Bringing Back Monad Comprehensions and Extending Database Supported Haskell

George Giorgidze

University of Tübingen

(joint work at various times with Torsten Grust, Andres Löh, Tom Schreiber, Nils Schweinsberg, Alexander Ulrich, and Jeroen Weijers.)

Functional Programming Laboratory Seminar
Nottingham, England
2012 MAR 02
This talk

- Bringing back monad comprehensions [Haskell 2011]
- Extending Database Supported Haskell (DSH) [IFL 2010 + ongoing work]
Reddit celebration

Monad comprehensions are back in GHC
Outline

▶ History
  ▶ Inception (Wadler 1992)
  ▶ Rise (Haskell 1.4)
  ▶ Fall (Haskell Workshop 1997)

▶ Story
  ▶ Haskell extension available in GHC-7.2
  ▶ Generators and filters as in Haskell 1.4
  ▶ SQL-like comprehensions (Wadler and Peyton Jones 2007)
  ▶ Parallel/zip comprehensions (GHC and Hugs)

▶ Future
\[ [x | x \leftarrow ys, x < y] \]

\[
\text{concat (map (\lambda x \rightarrow map (\lambda() \rightarrow x))}
\]

\[
(\text{if (x < y) then [()] else []})
\]

\[
ys
\]

\[
\text{join (liftM (\lambda x \rightarrow liftM (\lambda() \rightarrow x))}
\]

\[
(\text{guard (x < y)})
\]

\[
ys
\]
“generators of the form \( p \leftarrow e \), where \( p \) is a pattern of type \( t \) and \( e \) is an expression of type \( \text{Monad } m \Rightarrow m \; t \)”
“The problem is that if list operations, and especially list comprehensions, are overloaded, then in some programs the overloading will be ambiguous.”

“... the compiler rejects the program, with an error message along the lines of ambiguous type variable in class Monad”

“Imagine the first year student, taking the first programming course, who is struggling to understand list programming and recursion, and is suddenly faced with the error message above!”

“. . . we took a show of hands on it at the Haskell workshop. 90% voted to restrict comprehensions to lists.”
ACM SIGPLAN Workshop Program

Haskell Workshop

Held in conjunction with ICFP97

Amsterdam, The Netherlands

Saturday, June 7, 1997
Motivation

- The comprehension notation is concise and expressive.
- But it only works for lists in Haskell.
- List is not always the best choice.
  - Performance
  - Memory
  - Strictness
  - Parallelism
- Monad comprehensions would be useful for libraries and EDSLs (especially for collection-based ones).
Database Supported Haskell (DSH)

- Database-executable combinators:

  \[ \text{map} :: (QA \ a, QA \ b) \Rightarrow (Q \ a \rightarrow Q \ b) \rightarrow Q \ [a] \rightarrow Q \ [b] \]
  \[ \text{filter} :: (QA \ a) \Rightarrow (Q \ a \rightarrow Q \ \text{Bool}) \rightarrow Q \ [a] \rightarrow Q \ [a] \]
  \[ \text{concat} :: (QA \ a) \Rightarrow Q \ [[[a]]] \rightarrow Q \ [a] \]
  \[ \text{zip} :: (QA \ a, QA \ b) \Rightarrow Q \ [a] \rightarrow Q \ [b] \rightarrow Q \ [(a, b)] \]
  ...

- Run an *existing* Haskell program on large-scale data (e.g., larger than the available memory).

- Query *existing* data in a high-level functional language with nested and ordered collections instead of a low-level relational language with flat collections.

- *Haskell Boards the Ferry (IFL 2010)*

- cabal install DSH
Database Supported Haskell (DSH)

- Database-executable combinators:

  - `map :: (Q a → Q b) → Q [a] → Q [b]`
  - `filter :: (Q a → Q Bool) → Q [a] → Q [a]`
  - `concat :: Q [[a]] → Q [a]`
  - `zip :: Q [a] → Q [b] → Q [(a, b)]`
  - ...

- Reinventing the wheel as a quasiquoter:

  ```haskell
  [qc | x | x ← ys, x < y ]
  ```
Accident

\[ [qc \mid x \mid x \leftarrow ys, x < y \mid] \]
Accident

\[ \{ q \mid x \mid x \leftarrow ys, x < y \} \]
Examples

\[
\begin{align*}
\texttt{quickSort} & \quad :: \text{Ord } a \Rightarrow [a] \rightarrow [a] \\
\texttt{quickSort } ys & \\
& \quad | \text{null } ys = \text{mzero} \\
& \quad | \text{otherwise} = \text{quickSort } [x | x \leftarrow ys, x < y] \text{`mplus`} \\
& \quad \quad [x | x \leftarrow ys, x \equiv y] \text{`mplus`} \\
& \quad \quad \text{quickSort } [x | x \leftarrow ys, x > y] \\
\text{where} & \\
& \quad y = \text{head } ys
\end{align*}
\]
Examples

quickSort :: Ord a ⇒ [a] → [a]
quickSort ys
  | null ys   = mzero
  | otherwise = quickSort [x | x ← ys, x < y] `mplus`
                   [x | x ← ys, x ≡ y] `mplus`
                   quickSort [x | x ← ys, x > y]

  where
    y = head ys

{-# LANGUAGE MonadComprehensions #-}

quickSort :: Ord a ⇒ Seq a → Seq a
quickSort :: Ord a ⇒ DLList a → DLList a
quickSort :: Ord a ⇒ ALList a → ALList a

quickSort :: (Ord a, ListLike m a, MonadPlus m) ⇒ m a → m a
Data Parallel Haskell (DPH)

- **Parallel Array Comprehensions:**
  
  $$
  \textit{sparseMul} :: \[(\text{Int}, \text{Float}):] \rightarrow [\text{Float}:] \rightarrow \text{Float}
  \textit{sparseMul sv v} = \text{sumP} [f * (v !: i) | (i, f) \leftarrow sv:]
  $$

- **Monad Comprehensions:**
  
  $$
  \textit{sparseMul} :: \[(\text{Int}, \text{Float}):] \rightarrow [\text{Float}:] \rightarrow \text{Float}
  \textit{sparseMul sv v} = \text{sumP} [f * (v !: i) | (i, f) \leftarrow sv]
  $$
Parallel List Comprehensions

\[
denseMult :: [\text{Float}] \to [\text{Float}] \to [\text{Float}]
denseMult \; xs \; ys = \text{sum} \; [x \ast y \mid x \leftarrow xs \mid y \leftarrow ys]
\]

\[
denseMult :: [\text{Float}] \to [\text{Float}] \to [\text{Float}]
denseMult \; xs \; ys = \text{sum} \; [x \ast y \mid (x, y) \leftarrow \text{zip} \; xs \; ys]
\]
MonadZip

```haskell
class Monad m ⇒ MonadZip m where
    mzip :: m a → m b → m (a, b)
    mzipWith :: (a → b → c) → m a → m b → m c
    munzip :: m (a, b) → (m a, m b)

instance MonadZip [] where
    mzip = zip
    munzip = unzip

instance MonadZip [::] where
    mzip = zipP
    munzip = unzipP
```
Parallel Monad Comprehensions

{-# LANGUAGE MonadComprehensions, ParallelListComp #-}

```haskell

denseMultP :: [:Float:] → [:Float:] → [:Float:]
denseMultP xs ys = sumP [x * y | x ← xs | y ← ys]
```

- Parallel monad comprehension examples in Tomas Petricek’s latest Monad.Reader article, including parallel parsing and parallel evaluation monads.
MonadZip

\[
\text{class } \textit{Monad } m \Rightarrow \textit{MonadZip } m \textit{ where }
\]
\[
mzip :: m \, a \rightarrow m \, b \rightarrow m \, (a, b)
\]
\[
mzip = mzipWith (,)
\]
\[
mzipWith :: (a \rightarrow b \rightarrow c) \rightarrow m \, a \rightarrow m \, b \rightarrow m \, c
\]
\[
mzipWith \, f \, ma \, mb = \text{liftM} \, (\text{uncurry} \, f) \, (mzip \, ma \, mb)
\]
\[
munzip :: m \, (a, b) \rightarrow (m \, a, m \, b)
\]
\[
munzip \, mab = (\text{liftM} \, \text{fst} \, mab, \text{liftM} \, \text{snd} \, mab)
\]
MonadZip Laws

▶ Naturality

\[
\text{liftM } (f \circ \circ \circ g) (mzip ma mb) \\
\equiv mzip (\text{liftM } f ma) (\text{liftM } g mb)
\]

▶ Associativity

\[
\text{liftM } (\lambda (a, (b, c)) \rightarrow ((a, b), c)) (mzip ma (mzip mb mc)) \\
\equiv mzip (mzip ma mb) mc
\]

▶ Information Preservation

\[
\text{liftM } (\text{const } ()) ma = \text{liftM } (\text{const } ()) mb \\
\Rightarrow \text{munzip } (mzip ma mb) \equiv (ma, mb)
\]
SQL-like List Comprehensions

{-# LANGUAGE TransformListComp #-}

employees :: [(String, String, Integer)]
employees = [ ("Simon","MS", 80), ("Erik", "MS", 90),
              ("Phil", "Ed", 40), ("Gordon", "Ed", 45),
              ("Paul", "Yale", 60) ]

query :: [(String, Integer)]
query = [ (the dept, sum salary)
           | (name, dept, salary) ← employees,
             then group by dept,
             then sortWith by (sum salary)
         ]

[ ("Yale", 60), ("Ed", 85), ("MS", 170) ]
query :: [(String, Integer)]
query = [ (the dept, sum salary) |
(name, dept, salary) ← employees
, then group by dept
, then sortWith by (sum salary)
]

map
(λ(_, dept, salary) → (the dept, sum salary))

sortWith
(λ(_, _, salary) → sum salary)
(map (λl → (map (λ(name, _, _) → name) l
, map (λ(_, dept, _) → dept) l
, map (λ(_, _, salary) → salary) l)))
(groupWith (λ(_, dept, _) → dept) employees))
SQL-like Monad Comprehensions

- **Lists**

  \[
  [a] \rightarrow [a] \\
  [a] \rightarrow [[a]]
  \]

- **Monads**

  \[
  \text{Monad } m \Rightarrow m \ a \rightarrow m \ a \\
  \text{Monad } m \Rightarrow m \ a \rightarrow m (m \ a)
  \]
Deriving parallel tree scans
1st March 2011, 12:41 pm

The post Deriving flat scans explored folds and scans on lists and showed how the usual, efficient scan implementations can be derived from simpler specifications.

Let’s see now how to apply the same techniques to scans over trees.

This new post is one of a series leading toward algorithms optimized for execution on massively parallel, consumer hardware, using CUDA or OpenCL.

Edits:

- 2011-03-01: Added clarification about "$a^n$" and "$f(@)$".
- 2011-03-23: corrected "linear-time" to "linear-work" in two places.

Trees
Our trees will be non-empty and binary:

```haskell
data T a = Leaf a | Branch (T a) (T a)
```
Would do do?

\[ [x \mid x \leftarrow xs, x > 0] \]

do \ x \leftarrow xs
  guard (x > 0)
  return x
Would do do?

```
query :: [(String, Integer)]
query = [  (the dept, sum salary)
  |  (name, dept, salary) <- employees
  ,  then group by dept
  ,  then sortWith by (sum salary)
  ]

do let g = do l <- mgroupWith (\(_, dept, _\) -> dept) employees
  return (liftM (\(name, _, _\) -> name) l
  , liftM (\(_, dept, _\) -> dept) l
  , liftM (\(_, _, salary\) -> salary) l)
  (--, dept, salary) <- sortWith (\(_, _, s\) -> sum s) g
  return (the dept, sum salary)
```
A Relational Model of Data for Large Shared Data Banks

E. F. Codd
IBM Research Laboratory, San Jose, California

\[ R \times S = \{ (a, b, c) : R(a, b) \land S(b, c) \} \]

Comprehensions in programming languages

- SETL
- LINQ
- F#
- Erlang
- Python
- Perl 6
- Links
- . . .
Syntax

\[ p, q ::= w \leftarrow e \]
- let \( w = e \)  
- \( g \) 
- \( p, q \) 
- \( p \mid q \) 
- \( q, \text{then } f \) 
- \( q, \text{then } f \text{ by } e \) 
- \( q, \text{then group by } e \) 
- \( q, \text{then group using } f \) 
- \( q, \text{then group by } e \text{ using } f \)
Typing Rules

List comprehensions

\[
P, \Gamma \vdash q : (m, \Delta) \quad \Gamma, \Delta \vdash e : \tau
\]
\[
\{ \text{Monad } m \} \cup P, \Gamma \vdash [e \mid q] : m \tau
\text{[Comp]}
\]

Variables

\[
\vdash x : \tau \Rightarrow \{ x : \tau \}
\text{[Var]}
\]
\[
\vdash w_1 : \tau_1 \Rightarrow \Delta_1 \quad \ldots \quad \vdash w_n : \tau_n \Rightarrow \Delta_n
\]
\[
\vdash (w_1, \ldots, w_n) : (\tau_1, \ldots, \tau_n) \Rightarrow \Delta_1 \cup \ldots \cup \Delta_n
\text{[Tup]}
\]

Basic list comprehension body

\[
\Gamma \vdash e : \text{Bool}
\]
\[
\{ \text{MonadPlus } m \}, \Gamma \vdash e \Rightarrow (m, \emptyset)
\text{[Guard]}
\]
\[
\emptyset, \Gamma \vdash () : (m, \emptyset)
\text{[Unit]}
\]
\[
\Gamma \vdash e : m \tau \vdash w : \tau \Rightarrow \Delta
\text{[Gen]}
\]
\[
\emptyset, \Gamma \vdash w \leftarrow e : (m, \Delta)
\]
\[
\Gamma \vdash p : (m, \Delta)
\]
\[
P, \Gamma \vdash p \Rightarrow (m, \Delta)
\text{[Comma]}
\]
\[
\emptyset, \Gamma \vdash \text{let } x = e : (m, \Delta)
\text{[Let]}
\]

Parallel list comprehension body

\[
P, \Gamma \vdash p : (m, \Delta)
\]
\[
P', \Gamma \cup \Delta \vdash q : (m, \Delta')
\]
\[
\{ \text{MonadZip } m \} \cup P \cup P', \Gamma \vdash p \mid q : (m, \Delta \cup \Delta')
\text{[Bar]}
\]

Comprehensive list comprehension body

\[
P, \Gamma \vdash q : (m, \Delta)
\]
\[
\Gamma \vdash f : \forall \alpha. m \alpha \to m \alpha
\]
\[
P, \Gamma \vdash q, \text{then } f \Rightarrow (m, \Delta)
\text{[then]}
\]
\[
P, \Gamma \vdash q \Rightarrow (m, \Delta)
\]
\[
\Gamma \cup \Delta \vdash e : \tau
\]
\[
P, \Gamma \vdash q \Rightarrow (m, \Delta)
\]
\[
\Gamma \vdash f : \forall \alpha. (\alpha \to \tau) \to m \alpha \to m \alpha
\]
\[
P, \Gamma \vdash q, \text{then by } e \Rightarrow (m, \Delta)
\text{[thenBy]}
\]
\[
P, \Gamma \vdash q \Rightarrow (m, \Delta)
\]
\[
\Gamma \cup \Delta \vdash e : \tau
\]
\[
P \cup \{ \text{MonadGroup } m \}, \Gamma \vdash q, \text{then group by } e \Rightarrow m \Delta
\text{[groupBy]}
\]
\[
P, \Gamma \vdash q \Rightarrow (m, \Delta)
\]
\[
\Gamma \vdash f : \forall \alpha. m \alpha \to m (m \alpha)
\]
\[
P, \Gamma \vdash q, \text{then group using } f \Rightarrow m \Delta
\text{[groupUsing]}
\]
\[
P, \Gamma \vdash q \Rightarrow (m, \Delta)
\]
\[
\Gamma \cup \Delta \vdash e : \tau
\]
\[
P, \Gamma \vdash q \Rightarrow (m, \Delta)
\]
\[
\Gamma \vdash f : \forall \alpha. (\alpha \to \tau) \to m \alpha \to m (m \alpha)
\]
\[
P, \Gamma \vdash q, \text{then by using } f \Rightarrow m \Delta
\text{[groupByUsing]}
\]

The question whether the monad comprehension extension will be incorporated into the Haskell language standard or not depends on several factors. Perhaps the two most important factors are the extension's correctness and its potential for overloading. The correctness of the extension is ensured by the type inference rules presented above, which guarantee that the extension is type-safe. Overloading is a potential concern because it increases the complexity of the type system. However, GHC's type checker and desugaring rules have been reused to handle overloading, and thus the extension is designed to be overloading compatible. Further extensions may be considered, but the existing ones should be fully implemented and tested before considering additional ones. The question of whether the monad comprehension extension will be incorporated into the Haskell language standard is not yet answered.
Desugaring

\[
\begin{align*}
[e \mid q] &= liftM (\lambda q_v \to e)[q] \\
[w ← e] &= e \\
[\text{let } w = d] &= return d \\
[q] &= \text{guard } q \\
[p, q] &= \text{join } (liftM (\lambda p_v \to liftM (\lambda q_v \to (p_v, q_v))[q]))[p] \\
[p \mid q] &= mzip[p][q] \\
[q, \text{then } f] &= f[q] \\
[q, \text{then } f \text{ by } e] &= f (\lambda q_v \to e)[q] \\
[q, \text{then group by } e] &= liftM unzip_{q_v} (mgroupWith (\lambda q_v \to e)[q]) \\
[q, \text{then group by } e \text{ using } f] &= liftM unzip_{q_v} (f (\lambda q_v \to e)[q]) \\
[q, \text{then group using } f] &= (liftM unzip_{q_v} (f[q])) \\
\end{align*}
\]

\[unzip() = id\]  
\[unzip_x = id\]  
\[unzip_{(w_1, w_2)} = \lambda e \to (unzip_{w_1} (liftM (\lambda (x, y) \to x) e) \to (unzip_{w_2} (liftM (\lambda (x, y) \to y) e)))\]
Error Messages

\[(x, y) \mid x \leftarrow [1], y \leftarrow Just\ 5\]

Code/Error.hs:45:30:
Couldn’t match expected type ‘[t0]’
  with actual type ‘Maybe a0’
In the return type of a call of ‘Just’
In a stmt of a monad comprehension: \(y \leftarrow Just\ 5\)
In the expression:
\([(x, y) \mid x \leftarrow [1], y \leftarrow Just\ 5]\)
List notation overloading and defaulting

- Concrete proposal on list notation overloading for collections.
- Not so concrete proposal on extending Haskell’s existing defaulting mechanism (e.g., for backwards compatibility and for resolving type ambiguities).
List notation overloading

    class FromList l where
        type Item l
        fromList :: [Item l] → l

    instance FromList [a] where
        type Item [a] = a
        fromList = id

    instance (Ord a) ⇒ FromList (Set a) where
        type Item (Set a) = a
        fromList = Set.fromList

    instance (Ord k) ⇒ FromList (Map k v) where
        type Item (Map k v) = (k, v)
        fromList = Map.fromList

    instance FromList (IntMap v) where
        type Item (IntMap v) = (Int, v)
        fromList = IntMap.fromList

    instance FromList Text where
        type Item Text = Char
        fromList = Text.pack
Conclusions

- The monad comprehension notation with generators and filters has been brought back to GHC Haskell.
- SQL-like and parallel/zip comprehensions have been generalised to monads and implemented in GHC.
- We welcome feedback from the Haskell community on usability, syntax, typing, implemented generalisation and library additions, and suggested laws.
- GHC is a great platform for development and experimentation with language features.
- We thank Simon for enhancing and integrating the monad comprehensions patch in GHC.
Monad comprehensions are back in GHC

Available in GHC-7.2
Database Supported Haskell (DSH)

- Database-executable combinators:

  \[\text{map} :: (QA \ a, QA \ b) \Rightarrow (Q \ a \rightarrow Q \ b) \rightarrow Q \ [a] \rightarrow Q \ [b]\]
  \[\text{filter} :: (QA \ a) \Rightarrow (Q \ a \rightarrow Q \ \text{Bool}) \rightarrow Q \ [a] \rightarrow Q \ [a]\]
  \[\text{concat} :: (QA \ a) \Rightarrow Q \ [[a]] \rightarrow Q \ [a]\]
  \[\text{zip} :: (QA \ a, QA \ b) \Rightarrow Q \ [a] \rightarrow Q \ [b] \rightarrow Q \ [(a, b)]\]
  ...

- Run an *existing* Haskell program on large-scale data (e.g., larger than the available memory).

- Query *existing* data in a high-level functional language with *nested* and *ordered* collections instead of a low-level relational language with *flat* collections.

- *Haskell Boards the Ferry (IFL 2010)*

- cabal install DSH
Related work

- Language integration:
  - LINQ (Meijer et al.)
  - Links (Wadler et al.)

- Implementation:
  - Flattening transformation in NESL and Data Parallel Haskell (DPH)
DSH features

- Order and nesting
- Types determine the number of generated queries

Supported types:

\[ t ::= () \]
| Bool
| Char
| Integer
| Double
| Text
| \((t, \ldots, t)\)
| \([t]\)
Ongoing work

- Algebraic data types through generic deriving of $QA$ instances
- Number of generated queries in the extended DSH
- Optimising queries in the extended DSH
Questions/Comments/Suggestions