Comprehensive Comprehensions

Comprehensions with ‘Order by’ and ‘Group by’

Philip Wadler

University of Edinburgh

Abstract

We propose an extension to list comprehensions that makes it easy to express the kind of queries one would write in SQL using `ORDER BY`, `GROUP BY`, and `LIMIT`. Our extension adds expressive power to comprehensions, and generalises the SQL constructs that inspired it. Moreover, it is easy to implement, using simple desugaring rules.

1. Introduction

List comprehensions are a popular programming language feature. Originally introduced in NPL [Dar77], they have made their way into Miranda, Haskell, Erlang, Python, and Scala, among other languages.

It is well known that list comprehensions have much in common with database queries [TW89], but they are significantly less powerful. For example, consider this SQL query

```
SELECT dept, SUM(salary)
FROM employees
GROUP BY dept
ORDER BY SUM(salary) DESCENDING
LIMIT 5
```

The `GROUP BY` clause groups records together; the `ORDER BY` sorts the departments in order of salary bill; and the `LIMIT` clause picks just the first five records. This support for grouping and sorting is extremely useful in practice, but is not available in list comprehensions.

In this paper we propose an extension to list comprehensions that makes it easy to express the kind of queries one would write in SQL using `ORDER BY`, `GROUP BY`, and `LIMIT`. Here, for example, is how the above SQL query would be rendered in our extension.

```
[ (the dept, sum salary)
| (name, dept, salary) <- employees
, group by dept
, order by Down (sum salary)
, order using take 5 ].
```

Moreover, our extensions are significantly more general than SQL's facilities. We make the following contributions.

Simon Peyton Jones
Microsoft Research

- We introduce two new qualifiers for list comprehensions, `order` and `group` (Section 3). Unusually, `group` redefines the value and type of bound variables, replacing each bound variable by a list of grouped values. Unlike other approaches to grouping (as found in Kleisli, XQuery, or LINQ), this makes it easy to aggregate groups without nesting comprehensions.
- Rather than having fixed sorting and grouping functions, both

`order by` and `group by` are generalised by an optional `using` clause that accept any function of types

$$\forall a.(a \to \tau) \to [a] \to [a]$$
$$\forall a.(a \to \tau) \to [a] \to [[a]]$$

respectively (Sections 3.2 and 3.5). Polymorphism elegantly guarantees that the semantics of the construct is independent of the particulars of how comprehensions are compiled.

- We present the syntax, typing rules, and formal semantics of our extensions, explaining the role of parametricity (Section 4). Our semantics naturally accommodates the zip comprehensions that are implemented in Hugs and GHC (Section 3.8).
- We show that the extended comprehensions satisfy the usual comprehension laws, plus some new specific laws (Section 5).

Other database languages, such as LINQ and XQuery, have similar constructs, as we discuss in Section 7. However, we believe that no other language contains the same general constructs.

2. The problem we address

List comprehensions are closely related to relational calculus and SQL [TW89]. Database languages based on comprehensions include CPL [BLS+94], Kleisli [Won00], Links [CLWY06], and the LINQ features of C# and Visual Basic [MBB06]. XQuery, a query language for XML, is also based on a comprehension notation, called FLWOR expressions [BCF+07]. Kleisli, Links, and LINQ provide comprehensions as a flexible way to query databases, compiling as much of the comprehension as possible into efficient SQL;

and LINQ can also compile comprehensions into XQuery.

Many SQL queries can be translated into list comprehensions straightforwardly. For example, in SQL, we can find the name and salary of all employees that earn more than 50K as follows.

```
SELECT name, salary
FROM employees
WHERE salary > 50
```

As a list comprehension in Haskell, assuming tables are represented by lists of tuples, this looks very similar:

```
[ (name, salary)
| (name, salary, dept) <- employees
, salary > 50 ]
```


Here we assume that `employees` is a list of tuples giving name, salary, and department name for each employee.

While translating `SELECT-FROM-WHERE` queries of SQL into list comprehensions is straightforward, translating other features, including `ORDER BY` and `GROUP BY` clauses, is harder. For example, here is an SQL query that finds all employees paid less than 50K, ordered by salary with the least-paid employee first.

```
SELECT name
FROM employees
WHERE salary < 50
ORDER BY salary
```

The equivalent in Haskell would be written as follows.

```
map (\(name,salary) -> name)
  (sortWith (\(name,salary) -> salary)
    [ (name,salary)
    | (name, salary, dept) <- employees
    , salary < 50 ])
```

Since we cannot sort within a list comprehension, we do part of the job in a list comprehension (filtering, and picking just the name and salary fields), before reverting to conventional Haskell functions to first sort, and then project out the name field from the sorted result. The function `sortWith` is defined as follows:

```
sortWith :: Ord b => (a -> b) -> [a] -> [a]
sortWith f = sortBy (\ x y ->
                     compare (f x) (f y))
```

It takes a comparison-key extractor function `f`, which extracts from each input record the key to be used as a basis for sorting. The function `sortBy` is part of the Haskell Prelude, and has type

```
sortBy :: (a -> a -> Bool) -> [a] -> [a]
```

It is given the function to use when comparing two elements of the input list.

Translating `GROUP BY` is trickier. Here is an SQL query that returns a table showing the total salary for each department.

```
SELECT dept, sum(salary)
FROM employees
GROUP BY dept
```

An equivalent in Haskell is rather messy:

```
let
 depts =
   nub [ dept
       | (name,dept,salary) <- employees ]
in
 [ (dept,
    sum [ salary
        | (name,dept',salary) <- employees
        , dept == dept'])
 | dept <- depts ]
```

This uses the library function

```
nub :: Eq a => [a] -> [a]
```

which removes duplicates in a list. Not only is the code hard to read, but it is inefficient too: the `employees` list is traversed once to extract the list of departments, and then once for each department to find that department's salary bill. There are other ways to write this in Haskell, some with improved efficiency but greater complexity. None rivals the corresponding SQL for directness and clarity.

It is tantalising that list comprehensions offer a notation that is compact and powerful – and yet fails to match SQL. Furthermore, `ORDER BY` and `GROUP BY` are not peripheral parts of SQL: they are both heavily used.

Thus motivated, we propose some modest extensions to the list-comprehension notation that allows such SQL queries to be expressed neatly. For example, the two queries above can be ex-

pressed using our extensions like this:

```
[ name
| (name, salary, dept) <- employees
, salary < 50
, order by salary ]

[ (the dept, sum salary)
| (name, salary, dept) <- employees
, group by dept ]
```

Our extensions are modest in the sense that they can be explained in the same way as before, by a simple desugaring translation. Furthermore, they embody some useful generalisations that are not available in SQL.

3. The proposal by example

We now explain our proposal in detail, using a sequence of examples, starting with `order by` and moving on to `group by`. We use informal language, but everything we describe is made precise in Section 4. To avoid confusion we concentrate on one particular set of design choices, but we have considered other variants, as we discuss in Section 6.

We will use a table listing the name, department, and salary of employees as a running example.

```
employees :: [(Name, Dept, Salary)]
employees =  [ ("Simon",  "MS",   80)
             , ("Erik",   "MS",   100)
             , ("Phil",   "Ed",   40)
             , ("Gordon", "Ed",   45)
```

```
, ("Paul",   "Yale", 60)]
```

3.1 Order by

The SQL query

```
SELECT name, salary
FROM employees
ORDER BY salary
```

is expressed by the following comprehension

```
[ (name, salary)
| (name, dept, salary) <- employees
, order by salary ]
```

which returns

```
[ ("Phil",   40)
, ("Gordon", 45)
, ("Paul",   60)
, ("Simon",  80)
, ("Erik",   100)]
```

The sort key (written after the keyword `by`) is an arbitrary Haskell expression, not just a simple variable. Here, for example, is a rather silly comprehension, which sorts people by the product of their salary and the length of their name:

```
[ (name, salary)
| (name, dept, salary) <- employees
, order by salary * length name ]
```

However, this generality has more than frivolous uses. We can

readily sort by multiple keys, simply by sorting on a tuple:

```
[ (name, salary)
| (name, dept, salary) <- employees
```



```
, order by (salary, name) ]
```

But suppose we want the *highest* salary first? SQL uses an additional keyword, `DESCENDING`:

```
SELECT name, salary
FROM employees
ORDER BY salary DESCENDING name ASCENDING
```

We can use the power of Haskell to express this, simply by using a different key extractor:

```
[ (name, salary)
| (name, dept, salary) <- employees
, order by (Down salary, name) ]
```

where `Down` is elegantly defined thus:

```
newtype Down a = Down a deriving( Eq )
instance Ord a => Ord (Down a) where
  compare (Down x) (Down y) = y `compare` x
```

Since `Down` is a `newtype`, it carries no runtime overhead; it simply tells Haskell how to build the ordering dictionary that is passed to the sorting function.

3.2 User-defined ordering

Another useful way to generalise `order` is by allowing the user to provide the sorting function. For example, she may know a particularly efficient way to sort the records — perhaps these particular records have an integer index, so that radix sort is available — or perhaps she wants a non-lexicographic comparison method for people's names. We therefore generalise `order` to take an (optional) user-defined function:

```
[ (name, salary)
| (name, dept, salary) <- employees
, order by name using strangeSort ]
```

If `strangeSort` sorts on the second letter of the person's name we would get

```
[ ("Paul",   60)
, ("Phil",   40)
, ("Simon",  80)
, ("Gordon", 45)
, ("Erik",   100) ]
```

Here "`using`" is a new keyword that allows the user to supply the function used for ordering the results:

```
strangeSort :: (a -> String) -> [a] -> [a]
```

Omitting the "`using` f" clause, as we did in the previous section, is equivalent to writing "`using sortWith`" (a function introduced in Section 2).

Furthermore, there is nothing that requires that the user-supplied function should do *sorting*! Suppose, for example, that we want to

extract all employees with a salary greater than 70, highest salary first. In SQL, we could do so as follows:

```
SELECT name, salary
FROM employees
WHERE salary > 70
ORDER BY salary DESCENDING
```

This translates to the comprehension

```
[ (name, salary)
| (name, dept, salary) <- employees
, salary > 70
, order by Down salary ]
```

which returns


```
[ ("Erik",   100)
, ("Simon",  80) ]
```

However, we might want to write this more efficiently, first sort the list and then only take elements while the salary is above the limit.

```
[ (name, salary)
| (name, dept, salary) <- employees
, order by Down salary
, order by salary > 70 using takeWhile ]
```

This uses the standard library function to extract the initial segment of a list satisfying a predicate.

```
takeWhile :: (a -> Bool) -> [a] -> [a]
```

In general, we can write

$$\texttt{order by}\ e\ \texttt{using}\ f$$

whenever e has type τ and f has type

$$\forall a.\ (a \to \tau) \to [a] \to [a].$$

We require f to be polymorphic in the element type a, which guarantees that it gives uniform results regardless of the type of tuple we present, but we do not require it to be polymorphic in the comparison-key type τ. Intuitively, the user-supplied function will be given a list of records whose exact shape (how many fields, laid out how) is a matter for the desugaring transformation. So the desugaring transform supplies the function f with a comparison-key extraction function, which f in turn uses to extract a comparison key from each record. This key has a type τ fixed by the sorting function (not the desugaring transform). We return to the question of polymorphism in Section 4.4.

3.3 Dropping the `by` clause in ordering

The ability to process the record stream with a user-defined function, rather than with a fixed set of functions (sort ascending, sort descending, etc) is a powerful generalisation that takes us well beyond SQL. Indeed, another apparently-unrelated SQL construct, `LIMIT`, turns out to be expressible using `order using`. Suppose we want to find the three employees with the highest salary. In SQL, we would use the `LIMIT` notation:

```
SELECT name, salary
FROM employees
```

```
ORDER BY salary DESCENDING
LIMIT 3
```

We can do this using a trivial variant of `order` that drops the "`by`" clause:

```
[ (name, salary)
| (name, dept, salary) <- employees
, order by Down salary
, order using take 3 ]
```

which returns

```
[ ("Erik",   100),
  ("Simon",  80),
  ("Paul",   60)]
```

The effect of omitting the `by` clause is simply that the supplied function is used directly without being applied to a key-extractor function.

As a second (contrived) example, we could sort into descending salary order by first sorting into ascending order and then reversing the list:

```
[ (name, salary)
| (name, dept, salary) <- employees
```



```
, order by salary
, order using reverse ]
```

In general, we can write

$$\texttt{order using}\ f$$

whenever f is an arbitrary Haskell expression with type

$$\forall a.\ [a] \to [a].$$

Again, we require f to be polymorphic in the element type `a`. However, omitting "`by`" is mere convenience, since

$$\texttt{order using}\ f \quad \equiv \quad \texttt{order by ()}\ \texttt{using}\ \lambda x.\ f$$

where x does not appear in f.

3.4 Group by

Having described how `order by` works, we now move on to `group by`. As an example, the SQL query

```
SELECT dept, SUM(salary)
FROM employees
GROUP BY dept
```

translates to the comprehension

```
[ (the dept, sum salary)
| (name, dept, salary) <- employees
, group by dept ]
```

which returns

```
[("MS", 180), ("Ed", 85), ("Yale", 60)]
```

The only new keywords in this comprehension are `group by`. Both `the` and `sum` are ordinary Haskell functions. The Big Thing to notice is that `group by` has changed the type of all the variables in

scope: before the `group by` each tuple contains a name, a department and a salary, while after each tuple contains a *list* of names, a *list* of departments, and a *list* of salaries! Here is the comprehension again, decorated with types:

```
[ (the (dept::[Dept]), sum (salary::[Salary])
| (name::Name, dept::Dept, salary::Salary)
          <- employees
, group by (dept::Dept) ]
```

Hence we find the sum of the salaries by writing `sum salary`. Function `the` returns the first element of a non-empty list of equal elements:

```
the :: Eq a => [a] -> a
the (x:xs) | all (x ==) xs  =  x
```

Thanks to the `group by` all values in the `dept` list will be the same, and so we extract the department name by writing `the dept`.

Unlike SQL, which always returns a flat list, we can use comprehensions to compute more complex structures. For example, to find the names of employees grouped by department, we could write

```
[ (the dept, name)
| (name, dept, salary) <- employees
, group by dept ]
```

which returns

```
[ ("MS",   ["Simon","Erik"]  )
, ("Ed",   ["Phil","Gordon"] )
, ("Yale", ["Paul"]          ) ]
```

Or if we want to pair names with salaries, we could write

```
[ (the dept, namesalary)
| (name, dept, salary) <- employees
```



```
, let namesalary = (name, salary)
, group by dept ]
```

which returns

```
[ ("MS",   [ ("Simon", 80), ("Erik", 100)  ] )
, ("Ed",   [ ("Phil", 40),  ("Gordon", 45) ] )
, ("Yale", [ ("Paul", 60)                  ] ) ]
```

As above, the type of `namesalary` is changed by the `group` qualifier. Before the `group` qualifier `namesalary` has type `(Name,Salary)`, but after it has type `[(Name,Salary)]`. In Section 4 we make precise what "before" and "after" mean, and we also formalise the usual `let` notation for Haskell list comprehensions used in this example.

3.5 User-defined grouping

Just as with `order`, we can generalise `group` to take an (optional) user-defined function. The default grouping function is `groupWith`, which sorts on the group key:

```
groupWith :: Ord b => (a -> b) -> [a] -> [[a]]
groupWith f = groupBy (\x y -> f x == f y)
            . sortWith f
```

To accumulate adjacent groups *without* sorting, we may use the following variant:

```
groupRun :: Eq b => (a -> b) -> [a] -> [[a]]
groupRun f = groupBy (\x y -> f x == f y)
```

For example, we may count the length of adjacent runs of trades on a given stock with the following.

```
[ (the stock, length stock, average price)
| (stock, price) <- trades
, group by stock using groupRun ]
```

If trades is the list

```
[ ("MSFT",  80.00)
, ("MSFT",  70.00)
, ("GOOG", 100.00)
, ("GOOG", 200.00)
, ("GOOG", 300.00)
, ("MSFT",  30.00)
, ("MSFT",  20.00) ]
```

this returns

```
[ ("MSFT", 2,  75.00)
, ("GOOG", 3, 200.00)
, ("MSFT", 2,  55.00) ]
```

In general, we can write

$$\texttt{group by}\ e\ \texttt{using}\ f$$

whenever e has type τ and f has type

$$\forall a.\ (a \to \tau) \to \texttt{[}a\texttt{]} \to \texttt{[[}a\texttt{]]}.$$

As before, we require f to be polymorphic in the element type a. The only difference between `sort by` and `group by` is that the former takes a list to a list, while the latter takes a list to a list of lists.

3.6 Dropping the `by` clause in grouping

It is also possible to drop the "`by`" clause in a group. For example, the following function breaks a stream into successive runs of a given length.

```
runs :: Int -> [a] -> [[a]]
runs n  =  map (take n) . iterate (drop 1)
```


For example, one can compute a running average over the last three trades for a given stock as follows.

```
[ average price
| (stock, price) <- trades
, stock == 'MSFT'
, group using runs 3 ]
```

For the data above, this returns

```
[ 60.00, 40.00 ]
```

(since 60 = (80 + 70 + 30) / 3 and 40 = (70 + 30 + 20) / 3).

In general, we can write

`group using` f

whenever f has type

$$\forall a.\ [a] \to [[a]]$$

Again, we require f to be polymorphic in the element type a. As before, omitting "by" is mere convenience, since

$$\texttt{group using}\ f \quad \equiv \quad \texttt{group by ()}\ \texttt{using}\ \lambda x.\ f$$

where x does not appear in f.

3.7 Having

In SQL, while one filters rows with `WHERE`, one filters groups with `HAVING`. Here is the previous query, modified to consider only employees with a salary greater than 50K, and departments having at least ten such employees.

```
SELECT dept, SUM(salary)
FROM employees
WHERE salary > 50
GROUP BY dept
HAVING COUNT(name) > 10
```

In our notation, both the `WHERE` and `HAVING` clauses translate into guards of the comprehension.

```
[ (the dept, sum salary)
| (name, dept, salary) <- employees
, salary > 50
, group by dept
, length name > 10 ]
```

The rebinding of variables to lists leads naturally to guards serving the same purpose as `HAVING` clauses, when they appear after the

grouping operator.

3.8 Zip

GHC and Hugs have for some time supported an extension to list comprehensions that makes it easy to draw from two lists in parallel. For example:

```
[ x+y
| x <- xs
| y <- ys ]
```

Here we draw simultaneously from `xs` and `ys`, so that if `xs` is `[1,2,3]` and `ys` is `[4,5,6]` the comprehension returns `[5,7,9]`. Of course there can be multiple qualifiers in each of the parallel parts. For example:

```
[ x+y
| x <- xs, order by x
| y <- ys ]
```

Here we sort the list `xs` before pairing with the corresponding element of `ys`.

3.9 Parenthesised qualifiers

With the new generality of qualifiers, it makes sense to parenthesise qualifiers. For example, consider

```
p1 = [ (x,y,z)
     | ( x <- xs
```

```
        | y <- ys )
      , z <- zs ]
```

Here we draw from `xs` and `ys` in parallel, and then take all combinations of such pairs with elements of `zs`. For example, if

```
xs = [1,2]
ys = [3,4]
zs = [5,6]
```

then the comprehension would return

```
[(1,3,5), (1,3,6), (2,4,5), (2,4,6)]
```

It would mean something quite different if we wrote

```
p2 = [ (x,y,z)
     | x <- xs
     | ( y <- ys
       , z <- zs) ]
```

Here we take all combinations of elements from `ys` and `zs`, and draw from that list and `xs` in parallel. There are four elements in list of combinations, but only two in `xs`, so the extra ones are dropped, and the query returns

```
[(1,3,5), (2,3,6)]
```

(The parentheses on the qualifiers are redundant in this second example, because we take comma to bind more tightly than bar.)

Similar considerations apply to `order` and `group`. Consider

```
p3 = [ (x,y)
     | (  x <- [1..3]
```

```
        ,  y <- [1..3] )
      , order by x >= y using takeWhile ]
```

and

```
p4 = [ (x,y)
     | x <- [1..3],
       ( y <- [1..3],
       , order by x >= y using takeWhile ) ]
```

which differ only in how the qualifiers are parenthesised. The first returns

```
[ (1,1) ]
```

while the second returns

```
[ (1,1), (2,1), (2,2), (3,1), (3,2), (3,3) ].
```

Similarly, parentheses can be used to control exactly how `group` works. Consider

```
p5 = [ (x, y, the b)
     | ( x <- [1..3]
       , y <- [1..3]
       , let b = (x >= y) )
     , group by b ]
```

and

```
p6 = [ (x, y, the b)
     | x <- [1..3],
       ( y <- [1..3],
       , let b = (x >= y)
```

```
        , group by b ) ]
```


which differ only in how the qualifiers are parenthesised. The first returns

```
[ ([1,2,2,3,3,3], [1,1,2,1,2,3], True),
  ([1,1,2],[2,3,3], False) ]
```

while the second returns

```
[ (1, [1], True), (1, [2,3], False),
  (2, [1,2], True), (2, [3], False),
  (3, [1,2,3], True), (3, [], False) ]
```

Not only the answers are different, but even the *types* of the answers. Since `x` is in scope of the `group` in `p5`, it is bound to a list of integers in the result, while since `x` is not in scope of `group` in `p6` comprehension, it is bound to an integer in the result.

If no parentheses are used, both `order` and `group` scope as far to the left as possible, so that

```
p3' = [ (x,y)
      | x <- [1..3]
      , y <- [1..3]
      , order by x >= y using takeWhile ]
```

behaves the same as example `p3`. As we shall see in the next section, the syntax ensures that there is always a qualifier to the left of an `order` or `group`.

All of this may seem a little tricky, but the good news is that parentheses are never required. Instead, one can simply use a nested

comprehension, at some modest cost in duplicated variable bindings. For example, `p4` can be written:

```
p4' = [ (x,y)
      | x <- [1..3],
      , y <- [y | y <- [1..3]
                , order by x >= y using takeWhile] ]
```

4. Semantics

We now explain the semantics of extended comprehensions, looking at the syntax, the translation into a language without comprehensions, the type rules, the role of parametricity, and an alternate translation.

4.1 Syntax

The syntax of comprehensions is given in Figure 1. We let x, y, z range over variables, e, f, g over expressions, w over patterns, and p, q, r range over qualifiers. A comprehension consists of an expression and a qualifier. There are two qualifiers that bind a single variable (generators and `let`), two that bind no variables (guards and empty qualifiers), two that combine two qualifiers (cartesian product and zip), and two that modify a single qualifier (order and group). In a generator the expression is list-valued, while in a guard the expression is boolean-valued. On the left hand side of a generator is a pattern which is (for now) an arbitrarily-nested tuple of variables. The empty qualifier is not much use in practical programs, but can be useful when manipulating comprehensions using laws.

The grammar explicitly indicates that parentheses may be used

with qualifiers. The order and group constructs extend as far to the left as possible, and comma binds more tightly than bar. The cartesian product of qualifiers is associative, so that $p, (q, r)$ and $(p, q), r$ are equivalent.

In the order and group constructs, either the `by` clause or the `using` clause may be optionally omitted, but not both. A missing `using` clause expands to invoke the default functions `sortWith` and `groupWith` as defined in Sections 3.1 and 3.4:

$$q, \texttt{order by}\ e \quad = \quad q, \texttt{order by}\ e\ \texttt{using sortWith}$$
$$q, \texttt{group by}\ e \quad = \quad q, \texttt{group by}\ e\ \texttt{using groupWith}$$

Variables x, y, z

Expressions $e, f, g ::= \ldots \mid$ `[` e `|` q `]`

Patterns $w ::= x \mid (w_1, \ldots, w_n)$

Qualifiers

p, q, r	$::=$	w `<-` e	Generator
	$\mid$	`let` w `=` e	Let
	$\mid$	e	Guard
	$\mid$	`()`	Empty qualifier
	$\mid$	p, q	Cartesian product
	$\mid$	p `|` q	Zip
	$\mid$	q, `order` [`by` e] [`using` f]	Order
	$\mid$	q, `group` [`by` e] [`using` f]	Group

	$($ q $)$	Parentheses

Figure 1. Syntax of list comprehensions

$$\boxed{\Gamma \vdash e : \tau}$$

$$\frac{\Gamma \vdash q \Rightarrow \Delta \quad \Gamma, \Delta \vdash e : \tau}{\Gamma \vdash [\, e \mid q \,] : [\tau]} \text{ COMP}$$

$$\boxed{\vdash w : \tau \Rightarrow \Delta}$$

$$\frac{}{\vdash x : \tau \Rightarrow \{x : \tau\}} \text{ VAR}$$

$$\frac{\vdash w_1 : \tau_1 \Rightarrow \Delta_1 \quad \ldots \quad \vdash w_n : \tau_n \Rightarrow \Delta_n}{\vdash (w_1, \ldots, w_n) : (\tau_1, \ldots, \tau_n) \Rightarrow \Delta_1 \cup \cdots \cup \Delta_n} \text{ TUP}$$

$$\boxed{\Gamma \vdash q \Rightarrow \Delta}$$

$$\frac{\Gamma \vdash e : \texttt{Bool}}{\Gamma \vdash e \Rightarrow ()} \text{ GUARD} \qquad \frac{}{\Gamma \vdash () \Rightarrow ()} \text{ UNIT}$$

$$\frac{\Gamma \vdash e : [\tau] \quad \vdash w : \tau \Rightarrow \Delta}{\Gamma \vdash w \texttt{ <- } e \Rightarrow \Delta} \text{ GEN}$$

$$\frac{\Gamma \vdash e : \tau \quad \vdash w : \tau \Rightarrow \Delta}{\Gamma \vdash \texttt{let } x \texttt{ = } e \Rightarrow (x : \tau)} \text{ LET}$$

$$\frac{\Gamma \vdash p \Rightarrow \Delta \quad \Gamma, \Delta \vdash q \Rightarrow \Delta'}{\Gamma \vdash p, q \Rightarrow \Delta, \Delta'} \text{ COMMA}$$

$$\frac{\Gamma \vdash p \Rightarrow \Delta \quad \Gamma \vdash q \Rightarrow \Delta'}{\Gamma \vdash p \mid q \Rightarrow \Delta, \Delta'} \text{ BAR}$$

$$\frac{\begin{array}{c}\Gamma \vdash q \Rightarrow \Delta \quad \Gamma, \Delta \vdash e : \tau \\ \Gamma \vdash f : \forall a.\, (a \to \tau) \to [a] \to [a]\end{array}}{\Gamma \vdash q, \texttt{order by}\ e\ \texttt{using}\ f \Rightarrow \Delta} \text{ ORDER1}$$

$$\frac{\Gamma \vdash q \Rightarrow \Delta \quad \Gamma \vdash f : \forall a.\, [a] \to [a]}{\Gamma \vdash q, \texttt{order using}\ f \Rightarrow \Delta} \text{ ORDER2}$$

$$\frac{\begin{array}{c}\Gamma \vdash q \Rightarrow \Delta \quad \Gamma, \Delta \vdash e : \tau \\ \Gamma \vdash f : \forall a.\, (a \to \tau) \to [a] \to [[a]]\end{array}}{\Gamma \vdash q, \texttt{group by}\ e\ \texttt{using}\ f \Rightarrow [\Delta]} \text{ GROUP1}$$

$$\frac{\Gamma \vdash q \Rightarrow \Delta \quad \Gamma \vdash f : \forall a.\, [a] \to [[a]]}{\Gamma \vdash q, \texttt{group by}\ e\ \texttt{using}\ f \Rightarrow [\Delta]} \text{ GROUP2}$$

Figure 2. Typing of list comprehensions

A missing by clause expands as described in Sections 3.3 and 3.6:

$$\begin{array}{lcl} q, \texttt{order using}\ f & = & q, \texttt{order by ()}\ \texttt{using}\ \lambda x.\, f \\ q, \texttt{group using}\ f & = & q, \texttt{group by ()}\ \texttt{using}\ \lambda x.\, f \end{array}$$

where x does not appear in f. We give typing rules and translations for missing `by` clauses directly, but the same rules can be derived by applying the above expansion.

4.2 Types

The type rules for comprehensions are given in Figure 2. We let τ range over types, a range over type variables, and Γ and Δ range over environments mapping variables to types. The typing judgement $\Gamma \vdash e : \tau$ indicates that in environment Γ the term e has type τ. We only give here the rule for comprehensions (rule COMP).

The typing judgement $\vdash w : \tau \Rightarrow \Delta$ indicates that pattern w of type τ binds variables with typings described by Δ. A variable yields a single binding (rule VAR), while a tuple yields the union of its bindings (rule TUP).

The typing judgement $\Gamma \vdash q \Rightarrow \Delta$ indicates that in environment Γ the qualifier q binds variables with typings described by Δ. A guard and the empty qualifier yield no bindings (rules GUARD and UNIT), while a generator and a `let` binding yield a binding for each variable in w (rules GEN and LET).

A cartesian product and a zip yield the bindings introduced by their contained qualifiers (rules COMMA and BAR). However these two rules are not identical: in the cartesian product all bindings introduced by the qualifier on the left p are in scope for the qualifier on the right q, while this is not the case for a zip.

The rules for `order` and `group` require that f has a polymorphic type and, in the case where there is a `by` clause, the return type τ of f's argument function must match the type of e (rules ORDER1 and GROUP1). The typing rules for group also indicate that the type of the bound variables changes to contain *lists* of the previous type

(rules GROUP1 and GROUP2). If Δ is the environment

$$x_1 : \tau_1, \ldots, x_n : \tau_n$$

then $[\Delta]$ is the environment

$$x_1 : [\tau_1], \ldots, x_n : [\tau_n].$$

Note that in an `order` or a `group`, the bindings yielded by the contained qualifier q are in scope for the expression e in the `by` clause, but not in scope for the expression f in the `using` clause.

4.3 Translation

We define the dynamic semantics of comprehensions by giving a translation into a simpler, comprehension-free language. The translation is given in Figure 3. It is specified in terms of two operations on qualifiers. If q is a qualifier, then

- q_v is a tuple of the variables bound in q,
- $[\![q]\!]$ is the list of tuples computed by q

For example, for the qualifier

$$q \;=\; \texttt{x <- [1,2,3], y <- [4,5]}$$

the tuple of bound variables is

$$q_v \;=\; \texttt{(x,y)}$$

while the list of bindings is

$$[\![q]\!] \;=\; \texttt{[(1,4),(1,5),(2,4),(2,5),(3,4),(3,5)]}.$$

The top-level translation for comprehensions is given by

$$\texttt{[}\, e \;\texttt{|}\; q \,\texttt{]} \;=\; \texttt{map}\ (\lambda q_v.\ e)\ [\![q]\!].$$

The definition of q_v, the tuple of variables bound by q, is straightforward (Figure 3). A generator or `let` binds the variables

in the pattern, a guard or empty qualifiers binds no variables, a cartesian product or zip binds a pair consisting of the bound variables of the two contained qualifiers, and an order or group binds the same tuple as its contained qualifier.

The semantics of qualifiers is also straightforward (Figure 3). A generator just returns its associated list, and a `let` returns a singleton list consisting of the bound value. A guard returns either a singleton list or an empty list, depending on whether the boolean expression is true or false. The empty qualifier returns a singleton list containing the empty tuple. The cartesian product of two qualifiers is computed in the usual way [Wad92], mapping over each list of bindings to form a list of list of tuples, and concatenating the result. The zip of two qualifiers is particularly straightforward — it just applies `zip`! Note that p, q is defined so that the bound variables of p are in scope when evaluating q, while $p \mid q$ is defined so that the bound variables of p are *not* in scope when evaluating q.

The `order` construct simply applies the function in the `using` clause to a lambda expression over the bound tuple with the body given in the `by` clause and the bindings returned by the contained qualifier. The `group` construct is similar, except the given function returns a list of list of tuples, which is converted to a list of tuples of lists by mapping with the `unzip` function. An auxiliary definition specifies a suitable version of `unzip` corresponding to the structure of the tuple of bound variables.

To illustrate the `unzip`, consider again our example from Section 3.4:

```
[ (the dept, sum salary)
| (name, dept, salary) <- employees
, group by dept ]
```

The comprehension desugars as follows:

```
map (\ (name,dept,sal) -> (the dept, sum sal))
  (map unzip3
     (groupWith (\ (name,dept,sal) -> dept)
      employees))
```

The functions `groupWith` and `sortWith` were introduced in Sections 2 and 3.5 respectively, while the standard Prelude function

```
unzip3 :: [(a,b,c)] -> ([a],[b],[c])
```

implements $\texttt{unzip}_{\texttt{(name,dept,sal)}}$. Let us follow how this works in detail. Here is the original list of employees:

```
employees = [ ("Simon",  "MS",   80)
            , ("Erik",   "MS",   100)
            , ("Phil",   "Ed",   40)
            , ("Gordon", "Ed",   45)
            , ("Paul",   "Yale", 60) ]
```

After applying `groupWith` we get

```
groupWith (\(name,dept,sal) -> dept) employees
= [ [ ("Simon",  "MS",   80)
    , ("Erik",   "MS",   100) ]
  , [ ("Phil",   "Ed",   40)
    , ("Gordon", "Ed",   45) ]
  , [ ("Paul",   "Yale", 60) ] ]
```

Unzipping turns each list of triples into a triple of lists:

```
map unzip3
  (groupWith (\(name,dept,sal) -> dept) employees)
  = [ (["Simon","Erik"],   ["MS","MS"], [80,100] )
    , (["Phil","Gordon"], ["Ed","Ed"], [40,45]  )
    , (["Paul"],          ["Yale"],    [60]     ) ]
```

Finally, mapping `(\ (name,dept,sal) -> (the dept, sum sal))` over this list gives the desired result

2007/6/18

$$
\begin{array}{rcl}
[\, e \mid q \,] & = & \texttt{map}\ (\lambda q_v.\ e)\ [\![q]\!] \\
\\
(w \texttt{ <- } e)_v & = & w \\
(\texttt{let } w \texttt{ = } d)_v & = & w \\
(g)_v & = & () \\
()_v & = & () \\
(p, q)_v & = & (p_v, q_v) \\
(p \mid q)_v & = & (p_v, q_v) \\
(q, \texttt{order [by } e\texttt{] [using } f\texttt{]})_v & = & q_v \\
(q, \texttt{group [by } e\texttt{] [using } f\texttt{]})_v & = & q_v \\
\\
[\![w \texttt{ <- } e]\!] & = & e \\
[\![\texttt{let } w\texttt{=}d]\!] & = & [d] \\
[\![g]\!] & = & \texttt{if } g \texttt{ then [()] else []} \\
[\![()]\!] & = & [()] \\
[\![p, q]\!] & = & \texttt{concat}\ (\texttt{map}\ (\lambda p_v.\ \texttt{map}\ (\lambda q_v.\ (p_v, q_v))\ [\![q]\!])\ [\![p]\!]) \\
[\![p \mid q]\!] & = & \texttt{zip}\ [\![p]\!]\ [\![q]\!] \\
[\![q, \texttt{order by } e \texttt{ using } f]\!] & = & f\ (\lambda q_v.\ e)\ [\![q]\!] \\
[\![q, \texttt{order using } f]\!] & = & f\ [\![q]\!] \\
[\![q, \texttt{group by } e \texttt{ using } f]\!] & = & \texttt{map unzip}_{q_v}\ (f\ (\lambda q_v.\ e)\ [\![q]\!]) \\
[\![q, \texttt{group using } f]\!] & = & \texttt{map unzip}_{q_v}\ (f\ [\![q]\!]) \\
\\
\texttt{unzip}_{()}\ e & = & () \\
\texttt{unzip}_{x}\ e & = & e \\
\texttt{unzip}_{(w_1,\ldots,w_n)}\ e & = & (\texttt{unzip}_{w_1}\ (\texttt{map}\ (\lambda(x_1,\ldots,x_n).x_1)\ e),\ \ldots,\ \texttt{unzip}_{w_n}\ (\texttt{map}\ (\lambda(x_1,\ldots,x_n).x_n)\ e))
\end{array}
$$

Figure 3. Translation of comprehensions

$$
\begin{array}{rcl}
(p, q)_v & = & p_v \otimes q_v \\
(p \mid q)_v & = & p_v \otimes q_v \\
\\
[\![p, q]\!] & = & \texttt{concat}\ (\texttt{map}\ (\lambda p_v.\ \texttt{map}\ (\lambda q_v.\ p_v \otimes q_v)\ [\![q]\!])\ [\![p]\!]) \\
[\![p \mid q]\!] & = & \texttt{zipWith}\ (\lambda p_v.\ \lambda q_v.\ p_v \otimes q_v)[\![p]\!]\ [\![q]\!]
\end{array}
$$

Figure 4. Translation of comprehensions with tuple concatenation

$$
\begin{array}{rcl}
[\, e \mid x \texttt{ <- } e' \,] & = & \texttt{map}\ (\lambda x.\ e)\ e' \\
[\, e \mid \texttt{let } w \texttt{ = } d \,] & = & \texttt{let } w = d \texttt{ in } [e] \\
[\, e \mid e' \,] & = & \texttt{if } e' \texttt{ then } [e] \texttt{ else } [] \\
[\, e \mid () \,] & = & [\, e \,]
\end{array}
$$

$$
\begin{array}{rcl}
\texttt{[}\, e \mid p, q \,\texttt{]} & = & \texttt{concat [[} e \mid q \,\texttt{] | } p \,\texttt{]} \\
\texttt{[}\, e \mid q, \texttt{ order by } e' \texttt{ using } f \,\texttt{]} & = & \texttt{[}\, e \mid q_v \texttt{ <- } f\,(\lambda q_v.\, e')\,\texttt{[}\, q_v \mid q \,\texttt{]]} \\
\texttt{[}\, e \mid q, \texttt{ order using } f \,\texttt{]} & = & \texttt{[}\, e \mid q_v \texttt{ <- } f\,\texttt{[}\, q_v \mid q \,\texttt{]]} \\
\texttt{[}\, e \mid q, \texttt{ group by } e' \texttt{ using } f \,\texttt{]} & = & \texttt{[}\, e \mid q_v \texttt{ <- map unzip}_{q_v}\,(f\,(\lambda q_v.\, e')\,\texttt{[}\, q_v \mid q \,\texttt{]})\,\texttt{]} \\
\texttt{[}\, e \mid q, \texttt{ group using } f \,\texttt{]} & = & \texttt{[}\, e \mid q_v \texttt{ <- map unzip}_{q_v}\,(f\,\texttt{[}\, q_v \mid q \,\texttt{]})\,\texttt{]}
\end{array}
$$

Figure 5. Another translation of comprehensions

```
[ ("MS",   180)
, ("Ed",    95)
, ("Yale", 60)]
```

4.4 Parametricity

The type rules for `order` and `group` require the supplied function to have a universally quantified type. Here, for instance, is the rule for `order`:

$$
\frac{\Gamma \vdash q \Rightarrow \Delta \quad \Gamma, \Delta \vdash e : \tau \qquad \Gamma \vdash f : \forall a.\, (a \to \tau) \to \texttt{[}a\texttt{]} \to \texttt{[}a\texttt{]}}{\Gamma \vdash q, \texttt{order by } e \texttt{ using } f \Rightarrow \Delta}\ \text{ORDER1}
$$

Arguably, we might instead have chosen f to have a more general type:

$$f : \forall ab.\, (a \to b) \to \texttt{[}a\texttt{]} \to \texttt{[}a\texttt{]}.$$

Or a more specific one:

$$f : (\sigma \to \tau) \to \texttt{[}\sigma\texttt{]} \to \texttt{[}\sigma\texttt{]}$$

where σ is the tuple type $(\tau_1, \ldots, \tau_n)$ when Δ is $x_1 : \tau_1, \ldots, x_n :$

τ_n. Why do we choose to universally quantify one argument but not the other?

We do not choose the more general type because it is *too* general. The choices we have seen for f include the following.

$$\begin{array}{ll} \texttt{sortWith} & : \quad \forall ab.\, \texttt{Ord}\; b \Rightarrow (a \to b) \to [a] \to [a] \\ \texttt{takeWhile} & : \quad \forall a.\, (a \to \texttt{Bool}) \to [a] \to [a] \end{array}$$

If we required f to have the more general type, then we could not instantiate f to either of these functions. So we need a more specific type.

Similarly, we do not choose the more specific type because it is *too* specific; it requires us to fix details of how tuples of bound variables are encoded. Indeed, the nested encoding of tuples in the preceding section does not quite match the flat encoding of environments given above. For example, recall that the qualifier

$$q \;=\; \texttt{x <- xs, y <- ys, z <- zs}$$

yields the tuple of bound variables $q_v \;=\; \texttt{((x,y),z)}$, whereas to use the more specific type given above we would need to choose $q_v \;=\; \texttt{(x,y,z)}$. So we need a more general type.

Using a universally quantified type not only ensures that the function has the right type to work with arbitrary encodings of tuples, but also ensures that changing the encoding will not change the semantics. This follows because of semantic parametricity (sometimes called 'theorems for free'), which ensures universally quantified functions satisfy certain properties [Rey83, Wad89].

In particular, the type

$$f : \forall a.\, (a \to \tau) \to [a] \to [a]$$

has the following free theorem

$$\texttt{map}\ h \cdot f\ (g \cdot h) = f\ g \cdot \texttt{map}\ h$$

and this is exactly what is required to ensure that f gives the same result for different ways of encoding the environment. For example, we can relate the operation of f on the two different encodings of tuples discussed above by choosing

$$h\ (x, (y, z)) = (x, y, z).$$

This has the consequence—which is exactly what we would expect and hope for!—that the meaning of a comprehension is independent of precise details of how binding tuples are encoded. For instance, Figure 4 shows how to modify Figure 3 to use a flat rather than a nested encoding. The new translation modifies the definitions of q_v, $[\![q]\!]$, and $\texttt{unzip}_{q_v}$ to use tuple concatenation rather than pairing, where tuple concatenation takes an m-tuple and an n-tuple and yields an $(m+n)$-tuple,

$$(x_1, \ldots, x_m) \otimes (y_1, \ldots, y_n) \quad = \quad (x_1, \ldots, x_m, y_1, \ldots, y_n).$$

In the special cases where $m = 0$ and $m = 1$ we have

$$\begin{array}{rcl} () \otimes (y_1, \ldots, y_n) & = & (y_1, \ldots y_n) \\ x \otimes (y_1, \ldots, y_n) & = & (x, y_1, \ldots, y_n) \end{array}$$

and similarly when $n = 0$ or $n = 1$.

For instance, consider the qualifier

$$q \ = \ (\texttt{x <- xs, y <- ys), z <- zs}$$

With the old translation (Figure 3), this yields the tuple of bound variables

$$q_v \;=\; \texttt{((x,y),z)}$$

while with the new translation (Figure 4), this yields the tuple of bound variables

$$q_v \;=\; \texttt{(x,y,z)}$$

The definition of $[\![q]\!]$ is changed correspondingly. Thanks to our requirement of universally quantified types for `order` and `group`, we can guarantee that both choices of translation yield the same results.

4.5 Another translation

The style of translation given here differs from that in, say [Wad87], in that qualifiers q are translated separately into tuples of bound variables q_v and lists of bindings $[\![q]\!]$. Figure 5 gives an alternative, and more conventional translation, where qualifiers translate directly to binding constructs. It is easy to check that the translations of Figures 3 and 5 are equivalent. In particular, this means

Patterns $w ::= x \mid K\ w_1 \ldots w_n$

$$\begin{array}{rcl} (w \texttt{ <- } e)_v & = & w_v \\ (x)_v & = & x \\ (K\ w_1 \ldots w_n)_v & = & ((w_1)_v, \ldots, (w_n)_v) \end{array}$$

$$[\![w \texttt{ <- } e]\!] \quad = \quad \texttt{concat}\,(\texttt{map}\left(\begin{array}{l}\texttt{case } x \texttt{ of}\\ \lambda x. \quad w \texttt{ -> } [w_v]\\ \qquad\quad \texttt{other -> } []\end{array}\right) e)$$

Figure 6. Refutable patterns in generators

that the new definitions of the traditional qualifiers (generator, `let`, guard, empty qualifier, and cartesian product) coincide with the traditional definitions, and hence that the new definition is a conservative extension of the old.

We choose the formulation of Figure 3 partly on aesthetic grounds, because it gives a direct, compositional translation to qualifiers *themselves* rather than only to qualifiers embedded in a comprehension. Furthermore, for `group` and `order` the translation is somewhat more compact and efficient, because it does not require the construction of nested comprehensions.

4.6 Refutable patterns in generators

In Haskell, patterns built of variables and tuples are called *irrefutable*, because a match against such a pattern cannot fail; while other patterns are called *refutable*. A generator containing a refutable pattern acts as an implicit filter. For example:

```
f :: [Maybe Int] -> Int
f xs = sum [x | Just x <- xs]
```

Here, only the elements of `xs` that match the pattern `(Just x)` are chosen from `xs`.

Thus far, the syntax in Figure 1 and semantics in Figure 3

permits only irrefutable patterns, a choice we made to reduce clutter and focus attention on `order` and `group`. However, it is easy to accommodate refutable patterns in generators, as we show in Figure 6.

5. Laws

The semantics we have given validates a number of laws.

We begin with a number of laws that carry over unchanged from the usual treatment of comprehensions [Wad92]. It is a significant feature of the new formulation that it does not violate any of these laws.

The most significant law is the nesting law (which also appears as a line in Figure 5).

$$[\, e \mid p, q \,] \;=\; \texttt{concat}\ [\, [\, e \mid q \,] \mid p \,]$$

This is easily checked, as the left and right sides yield to the same term using the translation of Figure 3.

We also have a flattening law.

$$[\, e \mid p, x \leftarrow [\, f \mid q \,], r \,] \;=\; [\, e[x := f] \mid p, q, r[f := x] \,]$$

This is an immediate consequence of nesting and the following simpler law.

$$[\, e \mid x \leftarrow [\, f \mid q \,] \,] \;=\; [\, e[x := f] \mid q \,]$$

The simpler law is an immediate consequence of the translation and the map composition law.

$$\texttt{map}\ f \cdot \texttt{map}\ g \;=\; \texttt{map}\ (f \cdot g)$$

([Wad92] suggests the use of induction over the structure of com-

prehensions to prove the flattening law, but this is not necessary.)

2007/6/18

A special case of the flattening law is:

$$q \quad = \quad q_v \texttt{ <- [} q_v \texttt{ | } q \texttt{]}$$

Among other things, this law can be used to make clear grouping without using parentheses, as we saw in Section 3.9.

Cartesian product is associative and has the empty qualifier as unit.

$$\begin{array}{rcl} \texttt{[} e \texttt{ | } (p,q), r \texttt{]} & = & \texttt{[} e \texttt{ | } p, (q,r) \texttt{]} \\ \texttt{[} e \texttt{ | } p, \texttt{()]} & = & \texttt{[} e \texttt{ | } p \texttt{]} \\ \texttt{[} e \texttt{ | } \texttt{(), } p \texttt{]} & = & \texttt{[} e \texttt{ | } p \texttt{]} \end{array}$$

This is easily checked, using the fact that `concat` and `unit` are natural transformations and form a monad (where `unit x = [x]`).

$$\begin{array}{rcl} \texttt{map } f \cdot \texttt{concat} & = & \texttt{concat} \cdot \texttt{map } (\texttt{map } f) \\ \texttt{map } f \cdot \texttt{unit} & = & \texttt{unit} \cdot f \\ \texttt{concat} \cdot \texttt{concat} & = & \texttt{concat} \cdot \texttt{map concat} \\ \texttt{concat} \cdot \texttt{unit} & = & \texttt{id} \\ \texttt{concat} \cdot \texttt{map unit} & = & \texttt{id} \end{array}$$

Another law relates zip of cartesian product to cartesian product of zip. If `xs` and `ys` have the same length, and `us` and `vs` have the same length, then

```
(x <- xs | y <- ys), (u <- us | v <- vs)
=
(x <- xs, u <- us) | (y <- ys, v <- vs)
```

The proof is by induction of `xs` and `ys`, with lemmas proved by

inducting over `us` and `vs`.

We also have some laws specifically applicable to `order`. Since the default ordering function is a stable sort, sorting on two keys in succession is equivalent to sorting on a pair of keys:

$$\texttt{order by}\ d,\ \texttt{order by}\ e\ =\ \texttt{order by}\ (d, e)$$

Applying two ordering functions in succession (when there is no `by` clause) is equivalent to applying the composition of the two functions:

$$\texttt{order using}\ f,\ \texttt{order using}\ g\ =\ \texttt{order using}\ (g \cdot f)$$

Combining `by` and `using` is a bit messier:

$$\begin{array}{l}\texttt{order by}\ d\ \texttt{using}\ f,\ \texttt{order by}\ e\ \texttt{using}\ g\\ =\\ \texttt{order by}\ (d, e)\ \texttt{using}\ \lambda h.\ g\ (\texttt{snd} \cdot h) \cdot f\ (\texttt{fst} \cdot h)\end{array}$$

The `using` function takes function h that extracts a pair of keys, runs f passing it the extractor function for the first key, and then similarly g passing it the extractor function for the second key.

However, analogues of the three above laws do not appear to hold for `group`, since a single group changes all bound variables to lists, while two adjacent groups change all bound variables to lists of lists.

6. Variations on the theme

Thus far we have concentrated on describing a *particular* design in complete detail. However there are many design choices to be made, and we explore a few of them here, albeit in less detail.

6.1 Concrete syntax

We are unhappy with the use of the keyword `order` because, with a user-defined function such as `take`, no reordering at all may be involved. One suggestion is to re-use the keyword "`then`", followed immediately by the function to use:

```
[ (the dept, sum salary)
| (name, dept, salary) <- employees
, then sortWith  by salary
, then takeWhile by salary < 50
, then take 5 ]
```


The "`by`" clause remains optional, but the ordering function is not. (Perhaps it would read better to say "`using`' instead of "`by`" in this context.)

6.2 Binding in `group`

In our main design, `group` implicitly re-binds all the in-scope variables to *lists* of their previous type. This implicit re-binding is very convenient in small examples, but it is arguably rather surprising – it is certainly unique in Haskell's design – so it might be worth considering a more verbose but explicit syntax such as:

```
[ (the_dept, namesalary)
| (name, dept, salary) <- employees
, the_dept <- group by dept
      where (name,salary) -> namesalary
]
```

Here, the `group` form is extended to bind a fresh variable, `the_dept`,

which is of course takes one value for each group. The `where` clause specifies that the `namesalary` list is constructed by stuffing all the `(name,salary)` pairs from a group into a list. In general one could have an arbitrary expression to the left of the "`->`".

One could debate the concrete syntax, but the main design question is whether the clunkiness of extra syntax justifies the extra clarity.

6.3 Cubes and hierarchies

Another extension to SQL is the `CUBE` construct. The main idea is to support multi-level aggregation. For example, suppose we have a relation `sales` that gives the name, colour, size, and cost, of a number of products. Consider the query

```
SELECT size, colour, sum(cost)
FROM sales
GROUP BY CUBE( size, colour )
```

This query shows the total cost of items in the following groups:

- All items with the same size and colour.
- All items with the same size.
- All items with the same colour.
- All items.

The result relation is a table of triples, and `NULL` is used to indicate an aggregated attribute. For example, the result records for all items with the same colour might look like

$$(\texttt{NULL}, \texttt{"red"}, 23), (\texttt{NULL}, \texttt{"blue"}, 16), \ldots$$

To support this kind of multi-level aggregation we need one further

generalisation of our notation:

```
[ (atts, sum cost)
| (size, colour, cost) <- sales
, atts <- hgroup by [size,colour] using groupCube ]
```

The construct is introduced by a new keyword `hgroup`, and the user-supplied grouping function `groupCube` has type

```
groupCube :: (a -> [String])
          -> [([Maybe String], [a])]
```

Here, the key-extractor function returns a list of strings (size and colour in this case), which `groupCube` uses to make groups under various combinations of this list (as above). It differs from the previous `group by` construct, because the grouping function must return a list of *pairs*: the first component records which subset of the key list identifies this group, while the second component holds the members of the group. So the result of the above query might look like


```
[ ([Nothing, Nothing], 302)    -- All items
, ([Nothing, Just "red"], 45)  -- Red items
, ([Nothing, Just "blue"], 8)  -- Blue items
, (Just "big", Nothing),  99)  -- Big items
...etc... ]
```

In general, `hgroup` requires the user-supplied function f to have type:

$$f :: \forall a.(a \to \tau) \to [a] \to [(\phi, [a])]$$

for some types τ, ϕ.

Whether this extra generalisation is worth the bother is open to debate.

6.4 Implicit result concatenation

In a breadth-first search over a tree, one might write this:

```
concat [ [t1,t2] | Node _ t1 t2 <- trees ]
```

or alternatively

```
[ t | Node _ t1 t2 <- trees, t <- [t1,t2] ]
```

Neither is very appealing. A simple possibility, suggested to us by Koen Claessen, is to allow the programmer to write a comma-separated list of values before the initial vertical bar of the comprehension, thus:

```
[ t1, t2 | Node _ t1 t2 <- trees ]
```

The semantics is given by either of the expressions above. More precisely:

$$[\, e_1, \ldots, e_n \mid q \,] \quad = \quad \texttt{concatMap}\ (\lambda q_v . [e_1, \ldots, e_n])\ [\![q]\!]$$

This proposal is orthogonal to the rest of this paper.

7. Related work

We now consider how we would express the two SQL queries from the introduction in XQuery and LINQ. Recall the queries are

```
SELECT name
FROM employees
```

```
WHERE salary < 50
ORDER BY salary
```

and

```
SELECT dept, SUM(salary)
FROM employees
GROUP BY dept
```

7.1 XQuery

We assume that the XQuery variable `$employees` is bound to a sequence of `employee` elements, where each `employee` element contains a `name`, `dept`, and `salary` element.

In XQuery, we would write the first query above as

```
<query1>{
  for $employee in $employees
  where $employee/salary > 50
  order by $employee/salary
  return $employee/name
}</query1>
```

XQuery is based on a notion of comprehension (called a FLWOR expression), which includes an `order by` clause similar to the one described here, added precisely in order to make it easy to parallel the behaviour of SQL. Unlike our extension to Haskell, uses of `order by` are limited to sorting, with options for multiple keys each in ascending or descending order.

We would write the second query as:

```
<query2>{
  for $d in fn:distinct-values($employees/dept)
  let $g = $employees[dept = $d]
  return
    <group>{
      $dept,
      <sum>{ fn:sum($g/salary) }</sum>
    }</group>
}</query2>
```

This is similar to the technique used for Haskell of writing nested comprehensions, but slightly smoother because the XPath subset of XQuery provides compact notation for extracting elements from a sequence or filtering on the value of an element. XQuery has no construct that parallels `GROUP BY` directly.

Two proposals to add grouping constructs to XQuery have been put forward by others. The first of these [BCC+05] resembles ours in that the grouping construct changes the sequence of bindings, but it has explicit constructs to bind values that index groups (such as `dept`) and values aggregated within groups (such as `salary`). Here is how the running example would look:

```
<query2>{
  for $e in $employees
  group by $e/dept into $dept
        nest $e/salary into $salaries
  return
    <group>{
      $dept,
      <sum>{ fn:sum($salaries) }</sum>
```

```
    }</group>
}</query2>
```

The second proposal [Kay06] uses a predicate on adjacent elements to decide where a break between groups should occur (similar to `groupBy` in the current Haskell library), whereas our construct looks at individual bindings. Neither proposal supports user-defined functions for grouping or ordering.

7.2 LINQ

Using the LINQ features of C# 3.0, the first query is written as

```
from e in employees
where e.salary < 50
orderby e.salary
select e.name
```

As with XQuery, this is easy to write because comprehensions are extended with an `orderby` construct that parallels the behaviour of SQL, and is limited to sorting, again with options for multiple keys each in ascending or descending order.

We would write the second query as:

```
from e in employees
group e by e.dept into g
select new { g.Key, g.Sum( e => e.salary ) }
```

This is shorthand for a nested comprehension

```
from g in
  from e in employees
  group e by e.dept
```

```
select new { g.Key, g.Sum( e => e.salary ) }
```

LINQ can return nested structures, whereas SQL can only return flat relations. However, the LINQ construct is tied to a specific grouping function, which returns a specific tuple with two components, the key and the group.

The LINQ construct is rather different in structure than the one we propose here; it introduces a new data structure to represent

groups, and a new construct that invokes the grouping function and loops over the returned groups; and it is tied to a specific grouping function.

LINQ queries are general in a way that ours are not: LINQ queries operate over an arbitrary *container* type, provided it supports a particular set of operations (including `orderby`, `groupby` and several others). One reason for this generality is to support meta-programming, so that a query generates a so-called *expression tree* that can (in many cases) be translated to SQL. It is natural to ask whether our extensions could similarly extend to an arbitrary monad (or sub-class thereof), a direction we have not yet investigated.

8. Conclusion

List comprehensions are a very modest language construct: they provide syntactic sugar, but offer no new expressive power. Nevertheless, syntactic sugar can be important and, in the Darwinian process of language evolution, list comprehensions have prospered. It therefore seems productive to consider extensions of this syntactic

sugar that share the modest cost of existing comprehensions while extending their power.

In this paper we have presented extensions to Haskell list comprehensions that parallel the `ORDER BY` and `GROUP BY` clauses of SQL. Constructs that parallel `ORDER BY` are also found in XQuery, LINQ, and Links, but not in (unextended) Haskell, CPL, Erlang, or Kleisli. A construct that parallels `GROUP BY` is found in LINQ, and proposed for extensions to XQuery, but does not appear in any other language so far as we know.

The new constructs proposed here are more general than the constructs in the other languages, because they work with any function of a given type, rather than being limited to specific functions. Parametricity of these functions plays an important role in ensuring the semantics of such constructs is independent of particular details of how tuples of bindings are represented.

The grouping construct is also unusual in that it rebinds each variable in scope, from a single value to a list of values. This seems close in spirit to the behaviour of `GROUP BY` in SQL, but is arguably more uniform. The separate `WHERE` and `HAVING` clauses are subsumed by comprehension guards, and the same construct supports both aggregation and nested lists.

We have implemented a simple prototype of the translation given here to confirm its correctness. We plan to implement the new construct both in the GHC compiler for Haskell and in the Edinburgh implementation of Links, and look forward to feedback from their use. Links uses comprehensions to write queries that access a database, and the compiler converts as much of these as possible into SQL. The new constructs should allow us to compile into queries that use SQL `GROUP BY` and aggregate functions where appropriate.

Because of the generality of the new constructs, we wonder whether they might also constructively feed back into the design of new database programming languages.

Acknowledgements

Many thanks to Erik Meijer, who prodded us to find comprehension equivalents for ‘order by’ and ‘group by’, and to David Balaban, Ezra Cooper, Gavin Bierman, Sam Lindley, Tom Schrijvers, Jerome Simeon, and Don Syme for their helpful feedback.

References

[BCC+05] Kevin Beyer, Don Chamberlin, Lath S. Colby, Fatma Özcan, Hamid Pirahesh, and Yu Xu. Extending XQuery for analytics. In *ACM SIGMOD International Conference on Management of Data*, pages 503–514. ACM Press, June 2005.

[BCF+07] Scott Boag, Don Chameberlin, Mary F. Fernández, Daniela Florescu, Jonathan Robie, and Jérome Siméon. Xquery 1.0: An xml query language. Technical report, W3C Recommendation, January 2007.

[BLS+94] P Buneman, L Libkin, D Suciu, V Tannen, and L Wong. Comprehension syntax. *SIGMOD Record*, 23(1):87–96, March 1994.

[CLWY06] Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. Links: Web programming without tiers. In *Formal Methods for Components and Objects*. Springer Verlag, October 2006.

[Dar77] John Darlington. Program transformation and synthesis: Present capabilities. Technical Report Report 77/43, Imperial College of Science and Technology, London, September 1977.

[Kay06] Michael Kay. Positional grouping in XQuery. In *Third International Workshop on XQuery Implementation, Experiences, and Perspectives (XIME-P)*. ACM Press, June 2006.

[MBB06] Erik Meijer, Brian Beckman, and Gavin Bierman. LINQ: reconciling object, relations and xml in the .NET framework. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, page 706. ACM Press, June 2006.

[Rey83] JC Reynolds. Types, abstraction and parametric polymorphism. In REA Mason, editor, *Information Processing 83*, pages 513–523. North-Holland, 1983.

[TW89] P Trinder and PL Wadler. Improving list comprehension database queries. In *Fourth IEEE Region 10 Conference (TENCON)*, pages 186–192. IEEE, November 1989.

[Wad87] Phil Wadler. List comprehensions. In Simon Peyton Jones, editor, *The Implementation of Functional Programming Languages*, pages 127–138. Prentice Hall, 1987.

[Wad89] PL Wadler. Theorems for free! In MacQueen, editor, *Fourth International Conference on Functional Programming and Computer Architecture, London*. Addison Wesley, 1989.

[Wad92] Philip Wadler. Comprehending monads. *Mathematical Structures in Computer Science*, 2:461–493, 1992.

[Won00] Limsoon Wong. Kleisli, a functional query system. *Journal of Functional Programming*, 10(1):19–56, January 2000.