Bringing Back Monad Comprehensions

George Giorgidze Torsten Grust Nils Schweinsberg Jeroen Weijers

Wilhelm-Schickard-Institut für Informatik
Eberhard Karls Universität Tübingen
{george.giorgidze,torsten.grust,jeroen.weijers}@uni-tuebingen.de
nils.schweinsberg@student.uni-tuebingen.de

Abstract

This paper is about a Glasgow Haskell Compiler (GHC) extension that generalises Haskell's list comprehension notation to monads. The monad comprehension notation implemented by the extension supports generator and filter clauses, as was the case in the Haskell 1.4 standard. In addition, the extension generalises the recently proposed parallel and SQL-like list comprehension notations to monads. The aforementioned generalisations are formally defined in this paper. The extension will be available in GHC 7.2.

This paper gives several instructive examples that we hope will facilitate wide adoption of the extension by the Haskell community. We also argue why the **do** notation is not always a good fit for monadic libraries and embedded domain-specific languages, especially for those that are based on collection monads. Should the question of how to integrate the extension into the Haskell standard arise, the paper proposes a solution to the problem that led to the removal of the monad comprehension notation from the language standard.

Categories and Subject Descriptors D.3.3 [*Language Constructs and Features*]: Data types and structures; H.2.3 [*Languages*]: Query languages

General Terms Languages, Design, Theory

Keywords Haskell, monad, comprehension

1. Introduction

List comprehensions provide for a concise and expressive notation for writing list-processing code. Although the notation itself is extremely useful, Haskell's built-in list data structure is not always a good choice. Performance and memory requirements, and the need to use data structures admitting strict and parallel operations are just a few factors that may render Haskell lists unsuitable for many applications. Currently, in Haskell, the comprehension notation only works for lists.

We think that the notation is too good for being confined to just lists. In this paper, we present a language extension that generalises the list comprehension notation to monads. When it comes to the generator and filter clauses of the standard list comprehension notation, the extension is based on the generalisation that was proposed by Wadler [28] and was subsequently integrated in the Haskell 1.4 standard [4]. In addition, we generalise recently proposed extensions to the list comprehension notation to monads: SQL-like list comprehensions and parallel list comprehensions.

SQL-like list comprehensions [25] extend the notation with SQL-inspired clauses, most notably for grouping. Currently, GHC supports SQL-like comprehensions only for lists. Parallel list comprehensions as implemented by GHC and Hugs [2] extend the list comprehension with a clause that supports zipping of lists. Generalisations of the two aforementioned list comprehension notation extensions are formally described in this paper.

The extension has been implemented in the Glasgow Haskell Compiler (GHC) [3] and will be available in GHC 7.2. A GHC patch that implements the monad comprehensions extension was implemented by the third author. The patch was subsequently modified and applied to the GHC source tree by Simon Peyton Jones.

This paper gives several instructive examples that we hope will facilitate wide adoption of the extension by the Haskell community. Some of the examples demonstrate that the **do** notation is not always a good fit for monadic libraries and embedded domain-specific languages (EDSLs), especially for those that are based on collection monads. We also show that monad comprehensions can be used for data-parallel programming as a drop-in replacement for array comprehensions [9]. This obviates the need for the special array comprehension notation currently implemented in GHC.

As we have already mentioned, monad comprehensions, once part of the Haskell standard, were dropped from the language [4, 24]. The reasons included monad-comprehensions–related error messages produced by Haskell implementations. This was considered as a barrier too high for new users of Haskell [15]. Type ambiguity errors were of particular concern. Should the question of how to integrate the extension into the Haskell standard arise, this paper proposes to extend Haskell's defaulting mechanism and use it for disambiguation of comprehension-based code just like defaulting is used for disambiguation of numerical code.

The following list outlines the contributions of this paper.

- We present a language extension that brings back monad comprehensions to GHC. In addition to the well-known generalisation of the standard list comprehension notation with generator and filter clauses, we also generalise the clauses introduced by SQL-like and parallel list comprehensions to monads.
- We formally define typing and desugaring rules implemented by the language extension.
- We provide several instructive examples that make use of monad comprehensions and demonstrate that the **do** notation is not always a good fit for monadic programming. We also show that monad comprehensions subsume array comprehensions.

Haskell'11, September 22, 2011, Tokyo, Japan.
Copyright © 2011 ACM 978-1-4503-0860-1/11/09. . . $10.00

```
quickSort         :: Ord α ⇒ [α] → [α]
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
```

Figure 1. Quicksort for lists.

- We overview the reasons that led to the removal of monad comprehensions from the language standard and propose a possible solution.

The rest of this paper is structured as follows: Section 2 informally introduces monad comprehensions by means of instructive examples; Section 3 formalises the syntax, typing rules, and desugaring translation of monad comprehensions; Section 4 overviews the GHC extension that implements monad comprehensions; Section 5 proposes a solution to the type ambiguity problem that influenced the removal of monad comprehensions from the Haskell language standard and proposes a Haskell extension for list literal overloading; Section 6 reviews related work; and finally, Section 7 concludes the paper.

2. Monad Comprehension Examples

The purpose of this section is twofold; firstly to informally introduce monad comprehensions by means of examples and secondly to show the reader that monad comprehensions can be used as an expressive and concise notation in a range of application domains. The examples provided in this section should provide enough insight in the use of monad comprehensions for the reader to make use of monad comprehensions in her own code. For further details, the reader can consult a formal description of the monad comprehensions syntax, typing rules and desugaring translation given in Section 3.

2.1 Quicksort

Let us consider how monad comprehensions can be used to generalise existing code that is based on list comprehensions to work with other monadic structures. We use the well-known Haskell implementation of the Quicksort algorithm given in Figure 1 to demonstrate several possible generalisations.

The *quickSort* definition makes use of the overloaded functions *mzero* and *mplus* for empty list construction and list concatenation, respectively. These overloaded functions come from the *MonadPlus* class, which is defined in the *Control.Monad* module of the standard Haskell library. The *MonadPlus* class is a subclass of *Monad* for those monads that are monoids as well. The comprehension expressions, and the *null* and *head* functions are specific to lists.

Consider the sub-expression $[x \mid x \leftarrow ys, x < y]$ from the Quicksort example. This sub-expression is, in the case of normal list comprehensions, desugared into the following code:

```
concat (map (λx → map (λ() → x)
                          (if (x < y) then [()] else []))
          ys)
```

Note the use of the *concat* and *map* list processing combinators.

When the monad comprehension extension is turned on, GHC considers the same comprehension expression as a monad comprehension and desugars it to the following code:

```
join (liftM (λx → liftM (λ() → x)
                          (guard (x < y)))
          ys)
```

Note the use of the *join*, *liftM* and *guard* monadic combinators. The monad comprehensions extension can be turned on by placing the following pragma on top of a Haskell module:

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

By overloading the comprehension notation and hiding the Haskell Prelude definitions of the *null* and *head* functions, the *quickSort* definition can be given any of the following type signatures:

```
quickSort :: Ord α ⇒ Seq α     → Seq α
quickSort :: Ord α ⇒ DList α → DList α
quickSort :: Ord α ⇒ AList α → AList α
```

- The first type signature allows the *quickSort* definition to be used for the strict sequence type defined in the *containers* package. This version benefits from $O(log\, n)$ append compared to the $O(n)$ append used in the list version.
- The second type signature allows the *quickSort* definition to be used for the difference list type defined in the *difference-list* package. This version benefits from $O(1)$ append.
- The third type signature allows the *quickSort* definition to be used for the catenation list type defined in the *monad-par* package. This version benefits from $O(1)$ append and parallel filters.

The package *ListLike* provides a type class that overloads regular list functions (including *null* and *head*). Using this package it is possible to give the *quickSort* definition the following type signature:

```
quickSort :: (Ord α, ListLike m α, MonadPlus m) ⇒
                 m α → m α
```

With this type signature the definition works for any list-like structure that is also a monad and a monoid. This includes the four data structures described earlier in this section. Note that when performing the generalisation described in this section, we do *not* have to change the definition of *quickSort*, we only have to change its type signature.

2.2 Data-Parallel Arrays

GHC supports *array comprehensions* for processing strict, data-parallel arrays [9]. For example, the following function, that multiplies a sparse vector with a dense vector, makes use of the array comprehension notation:

```
sparseMul         :: [: (Int, Float) :] → [: Float :] → Float
sparseMul sv v = sumP [: f ∗ (v !: i) | (i, f) ← sv :]
```

GHC desugars array comprehensions into array processing combinators. Because a monad instance for data-parallel arrays is provided by the data-parallel programming library that is shipped with GHC, monad comprehensions can be used as a drop-in replacement for array comprehensions. For example, the sparse vector multiplication can now be defined as follows (note that the array comprehension notation has been replaced with monad comprehension notation):

```
sparseMul         :: [: (Int, Float) :] → [: Float :] → Float
sparseMul sv v = sumP [f ∗ (v !: i) | (i, f) ← sv]
```

Because monad comprehensions subsume array comprehensions, it would be possible to drop support for array comprehensions in favour of the more general comprehension construct. This will simplify the maintenance of GHC's front-end. Currently, GHC provides comprehension notation for lists, as specified in the

Haskell language standard; for data-parallel arrays as implemented in the data-parallel Haskell extension; and for monads as implemented in the present monad comprehensions extension.

2.3 Zip Comprehensions

GHC and Hugs support the *ParallelListComp* extension that allows for drawing elements from lists in parallel. To our knowledge, parallel list comprehensions were first introduced in a nested data-parallel language called NESL [6]. The *ParallelListComp* extension featured in Haskell was ported from Cryptol [19], a purely functional DSL for writing cryptographic applications. We refer to the notation supported by this extension as *zip comprehensions*, because it is syntactic sugar for the *zip* combinator from the standard Haskell Prelude. For example, consider the following function that multiplies two dense vectors of floating point numbers:

```
denseMult       :: [Float] → [Float] → [Float]
denseMult xs ys = sum [x ∗ y | x ← xs | y ← ys]
```

Here, vectors are represented as lists. This definition is desugared into the following code:

```
denseMult       :: [Float] → [Float] → [Float]
denseMult xs ys = sum [x ∗ y | (x, y) ← zip xs ys]
```

If the *ParallelListComp* extension is used in conjunction with the *MonadComprehensions* extension, the aforementioned zip comprehension is desugared to the following code:

```
denseMult       :: [Float] → [Float] → [Float]
denseMult xs ys = sum [x ∗ y | (x, y) ← mzip xs ys]
```

The *mzip* function is a member of the *MonadZip* type class that we have introduced to support generalisation of zip comprehensions as a subclass of monads that admit zipping. The class *MonadZip* is defined as follows:

```
class Monad m ⇒ MonadZip m where
  mzip :: m α → m β → m (α, β)
  mzip = mzipWith (,)

  mzipWith :: (α → β → c) → m α → m β → m c
  mzipWith f ma mb = liftM (uncurry f) (mzip ma mb)

  munzip :: m (α, β) → (m α, m β)
  munzip mab = (liftM fst mab, liftM snd mab)
```

The laws that the class methods are expected to satisfy are given in Section 3. As a minimal definition of a *MonadZip* instance one needs to provide either an implementation of *mzip* or an implementation of *mzipWith*. The default implementation of the *munzip* method can be overridden by a more efficient, instance-specific version. The instance for lists, for example, is defined as follows:

```
instance MonadZip [] where
  mzip = zip
  munzip = unzip
```

Let us demonstrate how to use monad comprehensions to implement a data-parallel version of the *denseMult* function. We start by defining a *MonadZip* instance for data-parallel arrays as follows:

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

The *zipP* and *unzipP* functions are defined in the data-parallel programming library that ships with GHC. We can now implement data-parallel, dense-vector multiplication using the monad comprehension notation, instead of array comprehension notation, as follows:

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

In a recently published article, Patricek demonstrates several examples that make use of zip comprehensions and are complementary to the examples given in this paper [23]. Specifically, the article demonstrates the usefulness of zip comprehensions for several monads that are not collections (e.g., monads for parallel parsing and parallel evaluation).

2.4 SQL-like Comprehensions

GHC supports the *TransformListComp* language extension[1] that allows SQL-like constructs to be used for transforming and grouping of results of list comprehension expressions. Consider the following example (adapted from [25]):

```
employees :: [(String, String, Integer)]
employees = [("Dilbert", "Eng", 80)
            ,("Alice",   "Eng", 100)
            ,("Wally",   "Eng", 40)
            ,("Catbert", "HR",  150)
            ,("Dogbert", "Con", 500)
            ,("Ratbert", "HR",  90)
            ]

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

The SQL-like list comprehension expression groups the employees by department, sorts the departments by cumulative salaries of its employees, and returns the sorted list of departments and corresponding cumulative salaries. The SQL-like list comprehension expression evaluates to the following list of tuples:

```
[("Eng", 220), ("HR", 240), ("Con", 500)]
```

The SQL-like comprehension desugars into the following code:

```
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))
```

The functions *the*, *sortWith* and *groupWith* are exported from the *GHC.Exts* module and have the following type signatures:

```
the       :: Eq α  ⇒ [α] → α
sortWith  :: Ord β ⇒ (α → β) → [α] → [α]
groupWith :: Ord β ⇒ (α → β) → [α] → [[α]]
```

The *the* function returns an element of a given list if all elements of the list are equal, otherwise it returns bottom. The *sortWith* function sorts a list by using the provided function to calculate the sorting criteria for each element of the list. The *groupWith* function groups a given list elements by using the provided function to calculate the grouping criteria for each element of the list.

[1] `{-# LANGUAGE MonadComprehensions, TransformListComp #-}` enables SQL-like monad comprehensions.

```
data Tree α = Leaf α | Branch (Tree α) (Tree α)

fmapT :: (α → β) → Tree α → Tree β
fmapT f (Leaf x)       = Leaf (f x)
fmapT f (Branch l r) = Branch (fmapT f l)
                                       (fmapT f r)

instance Functor Tree where
  fmap = fmapT

joinT :: Tree (Tree α) → Tree α
joinT (Leaf x)       = x
joinT (Branch l r) = Branch (joinT l) (joinT r)

instance Monad Tree where
  return   = Leaf
  xs ≫= f = joinT (fmap f xs)
```

Figure 2. The monad of binary trees with labelled leaves.

Intuitively, the **then** clause provides two types of transformations of list comprehension expressions; namely, transformations of type $[\alpha] \rightarrow [\alpha]$ and transformations of type $[\alpha] \rightarrow [[\alpha]]$. Note that, just like its SQL counterpart, the **then group** clause changes types of already bound variables: in our example, the *salary* variable is of type *Integer* at the binding site, but its type changes to [*Integer*] after the **then group** clause.

The SQL-like comprehension example considered in this section makes use of only two syntactic forms of the **then** clause. The notation also supports three other forms of the **then** clause. Having said that, all forms fall into the two types of transformations discussed earlier. A detailed description of all five forms of the **then** clause is given in Section 3.

The monad comprehensions extension generalises SQL-like list comprehensions to monads. Specifically, it generalises the aforementioned types of transformations to $Monad\ m \Rightarrow m\ \alpha \rightarrow m\ \alpha$ and $Monad\ m \Rightarrow m\ \alpha \rightarrow m\ (m\ \alpha)$, respectively. Section 2.5 presents an instructive example that makes use of this generalisation.

2.5 Tree Comprehensions and Scans

The variety of monad instances that have been—and undoubtedly will be—developed and deployed in Haskell code is enormous. Just as diverse are the potential uses of monad comprehension syntax. Consider the monad *Tree* α of full binary trees whose leaves carry labels of type α (Figure 2). Associated with this monad instance are *tree comprehensions* which provide a natural way to express tree traversals.

Here, let us consider *scans* over trees, essential building blocks in the construction of parallel algorithms over lists and tree-shaped structures [5]. On his blog[2], Conal Elliott recently discussed the derivation of such left and right tree scans. His specification starts out with functions *initTs* and *tailTs*, variants of the similarly named list combinators (Figure 3). These functions immediately lead to succinct formulations of scans in terms of SQL-like tree comprehensions:

```
instance Foldable Tree where
  fold (Leaf x)       = x
  fold (Branch l r) = fold l `mappend` fold r

scanlT, scanrT :: (Monoid α) ⇒ Tree α → Tree α
scanlT t = [fold x | x ← t, then group using initTs]
scanrT t = [fold x | x ← t, then group using tailTs]
```

[2] http://conal.net/blog/posts/deriving-parallel-tree-scans/

```
initTs, tailTs :: Tree α → Tree (Tree α)
initTs (Leaf x)       = Leaf (Leaf x)
initTs (Branch l r) = Branch
                             (initTs l)
                             (fmap (l`Branch`) (initTs r))
tailTs (Leaf x)       = Leaf (Leaf x)
tailTs (Branch l r) = Branch
                             (fmap (`Branch`r) (tailTs l))
                             (tailTs r)
```

Figure 3. Conal Elliot's tree variants of *inits* and *tails* (originally defined over lists).

As we have mentioned earlier in this paper, in these comprehensions, variable x is of type α before the **group using** qualifier but of type *Tree* α in the comprehensions' heads. In the definition of *scanlT* t, the clause **group using** *initTs* groups leaf x with those leaves that appear before x in a preorder traversal of t. Note that these groups take the shape of a *Tree* α themselves. The final *fold* flattens out the resulting groups' structure, leaving us with the desired left scan result. The right scan *scanrT* behaves accordingly.

2.6 Rebindable Syntax and Set Comprehensions

The *MonadComprehensions* extension can be used in conjunction with the *RebindableSyntax* extension. This combination of extensions allows the user to rebind the monadic combinators from the Haskell Prelude. This opens further opportunities to make use of monad comprehensions in Haskell libraries and EDSLs.

One such application is to use monad comprehensions as *set comprehensions*. This can be done by hiding the monadic combinators from the Haskell Prelude and importing the *Control.RMonad* module from the *rmonad* package. This module exports the *RMonad* class that allows its instances to introduce constraints on the type contained by the monad. This is how the *rmonad* package defines a monad instance for the *Set* data type from the *containers* package.

Having done the aforementioned preparatory step, monad comprehensions can be used to recreate the original mathematical notation of set comprehensions that inspired list comprehensions in the first place. We leave this as an exercise to the reader.

2.7 Database-Supported Haskell (DSH)

Our initial motivation for bringing back monad comprehensions to GHC, was to improve the Database-Supported Haskell (DSH) library [1, 12]. DSH is a Haskell library for database-supported program execution. Using the library, a relational database management system (RDBMS) can be used as a coprocessor for the Haskell programming language, especially for those program fragments that carry out data-intensive and data-parallel computations. Rather than embedding a relational language into Haskell, DSH turns idiomatic Haskell programs into SQL queries.

In order to use the monad comprehension notation to write database-executable program fragments, we had to re-implement the SQL-like comprehension notation in terms of a quasiquoter. The example from Section 2.4, which makes use of the **group by** clause for lists, can be turned into the database-executable program fragment given in Figure 4.

The quasiquoter *qc* parses the enclosed comprehension and generates the corresponding database-executable combinators at compile time before type checking takes place. These combinators, in the tradition of deep embeddings, construct the query representation as data in order to compile the query into equivalent SQL

```
query :: Q [(String, Integer)]
query = [qc | (the dept, sum salary)
            | (name, dept, salary) ← table "employees"
            , then group by dept using groupWith
            , then sortWith by (sum salary)
          |]
```

Figure 4. Database-executable program fragment.

code. Queries of type $Q\ [(\mathit{String}, \mathit{Integer})]$ evaluate to Haskell values of type $[(\mathit{String}, \mathit{Integer})]$ when executed on a database back-end. The function *table* is used to reference database-resident tables. In this example we assume that the table *employees* exists in the database executing the query.

While workable, the quasiquoting approach has two limitations. Firstly, it requires additional implementation effort from the library developer. Secondly, the error messages that are presented to the library user reference generated code that was not written by the user herself.

By using the monad comprehensions extension that is implemented in GHC and is described in this paper, we managed to solve the aforementioned two problems for DSH. Specifically, we implemented *queryable lists* as restricted monads. We restricted the types contained by the monad to basic types supported by the underlying database as well as arbitrarily nested tuples and lists of these basic types.

We have not yet released the new monad-comprehensions–based DSH because final details (e.g., how desugaring of the **group by** clause interacts with the *RebindableSyntax* extension) still need to be worked out[3].

We think that, just like DSH, other Haskell libraries and EDSLs will also benefit from the extension, especially those that are based on collection monads.

2.8 The do Notation

The **do** notation is probably the most popular notation for writing monadic program fragments in Haskell. However, we think that the **do** notation is, for some applications, not always a good fit and the monad comprehension notation provides a useful alternative.

Although resulting in an arguably less elegant formulation, it is manageable to use the **do** notation as a substitute for monad comprehension expressions that only use generators and filters; for example, the comprehension $[x \mid x \leftarrow xs, x > 0]$ can be written as follows:

```
do x ← xs
   guard (x > 0)
   return x
```

Having said that, the **do** notation is a poor fit as a substitute for SQL-like monad comprehensions; for example the *query* expression given in Section 2.4 has the following equivalent formulation using the **do** notation:

```
do let g = do l ← groupWith (λ(_, dept, _) → dept)
                             employees
              return (liftM (λ(name, _, _)  → name)  l
                     , liftM (λ(_, dept, _)  → dept)  l
                     , liftM (λ(_, _, salary) → salary) l)
   (_, dept, salary) ← sortWith (λ(_, _, s) → sum s) g
   return (the dept, sum salary)
```

[3] http://hackage.haskell.org/trac/ghc/ticket/4370

Of course one could extend the **do** notation with SQL-like clauses, but we decided to generalise the SQL-like list comprehensions instead. From the (query) language design point of view, our decision was inspired by the fact that ordering and grouping constructs have already found their place in other successful comprehensions-based languages (see Section 6). From the language implementation point of view, our decision was influenced by the fact that GHC already had syntactic constructs for the **then** and **then group** clauses for list comprehensions. We think that SQL-like monad comprehensions are a better fit for applications considered in this paper; that is, processing of collection monads and declarative querying facilities.

3. Formalisation

In this section we define how monad comprehensions are typed (Section 3.2), desugared (Section 3.3) and how the standard libraries have to be extended to support monad comprehensions (Section 3.1). Monad comprehensions are syntactically identical to list comprehensions. However, in order to make this paper self-contained, and to make the translation rules easier to read we include the full syntax diagram for monad comprehensions in Figure 5.

Variables
x, y, z

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

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

Qualifiers

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

Figure 5. Monad comprehension syntax.

3.1 Proposed Library Additions

In order to deal with basic comprehension syntax, the type classes provided in the standard libraries suffice. To generalise the syntactic extensions made to list comprehensions with parallel list comprehensions and SQL-like list comprehensions [25], a few new monad-related type classes have to be introduced.

The type class *MonadZip* is introduced to support parallel list comprehensions. Its class definition is given in Figure 6. A minimal complete instance has to provide one of the first two member functions. We also introduce three laws with the type class (Figure 7), that instances should conform with. The first law, the *naturality law*, states that the *mzip* function is a natural transformation (i.e., it is structure preserving). The second law, the *associativity law*, states that the *mzip* function, which is a binary function, is associative. The third law, the *information preservation law*, states that if the *mzip* function is applied to two monadic values with same effect the monadic values can be recovered by the *munzip* function.

We point out that the first two *MonadZip* laws have equivalent *applicative functor* laws [20] and the third *MonadZip* law extends the applicative laws. This suggests that, in principle, the $MonadZip$ class should be a subclass of the $Applicative$ class. Because the $Monad$ class is not declared as a subclass of $Applicative$ and our generalisation is based on monads, we decided to introduce the $MonadZip$ class as a subclass of $Monad$ for those monads that support zipping.

$$
\begin{array}{l}
\mathbf{class}\ \mathit{Monad}\ m \Rightarrow \mathit{MonadZip}\ m\ \mathbf{where} \\
\quad \mathit{mzip} :: m\ \alpha \to m\ \beta \to m\ (\alpha, \beta) \\
\quad \mathit{mzip} = \mathit{mzipWith}\ (,) \\
\quad \mathit{mzipWith} :: (\alpha \to \beta \to c) \to m\ \alpha \to m\ \beta \to m\ c \\
\quad \mathit{mzipWith}\ f\ \mathit{ma}\ \mathit{mb} = \mathit{liftM}\ (\mathit{uncurry}\ f)\ (\mathit{mzip}\ \mathit{ma}\ \mathit{mb}) \\
\quad \mathit{munzip} :: m\ (\alpha, \beta) \to (m\ \alpha, m\ \beta) \\
\quad \mathit{munzip}\ \mathit{mab} = (\mathit{liftM}\ \mathit{fst}\ \mathit{mab}, \mathit{liftM}\ \mathit{snd}\ \mathit{mab})
\end{array}
$$

Figure 6. Definition of class $MonadZip$.

Naturality

$$
\begin{array}{l}
\mathit{liftM}\ (f *** g)\ (\mathit{mzip}\ \mathit{ma}\ \mathit{mb}) \\
\qquad \equiv \mathit{mzip}\ (\mathit{liftM}\ f\ \mathit{ma})\ (\mathit{liftM}\ g\ \mathit{mb})
\end{array}
$$

Associativity

$$
\begin{array}{l}
\mathit{liftM}\ (\lambda(\alpha, (\beta, c)) \to ((\alpha, \beta), c))\ (\mathit{mzip}\ \mathit{ma}\ (\mathit{mzip}\ \mathit{mb}\ \mathit{mc})) \\
\qquad \equiv \mathit{mzip}\ (\mathit{mzip}\ \mathit{ma}\ \mathit{mb})\ \mathit{mc}
\end{array}
$$

Information Preservation

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

Figure 7. The three $MonadZip$ laws.

To generalise the grouping parts of the SQL-like list comprehensions, we depend on a generalised version of $groupWith$. This function is provided through the type class $MonadGroup$ (see Figure 8). The $groupWith$ function transforms a collection of elements into a collection of collections using a given function of type $\alpha \to \tau$ that extracts a grouping criterion of type τ.

Shall that grouping criterion be special? One sensible sample instance of $MonadGroup$ could be defined for [`Int`], where the extraction function determines whether a list element is even or odd (i.e., τ = `Bool`) and then groups the elements accordingly. This particular instance of $MonadGroup$ suggests the constraint $(Eq\ \tau)$. Just as sensible is the supplied $MonadGroup$ instance for lists which sorts the list elements based on the extract grouping criterion—runs of elements with identical criteria will form a group. This instance rather suggests an $(Ord\ \tau)$ constraint.

As a result, we decided to not pose constraints on the type τ of the grouping criterion: appropriate constraints may very well differ for different monads. Requiring the Ord constraint for an instance for a set monad, for example, does not make sense as a set is not ordered. Also, there is no immediately obvious notion of grouping for certain monads (e.g., for IO and $State$). This should, however, not imply that a programmer cannot come up with a useful instance and be able to use the **then group by** syntax.

$$
\begin{array}{l}
\mathbf{class}\ \mathit{Monad}\ m \Rightarrow \mathit{MonadGroup}\ m\ \tau\ \mathbf{where} \\
\quad \mathit{mgroupWith} :: (\alpha \to \tau) \to m\ \alpha \to m\ (m\ \alpha)
\end{array}
$$

Figure 8. Definition of class $MonadGroup$.

3.2 Typing rules

Figure 9 provides the typing rules for monad comprehensions. At first glance these typing rules look similar to the typing rules for comprehensive list comprehensions [25]. The rules are similar indeed, but feature the appropriate generalisations needed to type monad comprehensions. We will only discuss those rules that are notably different. In these rules τ ranges over types, m and α range over type variables. We define Δ and Γ to range over type environments, and P to range over sets of predicates. Most typing rules propagate a set of predicates P along with a type environment Γ.

The predicate environment is used to record which type classes have to be defined for the comprehension to be typeable. In the case of list comprehensions such an environment is not needed as all involved functions are defined for lists. For the more general monad comprehensions this might not be the case. It is, for example, not required that an instance for $MonadGroup$ is available if a comprehension does not use **then group by** e qualifiers.

A rule of the form $\vdash\ w\ \Rightarrow\ \Delta$ is read as *the variables in pattern w have types described in environment* Δ. Rule of the form $P, \Gamma\ \vdash\ e\ :\ \tau$ are read as *under predicate environment P and type environment Γ expression e has the type τ*. Finally, $m\ \Delta$ is a shorthand for: $\{m\ \tau \mid \tau\ \in\ \Delta\}$.

We will now discuss the notable changes in the typing rules for basic monad comprehensions. In the rule $[\mathit{Comp}]$, the constraint $Monad\ \ m$ is added to the predicate environment to ensure that the resulting structure of the comprehension is indeed a monad. To deal with filters, a $zero$ function for monads is needed [28]. Such a function is provided by the type class $MonadPlus$. We thus add a constraint requiring an instance for $MonadPlus$ to the predicate environment in the rule $[\mathit{Guard}]$.

To cater for parallel monad comprehensions, the $MonadZip$ class was introduced (Section 3.1). The typing rule for parallel comprehensions is very similar to the rule for products $[\mathit{Comma}]$: on the type level, these two rules are indeed equivalent except for the additional $MonadZip$ constraint.

The typing rules for dealing with **then** clauses is very similar to the rules for SQL-like list comprehensions. The only notable difference is the added constraint in the $[\mathit{groupBy}]$ rule.

3.3 Translation rules

Desugaring basic monad comprehensions into monadic combinators has been discussed by Wadler [28]. A generalisation for the extensions to list comprehensions (SQL-like list comprehensions, and parallel comprehensions) has not been described before. In this section, we will discuss how monad comprehensions can be desugared. The strategy is very similar to the desugaring of list comprehensions presented by Wadler and Peyton Jones [25].

The desugaring rules are presented in Figure 10. The main difference with the desugaring of list comprehensions is that all list combinators have been replaced by their monadic counterparts. Putting an element into a list has been replaced by lifting an element into a monad using $return$ (let-bindings). Occurrences of map have been replaced with $liftM$ (Cartesian products, **then group** clauses and $unzip$). The function $concat$, originally used to flatten nested lists, is replaced by $join$. The difference between desugaring parallel monad comprehensions with parallel list comprehensions is the replacement of zip by $mzip$ (member of class $MonadZip$, Section 3.1). The group by clause uses $mgroupWith$ instead of $groupWith$.

4. Implementation

4.1 Summary of GHC changes

The implementation of monad comprehensions in GHC mainly affects the type checker and desugarer phases of the compiler.

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 $\boxed{P, \Gamma \vdash e \Rightarrow \Delta}$

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

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

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 $\boxed{P, \Gamma \vdash e \Rightarrow \Delta}$

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

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

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

Figure 9. Typing monad comprehensions.

For parts of the implementation, existing rules in both the type checker and desugarer have been reused (e.g., binding statements and pattern matches from **do**-notation). Other parts required more technical changes to existing rules (e.g., grouping and parallel statements).

Small changes have also been made to GHC's representation of syntax trees. More specifically, we changed the data type that represents the body of **do** blocks, list comprehensions, and monad comprehensions. This change was necessary so that the different types of qualifiers as well as rebindable syntax could be supported.

As stated in Section 3.1, the *MonadZip* laws require the *mzip* function to be associative. For law abiding *MonadZip* instances the implementation can desugar parallel monad comprehensions in a left or right associative manner without changing the program's semantics. The implementation in GHC is right associative.

The exact details of these changes to GHC, their motivation, and the alternatives considered are documented in a discussion on GHC Trac[4].

4.2 Error Messages

As discussed in the previous section, monad comprehensions are type checked *before* being desugared. Late desugaring enables the generation of warning and error messages that may refer to the actual code the programmer wrote. Error messages relating to monad comprehension thus should be as readable as their list comprehension counterparts.

Consider for example the following monad comprehension expression: $[(x, y) \mid x \leftarrow [1], y \leftarrow \mathit{Just}\ 5]$. In this expression, the generators draw elements out of different monadic structures. For endo-monadic comprehensions, this is forbidden, as it is not possible to determine a unique monadic type for the final result. GHC will emit the error message presented in Figure 11 when given an input file containing the above expression. The monad used in the first generator is the expected monadic type for any following generator. We believe that this error message accurately explains that the *Maybe* type is not compatible with the expected list type.

5. Proposals

The monad comprehension extension for Haskell described in this paper will be available in GHC 7.2. In this section we propose two additional extensions that are closely related to monad comprehensions, both of which we have not implemented yet. In Section 5.1 we discuss a proposal to extend the defaulting mechanism to remedy potential ambiguity errors. In Section 5.2 we propose a way for overloading list literals.

5.1 Defaulting Proposal

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

[4] `http://hackage.haskell.org/trac/ghc/ticket/4370`

$$
\begin{aligned}
[\, e \mid q \,] &= \mathit{liftM}\ (\lambda q_v \to e)[\![q]\!] \\[1ex]
[\![w \leftarrow e]\!] &= e \\
[\![\mathbf{let}\ w = d]\!] &= \mathit{return}\ d \\
[\![g]\!] &= \mathit{guard}\ g \\
[\![p, q]\!] &= \mathit{join}\ (\mathit{liftM} \\
&\qquad (\lambda p_v \to \mathit{liftM} \\
&\qquad\quad (\lambda q_v \to (p_v, q_v))[\![q]\!]) \\
&\qquad [\![p]\!]) \\
[\![p \mid q]\!] &= \mathit{mzip}[\![p]\!][\![q]\!] \\
[\![q, \mathbf{then}\ f]\!] &= f[\![q]\!] \\
[\![q, \mathbf{then}\ f\ \mathbf{by}\ e]\!] &= f\ (\lambda q_v \to e)\ [\![q]\!] \\
[\![q, \mathbf{then\ group\ by}\ e]\!] &= \mathit{liftM}\ \mathit{unzip}_{q_v} \\
&\qquad (\mathit{mgroupWith} \\
&\qquad\quad (\lambda q_v \to e)[\![q]\!]) \\
[\![q, \mathbf{then\ group\ by}\ e\ \mathbf{using}\ f]\!] &= \mathit{liftM}\ \mathit{unzip}_{q_v} \\
&\qquad (f\ (\lambda q_v \to e)[\![q]\!]) \\
[\![q, \mathbf{then\ group\ using}\ f]\!] &= (\mathit{liftM}\ \mathit{unzip}_{q_v}\ (f[\![q]\!])
\end{aligned}
$$

$$
\begin{aligned}
(w \leftarrow e)_v &= w \\
(\mathbf{let}\ w = d)_v &= w \\
(g)_v &= () \\
(p, q)_v &= (p_v, q_v) \\
(p \mid q)_v &= (p_v, q_v) \\
(q, \mathbf{then}\ f)_v &= q_v \\
(q, \mathbf{then}\ f\ \mathbf{by}\ e)_v &= q_v \\
(q, \mathbf{then\ group\ by}\ e)_v &= q_v \\
(q, \mathbf{then\ group\ by}\ e\ \mathbf{using}\ f)_v &= q_v \\
(q, \mathbf{then\ group\ using}\ f)_v &= q_v
\end{aligned}
$$

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

Figure 10. Desugaring monad comprehensions

```
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]
```

Figure 11. Monad comprehension error message.

uptake of the extension by the Haskell community and ease of integration into the existing standard.

Monad comprehensions, once part of the Haskell standard, were removed from the language [4, 24]. The reasons included monad-comprehensions–related error messages produced by Haskell implementations. The error messages were considered too complicated for new users of Haskell [15].

As we briefly discussed in Section 4, monad-comprehensions–related error messages produced by GHC are reasonable. In fact, the error messages are almost the same as for list comprehensions, the only difference being the mention of "monad comprehensions" instead of "list comprehensions". We do not consider this aspect of error messages as problematic.

Having said that, should the monad comprehensions extension become a part of the Haskell language standard, type ambiguity errors may occur that would not occur before. This may especially

$$
\begin{aligned}
&\mathbf{class}\ \mathit{IsList}\ l\ \mathbf{where} \\
&\quad \mathbf{type}\ \mathit{Item}\ l \\
&\quad \mathit{fromList} :: [\mathit{Item}\ l] \to l
\end{aligned}
$$

Figure 12. Definition of class IsList. The associated type synonym family Item is used to specify the type of list items from which the structure l is constructed.

be problematic for existing list-comprehensions–based code. In fact, John Hughes identified this problem as one of the main reasons that led to the removal of monad comprehensions from the Haskell standard[5].

To address the aforementioned problem, we propose to extend Haskell's defaulting mechanism to type classes and use it for disambiguation of comprehension-based code just like defaulting is used for disambiguation of numerical code; for example, the following declaration could be used to state that ambiguous type variables in the Monad class must default to lists:

$$\mathbf{default}\ \mathit{Monad}\ ([\,])$$

Note that currently Haskell's defaulting mechanism is only used to disambiguate type variables in the Num class.

The proposed defaulting mechanism would also affect monadic code written using the **do** notation or monadic combinators. In some cases, this behaviour may not be desirable as the type ambiguity error messages may point to problems that are better resolved manually (e.g., by providing explicit type signatures). Compiler warnings can be used to address this problem. GHC already supports the `-fwarn-type-defaults` compiler flag which can be used to warn users when type variables in the Num class are defaulted. A similar approach can be used for the type class defaulting proposal briefly discussed here.

The defaulting proposal discussed in this section is an informal one. Many details still need to be worked out. However, we think that it is still worthwhile to point out that Haskell already provides a language construct that can be generalised to address undesirable type ambiguity errors that arise from overloading.

5.2 Overloading List Literals

The $\mathit{MonadComprehensions}$ extension does not overload literal lists. We recognise that overloading literal lists can be useful for Haskell libraries and EDSLs. However, we think that the list literal overloading should *not* be tied to the concept of monad because the overloading can be useful for Haskell structures that are not monads (e.g., for sets, bags, maps and hash tables).

We propose to introduce an extension that supports overloading of literal lists by using the type class definition that is given in Figure 12. The idea is that when the extension is turned on the Haskell compiler applies the *fromList* function to every literal list in the source code.

The IsList instance for lists given in Figure 13 would allow literal lists to represent the same values as in standard Haskell. The IsList instances for the Set and Map data types given in Figure 14, on the other hand, would allow the use of literal lists for construction of sets and maps. The type class defaulting proposal given in Section 5.1 would also be useful for disambiguating code involving overloaded literal lists.

The proposed definitions of the IsList type class and its instances make use of the $\mathit{TypeFamilies}$ extension. Equivalent definitions can also be written using the $\mathit{FunctionalDependencies}$ extension.

[5] `http://web.archive.org/web/20010720221805/www.dcs.gla.ac.uk/mail-www/haskell/msg00172.html`

```
instance IsList [α] where
  type Item [α] = α
  fromList = id
```

Figure 13. Definition of *IsList* instance for lists.

```
import qualified Data.Set as Set
import Data.Set (Set)

import qualified Data.Map as Map
import Data.Map (Map)

instance (Ord α) ⇒ IsList (Set α) where
  type Item (Set α) = α
  fromList = Set.fromList

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

Figure 14. Definition of *IsList* instances for the *Set*, and *Map* data types.

6. Related Work

Any monad immediately yields its associated comprehension structure (see Section 3). Conversely, any comprehension structure gives rise to a monad. Philip Wadler [28] has been the first to observe that monads and comprehensions are inseparable. We could go as far and argue that a language's support for monads is incomplete as long as it lacks comprehensions.

From the early days of NPL [8] and KRC [27] until today, comprehensions have indeed found their way into a large number of programming languages, including some in the mainstream. Variants of comprehension syntax are found, for example, in Erlang, Perl 6, and Python. These languages, however, fix a small set of collection types over which comprehensions may be defined—lists, sets, or dictionaries are typical. True general monad comprehensions were first provided by Gofer [17] and applications of comprehensions over non-collection monads (e.g., for parser construction [16]) readily appeared.

Monad operations have been deeply built into LINQ [21, 22], a framework that seamlessly integrates queries into the languages supported by the .NET platform (C#, F#, and Visual Basic): LINQ's `SelectMany` operator is the monadic bind ($\gg=$), for example. Based on these operations, LINQ defines a comprehension-style syntax that provides a uniform way to query a diversity of collection types, including XML documents, relational tables, and class extents. To date, LINQ may well be considered as *the* programming environment that popularised monad comprehensions.

Outside the programming language community, monad comprehensions had a notable impact on the study of the semantics and optimisation of database query languages [7, 13, 30]. Many query languages—this includes LINQ, SQL, XQuery—may be understood in terms of (1) operations that are specific to a given data model, and (2) a "backbone" that provides generic iteration facilities. Monad comprehensions have been found to provide an ideal backbone: their syntax quite closely resembles that of the pervasive relational calculus while their semantics constitute a considerable generalisation of the calculus. Comprehensions can uniformly describe iteration, joins, as well as grouping, an observation that led to the SQL-like comprehensions (see Section 2.4) [25] and motivated our use of monad comprehensions as the query surface syntax in DSH (Section 2.7). Aggregation, quantification, and updates have been expressed in terms of comprehensions as well [11, 26].

Monad comprehensions succeed in extracting and emphasising the structural gist of a query rather than to stress the diversity of query constructs. Different types of query nesting lead to few nested forms of monad comprehensions. Much of the seminal work on query optimisation and unnesting [18] can be understood in terms of simple syntactical rewrites, the monad comprehension normalisation rules [14, 28, 29].

In [10], Martin Erwig discusses a generalised comprehension syntax for any abstract data type (ADT) whose constructors and destructors can be combined to define a monad. These monads feature a bind operation that is parameterised by source and target ADTs, ultimately resulting in ADT comprehensions that may be non-endomorphic and thus allow mappings between different types.

7. Conclusions

We presented a Glasgow Haskell Compiler extension that generalises Haskell's list comprehension notation to monads. The extension implements well-known generalisations of generator and filter clauses, as well as new generalisations of parallel and SQL-like clauses. We formally described the generalisations. The formal description allows other Haskell implementations to incorporate the extension.

We hope that the extensive set of instructive examples presented in this paper will facilitate wide adoption of the monad comprehensions extension by the Haskell community. We briefly discussed how to integrate monad comprehensions in the Haskell language standard. Also we proposed an extension that allows overloading of literal lists. A thorough treatment of both the integration into the language standard and overloading of literal lists are subject of future work.

Acknowledgments

We are indebted to Simon Peyton Jones for advising us on how to integrate monad comprehensions into GHC and for his efforts in modifying our patch and applying it to the GHC source tree. We are grateful to Jeremy Gibbons, Tomas Petricek and the anonymous referees for their valuable comments that helped to improve this paper. Jeroen and Nils have been supported by the German Research Foundation (DFG), Grant GR 2036/3-1. George has received support from the ScienceCampus Tübingen: Informational Environments.

References

[1] Database-Supported Haskell. `http://hackage.haskell.org/package/DSH`.

[2] Hugs - a functional programming system based on Haskell 98. `http://www.haskell.org/hugs/`.

[3] The Glasgow Haskell Compiler. `http://www.haskell.org/ghc/`.

[4] Haskell 1.4: A Non-Strict, Purely Functional Language, 1997.

[5] Guy Blelloch. Scans as Primitive Parallel Operations. *IEEE Transactions on Computers*, 38(11), 1989.

[6] Guy Blelloch. NESL: A nested data-parallel language. Technical report, Pittsburgh, PA, USA, 1992.

[7] Peter Buneman, Leonid Libkin, Dan Suciu, Val Tannen, and Limsoon Wong. Comprehension Syntax. *SIGMOD Record*, 23:87–96, 1994.

[8] Rod Burstall and John Darlington. Transformation Systems for Developing Recursive Programs. *Journal of the ACM*, 24(1):44–67, 1977.

[9] Manuel M. T. Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, Gabriele Keller, and Simon Marlow. Data Parallel Haskell: A Status Report. In *Proceedings of the 2007 Workshop on Declarative Aspects of Multicore Programming*, DAMP '07, pages 10–18, Nice, France, 2007. ACM.

[10] Martin Erwig. Comprehending ADTs. Unpublished draft, `http://web.engr.oregonstate.edu/~erwig/adtfold/compr.pdf`, 2011.

[11] Leonidas Fegaras and David Maier. Towards an Effective Calculus for Object Query Languages. In *Proceedings of the SIGMOD Conference*, San Jose, CA, USA, 1995. ACM.

[12] George Giorgidze, Torsten Grust, Tom Schreiber, and Jeroen Weijers. Haskell boards the Ferry: Database-supported program execution for Haskell. In *Revised selected papers of the 22nd international symposium on Implementation and Application of Functional Languages, Alphen aan den Rijn, Netherlands*, volume 6647 of *Lecture Notes in Computer Science*. Springer, 2010. Peter Landin Prize for the best paper at IFL 2010.

[13] Torsten Grust. How to Comprehend Queries Functionally. *Intelligent Information Systems*, 12(2/3):191–218, 1999.

[14] Torsten Grust. Monad Comprehensions: A Versatile Representation for Queries. In Peter M.D. Gray, Larry Kerschberg, Peter J.H. King, and Alexandra Poulovassilis, editors, *The Functional Approach to Data Management*. Springer, 2003.

[15] Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. A History of Haskell: Being Lazy with Class. In *Proceedings of the third ACM SIGPLAN Conference on History of Programming Languages*, HOPL III, pages 1–55, San Diego, CA, USA, 2007. ACM.

[16] Graham Hutton and Erik Meijer. Monadic Parser Combinators. Technical Report NOTTCS-TR-96-4, University of Nottingham, Nottingham, UK, 1996.

[17] Mark Jones. The Implementation of the Gofer Functional Programming System. Technical Report YALEU/DCS/RR-1030, Yale University, New Haven, CT, USA, 1994.

[18] Won Kim. On Optimizing an SQL-like Nested Query. *Transactions on Database Systems (TODS)*, 7(3), 1982.

[19] Jeff Lewis. Cryptol: specification, implementation and verification of high-grade cryptographic applications. In *Proceedings of the 2007 ACM workshop on Formal methods in security engineering*, FMSE '07, pages 41–41, New York, NY, USA, 2007. ACM.

[20] Conor McBride and Ross Paterson. Applicative programming with effects. *Journal of Functional Programming*, 18(01):1–13, 2008.

[21] Erik Meijer and Brian Beckman. XLinq: XML Programming Refactored (The Return of the Monoids). In *Proceedings of the XML Conference*, Atlanta, GA, USA, 2005.

[22] Erik Meijer, Brian Beckman, and Gavin Bierman. LINQ: Reconciling Objects, Relations, and XML in the .NET Framework. In *Proceedings of the SIGMOD Conference*, Chicago, IL, USA, 2006. ACM.

[23] Tomas Petricek. Fun with parallel monad comprehensions. *The Monad Reader Issue 18*, 2011. `http://themonadreader.wordpress.com/`.

[24] Simon Peyton Jones, editor. *Haskell 98 Language and Libraries – The Revised Report*. Cambridge University Press, Cambridge, England, 2003.

[25] Simon Peyton Jones and Philip Wadler. Comprehensive Comprehensions. In *Proceedings of the ACM SIGPLAN Workshop on Haskell*, pages 61–72, Freiburg, Germany, 2007. ACM.

[26] Phil Trinder. Comprehensions, a Query Notation for DBPLs. In *Proceedings of the 3rd International Workshop on Database Programming Languages*, DBPL, Nafplion, Greece, 1991.

[27] David Turner. The Semantic Elegance of Applicative Languages. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, FPCA. ACM, 1981.

[28] Philip Wadler. Comprehending Monads. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, LFP '90, pages 61–78, Nice, France, 1990. ACM.

[29] Limsoon Wong. *Querying Nested Collections*. PhD thesis, University of Pennsylvania, Philadelphia, PA, USA, 1994.

[30] Limsoon Wong. Kleisli, A Functional Query System. *Journal of Functional Programming*, 10(1):19–56, 2000.