Comprehending Ringads

Jeremy Gibbons
WadlerFest, April 2016
1. Comprehensions

- ZF axiom schema of specification:
  \[ \{ x^2 \mid x \in Nat \land x < 10 \land x \text{ is even} \} \]

- SETL set-formers:
  \[ \{ x \times x \mid x \in \{0..9\}, \text{mod } x \text{ mod } 2 = 0 \} \]

- Eindhoven Quantifier Notation:
  \[(x : 0 \leq x < 10 \land x \text{ is even} : x^2)\]

- Haskell (NPL, Python, ...) list comprehensions:
  \[ [x^2 \mid x \leftarrow [0..9], \text{even } x] \]
2. Comprehending monads (Wadler, 1992)

Structure is that of a monad with zero \((T, \text{return}, \text{mult}, \emptyset)\):

\[
\text{mult} :: T (T a) \rightarrow T a \\
\text{mult} (\text{mult} xss) = \text{mult} (\text{fmap} \text{mult} xss) \\
\text{return} :: a \rightarrow T a \\
\text{mult} (\text{return} x) = x \\
\text{mzero} :: T a \\
\text{mult} \emptyset = \emptyset
\]

Comprehensions can then be used for any monad-with-zero:

\[
\mathcal{D} [ e \mid ] = \text{return} e \\
\mathcal{D} [ e \mid p \leftarrow e', Q ] = e' \Rightarrow \lambda p \rightarrow \mathcal{D} [ e \mid Q ] \\
\mathcal{D} [ e \mid e', Q ] = \text{guard} e' \Rightarrow \lambda() \rightarrow \mathcal{D} [ e \mid Q ] \\
\mathcal{D} [ e \mid \text{let} d, Q ] = \text{let} d \text{ in } \mathcal{D} [ e \mid Q ]
\]

(where \(x \Rightarrow k = \text{mult} (\text{fmap} k x)\) and \(\text{guard} b = \text{if } b \text{ then return } () \text{ else } \emptyset\)).

Hence monad comprehensions for sets, bags, (sub-)distributions, exceptions...
3. Collection monads

Finite *collection* types are monads. But the operations of a monad-with-zero cannot introduce multiplicity; need also

\[(\uplus) : T a \to T a \to T a\]

such that

\[
\begin{align*}
mult (xs \uplus ys) &= (mult xs) \uplus (mult ys) \\
x \uplus \emptyset &= x \\
\emptyset \uplus y &= y
\end{align*}
\]

*Sets, bags, sub-distributions* are collection monads; but *exceptions* are not.

Eg the *Boom Hierarchy*: trees

| lists | \(\uplus\) is associative |
|bags | \(\ldots\) and commutative |
|sets | \(\ldots\) and idempotent |
4. Aggregations

Well-behaved operations $h$ over collections: \textit{count}, \textit{sum}, \textit{some}, \ldots

\[
h (\text{return } a) = a \\
h (\text{mult } xs) = h (\text{fmap } h \; xs)
\]

—the \textit{algebras} for the monad $T$.

Define $\odot$ and $\varepsilon$ by

\[
a \odot b = h (\text{return } a \odot \text{return } b) \\
\varepsilon = h \emptyset
\]

Then

\[
h (x \odot y) = h x \odot h y
\]

Moreover, $\odot$ and $\varepsilon$ satisfy whatever laws $\cup$ and $\emptyset$ do:

$\varepsilon$ is the unit of $\odot$; $\odot$ is associative if $\cup$ is; etc (at least on the range of $h$).
5. Comprehending queries

Wadler & Trinder (1991) argued for comprehensions as a query notation: Given input tables

- customers :: Bag (CID, Name, Address)
- invoices :: Bag (IID, CID, Amount, Date)

then

\[
\text{overdueInvoices} = \left[ (c.\text{name}, c.\text{address}, i.\text{amount}) \mid c \leftarrow \text{customers}, \\
i \leftarrow \text{invoices}, i.\text{due} < \text{today}, \\
c.\text{cid} == i.\text{customer} \right]
\]

Works similarly in any collection monad, not just bags.

An influential observation in the DBPL community: basis of languages such as Buneman’s Kleisli, Microsoft LINQ, Wadler’s Links, as well as querying for objects (OQL) and XML (XQuery).
6. The problem with joins

The comprehension yields a terrible query plan!
Constructs entire cartesian product, then discards most of it:

\[
\text{fmap } (\lambda(c, i) \rightarrow (c.\text{name}, c.\text{address}, i.\text{amount})) (\\text{filter } (\lambda(c, i) \rightarrow c.\text{cid} == i.\text{customer})) (\\text{filter } (\lambda(c, i) \rightarrow i.\text{due} < \text{today})) (\\text{cp customers invoices}))
\]

Better to group by customer identifier, then handle groups separately:

\[
\text{fmap } (\text{fmap } (\lambda c \rightarrow (c.\text{name}, c.\text{address})) \times \text{fmap } (\lambda i \rightarrow i.\text{amount})) (\\text{fmap } (\text{id} \times \text{filter } (\lambda i \rightarrow i.\text{due} < \text{today}))) (\\text{merge } \text{indexBy cid customers, indexBy customer invoices}))
\]

(where \text{indexBy} partitions, and \text{merge} pairs on common index).
But this doesn’t correspond to anything expressible in comprehensions.
7. Comprehensive comprehensions

Parallel (‘zip’) comprehensions (Clean 1.0, 1995):

\[
[ (x, y) | x \leftarrow [1, 2, 3] | y \leftarrow [4, 5, 6] ] = [(1, 4), (2, 5), (3, 6)]
\]

‘Order by’ and ‘group by’ (Wadler & Peyton Jones, 2007):

\[
[ (\text{the dept}, \text{sum salary})
| (\text{name, dept, salary}) \leftarrow \text{employees}
, \text{ then group by dept using groupWith}
, \text{ then sortWith by sum salary} ]
\]

(NB group by rebinds the variables salary etc bound earlier!)

Initially just for lists, but also generalizable (Giorgidze et al., 2011):

\[
mzip_T :: T \ a \rightarrow T \ b \rightarrow T \ (a, b)
mgroupWith_{T,U,F} :: Eq \ b \Rightarrow (a \rightarrow b) \rightarrow T \ a \rightarrow U \ (F \ a)
\]

(Note heterogeneous type: T, U should be monads, F a functor.)
8. Solving the problem with (equi-)joins

Maps-to-bags form a monad-with-zero—roughly:

\[
\textbf{type } \text{Map } k \; \nu \; = \; k \rightarrow \nu \\
\textbf{type } \text{Table } k \; \nu \; = \; \text{Map } k \; (\text{Bag } \nu)
\]

Now define

\[
\textbf{merge } :: (\text{Table } k \; \nu, \text{Table } k \; \omega) \rightarrow \text{Table } k \; (\nu, \omega) \\
\text{merge } (f, g) = \lambda k \rightarrow \text{cp } (f \; k) (g \; k)
\]

\[
\textbf{indexBy } :: \text{Eq } k \Rightarrow (\nu \rightarrow k) \rightarrow \text{Bag } \nu \rightarrow \text{Table } k \; \nu \\
\text{indexBy } f \; xs \; k = \text{filter } (\lambda \nu \rightarrow f \; \nu == \; k) \; xs
\]

and use \textbf{merge} for zipping, \textbf{indexBy} for grouping.

With care, \textbf{indexBy} can be evaluated in linear time.
Now represent query as:

\[
\text{overdueInvoices} :: \text{Map Int (Name, Address, Bag Amount)}
\]

\[
\text{overdueInvoices} = \left[ \text{(the name, the addr, amount)} \right.
\]
\[
\left| \text{(cid, name, addr)} \leftarrow \text{customers} \right.
\]
\[
\left| \text{then group by cid using indexBy} \right.
\]
\[
\left| \text{(iid, customer, amount, due)} \leftarrow \text{invoices} \right.
\]
\[
\left| \text{due < today} \right.
\]
\[
\left| \text{then group by customer using indexBy} \right]
\]

Avoids expanding the whole cartesian product.
9. Finite maps

A catch:

- need *monads*, for comprehensions
- need *Maps*, for indexing
- need *finite collections*, for aggregation
- but *finite maps* don’t form a monad (no *return*)

Solution?
10. Graded monads (Katsumata et al, 2016)

Monad \((T, \text{return}, \text{mult})\) has endofunctor \(T : C \to C\), polymorphic functions

\[
\begin{align*}
\text{return} & : a \to T a \\
\text{mult} & : T (T a) \to T a
\end{align*}
\]
satisfying certain laws.

\(M\)-graded monad \((T, \text{return}, \text{mult})\) for monoid \((M, \varepsilon, \cdot)\) has (non-endo-)functor \(T : M \to [C, C]\) and

\[
\begin{align*}
\text{return} & : a \to T \varepsilon a \\
\text{mult} & : T m (T n a) \to T (m \cdot n) a
\end{align*}
\]
with same laws. (Eg for collecting \textit{effects}; think also of \textit{vectors}.)

We use \(T = Table\), with monoid \((K, \langle \rangle, \oplus)\) of finite type sequences. Not an endofunctor, but there is still a story involving adjunctions.
11. Ringads (Wadler, 1990)

Wadler called collection monads *ringads*.

Ringads are (roughly) *right near-semirings* in the right near-semiringy category of endofunctors under composition and product.

Wadler’s note cited in numerous papers from the 1990s (with varying degrees of accuracy), but long thought lost...
12. Conclusions: Comprehending Wadler

- list comprehensions
- monads for functional programming
- monad comprehensions and do notation
- comprehensions for queries
- comprehensive comprehensions
- graded monads for the marriage with effects

- thank you, Phil!