Bringing Back Monad Comprehensions and Extending Database Supported Haskell

George Giorgidze

University of Tübingen

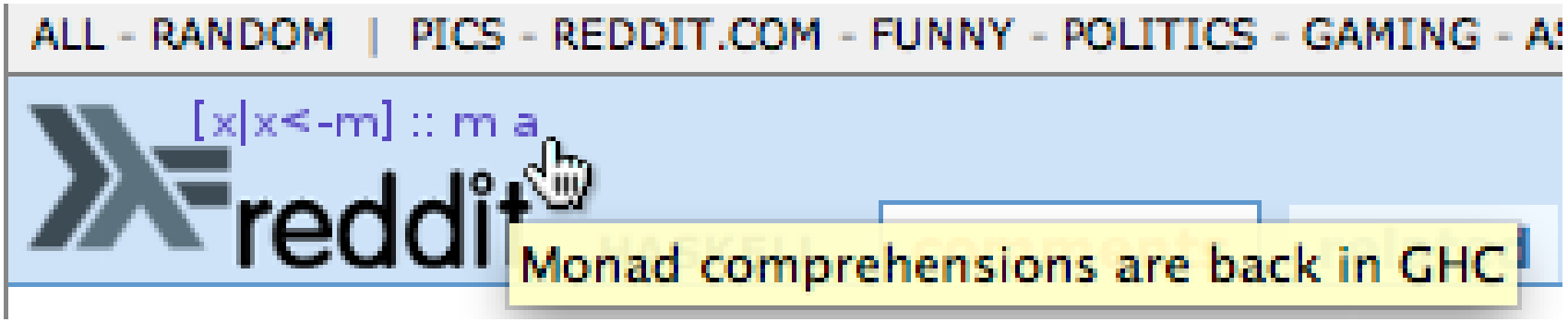
(joint work at various times with Torsten Grust, Andres Löh, Tom Schreiber, Nils Schweinsberg, Alexander Ulrich, and Jeroen Weijers.)

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



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

Inception - Comprehending Monads (Wadler 1992)

$$[x \mid x \leftarrow ys, x < y]$$

$$\begin{aligned}
&\mathit{concat}\ (\mathit{map}\ (\lambda x \rightarrow \mathit{map}\ (\lambda() \rightarrow x) \\
&\qquad\qquad\qquad\qquad (\mathbf{if}\ (x < y)\ \mathbf{then}\ [()]\ \mathbf{else}\ [])) \\
&\qquad\qquad ys)
\end{aligned}$$

$$\begin{aligned}
&\mathit{join}\ (\mathit{liftM}\ (\lambda x \rightarrow \mathit{liftM}\ (\lambda() \rightarrow x) \\
&\qquad\qquad\qquad\qquad (\mathit{guard}\ (x < y))) \\
&\qquad\qquad ys)
\end{aligned}$$

Rise - Haskell 1.4 Report (1997)

HASKELL 1.4

"generators of the form $p \leftarrow e$, where p is a pattern of type t and e is an expression of type $Monad\ m \Rightarrow m\ t$"

Fall - John Hughes' summary (1997 Aug 27)

"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:

$$
\begin{array}{rll}
\mathit{map} & :: (QA\ a, QA\ b) & \Rightarrow (Q\ a \to Q\ b) \to Q\ [a] \to Q\ [b] \\
\mathit{filter} & :: (QA\ a) & \Rightarrow (Q\ a \to Q\ \mathit{Bool}) \to Q\ [a] \to Q\ [a] \\
\mathit{concat} & :: (QA\ a) & \Rightarrow Q\ [[a]] \to Q\ [a] \\
\mathit{zip} & :: (QA\ a, QA\ b) & \Rightarrow Q\ [a] \to Q\ [b] \to Q\ [(a, b)] \\
\ldots & &
\end{array}
$$

- 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:

$$
\begin{aligned}
\mathit{map} &:: (Q\ a \to Q\ b) \to Q\ [a] \to Q\ [b] \\
\mathit{filter} &:: (Q\ a \to Q\ \mathit{Bool}) \to Q\ [a] \to Q\ [a] \\
\mathit{concat} &:: Q\ [[a]] \to Q\ [a] \\
\mathit{zip} &:: Q\ [a] \to Q\ [b] \to Q\ [(a, b)] \\
&\ldots
\end{aligned}
$$

- Reinventing the wheel as a quasiquoter:

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

Accident

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

Accident

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

Examples

$$
\begin{array}{lll}
quickSort & :: & Ord\ a \Rightarrow [\,a\,] \rightarrow [\,a\,] \\
quickSort\ ys & & \\
\quad \mid null\ ys & = & mzero \\
\quad \mid otherwise & = & quickSort\ [\,x \mid x \leftarrow ys, x < y\,]\ `mplus` \\
 & & \qquad\qquad\qquad [\,x \mid x \leftarrow ys, x \equiv y\,]\ `mplus` \\
 & & \qquad\quad quickSort\ [\,x \mid x \leftarrow ys, x > y\,] \\
\quad \textbf{where} & & \\
\qquad y = head\ ys & &
\end{array}
$$

Examples

$$
\begin{aligned}
&\mathit{quickSort} \quad :: \mathit{Ord}\ a \Rightarrow [\,a\,] \to [\,a\,] \\
&\mathit{quickSort}\ \mathit{ys} \\
&\quad \mid \mathit{null}\ \mathit{ys} \quad = \mathit{mzero} \\
&\quad \mid \mathit{otherwise} = \mathit{quickSort}\ [\,x \mid x \leftarrow \mathit{ys}, x < y\,]\ `\mathit{mplus}` \\
&\qquad\qquad\qquad\qquad\qquad [\,x \mid x \leftarrow \mathit{ys}, x \equiv y\,]\ `\mathit{mplus}` \\
&\qquad\qquad\qquad\quad \mathit{quickSort}\ [\,x \mid x \leftarrow \mathit{ys}, x > y\,] \\
&\qquad \textbf{where} \\
&\qquad\quad y = \mathit{head}\ \mathit{ys}
\end{aligned}
$$

```
{-# LANGUAGE MonadComprehensions #-}
```

$$
\begin{aligned}
&\mathit{quickSort} :: \mathit{Ord}\ a \Rightarrow \mathit{Seq}\ a \quad\ \to \mathit{Seq}\ a \\
&\mathit{quickSort} :: \mathit{Ord}\ a \Rightarrow \mathit{DList}\ a \to \mathit{DList}\ a \\
&\mathit{quickSort} :: \mathit{Ord}\ a \Rightarrow \mathit{AList}\ a \to \mathit{AList}\ a
\end{aligned}
$$

$$
\mathit{quickSort} :: (\mathit{Ord}\ a, \mathit{ListLike}\ m\ a, \mathit{MonadPlus}\ m) \Rightarrow m\ a \to m\ a
$$

Data Parallel Haskell (DPH)

- Parallel Array Comprehensions:

$$
\begin{aligned}
&\mathit{sparseMul} && :: [:(\mathit{Int}, \mathit{Float}):] \rightarrow [:\mathit{Float}:] \rightarrow \mathit{Float} \\
&\mathit{sparseMul}\ \mathit{sv}\ v && = \mathit{sumP}\ [:f * (v\ !:\ i) \mid (i, f) \leftarrow \mathit{sv}:]
\end{aligned}
$$

- Monad Comprehensions:

$$
\begin{aligned}
&\mathit{sparseMul} && :: [:(\mathit{Int}, \mathit{Float}):] \rightarrow [:\mathit{Float}:] \rightarrow \mathit{Float} \\
&\mathit{sparseMul}\ \mathit{sv}\ v && = \mathit{sumP}\ [f * (v\ !:\ i) \mid (i, f) \leftarrow \mathit{sv}]
\end{aligned}
$$

Parallel List Comprehensions

$$
\begin{aligned}
&\mathit{denseMult} && :: [\mathit{Float}] \to [\mathit{Float}] \to [\mathit{Float}] \\
&\mathit{denseMult}\ \mathit{xs}\ \mathit{ys} && = \mathit{sum}\ [x * y \mid x \leftarrow \mathit{xs} \mid y \leftarrow \mathit{ys}]
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{denseMult} && :: [\mathit{Float}] \to [\mathit{Float}] \to [\mathit{Float}] \\
&\mathit{denseMult}\ \mathit{xs}\ \mathit{ys} && = \mathit{sum}\ [x * y \mid (x, y) \leftarrow \mathit{zip}\ \mathit{xs}\ \mathit{ys}]
\end{aligned}
$$

MonadZip

$$
\begin{array}{l}
\textbf{class}\ \mathit{Monad}\ m \Rightarrow \mathit{MonadZip}\ m\ \textbf{where} \\
\quad \mathit{mzip} \qquad\quad :: m\ a \to m\ b \to m\ (a, b) \\
\quad \mathit{mzipWith} :: (a \to b \to c) \to m\ a \to m\ b \to m\ c \\
\quad \mathit{munzip} \qquad :: m\ (a, b) \to (m\ a, m\ b)
\end{array}
$$

$$
\begin{array}{l}
\textbf{instance}\ \mathit{MonadZip}\ [\,]\ \textbf{where} \\
\quad \mathit{mzip} = \mathit{zip} \\
\quad \mathit{munzip} = \mathit{unzip}
\end{array}
$$

$$
\begin{array}{l}
\textbf{instance}\ \mathit{MonadZip}\ [::]\ \textbf{where} \\
\quad \mathit{mzip} = \mathit{zipP} \\
\quad \mathit{munzip} = \mathit{unzipP}
\end{array}
$$

Parallel Monad Comprehensions

```
{-# LANGUAGE  MonadComprehensions, ParallelListComp #-}
```

$$
\begin{aligned}
&\mathit{denseMultP} && :: [:\mathit{Float}:] \to [:\mathit{Float}:] \to [:\mathit{Float}:] \\
&\mathit{denseMultP}\ \mathit{xs}\ \mathit{ys} && = \mathit{sumP}\ [x * y \mid x \leftarrow \mathit{xs} \mid y \leftarrow \mathit{ys}]
\end{aligned}
$$

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

MonadZip

$$
\begin{aligned}
&\textbf{class}\ \mathit{Monad}\ m \Rightarrow \mathit{MonadZip}\ m\ \textbf{where}\\
&\quad \mathit{mzip} :: m\ a \to m\ b \to m\ (a, b)\\
&\quad \mathit{mzip} = \mathit{mzipWith}\ (,)\\
&\quad \mathit{mzipWith} :: (a \to b \to c) \to m\ a \to m\ b \to m\ c\\
&\quad \mathit{mzipWith}\ f\ \mathit{ma}\ \mathit{mb} = \mathit{liftM}\ (\mathit{uncurry}\ f)\ (\mathit{mzip}\ \mathit{ma}\ \mathit{mb})\\
&\quad \mathit{munzip} :: m\ (a, b) \to (m\ a, m\ b)\\
&\quad \mathit{munzip}\ \mathit{mab} = (\mathit{liftM}\ \mathit{fst}\ \mathit{mab}, \mathit{liftM}\ \mathit{snd}\ \mathit{mab})
\end{aligned}
$$

MonadZip Laws

- Naturality

$$\begin{aligned} &\mathit{liftM}\ (f \ast\!\!\ast\!\!\ast\ g)\ (\mathit{mzip}\ \mathit{ma}\ \mathit{mb}) \\ &\qquad \equiv \mathit{mzip}\ (\mathit{liftM}\ f\ \mathit{ma})\ (\mathit{liftM}\ g\ \mathit{mb}) \end{aligned}$$

- Associativity

$$\begin{aligned} &\mathit{liftM}\ (\lambda(a,(b,c)) \to ((a,b),c))\ (\mathit{mzip}\ \mathit{ma}\ (\mathit{mzip}\ \mathit{mb}\ \mathit{mc})) \\ &\qquad \equiv \mathit{mzip}\ (\mathit{mzip}\ \mathit{ma}\ \mathit{mb})\ \mathit{mc} \end{aligned}$$

- Information Preservation

$$\begin{aligned} &\mathit{liftM}\ (\mathit{const}\ ())\ \mathit{ma} = \mathit{liftM}\ (\mathit{const}\ ())\ \mathit{mb} \\ &\qquad \Rightarrow \mathit{munzip}\ (\mathit{mzip}\ \mathit{ma}\ \mathit{mb}) \equiv (\mathit{ma}, \mathit{mb}) \end{aligned}$$

SQL-like List Comprehensions

```
{-# LANGUAGE TransformListComp #-}
```

$$
\begin{aligned}
&\mathit{employees} :: [(\mathit{String}, \mathit{String}, \mathit{Integer})] \\
&\mathit{employees} = [(\texttt{"Simon"}, \texttt{"MS"}, \quad 80), (\texttt{"Erik"}, \quad \texttt{"MS"}, 90) \\
&\qquad\qquad\quad , (\texttt{"Phil"}, \ \texttt{"Ed"}, \quad 40), (\texttt{"Gordon"}, \texttt{"Ed"}, 45) \\
&\qquad\qquad\quad , (\texttt{"Paul"}, \ \texttt{"Yale"}, 60) \\
&\qquad\qquad\quad]
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{query} :: [(\mathit{String}, \mathit{Integer})] \\
&\mathit{query} = [\ (\mathit{the}\ \mathit{dept}, \mathit{sum}\ \mathit{salary}) \\
&\qquad\qquad \mid (\mathit{name}, \mathit{dept}, \mathit{salary}) \leftarrow \mathit{employees} \\
&\qquad\qquad , \ \textbf{then group} \quad \textbf{by}\ \mathit{dept} \\
&\qquad\qquad , \ \textbf{then}\ \mathit{sortWith}\ \textbf{by}\ (\mathit{sum}\ \mathit{salary}) \\
&\qquad\qquad]
\end{aligned}
$$

$$[(\texttt{"Yale"}, 60), (\texttt{"Ed"}, 85), (\texttt{"MS"}, 170)]$$

$$
\begin{array}{l}
query :: [(String, Integer)] \\
query = [\ (the\ dept, sum\ salary) \\
\qquad\qquad \mid (name, dept, salary) \leftarrow employees \\
\qquad\qquad ,\ \textbf{then group}\quad \textbf{by}\ dept \\
\qquad\qquad ,\ \textbf{then}\ sortWith\ \textbf{by}\ (sum\ salary) \\
\qquad\qquad]
\end{array}
$$

$$
\begin{array}{l}
map \\
\quad (\lambda(\_, dept, salary) \rightarrow (the\ dept, sum\ salary)) \\
\quad sortWith \\
\qquad (\lambda(\_, \_, salary) \rightarrow sum\ salary) \\
\qquad (map\ (\lambda l \rightarrow (map\ (\lambda(name, \_, \_)\ \rightarrow name)\ \ l \\
\qquad\qquad\qquad\qquad , map\ (\lambda(\_, dept, \_)\quad \rightarrow dept)\quad l \\
\qquad\qquad\qquad\qquad , map\ (\lambda(\_, \_, salary) \rightarrow salary)\ l)) \\
\qquad\qquad (groupWith\ (\lambda(\_, dept, \_) \rightarrow dept)\ employees))
\end{array}
$$

SQL-like Monad Comprehensions

- Lists

$$[a] \to [a]$$
$$[a] \to [[a]]$$

- Monads

$$Monad\ m \Rightarrow m\ a \to m\ a$$
$$Monad\ m \Rightarrow m\ a \to m\ (m\ a)$$

Tree Scans

Conal Elliott

Inspirations & experiments, mainly about denotative/functional programming in Haskell

Search...

HOME | ABOUT

« Deriving list scans

Composable parallel scanning »

Deriving parallel tree scans

1st March 2011, 12:41 pm

The post *Deriving list 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 "∅" and "(⊕)".
- 2011-03-23: corrected "linear-time" to "linear-work" in two places.

Trees

Our trees will be non-empty and binary:

```
data T a = Leaf a | Branch (T a) (T a)
```

RECENT POSTS

- A third view on trees
- Parallel tree scanning by composition
- Composable parallel scanning
- Deriving parallel tree scans

Would **do** do?

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

$$\begin{array}{l} \textbf{do}\ x \leftarrow xs \\ \quad \mathit{guard}\ (x > 0) \\ \quad \mathit{return}\ x \end{array}$$

Would **do** do?

$$
\begin{aligned}
&\mathit{query} :: [(\mathit{String}, \mathit{Integer})] \\
&\mathit{query} = [\ \ (\mathit{the}\ \mathit{dept}, \mathit{sum}\ \mathit{salary}) \\
&\qquad\qquad \mid (\mathit{name}, \mathit{dept}, \mathit{salary}) \leftarrow \mathit{employees} \\
&\qquad\qquad ,\ \mathbf{then\ group}\quad \mathbf{by}\ \mathit{dept} \\
&\qquad\qquad ,\ \mathbf{then}\ \mathit{sortWith}\ \mathbf{by}\ (\mathit{sum}\ \mathit{salary}) \\
&\qquad\qquad]
\end{aligned}
$$

$$
\begin{aligned}
&\mathbf{do\ let}\ g = \mathbf{do}\ l \leftarrow \mathit{mgroupWith}\ (\lambda(\_, \mathit{dept}, \_) \rightarrow \mathit{dept}) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{employees} \\
&\qquad\qquad\qquad\qquad\qquad\qquad \mathit{return}\ (\mathit{liftM}\ (\lambda(\mathit{name}, \_, \_) \rightarrow \mathit{name})\ l \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad ,\mathit{liftM}\ (\lambda(\_, \mathit{dept}, \_) \rightarrow \mathit{dept})\ l \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad ,\mathit{liftM}\ (\lambda(\_, \_, \mathit{salary}) \rightarrow \mathit{salary})\ l) \\
&\qquad (\_, \mathit{dept}, \mathit{salary}) \leftarrow \mathit{sortWith}\ (\lambda(\_, \_, s) \rightarrow \mathit{sum}\ s)\ g \\
&\qquad \mathit{return}\ (\mathit{the}\ \mathit{dept}, \mathit{sum}\ \mathit{salary})
\end{aligned}
$$

- Communications of the ACM (June, 1970)

A Relational Model of Data for Large Shared Data Banks

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

$$R*S = \{(a, b, c) : R(a, b) \wedge S(b, c)\}$$

- Comprehensions in programming languages
  - SETL
  - LINQ
  - F#
  - Erlang
  - Python
  - Perl 6
  - Links
  - . . .

Syntax

$p, q ::= w \leftarrow e$		generator
$\mid$	$\textbf{let}\ w = e$	let binding
$\mid$	g	guard
$\mid$	$p\,\textbf{,}\,q$	Cartesian product
$\mid$	$p \mid q$	zipping
$\mid$	$q\,\textbf{,then}\ f$	transformation
$\mid$	$q\,\textbf{,then}\ f\ \textbf{by}\ e$	and projection
$\mid$	$q\,\textbf{,then group by}\ e$	grouping
$\mid$	$q\,\textbf{,then group using}\ f$	user-defined
$\mid$	$q\,\textbf{,then group by}\ e\ \textbf{using}\ f$	grouping

Typing Rules

List comprehensions $\boxed{P, \Gamma \vdash e : \tau}$

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

Variables $\boxed{\vdash w \Rightarrow \Delta}$

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

Basic list comprehension body $\boxed{P, \Gamma \vdash e \Rightarrow \Delta}$

$$\frac{\Gamma \vdash e : \texttt{Bool}}{\{\mathit{MonadPlus}\ m\}, \Gamma \vdash e \Rightarrow (m, \emptyset)}\ [\mathit{Guard}] \qquad \frac{}{\emptyset, \Gamma \vdash () \Rightarrow (m, \emptyset)}\ [\mathit{Unit}] \qquad \frac{\Gamma \vdash e : m\ \tau \quad \vdash w : \tau \Rightarrow \Delta}{\emptyset, \Gamma \vdash w \leftarrow e \Rightarrow (m, \Delta)}\ [\mathit{Gen}]$$

$$\frac{\Gamma \vdash e : \tau \quad \vdash x : \tau \Rightarrow \Delta}{\emptyset, \Gamma \vdash \mathbf{let}\ x = e \Rightarrow (m, \Delta)}\ [\mathit{Let}] \qquad \frac{P, \Gamma \vdash p \Rightarrow (m, \Delta) \quad P', \Gamma \cup \Delta \vdash q \Rightarrow (m, \Delta')}{P \cup P', \Gamma \vdash p, q \Rightarrow (m, \Delta \cup \Delta')}\ [\mathit{Comma}]$$

Parallel list comprehension body $\boxed{P, \Gamma \vdash e \Rightarrow \Delta}$

$$\frac{P, \Gamma \vdash p \Rightarrow (m, \Delta) \quad P', \Gamma \cup \Delta \vdash q\ (m, \Delta')}{\{\mathit{MonadZip}\ m\} \cup P \cup P', \Gamma \vdash p \mid q \Rightarrow (m, \Delta \cup \Delta')}\ [\mathit{Bar}]$$

Comprehensive list comprehension body $\boxed{P, \Gamma \vdash e \Rightarrow \Delta}$

$$\frac{P, \Gamma \vdash q \Rightarrow (m, \Delta) \quad \Gamma \vdash f : \forall \alpha.\, m\ \alpha \to m\ \alpha}{P, \Gamma \vdash q, \mathbf{then}\ f \Rightarrow (m, \Delta)}\ [\mathit{then}] \qquad \frac{\begin{array}{c} P, \Gamma \vdash q \Rightarrow (m, \Delta) \quad \Gamma \cup \Delta \vdash e : \tau \\ \Gamma \vdash f : \forall \alpha.\, (\alpha \to \tau) \to m\ \alpha \to m\ \alpha \end{array}}{P, \Gamma \vdash q, \mathbf{then}\ f\ \mathbf{by}\ e \Rightarrow (m, \Delta)}\ [\mathit{thenBy}]$$

$$\frac{P, \Gamma \vdash q \Rightarrow (m, \Delta) \quad \Gamma \cup \Delta \vdash e : \tau}{P \cup \{\mathit{MonadGroup}\ m\}, \Gamma \vdash q, \mathbf{then\ group\ by}\ e \Rightarrow m\ \Delta}\ [\mathit{groupBy}]$$

$$\frac{P, \Gamma \vdash q \Rightarrow (m, \Delta) \quad \Gamma \vdash f : \forall \alpha.\, m\ \alpha \to m\ (m\ \alpha)}{P, \Gamma \vdash q, \mathbf{then\ group\ using}\ f \Rightarrow m\ \Delta}\ [\mathit{groupUsing}] \qquad \frac{\begin{array}{c} P, \Gamma \vdash q \Rightarrow (m, \Delta) \quad \Gamma \cup \Delta \vdash e : \tau \\ \Gamma \vdash f : \forall \alpha.\, (\alpha \to \tau) \to m\ \alpha \to m\ (m\ \alpha) \end{array}}{P, \Gamma \vdash q, \mathbf{then\ group\ by}\ e\ \mathbf{using}\ f \Rightarrow m\ \Delta}\ [\mathit{groupByUsing}]$$

Desugaring

$$
\begin{aligned}
[\,e \mid q\,] &= \mathit{liftM}\ (\lambda q_v \to e)[\![q]\!] \\
\\
[\![w \leftarrow e]\!] &= e \\
[\![\mathbf{let}\ w = d]\!] &= \mathit{return}\ d \\
[\![g]\!] &= \mathit{guard}\ g \\
[\![p, q]\!] &= \mathit{join}\ (\mathit{liftM} \\
&\qquad (\lambda p_v \to \mathit{liftM} \\
&\qquad\quad (\lambda q_v \to (p_v, q_v))[\![q]\!]) \\
&\qquad [\![p]\!]) \\
[\![p \mid q]\!] &= \mathit{mzip}[\![p]\!][\![q]\!] \\
[\![q, \mathbf{then}\ f]\!] &= f[\![q]\!] \\
[\![q, \mathbf{then}\ f\ \mathbf{by}\ e]\!] &= f\ (\lambda q_v \to e)\ [\![q]\!] \\
[\![q, \mathbf{then\ group\ by}\ e]\!] &= \mathit{liftM}\ \mathit{unzip}_{q_v} \\
&\qquad (\mathit{mgroupWith} \\
&\qquad\quad (\lambda q_v \to e)[\![q]\!]) \\
[\![q, \mathbf{then\ group\ by}\ e\ \mathbf{using}\ f]\!] &= \mathit{liftM}\ \mathit{unzip}_{q_v} \\
&\qquad (f\ (\lambda q_v \to e)[\![q]\!]) \\
[\![q, \mathbf{then\ group\ using}\ f]\!] &= (\mathit{liftM}\ \mathit{unzip}_{q_v}\ (f[\![q]\!])
\end{aligned}
$$

$$
\begin{aligned}
\mathit{unzip}_{()} &= \mathit{id} \\
\mathit{unzip}_{x} &= \mathit{id} \\
\mathit{unzip}_{(w1,w2)} &= \lambda e \to (\mathit{unzip}_{w_1}\ (\mathit{liftM}\ (\lambda(x,y) \to x)\ e) \\
&\qquad\quad , \mathit{unzip}_{w_2}\ (\mathit{liftM}\ (\lambda(x,y) \to y)\ e))
\end{aligned}
$$

Error Messages

$$[(x, y) \mid x \leftarrow [1], y \leftarrow \mathit{Just}\ 5]$$

```
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 <- Just 5
 In the expression:
   [(x, y) | x <- [1], y <- Just 5]
```

List notation overloading and defaulting

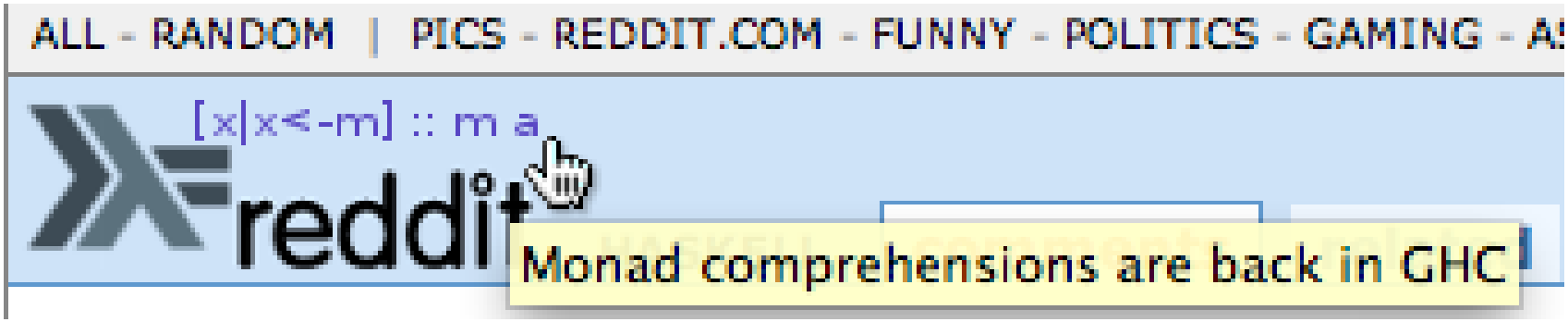
- Concrete proposal on list notation overloading for collections.
- Not so concrete proposal on extending Haskell's existing defaulting mechanism (e.g., for backwards compatibility and for resolving type ambiguities).

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.



Available in GHC-7.2

Database Supported Haskell (DSH)

- Database-executable combinators:

$$
\begin{array}{rll}
\mathit{map} & :: (QA\ a, QA\ b) & \Rightarrow (Q\ a \to Q\ b) \to Q\ [a] \to Q\ [b] \\
\mathit{filter} & :: (QA\ a) & \Rightarrow (Q\ a \to Q\ \mathit{Bool}) \to Q\ [a] \to Q\ [a] \\
\mathit{concat} & :: (QA\ a) & \Rightarrow Q\ [[a]] \to Q\ [a] \\
\mathit{zip} & :: (QA\ a, QA\ b) & \Rightarrow Q\ [a] \to Q\ [b] \to Q\ [(a, b)] \\
\ldots & &
\end{array}
$$

- 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:

$$
\begin{aligned}
t ::= \; & () \\
\mid \; & \mathit{Bool} \\
\mid \; & \mathit{Char} \\
\mid \; & \mathit{Integer} \\
\mid \; & \mathit{Double} \\
\mid \; & \mathit{Text} \\
\mid \; & (t, ..., t) \\
\mid \; & [t]
\end{aligned}
$$

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