Bulk types with class

Simon Peyton Jones
University of Glasgow
Email: `simonpj@dcs.gla.ac.uk`.
WWW: `http::/www.dcs.gla.ac.uk/~simonpj`

May 1, 1997

Abstract

Bulk types — such as lists, bags, sets, finite maps, and priority queues — are ubiquitous in programming. Yet many languages don't support them well, even though they have received a great deal of attention, especially from the database community. Haskell is currently among the culprits.

This paper has two aims: to identify some of the technical difficulties, and to attempt to address them using Haskell's constructor classes.

This paper appears in the proceedings of the 1997 Haskell Workshop, Amsterdam, 7 June 1997. A slightly earlier version appears in the (electronic) proceedings of the 1996 Glasgow Functional Programming Workshop:

`http://www.dcs.gla.ac.uk/fp/workshops/fpw96/Proceedings96.html`

1 Introduction

Functional programs use a lot of lists, but often a list is actually used to represent:

> a stack, a queue, a deque, a bag, a set, a finite map (by way of an association list of (key,value) pairs), or a priority queue.

Using lists for all of these so-called *bulk types* is bad programming style for two reasons:

1. The type of the object does not specify its invariant (e.g. in a set there are no duplicates) and its expected operations (e.g. lookup in a finite map). The lack of these invariants makes the program harder to understand, harder to prove properties about, and harder to maintain.
2. Operations on lists may be less efficient, or perhaps even in a different complexity class, than operations on a suitably optimised abstract data type. For example, list append

(++) takes time linear in the size of its first argument, whereas it is easy to implement an ordered sequence ADT with constant-time concatenation[1].

Everyone knows this, but everyone still uses lists! Why? Because lists are well supported by the language: they admit pattern matching, there is built-in syntax (list comprehensions), and there is a rich library of functions that operate over lists. Even experienced functional programmers knowingly write an $O(n^2)$ algorithm where an $O(n)$ algorithm would do, because it is just so convenient to use lists and append them rather than to design and implement and use an abstract data type.

Why, then, aren't there well-engineered libraries to support sets, bags, finite maps, and so on? Many decent attempts have been made, notably C++'s standard template library (STL) – see Section 5 – but all have technical difficulties. This paper identifies some of these difficulties and attacks them using Haskell's type classes.

2 The problem with bulk types

The central difficulty with bulk types is their degree of polymorphism. First, there are many different sorts of collections — lists, sets, queues, and so on. Second, one such sort may have many different possible representations — lists, trees, hash tables, and so on. Lastly, each such representation may have many different element types — integers, booleans, characters, pairs, and so on.

A language that supports *polymorphism* allows the programmer to write a single algorithm that can be used in many "essentially similar" situations. For example, suppose we want to construct the list (or set, or bag) of leaves of a tree, where the tree is defined by the following data type:

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

Here is a possible algorithm that works for a tree with `Int` leaves, constructing a set of `Int`s:

```
leavesSetInt :: Tree Int    -> SetInt
leavesSetInt (Leaf a)       = singletonSetInt a
leavesSetInt (Branch t1 t2) = leaves t1 `unionSetInt` leaves t2
```

This code assumes the existence of the following set construction functions:

```
singletonSetInt :: Int -> SetInt
unionSetInt     :: SetInt -> SetInt -> SetInt
```

There are two ways in which this program can be made more polymorphic:

Element polymorphism Firstly, it is obvious that code of precisely the same form would be required for a tree of booleans. We would like to be able to generalise `leaves` like this:

[1] At least, it is easy if one is prepared to give up O(1) head and tail functions. It is possible, albeit somewhat more complex, to support append, head and tail all in constant (amortized) time (Okasaki [1995]).

```
leavesSet :: Tree a      -> Set a
leavesSet (Leaf a)       = singletonSet a
leavesSet (Branch t1 t2) = leaves t1 `unionSet` leaves t2
```

To make this work we would need to have these set operations:

```
singletonSet :: a -> Set a
unionSet     :: Set a -> Set a -> Set a
```

Bulk-type polymorphism Suppose that we have a second data type, `OrdSet`, that uses a different representation from that of `Set` — perhaps `Set` represents the set as a list with no duplicates, while `OrdSet` uses a balanced tree, for example. The function to gather the leaves of a tree into an `OrdSet` will be of just the same form as that for `Set`. The same is true if we want to collect the leaves into a bag, or a priority queue, or a list. Ideally, then, we would like to make `leaves` more polymorphic still, something like this:

```
leaves :: Tree a      -> c a  -- where c is a bulk type
leaves (Leaf a)       = singleton a
leaves (Branch t1 t2) = leaves t1 `union` leaves t2
```

where the bulk-type constructors are now something like:

```
singleton :: a -> c a          -- where c is a bulk type
union     :: c a -> c a -> c a -- where c is a bulk type
```

The trouble is that neither of these two generalisations is straightforward. We discuss each in turn.

2.1 Element polymorphism

Consider the goal of making `leaves` polymorphic in the elements of the `Set`. The tidiest kind of polymorphism, *parametric polymorphism*, works when the very same source code will work regardless of the argument type. It is supported by many modern programming languages, including C++, ML, and Haskell[2]. If we were collecting the leaves of a tree into a list, then we could use parametric polymorphism very easily:

```
leavesList :: Tree a -> [a]
leavesList (Leaf x) = singletonList x
leavesList (Branch t1 t2) = leavesList t1 `unionList` leavesList t2
```

Here, `unionList` has type `[a] -> [a] -> [a]`; it is just list append, commonly written `++`.

The trouble arises with sets, because *we cannot make a* `union` *operation that works on sets whose elements of arbitrary type.* To remove duplicates we must at least have equality on the set elements! Furthermore, equality may not be enough:

[2] In the case of ML and Haskell, this polymorphism extends to the executable code too; that is, the same executable code works regardless of the argument type. In the case of C++, using templates one can have a single source-code function, but the compiler must instantiate it separately for each type at which it is used.

- If the element type admits only equality, then determining whether an element is a member of the set must take linear time.
- If the element type supports a total order then a tree (balanced or otherwise) may be more appropriate, and set membership can be determined in logarithmic time.
- If the element type admits a hash function, then the set might be represented by a hash table, or — in a purely-functional language where *persistent data structures*[3] are the rule — by a tree indexed on the hash key.
- If the element type has a one-to-one function mapping elements to integers, then radix-based tree representations become possible.

One way out of this dilemma, taken by Java for example, is to decide that every data type supports equality, together with ordering and/or a hash function. This is simple but crude — what about equality of functions, for example? A cleaner solution, adopted by ML for equality, and generalised in Haskell by type classes, is to use a type system that allows type variables to be qualified by the operations they support. Thus, in Haskell we can give the following type for union on a set data type that required only equality:

```
unionSet :: (Eq a) => Set a -> Set a -> Set a
```

This type specifies that the element type, `a`, must lie in the class `Eq`[4]. The class `Eq` is defined like this:

```
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
```

The declaration says that types that are instances of `Eq` must provide operations `(==)` and `(/=)` with the given types. For each data type that we want to be in `Eq` we must give an instance declaration that defines `(==)` and `(/=)` at that type. For example:

```
instance Eq Int where
  x == y = x `eqInt` y
  x /= y = not (x `eqInt` y)

instance (Eq a, Eq b) => Eq (a,b) where
  (a1,b1) == (a2,b2) = (a1==a2) && (b1==b2)
```

Given this type for `unionSet`, the type of `leavesSet` is now inferred to be:

```
leavesSet :: (Eq a) => Tree a -> EqSet a
```

[3]A data structure is "persistent" if, following an update, the old version of the data structure is still available (Okasaki [1996]).

[4]Strictly speaking, the semantics of `union` does not require the elimination of duplicates — that could be postponed until the set is observed by a membership test or by enumerating its elements. However, nothing fundamental is changed by such an implementation decision so in this paper we will stick with the naive view that `union` requires equality.

If our `Set` type required ordering as well as equality, we would simply replace `(Eq a)` by `(Ord a)` in the above types.

2.1.1 Other approaches

ML has equality types built in, but not ordered types, so the Haskell solution is not available in ML. (Restricting to equality only would be unreasonable, because sets based only on equality are hopelessly inefficient.) The solution adopted by some ML libraries is to make `Set` into a functor:

```
functor Set( ORD:ORD_SIG ) : SET
```

That is, `Set` is a functor taking an ordering as its argument, and producing a set structure (i.e. module) as its result. One can thereby construct efficient set-manipulation functions for particular element types:

```
IntSet  = Set IntOrd
CharSet = Set CharOrd
```

but now the `leaves` function has to mention either `IntSet.union` or `CharSet.union` — `leaves` cannot be polymorphic in the element type. To solve this, `leaves` must be defined in a functor that takes the `Set` structure as argument, and so on.

2.2 Bulk-type polymorphism

Next, we consider how to generalise `leaves` to work over arbitrary bulk types. To begin with we will consider only types — such as lists, queues, and stacks — that are truly parametric in their element types.

2.2.1 Using type classes

We start off with one union operation for each collection type, each of which has quite different code to the others:

```
unionList :: [a] -> [a] -> [a]
unionQueue :: Queue a -> Queue a -> Queue a
...etc...
```

In order to generalise `leaves`, we earlier informally suggested the type:

```
leaves :: Tree a -> c a  -- where c is a bulk type
```

We are suggesting here that `leaves` is polymorphic in `c`, the bulk type constructor. The polymorphism is not parametric, however, because each `union` operation uses different code; `leaves` should call a different union operation for each type. This is exactly what type classes are for! Perhaps we could write:

```
leaves :: (Bulk c) => Tree a -> c a
```

where `Bulk` is the class of bulk types, defined thus:

```
class Bulk c where
  empty     :: c a
  singleton :: a -> c a
  union     :: c a -> c a -> c a
```

Now we can give an instance declaration for each Bulk type:

```
instance Bulk [] where          -- [] is the List type constructor
  empty       = []
  singleton x = [x]
  union       = (++)

instance Bulk Queue where
  empty     = emptyQueue
  singleton = singletonQueue
  union     = unionQueue
```

All of this is legal Haskell, but notice that `c` is a variable that ranges over *type constructors* rather than *types*. This sort of higher-kind quantification is a fairly straightforward but powerful extension of the Hindley-Milner type system (Jones [1995]). It can be used in ordinary data type declarations but, as we shall see, it is particularly useful in Haskell's system of classes, which are thereby generalised from type classes to constructor classes.

Alas, things go wrong when we try to deal with non-parametric element types. We cannot give an instance declaration:

```
instance Bulk Set where
  empty     = emptySet
  singleton = singletonSet
  union     = unionSet
```

because `unionSet` has the wrong type! It requires that the element type be in `Eq`, whereas the overloaded `union` operator does not.

2.2.2 Other approaches

A possible alternative approach is to use *ad hoc* polymorphism. The symbol `union` would stand for a whole family of `union` operations, each with a different type. The choice of which to use would be made statically by the compiler, based on local type information. ML uses this sort of overloading for numeric operators, and so does C++, Ada, and other languages. The small disadvantage of *ad hoc* overloading is that one may need to write type signatures to specify which type to use; "small" because writing type signatures is a Good Idea anyway.

The big disadvantage is that one cannot write generic operations over collections. For example, we could write `leaves` thus:

```
leaves (Leaf a)        = singleton a
leaves (Branch t1 t2) = leaves t1 `union` leaves t2
```

but the compiler would have to resolve the `union` to `unionList`, or `unionQueue` or `unionSet`, or whatever. This resolution might be done implicitly, or by requiring the programmer to add a type signature; but however it is done `leaves` will only work on collections of one type. An exact copy of the code, with a different type signature, would deal with one more type, and so on. Every time you add a new collection type you would need to add a new copy of `leaves`. (Or perhaps the compiler could automatically make them all for you, in which case the issue is one of code size.)

2.3 Adaptive representations

There is a third issue which adds yet more spice to the challenge of implementing bulk types: that of choosing an appropriate representation. The appropriate representation of a collection depends on:

1. The size of the collection.
2. The relative frequency of the operations supported by the bulk type.
3. The operations that are available on the underlying element type.

Of course, we can simply dump the problem in the programmer's lap, by providing a large variety of different set data types, and leaving the choice to the programmer. (This is precisely what STL does.) A more attractive alternative is to make the bulk type choose its own representation.

Items (1) and (2) have been fairly well studied. Clever algorithms have been developed that adapt the representation of a data type based on its size and usage (Brodal & Okasaki [1996]; Chuang & Hwang [1996]; Okasaki [1996]). It is less obvious how to tackle item (3). How can we build an implementation of `Set` that chose its representation based on what operations are available on the elements? We return to this question in Section 3.3.

2.4 Summary

In this section we have reviewed various approaches to manipulating bulk types in polymorphic fashion. The bottom line is that "nothing quite works". Bulk types seem quite innocent, but the combination of polymorphism in both element and bulk types, and the non-parametric nature of both, conspire to defeat even the most sophisticated type systems.

3 First design: the `XOps` route

In this section we turn to our first solution, based on the most promising of the approaches reviewed, namely constructor classes. The solution we present has the merit of being implementable in standard Haskell (1.3), but it has some shortcomings that we will address in our second solution (Section 4).

Like C++'s STL, we identify two main groups of bulk types:

1. *Sequences*, where the order of insertion is significant (e.g. one can extract the most recently inserted element), but where no operations need be performed on the elements themselves.

2. *Collections*, where the order of insertion is unimportant, but where the elements must admit at least equality and preferably some other operations[5].

3.1 Sequences

A *sequence* contains a linear sequence of zero or more elements. The order of insertion and removal of elements is significant, and elements can be added or removed at either end. Examples of sequences are: *lists, catenable lists*[6]*, stacks, queues, deques.* They all support the same set of operations, but they differ in the complexity bounds for these operations.

Figures 1 and 2 defines a module `Sequence` whose main declaration is a type class, also called `Sequence`, that defines the set of operations on sequences. The names of the operations are chosen to be compatible with Haskell's current nomenclature for lists. `front` and `back` return both the first (respectively, last) element of the sequence, together with the remaining sequence; they return `Null` if the sequence is empty. The `SeqView` type is used as the return type for both of these functions: you can think of `front` and `back` as providing a head-and-tail-like "view" of each end of the sequence.

The `fold` functions, along with `length`, `filter`, `partition`, `reverse`, are straightforward generalisations of their list counterparts. They can all readily be defined in terms of either `front` or `back`. Indeed, each of them has a default method in the class declaration, indicating that an instance of `Sequence` may (but is not compelled to) provide a method for these operations. The reason for this decision is that for at least some instances of `Sequence` (snoc-lists, say) the default definition of some functions (`foldr`, in this case) is likely to be outrageously inefficient. Making these functions into class methods gives the implementor the option (though not the obligation) of providing more efficient definitions.

The standard classes `MonadPlus` and `Functor` are superclasses of `Sequence`; that is, any type in `Sequence` must also be in `MonadPlus` and `Functor`. Both of the latter are defined by the Haskell 1.3 prelude. Figure 3 gives their definitions, except that we have added `cons` and `snoc` to the class `MonadPlus`. They can both be implemented in terms of `++`, as their default methods show, but for many types they can be more efficiently implemented directly.

All the operations of the standard classes `Monad`, `MonadZero`, `MonadPlus`, and `Functor` make sense for sequences: `++` appends two sequences; `map` applies a function to each element of a

[5] STL refers to these as "associative containers".

[6] Catenable lists support constant-time append.

```
module Sequence where

data SeqView s a = Null | Cons a (s a)

empty :: Sequence s => s a
empty = zero

singleton :: Sequence s => a -> s a
singleton x = return x

fromList :: Sequence s => [a] -> s a
fromList xs = foldr ((++) . return) zero xs

toList   :: Sequence s => s a -> [a]
toList s = foldr (:) [] s

class (Functor s, MonadPlus s) => Sequence s where
  null  :: s a -> Bool
  front :: s a -> SeqView s a
  back  :: s a -> SeqView s a

  (!!) :: s a -> Int -> a
  update :: s a -> Int -> a -> s a

  foldr  :: (a -> b -> b) -> b -> s a -> b
  foldr1 :: (a -> a -> a) -> s a -> a
  foldl  :: (b -> a -> b) -> b -> s a -> b
  foldl1 :: (a -> a -> a) -> s a -> a

  length :: s a -> Int

  elem :: (Eq a) => a -> s a -> Bool

  filter :: (a -> Bool) -> s a -> s a
  partition :: (a -> Bool) -> s a -> (s a, s a)

  reverse :: s a -> s a
```

Figure 1: The sequence class

```
        -- Default methods
  foldr k z xs = case front xs of
                   Null -> z
                   Cons x xs -> x `k` foldr k z xs
  foldr1 k xs = case back xs of
                  Cons x xs -> foldr k x xs
  foldl k z xs = case front xs of
                   Null -> z
                   Cons x xs -> foldl k (z `k` x) xs
  foldl1 k xs = case front xs of
                  Cons x xs -> foldl k x xs

  length xs = foldr (\_ n -> n+1) 0 xs
  filter p xs = foldr f zero xs
              where f x ys | p x = x `cons` ys
                           | otherwise = ys
  partition p xs = (filter p xs, filter (not.p) xs)
  reverse xs = foldl (flip.cons) zero xs
  elem x xs = foldr ((||) . (==) x) False xs
```

Figure 2: The sequence class, continued

```
class Monad m where
  (>>=) :: m a -> (a -> m b) -> m b
  return :: a -> m a

class (Monad m) => MonadZero m where
  zero :: m a

class (MonadZero m) => MonadPlus m where
  (++) :: m a -> m a -> m a

  cons :: a -> s a -> s a
  snoc :: s a -> a -> s a

  cons x xs = (return x) ++ xs          -- Not yet in 1.3
  snoc xs x = xs ++ (return x)          -- Not yet in 1.3

class Functor m where
  map :: (a->b) -> m a -> m b
```

Figure 3: Monad and functor classes

sequence; `zero` is the empty sequence; `return` forms a singleton sequence; and `>>=` takes a function that maps each element of a sequence to a new sequence, and concatenates the results.

Because sequences lie in the class `Monad` we can use the `do` notation to describe sequence-valued expressions. For example:

```
do { x<-xs; y<-ys; return (x,y) }
```

will deliver the sequence composed of all `(x,y)` pairs, where `x` is drawn from the sequence `xs` and `y` is drawn from `ys`. The same applies to Haskell's comprehension notation, which also defines an expression over any monad. For example, this comprehension defines the same sequence as that above.

```
[(x,y) | x<-xs, y<-ys]
```

A great deal of work has been done on the connection between bulk types and comprehension syntax, especially in the context of database queries and their optimisation (Buneman et al. [1994]; Trinder [1991]; Trinder & Wadler [1989]; Wadler [1992]).

Lastly, the `Sequence` module contains a couple of ordinary declarations that give more collection-oriented names to the monadic functions `zero` and `return`, namely `empty` and `singleton` respectively.

3.2 Collections

We observed earlier that the problem with collections is that the `union` operation may impose different constraints on the element type, depending on which collection we are dealing with. Our solution is very simple, namely to give them all the same type. First we define a new class `XOps`, the class of element operations, thus:

```
class XOps a where
  xEq    :: a -> a -> Bool
  xCmp   :: Maybe (a -> a -> Ordering)  -- Three-way comparison
  xHash  :: Maybe (a -> Int)            -- Hash function; could be many-one
  xToInt :: Maybe (a -> Int)            -- Injection; guaranteed one-one
```

(`Ordering` is a standard Haskell data type with three constructors, `LT`, `EQ` and `GT`.) The point about `XOps` is that it tells not only how to (say) compare two elements, but *also whether such a comparison is available*. The equality operation, however, is mandatory, so it is not wrapped in a `Maybe` type[7]. For example, for a particular type `T`, we might have an instance declaration:

```
instance XOps T where
  xEq    = (==)
  xCmp   = Just cmpT
  xHash  = Nothing
```

[7] We could make `Eq` a superclass of `XOps` instead of having `xEq`, but it is sometimes convenient to define a non-standard equality for collection operations — see Section 3.6 — and it is confusing to have non-standard instances of `Eq`.

```
  xToInt = Nothing
```

to say that `T` had a comparison operation, `cmpT`, but no `xHash` or `xToInt` operation.

Next, we define the class `Collection`, of collections, like this:

```
class Collection c where
  empty :: c a
  insert :: XOps a => a -> c a -> c a
  ...and much more...
```

We will add many further operations shortly. We use `empty` for both collections and sequences, relying on the use of qualified names (such as `C.empty`) to distinguish them when necessary.

With these definitions, it is now possible to give a fully-respectable type to `leaves`:

```
leaves :: (Collection c, XOps a) => Tree a -> c a
```

Notice that, unlike the `Sequence` class, *we cannot make* `MonadPlus` *and* `Functor` *into superclasses of* `Collection`. Why not? Because for collections we cannot give sufficiently polymorphic definitions for `++` and `>>=`. To perform these operations we will need the constraint `XOps t` on the element type `t` — but that would not fit the signature of the classes `MonadPlus` and `Functor`.

3.3 Instances of `Collection`

Next, suppose we have a datatype, `OrdSet` that implements sets using trees, making use of an ordering operation on the set's elements. We can make `OrdSet`, the type of ordered sets (whose implementation depends on an element ordering), an instance of `Collection` thus:

```
instance Collection OrdSet where
  empty = Empty
  insert x t = case xCmp of
                 Just cmp -> insertTree cmp x t
                 Nothing  -> error "OrdSet.insert"
```

(Here we are assuming the existence of a suitable data type of `Tree`s, with operations `insertTree` to insert an element.) An obvious sadness is that if we try to build an `OrdSet` of things that only admit equality then we will only get a runtime error, not a compile-time type error. Whilst this is undoubtedly sad, we will see shortly how to design set datatypes that cannot fail in this way. Furthermore, it is worth remembering that most programs contain quite a few functions with incomplete patterns. To take a simple example:

```
head :: [a] -> a
head (x:xs) = x
head [] = error "head"
```

Of course, `head` is only called when (we think that) we know the argument is a non-empty list. It would be nice if the type system proved this, and one could imagine more sophisticated type systems that could (Freeman & Pfenning [1991]), but Haskell and ML are certainly not rendered unusable by the possibility of such runtime errors. The error in `insert` is arguably in this class.

However, it would really be best to avoid even the possibility of run-time failure, and we can do this by building a `Set` data type that *chooses its representation based on the available operations on elements.* Here is a sketch of one possible implementation:

```
data Set a = Empty
           | List [a]        -- No duplicates
           | Tree (Tree a)

instance Collection Set where
  empty = Empty

  insert x Empty
      = case xCmp of
          Just cmp -> Tree (Branch x Empty Empty)
          Nothing  -> List [x]

  insert x (List xs) = List (insertList xEq x xs)

  insert x (Tree t)
      = case xCmp of
           Just cmp -> Tree (insertTree cmp x t)
```

The point of the game is that `insert` dynamically selects which representation to use in the `Empty` case depending on whether or not there is a comparison operation. Notice, crucially, that *the extraction of* `cmp` *in the final equation for* `insert` *cannot fail, because one of the arguments is already a* `Tree`*, and it could only have become so by virtue of the* `Empty` *equation deciding that there was a comparison operation.*

Not only have we eliminated runtime errors, but we have also delegated to the abstract data type the choice of representation. This is a rather attractive property. When computing with sets, most programmers do not want to have to look up the operations that are available for the element type, and choose which set implementation to use depending on the answer. Being able to use a single type, `Set`, and having the implementation choose the representation automatically is a big advantage. Of course, we are still free to fix a particular representation by using a simpler, more specific set implementation (such as `OrdSet`).

3.4 Efficiency

The generic `Set` implementation sketched above is just a start. A real implementation would be rather cleverer.

- Very small sets should probably be represented by lists even if ordering is available. This is easily programmed.

- A good compiler should be able to create specialised instances of `insert` at widely-used types. For example, if it sees that `insert` is often used at the type

  ```
  insert :: String -> Set String -> Set String
  ```

  then it can create a specialised version of `insert`, in which `c` is fixed to `Set` and `a` is fixed to `String`, and hence the comparison operations ought to be turned into inline code.

There are two other efficiency concerns about `Set` that turn out to be relatively unimportant:

- The implementation of `insert` has to choose which equation to use based on which constructor it finds in its second argument. However, in most implementations the major cost is doing pattern-matching (and hence forcing evaluation) at all; it is very little more expensive to choose between equations based on the constructor found.

- One might worry that every call to `insert` has to pattern-match on `xCmp` to extract the comparison operation, which carries an efficiency cost. This can be done once and for all when a tree is first built:

  ```
  data Tree a = Tree (a->a->Ordering)   -- Comparison
                     Int                -- Size
                     (TreeR a)
  data TreeR a = Empty | Branch a (TreeR a) (TreeR a)
  ```

  On the whole, though, this is probably a bad thing to do. If the implementation fetches the ordering function from the tree, it is less likely that the compiler will be able to prove that for some given type, `Int` say, the ordering function is bound to be `cmpInt`. So it may be less easy for the compiler to generate improved code when the types are known.

3.5 Taking collections apart

The operations on collections we have suggested so far (`empty`, `insert`, `union`) only deal with *constructing* collections. What about taking collections apart? The obvious thing to do is to augment class `Collection` with a homomorphism over the constructors of the collection. Since our "constructors" (so far) are `empty` and `insert` the obvious homomorphism to add to `Collection` is:

```
class Collection c where
  ...
  fold :: (a->b->b) -> b -> c a -> b
```

```
module Collection where

class Collection c where
  empty  :: c a
  null   :: c a -> Bool
  size    :: c a -> Int

  singleton :: (XOps a) => a -> c a
  fromList  :: (XOps a) => [a] -> c a
  toList    :: c a -> [a]

  fold    :: (a->b->b) -> b -> c a -> b
  fold1   :: (a->b->b) -> c a -> b

  filter    :: (XOps a) => (a -> Bool) -> c a -> c a
  partition :: (XOps a) => (a -> Bool) -> c a -> (c a,c a)
  elem      :: (XOps a) => a -> c a -> Bool

  flatMap :: (XOps b) => c a -> (a -> c b) -> c b

  insert     :: (XOps a) => a -> c a -> c a
  insertWith :: (XOps a) => (a->a->a) -> a -> c a -> c a
  insertK    :: (XOps k) => k -> a -> c (Pr k v) -> c (Pr k v)

  union     :: (XOps a) => c a -> c a -> c a
  unionWith :: (XOps a) => (a->a->a) -> c a -> c a -> c a

  delete     :: (XOps a) => a -> c a -> c a
  deleteK    :: (XOps k) => k -> c (Pr k v) -> c (Pr k v)

  lookup      :: (XOps k) => k -> c (Pr k v) -> Maybe v

  intersect, without :: (XOps a) => c a -> c a -> c a
```

Figure 4: The collection class

```
      -- Default methods (part of class declaration)
  size c = fold (\_ n -> n+1) 0 c
  null c = size c == 0

  singleton x = insert x emptyC

  toList c    = fold (:) [] c
  fromList xs = insertList xs emptyC

  filter p c = fold f empty c
             where
               f x r | p x       = x `insert` r
                     | otherwise = r
  partition p c = (filter p c, filter (not.p) c)

  elem x c = fold (\y r -> if (x==y) then True else r) False c

  flatMap c f = fold (union.f) empty c

  insert           = insertWith (\x y -> y)
  insertWith f x c = unionWith f c (singleton x)
  insertK k x c    = insert  (k:>x) c

  union             = unionWith (\x y -> y)
  unionWith f c1 c2 = fold (insertWith f) c1 c2

  delete x c      = filter (/= x) c
  deleteK k c     = delete (k :> error "Collection.deleteK") c

  without   c1 c2 = filter (\x -> not (elem x c2)) c1
  intersect c1 c2 = filter (\x -> elem x c2) c1


-- Standard functions defined using class operations
insertList,deleteList :: (XOps a) => [a] -> c a -> c a
insertList xs c  = foldr insert c xs
deleteList xs c  = foldr delete c xs

unionList, intersectList :: XOps a) => [c a] -> c a
unionList cs      = foldr union empty cs
intersectList cs  = foldr1 intersect cs
```

Figure 5: The collection class continued

The question is, of course, what meaning we should give to a call such as `(fold (-) 0 s)` where `s` is a set. Since `(-)` is not commutative such a call is nonsense. There is no way out of this. All we can do is specify in any particular instance of `Collection` what property `fold` assumes of its arguments. For example, for sets and bags `fold`'s first argument should be left-commutative (i.e. $f\ x\ (f\ y\ a) = f\ y\ (f\ x\ a)$), but there may be instances of `Collection` for which this property need not hold (ones which guarantee to apply `fold` to their elements in sorted order, for example). For arguments that do not satisfy the required properties, `fold` delivers a result based on an unspecified ordering of the elements of the collection.

`fold` is a compositional form of what in STL is called an *iterator*. It lays out the collection in some order, ready to be operated on by some consuming function.

This `fold` is a catamorphism if we regard a collection as built by the constructors (`empty`, `insert`). An equally valid alternative set of constructors is (`empty`, `singleton`, `union`), leading to a different catamorphism:

```
fold' :: (b->b->b) -> (a->b) -> b -> c a -> b
```

These two algebras have been explored by Buneman et al. [1995], who use the terminology `sr_add` for `fold`, and `sr_comb` for `fold'`. We have chosen to use `fold` because it is easier to use than `fold'` — only two arguments need be provided.

As we have seen, `fold` is a bit too powerful because in order to be well defined we have to assume undecidable properties of its argument. Buneman et al. [1995] also discusses ways to avoid this by instead using a function they call `ext`, but which we called `flatMap` in Figure 4. The advantage of `ext/flatMap` is that it requires no particular properties of its argument; yet using it one can define a bunch of useful functions.

3.6 Finite maps

Finite maps (in various guises) are ubiquitous in functional programs. In mathematics, a function (or map) is defined by a set of ordered (argument,result) pairs. The natural thing to do is therefore to represent a finite map by a set of ordered pairs, thus:

```
type FM k v = Set (Pr k v)
data Pr k v = k :> v deriving( Show )

instance (Eq k) => Eq (Pr k v) where


instance (XOps k) => XOps (Pr k v) where
  (k1:>v1) `xEq` (k2:>v2) = k1 `xEq` k2
  xCmp = case xCmp of
           Just cmp -> Just (\(k1:>v1) (k2:>v2) -> k1 `cmp` k2)
           Nothing  -> Nothing
  ...similarly the other operations...
```

Here, `(k :> v)` is a key-value pair, read "`k` maps to `v`". Comparison of a key-value pair is done solely on the basis of the key. It is crucial that we use a new data type for key-value pairs, rather than using the built-in pair constructor, because the latter has equality and ordering instances that look at both components of the pair, not just the first.

A `Set` of key-value pairs, with comparison done on this basis, is a finite map. All that is needed to complete the picture is to add some crucial functions to the `Collection` class:

```
class Collection c where
  ...as before...
  insertWith  :: (XOps a) => (a->a->a) -> a -> c a -> c a
  unionWith   :: (XOps a) => (a->a->a) -> c a -> c a -> c a
  lookup      :: (XOps k) => k -> c (Pr k v) -> Maybe v
```

The "`With`" variants have a function that combines values that compare as equal when doing insertion or union. This is very important when those values are equal because they have equal keys, but we might wish (for example) to add the second component of the pairs.

A disadvantage of this approach is that every instance of `Collection` must, in principle, provide an implementation of `lookup`. While doing so is always possible — indeed one could write a default declaration for `lookup` using `fold` — it is not desirable because for many instances of `Collection` a lookup might be wildly inefficient and inappropriate.

3.7 The complete class

Figures 4 and 5 give the complete definition of the collections module. There are several points to note:

- The type of `elem` is a bit more specific than the default method requires. Again, this is to allow an implementor to make a more efficient `elem` that exploits the ordering on elements.
- If the representation of a non-empty collection always included the necessary comparison operations (see item (1) in Section 3.4), it would be possible to give many operations a rather simpler type, by omitting the `(XOps a)` context. Doing so would place more constraints on the implementor, so we have refrained from doing so.
- `toList` is a pretty dodgy looking operation because `(:)` is not left-commutative. Nevertheless, lists are so ubiquitous (albeit perhaps less so once these libraries are in place!) that it may be more convenient to use `toList` followed by a list operation rather than a single more respectable `fold`. The final result may (indeed should) still be independent of the order in which the `fold` chose to lay out the collection.

4 Second design: multi-parameter constructor classes

Our first design was written in standard Haskell, but it has three fundamental deficiencies:

- It defers to run time some checks that one might intuitively expect to be statically checked.
- It separates sequences and collections entirely, whereas one might have expected that they would share common operations.
- It does not separate (say) lists, from FIFOs, from deques. These are all in class `Sequence` and provide the same operations, but one might prefer the type system to express the idea that FIFOs have more operations than lists, and deques than FIFOs.

Our second design, which we give in much less detail than the first, overcomes both these objections, but at the expense of stepping outside standard Haskell by using *multi-parameter* constructor classes. In the view of the author, the clean way that multi-parameter constructor classes turn out to accommodate bulk types is a very persuasive reason for extending Haskell to embrace them, just as monads provide the key motivation for adding constructor classes.

4.1 The key idea

The key idea is very simple. Suppose we (re-)define the class of collections like this:

```
class Collection c a where
  size   :: c a -> Int
  empty  :: c a
  cons   :: a -> c a -> c a
  union  :: c a -> c a -> c a
  fold   :: (a->b->b) -> b -> c a -> b
  filter    :: (a->Bool) -> c a -> c a
  partition :: (a->Bool) -> c a -> (c a, c a)
```

Notice that `Collection` has two parameters: `c`, the type constructor of he collection, and `a`, the element type. Notice too that `insert` has no `XOps` constraint. The type of `cons`, for example, is now:

```
cons :: Collection c a => a -> c a -> c a
```

The interesting part comes when we define instances of `Collection`:

```
instance Collection [] a where
  empty  = []
  insert = (:)
  ...and so on...

instance Ord a => Collection OrdSeq a where
  empty  = emptyTree
  insert = insertTree
  ...and so on...
```

The exciting thing is that now *we can provide instance-specific constraints on the element type.* In the first instance declaration, for lists, no constraints are placed on `a`, so `insert` can be used on lists without placing any constraints on the element type. In contrast, the second instance declaration specifies that the element type `a` must be in class `Ord`, just what is needed to allow the use of `insertTree` (here assumed to have type `Ord a => a -> Tree a -> Tree a`) to define `insert`.

This simple extension solves at a stroke both of the deficiencies of our first design:

- Things that "should" be checked statically are checked statically. In particular, an attempt to use an `OrdSet` with an element type that has no ordering will provoke an error at compile time rather than at run time.
- The same class embraces both collections with constraints on the elements, and collections with none (termed sequences of the first design). There is no need for both a `filter` on sequences and a separate `filter` on collections; plain `filter` will work on both.

It remains possible to have adaptive representations for collections, using the same `XOps` class as before, thus:

```
instance XOps a => Collection Set a where
  ...as before...
```

This instance declaration makes clear that the type `Set` of adaptive sets requires its element type to be in class `XOps`. The implementation can now be given exactly as before.

Notice that `MonadZero` and `Functor` are not superclasses of `Collection`, as they were of `Sequence`, because not all instances of `Collection` could be instances of `Monad` since the latter requires operations polymorphic in the elements. We can still make *particular* bulk types (the polymorphic ones) instances of `Monad`, of course, by giving a suitable instance declaration, so we are not giving up the possibility of using monad comprehensions to create and filter collections.

4.2 Using the class hierarchy

It seems obvious that sequences should have all the operations that unordered collections have, and some more besides. Now that the operations in `Collection` apply to sequences as well as unordered collections, we can use the class hierarchy to express precisely the inheritance we want:

```
class (Collection s a) => Sequence s a where
  snoc  :: s a -> a -> s a
  first :: s a -> SeqView s a
  last  :: s a -> SeqView s a
  foldl :: (b->a->b) -> b -> s a -> b
  reverse :: s a -> s a
```

There are now quite a few design decisions to make. For example:

- Does one want one class that supports `front` but not `back`, another that supports `back` but not `front`, and a third that combines these capabilities? Or is it best to have one class (such as the `Sequence` just defined above) that has both. After all, one can get the last element of a list — it's just rather inefficient to do so.

- Should `cons` (implying "add an element to the front" for sequences, and just "add an element" for unordered collections) be in `Collection`, and `snoc` ("add an element to the back") in `Sequence`, or should both be in `Sequence`, with some other subclass of `Collection` having a neutral `insert` for unordered collections?
- Similar questions arise for `fold` and its directional cousins `foldr` and `foldl`.
- Should every collection support `union` when for some it may be a constant time operation while for others it is an O(N) operation?

The answers to these questions are not obvious, but the the collection classes of Smalltalk and C++ provide a good deal of guidance. For example, Smalltalk's collection-class hierarchy looks like this:

```
Collection
  Bag
  Set
    Dictionary
  Sequencable collection
    Interval
    LinkedList
    OrderedCollection
      SortedCollection
    ArrayedCollection
      Array
```

4.3 Finite maps

Finite maps can be still handled exactly as described in Section 3.6, but multi-parameter type classes opens up another intriguing possibility:

```
class Collection (c k) a => FM c k a where
  extend :: k -> a -> c k a -> c k a
  lookup :: c k a -> c -> k -> a
```

This declares the three-parameter type class `FM`, parameterised over `c`, the type constructor of the map, `k`, the key type, and `a`, the value type. It requires that the partial application of `c` to `k` is a collection type constructor. Now all the collection operations work on finite maps, but the latter add two new operations, `extend` and `(!!)` (i.e. lookup).

One advantage of this approach is that it makes it possible to include Haskell's standard arrays in `FM` — which is nice, because arrays are plainly finite maps:

```
instance Ix k => FM Array k a where
  lookup = (!!)
```

Of course, Haskell arrays don't support `extend` or any of the operations in `Collection`, so one might change the hierarchy to look like this:

```
class Indexable c k a where
  lookup :: c k a -> c -> k -> a

class (Collection (c k) a, Indexable c k a) => FM c k a where
  extend :: k -> a -> c k a -> c k a
```

Again, there are many possible design choices.

4.4 Summary

Multi-parameter constructor classes seem to be just what is needed to make a clean job of bulk types. What we have done here is only to sketch the basic idea. A considerable amount of design work remains to flesh it out into a concrete design, even assuming the existence of multi-parameter constructor classes.

5 Related work

There is a large literature on collection types, also known as *bulk types*. Tannen [1994] gives a useful bibliography, from a database perspective. Buneman et al. [1995] explores the algebra and expressiveness of algebras based on (`empty`, `insert`) and (`empty`, `singleton`, `union`).

C++ has a well-developed library called the Standard Template Library (STL) which is specifically aimed at collection types (Stepanov & Lee [1994]). There are major differences from the work described here. Rather than a collection being a value which can be combined with other similar values, it is regarded as a container into which new values can be placed. There is no equivalent of `fold`; instead *iterators* are provided, which specify a location within a container. It does handle polymorphism, however, using C++ templates; when a collection is declared one specifies both the element type and the comparison operation to use.

Parametric type classes (Chen, Hudak & Odersky [1992]) have similar power to multi-parameter constructor classes. Indeed Chen's thesis uses bulk types as the main motivating example for parametric type classes (Chen [1994]).

6 Summary

Designing suitable signatures for bulk types is surprisingly tricky. The number of different kinds of collection, and the number of possible implementations of each kind of collection, makes it rather unattractive to use distinct names for the operations of each. Furthermore, if we do so we cannot write polymorpic algorithms; that is, algorithms that work regardless of which kind of collection is involved.

The first design proposed here exploits type classes to obtain a substantial amount of polymorphism. Algorithms can be polymorphic over the elements of the collection, the implementation of the collection, and the nature of the collection.

Apart from the use of type classes, the two key design decisions are these:

- At first sight it seems attractive to unify all bulk types into a single class. We propose instead to use two classes, one for sequences and one for collections. Sequences are parametric in their element type, and are sensitive to insertion order, while the reverse holds for collections.
- We solve the typing problems of collections with the `XOps` class, thereby requiring a small amount of run-time type-checking (at least when the types are not statically known). Whilst it is not perfect, this can be turned to our advantage by allowing the programmer to design data types that choose their representation based on the operations available for the element type.

The second design uses multi-parameter type classes to unify sequences and unordered collections into a single class hierarchy. It seems to be a noticeably cleaner solution, but requires a signficant extension to Haskell.

An attractive property of both designs is the possibility of writing adaptable implementations, that automatically choose their representation based on the operations available on the underlying data type.

Acknowledgements

I would like to thank Peter Buneman, Laszlo Nemeth, and David Watt, for their helpful comments on earlier drafts of this paper. Mark Jones and Peter Thiemann both pointed out to me the beautiful fit between multi-parameter constructor classes and bulk types that is sketched in Section 4. They both also gave me particularly useful feedback on other aspects of the paper.

I gratefully acknowledge the support of the Oregon Graduate Institute during my sabbatical, funded by a contract with US Air Force Material Command (F19628-93-C-0069).

References

GS Brodal & C Okasaki [Dec 1996], "Optimal purely-functional priority queues," *Journal of Functional Programming* 6.

P Buneman, L Libkin, D Suciu, V Tannen & L Wong [March 1994], "Comprehension syntax," in *SIGMOD Record* #23 #1, 87–96.

P Buneman, S Naqvi, V Tannen & L Wong [Sept 1995], "Principles of programming with complex objects and collection types," *Theoretical Computer Science* 149.

K Chen [1994], "A parametric extension of Haskell's type classes," PhD thesis, Department of Computer Science, Yale University.

K Chen, P Hudak & M Odersky [June 1992], "Parametric type classes," in *ACM Symposium on Lisp and Functional Programming, Snowbird*, ACM.

T-R Chuang & WL Hwang [May 1996], "A probabilistic approach to the problem of automatic selection of data representations," in *Proc International Conference on Functional Programming, Philadelphia*, ACM, 190–200.

T Freeman & F Pfenning [June 1991], "Refinement types for ML," in *SIGPLAN Symposium on Programming Language Design and Implementation (PLDI'91), Toronto*, ACM, 268–277.

MP Jones [Jan 1995], "A system of constructor classes: overloading and implicit higher-order polymorphism," *Journal of Functional Programming* 5, 1–36.

C Okasaki [May 1996], "The role of lazy evaluation in amortized data structures," in *Proc International Conference on Functional Programming, Philadelphia*, ACM, 62–72.

C Okasaki [Oct 1995], "Amortization, lazy evaluation, and persistence: lists with catenation via lazy linking," in *IEEE Symposium on Foundations of Computer Science*, 646–654.

A Stepanov & M Lee [Dec 1994], "The Standard Template Library," Hewlett-Packard Laboratories, Palo Alto.

V Tannen [1994], "Tutorial: languages for collection types," University of Pennsylvania.

PW Trinder [Aug 1991], "Comprehensions: a query notation for DBPLs," in *Proc 3rd International Workshop on Database Programming Languages, Nahplion, Greece*, Morgan Kaufman, 49–62.

PW Trinder & PL Wadler [1989], "List comprehensions and the relational calculus," in *Functional Programming, Glasgow 1988*, Workshops in Computing, Springer Verlag, 115–123.

PL Wadler [1992], "Comprehending monads," *Mathematical Structures in Computer Science* 2, 461–493.