs are constructed using a standard set of primitive functions and function combinators. The most fundamental are the identity function and the combinator for composing functions.

$$\begin{aligned}&\mathit{id} :: A{\rightarrow}A\\&(\circ) :: (B{\rightarrow}C) \rightarrow (A{\rightarrow}B) \rightarrow (A{\rightarrow}C)\end{aligned}$$

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

$$\begin{aligned}&\mathit{fst} :: A{\times}B \rightarrow A \qquad \mathit{snd} :: A{\times}B \rightarrow B\\&(\triangle) :: (A \rightarrow B) \rightarrow (A{\rightarrow}C) \rightarrow (A \rightarrow B{\times}C)\\&(\times) :: (A{\rightarrow}B) \rightarrow (C{\rightarrow}D) \rightarrow (A{\times}C \rightarrow B{\times}D)\end{aligned}$$

For manipulating lists we 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.

$$\begin{aligned}&\mathit{map} :: (A{\rightarrow}B) \rightarrow ([A]{\rightarrow}[B])\\&\mathit{wrap} :: A{\rightarrow}[A]\\&\mathit{filter} :: (A{\rightarrow}\mathit{Bool}) \rightarrow ([A]{\rightarrow}[A])\end{aligned}$$

In this paper we also use some overloaded functions for processing monoid types: *zero* returns the zero element of monoid A given any value, *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.

$$\mathit{zero} :: B{\rightarrow}A \qquad \mathit{plus} :: A{\times}A \rightarrow A \qquad \mathit{fold} :: [A]{\rightarrow}A$$

We also use point-free versions of if-then-else and constant *true*.

$$\begin{aligned}&\mathit{cond} :: (A{\rightarrow}\mathit{Bool}) \rightarrow (A{\rightarrow}B) \rightarrow (A{\rightarrow}B) \rightarrow (A{\rightarrow}B)\\&\mathit{true} :: A{\rightarrow}\mathit{Bool}\end{aligned}$$

Notice that false is captured by *zero* of the boolean monoid (where *plus* stands for disjunction). Combinators for further types such as sums or finite maps can also be defined, but are not used here.

Figure 5. Combinators for point-free programming.

3. Algebraic laws of structure-shy programs

In his 1977 Turing Award lecture, Backus advocated a variable-free style of functional programming, on the basis of the ease of formulating and reasoning with algebraic laws over such programs [3]. After Backus, others have adopted, complemented, and extended his work; an overview of this so-called point-free style of programming is found in [9, 6]. The function combinators and associated laws that are used in this paper are shown in Figure 5 and Figure 6.

The applicability of point-free program transformation has been enlarged to point-wise functional programs by defining a systematic way of turning functions with variables and pattern matching into equivalent point-free forms [7]. This pointwise-pointfree transform 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. 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 for pointwise programs. Note, however, that point-free and pointwise are extremities on a continuum rather than disjoint spaces. We will show that point-free program transformation may similarly be used as the solution space for transformation of structure-shy programs.

3.1 Strategic programming laws

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. Figure 7 provides an overview of laws that govern the strategic programming combinators. For example, the $\triangleright$-ID laws state 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.

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 rules, 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. Figure 8 provides an overview of laws that mediate between strategic and point-free combinators. For instance, the $\circ$-PULLT law states that sequential composition can be pulled up through mkT to obtain generic sequential composition. The first equation of the mapT-APPLY law states that application of a generic map to a product can be specialized to a product-map. Note especially the mkT-APPLY law that shows how apT and mkT can

$f \circ id = f$	$\circ$-IdR
$id \circ f = f$	$\circ$-IdL
$f \circ (g \circ h) = (f \circ g) \circ h$	$\circ$-Assoc
$f \times g = (f \circ fst) \triangle (g \circ snd)$	$\times$-Def
$fst \circ (f \triangle g) = f$	$\times$-CancelL
$snd \circ (f \triangle g) = g$	$\times$-CancelR
$(f \triangle g) \circ h = (f \circ h) \triangle (g \circ h)$	$\times$-Fusion
$fst \triangle snd = id$	$\times$-Reflex
$map\ id = id$	map-Id
$map\ f \circ zero = zero$	map-Zero
$map\ f \circ map\ g = map\ (f \circ g)$	map-Fusion
$map\ f \circ wrap = wrap \circ f$	map-Wrap
$filter\ true = id$	$filter$-True
$filter\ zero = zero$	$filter$-False
$filter\ f \circ zero = zero$	$filter$-Zero
$filter\ f \circ plus = plus \circ (filter\ f \times filter\ f)$	$filter$-Plus
$filter\ f \circ map\ g = map\ g \circ filter\ (f \circ g)$	$filter$-Map
$filter\ f \circ fold = fold \circ map\ (filter\ f)$	$filter$-Fold
$filter\ f \circ wrap = cond\ f\ wrap\ zero$	$filter$-Wrap
$true \circ f = true$	$true$-Fusion
$cond\ true\ f\ g = f$	$cond$-True
$cond\ zero\ f\ g = g$	$cond$-False
$(cond\ f\ l\ r) \circ g = cond\ (f \circ g)\ (l \circ g)\ (r \circ g)$	$cond$-Fusion
$plus \circ (zero \triangle f) = f$	$plus$-ZeroL
$plus \circ (f \triangle zero) = f$	$plus$-ZeroR
$zero \circ f = zero$	$zero$-Fusion
$fold \circ (map\ zero) = zero$	$fold$-MapZero
$fold \circ wrap = id$	$fold$-Wrap
$fold \circ map\ wrap = id$	$fold$-MapWrap
$fold \circ plus = plus \circ (fold \times fold)$	$fold$-Plus
$map\ f \circ plus = plus \circ (map\ f \times map\ f)$	map-Plus
$map\ f \circ zero = zero$	map-Zero
$fold \circ zero = zero$	$fold$-Zero
$map\ f \circ fold = fold \circ map\ (map\ f)$	map-Fold
$fold \circ fold \circ map\ f = fold \circ map\ (fold \circ f)$	$fold$-FoldMap

Figure 6. Some laws for point-free program calculation. The last set is only valid for a list monoid.

$f \rhd nop = f$	$\rhd$-IdR
$nop \rhd f = f$	$\rhd$-IdL
$f \rhd (g \rhd h) = (f \rhd g) \rhd h$	$\rhd$-Assoc
$mapT\ nop = nop$	$mapT$-nop
$mapT\ f \rhd mapT\ g = mapT\ (f \rhd g)$	$mapT$-Fusion
$f \cup \emptyset = f$	$\cup$-EmptyR
$\emptyset \cup f = f$	$\cup$-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
$everywhere\ f = f \rhd mapT\ (everywhere\ f)$	$everyw$-Def
$everything\ f = f \cup mapQ\ (everything\ f)$	$everyt$-Def

Figure 7. Laws for strategic program combinators. The $mapQ$-Fusion law is only valid for commutative monoids.

be cancelled against each other. Such cancellation may result in the elimination of unreachable nested functions.

Such mediation laws can be leveraged for specialization of structure-shy programs to structure-sensitive ones. To, conversely, increase a program's degree of structure-shyness, some additional heuristic laws are needed, which will be presented in Section 4.4.

$apT_A\ nop = id$	nop-Apply
$apT_A\ (f \rhd g) = apT_A\ f \circ apT_A\ g$	$\rhd$-Apply
$\left.\begin{array}{l} apT_A\ (mkT_A\ f) = f \\ apT_A\ (mkT_B\ f) = id, \textbf{ if } A \not\equiv B \end{array}\right\}$	mkT-Apply
$\left.\begin{array}{l} apT_{(A \times B)}\ (mapT\ f) = apT_A\ f \times apT_B\ f \\ apT_{[A]}\ (mapT\ f) = map\ (apT_A\ f) \\ apT_A\ (mapT\ f) = id, \textbf{ if } A \text{ simple type} \end{array}\right\}$	$mapT$-Apply
$mkT_A\ id = nop$	id-PullT
$mkT_A\ (f \circ g) = mkT_A\ g \rhd mkT_A\ f$	$\circ$-PullT

$apQ_A\ \emptyset = zero$	$\emptyset$-Apply
$apQ_A\ (f \cup g) = plus \circ (apQ_A\ f \triangle apQ_A\ g)$	$\cup$-Apply
$\left.\begin{array}{l} apQ_A\ (mkQ_A\ f) = f \\ apQ_A\ (mkQ_B\ f) = zero, \textbf{ if } A \not\equiv B \end{array}\right\}$	mkQ-Apply
$\left.\begin{array}{l} apQ_{(A \times B)}\ (mapQ\ f) \\ \quad = plus \circ ((apQ_A\ f) \times (apQ_B\ f)) \\ apQ_{[A]}\ (mapQ\ f) = fold \circ map\ (apQ_A\ f) \\ apQ_A\ (mapQ\ f) = zero, \textbf{ if } A \text{ simple type} \end{array}\right\}$	$mapQ$-Apply
$mkQ_A\ zero = \emptyset$	$\emptyset$-PullQ
$mkQ_A\ (plus \circ (f \triangle g)) = mkQ_A\ f \cup mkQ_A\ g$	$plus$-PullQ

Figure 8. Laws for mediating between strategic and point-free programs.

3.2 XML programming laws

Many XPath constructs can be expressed directly as strategic combinators of type Q $[\star]$, where $\star$ represents a universal node type. A summary of such an encoding is shown in Figure 9. As expressed by the $\star$ 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 following function is used to inject any type A into the universal type.

$$mkAny :: A \to \star$$

This function is idempotent, i.e. $mkAny\ (x :: \star) = x$. Likewise, the behaviour of combinators like $mapQ$, mkQ, and apQ on $\star$ is defined by their behaviour on the injected type.

Some algebraic laws for XPath combinators are presented in Figure 10. For instance, the $\cup$-Dist and $/$-Assoc combinators state distributivity and associativity properties of XPath combinators. Figure 11 presents laws for conversion and type-specialization of XPath expression. The $child$, $desc$, and $descself$ combinators are convertible to strategic combinators, as stated by various Def laws, after which the previously presented specialization laws of strategic queries can take effect. The remaining combinators can be converted directly to point-free expressions, using Apply rules. The $\star$ laws allows elimination of the $mkAny$ function. Below we will demonstrate how these rules together mediate between XPath and point-free expressions.

4. 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.

4.1 Type-safe representation of types

Some of our algebraic laws, especially those of Figures 8 and 11, make explicit reference to types. Some expose the structure of types (e.g. $\times$-Apply). Others include type equality tests (e.g. mkT-Apply). To encode these laws, we will need type-representations

Assuming a universal node type $\star$, we can encode XPath combinators as strategic program combinators (see also Figure 2):

```
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 $\langle foo \rangle$, which should not be confused with the XPath abbreviated syntax for `child::foo`. For example, the abbreviated XPath query `//movie[//review]`, which expands to:

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

is encoded as:

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

This in turn, we write shorter as *descself* / *child* / $\langle movie \rangle$? *descself* / *child* / $\langle review \rangle$ / *nonempty*.

Figure 9. Expression of XPath constructs using strategic program combinators. For a similar encoding, see [13].

$(f \cup g) \;/\; h = (f \;/\; h) \cup (g \;/\; h)$	$\cup$-DIST
$\emptyset \;/\; f = \emptyset$	/-EMPTY
$self \;/\; f = f$	/-SELFL
$f \;/\; self = f$	/-SELFR
$name\ n \;/\; name\ n = name\ n$	/-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

Figure 10. 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; $apQ_A\ (name\ n) = zero$, **otherwise**	*name*-APPLY
$apQ_\star\ f \circ mkAny\ A = apQ_A\ f$	$\star$-APPLY
$mkQ_A\ (wrap \circ mkAny\ A) = name\ n$, **if** A has name n; $mkQ_A\ (wrap \circ mkAny\ A) = self$, **otherwise**	$\star$-PULLQ

Figure 11. Laws for mediating between XPath combinators and strategic and point-free combinators. The axis definitions in term of strategic combinators can be found in [13]. A type A has name n if it is a datatype that encodes XML elements named n.

at the value level, which can be provided with the following *generalized algebraic datatype* (GADT) adapted from [10]:

```
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*)]. 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
```

Type allows the representation of some basic types, products, sums, functions, and lists. To represent user-defined datatypes, we extend it as follows (*cf* [30]):

```
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 type into values of an isomorphic type. The type b is expected to be the sum-of-products representation of the user-defined type a. The first parameter stores the name of the datatype.

Our movie database schema of Figure 1 can be represented in Haskell by the user-defined datatypes shown in Figure 12. 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 tests are performed by induction on type representations (*cf* [21]):

```
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.

```
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
```

Figure 12. Haskell datatypes for the schema of Figure 1.

4.2 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))
  Plus    :: Monoid a → F ((a, a)→a)
  Datamap :: Type b → F (b→b) → F (a→a)
  unData  :: F (a→b)
  MkAny   :: 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 $a \to b$. Constructors with an (implicitly) existentially quantified variable, such as *Comp* and *Datamap*, take a corresponding type representation as 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}
```

This argument guarantees that *Plus* can only be used for types that are monoids. The *Datamap* and *unData* constructors represent congruence and destructor functions for user-defined datatypes. The *Fun* constructor allows us to include (point-wise) functions in point-free expressions without converting them to point-free shape; it can be used for functions over which no reasoning is performed.

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 of Figure 3. Note that the query combinators take an additional argument *Monoid r* because the result type is required to be a monoid.

Finally, the XPath combinators of Figure 9 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)
data ⋆ where Any :: Type a → a → ⋆
```

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

4.3 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 monad *RewriteM* models partiality of rewrite rules, being an instance of the *MonadPlus* class:

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

Furthermore, our *RewriteM* monad offers the capability of generating rewrite traces, of which we will see an example below.

Here is an encoding of the ∘-ID laws, applied left to right:

```
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 law that involves type information is ×-DEF:

```
prod_def :: Rule
prod_def (Func (Prod a b) _) (f × g)
   = return ((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. 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))
   = return (f × g)
prod_def_inv _ ((Comp _ f Fst)△Snd) = return (f × Id)
prod_def_inv _ (Fst△(Comp _ g Snd)) = return (Id × g)
```

The extra two equations take care of cases where the inverse of *comp_id* would first need to be applied to trigger the first equation.

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 → return f
                     Nothing → return 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 :: Rule
mapT_apply _ (ApT t (MapT f)) = return (aux t)
  where aux :: Type a → F (a → a)
        aux (Prod a b) = (ApT a f) × (ApT b f)
        aux (List a)   = Listmap (ApT a f)
        aux Int        = Id
        ...
```

Note that the auxiliary function performs pattern matching on the type of ApT to dispatch to the appropriate equation.

4.4 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 suite of strategic function combinators, similar to those in Figure 3:

```
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

many r = (r ⊜ (many r)) ⊘ nop
once r = r ⊘ one (once r)
innermost r
  = all (innermost r) ⊜ ((r ⊜ innermost r) ⊘ nop)
```

The choice combinator $\oslash$ attempts to apply its first rule argument, and, if it fails, resorts to the second rule argument. The top-level application function $rewrite$ takes the result of rewriting out of the $RewriteM$ monad; in case of failure it returns the original function representation. The derived combinator $innermost$ 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 opt = comp_id ⊘ prod_def ⊘ prod_cancel ⊘ ...
        inv = prod_dev_inv ⊘ prod_fusion_inv ⊘ ...
```

The $optimize\_pf$ strategy first performs optimization of point-free functions by exhaustive application of the laws in Figure 6, oriented as rewrite rules from left to right. After that, some inverse rules are applied to make the resulting function more concise.

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 Figure 8, followed by the point-free optimization strategy:

```
optimize_t = t2pf ⊜ optimize_pf
t2pf = innermost (mapT_apply ⊘ mkT_apply ⊘ ...)
```

For generic queries we have similar optimization strategies.

Increasing structure-shyness To increase structure-shyness, we complement the laws presented in Section 3 with additional rules that are *not* valid in general, but are rather *heuristic*. Figure 13 provides a list. To prevent application of these heuristic laws when they are not valid, they must be *guarded*. For type-preserving functions, this can be done with:

```
guardT :: Rule → Rule
guardT r t f = do
  g ← r t f; f′ ← optimize_t t f; g′ ← optimize_t t g
  if (f′ ≡ g′) then return g else mzero
```

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

Figure 13. Heuristic laws for strategic and XPath combinators. These laws are not valid in general; they must be used in a *guarded* fashion.

Thus, the application of a heuristic rule r is considered successful only if its argument and result can be rewritten to the same optimized point-free expression (using $optimize\_t$). For queries, we have a similar function $guardQ$.

We have devised strategies for increasing the structure-shyness of structure-shy transformations that work in three phases. 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 Figures 8, 11, and 13, 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 Figures 7, 10 and 11, combined with the ELIM and INTRO laws of Figure 13. Thus:

```
generalize_t =
  optimize_t ⊜ mkT_apply_inv ⊜
  many (once (id_pullT ⊘ comp_pullT ⊘ ...)
          ⊘ guardT (once (map_pullT ⊘ ...))) ⊜
  many (once (seq_id ⊘ mapT_fusion ⊘ ...) ⊘ ...)
```

The mkT_apply_inv rule inserts $apT_A \circ mkT_A$ to seed the pull process of the second phase. The strategies for strategic and XPath queries are similar.

5. Application scenarios

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.

5.1 Transformations

Recall the example transformation for truncating reviews:

```
> 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's try this first for the type $Imdb$:

$$\begin{aligned}
& apT_{R}\ trunc \\
= & \{ everywhere\_apply \} \\
& apT_{R}\ (mkT_{R}\ tk \triangleright mapT\ trunc) \\
= & \{ seq\_apply \} \\
& apT_{R}\ (mkT_{R}\ tk) \circ apT_{R}\ (mapT\ trunc) \\
= & \{ mkT\_apply \} \\
& tk \circ apT_{R}\ (mapT\ trunc) \\
= & \{ gmapT\_apply \} \\
& tk \circ R\ (apT_{S}\ trunc) \\
= & \{ everywhere\_apply \} \\
& tk \circ R\ (apT_{S}\ (mkT_{R}\ tk \triangleright mapT\ trunc)) \\
= & \{ seq\_apply \} \\
& tk \circ R\ (apT_{S}\ (mkT_{R}\ tk) \circ apT_{S}\ (mapT\ trunc)) \\
= & \{ mkT\_apply \} \\
& tk \circ R\ (id \circ apT_{S}\ (mapT\ trunc)) \\
= & \{ gmapT\_apply \} \\
& tk \circ R\ id \\
= & \{ datamap\_id \} \\
& tk
\end{aligned}$$

Figure 14. Optimization of $apT_{Review}\ \ trunc$. We abbreviate $Review$ to R, $String$ to S, and $take100$ to tk.

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

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 to the result presented in Figure 4 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:

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

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 Figure 14.

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

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

A programmer that 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:

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

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

5.2 Queries

The following query computes the total length of reviews:

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

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

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

Instead of the overloaded monoid functions we present the specific ones in order to increase readability. Again we get a similar result to the one in Figure 4; in this case the difference is that the selection 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.

5.3 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):

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

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

By application of the $generalize\_xp$ query, we can maximize the structure-shyness of our queries, resulting in reconstruction of the most structure-shy of the tree, i.e. `//director`.

More challenging is the specialization of queries with predicates, such as retrieving all movies with an actor descendant and all elements with a director child:

$> \textbf{let}\ movactors = descself\ /\ \langle movie \rangle\ ?$
$\qquad\qquad\qquad\quad descself\ /\ \langle actor \rangle\ /\ nonempty$
$> rewrite\ optimize\_xp\ (apQ_{[Imdb]}\ empty)$
nil
$> \textbf{let}\ dirparents = descself\ ?\ child\ /\ \langle director \rangle\ /\ nonempty$
$> rewrite\ optimize\_xp\ (apQ_{[Imdb]}\ dirparents)$
$concat \circ map\ (map\ mkDyn \circ fst \circ unImdb)$

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`.

6. Related work

PAT-algebra Che *et al* [4] perform XML query optimization with a transformation system based on algebraic equivalences of so-called PAT-algebra expression. PAT-algebra expressions are meant to represent XPath queries, though they return nodes sets of a single static type. Numerous equivalences and corresponding rules are presented, among which 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 than PAT-algebra. PAT-algebra, as presented in [4], does not offer the child, self, or self-or-descendant 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 [13] 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 downward axes are modeled, but also upward (parent, ancestor), and sideways (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 Figure 9 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 ($Data\ \ a\ \Rightarrow\ ...$) are used to pass implicit dictionaries, rather than additional arguments ($Type\ \ a\ \rightarrow\ ...$) 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 [16, 14]. The type-preserving $mapT$-FUSION law was stated before [14] and has been proved by Reig [22]. 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 [25, 12, 17, 16]. 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 [24] 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]. These techniques do not take into account recent extensions of these languages that allow encoding of strategic program combinators [11].

Adaptive programming Lämmel *et al* [15] 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 [20]. Lieberherr *et al* [19] have proposed an approach to 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).

Coupled rewriting Previously, we have shown how a strategic rewrite system for two-level transformations [5], such as data mappings and format evolution, can be combined with a type-preserving rewrite system for point-free program transformation, to support coupled transformation of data formats, instances, and processors [8]. These point-free program transformations have been generalized to structure-shy programs by the current paper, which has the immediate consequence that our approach to coupled transformation now also encompasses migration and mapping of structure-shy queries.

7. Concluding remarks

7.1 Contributions

We have presented an algebraic approach to transformation of 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.

7.2 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.

Recursion A limitation of our approach is that we do not yet handle (mutually) recursive datatypes and functions. A finite representation of such recursive types and functions is needed for terminating program transformations. We are currently developing such representations.

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 [23] could prove instrumental in these transformations.

Front-ends We are currently developing a front-end that allows parsing of the XPath surface language into our GADT for type-safe representation of functions, and pretty-printing them back into textual form.

Acknowledgments

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

References

[1] A. Alimarine and S. Smetsers. Optimizing generic functions. In D. Kozen, editor, *Mathematics of Program Construction*, volume 3125 of *LNCS*, pages 16–31. Springer, 2004.

[2] A. Alimarine and S. Smetsers. Improved fusion for optimizing generics. In M.V. Hermenegildo and D. Cabeza, editors, *Proc. 7th Int. Symp. Practical Aspects of Declarative Languages*, volume 3350 of *LNCS*, pages 203–218. Springer, 2005.

[3] J.W. Backus. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Commun. ACM*, 21(8):613–641, 1978.

[4] D. Che, K. Aberer, and T. Özsu. Query optimization in XML structured-document databases. *The VLDB Journal*, 15(3):263–289, 2006.

[5] A. Cunha, J.N. Oliveira, and J. Visser. Type-safe two-level data transformation. In J. Misra, T. Nipkow, and E. Sekerinski, editors, *Proc. Formal Methods, 14th Int. Symp. Formal Methods Europe*, volume 4085 of *LNCS*, pages 284–299. Springer, 2006.

[6] A. Cunha and J. Sousa Pinto. Point-free program transformation. *Fundam. Inform.*, 66(4):315–352, 2005.

[7] A. Cunha, J. Sousa Pinto, and J. Proença. A framework for point-free program transformation. In A. Butterfield, C. Grelck, and F. Huch, editors, *Selected papers 17th Int. Workshop on Implementation and Application of Functional Languages*, volume 4015 of *LNCS*, pages 1–18. Springer, 2006.

[8] A. Cunha and J. Visser. Strongly typed rewriting for coupled software transformation. In M. Fernandez and R. Lämmel, editors, *Proc. 7th Int. Workshop on Rule-Based Programming (RULE 2006)*, ENTCS. Elsevier, 2006. To appear.

[9] J. Gibbons. Calculating functional programs. In R. Backhouse et al., editors, *Algebraic and Coalgebraic Methods in the Mathematics of Program Construction*, volume 2297 of *LNCS*, chapter 5, pages 148–203. Springer, 2002.

[10] R. Hinze, A. Löh, and B.C.d.S. Oliveira. "Scrap your boilerplate" reloaded. In M. Hagiya and P. Wadler, editors, *Proc. Functional and Logic Programming, 8th Int. Symp.*, volume 3945 of *LNCS*, pages 13–29. Springer, 2006.

[11] S. Holdermans, J. Jeuring, A. Löh, and A. Rodriguez. Generic views on data types. In Tarmo Uustalu, editor, *Proc. 8th Int. Conf. Mathematics of Program Construction*, volume 4014 of *LNCS*, pages 209–234. Springer, 2006.

[12] R. Lämmel. Typed Generic Traversal With Term Rewriting Strategies. *Journal of Logic and Algebraic Programming*, 54, 2003.

[13] R. Lämmel. Scrap your boilerplate with XPath-like combinators, 15 July 2006. Draft, 6 pages, Accepted as short paper at POPL 2007.

[14] R. Lämmel and S. Peyton Jones. Scrap your boilerplate: a practical design pattern for generic programming. *ACM SIGPLAN Notices*, 38(3):26–37, March 2003. Proc. ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI 2003).

[15] R. Lämmel, E. Visser, and J. Visser. Strategic programming meets adaptive programming. In *AOSD '03: Proc. 2nd Int. Conf. Aspect-Oriented Software Development*, pages 168–177. ACM Press, 2003.

[16] R. Lämmel, E. Visser, and J. Visser. The Essence of Strategic Programming. Available at `http://www.cwi.nl/~ralf`, 2003.

[17] R. Lämmel and J. Visser. Typed Combinators for Generic Traversal. In *Proc. Practical Aspects of Declarative Programming PADL 2002*, volume 2257 of *LNCS*, pages 137–154. Springer, January 2002.

[18] R. Lämmel and J. Visser. A Strafunski Application Letter. In V. Dahl and P. Wadler, editors, *Proc. of Practical Aspects of Declarative Programming (PADL'03)*, volume 2562 of *LNCS*, pages 357–375. Springer, January 2003.

[19] K. Lieberherr, B. Patt-Shamir, and D. Orleans. Traversals of object structures: Specification and efficient implementation. *ACM Trans. Program. Lang. Syst.*, 26(2):370–412, 2004.

[20] Karl J. Lieberherr. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*. PWS Publishing Company, 1996.

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

[22] F. Reig. Generic proofs for combinator-based generic programs. In H.-W. Loidl, editor, *Trends in Functional Programming*, volume 5, pages 17–32. Intellect, 2006.

[23] Alexandra Silva and Joost Visser. Strong types for relational databases. In *Proc. ACM SIGPLAN workshop on Haskell*, pages 25–36. ACM Press, 2006.

[24] E. Visser. Stratego: A language for program transformation based on rewriting strategies. System description of Stratego 0.5. In A. Middeldorp, editor, *Rewriting Techniques and Applications*, volume 2051 of *LNCS*, pages 357–361. Springer, May 2001.

[25] E. Visser and Z. Benaissa. A core language for rewriting. *Electr. Notes Theor. Comput. Sci.*, 15, 1998.

[26] J. Visser. Visitor combination and traversal control. *ACM SIGPLAN Notices*, 36(11):270–282, 2001. Proc. ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications.

[27] J. Visser. *Generic Traversal over Typed Source Code Representations*. PhD thesis, University of Amsterdam, 2003.

[28] M. de Vries. Specializing type-indexed values by partial evaluation. Master's thesis, Rijksuniversiteit Groningen, 2004.

[29] W3C. XML path language (XPath) 2.0, W3C candidate recommendation, June, 8 2006.

[30] S. Weirich. RepLib: a library for derivable type classes. In *Proc. ACM SIGPLAN workshop on Haskell*, pages 1–12. ACM Press, 2006.