When is an abstract data type a functor?

Pablo Nogueira

School of Computer Science and Information Technology
University of Nottingham, UK
pni@cs.nott.ac.uk

Abstract

A parametric algebraic data type is a functor when we can apply a function to its data components while satisfying certain equations. We investigate whether parametric *abstract* data types can be functors. We provide a general definition for their map operation that needs only satisfy one equation. The definability of this map depends on properties of interfaces and is a sufficient condition for functoriality. Instances of the definition for particular abstract types can then be constructed using their axiomatic semantics. The definition and the equation can be adapted to determine, necessarily and sufficiently, whether an ADT is a functor for a given implementation.

1 INTRODUCTION

The application of a function to the data components of a data type has been recognised as a fundamental operation at least since the *maplist* function of LISP [McC78]. A data type that supports the operation while satisfying certain equations is said to be 'mappable' and its formal characterisation is provided by the category-theoretic concept of *functor*. The functor concept is also important because it is preliminary to the concept of *natural transformation* [BW90] which provides a formal characterisation of parametrically polymorphic functions between mappable data types and is central to the notion of parametricity [Wad89].

In functional languages, mappable algebraic data types are recognised as functors. Such types are free algebras: there are no equations among their value constructors and therefore construction and observation (pattern matching) are inverse operations. In contrast, *abstract* data types (ADTs) may not be free algebras and therefore construction and observation may not be inverses. ADTs are defined in terms of ordinary functions and equations, in particular among constructors, that specify the axiomatic semantics of the type [Mit96, Mar98].

We investigate the functoriality of unary parametric ADTs such as queues, stacks, ordered sets, sets, bags, etc. More precisely, we provide a general definition for their map operation that needs only satisfy one equation. The definability of this operation depends on properties of interfaces and is a sufficient condition for functoriality. Instances of the definition for particular abstract types can then be constructed using their axiomatic semantics. The definition and the equation can be adapted to determine, necessarily and sufficiently, whether an ADT is a functor for a given implementation.

2 CATEGORIES AND FUNCTORS

A category **C** is an algebraic structure consisting of a collection $\mathrm{Obj}(\mathbf{C})$ of *objects* (entities with structure) and a collection $\mathrm{Arr}(\mathbf{C})$ of *arrows* (structure-preserving maps between objects) such that there is a binary arrow-composition operator, written $\circ$, that is closed (yields an arrow in $\mathrm{Arr}(\mathbf{C})$), is associative, and has unique left and right neutral element (the identity arrow) for every object. Every arrow has only one *source* and one *target* object. We write $f :: a \to b$ to denote the arrow f with source a and target b, and write $id_a :: a \to a$ to denote the unique identity arrow for a.

A *functor* $F :: \mathbf{C} \to \mathbf{D}$ maps all objects and arrows in category **C** respectively to objects and arrows in category **D** while preserving the categorical structure.[1] More precisely, F consists of a pair of total maps $F :: \mathrm{Obj}(\mathbf{C}) \to \mathrm{Obj}(\mathbf{D})$ (the object-level map) and $F :: \mathrm{Arr}(\mathbf{C}) \to \mathrm{Arr}(\mathbf{D})$ (the arrow-level map) such that source, target, composition, and identities are preserved. Given an arrow $f :: a \to b$ we have $Ff :: Fa \to Fb$ and F distributes over composition and preserves identities:

$$\begin{array}{lcl} F(g \circ f) & = & Fg \circ Ff \\ Fid_a & = & id_{Fa} \end{array}$$

We refer to the equations as *functorial laws*. Notice that the symbol F is heavily overloaded. A functor $F :: \mathbf{C} \to \mathbf{C}$ is called an *endofunctor*.

3 CATEGORIES AND TYPES

A connection between functional programming (*e.g.*, Haskell) and category theory is the category **Type** where, broadly, objects are monomorphic types and arrows are functional programs involving those types. Arrow composition is function composition (which is closed and associative) and identity arrows are instances of the polymorphic identity function for monomorphic types [BW90]. A polymorphic function $f :: a \to b$ is a collection of arrows of $\mathrm{Arr}(\mathbf{Type})$, but we informally refer to f as 'an' arrow.

A unary parametrically polymorphic algebraic data type F is an endofunctor $F :: \mathbf{Type} \to \mathbf{Type}$ when at the object level F maps monomorphic types to monomorphic types and at the arrow level *mapF* satisfies the functorial laws. The categorical convention is to write F for *mapF*. For example, the list type is a typical functor. At the object level, the type constructor [] maps monomorphic types (*e.g.*, *Bool*) to monomorphic types (*e.g.*, [*Bool*]). At the arrow level, the list function *map* satisfies the functorial laws. The categorical convention is to write *List* for *map*.

There are two important points. First, *map* respects the structure or *shape* of the list, mapping only the data or *payload*—from now on we refer to the type-argument

[1] We use a double colon when writing the source and target categories because functors are also arrows in functor categories [BW90].

of a data type as its payload. Second, *map*'s definition follows the structure of the list type and the proof that it satisfies the functorial laws proceeds by structural induction on lists [Mit96].

In this paper we are only concerned with unary first-order data types. Mappable algebraic data types of arbitrary arity and order can be recognised as functors with the help of other categorical devices such as product and functor categories [BW90].

We conclude the section with a non-functor example: $Fix\ a = (a \to a) \to a$ is the type of the fixed-point combinator $fix\ f = f\ (fix\ f)$. It is not possible to write a map function for *Fix* that satisfies the functorial laws. Unfortunately, proving this formally is beyond the scope of this paper [BW90].

3.1 Subcategories and type classes

In Haskell, algebraic data types can be constrained in their type-argument range by type-class membership. For example:

```
data Ord a => Tree a = Empty | Node a (Tree a) (Tree a)
```

This is the type of binary trees whose nodes contain values of types in class *Ord*. Constrained data types are often used in the implementation of ADTs. For example, *Tree* can be used in the implementation of binary search trees, ordered bags, ordered sets, priority queues, etc [Oka98].

A *subcategory* **S** of a category **C** is such that $\mathrm{Obj}(\mathbf{S}) \subseteq \mathrm{Obj}(\mathbf{C})$, $\mathrm{Arr}(\mathbf{S})$ contains all the arrows in $\mathrm{Arr}(\mathbf{C})$ involving only **S**-objects, and composition and identities in $\mathrm{Arr}(\mathbf{S})$ are those in $\mathrm{Arr}(\mathbf{C})$ involving only **S**-objects [BW90].

Mappable constrained data types are functors $F :: \mathbf{S} \to \mathbf{Type}$, where **S** is a subcategory of **Type**. For example, in **Ord** objects are types in *Ord* and arrows are functions on those objects. An overloaded function $f :: (Ord\ a, Ord\ b) \Rightarrow a \to b$ is a collection of arrows of $\mathrm{Arr}(\mathbf{Ord})$, but we informally refer to f as 'an' arrow.

3.2 Algebras, Coalgebras, and Bialgebras

Free algebras are neatly characterised as the least fixed point of the functor expression determined by their constructors [MFP91]. To illustrate this, we need to introduce $\triangledown$, the case eliminator for sums, and $\triangle$, the lifted pair constructor (we occasionally deviate from Haskell notation and write $a \times b$ for the product type and 1 for the unit type):

```
▽ :: (a → c) → (b → c) → a + b → c
(f ▽ g) (Inl x) = f x
(f ▽ g) (Inr y) = g x
△ :: (c → a) → (c → b) → c → a × b
(f △ g) x = (f x, g x)
```

Take the list type, for example. Let us write the type signatures of list constructors slightly differently as $[\,] :: 1 \to [a]$ and $(:) :: a \times [a] \to [a]$. The 'nil' constructor is

lifted to a function from the unit type and the 'cons' constructor is uncurried. The list type $[a]$ is characterised as the least fixed point of the functor $Fx = 1 + a \times x$ determined by the list constructors: $([\,]\triangledown(:)) :: F[a] \to [a]$.

In a category **C**, the pair consisting of an object X and an arrow $\alpha :: FX \to X$ is called an *F-algebra*. The pair consisting of an object Y and an arrow $\beta :: Y \to FY$ is called an *F-coalgebra*. In **Type**, the pair consisting of the algebraic data type Pa and the arrow $\alpha :: F(Pa) \to Pa$ is an F-algebra. Because Pa is free, observation can be characterised either by the inverse F-coalgebra $\alpha^{-1} :: Pa \to F(Pa)$ which formalises pattern matching or, alternatively, by the G-coalgebra $\beta :: Pa \to G(Pa)$, where $\alpha^{-1} = cond \circ \beta$ and $\beta^{-1} = \alpha \circ cond$. Function $cond :: Gx \to Fx$ formalises discriminated (guarded) selection [MFP91]. The pair (α, β) is called an *FG-bialgebra*. For example, in the case of lists we have:

$$
\begin{array}{lcl}
Gx & = & Bool \times a \times x \\
\beta & = & (null \triangle head \triangle tail) \\
cond & :: & (Bool \times a \times x) \to (1 + a \times x) \\
cond(p,t,e) & = & if\ p\ then\ Inl\ ()\ else\ Inr\ (t,e)
\end{array}
$$

The following equation follows from *cond*'s definition:

$$((f \triangledown g) \circ cond \circ (h \triangle i \triangle j))\ x\ =\ if\ h\ x\ then\ f()\ else\ g(i\ x,\ j\ x)$$

4 ABSTRACT DATA TYPES

ADTs are typically specified in terms of function interfaces. In Haskell, the type-class system is employed for this purpose. For example, below type class *OrdSet* specifies the interface of ordered sets and type class *Queue* the interface of FIFO queues:[2]

```
class OrdSet s where
  emptyS    :: Ord a => s a
  insert    :: Ord a => a -> s a -> s a
  isEmptyS  :: Ord a => s a -> Bool
  min       :: Ord a => s a -> a
  remove    :: Ord a => a -> s a -> s a
  member    :: Ord a => a -> s a -> Bool
```

```
class Queue q where
  emptyQ    :: q a
  enq       :: a -> q a -> q a
  isEmptyQ  :: q a -> Bool
  front     :: q a -> a
  deq       :: q a -> q a
```

The nullary constructors *emptyS* and *emptyQ* return respectively an empty ordered set and an empty queue. Constructor *insert* inserts an element into a given ordered set whereas *enq* enqueues a value at the rear of a queue. Functions *isEmptyS* and *isEmptyQ* discriminate respectively whether an ordered set or a queue is empty. Function *min* returns the minimum element in an ordered set and *front* returns the front element in a queue. Function *deq* dequeues the front element in a queue and *remove* removes a specified element from an ordered set. Finally, *member* performs a membership test.

[2] Strictly speaking, *OrdSet* and *Queue* are so-called *constructor* classes.

ADTs are not formally defined by interfaces alone but also by a set of (conditional) equations among their operators that specify the axiomatic semantics of the type [Mit96, Mar98]. There may be equations among constructors, in which case construction and observation are not inverses (*e.g.*, queue values are inserted at the rear of the queue but selected from the front). Equations are necessary to *implement* and *reason about* ADTs. The following is a sample of *Queue*'s and *OrdSet*'s equations:

$$
\begin{aligned}
& (\mathit{isEmptyQ}\ q) \Rightarrow \mathit{deq}\ (\mathit{enq}\ x\ q) = q \\
\neg\ & (\mathit{isEmptyQ}\ q) \Rightarrow \mathit{deq}\ (\mathit{enq}\ x\ q) = \mathit{enq}\ x\ (\mathit{deq}\ q) \\
& \mathit{member}\ x\ s \Rightarrow \mathit{insert}\ x\ s = s
\end{aligned}
$$

Programmers supply implementations by providing an implementation type and definitions for the operators that satisfy the equations. For example, ordered sets can be implemented using lists. Below, the standard functions *sort* and *nub* respectively sort and remove duplicates from a list:

```
instance OrdSet [] where
  emptyS     =  []
  min        =  head
  insert x   =  sort ∘ nub ∘ (x:)
  ...
```

An abstract type can be characterised by an FG-bialgebra (γ, ν) for some functors F and G. In the case of queues, for example, we have $\gamma = (\mathit{emptyQ} \triangledown \mathit{enq})$ and $\nu = (\mathit{isEmptyQ} \triangle \mathit{front} \triangle \mathit{deq})$, where F and G are the same functors of the list bialgebra of Section 3.2. For queues, however, the equations $\gamma^{-1} = \mathit{cond} \circ \nu$ and $\nu^{-1} = \gamma \circ \mathit{cond}$ do not hold.

4.1 Payload, clutter, representation invariant, and repair function

ADTs arise because programmers cannot express their imagined types as free algebras. The values of an ADT *Aa* are represented (simulated) by a subset of the values of an algebraic data type *Ia*. The subset is characterised formally by a *representation invariant* [Hoa72], that is, a predicate $\mathit{rep} :: \mathit{Ia} \rightarrow \mathit{Bool}$ such that $\mathit{rep}(i)$ holds iff the value i of type *Ia* represents a value of *Aa*. Interface operators maintain the equations of the ADT and maintain the representation invariant by implementing a conceptual function $\phi :: \mathit{Ia} \rightarrow \mathit{Ia}$ such that $\forall i.\mathrm{rep}(\phi(i))$. We call ϕ a *repair function* because it re-establishes the representation invariant.

The type *Ia* contains payload data and may also contain *clutter*, *i.e.*, extra data used for efficiency or structuring purposes. Clutter data is either of a fixed monomorphic type or is parametric on the payload type (otherwise, *Ia* would have to be parametric on the same types as the clutter). Examples of clutter are the colour of nodes in Red-Black Trees (monomorphic), the height of a sub-tree in a heap (monomorphic), the cached front list in a physicists' implementation of FIFO queues (payload-parametric), etc [Oka98]. Clutter is more the norm than the exception: data (space) will be used to improve operator speed (time).

5 ABSTRACT DATA TYPES AND FUNCTORS

In order to establish the connection between ADTs and functors we need to define source and target categories and the object-level and arrow-level mappings that satisfy the functorial laws. To identify the categories, we classify ADTs into *sensitive* or *insensitive* depending on whether the internal arrangement of payload elements is, respectively, dependent or independent of properties of the payload type. We assume properties of the payload type are expressed via type-class constraints (*e.g.*, the payload type has equality if it is an instance of *Eq*).

Examples of insensitive ADTs are FIFO queues, double-ended queues, arrays, matrices, etc. The order of insertion is determined by the operators (*e.g.*, *enq* enqueues payload at the rear), and the ADTs can be characterised by the number and logical position of payload elements in the abstract structure.

Examples of sensitive ADTs are sets, bags, ordered sets, ordered bags, binary search trees, heaps, priority queues, hash tables, dictionaries, etc. In a set, payload position is irrelevant and cannot be used in the characterisation of the type. Sets require payload types in *Eq* to define the membership operator. Ordered sets require payload in *Ord*, etc.

We recognise mappable ADTs as functors $F :: \mathbf{S} \rightarrow \mathbf{ADT}$. The source category **S** is a subcategory of **Type** to account for constraints. There are ADTs for which we cannot characterise **S**. We postpone this discussion until Section 6.2 and concentrate for the moment on ADTs where the characterisation is possible. We cannot take **Type** as the target category for the reasons given in Section 4.1. Instead, **ADT** is the category whose objects are monomorphic ADTs (*e.g.*, *Queue Char*, *OrdSet Int*) and arrows are functions on those ADTs. The category can be formalised in various ways which make use of representation invariants or equational laws to pin down the type [BW90, Fok96].

The arrow-level part of the functor is the ADT's map operation that satisfies the functorial laws. To the programmer, the operation can be defined *internally* for the implementation type (map is part of the interface), or *externally*, in terms of interface operators. In the internal approach, the operation must satisfy the functorial laws and preserve representation invariants. If this is the case, it can only be claimed that *A* implemented as *I* is a functor. The external approach is independent of implementations, but there are several obstacles. First, programmers have to find a definition that satisfies the functorial laws and the equations of the ADT. Second, structural induction cannot be deployed in proofs. Take for example $OrdSet :: \mathbf{Ord} \rightarrow \mathbf{ADT}$.[3] The following is a sensible definition for its map operation:

```
ordSet :: (Ord a, Ord b, OrdSet s) ⇒ (a → b) → s a → s b
ordSet f s = if isEmptyS s then emptyS else
                let (m,r) = (min s,remove m s) in insert (f m) (ordSet f r)
```

[3]Do not confuse *OrdSet* with a functor $F :: \mathbf{Poset} \rightarrow \mathbf{Poset}$, where **Poset** is the category of partially ordered sets [BW90].

Haskell functions must start with lowercase letters so we use *ordSet* instead of *OrdSet* as the name for the map. Function *ordSet* must satisfy the functorial laws and the equations of the ADT. For example, the following equation must hold:

$$\mathit{member}\ x\ s\ \Rightarrow\ \mathit{ordSet}\ f\ (\mathit{insert}\ x\ s)\ =\ \mathit{ordSet}\ f\ s$$

Because of this equation, the map operation does not respect the shape of the ordered set. An expression such as $\mathit{ordSet}\ (\mathit{const}\ 0)\ \{1,2,3\}$ must yield the value $\{0\}$. We should not be deceived into thinking that, for this reason, *ordSet* is not a valid map operation. The functoriality condition is that *ordSet* satisfies the functorial laws, and this may be the case regardless of whether it satisfies extra equations. Unfortunately, we cannot deploy structural induction in proofs. For example, given a non-empty ordered set *insert x s*, function *ordSet* is defined in terms of observers *min* and *remove*, with the recursive call involving *m* and *r*, not *x* and *s*. To deploy induction we have to prove intermediate lemmas such as $\neg(\mathit{isEmptyS}\ s) \wedge \neg(\mathit{member}\ m\ r) \Rightarrow \mathit{insert}\ m\ r = s$.

6 FUNCTORIAL LAWS FOR ADTS

In this section we provide a general definition for the map operation of ADTs whose interfaces satisfy certain properties made precise below. The definition needs only satisfy one equation that makes use of the notion of payload and repair function. Instances of the definition for particular ADTs can be constructed using their axiomatic semantics. The definability of the operation is a sufficient condition for functoriality (Section 6.2).

Let *Aa* be an abstract data type and *Pa* be an algebraic data type. Let us assume that *P* is a functor and that it is possible to define an *extraction* function $\varepsilon :: Aa \to Pa$ and an *insertion* function $\iota :: Pa \to Aa$ such that $\iota \circ \varepsilon = id_{Aa}$. The intuition is that ε ignores clutter and collects the payload in *Pa* maintaining the logical positioning, whereas ι enforces the equations of the ADT for the given payload and sets up the clutter. Because *A* satisfies equations, ε may not be surjective and ι may not be injective. Consequently, $\varepsilon \circ \iota$ need not be the identity for *Pa*. (For example, we may not get the same list when inserting the elements into an ordered set and then extracting them back to a list.) Notice that $\varepsilon \circ \iota$ corresponds to $\phi_P :: Pa \to Pa$, the behaviour of the repair function ϕ on the payload. (For example, the resulting list will be ordered and without repetitions.)

Assuming the above premises, we can write a definition for the map operation, namely, $A f = \iota \circ P f \circ \varepsilon$. It is easy to prove that it preserves identities:

$$\begin{array}{ll}
 & A\ id_a \\
= & \quad \{\ \text{def. of } A\ \} \\
 & \iota \circ P\ id_a \circ \varepsilon \\
= & \quad \{\ P \text{ functor}\ \} \\
 & \iota \circ\ id_{Pa} \circ \varepsilon \\
= & \quad \{\ id_{Pa} \circ \varepsilon\ =\ \varepsilon\ \}
\end{array}$$

$$
\begin{array}{ll}
 & \iota \circ \varepsilon \\
= & \quad \{ \ \iota \circ \varepsilon = id_{Aa} \ \} \\
 & id_{Aa}
\end{array}
$$

We obtain a functorial law when attempting to prove that the map distributes over composition:

$$
\begin{array}{ll}
 & A\,(g \circ f) \ = \ Ag \circ Af \\
= & \quad \{ \ \text{def. of } A \ \} \\
 & \iota \circ P\,(g \circ f) \circ \varepsilon \ = \ \iota \circ Pg \circ \varepsilon \circ \iota \circ Pf \circ \varepsilon \\
= & \quad \{ \ P \text{ functor and } \varepsilon \circ \iota = \phi_P \ \} \\
 & \iota \circ Pg \circ Pf \circ \varepsilon \ = \ \iota \circ Pg \circ \phi_P \circ Pf \circ \varepsilon
\end{array}
$$

We call the last equation a *repair law*. The key element is the presence of the intermediate ϕ_P. Read from left to right, the law states that extracting payload, then distributing P over the composition, and finally inserting back payload must yield the same value as extracting payload, distributing P over the composition with an intermediate repair, and finally inserting payload. Due to the presence of ϕ_P, the equations $Af \circ \iota = \iota \circ Pf$ and $\varepsilon \circ Ag = Pg \circ \varepsilon$ do not hold (expand Af by its definition).[4]

Let Aa be characterised by the FG-bialgebra (γ, ν) where $\gamma :: F(Aa) \to Aa$ and $\nu :: Aa \to G(Aa)$. Definitions for ι and ε can be constructed for a Pa characterised by the FG-bialgebra (α, β), where $\alpha :: F(Pa) \to Pa$ and $\beta :: Pa \to G(Pa)$, as indicated by the following diagram:

$$
\begin{tikzcd}
F(Pa) \arrow[r, "\alpha"] \arrow[d, shift right, "F\iota"'] & Pa \arrow[r, "\beta"] \arrow[d, shift right, "\iota"'] & G(Pa) \arrow[d, shift right, "G\iota"'] \\
F(Aa) \arrow[r, "\gamma"'] \arrow[u, shift right, "F\varepsilon"'] & Aa \arrow[r, "\nu"'] \arrow[u, shift right, "\varepsilon"'] & G(Aa) \arrow[u, shift right, "G\varepsilon"']
\end{tikzcd}
$$

Because F and G are functors, we have:

$$
\begin{array}{lcl}
\iota \circ \alpha & = & \gamma \circ F\iota \\
\beta \circ \varepsilon & = & G\varepsilon \circ \nu
\end{array}
$$

From these equations we can obtain definitions for ι and ε using the fact that Pa is a free algebra:

$$
\begin{array}{lcl}
\iota & = & \gamma \circ F\iota \circ cond \circ \beta \\
\varepsilon & = & \alpha \circ cond \circ G\varepsilon \circ \nu
\end{array}
$$

The definitions must also satisfy $\iota \circ \varepsilon = id_{Aa}$. Unfortunately, this is not the case when $\alpha^{-1} = cond \circ \beta$ and $\beta^{-1} = \alpha \circ cond$ because it leads to $\gamma \circ cond \circ \nu = id_{Aa}$ which does not hold in general (Section 4).

[4] In other words, ε and ι are not natural transformations.

Notice however that α does not occur in ι's definition and β does not occur in ε's definition. Thus, for *insensitive* ADTs we can use for *Pa* an FG-bialgebra (α, β') for extraction and another (α', β) for insertion where α and β' are inverses and so are α' and β.

Formally, the pair (α', β') is to be obtained from (γ, ν) by an FG-bialgebra homomorphism $\Sigma(\gamma, \nu) = (\alpha', \beta')$. The intuition is that the operators in (α', β') are those that make P behave like A or, in other words, P is *viewed* as an A. In practice, the programmer is to find (α', β') by searching for (or programming) *Pa* operators that satisfy the same equations as those operators in (γ, ν). Then, α and β are defined as the inverses of β' and α', that is, $\alpha = \mathit{cond} \circ \beta'$ and $\beta = \alpha' \circ \mathit{cond}$.

For example, let A be *Queue*. We have $Fx = 1 + a \times x$, $Gx = \mathit{Bool} \times a \times x$, and $(\gamma, \nu) = (\mathit{emptyQ} \triangledown \mathit{enq},\ \mathit{isEmptyQ} \triangle \mathit{front} \triangle \mathit{deq})$. The obvious choice of P is the list type. We have the following:

$$
\begin{array}{lcl}
(\alpha', \beta') & = & ([\,] \triangledown \mathit{snoc},\ \mathit{null} \triangle \mathit{head} \triangle \mathit{tail}) \\
(\alpha, \beta) & = & ([\,] \triangledown (:),\ \mathit{null} \triangle \mathit{last} \triangle \mathit{init})
\end{array}
$$

(α', β') views lists as queues. Function $\mathit{snoc} :: a \times [a] \to [a]$ is the dual of 'cons'. It inserts an element at the rear of a list, *e.g.*, $\mathit{snoc}\ x\ xs = xs {+\!\!+} [x]$.

Knowing β, we can obtain the definition of $\iota :: \mathit{Queue}\ q \Rightarrow [a] \to q\ a$ from the equation $\iota = \gamma \circ F\iota \circ \mathit{cond} \circ \beta$:

$$
\begin{array}{ll}
 & \iota\, x \\
= & \quad \{ \text{ def. of } \iota \ \} \\
 & \gamma\,(F\iota\,(\mathit{cond}\,(\beta\, x))) \\
= & \quad \{ \text{ def. of } \beta \ \} \\
 & \gamma\,(F\iota\,(\mathit{cond}\,(\mathit{null}\, x,\ \mathit{last}\, x,\ \mathit{init}\, x))) \\
= & \quad \{ \text{ def. of } \mathit{cond} \ \} \\
 & \gamma\,(F\iota\,(\mathit{if}\ \mathit{null}\, x\ \mathit{then}\ \mathit{Inl}\ ()\ \mathit{else}\ \mathit{Inr}\ (\mathit{last}\, x,\ \mathit{init}\, x))) \\
= & \quad \{ \ F \text{ functor } \} \\
 & \gamma\,(\mathit{if}\ \mathit{null}\, x\ \mathit{then}\ \mathit{Inl}\ ()\ \mathit{else}\ \mathit{Inr}\ (\mathit{last}\, x,\ \iota\,(\mathit{init}\, x))) \\
= & \quad \{ \text{ def. of } \gamma \ \} \\
 & \mathit{if}\ \mathit{null}\, x\ \mathit{then}\ \mathit{emptyQ}\ ()\ \mathit{else}\ \mathit{enq}\ (\mathit{last}\, x,\ \iota\,(\mathit{init}\, x)) \\
= & \quad \{ \text{ isomorphism } \mathit{emptyQ}\ () \simeq \mathit{emptyQ} \text{ and curry } \mathit{enq} \ \} \\
 & \mathit{if}\ \mathit{null}\, x\ \mathit{then}\ \mathit{emptyQ}\ \mathit{else}\ \mathit{enq}\ (\mathit{last}\, x)\ (\iota\,(\mathit{init}\, x))
\end{array}
$$

Knowing α, we can obtain the definition of $\varepsilon :: \mathit{Queue}\ q \Rightarrow q\ a \to [a]$ from the equation $\varepsilon = \alpha \circ \mathit{cond} \circ G\varepsilon \circ \nu$:

$$
\begin{array}{ll}
 & \varepsilon\, x \\
= & \quad \{ \text{ def. of } \varepsilon \ \} \\
 & \alpha\,(\mathit{cond}\,(G\varepsilon\,(\nu\, x))) \\
= & \quad \{ \text{ def. of } \nu \ \} \\
 & \alpha\,(\mathit{cond}\,(G\varepsilon\,(\mathit{isEmptyQ}\, x,\ \mathit{front}\, x,\ \mathit{deq}\, x))) \\
= & \quad \{ \ G \text{ functor } \} \\
 & \alpha\,(\mathit{cond}\,(\mathit{isEmptyQ}\, x,\ \mathit{front}\, x,\ \varepsilon\,(\mathit{deq}\, x)))
\end{array}
$$

$$
\begin{array}{ll}
= & \quad \{ \text{ def. of } cond \ \} \\
 & \alpha\ (\mathit{if}\ \mathit{isEmptyQ}\ x\ \mathit{then}\ \mathit{Inl}\ ()\ \mathit{else}\ \mathit{Inr}\ (\mathit{front}\ x,\ \varepsilon\ (\mathit{deq}\ x))) \\
= & \quad \{ \text{ def. of } \alpha \ \} \\
 & \mathit{if}\ \mathit{isEmptyQ}\ x\ \mathit{then}\ [\,]\ ()\ \mathit{else}\ (:)\ (\mathit{front}\ x,\ \varepsilon\ (\mathit{deq}\ x)) \\
= & \quad \{ \text{ isomorphism } [\,]\ () \simeq [\,] \text{ and curry } (:) \ \} \\
 & \mathit{if}\ \mathit{isEmptyQ}\ x\ \mathit{then}\ [\,]\ \mathit{else}\ (\mathit{front}\ x)\ :\ \varepsilon\ (\mathit{deq}\ x)
\end{array}
$$

Other list operators are to be used in the case of stacks. More precisely, $(\alpha', \beta') = ([\,] \triangledown (:),\ \mathit{null} \vartriangle \mathit{head} \vartriangle \mathit{tail})$ and $(\alpha, \beta) = ([\,] \triangledown (:),\ \mathit{null} \vartriangle \mathit{head} \vartriangle \mathit{tail})$:

$$
\begin{array}{l}
\iota \ :: \ \mathit{Stack}\ s \Rightarrow [a] \rightarrow s\ a \\
\iota\ x = \mathit{if}\ \mathit{null}\ x\ \mathit{then}\ \mathit{emptyStack}\ \mathit{else}\ \mathit{push}\ (\mathit{head}\ x)\ (\iota\ (\mathit{tail}\ x)) \\
\varepsilon \ :: \ \mathit{Stack}\ s \Rightarrow s\ a \rightarrow [a] \\
\varepsilon\ x = \mathit{if}\ \mathit{isEmptyStack}\ x\ \mathit{then}\ [\,]\ \mathit{else}\ (\mathit{top}\ x)\ :\ \varepsilon\ (\mathit{pop}\ x)
\end{array}
$$

For *insensitive* ADTs, we can use the same FG-bialgebra for Pa for insertion and extraction because the observers in ν place payload in Pa at fixed positions but the constructors in γ place them according to the semantics of the type. For example, payload from an ordered set is extracted into an ordered list, but an arbitrary list is inserted into an ordered set. Thus, we can have:

$$
\begin{array}{lcl}
(\gamma, \nu) & = & (\mathit{emptyS} \triangledown \mathit{insert},\ \mathit{isEmptyS} \vartriangle \mathit{min} \vartriangle (\lambda s \rightarrow \mathit{remove}\ (\mathit{min}\ s)\ s)) \\
(\alpha, \beta) & = & ([\,] \triangledown (:),\ \mathit{null} \vartriangle \mathit{head} \vartriangle \mathit{tail})
\end{array}
$$

and the following definitions for ι and ε are obtained from their respective equations:

$$
\begin{array}{l}
\iota \ :: \ (\mathit{OrdSet}\ s,\ \mathit{Ord}\ a) \Rightarrow [a] \rightarrow s\ a \\
\iota\ x = \mathit{if}\ \mathit{null}\ x\ \mathit{then}\ \mathit{emptyS}\ \mathit{else}\ \mathit{insert}\ (\mathit{head}\ x)\ (\iota\ (\mathit{tail}\ x)) \\
\varepsilon \ :: \ (\mathit{OrdSet}\ s,\ \mathit{Ord}\ a) \Rightarrow s\ a \rightarrow [a] \\
\varepsilon\ x = \mathit{if}\ \mathit{isEmptyS}\ x\ \mathit{then}\ [\,]\ \mathit{else}\ (\mathit{min}\ x)\ :\ \varepsilon\ (\mathit{remove}\ (\mathit{min}\ x)\ x)
\end{array}
$$

Alternatively, we could have used the same (α, β) we used for queues and still satisfy $\iota \circ \varepsilon = id_{Aa}$.

The reader can verify that the repair law is satisfied and FIFO queues, stacks, and ordered sets are functors. In insensitive ADTs, ϕ_P does not reshuffle payload (constructors place payload in fixed positions). We conjecture that the repair law is always satisfied and therefore insensitive ADTs are functors. Other characterisations of insensitive ADTs also recognise them as functors (Section 8). In sensitive ADTs, ϕ_P may reshuffle payload, even remove some. The repair law need not hold. We conjecture that the repair law is broken only in a particular case of payload removal (Section 6.2).

6.1 Functorial laws for ADT implementations

The definition of Af can be adapted to define If, the map operation for the implementation type, where now $\varepsilon :: Ia \rightarrow Pa$ and $\iota :: Pa \rightarrow Ia$. Let us define $If = \phi \circ \sigma f$,

where $\phi = \iota$ and $\sigma f = P f \circ \varepsilon$. The repair law is now written:

$$\phi \circ \sigma\,(g \circ f) \;=\; \phi \circ \sigma g \circ \phi \circ \sigma f$$

However, $I f$ can be defined directly without using interface operators; that is, we need not be concerned with functors F and G, and we can define ϕ and σ directly in terms of the implementation type, not in terms of ι and ε.

More precisely, we can define Pa as the type in Ia that holds the payload. Function ϕ takes a value of Pa and creates a value of Ia by enforcing ϕ_P and creating the clutter. Function $\sigma :: (a \to b) \to Ia \to Pb$ is a 'selection' function that ignores clutter and maps over the payload.

Consider FIFO queues, for example. The so-called physicists' representation is a reasonably efficient way of implementing queues [Oka98]:

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

The implementation type consists of three lists and two integers. The first integer is the length of the second list, which contains the front elements, and the second integer is the length of the third list, which contains the rear elements in reverse order. Elements are moved from the rear list to the front list when the length of the former is less than the length of the latter. In a lazy language the move is computed on demand so a prefix of the front list (10 elements, say) is cached for efficiency. The first list is such prefix.

The second and third lists make up the payload type. The cached front list and the lengths are clutter. We have:

```
type P a = ([a],[a])
σ :: (a → b) → PhysicistQueue a → P b
σ f (PQ _ _ fs _ rs) = (map f fs , map f rs)
ϕ :: P a → PhysicistQueue a
ϕ (fs,rs) = PQ (init 10 fs) (length fs) fs (length rs) rs
queue :: (a → b) → PhysicistQueue a → PhysicistQueue a
queue f = ϕ ∘ σ f
```

Notice that by fusing ϕ and σ we obtain the definition for *queue* that the programmer would have written. The repair law can be proved directly from the fact that the list type is a functor:

$$\begin{array}{ll}
 & \phi\,(\sigma g\,(\phi\,(\sigma f\,(PQ\ cs\ lf\ fs\ lr\ rs)))) \\
= & \quad \{\ \text{def. of } \sigma\ \} \\
 & \phi\,(\sigma g\,(\phi\,(map\ f\ fs\ ,\ map\ f\ rs))) \\
= & \quad \{\ \text{def. of } \phi \text{ where } mf = map\ f\ fs \text{ and } mr = map\ f\ rs\ \} \\
 & \phi\,(\sigma g\,(PQ\,(init\ 10\ mf)\ mf\ (length\ mf)\ mr\ (length\ mr))) \\
= & \quad \{\ \text{def. of } \sigma\ \} \\
 & \phi\,(map\ g\ mf,\ map\ g\ mr) \\
= & \quad \{\ \text{def. of } mf \text{ and } mr, \text{ and list is functor}\ \} \\
 & \phi\,(map\ (g \circ f)\ fs,\ map\ (g \circ f)\ rs)
\end{array}$$

$$
\begin{aligned}
= \quad & \{ \text{ def. of } \sigma \ \} \\
& \phi\ (\sigma\ (g \circ f)\ (PQ\ cs\ lf\ fs\ lr\ rs))
\end{aligned}
$$

Now consider ordered sets. Suppose they are implemented in terms of lists:

```
type SetList a = [a]
```

There is no clutter in this implementation and the payload type is $Pa = [a]$:

```
σ :: (Ord a, Ord b) ⇒ (a → b) → SetList a → P b
σ = map
ϕ :: Ord a ⇒ P a → SetList a
ϕ = sort ∘ nub
set :: (Ord a, Ord b) ⇒ (a → b) → SetList a → SetList b
set f = ϕ ∘ σ f
```

Fusing ϕ and σ gives the definition the programmer would have written. The repair law is:

```
sort ∘ nub ∘ map (g ∘ f) = sort ∘ nub ∘ map g ∘ sort ∘ nub ∘ map f
```

which can be proved by induction on lists.

6.2 Non-functors

The definition of Af and its repair law are conditional on the definability of ι and ε such that $\iota \circ \varepsilon = id_{Aa}$. The definability of Af that satisfies the repair law is therefore a sufficient condition for functoriality in our source and target categories.

There are examples of ADTs for which either ι or ε are not definable but whose functoriality can be established internally for particular implementations (Section 6.1). Examples are ordinary (unordered) sets and bags where membership test, cardinality, and number of element occurrences in bags are the only observers. The definability of ε is conditional upon the existence of operators that enable the observation of everything that is constructed. In the case of sets, $\gamma = (emptyS \ \triangledown\ insert)$ where $insert :: Set\ s \Rightarrow a \times s\ a \to s\ a$, but we cannot find $\nu = (isEmptyS \triangle x \triangle y)$ such that $(x \triangle y) :: Set\ s \Rightarrow s\ a \to a \times s\ a$. However, if sets are implemented as lists then $\phi = nub$, $\sigma = map$, and the repair law of Section 6.1 is satisfied.

In general, if an ADT is a functor for a given implementation type then it should be a functor for alternative implementation types. Otherwise, the implementation types would not implement the same ADT. More work is needed to formalise this claim. For example, let k be a type of keys and v be a type of keyed values. Association lists $[(k,v)]$ and finite functions $k \to v$ can be used to implement dictionaries. However, with finite functions we cannot define the empty dictionary, nor remove a key-value pair from a dictionary, and remain total.

We have mentioned in Section 5 that it may not be possible to identify a mappable ADT with a functor $F :: \mathbf{S} \to \mathbf{C}$ where $\mathrm{Obj}(\mathbf{S})$ are monomorphic Haskell types. For example, let *OrdSetP* be the type of ordered sets with positive numbers.

The equation $x < 0 \Rightarrow \mathit{insert}\ x\ s = s$ is part of its axiomatic semantics. Type class *Num* contains types *Int*, *Integer*, *Float*, and *Double*, but we cannot define the class of positive numbers as a subclass of *Num*. In particular, we cannot define the type *Nat* of positive integers as a subtype of *Int*. At most we can define an algebraic data type for natural numbers, *e.g.*, $\mathit{data\ Nat} = \mathit{Zero} \mid \mathit{Succ\ Nat}$, and provide mappings $\mathit{in} :: \mathit{Int} \to \mathit{Nat}$ and $\mathit{out} :: \mathit{Nat} \to \mathit{Int}$. This is the idea behind *views* for ADTs [Wad87], a deprecated approach with serious drawbacks [PPN96]. (Notice the similarity with Section 4.1, where instead of *rep* and types I and A we have a property and type classes S and *Num*. We might define **S** as a subcategory of **ADT**.)

Despite the limitation, we can use the repair law to disprove that *OrdSetP* is a functor from **Num** to **ADT**. A counter-example is provided by $g(x) = x^2$ and $f(x) = -x$. Notice that g is not an arrow of $\mathrm{Arr}(\mathbf{S})$. The negative payload is removed by ϕ_P in $Pg \circ \phi_P \circ Pf$ whereas in $Pg \circ Pf$, function g 'corrects' f and the negative payload becomes positive. The resulting sets may have different cardinality. A particular way of determining whether the repair law is broken is to look for counter-examples in which the two sides of the equation produce ADTs with different sizes.

7 CONCLUSION

We have investigated the connection between first-order parametric ADTs and functors. At the object level, ADTs map objects in subcategories of the category of types to objects in the category of monomorphic ADTs. At the arrow-level we have presented a general definition for their map operation that needs only satisfy a repair law which makes the connection between functorial laws and representation invariants explicit. The definability of such a map is a sufficient condition for functoriality. We have shown how instances of the operation for particular ADTs can be constructed using the latter's axiomatic semantics. We have shown that the definition can be adapted to construct the map for an ADT with a particular implementation. Finally, we have discussed situations in which ADTs are not functors.

8 RELATED WORK

The C++ Standard Template Library [MS96] classifies containers with iterators into sequences and collections which are, respectively, insensitive and sensitive ADTs. We prefer to use terminology that highlights the role of parametricity.

Functions ι and ε are structurally polymorphic (or polytypic [JJ96]) on the structure of so-called F-views [Nog06] which are, roughly, a way of representing the operators and the functors of FG-bialgebras. Polytypic functions on ADTs can be programmed in terms of polytypic ι, polytypic ε, and polytypic functions on algebraic data types. In particular, Af can be programmed polytypically where Pf is the polytypic map for algebraic data types.

Functors and F-algebras provide a variable-free characterisation of free algebras. A variable-free characterisation of non-free algebras is presented in [Fok96] where equations are expressed as pairs of transformers, *i.e.*, mappings from dialgebras to dialgebras that satisfy certain properties. An FG-*di*algebra is a mapping $\varphi :: Fx \to Gx$. However, homomorphisms in the category of such dialgebras cannot map to dialgebras with less equations or would not be uniquely defined.

Bialgebras are used in [Erw99] to define catamorphisms (folds) for ADTs as 'metamorphisms' where the F-algebra of the target ADT is composed with the G-coalgebra of the source ADT. Functions ι and ε can be expressed as metamorphisms. There are notable differences with our approach. First, the notion of ADT is artificial and not aimed at hiding representations. For example, a queue is an FG-bialgebra (α, ν) where α is the list F-algebra and ν is the observer F-coalgebra where dequeuing reverses the list, gets the tail, and returns the reverse of the tail. Second, there is no discussion of maps or of the satisfiability of functorial laws apart from the mention of 'invertible' ADTs for which $F = G$ and $\beta \circ \alpha = id$. Invertibility is a stronger requirement than our repair law. Finally, although metamorphisms are defined as the least solutions of an equation, it is not shown how to arrive at the solutions.

A container type is defined in [HdM00] as a relator with membership. The result is presented in the context of allegory theory. The discussion of Section 6.2 suggests f and g must preserve some sort of membership test in order for the repair law to hold. More work is needed to understand a possible connection.

A container type is defined in [AAG05] as a dependent pair type $(s \in S) \times Ps$ where S is a set of shapes and for every $s \in S$, Ps is a set of positions. Containers can characterise algebraic data types and insensitive ADTs. 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 at the object level $Fx = (s \in S) \times (Ps \to x)$. In words, F sends a type x to the dependent pair type where the first component s is a type and the second component is a function 'labelling' the positions over s with values in x. The authors have extended the container definition with *quotient* containers (containers with an equivalence relation on positions) which capture a few sensitive ADTs such as bags [AAGM04]. This work is related to Joyal's combinatorial species [Joy86] where types are described combinatorially in terms of the power series of the arrangements of payload and shapes in an implementation type.

Acknowledgements

The author is grateful to Roland Backhouse, Jeremy Gibbons, Henrik Nilsson, Conor McBride, and the anonymous reviewers for their invaluable feedback and bibliographic pointers. This work has been partially funded by EPSRC grant GR/S27078/01.

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 *International Conference on Mathematics of Program Construction (MPC'04)*, 2004.

[BW90] Michael Barr and Charles Wells. *Category Theory for Computing Science*. Prentice-Hall, 1990.

[Erw99] Martin Erwig. Categorical programming with abstract data types. In *International Conference on Algebraic Methodology and Software Technology (AMAST'98)*, pages 406–421. Springer-Verlag, 1999.

[Fok96] Maarten M. Fokkinga. Datatype laws without signatures. *Mathematical Structures in Computer Science*, 6:1–32, 1996.

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

[Hoa72] C. A. R. Hoare. Proofs of correctness of data representations. *Acta Informatica*, 1:271–281, 1972.

[JJ96] Johan Jeuring and Patrik Jansson. Polytypic programming. In *Advanced Functional Programming*, number 1129 in LNCS, pages 68–114. Springer-Verlag, 1996.

[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.

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

[McC78] John McCarthy. History of LISP. *SIGPLAN Notices*, 13(8), August 1978.

[MFP91] Erik Meijer, Maarten Fokkinga, and Ross Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *Functional Programming Languages and Computer Architecture (FPCA'91)*, volume 523 of *LNCS*. Springer-Verlag, 1991.

[Mit96] John C. Mitchell. *Foundations for Programming Languages*. MIT Press, 1996.

[MS96] David R. Musser and Atul Saini. *STL Tutorial and Reference Guide*. Addison-Wesley, Reading, Mass., 1996.

[Nog06] Pablo Nogueira. *Polytypic Functional Programming and Data Abstraction*. PhD thesis, School of Computer Science and IT, University of Nottingham, UK, 2006.

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

[PPN96] Pedro Palao Gostanza, Ricardo Peña, and Manuel Núñez. A new look at pattern matching in abstract data types. In *International Conference on Functional Programming (ICFP'96)*, pages 110–121. ACM Press, 1996.

[Wad87] Philip Wadler. Views: A way for pattern matching to cohabit with data abstraction. In *Principles of Programming Languages (POPL'87)*, pages 307–312. ACM Press, 1987.

[Wad89] Philip Wadler. Theorems for free! In *Functional Programming Languages and Computer Architecture (FPCA'89)*, pages 347–359. ACM Press, 1989.