Comprehending ADTs*

Martin Erwig
School of EECS
Oregon State University
erwig@eecs.oregonstate.edu

Abstract

We show how to generalize list comprehensions to work on abstract data types. First, we make comprehension notation automatically available for any data type that is specified as a constructor/destructor-pair (bialgebra). Second, we extend comprehensions to enable the use of different types in one comprehension and to allow to map between different types. Third, we refine the translation of comprehensions to give a reasonable behavior even for types that do not obey the monad laws, which significantly extends the scope of comprehensions.

Keywords: Functional Programming, Comprehension Syntax, Abstract Data Type, Generalized Fold, Monad

1 Introduction

Set comprehensions are a well-known and appreciated notation from mathematics: an expression such as $\{E(x) \mid x \in S, P(x)\}$ denotes the set of values given by repeatedly evaluating the expression E for all x taken from the set S for which $P(x)$ yields true. This notation has found its way into functional languages as *list comprehensions* [3, 17, 18]. Assuming that s denotes a list of numbers, the following Haskell expression computes squares of the odd numbers in s.

$$[\, x * x \mid x \leftarrow s,\ \mathit{odd}\ x \,]$$

The main difference to the mathematical set comprehensions is the use of *lists* in the generator and as a result. In general, a comprehension consists of an expression defining the result values and of a list of *qualifiers*, which are either *generators* for supplying variable bindings or *filters* restricting these.

Phil Wadler has shown that the comprehension syntax can be used not only for lists and sets, but more generally for arbitrary monads [19]. Beyond their use in programming languages, comprehensions have also become quite popular as query languages for databases [16, 9, 11, 15]. One reason for this might be that comprehensions provide an intuitive notation for (possibly nested) iterations combined with selection conditions.

The meaning of comprehensions is defined by translation into two basic monad operations *return* and *bind*. Thus, to use the comprehension notation for a particular data type T it is sufficient to define these two operations for T. In order for comprehensions to be well defined, the operations must satisfy three laws; we will describe the operations and their laws in Section 2.

*This is a slightly revised version of an old draft paper that was written during my time at the University of Hagen.

It is not always obvious to find definitions for *return* and *bind*, and the first contribution of this paper is a higher-level interface to the definition of monadic structures: whenever a data type is defined as a particular kind of ADT (as defined in [6]), the monad operations and thus also the comprehension syntax is obtained for free. Defining an ADT is fairly simple, it essentially means to write down its constructors and destructors in a predefined shape. We will explain this in Section 3.

In Section 4 we then show how to define monad operations based on ADTs, which provides an alternative way to define monads, namely through ADTs. This also offers an alternative to defining comprehension syntax for data types. In addition, the definition of ADT monad operations will also reveal opportunities for making comprehensions even more expressible.

We will define a generalization of the comprehension syntax and a corresponding translation into ADT monad operations in Section 5. In particular, it will then be possible to query different monads and to map into monads that are different from the generators. Finally, conclusions follow in Section 6.

Let us give a few examples. For instance, we can define an ADT *nat* which constructs and destructs natural numbers by computing successors and predecessors, respectively. (The definition of *nat* will be shown and explained in Section 3.) Basing comprehensions on ADT con/destructors, we can use, for example, a natural number as a generator.

$$countdown\ n\ =\ [\ i \mid i \leftarrow n\ ::: nat\ ]\ ::: list$$

The specification "::: *nat*" says that the number value n is to be decomposed as specified by the ADT *nat*; in this case, it yields a decreasing sequence of numbers. In contrast, the outer specification "::: *list*" tells to process the values i with the constructor of the ADT *list*, which means here to simply collect them in a list. By adding a condition, we get a function for computing all of a number's divisors.

$$divisors\ n\ =\ [\ i \mid i \leftarrow n\ ::: nat,\ n\ `mod`\ i == 0\ ]\ ::: list$$

In the definition of *countdown* we could, instead of *list*, as well use an ADT like *prod*, which constructs numbers by multiplication, so that we obtain a comprehension defining the factorial function. If we omit the outer ADT specification, *list* is assumed by default, and if we omit ADT specifications from the generators, the outer ADT specification is taken as a default value. Thus, without any ADT specification, we arrive at plain list comprehensions. Now if we assume that *prod* has the same destructor as *nat*, we can express the factorial function concisely by the following comprehension.

$$fac\ n\ =\ [\ i \mid i \leftarrow n\ ]\ ::: prod$$

The generalized comprehension syntax gets really interesting when we work with more complex data types. Assume we have an ADT *graph* whose constructor inserts a single node together with incoming and outgoing edges and whose destructor removes a single node with all incident edges (see Section 3.5 and [4]). Then we can, for example, define the graph reverse operation simply by the following comprehension.

$$grev\ g\ =\ [\ (s, v, p) \mid (p, v, s) \leftarrow g\ ]\ ::: graph$$

This means that nodes are successively taken from the graph together with the list of their successors (s) and predecessors (p), and the nodes are reinserted with exchanged roles of the successor/predecessor lists.

With a final examples we demonstrate the huge potential of ADT comprehensions. Assume we have an ADT *rootGraph* (for acyclic graphs) whose destructor is defined to always return and remove a root node (that is, a node having no predecessors). This ADT allows us to define a function for topological sorting simply by systematic graph decomposition within a comprehension.

$$topsort\ g\ =\ [\ v \mid (\_, v, \_) \leftarrow g\ ::: rootGraph\ ]$$

All the presented examples have in common the general approach of defining ADTs "in the righ way" so that the iteration schema provided by the comprehension syntax delivers the intended results.

2 Monads and Comprehensions

The comprehension syntax can be defined not only for lists, but more generally for any data types that is a *monad*. The notion of monad originated in category theory [12]. Eugenio Moggi [13, 14] used monads to structure semantics definitions, which paved the way for using monads in functional languages [20]. A good survey is given by Phil Wadler in [21]. For this paper it is sufficient to regard a monad as a unary type constructor with two associated functions. In Haskell, this is expressed by a type class (more precisely, as a constructor class) *Monad*.[1]

```
class Monad m where
  return :: a  →  m a
  bind   :: m a  →  (a  →  m b)  →  m b
```

This definition says that any type constructor m can be made into a monad by providing the two operations shown. In addition, the monadic structure requires *return* to be a left and right unit of *bind* and *bind* to be associative in a certain sense. This is captured by the following laws.

$$(return\ x)\ `bind`\ f \;=\; f\ x \qquad \text{(LU)}$$
$$m\ `bind`\ return \;=\; m \qquad \text{(RU)}$$
$$(m\ `bind`\ f)\ `bind`\ g \;=\; m\ `bind`\ (\backslash x \rightarrow f\ x\ `bind`\ g) \qquad \text{(A)}$$

These laws are not enforced by Haskell. It is the programmer's responsibility to use only appropriate monad types.

One of the most prominent monad examples is the type of lists. The type constructor [ ] is made an instance of the monad class by the following definitions.

```
instance Monad [ ] where
  return x      =  [x]
  bind [ ] f    =  [ ]
  bind (x:xs) f =  f x++bind xs f
```

This means that *return* just injects an element into a singleton list and that *bind* applies f to each element of a list producing a list of singleton lists that are concatenated by ++.

For each monad the comprehension syntax (without conditions) is defined by the following translation.

$$\mathcal{T}([\ e \mid\ ]) \;=\; return\ e$$
$$\mathcal{T}([\ e \mid x \leftarrow g,\ gs\ ]) \;=\; g\ `bind`\ (\backslash x \rightarrow \mathcal{T}([\ e \mid gs\ ]))$$

In some cases monads have a richer structure and offer also a zero element. This is reflected by the extended class *MonadZero*.

```
class Monad m ⇒ MonadZero m where
  zero :: m a
```

[1] In the Haskell 98 standard, an expression m `` `bind` `` f is written as $m >>= f$. We use the name *bind* instead since we later generalize the *bind* and *return* operations by adding an ADT parameter. Since this extension will cause *bind* to be a ternary operation, an infix symbol doesn't work anymore as syntax.

For instances of *MonadZero* the following laws should hold.

$$zero \text{ `}bind\text{`} f = zero \quad \text{(LZ)}$$
$$m \text{ `}bind\text{`} \backslash x \to zero = zero \quad \text{(RZ)}$$

Whenever a type is an instance of *MonadZero*, comprehensions can also contain filter predicates in addition to generators. These are translated as follows.

$$\mathcal{T}([\, e \mid p,\ gs \,]) = \textbf{if } p \textbf{ then } \mathcal{T}([\, e \mid gs \,]) \textbf{ else } zero$$

The described translation is reasonable because of the above mentioned monad laws. In contrast, the use of comprehension syntax for structures not satisfying these laws can lead to unexpected results. One can well consider the use of comprehensions for such non-monadic types an error—this tacitly presumes the described translation to be fixed. On the other hand, one can also try to make the comprehension notation meaningful to a larger class of types, which, in fact, calls for a different translation. We will investigate this issue in detail in Section 5.

Example calculations showing the translation of list comprehensions in detail can be found, for example, in [18].

3 ADTs and Catamorphisms

In this section we sketch an approach to define fold operations for abstract data types. For a more comprehensive introduction, see [6]; the formal categorical background is developed in [5, 7].

Note that the purpose of this section is not to lobby for the general use of this specific ADT concept. We rather take this approach here as a basis for translating comprehensions, because it perfectly seems to fit the requirements. Therefore, most of the discussion of related work regarding catamorphisms is not repeated, cf. [5, 6]. We just mention that other work on catamorphisms has almost exclusively focussed on algebraic data types (see [2] and the contained references), one exception is Fokkinga's work [10]. The present framework covers and extends previous approaches at the cost of weakening some universal properties.

We deliberately have chosen a presentation based on Haskell syntax (in favor of the more general categorical notation) since (i) this facilitates the comparison with the traditional monad approach and (ii) all examples can be directly tested with any Haskell implementation.

3.1 Representation of Algebraic Data Types

The definition of a data type T introduces a set of constructors c_1, ..., c_n which can be viewed as functions of types $T_1 \to T$, ..., $T_n \to T$. The common result type T is also called the *carrier* of the data type. For instance, the list data type ($T = [a]$) is defined by the two constructors $[\,] :: [a]$ (with empty argument type) and $(:) :: a \to [a] \to [a]$. In order to define generic operations (such as *fold*) for arbitrary data types we need a way of encoding data types in Haskell itself. This can be achieved by combining all constructors into one constructor mapping from the separated sum of argument types to the carrier. The union of argument types can be represented by specific type constructors. For example, the argument type structure for the list data type is captured by the following type constructor.

$$\textbf{data } Binary\ a\ b = Unit_B \mid Two\ a\ b$$

The type $Binary\ a\ [a]$ is the proper argument type for the combined list constructor, which can then be defined as follows.

$$
\begin{array}{l}
cList :: Binary\ a\ [a] \rightarrow [a] \\
cList\ Unit_B \quad\ = [\,] \\
cList\ (Two\ x\ xs) = x{:}xs
\end{array}
$$

As another example consider a data type for natural numbers comprised of a zero constant and a successor function. To combine both into one constructor we need a slightly different type constructor to represent the argument type structure.

$$\textbf{data}\ Unary\ a = Unit_U \mid One\ a$$

With *Unary* we can define the constructor using built-in integer numbers as the carrier.

$$
\begin{array}{l}
cNat :: Unary\ Int \rightarrow Int \\
cNat\ Unit_U \quad = 0 \\
cNat\ (One\ n) = succ\ n
\end{array}
$$

Thus, in general, an algebraic data type can be regarded simply as a function $c :: f\ t \rightarrow t$ where f is the type constructor representing the argument type union of the constructor(s) c.

3.2 Type Constructors as Functors

As far as constructors and algebraic data types are concerned we can use arbitrary type constructors to describe the argument type structure. However, the definition of ADT fold to be given below requires that destructors (the dual of constructors) be defined with result types that are expressed by type constructors that offer a *map* function. In Haskell there is a predefined constructor class *Functor* for these types.

$$
\begin{array}{l}
\textbf{class}\ Functor\ f\ \textbf{where} \\
\quad fmap :: (t \rightarrow u) \rightarrow (f\ t \rightarrow f\ u)
\end{array}
$$

Binary and *Unary* are both examples of functors. The corresponding instance declarations are as follows.

$$
\begin{array}{l}
\textbf{instance}\ Functor\ (Binary\ a)\ \textbf{where} \\
\quad fmap\ f\ Unit_B \quad\ = Unit_B \\
\quad fmap\ f\ (Two\ x\ y) = Two\ x\ (f\ y) \\
\\
\textbf{instance}\ Functor\ Unary\ \textbf{where} \\
\quad fmap\ f\ Unit_U \quad = Unit_U \\
\quad fmap\ f\ (One\ x) = One\ (f\ x)
\end{array}
$$

These functors are also called *pattern functors* [1].

The definitions for *cList* and *cNat* have a uniform shape, and in fact, most of the constructor and destructor definitions are of a specific form that can be captured by standardized functions to map from and to functor type constructors. For example, a canonical way of mapping from and to *Binary* types is provided by the following functions.

$$
\begin{array}{l}
from_B :: t \rightarrow (a \rightarrow b \rightarrow t) \rightarrow Binary\ a\ b \rightarrow t \\
from_B\ u\ f\ Unit_B \quad\ = u \\
from_B\ u\ f\ (Two\ x\ y) = f\ x\ y \\
\\
to_B :: (t \rightarrow Bool) \rightarrow (t \rightarrow a) \rightarrow (t \rightarrow b) \rightarrow t \rightarrow Binary\ a\ b \\
to_B\ p\ f\ g\ x = \textbf{if}\ p\ x\ \textbf{then}\ Unit_B\ \textbf{else}\ Two\ (f\ x)\ (g\ x)
\end{array}
$$

These functions allow us to define the list constructor much more succinctly as follows.

$$cList \;=\; from_B\ [\,]\ (:)$$

Similarly, we can define $from_U$ and to_U.

$$\begin{aligned}
&from_U :: t \to (a \to t) \to Unary\ a \to t\\
&from_U\ u\ f\ Unit_U \;=\; u\\
&from_U\ u\ f\ (One\ x) \;=\; f\ x
\end{aligned}$$

$$\begin{aligned}
&to_U :: (t \to Bool) \to (t \to a) \to (t \to Unary\ a)\\
&to_U\ p\ f\ x \;=\; \textbf{if}\ p\ x\ \textbf{then}\ Unit_U\ \textbf{else}\ One\ (f\ x)
\end{aligned}$$

We can then also give the following definition for *cNat*.

$$cNat \;=\; from_U\ 0\ succ$$

Applications of to_B and to_U follow below.

3.3 Destructors and ADTs

A *data type destructor* is the dual of a constructor, that is, a function $d :: t \to g\ t$ mapping the carrier to, for example, a union of possible result types. For each algebraic data type $cT :: f\ t \to t$ we can easily define its canonical destructor $dT :: t \to f\ t$ by simply flipping both sides of the definition. For example, the canonical list destructor is defined as follows.

$$\begin{aligned}
&dList :: [a] \to Binary\ a\ [a]\\
&dList\ [\,] \;=\; Unit_B\\
&dList\ (x : xs) \;=\; Two\ x\ xs
\end{aligned}$$

Again, we can give a shorter version by using to_B, the dual of $from_B$.

$$dList \;=\; to_B\ null\ head\ tail$$

The definition of the canonical destructor for *cNat* is similar.

$$dNat \;=\; to_U\ (== 0)\ pred$$

We can also define binary *nat* destructors, for example, a destructor *dRng* that returns the number to be decomposed in addition to the decremented value.

$$dRng \;=\; to_B\ (== 0)\ id\ pred$$

An abstract data type can can be defined as a pair consisting of a constructor and a destructor with a common carrier type (formally, an ADT is defined as a bialgebra [10, 5]). We need no restriction on the argument type of the constructor (and we therefore represent it by a simple variable instead of a type expression), but we require the result type of the destructor to be given as an application of a functor to the carrier type. This is necessary since the definition of *fold* uses the function *fmap* to fold recursive occurrences of t-values.

$$\textbf{data}\ Functor\ g \Rightarrow ADT\ s\ g\ t \;=\; ADT\ (s \to t)\ (t \to g\ t)$$

The con/destructor of an ADT can be extracted with the following functions.

$$con\ (ADT\ c\ \_)\ =\ c$$
$$des\ (ADT\ \_\ d)\ =\ d$$

Occasionally, we will use the following type abbreviations for ADTs that are symmetrical in their constructor/destructor type structure.

$$\textbf{type}\ SymADT\ g\ t\ =\ ADT\ (g\ t)\ g\ t$$
$$\textbf{type}\ BinADT\ a\ t\ =\ SymADT\ (Binary\ a)\ t$$

With these definitions we can define a *list* or a *count* ADT simply as follows.

$$list\ ::\ BinADT\ a\ [a]$$
$$list\ =\ ADT\ cList\ dList$$

$$count\ ::\ SymADT\ Unary\ Int$$
$$count\ =\ ADT\ cNat\ dNat$$

We are, of course, not constrained to symmetric ADTs. We can define many ADT variants by exchanging either the constructor or the destructor. Concerning natural numbers, instead of the unary definition for *count* given above, we can alternatively use multiplication as a binary constructor or *dRng* as a binary destructor. We can even take a completely binary view as shown below by *nat* and *prod* (which we actually used in the examples of the Introduction).

$$rng\ ::\ ADT\ (Unary\ Int)\ (Binary\ Int)\ Int$$
$$rng\ =\ ADT\ cNat\ dRng$$

$$nat\ ::\ BinADT\ Int\ Int$$
$$nat\ =\ ADT\ (from_B\ 0\ (\backslash\_\ n \rightarrow n + 1))\ dRng$$

$$prod\ ::\ BinADT\ Int\ Int$$
$$prod\ =\ ADT\ (from_B\ 1\ (*))\ dRng$$

Many more examples can be found in [6].

3.4 ADT Folds and Transformers

Fold operations on algebraic data types are typically defined with the help of pattern matching: applied to a value v, *fold* determines the outermost constructor and applies an appropriate parameter function (that conforms to the type of the disclosed constructor). Formally, *fold* is defined as a homomorphism from the argument type to some result type, and this definition works only if the result type is a quotient of the argument type (because otherwise the homomorphism would not be uniquely defined). In other words, *fold* cannot map to less constrained structures. This is the reason that, for example, counting the elements of a set *cannot* be expressed as a fold operation. This restriction can be lifted if fold is not based on pattern matching, but on explicitly defined data type destructors.

Folding an ADT value of type t with a parameter function f then works as follows: first, the ADT destructor is applied, yielding a value x of type $g\ t$. Intuitively, one part of x contains values that are taken from (or split off) the ADT, and the other part represents the recursive occurrence(s) of t-values. The recursive part is then folded, followed by an application of f to the result and the non-recursive part of x. The recursive folding step is realized by using the function *fmap* that is defined for the functor g, which explains the requirement that the type of the ADT destructors is expressed by a functor.

$$\begin{aligned}&\mathit{fold} :: \mathit{Functor}\ g \Rightarrow (g\ u \to u) \to \mathit{ADT}\ s\ g\ t \to t \to u\\&\mathit{fold}\ f\ a = f\ .\ \mathit{fmap}\ (\mathit{fold}\ f\ a)\ .\ \mathit{des}\ a\end{aligned}$$

We observe that *fold*'s parameter function must map from the pattern functor of the ADT to the result type. For example, a function that multiplies all numbers in a list can be written as a fold:

$$\begin{aligned}&\mathit{mult} :: \mathit{Num}\ a \Rightarrow [a] \to a\\&\mathit{mult} = \mathit{fold}\ (\mathit{from}_B\ 1\ (*))\ \mathit{list}\end{aligned}$$

In this example it is striking that the parameter function of fold is nothing but a constructor of an ADT, namely the ADT *prod*. This special case occurs very often and is important enough to warrant a separate definition. We call this kind of fold an *ADT transformer*. In order to be able to transform an ADT with a result type functor g into an ADT whose argument type does not match g (that is, whose argument type cannot be expressed by an application of g) we include a parameter function (a *natural transformation*) that can be used to adjust the two type structures.

$$\begin{aligned}&\mathit{trans} :: (\mathit{Functor}\ g,\ \mathit{Functor}\ h) \Rightarrow (g\ u \to r) \to \mathit{ADT}\ s\ g\ t \to \mathit{ADT}\ r\ h\ u \to (t \to u)\\&\mathit{trans}\ f\ a\ b = \mathit{con}\ b\ .\ f\ .\ \mathit{fmap}\ (\mathit{trans}\ f\ a\ b)\ .\ \mathit{des}\ a\end{aligned}$$

In an expression *trans f a b* we call *a* the *source ADT* and *b* the *target ADT*; *f* is called the *map* of the transformer. The definition relates *trans* and *fold* in an obvious way.

$$\mathit{trans}\ f\ a\ b = \mathit{fold}\ (\mathit{con}\ b\ .\ f)\ a \qquad \text{(TransFold)}$$

When the type structures of the source and the target ADT agree, we have $f = \mathit{id}$. For this case we introduce the following abbreviation.

$$\mathit{transit} = \mathit{trans}\ \mathit{id}$$

We can thus rewrite the above *mult* example as follows.

$$\mathit{mult} = \mathit{transit}\ \mathit{list}\ \mathit{prod}$$

To understand the need for the additional parameter of *trans*, consider the task of determining the length of a list. We could express this function as a simple transformer from *list* to *nat*. However, the results delivered by *dList* do not have the proper shape for applying *cNat*, and we must provide a map p_2 from *Binary* to *Unary*:

$$\begin{aligned}\mathit{length} = {}& \mathit{trans}\ p_2\ \mathit{list}\ \mathit{count}\\&\textbf{where}\ p_2\ \mathit{Unit}_B = \mathit{Unit}_U\\&\qquad p_2\ (\mathit{Two}\ \_\ y) = \mathit{One}\ y\end{aligned}$$

By composing two or more transformers we can build streams of ADTs which can be used like filters. The most common case is to compose two transformers:

$$\begin{aligned}&\mathit{via} :: (\mathit{Functor}\ g,\ \mathit{Functor}\ h,\ \mathit{Functor}\ i) \Rightarrow\\&\qquad \mathit{ADT}\ s\ g\ t \to \mathit{ADT}\ (g\ u)\ h\ u \to \mathit{ADT}\ (h\ v)\ i\ v \to t \to v\\&\mathit{via}\ a\ b\ c = \mathit{transit}\ b\ c\ .\ \mathit{transit}\ a\ b\end{aligned}$$

With *via* we can give, for example, a nice implementation of heapsort:

$$\mathit{heapsort} = \mathit{via}\ \mathit{list}\ \mathit{heap}\ \mathit{list}$$

3.5 A Graph ADT

We close this section with an advanced ADT example that at the same time provides examples for more sophisticated ADT comprehensions. To define a graph ADT we can use the inductive definition of graphs introduced in [4, 8]: a graph is either empty, or it is constructed by adding a node together with edges from/to its predecessors/successors. Let *Node* be the type of node values, and let *Graph* be the type of directed, unlabeled graphs. A *node context* is a node together with a list of successors (third tuple component) and a list of predecessors (first component).

$$\textbf{type } \mathit{Context} \;=\; ([\mathit{Node}], \mathit{Node}, [\mathit{Node}])$$

Then we can define the following two graph constructors:

$$\begin{aligned} &\mathit{empty} \;::\; \mathit{Graph} \\ &\mathit{embed} \;::\; \mathit{Context} \;\to\; \mathit{Graph} \;\to\; \mathit{Graph} \end{aligned}$$

Note that *embed* yields a runtime error if either the node to be inserted is already present in the graph or if any of the predecessor or successor nodes does not exist in the graph. Both constructors can be combined as follows.

$$\begin{aligned} &\mathit{cGraph} \;::\; \mathit{Binary\ Context\ Graph} \;\to\; \mathit{Graph} \\ &\mathit{cGraph} \;=\; \mathit{from}_B\ \mathit{empty\ embed} \end{aligned}$$

A graph can be decomposed by successively extracting node contexts, that is, nodes together with their incident edges. We assume having a destructor *matchAny* that selects and removes an arbitrary node, that is, it returns the node context and the remaining part of the graph after removing the node and its incident edges. If the graph is empty, *matchAny* yields Unit_B.

$$\mathit{matchAny} :: \mathit{Graph} \;\to\; \mathit{Binary\ Context\ Graph}$$

Now we can easily define an ADT for unlabeled graphs.

$$\begin{aligned} &\mathit{graph} \;::\; \mathit{BinADT\ Context\ Graph} \\ &\mathit{graph} \;=\; \mathit{ADT\ cGraph\ matchAny} \end{aligned}$$

As an example for a transformer into graphs, consider the following function that builds a graph from a list of contexts.

$$\begin{aligned} &\mathit{build} \;::\; [\mathit{Context}] \;\to\; \mathit{Graph} \\ &\mathit{build} \;=\; \mathit{transit\ list\ graph} \end{aligned}$$

With transformers out of graphs we can compute, for example, a list of a graph's nodes.

$$\begin{aligned} &\mathit{nodes} \;::\; \mathit{Graph} \;\to\; [\mathit{Node}] \\ &\mathit{nodes} \;=\; \mathit{trans\ getNode\ graph\ list} \\ &\qquad \textbf{where } \mathit{getNode}\ \mathit{Unit}_B \;=\; \mathit{Unit}_B \\ &\qquad\qquad \mathit{getNode}\ (\mathit{Two}\ (\_, v, \_)\ g) \;=\; \mathit{Two}\ v\ g \end{aligned}$$

These two functions can be written nicely as ADT comprehensions as follows.

$$\begin{aligned} &\mathit{build\ cs} \;=\; [\ c \mid c \leftarrow \mathit{cs} ::: \mathit{list}\ ] ::: \mathit{graph} \\ &\mathit{nodes}\ g \;=\; [\ v \mid (\_, v, \_) \leftarrow g ::: \mathit{graph}\ ] \end{aligned}$$

4 ADTs and Monads

ADTs are a simple, yet very powerful abstraction—simple enough to allow data types to be easily defined as ADTs and having at the same time sufficient structure to define monad operations.

To make the comprehension syntax automatically available for ADTs, the class *ADT* had to be defined as a superclass of *Monad*, and the monad operations had to be implemented in terms of ADT operations. However, this does not work well for at least three reasons: first, the *Monad* class is already built into Haskell whereas the class *ADT* is not, so to make *ADT* a superclass of *Monad* we had to change the standard prelude of Haskell, which might not be desirable. Second, even if it were possible to make *Monad* a subclass of *ADT*, this would mean that *all* monads had to be defined as ADTs, which would be too strong a requirement. Third, the current comprehension syntax is not rich enough to support "non-endomorphic" comprehensions,[2] so that we need an own notation for ADT comprehensions anyhow.

Hence, we shall define an extension of the comprehension syntax and give a translation into Haskell. But before we get to this extension, it is instructive to consider how to define the monad operations for ADTs and how current comprehension "technology" works for ADTs because the limitations we encounter suggest a generalization of the comprehension syntax and semantics. In the following we focus on ADTs having the pattern functor *Binary*. As a running example we use the ADT *list* for illustrating the following definitions.

The definition for *return* just injects a single value into the ADT carrier.

$$\begin{aligned} &\mathit{return} :: \mathit{BinADT}\ a\ t \ \to\ a \ \to\ t \\ &\mathit{return}\ (\mathit{ADT}\ c\ \_)\ x \ =\ c\ (\mathit{Two}\ x\ (c\ \mathit{Unit}_B)) \end{aligned}$$

Note that the definition is parameterized by an ADT from which the constructor is taken. For lists we have $\mathit{cList}\ \mathit{Unit}_B = [\,]$, and $\mathit{cList}\ (\mathit{Two}\ x\ [\,]) = [x]$. Thus, $\mathit{return\ list}\ x \ =\ [x]$. The definition of *zero* is similar, it just applies the ADT's constructor to Unit_B.

$$\begin{aligned} &\mathit{zero} :: \mathit{BinADT}\ a\ t \ \to\ t \\ &\mathit{zero}\ (\mathit{ADT}\ c\ \_) \ =\ c\ \mathit{Unit}_B \end{aligned}$$

For lists, this means: $\mathit{zero\ list} \ =\ [\,]$. Monads are often described as encapsulating computations. But this is only one possible point of view; in what follows, it is instructive to think of a monad $m\ a$ as a collection of values of type a. A well-known example for this view is actually the list monad.

To give a definition for *bind* we have to explain how the elements of m can be extracted since we have to apply the function f to all of them. Since m is an ADT, all we can do is to apply its destructor, and we obtain (i) a single element (of type a) and (ii) the "remaining part" of m. By repeating the destruction we eventually extract all elements, say, $x_1, \ldots, x_n$. Then f is applied to each x_i, and we obtain a collection, say, $y_1, \ldots, y_n$, of $m\ b$-values which have to be combined into one $m\ b$-value. To accomplish this we can employ an operation *join* to combine two ADTs (that is, two $m\ b$-values) and then use *join* to fold the values $y_1, \ldots, y_n$.

Such an operation can be defined as an ADT transformer as follows: decompose the first argument and immediately rebuild it while replacing the eventually resulting Unit_B by the second argument. In the case of lists we have: combining two lists into one means their concatenation, and this can be defined by decomposing the first list into all of its elements which are then consed to the second list. First, we need an operation that redefines an ADT's constructor on Unit_B:

[2] This means comprehensions that map between different types.

$$
\begin{array}{l}
\mathit{withUnit} :: \mathit{BinADT}\ a\ t\ \to\ t\ \to\ \mathit{BinADT}\ a\ t \\
(\mathit{ADT}\ c\ d)\ \text{`}\mathit{withUnit}\text{`}\ u\ =\ \mathit{ADT}\ c'\ d\ \textbf{where}\ c'\ \mathit{Unit}_B\ =\ u \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad c'\ x\ \ =\ c\ x
\end{array}
$$

This operation can now be used to define the join of two ADT values. Since we do not (want to) have an intermediate data structure storing the values y_i, we integrate the application of f into this join operation that is actually to be applied to each decomposed x_i-value.

$$
\begin{array}{l}
\mathit{join} :: \mathit{BinADT}\ b\ t\ \to\ (a\ \to\ t)\ \to\ \mathit{Binary}\ a\ t\ \to\ t \\
\mathit{join}\ a\ f\ \mathit{Unit}_B\ \ =\ (\mathit{con}\ a)\ \mathit{Unit}_B \\
\mathit{join}\ a\ f\ (\mathit{Prod}_B\ x\ y)\ =\ \mathit{transit}\ a\ (a\ \text{`}\mathit{withUnit}\text{`}\ y)\ (f\ x)
\end{array}
$$

The function *join* works as follows: it takes an element x, applies f to it, and then applies an ADT transformer to $f\ x$. This transformer has source ADT a and target ADT a `*withUnit*` y which essentially means that $f\ x$ is decomposed and immediately rebuilt using y as a unit value. For lists this means to take an element of type a, apply f to obtain a list of type $[b]$, and to concatenate this list with the list y.

Next we try the to define *bind* by folding the ADT monad with *join*.

$$
\begin{array}{l}
\mathit{bind} :: \mathit{BinADT}\ a\ (m\ a)\ \to\ m\ a\ \to\ (a\ \to\ m\ b)\ \to\ m\ b \\
\mathit{bind}\ a\ m\ f\ =\ \mathit{fold}\ (\mathit{join}\ a\ f)\ a\ m \qquad\qquad -- \text{wrong!}
\end{array}
$$

Unfortunately, this definition does *not* work since the parameter ADT a is used at two different type instances: first, as a parameter of *fold* with carrier $m\ a$, and second, as a parameter of *join* with type $m\ b$, which is, in general, different from $m\ a$. We can remedy the situation by spending a further ADT parameter which will actually receive the same argument ADT as the first parameter, but which can be instantiated in the definition to a different type.

$$
\begin{array}{l}
\mathit{bind} :: (\mathit{BinADT}\ b\ t,\ \mathit{BinADT}\ a\ u)\ \to\ u\ \to\ (a\ \to\ t)\ \to\ t \\
\mathit{bind}\ (a_1, a_2)\ m\ f\ =\ \mathit{fold}\ (\mathit{join}\ a_1\ f)\ a_2\ m
\end{array}
$$

Now we can use comprehensions for arbitrary ADTs. The operations *result*, *bind*, *zero* are parameterized by ADTs; they are thus essentially operations to generate monad operations from ADTs. If an ADT a satisfies the monad laws (that is, the operations *result* a, *bind* (a, a), and *zero* a), the comprehensions work exactly like with the corresponding traditional monad operations. The possible problems that may arise if the laws do not hold are addressed in the next section.

5 Generalized Comprehensions

The fact that we were forced in the definition of *bind* to use two ADT parameters is not as unpleasant as it may seem at first; it actually shows an opportunity for generalization: it is intriguing to see what happens if we insert two different ADTs as a parameter. To better understand what is really going on it is helpful to understand the roles of the two ADTs. The first ADT (a_1) is used to decompose and join the y_i values, while the second ADT (a_2) is just used to decompose m. This scheme of computation is illustrated in Figure 1.

In particular, we can observe that a_1 and a_2 need not be identical.[3] Thus, we can use, for example, a_2 = *graph* to decompose a graph and a_1 = *list* to simply collect node values (recall that "::: *list*" is the default

[3]Figure 1 reveals a further way of generalization: since the destructor of a_1 does not depend on the constructor, the decomposition of the y_i values is in no way tied to a_1. Thus, we could use a further ADT, say a_3, to independently specify the destructor for the y_i values. However, there seem to be only few applications that do not justify the added complexity of such a generalization.

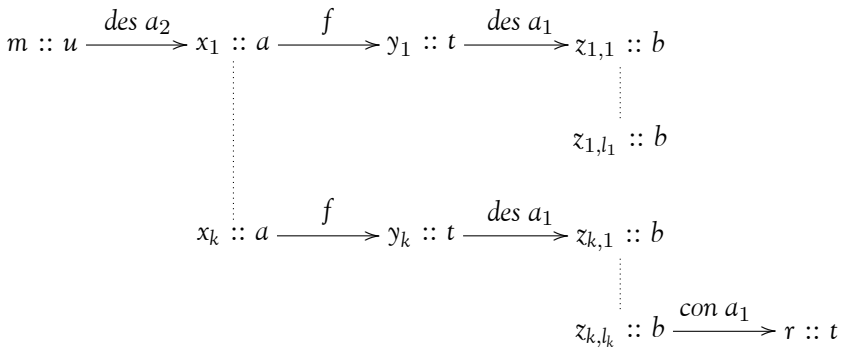

Figure 1: Computation of $r = bind\ (a_1, a_2)\ f\ m$.

outer ADT specification):

$$nodes\ g\ =\ [\ v\ |\ (\_, v, \_) \leftarrow g\ ::: graph\ ]$$

Although this example works well, other comprehensions behave in unexpected ways. The reason is that the monad laws do not hold in general for arbitrary ADTs, which means that comprehensions are not guaranteed to be well-defined, at least under the traditional translation scheme that was shown in Section 2. We illustrate the problems with two examples. First, consider the translation of the factorial comprehension:

$$foo\ n\ =\ bind\ (prod, prod)\ n\ (\backslash i \rightarrow return\ prod\ i)$$

One would expect that with this definition n is destructed using *prod*'s destructor yielding numbers that are multiplied by *prod*'s constructor. However, the definition of *bind* involves the combination of intermediate results (yielded by *return*) with *join*, and *join* is just defined to decompose and rebuild each number using *prod*. Thus, each number is decomposed and multiplied on its own, and so *foo* actually computes the product of the first n factorial numbers, which was not intended.

Another example is the comprehension defining graph reversal. Here the problem is even worse because the naively translated version will result, in general, in run time errors.

$$bar\ g\ =\ bind\ (graph, graph)\ g\ (\backslash(p, v, s) \rightarrow return\ graph\ (s, v, p))$$

This happens because the function *return graph* tries to construct intermediate graphs from isolated context values, and this fails whenever the successor or predecessor list is not empty since this context value is tried to be inserted into the empty graph.

A solution is suggested by the following observation: ADT values are built by *return* expressions only to be decomposed later. Moreover, the actual choice of ADT is not important in the sense that in almost all interesting cases the construction is simply undone (except for rather strange and unwanted situation like in *foo* above). In contrast, what really matters is the ADT into which the final values are inserted. Hence, we can try to use a very simple dummy ADT to store intermediate values of *return* expressions and translate the last generator and/or filter expression into calls to a specialized *bind* operation that is aware of this dummy ADT.

Below we use the ADT *singleton*, which offers nothing more than to insert and retrieve a value completely unchanged into/from an ADT containing just this element. It is also capable of dealing with *zero*

values (which will be needed for the translations of filter expressions).

$$
\begin{array}{lcl}
\mathit{singleton} & :: & \mathit{BinADT}\ a\ (\mathit{Maybe}\ a) \\
\mathit{singleton} & = & \mathit{ADT}\ \mathit{inj}\ \mathit{get}\ \textbf{where}\ \mathit{inj}\ \mathit{Unit}_B = \mathit{Nothing} \\
 & & \qquad\qquad\qquad\quad \mathit{inj}\ (\mathit{Two}\ (x\ \_)) = \mathit{Just}\ x \\
 & & \qquad\qquad\qquad\quad \mathit{get}\ \mathit{Nothing} = \mathit{Unit}_B \\
 & & \qquad\qquad\qquad\quad \mathit{get}\ (\mathit{Just}\ x) = \mathit{Two}\ x\ \mathit{Nothing}
\end{array}
$$

Now we need specialized definitions for *join* and *bind*.

$$
\begin{array}{lcl}
\mathit{join}_s\ a\ f\ \mathit{Unit}_B & = & (\mathit{con}\ a)\ \mathit{Unit}_B \\
\mathit{join}_s\ a\ f\ (\mathit{Two}\ x\ y) & = & \mathit{transit}\ \mathit{singleton}\ (a\ \text{`}\mathit{withUnit}\text{`}\ y)\ (f\ x)
\end{array}
$$

$$
\mathit{bind}_s\ (a_1, a_2)\ m\ f = \mathit{fold}\ (\mathit{join}_s\ a_1\ f)\ a_2\ m
$$

The differences between these and the more general definitions from above are as follows. First, join_s expects $f\ x$ to yield a value of ADT *singleton* and selects the stored element to put it into the target ADT a. Second, bind_s just calls join_s instead of *join*; it will be used as the innermost *bind* call in the translation of comprehensions.

Note that bind_s is essentially an ADT transformer. This fact is expressed in the following theorem. We need a function *mapFst* which is defined as follows.[4]

$$
\begin{array}{lcl}
\mathit{mapFst} & :: & (a \to b) \to \mathit{Binary}\ a\ t \to \mathit{Binary}\ b\ t \\
\mathit{mapFst}\ f\ \mathit{Unit}_B & = & \mathit{Unit}_B \\
\mathit{mapFst}\ f\ (\mathit{Two}\ x\ y) & = & \mathit{Two}\ (f\ x)\ y
\end{array}
$$

Then we have the following relationship.

Theorem 1 $\mathit{bind}_s\ (a_1, a_2)\ m\ (\mathit{Just}\ .\ f) = \mathit{trans}\ (\mathit{mapFst}\ f)\ a_2\ a_1\ m$

The proof is given the Appendix. With $f = id$ we obtain as a direct corollary the following.

Corollary 1 $\mathit{bind}_s\ (a_1, a_2)\ m\ \mathit{Just} = \mathit{transit}\ a_2\ a_1\ m$

The translations for the above two examples will yield the following definitions.

$$
\begin{array}{lcl}
\mathit{fac}\ n & = & \mathit{bind}_s\ (\mathit{prod}, \mathit{prod})\ n\ (\backslash i \to \mathit{Just}\ i) \\
\mathit{grev}\ g & = & \mathit{bind}_s\ (\mathit{graph}, \mathit{graph})\ g\ (\backslash (p, v, s) \to \mathit{Just}\ (s, v, p))
\end{array}
$$

These work as expected.

Next we describe how to translate ADT comprehensions. First, we define a straightforward extension of the traditional comprehension syntax. Essentially, we allow to add ADT "tags" to generators and to the result. By making these tags optional and using appropriate defaults we achieve that the new notation is a conservative extension of well-known list comprehension syntax. The syntax is shown in Figure 2.
We define the translation in two phases: in a first step, omitted tags are filled with default values, and in a second step, a fully annotated comprehension expression is translated into monad operations. Below we use the notation $a\{b\}$ (read: "a or else b") to denote b if $a = \epsilon$ and a otherwise. Likewise, $q\{b\}$ denotes $p \leftarrow e\ b$ if $q = p \leftarrow e\ \epsilon$ and q otherwise. With this notational help we can define tag completion as follows.

[4] More generally, *Binary* is a bifunctor with a map function for mapping along both argument types; *mapFst* is an instance of this map function mapping only along the first argument type. In a sense, *mapFst* is the analogue to *fmap*, see [6].

$$
\begin{array}{lcl}
\mathit{compr} & \rightarrow & [\ \mathit{expr} \mid \mathit{quals}\ ]\ \mathit{tag} \\
\mathit{quals} & \rightarrow & \mathit{qual},\ \mathit{quals} \mid \epsilon \\
\mathit{qual} & \rightarrow & \mathit{pat} \leftarrow \mathit{expr}\ \mathit{tag} \mid \mathit{expr} \\
\mathit{tag} & \rightarrow & ::: \mathit{expr} \mid \epsilon
\end{array}
$$

Figure 2: Comprehension Syntax.

$$
\begin{array}{rcl}
\mathcal{C}([\ e \mid q_1, \ldots, q_n\ ]\ a) & = & [\ e \mid q_1\{b\}, \ldots, q_n\{b\}\ ]\ b \\
 & & \textbf{where}\ \ b\ =\ a\{::: \mathit{list}\}
\end{array}
$$

This means, if there is no result tag given, *list* is used, and for each generator that does not have a tag, the outermost tag used, which, if not present is taken to be *list* by default.

After tag completion, the comprehension translation shown in Figure 3 can assume that all tags are present (note that $\mathit{zero}_s\ = \mathit{zero}\ \mathit{singleton}$).

$$
\begin{array}{lcl}
\mathcal{T}([\ e \mid p \leftarrow e'\ a'\ ]\ a) & = & \mathit{bind}_s\ (a, a')\ \mathcal{T}(e'\ a')\ (\backslash p \rightarrow \mathit{Just}\ e) \\
\mathcal{T}([\ e \mid p \leftarrow e'\ a',\ e''\ ]\ a) & = & \mathit{bind}_s\ (a, a')\ \mathcal{T}(e'\ a')\ (\backslash p \rightarrow \textbf{if}\ e''\ \textbf{then}\ \mathit{Just}\ e\ \textbf{else}\ \mathit{zero}_s) \\
\mathcal{T}([\ e \mid p \leftarrow e'\ a',\ qs\ ]\ a) & = & \mathit{bind}\ (a, a')\ \mathcal{T}(e'\ a')\ (\backslash p \rightarrow \mathcal{T}([\ e \mid qs\ ]\ a)) \\
\mathcal{T}([\ e \mid e'\ ]\ a) & = & \textbf{if}\ e'\ \textbf{then}\ \mathit{Just}\ e\ \textbf{else}\ \mathit{zero}_s \\
\mathcal{T}([\ e \mid e',\ qs\ ]\ a) & = & \textbf{if}\ e'\ \textbf{then}\ \mathcal{T}([\ e \mid qs\ ]\ a)\ \textbf{else}\ \mathit{zero}\ a \\
\mathcal{T}(e\ a) & = & e
\end{array}
$$

Figure 3: Translation of Comprehensions.

As an example consider the translation of the comprehension for *divisors*. Had we omitted the outer *list* ADT, tag completion would have added "::: *list*" to the result. In contrast, the generator is not affected since it already contains the tag "::: *nat*". After that, the second and the last equation of $\mathcal{T}$ are applied.

$$
\begin{array}{l}
\mathcal{T}([\ i \mid i \leftarrow n ::: \mathit{nat},\ n\ `\mathit{mod}`\ i == 0\ ] ::: \mathit{list}) = \\
\mathit{bind}_s\ (\mathit{list}, \mathit{nat})\ \mathcal{T}(n ::: \mathit{nat})\ (\backslash i \rightarrow \textbf{if}\ n\ `\mathit{mod}`\ i == 0\ \textbf{then}\ \mathit{Just}\ i\ \textbf{else}\ \mathit{zero}_s) = \\
\mathit{bind}_s\ (\mathit{list}, \mathit{nat})\ n\ (\backslash i \rightarrow \textbf{if}\ n\ `\mathit{mod}`\ i == 0\ \textbf{then}\ \mathit{Just}\ i\ \textbf{else}\ \mathit{zero}_s)
\end{array}
$$

Another illustration of the translation is contained in the proof of the following theorem that relates nested comprehensions to ADT filters.

Theorem 2 $[\ x \mid x \leftarrow [\ y \mid y \leftarrow e ::: a\ ] ::: b\ ] ::: c\ =\ \mathit{via}\ a\ b\ c\ e$

The proof is given in the Appendix. Thus, we can write, for example, *heapsort* as a comprehension:

$$\mathit{heapsort}\ l\ =\ [\ x \mid x \leftarrow [\ y \leftarrow l ::: \mathit{list}\ ] ::: \mathit{heap}\ ]$$

We finally have to ask whether the described translation scheme is correct or at least "reasonable" in some sense. In the case of monad comprehensions, there are intuitive laws that hold for the comprehension syntax and that can be proved by using elementary monad laws. For example, the following equation follows from the right unit (RU) monad law.

$$[\, x \mid x \leftarrow e \,] \;=\; e \qquad (*)$$

In fact, the same law holds for comprehensions on ADTs that satisfy (RU). However, the law does *not* hold in general for ADT comprehensions. This should be not too surprising since ADT constructors and destructors are, in general, not inverses of each other. But this is exactly what equation $(*)$ demands: taking the elements of e by way of its ADT destructor (say, a) apart and reassembling them with a's constructor should yield e. The fact that such a (strong) relationship does not hold, contributes to the additional power of ADT comprehensions, for example, the factorial comprehensions $[i \mid i \leftarrow n\,] ::: \mathit{prod}$ essentially exploits the fact that the constructor is *not* the inverse of the destructor.

Are there other invariants—weaker than the monad laws—that characterize a reasonable comprehension behavior? One result is that the comprehension syntax reflects the operations of the underlying ADT domain, that is, ADT transformers. A relationship that corresponds to $(*)$ is expressed in the following theorem.

Theorem 3 $[\, x \mid x \leftarrow e ::: a \,] ::: b \;=\; \mathit{transit}\ a\ b\ e$

Proof. The one-step translation of the comprehension yields $\mathit{bind}_s\ (b, a)\ e\ \mathit{Just}$. By Corollary 1 this is equal $\mathit{transit}\ a\ b\ e$. □

In particular, this result justifies the use of bind_s in the translation of comprehensions.

6 Conclusions

We have shown how comprehension syntax can work for ADTs that are defined using a standardized interface for constructors and destructors. This constructor/destructor-based ADT approach enables us, first, to automatically supply definitions for monadic operators and, second, derive the traditional comprehension syntax. Moreover, through a modest generalization of the comprehension syntax and an adaptation of the translation we have obtained a versatile language feature that can be used to express many algorithms in a clear and concise way.

Since ADTs are not restricted to "linearly recursive" data types, that is, ADTs with pattern functor *Binary*, we could, principally, define comprehensions also for differently shaped data types, such as trees. This is a topic of future research.

References

[1] R. Backhouse, P. Jansson, J. Jeuring, and L. Meertens. Generic Programming – An Introduction. In *Advanced Functional Programing*, LNCS 1608, pages 28–115, 1999.

[2] R. S. Bird and O. de Moor. *The Algebra of Programming*. Prentice-Hall International, 1997.

[3] R. M. Burstall. Design Considerations for a Functional Programming Language. In *Infotech State of the Art Conference*, Copenhagen, DK, 1977. Infotech.

[4] M. Erwig. Functional Programming with Graphs. In *2nd ACM Int. Conf. on Functional Programming*, pages 52–65, 1997.

[5] M. Erwig. Categorical Programming with Abstract Data Types. In *7th Int. Conf. on Algebraic Methodology and Software Technology*, LNCS 1548, pages 406–421, 1998.

[6] M. Erwig. Metamorphic Programming: Structured Recursion for Abstract Data Types. Technical Report 242, FernUniversität Hagen, 1998.

[7] M. Erwig. Random Access to Abstract Data Types. In *8th Int. Conf. on Algebraic Methodology and Software Technology*, LNCS 1816, pages 135–149, 2000.

[8] M. Erwig. Inductive Graphs and Functional Graph Algorithms. *Journal of Functional Programming*, 11(5):467–492, 2001.

[9] L. Fegaras and D. Maier. Towards an Effective Calculus for Object Query Languages. In *ACM SIGMOD Conf. on Management of Data*, pages 47–58, 1995.

[10] M. M. Fokkinga. Datatype Laws without Signatures. *Mathematical Structures in Computer Science*, 6:1–32, 1996.

[11] L. Libkin, R. Machlin, and L. Wong. A Query Language for Multidimensional Arrays: Design, Implementation, and Optimization Techniques. In *ACM SIGMOD Conf. on Management of Data*, pages 228–239, 1996.

[12] S. Mac Lane. *Categories for the Working Mathematician*. Springer-Verlag, 1971.

[13] E. Moggi. Computational Lambda-Calculus and Monads. In *IEEE Symp. on Logic in Computer Science*, pages 14–23, 1989.

[14] E. Moggi. Notions of Computation and Monads. *Information and Computation*, 93(1), 1991.

[15] A. Poulovassilis and C. Small. Algebraic Query Optimisation for Database Programming Languages. *The VLDB Journal*, 5(2):119–132, 1996.

[16] P. Trinder. Comprehensions: A Query Notation for DBPLs. In *3rd Int. Workshop on Database Programming Languages*, pages 55–68, 1991.

[17] D. A. Turner. Recursion Equations as a Programming Language. In J. Darlington, P. Henderson, and D. A. Turner, editors, *Functional Programming and its Applications*, pages 1–28. Cambridge University Press, 1982.

[18] P. Wadler. List Comprehensions. In S. L. Peyton Jones, editor, *The Implementation of Functional Programming Languages*, pages 127–138. Prentice-Hall International, Englewood Cliffs, NJ, 1987.

[19] P. Wadler. Comprehending Monads. *Mathematical Structures in Computer Science*, 2:461–493, 1992.

[20] P. Wadler. Monads for Functional Programming. In *Advanced Functional Programing*, LNCS 925, pages 24–52, 1995.

[21] P. Wadler. How to Declare an Imperative. *ACM Computing Surveys*, 29(3):240–263, 1997.

Appendix

Proof of Theorem 1. We first apply the definition of $bind_s$ to the left hand side and the equation (TransFold) to the right hand side. This yields:

$$fold\ (join_s\ a_1\ (Just\ .\ f))\ a_2\ m\ =\ fold\ (con\ a_1\ .\ mapFst\ f)\ a_2\ m$$

Thus, it remains to be shown: $join_s\ a_1\ (Just\ .\ f)\ =\ con\ a_1\ .\ mapFst\ f$. We consider the two possible cases of *Binary*. The case for $Unit_B$ is simple:

$$join_s\ a_1\ (Just\ .\ f)\ Unit_B\ =\ (con\ a_1)\ Unit_B\ =\ (con\ a_1\ .\ mapFst\ f)\ Unit_B$$

In the case for $Two\ x\ y$ we use a_1^y to denote a_1 '*withUnit*' y and we abbreviate *transit singleton* by $transit_s$:

$$\begin{aligned}
&join_s\ a_1\ (Just\ .\ f)\ (Two\ x\ y)\\
&= transit_s\ a_1^y\ (Just\ (f\ x))\\
&= (con\ a_1^y\ .\ fmap\ (transit_s\ a_1^y)\ .\ get)\ (Just\ (f\ x))\\
&= (con\ a_1^y\ .\ fmap\ (transit_s\ a_1^y))\ (Two\ (f\ x)\ Nothing)\\
&= (con\ a_1^y)\ (Two\ (f\ x)\ ((transit_s\ a_1^y)\ Nothing))\\
&= (con\ a_1^y)\ (Two\ (f\ x)\ ((con\ a_1^y\ .\ fmap\ (transit_s\ a_1^y)\ .\ get)\ Nothing))\\
&= (con\ a_1^y)\ (Two\ (f\ x)\ ((con\ a_1^y\ .\ fmap\ (transit_s\ a_1^y))\ Unit_B))\\
&= (con\ a_1^y)\ (Two\ (f\ x)\ ((con\ a_1^y)\ Unit_B))\\
&= (con\ a_1^y)\ (Two\ (f\ x)\ y)\\
&= (con\ a_1)\ (Two\ (f\ x)\ y)\\
&= (con\ a_1\ .\ mapFst\ f)\ (Two\ x\ y)
\end{aligned}$$

□

Proof of Theorem 2.

$$\begin{aligned}
&\mathcal{T}([\ x\ |\ x \leftarrow [\ y \leftarrow e\ :::\ a\ ]\ :::\ b\ ]\ :::\ c) =\\
&bind_s\ (c, b)\ \mathcal{T}([\ y \leftarrow e\ :::\ a\ ]\ :::\ b)\ (\backslash x \rightarrow Just\ x) =\\
&bind_s\ (c, b)\ (bind_s\ (b, a)\ \mathcal{T}(e\ :::\ a)\ (\backslash y \rightarrow Just\ y))\ Just =\\
&bind_s\ (c, b)\ (bind_s\ (b, a)\ e\ Just)\ Just
\end{aligned}$$

Applying Corollary 1 twice and folding the definition of *via* yields then:

$$transit\ b\ c\ (transit\ a\ b\ e) = (transit\ b\ c\ .\ transit\ a\ b)\ e = via\ a\ b\ c\ e$$

□