# 1. Monad Comprehensions: A Versatile Representation for Queries

Torsten Grust

University of Konstanz, Department of Computer and Information Science,
78457 Konstanz, Germany
e-mail: Torsten.Grust@uni-konstanz.de

This chapter is an exploration of the possibilities that open up if we consistently adopt a style of database query and collection processing which allows us to look *inside* collections and thus enables us to play with atomic constructors instead of the monolithic collection values they build.

This comprehension of values goes well together with a completely functional style of query formulation: queries map between the constructors of different collection types. It turns out that a single uniform type of mapping, the *catamorphism*, is sufficient to embrace the functionality of today's database query languages, like SQL, OQL, but also XPath. *Monad comprehensions* provide just the right amount of syntactic sugar to express these mappings in a style that is similar to relational calculus (but goes beyond its expressiveness).

The major portion of this chapter, however, demonstrates how monad comprehensions enable a succinct yet deep understanding of database queries. We will revisit a number of problems in the advanced query processing domain to see how monad comprehensions can (a) provide remarkably concise proofs of correctness for earlier work, (b) clarify and then broaden the applicability of existing query optimisation techniques, and (c) enable query transformations which otherwise require extensive sets of rewriting rules.

## 1.1 A Functional Seed

In line with the major theme of this book, we perceive query translation and transformation as a *functional programming activity*. Superficially, this concerns a number of notational conventions we will adopt. More deeply, you will note that we generate query results solely through the side-effect free construction of values from simpler constituents and that functional composition will be the predominant way of forming complex queries. Referential transparency is the key to transformational programming and equational reasoning.

Relatively few components are needed in our initial query language core. We grow this language through *function definitions* of the form

$$f \equiv e$$

where $e$ is an expression built from components we have already introduced. The functions $f$ so defined will get more complex as we go on until we are ready to give the meaning of SQL, OQL [1.4], or XPath [1.1] query clauses such as `select-from-where`, `exists-in`, `flatten`, or path expressions.

### 1.1.1 Notation, Types, and Values

If you are familiar with notational conventions of functional programming languages such as Haskell [1.14] you will feel at home right away. Figure 1.1 introduces the core expression forms $e$ and their notation.

| | | | |
|---|---|---|---|
| $e$ | ::= | $c$ | constants |
| | \| | $v$ | variables |
| | \| | $\lambda p \to e$ | lambda abstraction |
| | \| | $v \equiv e$ | (recursive) function definition |
| | \| | $(e,e)$ | pair former |
| | \| | $e\,e$ | function application |
| | \| | `case` $p$ `of` $e \to e \mid \ldots \mid e \to e$ | case (pattern matching) |
| | \| | $e \uparrow e$ | insertion constructor |
| | \| | `[]` \| `{|}` \| `{}` | empty list, empty bag, empty set |
| | \| | $e\ op\ e$ | infix operator ($op = +, *, =, <, >, \ldots$) |
| $p$ | ::= | $c$ | constants |
| | \| | $v$ | variable binding |
| | \| | $(p,p)$ | pair pattern |
| | \| | $p \uparrow p$ | collection pattern |

**Fig. 1.1.** Core language syntax. The *insertion constructor* $\uparrow$ will be introduced in Section 1.1.2.

We assume the presence of a *prelude*, *i.e.*, a library of basic function definitions which makes working with the core language somewhat less tedious, *e.g.*: $\texttt{id} \equiv \lambda x \to x$, $\texttt{fst} \equiv \lambda(v_1,v_2) \to v_1$ (and corresponding `snd`). The function definition $f \equiv \lambda x \to e$ may also be written as $f\,x \equiv e$. The core is *strongly* and *statically typed*. This means that any value—including functions—has a unique type which we can deduce from its definition alone. We write $e :: t$ to indicate that value $e$ has type $t$. The application of a function to wrongly typed arguments is bound to fail. Figure 1.2 summarizes the types $t$ we will encounter. Some values are *polymorphic*, *i.e.*, their type includes type variables which (consistently) assume specific types when the value is used. The left projection `fst` has the polymorphic type $\forall \alpha\beta.\, \alpha \times \beta \to \alpha$ and can thus be applied to pairs of arbitrarily typed constituents. (The type quantifier $\forall \alpha$ indicates that $\alpha$ may indeed be instantiated by any type; we assume its implicit presence whenever polymorphic types are used.)

We draw constants from a pool of domains of atomic types that we choose according to the actual query language we need to represent: if the query lan-

| | | | |
|---|---|---|---|
| $t$ | ::= | $\mathbb{N} \mid \mathbb{B} \mid \mathbb{S} \mid \ldots$ | atomic (numeric, boolean, string, ...) |
| | $\mid$ | $v$ | variables ($\alpha, \beta, \gamma, \ldots$) |
| | $\mid$ | $t \to t$ | functions |
| | $\mid$ | $t \times t$ | pairs |
| | $\mid$ | $[t] \mid \{\!\mid t \mid\!\} \mid \{t\}$ | list (bag, set) type constructor |

**Fig. 1.2.** Core language types.

guage supports numeric constants and arithmetic on these, we incorporate numeric type $\mathbb{N}$ and operations on it in the core language. If the query language supports dates *e.g.*, values of the form `Oct 8 2002`, we incorporate an atomic `Date` type or choose an *implementation type* such as $\mathbb{N} \times \mathbb{N} \times \mathbb{N}$ (which represents the month, day, year constituents of a date value via three numbers) or simply $\mathbb{S}$ (a character string using an appropriate date format).

### 1.1.2 Constructing Collections

Remember that we are growing this language for a specific purpose: to represent database query languages. So, where a typical functional language would offer *lists* only, the core supports the *collection types bags* (multi-sets) and *sets* as well. Again, this is a means to properly reflect the type system of the query language: SQL primarily operates on bags, while OQL includes clauses that operate on all three collection types.

Starting from an empty collection ($[]$, $\{\!\mid\!\}$, or $\{\}$), we can *insert* elements one by one using constructor $\uparrow$ to construct a more complex collection value.

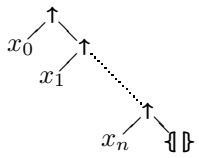
To aid compact notation, we define the *insertion constructor* $\uparrow$ as overloaded, *i.e.*, the type of its second argument determines its behaviour. Let $x :: \alpha$. Then:

$$x \uparrow xs = \begin{cases} [x] +\!\!+ xs & \text{if } xs :: [\alpha] \\ \{\!\mid x \mid\!\} \uplus xs & \text{if } xs :: \{\!\mid \alpha \mid\!\} \\ \{x\} \cup xs & \text{if } xs :: \{\alpha\} \\ \text{type error} & \text{otherwise} \end{cases}$$

($+\!\!+$ denotes list concatenation, while $\uplus$ is bag union respecting multiplicity of elements.) Note that *insertion order* is only relevant if $\uparrow$ constructs lists (in this case, $\uparrow$ is also widely known as *cons*). Insertion of *duplicates* is respected if $\uparrow$ constructs lists or bags. Set insertion $\uparrow :: \alpha \times \{\alpha\} \to \{\alpha\}$ disregards both order and duplicates, *i.e.*, the constructor is commutative and idempotent[1].

We assume that $\uparrow$ is right-associative so that $x_0 \uparrow x_1 \uparrow \ldots x_n \uparrow \{\!\mid\!\}$ corresponds to the following parse tree, which we also term the the *spine* of the collection:

[1] As the type of constructor $\uparrow$ suggests, we are actually talking of *left-commutativity* $y \uparrow x \uparrow xs = x \uparrow y \uparrow xs$ and/or *left-idempotence* $x \uparrow x \uparrow xs = x \uparrow xs$. Note that element type $\alpha$ in the set case requires a notion of equality, $= :: \alpha \times \alpha \to \mathbb{B}$, to decide if a duplicate has been inserted into a set.




We will also write this expression as $\{\!|\, x_0, x_1, \ldots, x_n \,|\!\}$.

## 1.2 Spine Transformers

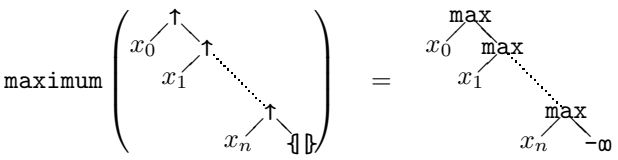
Programming with collections in our core language consequently means writing programs that create, transform, and analyse spines. To provide a taste of the resulting programming style, here is a function that computes the maximum element of a given collection of numbers assuming that the prelude contains a definition $\texttt{max}\,(x,y) \equiv \texttt{case}\, x < y \,\texttt{of}\, \texttt{true} \rightarrow y \mid \texttt{false} \rightarrow x$:

```
maximum :: {| ℕ |} → ℕ
maximum xs ≡ case xs of {| |}      → -∞
                      | x ↑ xs′ → max (x, maximum xs′)
```

There are two things to note here:

(1) As indicated in the introduction to this chapter, we are analysing and building collection values on the basis of their constructors.
(2) The two `case` branches exactly correspond with the two principal forms a collection value can take: empty (here: $\{\!|\,|\!\}$) or constructed ($x \uparrow xs'$). In the latter branch, `maximum` cuts off $x$ and recurses on $xs'$.

The second observation is particularly interesting for our forthcoming discussion. It effectively states that `maximum` acts like a *spine transformer*:




In other words, `maximum` performs its computation solely through *consistent replacement of constructors*.

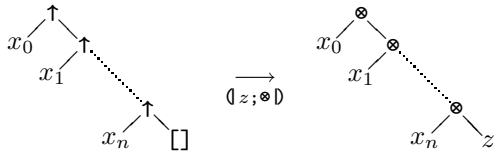
This pattern of computation seems to be rather rigid but in fact it is far from that: the expressive power of these spine transformers is sufficient to embrace almost all computations expressible by current database query languages. We will thus adopt spine transformers as *the* basic query building block.

### 1.2.1 Catamorphisms

To stress this idea of deriving a recursive computation from the recursive structure of the input collection, let us undertake a generalisation step. Given a collection $[\alpha]$ (or $\{\!|\alpha|\!\}$, $\{\alpha\}$) and values $z :: \beta$, $\otimes :: \alpha \times \beta \to \beta$ we define the overloaded mix-fix operator $(\!|\, |\!)$ as

$$\begin{aligned}
&(\!|z;\otimes|\!) :: \beta \times (\alpha \times \beta \to \beta) \to [\alpha] \to \beta \\
&(\!|z;\otimes|\!)\, xs \equiv \mathtt{case}\ xs\ \mathtt{of}\ [] \quad\ \to z \\
&\qquad\qquad\qquad\qquad\quad \mid x \uparrow xs' \to x \otimes (\!|z;\otimes|\!)\, xs'
\end{aligned}$$

Pictorially, $(\!|z;\otimes|\!)$ is the spine transformer




and we can immediately see that we could have defined $\mathtt{maximum} \equiv (\!|-\infty;\mathtt{max}|\!)$. When applied to lists, the operator $(\!|\, |\!)$ is known as `foldr` or `reduce`, especially in the functional programming community. In more general collection programming settings, $(\!|\, |\!)$ is also known as *sri* (structural recursion on insert) [1.2, 1.21].

We can give an algebraic account of the nature of $(\!|\, |\!)$. Observe that $(\!|z;\otimes|\!)$ is a solution to the equations below which effectively say that the unknown $h$ is a *homomorphism* from monoid $([], \uparrow)$ to monoid $(z, \otimes)$:

$$h\ [] = z \tag{1.1a}$$

$$h\ (x \uparrow xs) = x \otimes h\, xs \tag{1.1b}$$

It can be shown—based on the fact that $([], \uparrow)$ is the *term* or *initial algebra* of lists built using these two constructors—that $(\!|z;\otimes|\!)$ is the *unique* solution to these equations, completely determined by $z$ and $\otimes$ [1.16]. Homomorphisms of initial algebras have been dubbed *catamorphisms* [1.17] and this is the terminology we will adopt.

*Caveat*: Equation (1.1b) suggests that operator $\otimes$ of the target algebra must *not* be completely arbitrary: $\otimes$ needs to have the same algebraic properties as $\uparrow$: associativity, left-commutativity (if $\uparrow :: \alpha \times \{\!|\alpha|\!\} \to \{\!|\alpha|\!\}$ or $\uparrow :: \alpha \times \{\alpha\} \to \{\alpha\}$), or left-idempotence (if $\uparrow :: \alpha \times \{\alpha\} \to \{\alpha\}$).

Catamorphisms are a versatile tool. A number of useful collection processing functions turn out to be catamorphisms:

$$\begin{aligned}
\mathtt{maximum} &\equiv (\!|-\infty;\mathtt{max}|\!) \\
\mathtt{minimum} &\equiv (\!|+\infty;\mathtt{min}|\!) \\
\mathtt{or} &\equiv (\!|\mathtt{false};\vee|\!) \\
\mathtt{and} &\equiv (\!|\mathtt{true};\wedge|\!)
\end{aligned}$$

$$
\begin{aligned}
xs \oplus ys &\equiv (\!|\, ys ; \uparrow \,|\!)\, xs \\
\texttt{first} &\equiv (\!|\, 0 ; \texttt{fst} \,|\!) \\
\texttt{list\_map}\, f &\equiv (\!|\, [] ; \lambda(x, xs) \to (f\ x) \uparrow xs \,|\!) \\
\texttt{flatten} &\equiv (\!|\, [] ; \oplus \,|\!)
\end{aligned}
$$

Note that infix operator $\oplus$ is overloaded and behaves like $\dot{-}$, $\uplus$, or $\cup$ depending on the type of its arguments. As given, `list_map` is well-defined on lists only. The same is true for function `first`: `fst` is neither left-commutative nor left-idempotent, an expression of the fact that there is no notion of a first element in a bag or set.

### 1.2.2 Catamorphism Fusion

A query translator and optimizer based on the core language we have defined so far would more closely resemble a *program transformation system* than a traditional query optimizer. To ensure that the system can operate completely unguided and without the need for *Eureka steps*—transformation steps not immediately motivated by the goal the overall transformation strives for—we need to be restrictive in the program forms we may admit.

Catamorphisms represent this restricted form of computation and in our case, simplicity enables optimisation.

Reconsider `list_map`. We can turn this function into a generic `map` catamorphism if we make its implicit use of the list constructors `[]` and $\uparrow :: \alpha \times [\alpha] \to [\alpha]$ explicit and thus define:

$$\texttt{map}\ n\ c\ f \equiv (\!|\, n ; \lambda(x, xs) \to c\ (f\ x, xs) \,|\!)$$

Now, $\texttt{list\_map}\ f \equiv \texttt{map}\ []\ (\uparrow)\ f$, $\texttt{set\_map}\ f \equiv \texttt{map}\ \{\}\ (\uparrow)\ f$, and $\texttt{bag\_map}\ f \equiv \texttt{map}\, \{\!|\, |\!\}\, (\uparrow)\ f$.

Apart from this generalisation, factoring out the constructors out of a catamorphism opens up an important optimisation opportunity: we can "reach inside" a catamorphism and influence the constructor replacement it performs. This is all we need to formulate a simple yet effective *catamorphism fusion law*. Let `cata` denote any catamorphism with constructors factored out like above, then

$$(\!|\, z ; \otimes \,|\!) \cdot \texttt{cata}\ n\ c = \texttt{cata}\ z\ \otimes \tag{1.2}$$

Note that while the lefthand side walks the spine twice, the righthand side computes the same result in a single spine traversal. With catamorphisms being the basic program building blocks, a typical program form will be catamorphism compositions. These composition chains can be shortened and simplified using law (1.2). The two-step catamorphism chain below decides if there is *any* element in the input satisfying $p$. Catamorphism fusion merges the steps and yields a general purpose existential quantifier `exists` $p$:

$$\texttt{exists}\, p \quad \equiv \quad \texttt{or} \cdot \texttt{map}\ \{\}\ \uparrow\ p \quad = \quad \texttt{map}\ \texttt{false}\ \vee\ p$$

Law (1.2) is known as *cheap deforestation* [1.9] or the *acid rain theorem* [1.22]. Its correctness obviously depends on `cata` being well-behaved: `cata` is *required* to exclusively use the supplied constructors $c$ and $n$ to build its result. Perhaps surprisingly, one can formulate a prerequisite that restricts the *type* of `cata` to ensure this behaviour (parametricity of `cata` [1.23]).

## 1.3 Monad Comprehensions

We have seen that catamorphisms represent a form of computation restrictive enough to enable mechanical program optimisations, yet expressive enough to provide a useful target for query translation.

However, we need to make sure that query translation actually yields nothing but compositions of catamorphisms. This is what we turn to now.

To achieve this goal, we grow our language once more to include the expressions of the *monad comprehension calculus* [1.24, 1.25] whose syntactic forms closely resemble the well-known relational calculus. The calculus is a good candidate to serve as a translation target for user-level query syntax [1.3]. Its semantics can be explained in terms of catamorphisms which completes the desired query translation framework:

*Query syntax* → *monad comprehension calculus* → *catamorphisms.*

Figure 1.3 displays the syntactic sugar $mc$ introduced by the monad comprehension calculus.

| | | | |
|---|---|---|---|
| $mc$ | ::= | $e$ | core language (Figure 1.1) |
| | \| | $[\![ mc \mid qs ]\!] \mid \{\!\mid mc \mid qs \mid\!\}$ | monad comprehension |
| $qs$ | ::= | $\varepsilon$ | empty |
| | \| | $q$ | qualifier |
| | \| | $qs, qs$ | qualifiers |
| $q$ | ::= | $v \leftarrow mc$ | generator |
| | \| | $mc$ | filter |

**Fig. 1.3.** Syntax of the Monad Comprehension Calculus

We obtain a relational calculus-style sublanguage that can succinctly express computations over lists, bags, and sets (actually over any *monad*—we will shortly come to this). The general syntactic form is

$$[\![ e \mid q_0, \ldots, q_n ]\!]$$

Informally, the semantics of this comprehension read as follows: starting with *qualifier* $q_0$, a *generator* $q_i = v_i \leftarrow e_i$ sequentially binds $v_i$ to the elements of its

*range* $e_i$. This binding is propagated through the list of qualifiers $q_{i+1}, \ldots, q_n$. *Filters* are qualifiers of type $\mathbb{B}$ (boolean). A binding is discarded if a filter evaluates to `false` under it. The *head* expression $e$ is evaluated for those bindings that satisfy all the filters, and the resulting values are collected to form the final result list.

Here is how we can define `bag_map` $f$ and `flatten`:

$$\begin{aligned} \texttt{bag\_map}\ f\ xs &\equiv \{\!|\, f\ x \mid x \leftarrow xs \,|\!\} \\ \texttt{flatten}\ xss &\equiv \{x \mid xs \leftarrow xss, x \leftarrow xs\} \end{aligned}$$

SQL and OQL queries, like the following *semi-join* between relations $r$ and $s$, may now be understood as yet more syntactic sugar (we will encounter many more examples in the sequel):

$$\begin{aligned} &\texttt{select}\ r \\ &\quad \texttt{from}\ r, s \quad \equiv \quad \{\!|\, v_1 \mid v_1 \leftarrow r, v_2 \leftarrow s, p \,|\!\} \\ &\quad \texttt{where}\ p \end{aligned}$$

Note that the grammar in Figure 1.3 allows for arbitrary nesting of monad comprehensions. The occurrence of a comprehension as generator range, filter, or head will allows us to express the diverse forms of query nesting found in user-level query languages [1.10, 1.12].

Figure 1.4 gives the translation scheme in the core language for the monad comprehension calculus. It is based on the so-called *Wadler identities* which were originally developed to explain the semantics of list comprehensions. The scheme of Figure 1.4, however, is applicable to bag and set comprehensions as well (simply consistently replace all occurrences of $[\![\,,\,]\!]$ by $[\,,\,]$ or $\{\!|\,,\,|\!\}$ or $\{\,,\,\}$, respectively). These translation rules, to be applied top-down, reduce a

$$[\![ e \mid ]\!] \equiv \texttt{unit}\ e \tag{1.3a}$$

$$[\![ e \mid v \leftarrow e' :: [\![\alpha]\!] ]\!] \equiv \texttt{mmap}\ (\lambda v \rightarrow e)\ e' \tag{1.3b}$$

$$[\![ e \mid v \leftarrow e' :: [\alpha] ]\!] \equiv \texttt{mmap id}\ ([e \mid v \leftarrow e']) \tag{1.3c}$$

$$[\![ e \mid v \leftarrow e' :: \{\!|\alpha|\!\} ]\!] \equiv \texttt{mmap id}\ (\{\!|e \mid v \leftarrow e'|\!\}) \tag{1.3d}$$

$$[\![ e \mid v \leftarrow e' :: \{\alpha\} ]\!] \equiv \texttt{mmap id}\ (\{e \mid v \leftarrow e'\}) \tag{1.3e}$$

$$[\![ e \mid e' :: \mathbb{B} ]\!] \equiv \texttt{case}\ e'\ \texttt{of true} \rightarrow \texttt{unit}\ e \mid \texttt{false} \rightarrow \texttt{zero} \tag{1.3f}$$

$$[\![ e \mid qs, qs' ]\!] \equiv \texttt{join}\ ([\![ e \mid qs' ]\!] \mid qs ]\!]) \tag{1.3g}$$

$$\begin{aligned} \texttt{zero} &\equiv [\![\,]\!] \\ \texttt{unit}\ e &\equiv [\![ e ]\!] \\ \texttt{mmap} &\equiv \texttt{map}\ [\![\,]\!]\ (\uparrow) \\ \texttt{join} &\equiv (\!|\, [\![\,]\!]\ ; \oplus \,|\!) \end{aligned}$$

**Fig. 1.4.** Monad Comprehension Semantics

monad comprehension step by step until we are left with an equivalent core

language expression. Definition (1.3g) breaks a complex qualifier list down to single generator or filters. Note how (1.3c, 1.3d, 1.3e) examine the type of the generator range to temporarily switch to a list, bag, or set comprehension. The results are then coerced using `mmap id` which effectively enables us to mix and match comprehensions over different collection types. (Coercion is not completely arbitrary since the well-definedness condition for catamorphisms of Section 1.2.2 applies. This restriction is rather natural, however, as it forbids non-well-founded coercions like the conversion of a set into a list.)

Monad comprehensions provide quite powerful syntactic sugar and will save us from juggling with complex catamorphism chains. Consider, for example, the translation of `filter`$p$ (which evaluates predicate $p$ against the elements of the argument list):

$$\begin{aligned}
\texttt{filter}\,p\,xs &\equiv [x \mid x \leftarrow xs, p\,x] \\
&= \texttt{join}\,([[x \mid p\,x] \mid x \leftarrow xs]) \\
&= (\texttt{join} \cdot \texttt{mmap}\,(\lambda x \rightarrow [x \mid p\,x]))\,xs \\
&= \texttt{map}\,[]\,\oplus\,(\lambda x \rightarrow [x \mid p\,x])\,xs \\
&= \texttt{map}\,[]\,\oplus\,(\lambda x \rightarrow \texttt{case}\,p\,x\,\texttt{of}\,\texttt{true} \rightarrow [x] \mid \texttt{false} \rightarrow [])\,xs
\end{aligned}$$

Interestingly, comprehensions are just the "syntactic shadow" of a deeper, categorical concept: *monads* [1.24]. Comprehension syntax can be sensibly defined for any type constructor $[\![\,\alpha\,]\!]$ with operations `mmap`, `zero`, `unit`, `join` obeying the laws of a *monad with zero* which—for our collection constructors—are as follows:

$$\texttt{join} \cdot \texttt{unit} = \texttt{id} \tag{1.4a}$$

$$\texttt{join} \cdot \texttt{mmap}\,\texttt{unit} = \texttt{id} \tag{1.4b}$$

$$\texttt{join} \cdot \texttt{join} = \texttt{join} \cdot \texttt{mmap}\,\texttt{join} \tag{1.4c}$$

$$\texttt{join} \cdot \texttt{zero} = \texttt{zero} \tag{1.4d}$$

$$\texttt{join} \cdot \texttt{mmap}\,\texttt{zero} = \texttt{zero} \tag{1.4e}$$

With the definitions given in Figure 1.4, lists, bags, and sets are easily verified to be monad instances. Monads are a remarkably general concept that has been widely used by the functional programming community to study, among others, I/O, stateful computation, and exception handling [1.19]. Monad comprehensions have even found their way into mainstream functional programming languages[2]. We will meet other monads in the upcoming sections.

More importantly, though, we can exercise a large number of query transformations and optimisation exclusively in comprehension syntax.

[2] Haskell [1.14] being the primary example here, although monad comprehensions come in the disguise of Haskell's `do`-notation these days.

## 1.4 Type Conversion Saves Work

Perhaps the principal decision in solving a problem is the choice of language in which we represent both the problem and its possible solutions. Choosing the "right" language can turn the concealed or difficult into the obvious or simple. This section exemplifies one such situation and we argue that the functional language we have constructed so far provides and efficient framework to reason about queries.

Some constructs introduced in recent SQL dialects (being liberal, we count OQL as such) have no immediate counterpart in the traditional relational algebra. Among these, for example, are *type conversion* or *extraction* operators like OQL's `element`: the query `element` $e$ tests if $e$ evaluates to a singleton collection and, if so, returns the singleton element (tuple, object, ...) itself. Otherwise, an exception is raised. SQL 3 introduces so-called *row sub-queries* which exhibit the same behaviour. The type of such an operator is $[\![\,\alpha\,]\!] \to \alpha$.

Different placements of a type conversion operator in a query may have dramatic effects on the query plan's quality. Early execution of type conversion can lead to removal of joins or even query unnesting. Consider the OQL query below (we use the convention that a query expression like $f\,x\,y$ denotes a query $f$ containing free variables $x, y$, *i.e.*, $f$ is a function of $x, y$)

$$\texttt{element}\,(\texttt{select}\ f\,x\,y \quad \texttt{from}\ xs\ \texttt{as}\ x\texttt{,}\ ys\ \texttt{as}\ y)$$

Computing the join between $xs$ and $ys$ is wasted work as we are throwing the result away should the join (unexpectedly) contain more than one element (in which case the query raises an exception). A *type conversion aware* optimizer could emit the equivalent

$$f\ (\texttt{element}\ xs)\ (\texttt{element}\ ys)$$

The join is gone as is the danger of doing unnecessary work. Pushing down type conversion has a perilous nature, though:

- The above rewrite does not preserve equivalence if we compute with sets (`select distinct...`): function $f$ might not be one-to-one. If, for example, we have $f\,x\,y \equiv c$, then the query

  $$\texttt{element}\,(\texttt{select distinct}\ f\,x\,y \quad \texttt{from}\ xs\ \texttt{as}\ x\texttt{,}\ ys\ \texttt{as}\ y)$$

  effectively computes `element {`$c$`}` $= c$ for arbitrary non-empty collections $xs$ and $ys$, while the rewritten query will raise an exception should $xs$ or $ys$ contain more than one element.
- We must not push type conversion beyond a selection: the selection might select exactly one element (selection on a key) and thus satisfy `element`

while pushing down `element` beyond the selection might lead to an application of `element` to a collection of cardinality greater than one and thus raise an exception instead.

How do we safely obtain the optimized query? This is where our functional query language jumps in. First off, note that we can represent `element` as

$$\begin{aligned}&\texttt{element} \equiv \texttt{snd} \cdot (\!|\, z\,;\otimes \,|\!)\\ &\qquad\text{with}\qquad z \equiv (\texttt{true}, \bot)\\ &\qquad\qquad x \otimes (c,e) \equiv \texttt{case}\ c\ \texttt{of}\ \texttt{true}\ \to (\texttt{false}, x)\\ &\qquad\qquad\qquad\qquad\quad \mid \texttt{false} \to \bot\end{aligned}$$

Evaluating the *bottom* symbol $\bot$ yields an error and is our way of modeling the exception we might need to raise. Function `element` interacts with the collection monads list and bag (but not set) in the following ways:

$$\texttt{element} \cdot \texttt{mmap}\ f = f \cdot \texttt{element} \tag{1.5a}$$

$$\texttt{element} \cdot \texttt{unit} = \texttt{id} \tag{1.5b}$$

$$\texttt{element} \cdot \texttt{join} = \texttt{element} \cdot \texttt{element} \tag{1.5c}$$

This characterizes `element` as a *monad morphism* [1.24] from the list and bag monads to the *identity monad* (which is defined through the identity type constructor $\texttt{Id}\ \alpha = \alpha$ plus $\texttt{mmap}\ f\ e = f\ e, \texttt{join} = \texttt{unit} = \texttt{id}$). We can exploit the morphism laws to propagate `element` through the monad operations and implement type conversion pushdown this way. For the example query the rewrite derives the exact simplification we were after:

$$\begin{aligned}&\texttt{element}\ (\texttt{select}\ f\,x\,y\\ &\qquad\qquad \texttt{from}\ xs\ \texttt{as}\ x\texttt{,}\ ys\ \texttt{as}\ y)\\ &= \texttt{element}\ (\!|\, f\,x\,y \mid x \leftarrow xs\texttt{,}\, y \leftarrow ys \,|\!)\\ &= (\texttt{element} \cdot \texttt{join})\ (\!|\!|\, f\,x\,y \mid y \leftarrow ys \,|\!| \mid x \leftarrow xs \,|\!)\\ &= (\texttt{element} \cdot \texttt{join} \cdot \texttt{mmap})\ (\lambda x \to \texttt{mmap}\ (\lambda y \to f\,x\,y)\ ys)\ xs\\ &= (\texttt{element} \cdot \texttt{element} \cdot \texttt{mmap})\ (\lambda x \to \texttt{mmap}\ (\lambda y \to f\,x\,y)\ ys)\ xs\\ &= \texttt{element}\ ((\lambda x \to \texttt{mmap}\ (\lambda y \to f\,x\,y)\ ys)\ (\texttt{element}\ xs))\\ &= (\texttt{element} \cdot \texttt{mmap})\ (\lambda y \to f\ (\texttt{element}\ xs)\,y)\ ys\\ &= (\lambda y \to f\ (\texttt{element}\ xs)\,y)\ (\texttt{element}\ ys)\\ &= f\ (\texttt{element}\ xs)\ (\texttt{element}\ ys)\end{aligned}$$

The morphism laws push the type conversion down as far as possible but not beyond filters since these are mapped into `case` expressions (see Equation 1.3f) for which none of the morphism laws apply.

Early type conversion can indeed save a lot and even reduce the nesting depth of queries. As a another example, consider the following OQL query (note the nesting in the `select` clause):

$$\begin{aligned}&\texttt{element}\,(\texttt{select}\,(\texttt{select}\;f\,x\,y\\&\qquad\qquad\qquad\texttt{from}\;ys\;\texttt{as}\;y)\\&\qquad\quad\texttt{from}\;xs\;\texttt{as}\;x)\\&=\texttt{element}\,(\{\!|\,f\,x\,y\mid y\leftarrow ys\,|\!\}\mid x\leftarrow xs\,|\!\})\end{aligned}$$

Type conversion pushdown converts the above into a query of the form $\{\!|\,f\,(\texttt{element}\,xs)\,y\mid y\leftarrow ys\,|\!\}$ which simply maps $f$ over collection $ys$ instead of creating a nested bag of bags like the original query did.

To wrap up: Wadler [1.24] observed that the action of a monad morphism on a monad comprehension may more concisely described by way of the comprehension syntax itself. Space constraints force us to skip the details here, but the resulting rewriting steps are remarkably simple and thus especially suited for inclusion in a rule-based query optimizer [1.10].

## 1.5 Unraveling Deeply Nested Queries

Comprehensions may be nested within each other and a translator for a source query language that supports nesting can make good use of this: a nested user-level query may be mapped rather straightforwardly into a nested comprehension (see the example query at the end of the last section). However, deriving anything but a nested–loops execution plan from a deeply nested query is a hard task and a widely recognized challenge in the query optimisation community. We are really better off to try to *unnest* a nested query before we process it further.

The monad comprehension calculus provides particularly efficient yet simple hooks to attack this problem:

- Different types of query nesting lead to similar nested forms of monad comprehensions. Rather than to maintain and identify a number of special nesting cases—this route has been taken by numerous approaches, notably Kim's original and followup work [1.15, 1.8] on classifying nested SQL queries—we can concentrate on unnesting the relatively few comprehension forms.
- Much of the unnesting work can, once more, be achieved by application of a small number of syntactic rewriting laws, the *normalisation rules* (1.6a–1.6d below).

The normalisation rules exclusively operate on the monad comprehension syntax level. As before, we use generic monad comprehensions to introduce the rules and you can obtain the specific variants through a consistent replacement of $[\![\,,\,]\!]_n$ by $[\,,\,]$ or $\{\!|\,,\,|\!\}$ or $\{\,,\,\}$, respectively:

$$[\![\,e\mid qs\,,v\leftarrow[\![\,]\!]_2\,,qs'\,]\!]_1=[\![\,]\!]_1 \tag{1.6a}$$

$$[\![\,e\mid qs\,,v\leftarrow[\![\,e'\,]\!]_2\,,qs'\,]\!]_1=[\![\,e[e'/v]\mid qs\,,qs'[e'/v]\,]\!]_1 \tag{1.6b}$$

$$[\![\,e\mid qs\,,v\leftarrow[\![\,e'\mid qs''\,]\!]_2\,,qs'\,]\!]_1=[\![\,e[e'/v]\mid qs\,,qs''\,,qs'[e'/v]\,]\!]_1 \tag{1.6c}$$

$$\{e \mid qs, \texttt{or}\,[\![ e' \mid qs'' ]\!], qs'\} = \{e \mid qs, qs'', e', qs'\} \tag{1.6d}$$

(Expression $e[e'/v]$ denotes $e$ with all free occurrences of $v$ replaced by $e'$.)

The rules form a confluent and terminating set of rewriting rules which is our main incentive to refer to them as *normalisation rules*.

Normalisation gives an unnesting procedure that is *complete* in the sense that an exhaustive application of the rules leads to a query in which all semantically sound unnesting have been performed [1.7]. In the set monad, this may go as far as

$$\{e \mid v_1 \leftarrow e_1, v_2 \leftarrow e_2, \ldots, v_n \leftarrow e_n, p\}$$

with all $e_i$ being atomic expressions with respect to monad comprehension syntax, *i.e.*, the $e_i$ are references to database entry points (relations, class extents) or constants. Nested queries may only occur in the comprehension head $e$ or filter $p$ (to see that we really end up with a single filter $p$, note that we can always "push back" a filter in the qualifier list and that two adjacent filters $p_1, p_2$ may be merged to give $p_1 \wedge p_2$).

Unnesting disentangles queries and makes operands of formerly inner queries accessible in the outer enclosing comprehension. This, in turn, provides new possibilities for further rewritings and optimisations. We will see many applications of unnesting in the sequel.

Comprehension syntax provides a rather poor variety of syntactical forms, but in the early stages of query translation this is more of a virtue than a shortcoming. Monad comprehensions extract and emphasize the *structural* gist of a query rather than to stress the diversity of query constructs. It is this uniformity that facilitates query analysis like the completeness result for comprehension normalisation we have just mentioned. This can lead to new insights and simplifications, which is the next point we make.

In [1.20], Steenhagen, Apers, and Blanken analyzed a class of SQL-like queries which exhibit correlated nesting in the `where`-clause, more specifically:

$$\begin{array}{l}\texttt{select distinct}\ f\ x \\ \quad\texttt{from}\ xs\ \texttt{as}\ x \\ \texttt{where}\ p\ x\ z\end{array} \quad \text{with} \quad z = \begin{pmatrix}\texttt{select}\ g\ x\ y \\ \quad\texttt{from}\ ys\ \texttt{as}\ y \\ \texttt{where}\ q\ x\ y\end{pmatrix}$$

The question is, can queries of this class be rewritten into *flat join queries* of the form

$$\begin{array}{l}\texttt{select distinct}\ f\ x \\ \quad\texttt{from}\ xs\ \texttt{as}\ x, ys\ \texttt{as}\ y \\ \texttt{where}\ q\ x\ y \\ \quad\texttt{and}\ p'\ x\ (g\ x\ y)\end{array}$$

Queries for which such a replacement predicate $p'$ cannot be found have to be processed either (a) using a nested–loops strategy, or (b) by grouping.

Whether we can derive a flat join query is, obviously, dependent on the nature of the yet unspecified predicate $p$.

Steenhagen *et.al.* state the following theorem—reproduced here using our functional language—which provides a partial answer to the question:

> *Whenever $p\ x\ z$ can be rewritten into* $\texttt{or}\ [\![ p'\ x\ v \mid v \leftarrow z ]\!]$ *(i.e., $p$ is an existential quantification w.r.t. some $p'$) the original query may be evaluated by a flat join.*

The monad comprehension normalisation rules provide an elegant proof of this theorem:

$$
\begin{aligned}
&\texttt{select distinct}\ f\ x \\
&\qquad\qquad \texttt{from}\ xs\ \texttt{as}\ x \\
&\qquad\quad \texttt{where}\ p\ x\ z \\
&= \{f\ x \mid x \leftarrow xs, p\ x\ z\} \\
&= \{f\ x \mid x \leftarrow xs, \texttt{or}\ [\![ p'\ x\ v \mid v \leftarrow z ]\!]\} \\
&= \{f\ x \mid x \leftarrow xs, v \leftarrow z, p'\ x\ v\} \\
&= \{f\ x \mid x \leftarrow xs, v \leftarrow [\![ g\ x\ y \mid y \leftarrow ys, q\ x\ y ]\!], p'\ x\ v\} \\
&= \{f\ x \mid x \leftarrow xs, y \leftarrow ys, q\ x\ y, p'\ x\ (g\ x\ y)\}
\end{aligned}
$$

Observe that the normalisation result is the monad comprehension equivalent of the unnested SQL query.

But we can say even more and strengthen the statement of the theorem (thus answering an open question that has been put by Steenhagen *et.al.* in [1.20]):

> *If $p$ is not rewriteable into an existential quantification like above, then we can conclude—based on the completeness of comprehensions normalisation—that unnesting will in fact be impossible.*

Kim's fundamental work [1.15] on the unnesting of SQL queries may largely be understood in terms of normalisation if queries are interpreted in the monad comprehension calculus. We additionally gain insight into questions on the validity of these unnesting strategies in the context of complex data models featuring collection constructors other than the set constructor.

Monad comprehension normalisation readily unnests queries of Kim's *type J*, *i.e.*, SQL queries of the form

$$
\begin{aligned}
Q \quad \equiv \quad & \texttt{select distinct}\ f\ x \\
& \qquad\qquad \texttt{from}\ xs\ \texttt{as}\ x \\
& \qquad\quad \texttt{where}\ p\ x\ \texttt{in}\ (\texttt{select}\ g\ y \\
& \qquad\qquad\qquad\qquad \texttt{from}\ ys\ \texttt{as}\ y \\
& \qquad\qquad\qquad\quad \texttt{where}\ q\ x\ y)
\end{aligned}
$$

Note that predicate $q$ refers to query variable $x$ so that the outer and nested query blocks are correlated. (The SQL predicate `in` is translated into an existential quantifier.) The derivation of the normal form for this query effectively yields Kim's *canonical 2-relation query*:

$$\begin{aligned} Q &= \{f\ x \mid x \leftarrow xs, \mathtt{or}\ [\![p\ x = v \mid v \leftarrow [\![g\ y \mid y \leftarrow ys, q\ x\ y]\!]]\!]\} \\ &= \{f\ x \mid x \leftarrow xs, \mathtt{or}\ [\![p\ x = g\ y \mid y \leftarrow ys, q\ x\ y]\!]\} \\ &= \{f\ x \mid x \leftarrow xs, y \leftarrow ys, q\ x\ y, p\ x = g\ y\} \end{aligned}$$

We can see that Kim's *type J* unnesting is sound only if the outer query block is evaluated in the set monad. No such restriction, though, is necessary for the inner block—an immediate consequence of the well-definedness conditions for monad comprehension coercion (see Section 1.3).

## 1.6 Parallelizing Group-By Queries

The database backends of decision support or data mining systems frequently face SQL queries of the following general type (termed *group queries* in [1.5]):

$$\begin{aligned} Q\ f\ g\ a\ xs \quad \equiv \quad & \mathtt{select}\ f\ x, a\ (g\ x) \\ & \mathtt{from}\ xs\ \mathtt{as}\ x \\ & \mathtt{group\ by}\ f\ x \end{aligned}$$

Group queries extract a particular dimension or feature—described by function $f$—from given base data $xs$ and then pair each data point $f\ x$ in this dimension with aggregated data $a\ (g\ x)$ associated with that point; $a$ may be instantiated by any of the SQL aggregate functions, *e.g.*, `sum` or `max`.

Here is query $Q$ expressed in the monad comprehension calculus (the `group by` introduces nesting in the outer comprehension's head):

$$Q\ f\ g\ a\ xs \quad \equiv \quad \{(f\ x, (agg\ a)\ \{\!|g\ y \mid y \leftarrow xs, f\ y = f\ x|\!\})\mid x \leftarrow xs\}$$

Helper function *agg* translates SQL aggregates into their implementing catamorphisms, *e.g.*, $agg\ \mathtt{sum} = (\!|0;+|\!)$ and $agg\ \mathtt{max} = \mathtt{maximum}$.

We are essentially stuck with the inherent nesting. Normalisation is of no use in this case (the query is in normal form already). Chatziantoniou and Ross [1.5] thus propose to take a different three-step route to process this type of query.

(1) Separate the data points in dimension $f$ of $xs$ in a preprocessing step, *i.e.*, *partition* input $xs$ with respect to $f$.
(2) Evaluate a simplified variant $Q'$ of $Q$ on each partition. In particular, $Q'$ does not need to take care of grouping. Let $ps$ denote one partition of $xs$, then we have

$$Q'\ g\ a\ ps \quad \equiv \quad \texttt{select}\ a\ (g\ x)\ \texttt{from}\ ps\ \texttt{as}\ x$$

or, equivalently,

$$Q'\ g\ a\ ps \quad \equiv \quad (agg\ a)\,\{\!|\, g\ y \mid y \leftarrow ps \,|\!\}$$

(3) Finally, merge the results obtained in step (2) to form the query response.

This strategy clearly shows its benefit in step (2): first, since $xs$ has been split into disjoint partitions during the preprocessing step, we may execute $Q'$ on the different partitions in parallel. Second, there is a chance of processing the $Q'$ in main memory should the partitions $ps$ fit. Measurements reported in [1.5] show the performance gains in terms of time and I/O cost to compensate for the effort spent in the partitioning and joining stages.

In [1.5], classical relational algebra is the target language for the translation of group queries. This choice of query representation introduces subtleties. Relational algebra lacks canonical forms to express the grouping and aggregation found in $Q$. The authors thus propose to understand $Q$ as a *syntactical* query class: the membership of a specific query in this class and thus the applicability of the partitioning strategy is decided by the inspection of the SQL parse tree for that query.

Relational algebra also fails to provide idioms that could express the preprocessing, *i.e.*, partitioning, step of the strategy. To remedy this situation, Chatziantoniou and Ross add attributes to the nodes of query graphs to indicate which partition is represented by a specific node.

Finally, the core stage (2) of the strategy has no equivalent at the target language level as well. Classical relational algebra is unable to express the iteration (or parallel application) inherent to this phase. The authors implement this step *on top* of the relational database backend and thus outside the relational domain.

Facing this mix of query representations (SQL syntax, query graphs, relational algebra, procedural iteration), it is considerably hard to assess the correctness of this parallel processing strategy for query class $Q$.

Reasoning in the monad comprehension calculus can significantly simplify the matter. Once expressed in our functional query representation language, we can construct a correctness proof for the strategy which is basically built from the unfolding of definitions and normalisation steps. Let us proceed by filling the two gaps (partitioning and iteration) that relational algebra has left open.

First, partitioning the base data collection $xs$ with respect to a function $f$ is expressible as follows (note that we require type $\beta$ to allow equality tests):

$$\begin{aligned}&\texttt{partition} :: (\alpha \to \beta) \to [\![\,\alpha\,]\!] \to \{(\beta, [\![\,\alpha\,]\!])\}\\ &\texttt{partition}\ f\ xs \equiv \{(f\ x, [\![\, y \mid y \leftarrow xs, f\ x = f\ y \,]\!]) \mid x \leftarrow xs\}\end{aligned}$$

which builds a set of disjunct partitions such that all elements inside one partition agree on feature $f$ with the latter attached to its associated partition. We have, for example,

$$\texttt{partition odd [1...5]} \quad = \quad \texttt{\{(true,[1,3,5]),(false,[2,4])\}}$$

Second, recall that iteration forms a core building block of our functional language by means of `map`; `map` $f$ also adequately encodes parallel application of $f$ to the elements of its argument. See, for example, the work of Hill [1.13] in which a complete theory of *data-parallel programming* is developed on top of `map`.

With the definition of $Q'$ given earlier, we can compose the phases and express the complete parallel grouping plan as

$$(\texttt{map \{\}}\ (\uparrow)\ (\lambda(z,ps) \rightarrow (z,Q'\ g\ a\ ps)) \cdot \texttt{partition}\ f)\ xs$$

We can now derive a purely calculational proof of the correctness of the parallel grouping idea through a sequence of simple rewriting steps: unfold the definitions of $Q'$, `partition`, and `map`, then apply monad comprehension normalisation to finally obtain $Q\ f\ g\ a\ xs$, the original group query:

$$\begin{array}{ll}
 & (\texttt{map \{\}}\ (\uparrow)\ (\lambda(z,ps) \rightarrow (z,Q'\ g\ a\ ps)) \cdot \texttt{partition}\ f)\ xs \\
\underset{(\cdot)}{=} & (\texttt{map \{\}}\ (\uparrow)\ (\lambda(z,ps) \rightarrow (z,Q'\ g\ a\ ps))\ (\texttt{partition}\ f\ xs) \\
\underset{\texttt{partition}}{=} & (\texttt{map \{\}}\ (\uparrow)\ (\lambda(z,ps) \rightarrow (z,Q'\ g\ a\ ps)) \\
 & \qquad \texttt{\{(}f\ x\texttt{,\{|}y \mid y \leftarrow xs, f\ x = f\ y\texttt{|\})} \mid x \leftarrow xs\texttt{\}} \\
\underset{Q'}{=} & (\texttt{map \{\}}\ (\uparrow)\ (\lambda(z,ps) \rightarrow (z,(agg\ a)\texttt{\{|}g\ y' \mid y' \leftarrow ps\texttt{|\})}) \\
 & \qquad \texttt{\{(}f\ x\texttt{,\{|}y \mid y \leftarrow xs, f\ x = f\ y\texttt{|\})} \mid x \leftarrow xs\texttt{\}} \\
\underset{\texttt{map}}{=} & \texttt{\{}(\lambda(z,ps) \rightarrow (z,(agg\ a)\texttt{\{|}g\ y' \mid y' \leftarrow ps\texttt{|\})}\ v \mid \\
 & \qquad v \leftarrow \texttt{\{(}f\ x\texttt{,\{|}y \mid y \leftarrow xs, f\ x = f\ y\texttt{|\})} \mid x \leftarrow xs\texttt{\}\}} \\
\underset{1.6c}{=} & \texttt{\{(}f\ x\texttt{,(}agg\ a\texttt{)\{|}g\ y' \mid y' \leftarrow \texttt{\{|}y \mid y \leftarrow xs, f\ x = f\ y\texttt{|\}|\})} \mid x \leftarrow xs\texttt{\}} \\
\underset{1.6c}{=} & \texttt{\{(}f\ x\texttt{,(}agg\ a\texttt{)\{|}g\ y \mid y \leftarrow xs, f\ x = f\ y\texttt{|\})} \mid x \leftarrow xs\texttt{\}} \\
= & Q\ f\ g\ a\ xs\ .
\end{array}$$

## 1.7 A Purely Functional View of XPath

Monad comprehensions can serve as an effective "semantical backend" for other than SQL-style languages. To make this point and to conclude the chapter let us take a closer look at how monad comprehensions can provide a useful interpretation of XPath path expressions [1.1].

XML syntax provides an unlimited number of *tree dialects*: data (*document content*) is structured using properly nested opening <*t*> and matching closing tags </*t*>.

XPath provides operators to describe *path traversals* over such tree-shaped data structures. Starting from a *context node*, an XPath path expression traverses its input document via a sequence of *steps*. A step's *axis* (*e.g.*, `ancestor`, `descendant`, with the obvious semantics) indicates which tree nodes are reachable from the context node, a step's *node test* $::t$ filters these nodes to retain those with tag name $t$ only[3]. These new nodes are then interpreted as context nodes for subsequent steps, and so forth.

In XPath syntax, the steps of a path $p$ are syntactically separated by slashes `/`; a path originating in the document's root node starts with a leading slash: $/p$. In addition to node tests, XPath provides path predicates $q$ which may be evaluated against $p$'s set of result nodes: $p$`[`$q$`]`. Predicates have existential semantics: a node $c$ qualifies if path $q$ starting from context node $c$ evaluates to a non-empty set of nodes.

We can capture the XPath semantics by a translation function `xpath` $p\ c$ which yields a monad comprehension that computes the node set returned by path $p$ starting from context node $c$. Function `xpath` is defined by structural recursion over the XPath syntax:

$$
\begin{array}{lcl}
\texttt{xpath}\ (/p)\ c & \equiv & \texttt{xpath}\ p\ (\texttt{root}\ c) \\
\texttt{xpath}\ (p_1/p_2)\ c & \equiv & \texttt{\{}n' \mid n \leftarrow \texttt{xpath}\ p_1\ c\texttt{,}n' \leftarrow \texttt{xpath}\ p_2\ n\texttt{\}} \\
\texttt{xpath}\ (p\texttt{[}q\texttt{]})\ c & \equiv & \texttt{\{}n \mid n \leftarrow \texttt{xpath}\ p\ c\texttt{,or \{true} \mid n' \leftarrow \texttt{xpath}\ q\ n\texttt{\}\}} \\
\texttt{xpath}\ (a\texttt{::}t)\ c & \equiv & \texttt{step}\ (a\texttt{::}t)\ c
\end{array}
$$

The primitive `root` $c$ evaluates to the root of the document that includes node $c$. Function `step` does the actual evaluation of a step from a given context node. We will shortly come back to its implementation.

As given, function `xpath` fails to reflect one important detail of XPath: nodes resulting from path evaluation are returned in *document order*. The XML document order $\ll$ orders the nodes of a document according to a preorder traversal of the document tree. A complete XPath translation would thus read (`sidoaed` $\cdot$ `xpath`) $p\ c$ where `sidoaed`[4] (*sort in document order and eliminate duplicates*) orders a node set according to $\ll$.

Note that `sidoaed` is a catamorphism itself. Let `iidoaed (`$n$`,`$ns$`)` (*insert in document order and eliminate duplicates*) denote the function that inserts node $n$ into node list $ns$ with respect to $\ll$ if $n$ is not an element of $ns$ (by straightforward recursion over $ns$). We then have

```
sidoaed :: [[𝕏]] → [𝕏]
sidoaed ≡ (|[];iidoaed|)
```

with $\mathbb{X}$ being the implementation type for XML nodes (see below). Note that `sidoaed` is well-defined over any collection type since `iidoaed` is left-idempotent and left-commutative.

[3] For brevity, we omit XPath features like the `*`, `node()`, or `text()` node tests.

[4] The particular name `sidoaed` has been borrowed from an XQuery tutorial by Peter Fankhauser and Phil Wadler [1.6].

(We could even go a step further and integrate document order more tightly into our model. To this end, observe that

$$\begin{aligned} \texttt{zero} &\equiv [] \\ \texttt{unit}\ n &\equiv [n] \\ \texttt{mmap} &\equiv \texttt{map}\ [] \ (\uparrow) \\ \texttt{join} &\equiv (\!|\, [] \,;\oplus \,|\!) \qquad \text{with } xs \oplus ys \equiv (\!|\, ys \,;\texttt{iidoaed}\,|\!)\ xs \end{aligned}$$

yields a monad of *node sequences in document order* and its associated notion of node sequence comprehensions—see Figure 1.4.)

To illustrate, the XPath expression `/child::a[child::b]` is translated as follows (where $c$ denotes the context node):

$$\begin{aligned} &\texttt{sidoaed}\ (\texttt{xpath}\ (\texttt{/child::a[child::b]})\ c) \\ &= \texttt{sidoaed}\ (\texttt{xpath}\ (\texttt{child::a[child::b]})\ (\texttt{root}\ c)) \\ &= \texttt{sidoaed}\ (\{n \mid n \leftarrow \texttt{xpath}\ (\texttt{child::a})\ (\texttt{root}\ c), \\ &\qquad\qquad \texttt{or}\ \{\texttt{true} \mid n' \leftarrow \texttt{xpath}\ (\texttt{child::b})\ n\}\}) \\ &= \texttt{sidoaed}\ (\{n \mid n \leftarrow \texttt{step}\ (\texttt{child::a})\ (\texttt{root}\ c), n' \leftarrow \texttt{step}\ (\texttt{child::b})\ n\}) \end{aligned}$$

Note how the second step depends on the context nodes $n$ computed in the first step.

Thanks to the comprehension semantics for path expressions we are in a position to find concise proofs for a number of useful XPath equivalences. As an example, consider *predicate flattening*:

$$\begin{aligned} &\texttt{xpath}\ (p[p_1[p_2]])\ c \\ &= \{n \mid n \leftarrow \texttt{xpath}\ p\ c, \texttt{or}\ \{\texttt{true} \mid n' \leftarrow \texttt{xpath}\ (p_1[p_2])\ n\}\} \\ &= \{n \mid n \leftarrow \texttt{xpath}\ p\ c, \texttt{or}\ \{\texttt{true} \mid n' \leftarrow \{v \mid v \leftarrow \texttt{xpath}\ p_1\ n, \\ &\qquad\qquad\qquad\qquad\qquad\qquad \texttt{or}\ \{\texttt{true} \mid v' \leftarrow \texttt{xpath}\ p_2\ v\}\}\}\} \\ &= \{n \mid n \leftarrow \texttt{xpath}\ p\ c, \texttt{or}\ \{\texttt{true} \mid n' \leftarrow \{v \mid v \leftarrow \texttt{xpath}\ p_1\ n, v' \leftarrow \texttt{xpath}\ p_2\ v\}\}\} \\ &= \{n \mid n \leftarrow \texttt{xpath}\ p\ c, \texttt{or}\ \{\texttt{true} \mid n' \leftarrow \texttt{xpath}\ (p_1/p_2)\ n\}\} \\ &= \texttt{xpath}\ (p[p_1/p_2])\ c \end{aligned}$$

The more explicit we are in explaining the XPath semantics, the more opportunities for optimisation are created. Since XPath axes are defined with respect to document order and tag inclusion, let us make these notions explicit.

We choose a specific implementation type for an XML node, namely $\mathbb{X} = (\mathbb{S},(\mathbb{N},\mathbb{N}))$. While the first component will store the node's tag name as a string, the pair of numbers represents its *preorder* and *postorder* traversal rank, respectively. The ranks are sufficient to encode document order as well as to characterize the major XPath axes [1.11]. Figure 1.5 displays an XML document instance, its associated document tree as well as its internal

representation, the set `doc` of $\mathbb{X}$ values. Intuitively, the preorder rank of a node represents the position of its opening tag relative to the positions of the opening tags of all other nodes in the document. An equivalent observation applies to the postorder rank and the node's closing tag. Obviously, then,

$$v' \text{ is a descendant of } v$$
$$\Leftrightarrow$$
$$\texttt{pre}\ v < \texttt{pre}\ v' \wedge \texttt{post}\ v' < \texttt{post}\ v\ ,$$

*i.e.*, the tags of $v$ embrace those of $v'$. The other major XPath axes `ancestor`, `preceding`, and `following` may be understood in terms of preorder and postorder ranks, too.

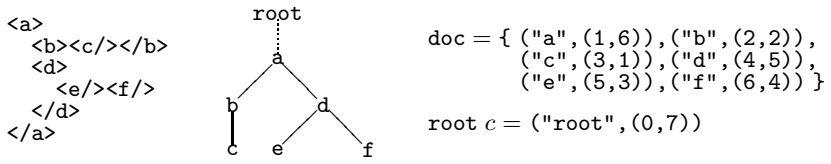



**Fig. 1.5.** XML Document and its *Preorder*/*Postorder* Encoding

Given the following function definitions:

$$\begin{aligned}
\texttt{tag}\ (t,(pre,post)) &\equiv t\\
\texttt{pre}\ (t,(pre,post)) &\equiv pre\\
\texttt{post}\ (t,(pre,post)) &\equiv post\\
n_1 \ll n_2 &\equiv (\texttt{pre}\ n_1) < (\texttt{pre}\ n_2)
\end{aligned}$$

we can encode XPath step evaluation as follows:

$$\begin{aligned}
&\texttt{step}\ (\texttt{descendant::}t)\ c \equiv \{n \mid n \leftarrow \texttt{doc}, c \ll n, \texttt{post}\ n < \texttt{post}\ c, \texttt{tag}\ n = t\}\\
&\texttt{step}\ (\texttt{following::}t)\ c \equiv \{n \mid n \leftarrow \texttt{doc}, c \ll n, \texttt{post}\ c < \texttt{post}\ n, \texttt{tag}\ n = t\}\\
&\texttt{step}\ (\texttt{preceding::}t)\ c \equiv \{n \mid n \leftarrow \texttt{doc}, n \ll c, \texttt{post}\ c < \texttt{post}\ n, \texttt{tag}\ n = t\}\\
&\texttt{step}\ (\texttt{ancestor::}t)\ c \equiv \{n \mid n \leftarrow \texttt{doc}, n \ll c, \texttt{post}\ n < \texttt{post}\ c, \texttt{tag}\ n = t\}
\end{aligned}$$

Now, given the XML instance of Figure 1.5, it is easy to verify that our monad comprehension semantics and the XPath semantics are indeed the same. We have, for example:

$$\texttt{xpath}\ (\texttt{/descendant::d[preceding::b]})\ c \quad = \quad \{(\texttt{"d"},(4,5))\}$$

Note that the choice of context node $c$ is immaterial here since the path expression is absolute, effectively having the document root node as the context node.

If you look at the definitions for the `preceding` and `ancestor` axes you will notice that both axes select nodes $n$ that are *before* context node $c$ in document order. Axes of this kind are referred to as *reverse axes*.

Reverse axes pose a problem for so-called *streaming XPath processors*. XPath engines of this type try to perform a single preorder traversal (*e.g.*, by receiving the events of a SAX parser) over the input document to evaluate a given path expression. The big win is that only very limited memory space is necessary to perform the evaluation: a streaming XPath processor can, in principle, operate on XML documents of arbitrary size.

To evaluate a reverse axis step in such a setup is problematic because the XPath processor would need temporary space to remember *past* SAX events. To restore the modest memory requirements we thus need to get rid of the reverse axes. Such an approach is indeed possible and discussed in [1.18]. The authors present a number of XPath equivalences, *e.g.*,

$$\texttt{/descendant::}t\texttt{/preceding::}t' \;=\; \texttt{/descendant::}t'\texttt{[following::}t\texttt{]}$$

(note that the righthand side trades a reverse axis for a forward axis and a step for a predicate, respectively).

A proof for this equality naturally depends on the path expression semantics as well as the semantics of the XPath axes themselves. As we have defined both semantics in terms of monad comprehensions, we can carry out the actual proof solely by means of equational reasoning, which is typical for a purely functional query representation. We first map the righthand side XPath expression into its monad comprehension equivalent and then exhaustively apply the monad comprehension normalisation rules 1.6a–1.6d. For our current example, the normal form is reached after two normalisation steps (see below). Applied to the lefthand side of the above equation, mapping and normalisation (not shown here) yields an identical monad comprehension, which validates the equality.

$$
\begin{aligned}
&\texttt{xpath}\ (\texttt{/descendant::}t'\texttt{[following::}t\texttt{]})\ c\\
&\underset{\texttt{xpath}}{=}\ \texttt{xpath}\ (\texttt{descendant::}t'\texttt{[following::}t\texttt{]})\ (\texttt{root}\ c)\\
&\underset{\texttt{xpath}}{=}\ \{n \mid n \leftarrow \texttt{xpath}\ (\texttt{descendant::}t')\ (\texttt{root}\ c),\\
&\qquad\qquad \texttt{or}\ \{\texttt{true} \mid n' \leftarrow \texttt{xpath}\ (\texttt{following::}t)\ n\}\}\\
&\underset{\texttt{xpath}}{=}\ \{n \mid n \leftarrow \texttt{step}\ (\texttt{descendant::}t')\ (\texttt{root}\ c),\\
&\qquad\qquad \texttt{or}\ \{\texttt{true} \mid n' \leftarrow \texttt{step}\ (\texttt{following::}t)\ n\}\}\\
&\underset{\texttt{step}}{=}\ \{n \mid n \leftarrow \{v \mid v \leftarrow \texttt{doc}, (\texttt{root}\ c) \ll v, \texttt{post}\ v < \texttt{post}\ (\texttt{root}\ c), \texttt{tag}\ v = t'\},\\
&\qquad\qquad \texttt{or}\ \{\texttt{true} \mid n' \leftarrow \{v' \mid v' \leftarrow \texttt{doc}, n \ll v', \texttt{post}\ n < \texttt{post}\ v', \texttt{tag}\ v' = t\}\}\}\\
&\underset{1.6c}{=}\ \{v \mid v \leftarrow \texttt{doc}, (\texttt{root}\ c) \ll v, \texttt{post}\ v < \texttt{post}\ (\texttt{root}\ c), \texttt{tag}\ v = t',\\
&\qquad\qquad \texttt{or}\ \{\texttt{true} \mid v' \leftarrow \texttt{doc}, v \ll v', \texttt{post}\ v < \texttt{post}\ v', \texttt{tag}\ v' = t\}\}\\
&\underset{1.6d}{=}\ \{v \mid v \leftarrow \texttt{doc}, (\texttt{root}\ c) \ll v, \texttt{post}\ v < \texttt{post}\ (\texttt{root}\ c), \texttt{tag}\ v = t',\\
&\qquad\qquad v' \leftarrow \texttt{doc}, v \ll v', \texttt{post}\ v < \texttt{post}\ v', \texttt{tag}\ v' = t\}
\end{aligned}
$$

$$= \{v \mid v \leftarrow \texttt{doc}, \texttt{tag}\ v = t', v' \leftarrow \texttt{doc}, v \ll v', \texttt{post}\ v < \texttt{post}\ v', \texttt{tag}\ v' = t\}$$

To understand the last rewriting step above, note that $(\texttt{root}\ c) \ll v$ and $\texttt{post}\ v < \texttt{post}\ (\texttt{root}\ c)$ for arbitrary nodes $c, v$ of the same document (also see Figure 1.5).

We observe that the resulting normalised monad comprehension describes the same computation as the following SQL query:

```
select v
  from doc v, doc v'
 where tag v = t' and tag v' = t
   and v ≪ v' and post v < post v'
```

More generally, an XPath expression consisting of $n$ steps or predicates yields an $n$-ary join of the relation `doc` of $\mathbb{X}$ values with itself. The structural aspects of a path expression, implicitly given by the XPath axes, as well as name tests are mapped into a simple conjunctive predicate against this intermediary $n$-ary self-join result.

Although this XPath evaluation scheme may appear rather simplistic, it offers a number of—sometimes non-obvious—optimization hooks, especially if the scheme is used in a set-oriented manner [1.11] i.e. when a path expression is evaluated for a *context node set*, not just a single context node $c$ as discussed here.

## 1.8 Conclusion

In this chapter we have used monads in the role that sets play in the relational calculus. A feature of the monad notion is that it comes with just enough internal structure that is needed to interpret a query calculus. The resulting monad comprehension calculus is limitted with respect to the variety of syntactic forms it offers but this ultimately leads to a form of query representation that stresses the core structure inherent to a query.

We have seen that a monad comprehension $[\![ f\ x \mid x \leftarrow xs ]\!]$ can describe a variety of query constructs, *e.g.*, parallel application of $f$ to the elements of $xs$, iteration, duplicate elimination, aggregation, or a quantifier ranging over $xs$, depending on the actual choice of monad we are evaluating the comprehension in. This uniformity has enabled us to spot useful and sometimes unexpected dualities between query constructs, *e.g.*, the close connection of the class of flat join queries and existential quantification discussed in Section 1.5.

The terseness of the calculus additionally has a positive impact on the size of the rule sets necessary to express complex query rewrites, most notably monad comprehension normalisation.

This chapter has aimed to show that monad comprehensions provide an ideal framework in which the interaction of a diversity of query representation and optimisation techniques may be studied. We have found this purely functional representation of queries based on catamorphisms and monads to cover, simplify, and generalize many of the proposed views of classical database query languages as well as the more recent XML languages such as like XPath.

## References


1.1 Anders Berglund, Scott Boag, Don Chamberlin, Mary F. Fernandez, Michael Kay, Jonathan Robie, and Jérôme Siméon. XML Path Language (XPath) 2.0. Technical Report W3C Working Draft, Version 2.0, World Wide Web Consortium, December 2001. `http://www.w3.org/TR/xpath20/`.

1.2 Val Breazu-Tannen, Peter Buneman, and Limsoon Wong. Naturally Embedded Query Languages. In *Proc. of the Int'l Conference on Database Theory (ICDT)*, pages 140–154, Berlin, Germany, October 1992.

1.3 Peter Buneman, Leonid Libkin, Dan Suciu, Val Tannen, and Limsoon Wong. Comprehension Syntax. *ACM SIGMOD Record*, 23:87–96, March 1994.

1.4 Rick G. Cattell and Douglas K. Barry, editors. *The Object Database Standard: ODMG 2.0*. Morgan Kaufmann Publishers, San Francisco, California, 1997. Release 2.0.

1.5 Damianos Chatziantoniou and Kenneth A. Ross. Groupwise Processing of Relational Queries. In *Proc. of the 23rd Int'l Conference on Very Large Data Bases (VLDB)*, pages 476–485, Athens, Greece, August 1997.

1.6 Peter Fankhauser and Philip Wadler. XQuery Tutorial. *XML 2001*, Orlando, USA, December 2001.

1.7 Leonidas Fegaras and David Maier. Optimizing Object Queries Using an Effective Calculus. *ACM Transactions on Database Systems (TODS)*, 25(4):457–516, 2000.

1.8 Richard A. Ganski and Harry K. T. Wong. Optimization of Nested SQL Queries Revisited. In *Proc. of the ACM SIGMOD Int'l Conference on Management of Data*, pages 23–33, San Francisco, USA, 1987.

1.9 Andrew J. Gill, John Launchbury, and Simon L. Peyton Jones. A Short Cut to Deforestation. In *Proc. of the ACM Conference on Functional Programming and Computer Architecture (FPCA)*, pages 223–232, Copenhagen, Denmark, April 1993.

1.10 Torsten Grust. *Comprehending Queries*. PhD thesis, University of Konstanz, September 1999. Available at `http://www.ub.uni-konstanz.de/kops/volltexte/1999/312/312_1.pdf`.

1.11 Torsten Grust. Accelerating XPath Location Steps. In *Proc. of the 21st Int'l ACM SIGMOD Conference on Management of Data*, pages 109–120, Madison, Wisconsin, USA, June 2002.

1.12 Torsten Grust and Marc H. Scholl. How to Comprehend Queries Functionally. *Journal of Intelligent Information Systems*, 12(2/3):191–218, March 1999. Special Issue on Functional Approach to Intelligent Information Systems.

1.13 Jonathan M.D. Hill. *Data-Parallel Lazy Functional Programming*. PhD thesis, University of London, Queen Mary and Westfield College, September 1994.

1.14 John Hughes and Simon L. Peyton Jones (editors). Haskell 98: A Non-strict, Purely Functional Language. `http://haskell.org/definition/`, February 1999.
1.15 Won Kim. On Optimizing an SQL-like Nested Query. *ACM Transactions on Database Systems*, 7(3):443–469, September 1982.
1.16 Joachim Lambek. A Fixpoint Theorem for Complete Categories. *Mathematische Zeitschrift*, 103:151–161, 1968.
1.17 Erik Meijer, Marten M. Fokkinga, and Ross Paterson. Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire. In *Proc. of the ACM Conference on Functional Programming and Computer Architecture (FPCA)*, number 523 in Lecture Notes in Computer Science (LNCS), pages 124–144, Cambridge, USA, 1991. Springer Verlag.
1.18 Dan Olteanu, Holger Meuss, Tim Furche, and François Bry. Symmetry in XPath. Technical Report PMS-FB-2001-16, Institute of Computer Science, University of Munich, Germany, October 2001.
1.19 Simon Peyton-Jones. Tackling the Awkward Squad: Monadic Input/Output, Concurrency, Exceptions, and Foreign-Language Calls in Haskell. In Tony Hoare, Manfred Broy, and Ralf Steinbruggen, editors, *Engineering Theories of Software Construction*, pages 47–96. IOS Press, 2001.
1.20 Hennie J. Steenhagen, Peter M.G. Apers, and Henk M. Blanken. Optimization of Nested Queries in a Complex Object Model. In *Proc. of the 4th Int'l Conference on Extending Database Technology (EDBT)*, pages 337–350, Cambridge, UK, March 1994.
1.21 Dan Suciu and Limsoon Wong. On Two Forms of Strutural Recursion. In Georg Gottlob and Moshe Y. Vardi, editors, *Proc. of the 5th Int'l Conference on Database Theory (ICDT)*, number 893 in Lecture Notes in Computer Science (LNCS), pages 111–124, Prague, Czech Republic, January 1995. Springer Verlag.
1.22 Akihiko Takano and Erik Meijer. Shortcut Deforestation in Calculational Form. In *Proc. of the ACM Conference on Functional Programming and Computer Architecture (FPCA)*, pages 306–313, La Jolla, USA, June 1995. ACM Press.
1.23 Philip Wadler. Theorems for Free! In *Proc. of the 4th Int'l Conference on Functional Programming and Computer Architecture (FPCA)*, London, England, September 1989.
1.24 Philip Wadler. Comprehending Monads. In *Conference on Lisp and Functional Programming*, pages 61–78, June 1990.
1.25 Limsoon Wong. *Querying Nested Collections*. PhD thesis, University of Pennsylvania, Philadelphia, August 1994.