Comprehending Ringads

for Phil Wadler, on the occasion of his 60th birthday

Jeremy Gibbons

Department of Computer Science, University of Oxford
`http://www.cs.ox.ac.uk/jeremy.gibbons/`

Abstract. *List comprehensions* are a widely used programming construct, in languages such as Haskell and Python and in technologies such as Microsoft's Language Integrated Query. They generalize from lists to arbitrary *monads*, yielding a lightweight idiom of imperative programming in a pure functional language. When the monad has the additional structure of a so-called *ringad*, corresponding to 'empty' and 'union' operations, then it can be seen as some kind of collection type, and the comprehension notation can also be extended to incorporate *aggregations*. Ringad comprehensions represent a convenient notation for expressing *database queries*. The ringad structure alone does not provide a good explanation or an efficient implementation of relational *joins*; but by allowing heterogeneous comprehensions, involving both bag and indexed table ringads, we show how to accommodate these too.

1 Introduction

We owe a lot to Phil Wadler, not least for his work over the years on monads and comprehensions. Wadler was an early proponent of list comprehensions as a syntactic feature of functional programming languages, literally writing the book (Peyton Jones, 1987, Chapter 7) on them. Together with his student Phil Trinder, he argued (Trinder and Wadler, 1989; Trinder, 1991) for the use of comprehensions as a notation for database queries; this prompted a flourishing line of work within database programming languages, including the Kleisli language from Penn (Wong, 2000) and LINQ from Microsoft (Meijer, 2011), not to mention Wadler's own work on XQuery (Fernandez et al., 2001) and Links (Cooper et al., 2006).

Wadler was also the main driving force in explaining monads to functional programmers (Wadler, 1992b), popularizing the earlier foundational work of Moggi (1991). He showed that the notions of list comprehensions and monads were related (Wadler, 1992a), generalizing list comprehensions to arbitrary monads—of particular interest to us, to sets and bags, but also to monads like state and I/O. More recently, with Peyton Jones (Wadler and Peyton Jones, 2007) he has extended the list comprehension syntax to support 'SQL-like' ordering and grouping constructs, and this extended comprehension syntax has subsequently been generalized to other monads too (Giorgidze et al., 2011).

In this paper, we look a little more closely at the use of comprehensions as a notation for queries. The monadic structure explains most of standard relational algebra, allowing for an elegant mathematical foundation for those aspects of database query language design. Unfortunately, monads per se offer no good explanation of relational joins, a crucial aspect of relational algebra: expressed as a comprehension, a typical equijoin $[(a,b) \mid a \leftarrow x, b \leftarrow y, f\ a == g\ b]$ is very inefficient to execute. But the ingredients we need in order to do better are all there, in Wadler's work, as usual. The novel contribution of this paper is to bring together those ingredients: the generalizations of comprehensions to different monads and to incorporate grouping are sufficient for capturing a reasonable implementation of equijoins.

2 Comprehensions

The idea of comprehensions can be seen as one half of the adjunction between extension and intension in set theory—one can define a set by its *extension*, that is by listing its elements:

$$\{1, 9, 25, 49, 81\}$$

or by its *intension*, that is by characterizing those elements:

$$\{n^2 \mid 0 < n < 10 \wedge n \equiv 1 \pmod 2\}$$

Expressions in the latter form are called *set comprehensions*. They inspired the "set former" programming notation in the SETL language (Schwartz, 1975; Schwartz et al., 1986), dating back to the late 1960s, and have become widely known through list comprehensions in languages like Haskell and Python.

2.1 List comprehensions

Just as a warm-up, here is a reminder about Haskell's list comprehensions:

$$[2 \times a + b \mid a \leftarrow [1,2,3], b \leftarrow [4,5,6], b \text{ `}mod\text{` } a == 0]$$

This (rather concocted) example yields the list $[6,7,8,8,10,12]$ of all values of the expression $2 \times a + b$, as a is drawn from $[1,2,3]$ and b from $[4,5,6]$ and such that b is divisible by a.

To the left of the vertical bar is the *term* (an expression). To the right is a comma-separated sequence of *qualifiers*, each of which is either a *generator* (of the form $a \leftarrow x$, with a variable a and a list expression x) or a *filter* (a boolean expression). The scope of a variable introduced by a generator extends to all subsequent generators and to the term.

Note that, in contrast to the naive set-theoretic inspiration, bound variables in list comprehensions need to be explicitly generated from some existing list, rather than being implicitly quantified. Without such a condition, the set-theoretic *axiom of unrestricted comprehension* ("for any predicate P, there exists a set B whose elements are precisely those that satisfy P") leads directly to Russell's Paradox; with the condition, we get the *axiom of specification* ("for any set A and predicate P, there exists a set B whose elements are precisely the elements of A that satisfy P"), which avoids the paradox.

The semantics of list comprehensions is defined by translation; see for example Wadler's chapter of Peyton Jones's book (1987, Chapter 7). The translation can be expressed equationally as follows:

$$\begin{aligned}
[\,e \mid \,] &= [\,e\,] \\
[\,e \mid b\,] &= \mathbf{if}\ b\ \mathbf{then}\ [\,e\,]\ \mathbf{else}\ [\,] \\
[\,e \mid a \leftarrow x\,] &= \mathit{map}\ (\lambda a \rightarrow e)\ x \\
[\,e \mid q, q'\,] &= \mathit{concat}\ [[\,e \mid q'\,] \mid q\,]
\end{aligned}$$

(Here, the first clause involves the empty sequence of qualifiers. This is not allowed in Haskell, but it is helpful in simplifying the translation.)

Applying this translation to the example at the start of the section gives

$$\begin{aligned}
&[2 \times a + b \mid a \leftarrow [1,2,3], b \leftarrow [4,5,6], b\ \text{`}\mathit{mod}\text{`}\ a == 0] \\
= {}&\mathit{concat}\ (\mathit{map}\ (\lambda a \rightarrow \mathit{concat}\ (\mathit{map}\ (\lambda b \rightarrow \\
&\quad \mathbf{if}\ b\ \text{`}\mathit{mod}\text{`}\ a == 0\ \mathbf{then}\ [2 \times a + b]\ \mathbf{else}\ [\,])\ [4,5,6]))\ [1,2,3]) \\
= {}&[6,7,8,8,10,12]
\end{aligned}$$

More generally, a generator may match against a pattern rather than just a variable. In that case, it may bind multiple (or indeed no) variables at once; moreover, the match may fail, in which case it is discarded. This is handled by modifying the translation for generators to use a function defined by pattern-matching, rather than a straight lambda-abstraction:

$$[\,e \mid p \leftarrow x\,] = \mathit{concat}\ (\mathit{map}\ (\lambda a \rightarrow \mathbf{case}\ a\ \mathbf{of}\ p \rightarrow [\,e\,]; \_ \rightarrow [\,])\ x)$$

or, perhaps more perspicuously,

$$\begin{aligned}
[\,e \mid p \leftarrow x\,] = {}&\mathbf{let}\ h\ p = [\,e\,] \\
&\qquad h\ \_ = [\,] \\
&\mathbf{in}\ \mathit{concat}\ (\mathit{map}\ h\ x)
\end{aligned}$$

2.2 Monad comprehensions

It is clear from the above translation that the necessary ingredients for list comprehensions are *map*, singletons, *concat*, and the empty list. The first three are the operations arising from lists as a functor and a monad, which suggests that the same translation might be applicable to other monads too.

$$\begin{aligned}
&\mathbf{class}\ \mathit{Functor}\ m\ \mathbf{where} \\
&\quad \mathit{fmap} :: (a \rightarrow b) \rightarrow m\ a \rightarrow m\ b \\
&\mathbf{class}\ \mathit{Functor}\ m \Rightarrow \mathit{Monad}\ m\ \mathbf{where} \\
&\quad \mathit{return} :: a \rightarrow m\ a \\
&\quad (\gg\!=) \ :: m\ a \rightarrow (a \rightarrow m\ b) \rightarrow m\ b \\
&\quad \mathit{mult} \ \ :: m\ (m\ a) \rightarrow m\ a \\
&\quad \mathit{mult} \ \ = (\gg\!= \mathit{id})
\end{aligned}$$

(we write *mult* for the multiplication of the monad, rather than *join* as in Haskell, to avoid confusion with the relational joins discussed later). But the fourth ingredient, the empty list, does not come from the functor and monad structures; that requires an extra assumption:

$$\begin{array}{l}\textbf{class}\ \mathit{Monad}\ m \Rightarrow \mathit{MonadZero}\ m\ \textbf{where}\\ \quad \mathit{mzero} :: m\ a\end{array}$$

Given this extra assumption, the translation for list comprehensions can be generalized to other monads:

$$\begin{array}{ll}[\,e \mid\,] & = \mathit{return}\ e\\ [\,e \mid b\,] & = \textbf{if}\ b\ \textbf{then}\ \mathit{return}\ e\ \textbf{else}\ \mathit{mzero}\\ [\,e \mid p \leftarrow x\,] & = \textbf{let}\ h\ p = \mathit{return}\ e; h\ \_ = \mathit{mzero}\ \textbf{in}\ \mathit{mult}\ (\mathit{fmap}\ h\ x)\\ [\,e \mid q, q'\,] & = \mathit{mult}\ [[\,e \mid q'\,] \mid q\,]\end{array}$$

Note that $[\,e \mid\,] = [\,e \mid \mathit{True}\,]$, so the empty case is not really needed; and that we could have written $\mathit{mult}\ (\mathit{fmap}\ h\ x)$ as $x \ggg\!= h$, but this would not be so obviously a generalization of the list instance. The actual monad to be used is implicit in the notation, possibly determined by any input value m on the right-hand side of a generator; if we want to be explicit, we could write a subscript, as in "$[\,e \mid q\,]_{\mathit{List}}$".

This translation is different from the one used in the Haskell language specification for **do** notation (Haskell 2010, §3.14) and in the GHC documentation for monad comprehensions (GHC 7.10, §7.3.15), which are arguably a little awkward: the empty list crops up in two different ways in the translation of list comprehensions—for filters, and for generators with patterns—and these are generalized in two different ways to other monads: to the *mzero* method of the *MonadPlus* class in the first case, and the *fail* method of the *Monad* class in the second. It is perhaps neater to have a monad subclass *MonadZero* with a single method subsuming both these operators. (The *fail* method of Haskell's *Monad* class is generally unpopular. There is a current proposal (Luposchainsky, 2015) to remove *fail* to a *MonadFail* subclass, but the proposal would still retain both *fail* and *mzero*, and their applications in **do** notation and monad comprehensions.) Of course, this change does mean that the translation forces a monad comprehension with filters to be interpreted in an instance of the *MonadZero* subclass rather than just of *Monad*—the type class constraints that are generated depend on the features used in the comprehension (as already happens in Haskell for generators with patterns).

Taking this approach gives basically Wadler's monad comprehension notation (Wadler, 1992a); it loosely corresponds to Haskell's **do** notation, except that the term is a value to the left of a vertical bar rather than a computation at the end, and that filters are just boolean expressions rather than introduced using *guard*.

We might impose the law that *mult* distributes over *mzero*, in the sense

$$\mathit{mult}\ \mathit{mzero} = \mathit{mzero}$$

or, in terms of comprehensions,

$$[\,e \mid a \leftarrow \mathit{mzero}\,] = \mathit{mzero}$$

Informally, this means that any failing steps of the computation cleanly cut off subsequent branches. Another way of saying it is that *mzero* is a 'left' zero of composition:

$$\mathit{mzero} \ggg\!= k = \mathit{mzero}$$

Conversely, we do not require that *mzero* is a 'right' zero of composition:

$$m \ggg\!= \lambda a \rightarrow \mathit{mzero} \neq \mathit{mzero} \qquad \text{(in general)}$$

Imposing this law would have the consequence that a failing step also cleanly erases any effects from earlier parts of the computation, which is too strong a requirement for many monads—particularly those of the "launch missiles now" variety.

2.3 Heterogeneous comprehensions

We have seen that comprehensions can be interpreted in an arbitrary monad; for example, $[\,a^2 \mid a \leftarrow x, \mathit{odd}\ a\,]_{\mathit{Set}}$ denotes the set of the squares of the odd elements of the set x, whereas $[\,a^2 \mid a \leftarrow y, \mathit{odd}\ a\,]_{\mathit{Bag}}$ denotes the bag of squares of odd elements of bag y. These are both 'homogeneous comprehensions', each involving just one monad; can we make sense of 'heterogeneous comprehensions', involving several different monads?

For monads M and N, a monad morphism $\varphi : M \rightarrow N$ is a natural transformation $M \dot{\rightarrow} N$—that is, a family $\varphi_\alpha :: M\ \alpha \rightarrow N\ \alpha$ of arrows, coherent in the sense that $\varphi_\beta \cdot \mathit{fmap}_M\ f = \mathit{fmap}_N\ f \cdot \varphi_\alpha$ for $f :: \alpha \rightarrow \beta$—that also preserves the monad structure:

$$\begin{aligned} \varphi \cdot \mathit{return}_M &= \mathit{return}_N \\ \varphi \cdot \mathit{mult}_M &= \mathit{mult}_N \cdot \varphi \cdot \mathit{fmap}_M\ \varphi = \mathit{mult}_N \cdot \mathit{fmap}_N\ \varphi \cdot \varphi \end{aligned}$$

A monad morphism behaves nicely with respect to monad comprehensions—a comprehension interpreted in monad M, using inputs of type M, with the result coerced via a monad morphism $\varphi : M \dot{\rightarrow} N$ to monad N, is equivalent to the comprehension interpreted in monad N in the first place, with the inputs having been coerced to type N. Informally, there will be no surprises arising from when coercions take place, because the results are the same whatever stage this happens. This property is straightforward to show by induction over the structure of the comprehension.

For example, if $\mathit{bag2set} : \mathit{Bag} \dot{\rightarrow} \mathit{Set}$ is the obvious monad morphism from bags to sets, discarding information about the multiplicity of repeated elements, and x a bag of numbers, then

$$\mathit{bag2set}\ [\,a^2 \mid a \leftarrow x, \mathit{odd}\ a\,]_{\mathit{Bag}} = [\,a^2 \mid a \leftarrow \mathit{bag2set}\ x, \mathit{odd}\ a\,]_{\mathit{Set}}$$

and both yield the set of squares of the odd members of bag x. As a notational convenience, we might elide use of the monad morphism when it is 'obvious from context'—we might write just $[\,a^2 \mid a \leftarrow x, \mathit{odd}\ a\,]_{\mathit{Set}}$ even when x is a bag, relying on the 'obvious' morphism *bag2set*. This would allow us to write heterogeneous comprehensions such as

$$[\, a + b \mid a \leftarrow [1,2,3], b \leftarrow \wr 4,4,5 \int]_{Set} = \{5,6,7,8\}$$

(writing $\wr \ldots \int$ for the extension of a bag), instead of the more pedantic

$$[\, a + b \mid a \leftarrow \mathit{list2set}\ [1,2,3], b \leftarrow \mathit{bag2set}\ \wr 4,4,5 \int]_{Set}$$

Sets, bags, and lists are all members of the so-called *Boom Hierarchy* of types (Backhouse, 1988), consisting of types whose values are constructed from 'empty' and 'singleton's using a binary 'union' operator; 'empty' is the unit of 'union', and the hierarchy pertains to which properties out of associativity, commutativity, and idempotence the 'union' operator enjoys. The name 'Boom hierarchy' seems to have been coined at an IFIP WG2.1 meeting by Stephen Spackman, for an idea due to Hendrik Boom, who by happy coincidence is named in his native language after another member of the hierarchy, namely (externally labelled, possibly empty, binary) trees:

```
data Tree a = Empty | Tip a | Bin (Tree a) (Tree a)
```

for which 'union' is defined as a smart *Bin* constructor, ensuring that the empty tree is a unit:

```
bin Empty u = u
bin t Empty = t
bin t u     = Bin t u
```

but with no other properties of *bin*. It is merely a historical accident that the familiar members of the Boom Hierarchy are linearly ordered by their properties; one can perfectly well imagine other more exotic combinations of the laws—such as 'mobiles', in which the union operator is commutative but not associative or idempotent (Uustalu, 2015).

There is a forgetful function from any poorer member of the Boom Hierarchy to a richer one, flattening some distinctions by imposing additional laws—for example, from bags to sets, flattening distinctions concerning multiplicity—and one could class these forgetful functions as 'obvious' morphisms. On the other hand, any morphisms in the opposite direction—such as sorting, from bags to lists, and one-of-each, from sets to bags—are not 'obvious' (and in particular, contradicting Wong (1994, p114–115), they are not monad morphisms), and so should not be elided; and similarly, it would be hard to justify as 'obvious' any morphisms involving non-members of the Boom Hierarchy, such as probability distributions.

3 Ringads and collections

One more ingredient is needed in order to characterize monads that correspond to 'collection' types such as sets and lists, as opposed to other monads such as *State* and *IO*; that ingredient is an analogue of set union or list append. It's not difficult to see that this is inexpressible in terms of the operations introduced so far: given only collections x of at most one element, any comprehension using generators of the form $a \leftarrow x$ will only yield another such collection, whereas the union of two one-element collections will in general have two elements.

To allow any finite collection to be expressed, it suffices to introduce a binary union operator *mplus*:

$$\begin{array}{l}\mathbf{class}\ \mathit{Monad}\ m \Rightarrow \mathit{MonadPlus}\ m\ \mathbf{where} \\ \quad \mathit{mplus} :: m\ a \to m\ a \to m\ a\end{array}$$

We require *mult* to distribute over union, in the following sense:

$$\mathit{mult}\ (x\ \text{`}\mathit{mplus}\text{`}\ y) = \mathit{mult}\ x\ \text{`}\mathit{mplus}\text{`}\ \mathit{mult}\ y$$

or, in terms of comprehensions,

$$[\,e \mid a \leftarrow x\ \text{`}\mathit{mplus}\text{`}\ y, q\,] = [\,e \mid a \leftarrow x, q\,]\ \text{`}\mathit{mplus}\text{`}\ [\,e \mid a \leftarrow y, q\,]$$

or of monadic bind:

$$(x\ \text{`}\mathit{mplus}\text{`}\ y) \ggg k = (x \ggg k)\ \text{`}\mathit{mplus}\text{`}\ (y \ggg k)$$

Note that, in contrast to the Haskell libraries, we have identified separate type classes *MonadZero*, *MonadPlus* for the two methods *mzero*, *mplus*. We have already seen that there are uses of *mzero* that do not require *mplus*; and conversely, there is no a priori reason to insist that all uses of *mplus* have to be associated with an *mzero*.

But for our model of collection types, we will insist on a monad in both *MonadZero* and *MonadPlus*. We will call this combination a *ringad*, a name coined by Wadler (1990):

$$\mathbf{class}\ (\mathit{MonadZero}\ m, \mathit{MonadPlus}\ m) \Rightarrow \mathit{Ringad}\ m$$

There are no additional methods; the class *Ringad* is the intersection of the two parent classes *MonadZero* and *MonadPlus*, thereby denoting the union of the two interfaces. Haskell gives us no good way to state the laws that should be required of instances of a type class such as *Ringad*, but they are the three monad laws, distribution of *mult* over *mzero* and *mplus*:

$$\begin{array}{ll}\mathit{mult}\ \mathit{mzero} & = \mathit{mzero} \\ \mathit{mult}\ (x\ \text{`}\mathit{mplus}\text{`}\ y) & = \mathit{mult}\ x\ \text{`}\mathit{mplus}\text{`}\ \mathit{mult}\ y\end{array}$$

and *mzero* being the unit of *mplus*:

$$\mathit{mzero}\ \text{`}\mathit{mplus}\text{`}\ x = x = x\ \text{`}\mathit{mplus}\text{`}\ \mathit{mzero}$$

(There seems to be no particular reason to insist also that *mplus* be associative; we discuss the laws further in Section 3.2.) To emphasize the additional constraints, we will write "$\varnothing$" for "*mzero*" and "$\uplus$" for "*mplus*" when discussing a ringad. All members of the Boom hierarchy—sets, bags, lists, trees, and exotica too—are ringad instances. Another ringad instance, but one that is not a member of the Boom Hierarchy, is the type of probability distributions—either normalized, with a weight-indexed family of union operators, or unnormalized, with an additional scaling operator.

3.1 Aggregation

The well-behaved operations over monadic values are called the *algebras* for that monad—functions k such that $k \cdot \mathit{return} = \mathit{id}$ and $k \cdot \mathit{mult} = k \cdot \mathit{fmap}\ k$. In particular, *mult* is itself a monad algebra. When the monad is also a ringad, k necessarily distributes also over $\uplus$—it is a nice exercise to verify that defining $a \oplus b = k\ (\mathit{return}\ a \uplus \mathit{return}\ b)$ establishes that $k\ (x \uplus y) = k\ x \oplus k\ y$. Without loss of generality, we write $\mathit{reduce}(\oplus)$ for k; these are the 'reductions' of the Bird–Meertens Formalism (Backhouse, 1988). In that case, $\mathit{mult} = \mathit{reduce}(\uplus)$ is a ringad algebra.

The algebras for a ringad amount to aggregation functions for a collection: the sum of a bag of integers, the maximum of a set of naturals, and so on. We could extend the comprehension notation to encompass aggregations too, for example by adding an optional annotation, writing say "$[\,e \mid q\,]^{\oplus}$"; but this doesn't add much, because we could just have written "$\mathit{reduce}(\oplus)\ [\,e \mid q\,]$" instead. We could generalize from reductions $\mathit{reduce}(\oplus)$ to collection homomorphisms $\mathit{reduce}(\oplus) \cdot \mathit{fmap}\ f$; but this doesn't add much either, because the map is easily combined with the comprehension—it's easy to show the 'map over comprehension' property

$$\mathit{fmap}\ f\ [\,e \mid q\,] = [\,f\ e \mid q\,]$$

Fegaras and Maier (2000) develop a *monoid comprehension calculus* around such aggregations; but their name is arguably inappropriate, because it adds nothing essential to insist on associativity of the binary aggregating operator—'ringad comprehension calculus' might be a better term.

Note that, for $\mathit{reduce}(\oplus)$ to be well-defined, $\oplus$ must satisfy all the laws that $\uplus$ does—$\oplus$ must be associative if $\uplus$ is associative, and so on. It is not hard to show, for instance, that there is no $\oplus$ on sets of numbers for which $\mathit{sum}\ (x \cup y) = \mathit{sum}\ x \oplus \mathit{sum}\ y$; such an $\oplus$ would have to be idempotent, which is inconsistent with its relationship with *sum*. (So, although $[\,a^2 \mid a \leftarrow y, \mathit{odd}\ a\,]^{+}_{\mathit{Bag}}$ denotes the sum of the squares of the odd elements of bag y, the expression $[\,a^2 \mid a \leftarrow x, \mathit{odd}\ a\,]^{+}_{\mathit{Set}}$ (with x a set) is not defined, because $+$ is not idempotent.) In particular, $\mathit{reduce}(\oplus)\ \varnothing$ must be the unit of $\oplus$, which we write $1_{\oplus}$.

We can calculate from the definition

$$[\,e \mid q\,]^{\oplus} = \mathit{reduce}(\oplus)\ [\,e \mid q\,]$$

the following translation rules for aggregations:

$$\begin{aligned}
[\,e \mid \,]^{\oplus} &= e \\
[\,e \mid b\,]^{\oplus} &= \mathbf{if}\ b\ \mathbf{then}\ e\ \mathbf{else}\ 1_{\oplus} \\
[\,e \mid p \leftarrow x\,]^{\oplus} &= \mathbf{let}\ h\ p = e; h\ \_ = 1_{\oplus}\ \mathbf{in}\ \mathit{reduce}(\oplus)\ (\mathit{fmap}\ h\ x) \\
[\,e \mid q, q'\,]^{\oplus} &= [\,[\,e \mid q'\,]^{\oplus} \mid q\,]^{\oplus}
\end{aligned}$$

Multiple aggregations can be performed in parallel: the so-called *banana split theorem* (Fokkinga, 1990) shows how to lift two binary operators $\oplus$ and $\otimes$ into a single binary operator $\circledast$ on pairs, such that

$$(\mathit{reduce}(\oplus)\ x, \mathit{reduce}(\otimes)\ x) = \mathit{reduce}(\circledast)\ x$$

$$\begin{array}{lll}
(\textsc{UnionUnit}) & \varnothing \uplus x & = x = x \uplus \varnothing \\
(\textsc{UnionAssoc}) & x \uplus (y \uplus z) & = (x \uplus y) \uplus z \\
(\textsc{EmptyBind}) & \varnothing \gg\!= k & = \varnothing \\
(\textsc{BindEmpty}) & x \gg\!= \lambda a \to \varnothing & = \varnothing \\
(\textsc{UnionBind}) & (x \uplus y) \gg\!= k & = (x \gg\!= k) \uplus (y \gg\!= k) \\
(\textsc{BindUnion}) & x \gg\!= \lambda a \to k\ a \uplus k'\ a & = (x \gg\!= k) \uplus (x \gg\!= k')
\end{array}$$

Fig. 1. Six possible laws for ringads (not all of them required)

for every m.

If we are to allow aggregations over heterogeneous ringad comprehensions with automatic coercions, then we had better insist that the monad morphisms $\varphi : M \dot{\to} N$ concerned are also ringad morphisms, that is, homomorphisms over the empty and union structure:

$$\begin{array}{ll}
\varphi\ (\varnothing_M) & = \varnothing_N \\
\varphi\ (x \uplus_M y) & = \varphi\ x \uplus_N \varphi\ y
\end{array}$$

3.2 Notes on "Notes on Monads and Ringads"

As observed above, Wadler introduced the term 'ringad' a quarter of a century ago in an unpublished document *Notes on Monads and Ringads* (Wadler, 1990). He requires both distributivities for "monads with zero" (the EmptyBind and BindEmpty laws in Figure 1); for ringads, he additionally requires that $\uplus$ is associative (UnionAssoc), with $\varnothing$ as its unit (UnionUnit), and that bind distributes from the right through $\uplus$ (UnionBind). He does not require that bind also distributes from the left through $\uplus$ (BindUnion).

The name 'ringad' presumably was chosen because, just as monads are monoids in a category of endofunctors, considered as a monoidal category under composition, ringads are 'right near-semirings' in such a category, considered as what Uustalu (2015) calls a 'right near-semiring category' under composition and product. A *right near-semiring* is an algebraic structure $(R, +, \times, 0, 1)$ in which $(R, +, 0)$ and $(R, \times, 1)$ are monoids, $\times$ distributes rightwards over $+$ (that is, $(a + b) \times c = (a \times c) + (b \times c)$) and is absorbed on the right by zero (that is, $0 \times a = a$). It is called 'semi-' because there is no additive inverse (a semiring is sometimes called a 'rig'), 'near-' because addition need not be commutative, and 'right-' because we require distributivity and absorption only from the right, not from the left too. This structure doesn't quite fit our circumstances, because we haven't insisted on associativity of the additive operation $\uplus$; but Wadler does in his note (and there seems to be no standard name for the structure when associativity of addition is dropped).

Wadler's note was cited in a few papers from the 1990s (Trinder, 1991; Watt and Trinder, 1991; Boiten and Hoogendijk, 1995, 1996; Suciu, 1993a,b), of which only Trinder's DBPL paper (Trinder, 1991) seems to have been formally published. Some other works (Wong, 1994; Grust, 1999) describe ringads, but cite Trinder's published paper (Trinder, 1991) instead of Wadler's unpublished note (Wadler, 1990).

Somewhat frustratingly, despite all citing a common source, the various papers differ on the laws they require for a ringad. For the monoidal aspects, everyone agrees that $\varnothing$ should be a unit of $\uplus$ (UnionUnit), but opinions vary on whether $\uplus$ should at least be associative (UnionAssoc). Trinder (1991) does not require (UnionAssoc); Boiten and Hoogendijk (1995, 1996) do, as does Uustalu (2015), and Wadler (1997) in a mailing list message about monads with zero and plus but not explicitly mentioning ringads; Grust requires it for his own constructions (Grust, 1999, §72), but implies (Grust, 1999, §73) that ringads need not satisfy it; Kiselyov (2015) argues that one should not insist on associativity without also insisting on commutativity, which not everyone wants.

For the distributivity aspects, everyone agrees that monadic bind should distribute from the right over $\uplus$ (UnionBind). Moreover, everyone agrees that bind need not distribute from the left over $\uplus$ (BindUnion); such a law would at least suggest that $\uplus$ should be commutative—in particular, it does not hold for the list monad. Nearly everyone agrees that bind should also distribute from the right over $\varnothing$ (EmptyBind)—the exception being Boiten and Hoogendijk (1995, 1996), who appear not to require it. Opinions vary as to whether bind should also distribute from the left over $\varnothing$ (BindEmpty); such a law does not hold for a monad that combines a 'global' log with the possibility of failure (such as Haskell's $\mathit{MaybeT}\ (\mathit{Writer}\ w)\ a$, which is equivalent to $(w, \mathit{Maybe}\ a)$), although it does hold when the log is 'local' (as with Haskell's $\mathit{WriterT}\ w\ \mathit{Maybe}\ a$, which is equivalent to $\mathit{Maybe}\ (w, a)$). Trinder (1991) does require (BindEmpty), in addition to (EmptyBind). Boiten and Hoogendijk (1995, 1996) require (BindEmpty), *instead* of (EmptyBind). Grust (1999, §58) requires both directions (EmptyBind) and (BindEmpty) for "monads with zero", following Wadler (1992a), but imposes only distributivity from the right (EmptyBind) for ringads (Grust, 1999, §73). Wadler himself "would usually insist on" only (EmptyBind) and not (BindEmpty), writing later (Wadler, 1997) about "monads with zero and plus". Buneman et al. (1995) attribute to ringads distributivity from the right over both $\varnothing$ (EmptyBind) and $\uplus$ (UnionBind), saying that "they seem to express fundamental properties", but say of the two distributivities from the left (BindEmpty, BindUnion) that their "status [...] is unclear". (Whether one thinks of a particular property of bind as 'distributing from the left' or 'distributing from the right' depends of course on which way round one writes bind's arguments, and this varies from author to author.)

4 Comprehending joins

Comprehensions form a convenient syntax for database queries, explaining the selection and projection operations of relational algebra. For example, consider a table of invoices

```
invoices(name, address, amount, due)
```

The SQL query

```
SELECT  name, address, amount
FROM    invoices
WHERE   due < today
```

which selects the overdue invoices and projects out the corresponding customer names and addresses and the outstanding amounts, can be expressed as the following comprehension:

$$\begin{array}{l}[\ (\mathit{name}, \mathit{address}, \mathit{amount}) \\ \mid (\mathit{name}, \mathit{address}, \mathit{amount}, \mathit{due}) \leftarrow \mathit{invoices}, \\ \ \ \mathit{due} < \mathit{today}\,]\end{array}$$

This is a reasonable approximation to how database systems implement selection and projection. However, the comprehension notation does *not* explain the third leg of relational algebra, namely join operations. For example, if the invoice database is normalized so that customer names and addresses are in a separate table from the invoices,

```
customers(cid, name, address)
invoices(cust, amount, due)
```

then the query becomes

```
SELECT  name, address, amount
FROM    customers, invoices
WHERE   cid = cust AND due < today
```

and the obvious comprehension version

$$\begin{array}{l}[\ (\mathit{name}, \mathit{address}, \mathit{amount}) \\ \mid (\mathit{cid}, \mathit{name}, \mathit{address}) \leftarrow \mathit{customers}, (\mathit{cust}, \mathit{amount}, \mathit{due}) \leftarrow \mathit{invoices}, \\ \ \ \mathit{cid} == \mathit{cust}, \mathit{due} < \mathit{today}\,]\end{array}$$

entails a traversal over the entire cartesian product of the *customers* and *invoices* tables, only subsequently to discard the great majority of pairs of tuples. This is an extremely naive approximation to how database systems implement joins.

In this section, we sketch out how to achieve a more reasonable implementation for joins too, while still preserving the convenient comprehension syntax. We do this only for *equijoins*, that is, for joins where the matching condition on tuples is equality under two functions. The techniques we use are two extensions to comprehension syntax: *parallel* comprehensions (Plasmeijer and van Eekelen, 1995), which introduce a kind of 'zip' operation, and *comprehensive* comprehensions (Wadler and Peyton Jones, 2007), which introduce a 'group by'. Both were introduced originally just for list comprehensions, but have recently (Giorgidze et al., 2011) been generalized to other monads. The result will be a way of writing database queries using joins that can be executed in time linear in the number of input tuples, instead of quadratic.

4.1 Parallel comprehensions

Parallel list comprehensions were introduced in Clean 1.0 in 1995 (Plasmeijer and van Eekelen, 1995), and subsequently as a Haskell extension in GHC around 2001 (GHC 5.0), desugaring to applications of the *zip* function:

$$\begin{aligned}&\mathit{zip} :: [a] \to [b] \to [(a,b)]\\ &\mathit{zip}\ (a:x)\ (b:y) = (a,b) : \mathit{zip}\ x\ y\\ &\mathit{zip}\ \_\qquad\ \ \_\qquad\ \ = [\,]\end{aligned}$$

For example,

$$\begin{aligned}&\ [a+b \mid a \leftarrow [1,2,3] \mid b \leftarrow [4,5]]\\ =&\ \mathit{zip}\ [1,2,3]\ [4,5] \ggg\!= \lambda(a,b) \to \mathit{return}\ (a+b)\\ =&\ [5,7]\end{aligned}$$

Essentially the same translation can be used for an arbitrary monad (Giorgidze et al., 2011):

$$[e \mid (q \mid r), s] = \mathit{mzip}\ [\mathsf{v}^q \mid q]\ [\mathsf{v}^r \mid r] \ggg\!= \lambda(\mathsf{v}^q, \mathsf{v}^r) \to [e \mid s]$$

(where v^q denotes the tuple of variables bound by qualifiers q), provided that the monad supports an appropriate 'zip' function. In the GHC implementation, the appropriate choice of *mzip* function is type-directed: *mzip* is a method of a type class *MonadZip*, a subclass of *Monad*:

$$\begin{aligned}&\mathbf{class}\ \mathit{Monad}\ m \Rightarrow \mathit{MonadZip}\ m\ \mathbf{where}\\ &\quad \mathit{mzip} :: m\ a \to m\ b \to m\ (a,b)\end{aligned}$$

of which the monad in question should be an instance.

There is some uncertainty about what laws one should require of instances of *mzip*, beyond naturality. The GHC documentation specifies only an information preservation requirement: that if two computations x, y have the same 'shape'

$$\mathit{fmap}_M\ (\mathit{const}\ ())\ x = \mathit{fmap}_M\ (\mathit{const}\ ())\ y$$

then zipping them can be undone:

$$(x,y) = \mathbf{let}\ z = \mathit{mzip}\ x\ y\ \mathbf{in}\ (\mathit{fmap}_M\ \mathit{fst}\ z, \mathit{fmap}_M\ \mathit{snd}\ z)$$

Petricek (2011) observes that one probably also wants associativity:

$$\mathit{fmap}_M\ (\lambda(a,(b,c)) \to ((a,b),c))\ (\mathit{mzip}\ x\ (\mathit{mzip}\ y\ z)) = \mathit{mzip}\ (\mathit{mzip}\ x\ y)\ z$$

so that it doesn't matter how one brackets a comprehension $[(a,b,c) \mid a \leftarrow x \mid b \leftarrow y \mid c \leftarrow z]$ with three parallel generators. One might also consider unit and commutativity properties.

4.2 Grouping comprehensions

Grouping list comprehensions were introduced (along with 'ordering') by Wadler and Peyton Jones (2007), specifically motivated by trying to "make it easy to express the kind of queries one would write in SQL". Here is a simple example:

$$\begin{aligned}&[\,(\mathit{the}\ a, b) \mid (a, b) \leftarrow [\,(1, \texttt{'p'}), (2, \texttt{'q'}), (1, \texttt{'r'})\,],\\ &\qquad\qquad\qquad \textbf{then group by}\ a\ \textbf{using}\ \mathit{groupWith}\,]\\ = {}&[\,(1, \texttt{"pr"}), (2, \texttt{"q"})\,]\end{aligned}$$

Here, $\mathit{the} :: [\,a\,] \rightarrow a$ returns the common value of a non-empty list of equal elements, and $\mathit{groupWith} :: \mathit{Ord}\ b \Rightarrow (a \rightarrow b) \rightarrow [\,a\,] \rightarrow [\,[\,a\,]\,]$ groups a list into sublists by some ordered key. The comprehension desugars to

$$\begin{aligned}&\mathit{groupWith}\ (\lambda(a, b) \rightarrow a)\ [\,(1, \texttt{'p'}), (2, \texttt{'q'}), (1, \texttt{'r'})\,] \ggg\!= \lambda \mathit{abs} \rightarrow\\ &\quad \textbf{case}\ (\mathit{map}\ \mathit{fst}\ \mathit{abs}, \mathit{map}\ \mathit{snd}\ \mathit{abs})\ \textbf{of}\ (a, b) \rightarrow \mathit{return}\ (\mathit{the}\ a, b)\end{aligned}$$

The full translation is given below, but in a nutshell, the input list of a, b pairs $[\,(1, \texttt{'p'}), (2, \texttt{'q'}), (1, \texttt{'r'})\,]$ is grouped according to their a component, and then for each group we return the common a component and the list b of corresponding characters. Note in particular the ingenious trick, that the **group** qualifier rebinds the previously bound variables $a :: \mathit{Int}$ and $b :: \mathit{Char}$ as lists $a :: [\,\mathit{Int}\,]$ and $b :: [\,\mathit{Char}\,]$—so of *different types*. (Suzuki et al. (2016) point out that this ingenious trick can backfire: it would be a runtime error to try to compute $\mathit{the}\ b$ in the above query, because the b collection generally will not have all elements equal. They present a more sophisticated embedding of queries that turns this mistake into a type error.)

As with parallel comprehensions, grouping too generalizes to other monads, provided that they support an appropriate ‘group’ function. In Giorgidze et al.’s formulation (2011), this is again specified via a subclass of *Monad*:

$$\begin{aligned}&\textbf{class}\ \mathit{Monad}\ m \Rightarrow \mathit{MonadGroup}\ m\ b\ \textbf{where}\\ &\quad \mathit{mgroupWith} :: (a \rightarrow b) \rightarrow m\ a \rightarrow m\ (m\ a)\end{aligned}$$

(so any given *mgroupWith* method is polymorphic in the a, but for fixed m and b). However, this type turns out to be unnecessarily restricted: there’s no reason why the inner type constructor of the result, the type of each group, needs to coincide with the outer type constructor, the type of the collection of groups; in fact, all that the translation seems to require is

$$\mathit{mgroupWith} :: (a \rightarrow b) \rightarrow m\ a \rightarrow m\ (n\ a)$$

with n some *Functor*. One could add n as another type class parameter; but the current GHC implementation (GHC 7.10) instead dispenses with the *MonadGroup* type class altogether, and requires the grouping function to be explicitly specified. Let us call this variant “heterogeneous grouping”. The translation is then:

$$\begin{aligned}&[\,e \mid q, \textbf{then group by}\ b\ \textbf{using}\ f, r\,]\\ &\quad = f\ (\lambda \mathsf{v}^q \rightarrow b)\ [\,\mathsf{v}^q \mid q\,] \ggg\!= \lambda ys \rightarrow\\ &\qquad \textbf{case}\ (\mathit{fmap}\ \mathsf{v}^q_1\ ys, ..., \mathit{fmap}\ \mathsf{v}^q_n\ ys)\ \textbf{of}\ \mathsf{v}^q \rightarrow [\,e \mid r\,]\end{aligned}$$

where, as before, v^q denotes the tuple of variables bound by qualifiers q, and in addition v^q_i projects out the ith component from this tuple. We make crucial use below of this extra generality. (GHC also provides a simpler variant omitting the “**by** b” clause, but we don’t need it for this paper.)

It is not clear what laws we might expect of the grouping function f, at least partly because of its rather general type; moreover, because it is no longer associated with a type class, it is not clear where one ought morally to attach those laws. But for Giorgidze *et al.*'s homogeneous formulation, it is tempting to think of *mgroupWith f* as a partitioning function, which suggests

$$mult \cdot mgroupWith\ f = id$$

(which doesn't hold of the function *groupWith* on lists, since this reorders elements; that would be appropriate behaviour for bags, and Haskell's *groupBy* function, a pre-inverse of *concat*, more appropriate for lists). One might also want it to partition as finely as possible, so that subsequent grouping is redundant:

$$fmap_M\ (mgroupWith\ f) \cdot mgroupWith\ f = fmap_M\ return \cdot mgroupWith\ f$$

which would exclude size-based partitions, such as splitting a list of length n^2 into n lists of length n.

4.3 Relational tables

The implementation that we will present of relational joins using comprehensions will depend crucially on the interplay between *two different monads*; that's why we need the extra generality of heterogeneous grouping described above. One of these is the *Bag* monad—that is, a ringad in which $\uplus$ is also commutative. The other is the *Table* monad, which to a first approximation (not yet accommodating aggregation—we turn to that question in Section 4.4) is simply a combination of the *Reader* and *Bag* monads:

```
type Table k v = Reader k (Bag v)
```

Readers are essentially just functions; in Haskell, there are a few type wrappers involved too, but we can hide these via the following isomorphism:

```
apply    :: Reader k v → (k → v)
tabulate :: (k → v) → Reader k v
```

Consider the canonical equijoin of two bags x, y by the two functions f, g, specified by

$$equijoin\ f\ g\ x\ y = [(a, b) \mid a \leftarrow x, b \leftarrow y, f\ a == g\ b]$$

We can read this straightforwardly in the list monad; but as we have already observed, this leads to a very inefficient implementation. Instead, we want to be able to index the two input collections by their keys, and zip the two indices together. What else can we zip, apart from lists? *Reader*s are the obvious instance:

```
instance MonadZip (Reader k) where
  mzip x y = tabulate (λk → (apply x k, apply y k))
```

Plain *Reader*s aren't ringads; but *Table*s are, inheriting the ringad structure of *Bag*s:

$$\begin{aligned} \varnothing &= \mathit{tabulate}\ (\lambda k \rightarrow \varnothing) \\ x \uplus y &= \mathit{tabulate}\ (\lambda k \rightarrow \mathit{apply}\ x\ k \uplus \mathit{apply}\ y\ k) \end{aligned}$$

Note that *Bag*s alone won't do, because they don't support zip, only cartesian product.

For grouping, we will use a function

$$\mathit{indexBy} :: \mathit{Eq}\ k \Rightarrow (v \rightarrow k) \rightarrow \mathit{Bag}\ v \rightarrow \mathit{Table}\ k\ v$$

that partitions a bag of values by some key, collecting together subbags with a common key; with a careful choice of representation of the equality condition, this can be computed in linear time (Henglein and Larsen, 2010). We will also use its postinverse

$$\mathit{flatten} :: \mathit{Table}\ k\ v \rightarrow \mathit{Bag}\ v$$

that flattens the partitioning. We need bags rather than lists, because *indexBy* reorders elements; the equation

$$\mathit{flatten} \cdot \mathit{indexBy}\ f = \mathit{id}$$

holds for bags, but would be hard to satisfy had we used lists instead.

With these ingredients, we can capture the equijoin using comprehensions:

$$\begin{aligned} &\mathit{equijoin}\ f\ g\ x\ y \\ &\quad = \mathit{flatten}\ [\mathit{cp}\ (a, b) \mid a \leftarrow x, \textbf{then group by}\ f\ a\ \textbf{using}\ \mathit{indexBy} \\ &\qquad\qquad\qquad\qquad\quad \mid b \leftarrow y, \textbf{then group by}\ g\ b\ \textbf{using}\ \mathit{indexBy}] \end{aligned}$$

which desugars to

$$\mathit{flatten}\ (\mathit{fmap}\ \mathit{cp}\ (\mathit{mzip}\ (\mathit{indexBy}\ f\ x)\ (\mathit{indexBy}\ g\ y)))$$

Informally, this indexes the two bags as tables by the key on which they will be joined, zips the two tables, computes small cartesian products for subbags with matching keys, then discards the index. In the common case that one of the two functions f, g extracts a primary key, the corresponding subbags will be singletons and so the cartesian products are trivial. Better still, one need not necessarily perform the *cp*s and *flatten* immediately; stopping with

$$\begin{aligned} &[(a, b) \mid a \leftarrow x, \textbf{then group by}\ f\ a\ \textbf{using}\ \mathit{indexBy} \\ &\qquad\quad \mid b \leftarrow y, \textbf{then group by}\ g\ b\ \textbf{using}\ \mathit{indexBy}] \end{aligned}$$

yields a table of pairs of subbags, and with care the cartesian products may never need actually to be expanded (Henglein and Larsen, 2010).

4.4 Finite maps

If we want to retain the ability to compute aggregations over tables—and we do, not least in order to define *flatten*—then we must restrict attention to *finite* maps, disallowing infinite ones. Then we can equip tables also with a mechanism to support traversal over the keys in the domain; so they should not simply be

*Reader*s, they should also be paired with some traversal mechanism. (This is the 'refinement' mentioned earlier.) However, there is a catch: if we limit ourselves to finite maps, we can no longer define *return* for *Table*s or *Reader*s—*return a* yields an infinite map, when the key type itself is infinite.

Two possible solutions present themselves. One is to live without the *return* for *Table*, leaving what is sometimes called a *semi-monad* (but not obviously in the same sense as Fernandez et al. (2001)) or *non-unital monad*, that is, a monad with a *mult* but no unit. This seems to be feasible, while still retaining "semi-monad comprehensions"; we only really use *return* in the common base case $[\,e \mid\,]$ of comprehensions with an empty sequence of qualifiers, and this is mostly only for convenience of definition anyway (as noted already, it's not actually valid Haskell syntax). We instead have to provide separate base cases for each singleton sequence of qualifiers; we can't use a bare guard $[\,e \mid b\,]$, and we have to define comprehensions with guards and other qualifiers by

$$[\,e \mid b, q\,] = \mathbf{if}\ b\ \mathbf{then}\ [\,e \mid q\,]\ \mathbf{else}\ \varnothing$$

A second solution is to generalize from plain monads to what have variously been called *indexed monads* (Orchard et al., 2014), *parametric monads* (Katsumata, 2014), or *graded monads* (Milius et al., 2015; Orchard and Yoshida, 2016; Fujii et al., 2016); we use the latter term. In a graded monad $(M, \mathit{return}, \ggg\!=)$ over a monoid $(I, \varepsilon, \otimes)$, the functor M has a type index drawn from the monoid I as well as the usual polymorphic type parameter; *return* yields a result at the unit index ε, and $\ggg\!=$ combines indices using $\otimes$:

$$\begin{aligned} &\mathit{return} :: a \to M\ \varepsilon\ a \\ &(\ggg\!=)\ \ :: M\ i\ a \to (a \to M\ j\ b) \to M\ (i \otimes j)\ b \end{aligned}$$

A familiar example is given by vectors. Vectors of a fixed length—say, vectors of length 3—form a monad, in which *return* replicates its argument and *mult* takes the diagonal of a square matrix. Vectors of different lengths, however, form not a simple monad but a graded monad, over the monoid $(\mathit{Nat}, 1, \times)$ of natural numbers with multiplication: *return* yields a singleton vector, and *mult* flattens an i-vector of j-vectors into a single $(i \times j)$-vector. Technically, a graded monad is no longer simply a monad; but it is still essentially a monad—it has the same kind of categorical structure (Fujii et al., 2016), supports the same operations, and can even still be used with monad comprehensions and **do** notation in Haskell, by using GHC's *RebindableSyntax* extension (Williams, 2014). For our purposes, we want the monoid of finite sequences of finite types, using concatenation $+\!\!+$ and the empty sequence $\langle\rangle$, so that we can define

$$\begin{aligned} &\mathit{return} :: a \to \mathit{Table}\ \langle\rangle\ a \\ &(\ggg\!=)\ \ :: \mathit{Table}\ k\ a \to (a \to \mathit{Table}\ k'\ b) \to \mathit{Table}\ (k +\!\!+ k')\ b \end{aligned}$$

Now *return* yields a singleton table, which is a finite map.

5 Conclusions

We have explored the list comprehension notation from languages like Haskell, its generalization to arbitrary monads, and the special case of collection monads or 'ringads' that support aggregation; and we have shown how to use GHC's parallel and grouping constructs for monad comprehensions to express relational join accurately and efficiently. All of these ingredients owe their genesis or their popularization to Phil Wadler's work; we have merely drawn them together.

Having said all that, writing this paper was still a voyage of discovery. The laws that one should require for *MonadPlus* and *MonadZero* are a subject of great debate; arguably, one expects different laws for backtracking search, for nondeterministic enumeration, and for collections of results—despite all conforming to the same interface. The matter is not settled definitely here (and perhaps the waters have been muddied a bit further—should we insist on associativity of $\uplus$, as some other authors do?).

Wadler's unpublished note (Wadler, 1990) on ringads is not currently widely available; it is apparently not online anywhere, and even Phil himself does not have a copy (Wadler, 2011)—for a long time I thought it had been lost entirely to history, despite being relatively widely cited. However, I am happy to report that Eerke Boiten had preserved a copy, and the document has now been secured.

Trinder's work with Wadler (Trinder and Wadler, 1989; Trinder, 1991) on using list comprehensions for database queries had quite an impact at the time, but they weren't the first to make the connection: Nikhil (1990), Poulovassilis (1988), and Breuer (1989) at least had already done so.

Wadler and Peyton Jones (2007, §3.8) mention parallel list comprehensions, but don't connect them to grouping or to relational join; they write that "because of the generality of these new constructs, we wonder whether they might also constructively feed back into the design of new database programming languages". We hope that we have provided here more evidence that they could.

Acknowledgements

I am very grateful to Ralf Hinze, Nicolas Wu, Fritz Henglein, Tarmo Uustalu, and Oleg Kiselyov for their helpful discussions, to Eerke Boiten for finding a copy of Wadler's note (Wadler, 1990), to Shin-ya Katsumata for sharing unpublished work (Fujii et al., 2016) on graded monads, and to the anonymous reviewers for constructive criticism. This work was partially supported by EPSRC grants *Unifying Theories of Generic Programming* (EP/J010995/1) and *A Theory of Least Change for Bidirectional Transformations* (EP/K020919/1).

Bibliography

Roland Backhouse. An exploration of the Bird-Meertens formalism. Technical Report CS 8810, Department of Computer Science, Groningen University, 1988. `http://www.cs.nott.ac.uk/~psarb2/papers/abstract.html#exploration`.

Eerke Boiten and Paul Hoogendijk. A database calculus based on strong monads and partial functions. Submitted to DBPL, March 1995.

Eerke Boiten and Paul Hoogendijk. Nested collections and polytypism. Technical Report 96/17, Eindhoven, 1996.

Peter T. Breuer. Applicative query languages. *University Computing, the Universities and Colleges Information Systems Association of the UK (UCISA) Bulletin of Academic Computing and Information Systems*, 1989. `https://www.academia.edu/2499641/Applicative_Query_Languages`.

Peter Buneman, Shamim Navqi, Val Tannen, and Limsoon Wong. Principles of programming with collections and complex object types. *Theoretical Computer Science*, 149(1):3–48, 1995.

Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. Links: Web programming without tiers. In *Formal Methods for Components and Objects (FMCO)*, volume 4709 of *LNCS*. Springer, 2006.

Leonidas Fegaras and David Maier. Optimizing object queries using an effective calculus. *ACM Transactions on Database Systems*, 25(4):457–516, December 2000. doi: 10.1145/377674.377676.

Mary Fernandez, Jérôme Simeon, and Philip Wadler. A semi-monad for semistructured data. In *International Conference on Database Theory*, pages 263–300, 2001.

Maarten M. Fokkinga. Tupling and mutumorphisms. *The Squiggolist*, 1(4): 81–82, June 1990.

Soichiro Fujii, Shin-ya Katsumata, and Paul-André Melliès. Towards a formal theory of graded monads. In *Foundations of Software Science and Computation Structures*, Lecture Notes in Computer Science. Springer-Verlag, 2016.

GHC 5.0. *Glasgow Haskell Compiler Users' Guide, Version 5.00*, April 2001. `https://downloads.haskell.org/~ghc/5.00/docs/set/book-users-guide.html`.

GHC 7.10. *Glasgow Haskell Compiler Users' Guide, Version 7.10*, July 2015. `https://downloads.haskell.org/~ghc/latest/docs/html/users_guide/`.

George Giorgidze, Torsten Grust, Nils Schweinsberg, and Jeroen Weijers. Bringing back monad comprehensions. In *Haskell Symposium*, pages 13–22, 2011.

Torsten Grust. *Comprehending Queries.* PhD thesis, Universität Konstanz, 1999.

Haskell 2010. *Haskell 2010 Language Report*, April 2010. `https://www.haskell.org/onlinereport/haskell2010/`.

Fritz Henglein and Ken Friis Larsen. Generic multiset programming with discrimination-based joins and symbolic Cartesian products. *Higher-Order and Symbolic Computation*, 23(3):337–370, 2010. doi: 10.1007/s10990-011-9078-8.

Shin-ya Katsumata. Parametric effect monads and semantics of effect systems. In *Principles of Programming Languages*, pages 633–645, 2014.

Oleg Kiselyov. Laws of MonadPlus. `http://okmij.org/ftp/Computation/monads.html#monadplus`, January 2015.

David Luposchainsky. MonadFail proposal. `https://github.com/quchen/articles/blob/master/monad_fail.md`, June 2015.

Erik Meijer. The world according to LINQ. *Communications of the ACM*, 54 (10):45–51, 2011.

Stefan Milius, Dirk Pattinson, and Lutz Schröder. Generic trace semantics and graded monads. In Larry Moss and Pawel Sobocinski, editors, *6th International Conference on Algebra and Coalgebra in Computer Science (CALCO'15)*, pages 251–266, 2015.

Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.

Rishiyur S. Nikhil. The semantics of update in a functional database programming language. In François Bancilhon and Peter Buneman, editors, *Advances in Database Programming Languages (DBPL-1, 1987)*, pages 403–421. ACM Press / Addison-Wesley, 1990.

Dominic Orchard and Nobuko Yoshida. Effects as sessions, sessions as effects. In *Principles of Programming Languages*, 2016.

Dominic Orchard, Tomas Petricek, and Alan Mycroft. The semantic marriage of monads and effects. arXiv:1401.5391, 2014.

Tomas Petricek. Fun with parallel monad comprehensions. *The Monad Reader*, (18), July 2011. `https://themonadreader.wordpress.com/2011/07/05/issue-18/`.

Simon Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice Hall, 1987.

Rinus Plasmeijer and Marko van Eekelen. Concurrent Clean language report (version 1.0). Technical report, University of Nijmegen, 1995. `ftp://ftp.science.ru.nl/pub/Clean/old/Clean10/doc/refman.ps.gz`.

Alexandra Poulovassilis. FDL: an integration of the functional data model and the functional computational model. In *British National Conference on Databases*, pages 215–236, 1988.

Jack T. Schwartz, Robert B.K. Dewar, Edward Dubinsky, and Edmond Schonberg. *Programming with Sets: An Introduction to SETL*. Springer, New York, 1986.

Jacob T. Schwartz. On programming: An interim report on the SETL project. Technical report, Courant Institute of Mathematical Sciences, New York University, June 1975.

Dan Suciu. Fixpoints and bounded fixpoints for complex objects. Technical Report MS-CIS-93-32, University of Pennsylvania, 1993a.

Dan Suciu. Queries on databases with user-defined functions. Technical Report MS-CIS-93-62, University of Pennsylvania, 1993b.

Kenichi Suzuki, Oleg Kiselyov, and Yukiyoshi Kameyama. Finally, safely-extensible and efficient language-integrated query. In *Partial Evaluation and Program Manipulation*, 2016.

Phil Trinder. Comprehensions: A query notation for DBPLs. In *Database Programming Languages*, 1991.

Phil Trinder and Philip Wadler. Improving list comprehension database queries. In *TENCON'89: Fourth IEEE Region 10 International Conference*. IEEE, 1989.

Tarmo Uustalu. A divertimento on MonadPlus and nondeterminism. *Journal of Logical and Algebraic Methods in Programming*, to appear 2015. Special issue in honour of José Nuno Oliveira's 60th birthday. Extended abstract in HOPE 2015.

Philip Wadler. Notes on monads and ringads. Internal document, CS Department, University of Glasgow, September 1990.

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

Philip Wadler. Monads for functional programming. In Manfred Broy, editor, *Marktoberdorf Summer School on Program Design Calculi*, volume 118 of *NATO ASI Series F: Computer and Systems Sciences*. Springer Verlag, August 1992b. Also in J. Jeuring and E. Meijer, editors, Advanced Functional Programming, Springer Verlag, LNCS 925, 1995.

Philip Wadler. Laws for monads with zero and plus. Post to Haskell Mailing List, May 1997.

Philip Wadler. Monads and ringads. Personal communication, August 2011.

Philip Wadler and Simon Peyton Jones. Comprehensive comprehensions: Comprehensions with 'order by' and 'group by'. In *Haskell Symposium*, pages 61–72, 2007.

David Watt and Phil Trinder. Towards a theory of bulk types. FIDE technical report 91/26, Glasgow University, July 1991.

Tim Williams. Map comprehensions. `http://www.timphilipwilliams.com/posts/2014-06-05-map-comprehensions.html`, June 2014.

Limsoon Wong. *Querying Nested Collections*. PhD thesis, University of Pennsylvania, 1994.

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