When is an abstract data type a functor?

Pablo Nogueira

School of Computer Science and IT
University of Nottingham, UK

Abstract

A parametric algebraic data type in a functional program is a functor if we can define a map operation on that type that satisfies certain equations. Interestingly, in the equations the definitional structure of the type is immaterial. In this paper we recall the category-theoretic concept of functor, discuss it in programming terms, and investigate the conditions under which first-order parametric *abstract* data types (ADTs) can be functors. First, we provide an answer 'from inside', based on a conceptual model of ADT implementations. Then we provide an answer 'from outside' in terms of ADT interfaces.

1 INTRODUCTION

A parametric algebraic data type in a functional program is a functor if we can define a map operation on that type that satisfies certain equations. More precisely, a first-order unary type constructor `F` is a *covariant* functor if we can define a function `mapF :: (a → b) → F a → F b` that satisfies the following 'functorial laws':

```
mapF (f ∘ g)          ==  mapF f ∘ mapF g
mapF (id :: a → a)    ==  (id :: F a → F a)
```

(We have used explicit type annotations in the last equation to illustrate the types of each **id** instance.) For example, the `List` type constructor is a typical functor:

```
data List a = Nil | Cons a (List a)
```

whose map operation is defined as follows:

```
map :: (a → b) → List a → List b
map f Nil         = Nil
map f (Cons x xs) = Cons (f x) (map f xs)
```

The reader can verify that **map** satisfies the functorial laws.

A map operation for a first-order type constructor respects (or *ignores*, if you will) the *shape* of the data type, mapping only the data contents or *payload*. Interestingly, nothing in the map laws require `F` to be a concrete type. Can we convey the functor concept to *abstract* data types? Unlike concrete types, which are free algebras, abstract data types (ADTs) have structural context-dependent properties such as ordering, lack of repetition, etc, which cannot be captured by algebraic type

definitions. Instead, properties are stated as equational laws involving interface operators. Furthermore, parametric ADTs often have their type-argument range constrained, *i.e.*, they are implemented using *constrained* algebraic data types which are type constructors with some or all of their type-variable arguments constrained by type classes.

In this paper we recall the category-theoretical concept of functor (Section 2.1), discuss it in programming terms (Sections 2.2 and 2.3) (showing the role of type-class constraints in the conditions for functoriality) and investigate the conditions under which *first-order* ADTs can be functors. First, we provide an answer 'from inside' based on a conceptual model of ADT implementations (Section 3). Then we provide an answer 'from outside' in terms of ADT interfaces (Section 4).

2 TYPES AND FUNCTORS

2.1 Categories and functors

The functor concept comes from category theory. A mathematical structure constitutes a category if it is possible to identify a collection of 'entities with structure' (called *objects*), a collection of 'structure-preserving' maps between them (called *arrows*), and a binary arrow-composition operation that must be partial (not all arrows compose), closed (yields an arrow in the category), associative, and has unique left and right neutral element (the identity arrow) for every object.

In this paper, the category of interest is **Type**, the category of types where, broadly, objects are monomorphic algebraic types and arrows are functional programs involving those types. In this category, arrow composition is function composition and identity arrows are instances of the polymorphic identity function for specific types: *e.g.*, $\mathtt{id}_{\mathtt{Int}}$ with type $\mathtt{Int} \to \mathtt{Int}$, $\mathtt{id}_{\mathtt{Bool}}$ with type $\mathtt{Bool} \to \mathtt{Bool}$, etc.

A *functor* is a total map from a category **C** to another category **D** where objects and arrows in **C** are mapped respectively to objects and arrows in **D** while preserving the categorical structure. More precisely, given a category **C**, let us write $Obj(\mathbf{C})$ for its collection of objects, $Arr(\mathbf{C})$ for its collection of arrows, and let us write $\circ$ for arrow composition. A *covariant* functor $F :: \mathbf{C} \to \mathbf{D}$ consists of two total maps with overloaded name, *i.e.*, $F :: Obj(\mathbf{C}) \to Obj(\mathbf{D})$ and $F :: Arr(\mathbf{C}) \to Arr(\mathbf{D})$, which satisfy the following conditional definitions (expressed in natural-deduction style):

$$\frac{A \in Obj(\mathbf{C})}{F(A) \in Obj(\mathbf{D})} \qquad \frac{f \in Arr(\mathbf{C}) \qquad f :: A \to B}{F(f) \in Arr(\mathbf{D}) \qquad F(f) :: F(A) \to F(B)}$$

and, at the arrow level, F satisfies the *functorial laws*:

$$\begin{aligned} F(f \circ g) &= F(f) \circ F(g) \\ F(\mathrm{id}_A) &= \mathrm{id}_{F(A)} \end{aligned}$$

The functorial laws guarantee that the functor preserves the categorical structure. A functor is *contravariant* when $f :: A \to B$ but $F(f) :: F(B) \to F(A)$.

In the category of types, at the *object level* functors are *unconstrained* type constructors. By 'unconstrained' we mean that the range of the $\forall$ in the type signature of the map function is the whole collection of objects (*i.e.*, monomorphic types). We have already mentioned `List` as an example. It maps the monomorphic type **`Int`** to the monomorphic type `List` **`Int`**, the monomorphic type **`Bool`** to the monomorphic type `List` **`Bool`**, and so on.

In the category of types, at the *arrow level* a functor is identified with the map operation for the type. For lists, at the arrow level F is **`map`**, which satisfies the functorial laws.

2.2 Unconstrained types

Monomorphic types such as **`Int`** or **`Bool`** are not functors. The functor concept applies to type constructors. But monomorphic types can be lifted to functors where map is the identity, *e.g.*:

```
type IntF a = Int
mapIntF :: (a → b) → IntF a → IntF b
mapIntF f = (id :: Int → Int)
```

First-order unary type constructors are functors if we can find a map operator satisfying the functorial laws. Haskell even provides a type class for first-order unary functors:

```
class Functor f where
  fmap :: ∀ a b. (a → b) → f a → f b
```

Programmers can make their types functors by providing the instance declaration but, unfortunately, no Haskell compiler will check that the definition of `fmap` provided by the instance satisfies the functorial laws.

Extending the concept of functor to first-order *n-ary* type constructors is trivial. For instance, a binary type constructor `F` is a covariant functor if we can define:

```
mapF :: (a → c) → (b → d) → F a b → F c d
```

that satisfies:

```
mapF (f ∘ g) (h ∘ i) == mapF f h ∘ mapF g i
mapF (id :: a → a) (id :: b → b) == (id :: F a b → F a b)
```

The extension to *higher-order* type constructors is more elaborate. For example, given type constructor:

```
data F f a = ...
```

where `f` has kind $* \to *$, if `F` is a functor then `f` must be a functor. For this particular example it is possible to impose such a restriction using type classes:

```
data Functor f ⇒ F f a = ...
```

However, the **Functor** class is not enough for higher-order types because only types of kind $* \to *$ can be instances. Provided `f` is a functor, then `F` is a functor if we can define:

```
mapF :: ∀ f g. (f c → g c) → (a → b) → F f a → F g b
```

that satisfies:

```
mapF (μ ∘ η) (i ∘ j) == mapF μ i ∘ mapF η j
mapF (id :: ∀a. f a → f a) (id :: a → a)
  == (id :: F f a → F f a)
```

where μ and η are polymorphic functions (natural transformations) both of type $\forall f \forall g.\ f\ a \to g\ a$, and the first argument to `mapF` is the identity natural transformation. The functoriality requirement on f and g is needed for μ and η to be natural transformations, *i.e.*, to satisfy $\mu \circ \mathrm{map}_f\ i = \mathrm{map}_g\ i \circ \mu$ for any $i :: a \to b$. Notice that the map operation for higher-order types not only maps payload but also maps shape. Natural transformations pop up because the first argument of `F` must be a functor, and functors considered as objects form a category where arrows are natural tranformations.

The following is a non-functor example:

```
type Fix a = (a → a) → a
```

It is the type of a fixed-point combinator:

```
fix :: Fix a
fix f = f (fix f)
```

The type of `mapFix`, were it to exist, would be:

```
(a → b) → ((a → a) → a) → ((b → b) → b)
```

But when we try to write the body of `mapFix f h` we find there is no value of type `a` to which we can apply `f` or `h`. Similarly, assuming `Fix` might be contravariant, there is no value of type `b` to which we can apply `f`. (Nonetheless, the binary function-space type constructor '→' *is* a functor contravariant on its first argument.)

2.3 Constrained types

A type-class constraint imposes a bound on the type-argument range of a type constructor. For example, the following type:

```
data Ord a ⇒ List a = Nil | Cons a (List a)
```

can only be applied to monomorphic types that are instances of **Ord**. In Haskell, constraints are not associated with types but with value constructors so that construction or pattern-matching give rise to them. In short, the declaration is equivalent to:

```
data List a where
  Nil  :: ∀ a. List a
```

```
Cons :: ∀ a. Ord a ⇒ a → List a → List a
```

Somewhat intriguingly, the constraint is not imposed on the nullary product!

A constraint is a property of the payload, not the type constructor. The **Ord** constraint does not indicate that `List` is ordered. That sort of context-dependent property must be captured by an equational law on `List`'s constructors; that is, an ordered list is an ADT and its implementation should be hidden. (To belabour the point, replace **Ord** by **Num** if you cannot help 'seeing' an ordered list.)

The map function for an unconstrained polymorphic list is insensitive to the payload type. It is insensitive even when the payload range is restricted. Consequently, the *body* of **map** for unconstrained lists also computes the map for constrained lists. However, constraints must appear in the type signature, for they arise when pattern-matching against value constructors:

```
map :: (Ord a, Ord b) ⇒ (a → b) → List a → List b
map f Nil         =  Nil
map f (Cons x xs) = Cons (f x) (map f xs)
```

The implicit dictionary arguments introduced by constraints are ignored in the function body. Had the list type been constrained differently, **Num** say, then the type signature would differ:

```
map :: (Num a, Num b) ⇒ (a → b) → List a → List b
```

while the body would remain identical. Constraints only affect type signatures. For this reason, we cannot make a constrained list an instance of **Functor**, for `fmap` is universally polymorphic. In order to resolve the problem we have to provide a new *multi-parameter* **Functor** class that is parametric on payload types:

```
class Functor f a b where
 fmap ::  (a → b) → f a → f b

instance (Ord a, Ord b) ⇒ Functor List a b where
  fmap f Nil         = Nil
  fmap f (Cons x xs) = Cons (f x) (fmap f xs)
```

We now provide a category-theoretic explanation of the role of constraints in the conditions for functoriality.

DEFINITION 2.1 A category **S** is a *proper subcategory* of a category **C** when $Obj(\mathbf{S}) \subset Obj(\mathbf{C})$, $Arr(\mathbf{S})$ contains all the arrows in $Arr(\mathbf{C})$ involving S-objects, and composition and identities in $Arr(\mathbf{S})$ are those in $Arr(\mathbf{C})$ involving S-objects. □

First-order constrained types are functors in subcategories of the category of types. For example, an **Ord**-constrained type F is a functor $F :: \mathbf{Ord} \to \mathbf{Type}$ where **Ord** is the subcategory whose objects are monomorphic types with an ordering relation. (If it is possible to derive the order relation in $F(X)$ for every $X \in Obj(\mathbf{Ord})$, then $F :: \mathbf{Ord} \to \mathbf{Ord}$.) Annoyingly, when it comes to provide the arrow-level definition of F, the subcategory in question is irrelevant.

2.4 ADTs as signatures with laws

In order to establish whether an ADT is a functor we must make precise our definition of ADT. We focus on ADTs that are parametric, first-order, and described by an algebraic specification with partial operators and conditional equations, the latter needed to cope with partiality [Mar98, LEW96]. This is an example of a conditional equation in a FIFO-queue specification:

```
isEmpty q == False ⇒ front (enq x q) == front q
```

A conditional equation is required because `front` is a partial operator: when applied to an empty queue its value is undefined.

Algebraic specifications are axiomatic formal systems for specifying ADTs precisely, unambiguously, and independently of their implementation. They have universal-algebra semantics (*e.g.*, final or initial many-sorted partial algebras), they can be used in the formal construction and verification of client code, there is a formal relation between the specification and the implementation [Mar98], and prototype implementations can be obtained automatically, *e.g.*, [GWM+93]. Mainstream functional languages do not support equational specifications; they only provide mechanisms for specifying interfaces (signatures) and implementations. In Haskell, the module and type-class systems are used for this purpose. But equational specifications are needed to reason formally about properites of ADTs.

3 WHEN IS AN ADT A FUNCTOR (FROM INSIDE)?

ADTs arise because the type-definition mechanisms offered by programming languages are not expressive enough to capture a programmer's *imagined* (abstract) type. Or to replace a negative with a positive: the *implementation* type is too big. The values of the abstract type are represented (simulated) by a *subset* of the values of the implementation type, those that fulfil some particular criteria. The implementation type is hidden behind an interface of operators that only take and return values of the implementation type that represent values of the abstract type. The equational laws restrict the way values of the implementation type are constructed and observed, with the consequence that often construction and observation are not inverses.

It is typical for implementation types to contain *clutter*, *i.e.*, extra data required for efficiency or structuring purposes. Examples of clutter are the colour of nodes in Red-Black Trees, the height of a sub-tree in a heap, the length of the lists in a *banker's* implementation of FIFO queues, etc [Oka98]. Because of the well-known trade-off between time and space, clutter is more the norm than the exception: data (space) will be used to improve operator speed (time). With this in mind, we provide the following definitions.

DEFINITION 3.1 An ADT A with payload a is a functor iff we can define map_A

that satisfies:

$$\begin{array}{rcl} \mathrm{map}_A & :: & (a \to b) \to A\ a \to A\ b \\ \mathrm{map}_A\ (f \circ g) & = & \mathrm{map}_A\ f \circ \mathrm{map}_A\ g \\ \mathrm{map}_A\ \mathrm{id}_a & = & \mathrm{id}_A \end{array}$$

□

We will define map_A in terms of a *conceptual* model of ADT implementations.

DEFINITION 3.2 An ADT A with payload a consists of:

- An implementation type $I\ a = C\ a \times P\ a$ where $C\ a$ is the concrete type of clutter and $P\ a$ is the concrete type of payload. (Clutter must be either of monomorphic type or of payload type. In the former case, $C\ a = T$ for some monomorphic type T.)
- A predicate $Rep :: I\ a \to Bool$ such that for every value i of type $I\ a$, $Rep(i)$ holds when i represents a value of the abstract type.
- A function $\phi :: C\ b \times P\ a \to I\ a$ such that ϕ satisfies $\forall c \forall p.\ Rep(\phi(c,p))$. We call ϕ a 'repair function'.

□

A repair function is defined per ADT in terms of the particular clutter and payload types. Notice that for a clutter value c of type $C\ b$, $\phi(c,p)$ must compute a new clutter value of type $C\ a$.

DEFINITION 3.3 Let P be a functor. We define function σ that maps its argument function over the payload while ignoring the clutter:

$$\begin{array}{lcl} \sigma & :: & (a \to b) \to (C\ d \times P\ a) \to (C\ d \times P\ b) \\ \sigma\ f\ (c,p) & = & (c,\ \mathrm{map}_P\ f\ p) \end{array}$$

□

LEMMA 3.1 If P is a functor then σ is 'functorial', *i.e.*:

$$\begin{array}{rcl} \sigma\ (f \circ g) & = & \sigma\ f \circ \sigma\ g \\ \sigma\ \mathrm{id} & = & \mathrm{id} \end{array}$$

□

The proof of the lemma follows from the definition of σ.

Let P be a functor and let $\mathrm{map}_A\ f = \phi \circ \sigma\ f$. Does it satisfy the functorial laws? Let us try the second law first:

$$
\begin{array}{ll}
 & \mathrm{map}_A\ \mathrm{id} = \mathrm{id} \\
= & \quad \{\ \text{def. } \mathrm{map}_A\ \} \\
 & \phi \circ \sigma\ \mathrm{id} = \mathrm{id} \\
= & \quad \{\ \sigma \text{ is 'functorial'}\ \} \\
 & \phi \circ \mathrm{id} = \mathrm{id} \\
= & \quad \{\ \text{def. } \phi\ \} \\
 & \mathrm{true}
\end{array}
$$

In the last step, we have used the fact that map_A is only applied to values of $I\ a$ representing values of $A\ a$. Let us now tackle the first law:

$$
\begin{array}{ll}
 & \mathrm{map}_A\ (f \circ g) = \mathrm{map}_A\ f \circ \mathrm{map}_A\ g \\
= & \quad \{\ \text{def. } \mathrm{map}_A\ \} \\
 & \phi \circ \sigma\ (f \circ g) = \phi \circ \sigma\ f \circ \phi \circ \sigma\ g \\
= & \quad \{\ \sigma \text{ 'functorial'}\ \} \\
 & \phi \circ \sigma\ f \circ \sigma\ g = \phi \circ \sigma\ f \circ \phi \circ \sigma\ g
\end{array}
$$

The last equation is a condition that must be checked in order for A to be a functor. In words the equation states that if we map g on the payload, then we map f on the payload, and finally we repair, we must get the same value as if we map g on the payload, then we repair, then we map f, and finally we repair.

DEFINITION 3.4 Let A be an ADT (Definition 3.2). A is a functor if P is a functor and the *repair condition* $\phi \circ \sigma\ f \circ \sigma\ g = \phi \circ \sigma\ f \circ \phi \circ \sigma\ g$ is satisfied. □

3.1 Examples

Instead of the `List` type mentioned so far, the examples below use Haskell's built-in list type `[a]`. Details about the implementaions used can be found in [Oka98].

FIFO queues. We present only the *batched* and *physicist's* implementation. There is clutter only in the latter. In both cases, FIFO queues are functors. The batched implementation is shown first:

```
data BatchedQueue a = BQ [a] [a]
```

The first list contains front elements and the second list contains rear elements in reverse order. When the front list is emptied, the rear list is rotated and becomes the front list. We model this implementation in terms of clutter and payload types thus:

```
type C a = Unit
type P a = ([a],[a])
type BatchedQueue a = (C a, P a)
φ :: (C b, P a) → BatchedQueue a
φ = id
```

The reader can check that $\mathrm{map}_{\mathrm{FIFO}}\ f = \phi \circ \sigma\ f$ satisfies the repair condition and behaves like the map on the original implementation:

```
mapFIFO f (BQ fs rs) = BQ (map f fs) (map f rs)
```

Now to the physicist's implementation:

```
data PhysicistQueue a = PQ [a] Int [a] Int [a]
```

The first list is a prefix of the second. The integers hold the lengths of the second list (which contains the front elements) and third list (which contains the rear elements in reverse order). Elements are moved from the rear list r to the front list f when $\text{length}(f) = \text{length}(r)+1$. However, in a lazy language the move is performed on demand and a prefix of f is cached for efficiency. Accordingly:

```
type C a = ([a],Int,Int)    -- prefix list and lengths
type P a = ([a],[a])
type PhysicistQueue a = (C a, P a)
ϕ :: (C b, P a) → PhysicistQueue a
ϕ ((cs,lf,lr),(fs,rs)) = ((cs',lf,lr),(fs,rs))
 where cs' = prefix fs
```

Function `prefix` returns the required prefix of the front list. The lengths of the front and rear lists are not affected by the list maps performed by σ. The reader can check that $\text{map}_{\text{FIFO}}\ f = \phi \circ \sigma\ f$ satisfies the repair condition and behaves like the map on the original implementation:

```
mapFIFO f (PQ cs lf fs lr rs) = PQ cs' lf fs' lr rs'
  where fs' = map f fs
        rs' = map f rs
        cs' = prefix fs'
```

Ordered sets. We use the *ordered list* and *leftist heap* implementation. There is clutter only in the latter. In both cases, ordered sets are functors. First, the ordered-list implementation:

```
data Ord a ⇒ OrdList a = OL [a]
```

Accordingly:

```
type C a = Unit
type P a = [a]
type OrdList a = (C a, P a)
ϕ :: (Ord a, Ord b) ⇒ (C b, P a) → OrdList a
ϕ (c,p) = (c , (sort ∘ nub) p)
```

Standard function **nub** removes duplicates from a list. The reader can check that $\text{map}_{\text{OrdSet}}\ f = \phi \circ \sigma\ f$ satisfies the repair condition and behaves like:

```
mapOrdSet f (OL xs) = OL ((sort ∘ nub ∘ map f) xs)
```

Now to leftist heaps:

```
data Ord a ⇒ LHeap a = E | N Int a (LHeap a) (LHeap a)
```

Accordingly:

```
data Ord a ⇒ C a = EC | NC Int (C a) (C a)
data Ord a ⇒ P a = EP | NP a (P a) (P a)
type Heap a = (C a, P a)
ϕ :: (Ord a, Ord b) ⇒ (C b, P a) → LHeap a
ϕ (_,p) = (decorate p′, p′)
 where p′ = (fixP ∘ nubP) p
```

Function `fixP` sorts and balances a `P`-heap. Function `nubP` removes repeated elements. Function `decorate` returns a `C`-heap structurally isomorphic to a `P`-heap but with height values. The reader can check that $\text{map}_{\text{OrdSet}}\ f = \phi \circ \sigma\ f$ satisfies the repair condition.

4 WHEN IS AN ADT A FUNCTOR (FROM OUTSIDE)?

We state functoriality conditions independently of ever-changing implementation details given with respect to an artificial model.

DEFINITION 4.1 Let A be an abstract type and P a concrete type both with payload a. A is a functor if the following conditions hold:

1. P is a functor.
2. We can define an *extraction* function $\varepsilon :: A\ a \to P\ a$ and an *insertion* function $\iota :: P\ a \to A\ a$ using the operators in A's interface and any function definable on P with the condition that $\iota \circ \varepsilon = \text{id}_A$.
3. The repair condition of Definition 3.4 holds where now:

$$\begin{array}{lcl} \sigma & :: & (a \to b) \to A\ a \to P\ b \\ \sigma\ f & = & \text{map}_P\ f \circ \varepsilon \\ \phi & :: & P\ a \to A\ a \\ \phi & = & \iota \\ \text{map}_A & = & \phi \circ \sigma\ f \end{array}$$

In other words, the following condition holds:

$$\iota \circ \text{map}_P\ (f \circ g) \circ \varepsilon \quad = \quad \iota \circ \text{map}_P\ f \circ \varepsilon \circ \iota \circ \text{map}_P\ g \circ \varepsilon$$

□

The type of σ indicates that it is a function that 'forgets' what is not payload (*e.g.*, clutter), 'collects' it in P and maps over it. The type of ϕ indicates that it is a function that 'repairs' a value of $P\ a$ by 'turning it' into a value of $A\ a$. In practice, we should choose P to be the concrete type with an isomorphic signature to A but with no laws and more operators (Section 5.3).

The functorial law $\text{map}_A\ \text{id}_a = \text{id}_A$ is satisfied when P is a functor and $\iota \circ \varepsilon = \text{id}_A$:

$$
\begin{array}{ll}
 & \mathrm{map}_A\ \mathrm{id}_a\ =\ \mathrm{id}_A \\
= & \quad \{\ \text{def. } \mathrm{map}_A\ \} \\
 & \iota \circ \mathrm{map}_P\ \mathrm{id}_a \circ \varepsilon\ =\ \mathrm{id}_A \\
= & \quad \{\ P \text{ functor}\ \} \\
 & \iota \circ\ \mathrm{id}_P \circ \varepsilon\ =\ \mathrm{id}_A \\
= & \quad \{\ \mathrm{id}_P \circ \varepsilon\ =\ \varepsilon\ \} \\
 & \iota \circ \varepsilon\ =\ \mathrm{id}_A \\
= & \quad \{\ \text{Definition 4.1(2)}\ \} \\
 & \text{true}
\end{array}
$$

4.1 Examples

We illustrate Definition 4.1 with three examples: FIFO queues, stacks, and ordered sets. For all three we can choose P to be `List` and we can find suitable ε and ι satisfying $\iota \circ \varepsilon = \mathrm{id}$ and the repair condition. We start with FIFO queues:

```
ε  :: FIFO a → List a
ε q = if emptyQ q then Nil else Cons (front q) (ε (deq q))

ι  :: List a → FIFO a
ι l = if null l then emptyQ else enq (head l) (ι (tail l))
```

However, for stacks, we have to implement ι using different list operators in order for $\iota \circ \varepsilon$ to be the identity:

```
ε  :: Stack a → List a
ε s = if emptySt s then Nil else Cons (tos s) (ε (pop s))

ι  :: List a → Stack a
ι l = if null l then emptySt else push (snoc l) (ι (init l))
```

In the case of ordered sets, the choice of list operators in ι is irrelevant, for ι is implemented using `insert :: a → OrdSet a → OrdSet a` which is an 'intelligent' constructor that guarantees that payload elements end up in the right place within the abstract type:

```
ε  :: Ord a ⇒ OrdSet a → List a
ε s = if emptyS s then Nil
      else Cons (min s) (ε (remove (min s) s))

ι  :: Ord a ⇒ List a → OrdSet a
ι l = if null l then emptyS else insert (head l) (ι (tail l))
```

The partial operator `min :: OrdSet a → a` returns the minimum element in a non-empty ordered set. The total operator `remove :: a → OrdSet a → OrdSet a` removes the specified element from the set when it is in the set, otherwise behaves like the indetity.

5 FURTHER DISCUSSION

5.1 Extension to *n*-ary ADTs

Definitions 3.4 and 4.1 can be extended to first-oder *n*-ary ADTs by replacing a, f, and g for $\overline{a}$, $\overline{f}$, and $\overline{g}$, respectively, where $\overline{x} = x_1 \ \ldots \ x_n$ and $n > 0$.

5.2 Construction vs observation

The meaning of a type is typically given by an algebra. Let X be the carrier of the algebra giving meaning to a fixed payload type. The meaning of a unary type parametric on that payload is an algebra $\alpha_X :: F_X(A) \to A$ where $\alpha = (c_1 \triangledown \ldots \triangledown c_n)$, $n > 0$, c_i are the constructor operators in the signature of the type, F_X is a functor expression defined as the disjoint sum of the types of constructor arguments, and A is the carrier type, *e.g.*, the values in FIFO(X). The function $\triangledown :: (A \to C) \to (B \to C) \to A + B \to C$ pattern-matches on a sum value and applies the appropriate argument function to the alternative [MFP91]. In a *least* or *initial* algebra semantics, $A = \mu F_X$.

For example, the ADTs in Section 4.1 were *linear structures*, *i.e.*, structures whose signature is isomorphic to the signature of lists. The signatures are all described by the functor expression $F_X(Y) = 1 + X \times Y$. The initial algebra A is the solution to the equation $F_X(A) = 1 + X \times A$, which is μF_X. For instance:

$$\begin{array}{rcl}
(\texttt{Nil} \triangledown \texttt{Cons})_X & :: & F_X(\text{List}(X)) \to \text{List}(X) \\
(\texttt{emptyQ} \triangledown \texttt{enq})_X & :: & F_X(\text{FIFO}(X)) \to \text{FIFO}(X) \\
(\texttt{emptySt} \triangledown \texttt{push})_X & :: & F_X(\text{Stack}(X)) \to \text{Stack}(X) \\
(\texttt{emptyS} \triangledown \texttt{insert})_X & :: & F_X(\text{OrdSet}(X)) \to \text{OrdSet}(X)
\end{array}$$

The fact that we can express an ADT's signature using some functor expression F_X does not mean the ADT is a functor. ADTs have laws among constructors c_i which are not captured by their signatures. *Functions* ε *and* ι *and the repair condition arise because of these laws.*

Consequently, given the coalgebra $\alpha_X^{-1} :: A \to F_X(A)$ it need not be the case that the observers in α_X^{-1} are inverses of the constructors in α_X. More precisely, only when A is free then $(c_1 \triangledown \ldots \triangledown c_n)^{-1} = (c_1^{-1} \blacktriangledown \ldots \blacktriangledown c_n^{-1})$, where $\blacktriangledown$ is the inverse of $\triangledown$. For this reason, observation and construction must be separated in ADTs.

For example, *given the FIFO queue operators available* we can provide the coalgebra $(f \blacktriangledown g)_X :: \text{FIFO}(X) \to F_X(\text{FIFO}(X))$ where:

$$\begin{array}{lcl}
f & = & \texttt{isEmptyQ}^{-1} \\
g & = & \texttt{front} \vartriangle \texttt{deq} \\
\vartriangle & :: & (C \to A) \to (C \to B) \to C \to A \times B \\
(x \vartriangle y)\ z & = & (x\ z,\ y\ z)
\end{array}$$

However, $g^{-1} \neq$ `enq` because of the following equation which FIFO queues must satisfy when `q` is not empty:

```
deq (enq x q) == enq x (deq q)
```

By replacing `x` for `front q` and reversing the equation we arrive at:

```
enq (front q) (deq q) == deq (enq (front q) q)
```

which in point-free style reveals that `enq` (`front` $\triangle$ `deq`) $\neq$ **`id`**:

```
enq ∘ (front △ deq) == deq ∘ enq ∘ (front △ id)
```

One further point: when we define ι and ε we define a mapping of operators from A to P and viceversa. For example, `enq` is mapped to `Cons`, **`head`** is mapped to `front`, and so on. However, the mapping of operators must be such that $\iota \circ \varepsilon = \mathrm{id}$. More precisely, a concrete type is a type equipped with sufficient observers to get any chunk of payload (we can always program required observers), and we are free to choose constructors and observers for the concrete type in the definition of ε and ι that are not inverses. Formally, however, functions ε and ι extract to and insert from a *concrete algebra given through different signatures*. In the stack example of Section 4.1, we constructed the list using `Nil` and `Cons` but observed it using `snoc` and **`init`**, *i.e.*, as if it had been constructed using `Nil` and `Snoc`, where `snoc` plays the role of **`head`** and **`init`** the role of **`tail`**.

More formally then, let a type be described syntactically by a pair (Σ, E) where $\Sigma = (S, \Omega)$ is a signature consisting of a set of sorts (type names) S and a set of operator names Ω, and E is a set of equations involving operator names in Ω. A signature is a category where $Obj(\Sigma) = S$, $Arr(\Sigma) = \Omega$, and arrow composition is operator composition. A signature morphism $H : \Sigma_1 \to \Sigma_2$ is a functor. The meaning of (Σ, E) can be given by an interpretation function I such that $I(\Sigma, E) = \alpha_X$, where α_X is an initial object in the category of algebras (initial algebra semantics). With this machinery, we note that $\varepsilon :: (\Sigma, E) \to (H_1(\Sigma), \emptyset)$ and $\iota :: (H_2(\Sigma), \emptyset) \to (\Sigma, E)$, where H_1 and H_2 are signature morphisms such that $\iota \circ \varepsilon = \mathrm{id}$ and $I(H_1(\Sigma), \emptyset) = I(H_2(\Sigma), \emptyset)$.

5.3 Choice of P

According to what has been said in the previous section, it makes sense to pick P as the concrete type whose signature has the same functor expression as the ADT. Thus, for the ADT `BBTree a` of balanced binary trees with `a`-data in the leaves (implemented internally as Red-Black Trees, AVL-Trees, or what have you), in order to construct ε and ι it makes sense to pick for P the concrete type of binary trees (which can be unbalanced), *i.e.*, a concrete type with the same functor expression but no equations. We could have chosen P to be lists, but there are many ways to program ι `:: List a` $\to$ `BBTree a` and lists and binary trees do not have the same functor expression in their signature. We would have to provide some sort of conversion from one to the other.

5.4 ADT taxonomy

DEFINITION 5.1 An ADT is *insensitive* when the internal arrangement of payload is independent of any payload property. In other words, the structure of an insensitive ADT is described by the *number* and *position* of payload elements, where by position we mean the location of an element in the *abstract* structure, not in the implementation type. □

Examples of insensitive ADTs are stacks, FIFO queues, double-ended queues, arrays, matrices, etc. Notice that insensitive ADTs may have constrained payload. For instance, a FIFO queue can be constrained to payload with equality only to decide queue equality. Constraints play the same role in insensitive ADTs as in constrained concrete types.

For insensitive ADTs, ϕ does not reshuffle payload: construction is always performed by placing payload at distinguished places as marked by the semantics of constructors (*e.g.*, `enq` enqueues elements at the rear of a FIFO queue). Consequently, the repair condition of Definition 4.1 holds in general and insensitive ADTs are functors if ι and ε can be defined.

DEFINITION 5.2 An ADT is *sensitive* when the internal arrangement of payload depends on a property of the payload. More precisely, there are context-dependent laws such as ordering, lack of ordering, repetition, etc, which affect the position of payload elements and whose conformance imposes a constraint on the payload. What is more, the position of an element may be irrelevant. □

Examples of sensitive ADTs are sets, bags, ordered sets, ordered bags, binary search trees, heaps, priority queues, random queues, hash tables, dictionaries, etc. Sets are constrained on payload with equality not only to decide set equality but to be able to *define* the ADT: sets do not have repeated elements and set membership has to be implemented. (Sets are typical examples of insensitive types where the position of elements, being irrelevant, cannot be used in their characterisation). Ordered sets require an order relation on their payload. The payload of hash tables and dictionaries must be, respectively, hashable and 'keyable'.

For sensitive ADTs, ϕ may reshuffle payload, even remove some. The repair condition need not hold. We hypothesise that the repair condition is broken only in the case of payload removal (Section 5.5).

5.5 Counter-examples

Sensitive ADTs may not be functors. For example, we cannot define ε for ordinary sets: there is no observer that selects an element from an arbitrary set.[1] Similarly for ordinary bags. But even if ε and ι can be defined, the repair condition (Definition 4.1) may not be satisfied. A counter-example is the type of ordered sets of elements that satisfy a property *p*:

[1] This means that Definition 4.1 is stronger than Definition 3.4: a set implemented as a list can satisfy the latter's repair condition.

```
class Prop a where
  p :: a → Bool

data (Prop a, Ord a) ⇒ POrdSet a =  -- abstract
```

Among others, the ADT has the equation:

```
p x == False ⇒ insert x s == s
```

We cannot define the map operator:

```
mapPOrdSet :: (Prop a, Ord a, Prop b, Ord b) ⇒
   (a → b) → POrdSet a → POrdSet b
```

LEMMA 5.1 Let s be a value of type `POrdSet a`. Let f and g be such that $\forall n.p(f(n))$ and $\neg\forall n.p(g(n))$. Suppose we can write ι and ε using a concrete type P. The repair condition:

$$\iota \circ \text{map}_P\ (f \circ g) \circ \varepsilon = \iota \circ \text{map}_P\ f \circ \varepsilon \circ \iota \circ \text{map}_P\ g \circ \varepsilon$$

may not be satisfied. □

PROOF: in the right hand side, after mapping g over P, the rightmost ι only inserts into the set elements from the payload type that satisfy p. Hence, the cardinality of the resulting set may be less than the cardinality of the original set. (The expression $\iota \circ \text{map}_P\ f \circ \varepsilon$ maintains the cardinality.) However, in the left hand side, because $f \circ g$ yields elements that satisfy p (*i.e.*, f 'corrects' g), ι adds to the resulting set all the elements in the payload type, *i.e.*, all the elements in the original set. Hence, the cardinality of the resulting set is always the same as the cardinality of the original set. As a concrete counter-example, take $p(x) = x \geq 0$, $f(x) = x^2$, and $g(x) = -x$.

6 RELATED WORK

The impact of constraints in code reuse was highlighted in [Hug99] in relation to the implementation of abstract types in terms of constrained algebraic types. In [HdM00], a container data type is defined as a relator with membership. The result is presented in an allegorical (not categorical) setting. The counter-example of Section 5.5 suggests f and g must preserve some sort of membership test in order for the repair condition to hold. More work is needed to understand a possible connection. There is another definition of container [AAG05] as a dependent cartesian product $(s \in S) \times P(s)$ where S is a set of shapes and for every $s \in S$, $P(s)$ is a set of positions. The semantics of a container is a functor $F :: \mathbf{C} \to \mathbf{C}$ where $\mathbf{C}$ is the category giving meaning to shapes (*e.g.*, types) and where $F(X) = (s \in S) \times (P(s) \to X)$. That is, F sends a type X to the dependent cartesian product type[2] where the first component s is a shape (higher-order type) and the second component is a function 'labelling' the positions over s with values in X. Containers characterise

[2]The dependent cartesian product type is called 'dependent coproduct'.

concrete types. The authors have extended the container definition to capture some sensitive ADTs with equational laws, such as bags, introducing *quotient containers, i.e.*, containers where there is an equivalence relation on positions [AAGM04]. In Joyal's combinatorial species [Joy86] some insensitive types are described combinatorially in terms of the power series of the arrangements of payload and shapes in an implementation type.

Acknowledgements

The author would like to thank Roland Backhouse, Jeremy Gibbons, Graham Hutton, Henrik Nilsson, and Conor McBride for their comments and bibliographic pointers.

References

[AAG05] Michael Abott, Thorsten Altenkirch, and Neil Ghani. Containers - constructing strictly positive types. *Theoretical Computer Science*, 342:3–27, September 2005.

[AAGM04] Michael Abott, Thorsten Altenkirch, Neil Ghani, and Conor McBride. Constructing polymorphic programs with quotient types. In *7th International Conference on Mathematics of Program Construction (MPC)*, 2004.

[GWM+93] Joseph Goguen, Timothy Winkler, José Meseguer, Kokichi Futatsugi, and Jean-Pierre Jouannaud. Introducing OBJ. In Joseph Goguen, editor, *Applications of Algebraic Specification using OBJ*. Cambridge, 1993.

[HdM00] Paul F. Hoogendijk and Oege de Moor. Container types categorically. *Journal of Functional Programming*, 10(2):191–225, 2000.

[Hug99] John Hughes. Restricted data types in Haskell. Haskell Workshop '99, September 1999.

[Joy86] André Joyal. Foncteurs analytiques et espèces de structures. In G. Labelle and P. Leroux, editors, *Combinatoire énumérative: Proc. of "Colloque de Combinatoire Énumérative", Univ. du Québec à Montréal, 28 May–1 June 1985*, volume 1234 of *Lecture Notes in Mathematics*, pages 126–159. Springer-Verlag, Berlin, 1986.

[LEW96] Jacques Loeckx, Hans-Dieter Ehrich, and Markus Wolf. *Specification of Abstract Data Types*. John Wiley & Sons Ltd and B. G. Teubner, New York, NY, 1996.

[Mar98] Ricardo Peña Marí. *Diseño de Programas, Formalismo y Abstracción*. Prentice-Hall, 1998.

[MFP91] Erik Meijer, Maarten Fokkinga, and Ross Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In J. Hughes, editor, *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture (FPCA'91)*, volume 523 of *LNCS*. Springer-Verlag, 1991.

[Oka98] Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.