Complexity bounds for container functors and comonads

Dominic Orchard

University of Kent

Abstract

The notion of *containers*, due to Abbott *et al.*, characterises a subset of parametric data types which can be described by a set of shapes and a set of positions for each shape. This includes common data types such as tuples, lists, trees, arrays, and graphs. Various useful categorical structures can be derived for containers that have some additional structure on their shapes and positions. For example, the notion of a *directed container* (due to Ahman *et al.*) gives rise to container comonads. Containers, and refinements such as directed containers, provide a useful reasoning tool for data types and an abstraction mechanism for programming, *e.g.*, building libraries parameterised over containers. This paper studies the performance characteristics of traversal schemes over containers modelled by additional functor and comonad structure. A cost model for container transformations is defined from which complexity bounds for the operations of container functors and comonads are derived. This provides a reasoning principle for the performance of programs structured using these idioms, suggesting optimisations which follow from the underling mathematical structure. Due to the abstract interface provided by the syntax of containers and category theory, the complexity bounds and subsequent optimisations they imply are implementation agnostic (machine free). As far as we are aware, this is the first such study of the performance characteristics of containers.

1. Introduction

Consider the following two program fragments, written in some imperative language, where `A`, `B`, and `C` are one-dimensional arrays and `f` and `g` are pure functions:

```
for i = 1..n                    for i = 1..n
   for u = 1..n                    B(i) = f(A, i)
      B(u) = f(A, u)            for i = 1..n
   C(i) = g(B, i)                  C(i) = g(B, i)
```
(1)

The two programs are extensionally equivalent; given the same inputs, they compute the same result. However, the two programs are not *intensionally* equivalent; the left-hand program has worse performance. This inefficiency is revealed by an execution-time analysis. The right program has execution time $\in \mathcal{O}(n[\mathtt{f}] + n[\mathtt{g}])$ where $[\mathtt{f}], [\mathtt{g}]$ are the execution times of `f` and `g`. The left program however has execution time $\in \mathcal{O}(n^2[\mathtt{f}] + n[\mathtt{g}])$ where the nested loop for f performs redundant recomputation. Given this computational complexity information, it would be reasonable for an optimising compiler to transform the left program into the right, since this leads to a potential asymptotic improvement. If the term $n[\mathtt{g}]$ does not dominate (grow faster than) either $n^2[\mathtt{f}]$ or $n[\mathtt{f}]$ then the transformation from left to right is an improvement from quadratic to linear time in n. Otherwise, if $n[\mathtt{g}]$ is dominant, the optimisation does not make the complexity any worse.

This paper studies a generalisation of the above example and reasoning (and we return to the above example later, Example 6, p. 15). We consider a general class of parametric data types known as *containers*, as characterised by Abbott *et al.* [1, 2] (which includes arrays, lists, labelled trees and graphs) and consider *functor* and *comonad* structures which model common traversal schemes over containers akin to *map* and *gather* patterns. We give complexity bounds for the

operations of container functors and comonads. From these complexity bounds, the usual axioms of functors and comonads can be converted to optimising rewrite rules which can be exploited by a programmer or compiler (or by a library which can specify rules to the compiler, *e.g.*, as provided by the Glasgow Haskell Compiler's rewrite rules system [19]).

We study functors since they capture the common programming pattern of an element-wise traversal of a data structure, *e.g.*, the *map* function on lists, or transforming every element of an array [21]. We study comonads since they generalise a functor's traversal, capturing *local transformations* [17, 18, 23] which may depend on an element and its neighbours, (often called a *gather* [9]), *e.g.*, convolutions, fluid simulations, and cellular automata such as the Game of Life. We study containers with additional structure, called *directed containers*, which induce a comonad for the container, as introduced by Ahman, Chapman, and Uustalu [3]. The novelty in this work is the analysis of the complexity of these derived functor and comonad structures on containers.

Our general motivation is to provide a basis for deriving compiler optimisations on container-based programs. To this end, we consider two varieties of functor/comonad structure throughout: (1) those *derived* from (directed) container structures and (2) those that are *abstract* (non-derived), where the implementation is unknown (as in an abstract data type). An example abstract situation may be a piece of code parameterised by a (container) functor, *e.g.*, an overloaded Haskell function of type `f :: (Container c, Functor c) => c a -> ...`, where we wish to reason about complexity of `f` regardless of the unknown instantiation of the container functor `c`. In the abstract case, complexity bounds similar to those for the derived structure can be calculated, but with some approximation. This provides a kind of implicit-complexity measure, which is made explicit in the derived structure where the implementation is given. Complexity measures in the abstract setting enable optimising compilation even in the presence of abstract container parameters, or as-yet-unknown implementations of containers and their associated structures.

The following section introduces preliminary definitions of containers, container cost analysis, and notation. Section 3 and Section 4 study container functors and comonads, and their complexity. Section 5 considers related work and Section 6 concludes with some further discussion.

2. Preliminaries

2.1. Containers

The notion of a *container* corresponds to parametric data types that contain only strictly positive occurrences of the parameter type. An alternate characterisation is that containers comprise a set of *shapes* and a set of *positions* for each possible shape [1, 2]. Since we study complexity bounds on operations over containers, we restrict ourselves to a subset of containers that have finite sets of positions for every shape (but the number of shapes may be infinite) We introduce here the main definitions, but more detail can be found in the work of Abbott *et al.* [1, 2].

Section 4 introduces the additional structure of directed containers.

All definitions are made in some base category $\mathcal{C}$ assumed to be Cartesian closed and locally Cartesian closed. The λ-calculus is therefore used as the internal language of $\mathcal{C}$ for convenience.

Definition 1 (Containers)**.** A container comprises a set of shapes $S : \mathbf{Set}$ and a shape-indexed family of positions $P : S \to \mathbf{Set}$, often written as the pair $S \triangleleft P$. Shapes can be thought of as templates for a data structure where positions identify the "holes" that can be filled with data.

For the sake of generality, we can replace **Set** with $\mathcal{C}$ above, situating a container within some arbitrary (locally) Cartesian closed category. The reader may instantiate $\mathcal{C}$ to **Set** if they wish to think concretely, and our examples are all within **Set**.

Example 1 (Lists)**.** A running example will be the data type of lists, defined by the container $\mathbb{N} \triangleleft \mathsf{Fin}$. Shapes are natural numbers, corresponding to the length of a list, and positions are drawn from finite sets, where for a shape $s : \mathbb{N}$ positions are $\mathsf{Fin}\, s = \{0, \ldots, s-1\}$.

Definition 2 (Container data type)**.** From a container $S \triangleleft P$, a parametric data type (object mapping on $\mathcal{C}$) is defined $[\![S \triangleleft P]\!]\, A = \Sigma s : S.(P\, s \to A)$ as a dependent sum of a shape $s : S$ and a *valuation* from positions in the shape (taken from $P\, s$) to values A. These values are called the *elements* of the container.

Throughout, a tuple notation (s, v) will be used for inhabitants of the dependent sum of a container type, with shape $s : S$ and valuation $v : P\, s \to A$. These tuples (s, v) will be referred to as *container values*.

Definition 3 (Finite containers)**.** A container $S \triangleleft P$ is *finite* if for every shape $s : S$ its position set $P\, s$ is finite. That is, there is a natural transformation $\mathsf{size}_A : [\![S \triangleleft P]\!]A \to \mathbb{N}$, defined:

$$\mathsf{size}_A\ (s, v) = |P\, s|$$

where the size of a container value is the cardinality of the position set for its shape. Finiteness necessarily implies that the set of positions is discrete.

Lists are finite containers since for every shape $s : \mathbb{N}$ (length) the set of positions is the finite set $\{0, \ldots, s-1\}$ (see Example 1 above). Note that lists are still finite containers despite there being an infinite number of shapes. Other common containers in programming (trees, arrays) follow a similar scheme: for a given shape there are a finite number of positions.

Example 2 (Streams)**.** The stream container has a single shape, denoted $*$, whose positions are drawn from the natural numbers, *i.e.*, $\mathsf{Stream} = \{*\} \triangleleft (\mathit{const}\ \mathbb{N})$. Thus, streams are not finite containers; streams are infinite.

Finite containers relate to Girard's definition of a *normal functor* (representable by a power series) [8], which are equivalent to container functors with a countable set of shapes and finite sets of positions for each shape [2]. Whilst the above definition of finite containers is not restricted to countably many shapes, this is sufficient for our purposes since we need not enumerate all shapes. In any case, all the examples in this paper happen to have countable shapes since this is common in programming.

For brevity, the word *container* will refer throughout to finite containers unless explicitly stated otherwise. We restrict the development to finite containers since there is a clear notion of *input size* which give finite bounds on the complexity of whole-structure traversals. We discuss infinite containers briefly in Section 6, using the stream container example.

Example 3 (Real functions)**.** For completeness sake, an example of an infinite and non-discrete (continuous) container is the container of multi-dimensional real-valued functions $\mathbb{N} \triangleleft (\lambda n.\mathbb{R}^n)$ where shapes count the number of dimensions and positions are vectors of real numbers. We do not consider complexity bounds for such containers since there is no natural measure of input size.

2.2. Cost semantics

A cost model for container values is key to the results of this paper.

For a function $f : A \to B$, its execution time on inputs A of size n will be denoted $[f]_n$. We do not necessarily have a model for the execution time of every f in the underlying category, nor a size function for A values. Therefore, in some cases, this execution time will be left abstract. However, functions on containers $\mathsf{op} : [\![S \triangleleft P]\!]A \to B$ will have their execution time $[\mathsf{op}]_n$ given concretely by a cost model function $[-]$ which maps container terms to $\mathbb{N}$, where:

$$[\mathsf{op}]_n = [\mathsf{op}\,(s, v)] \quad \textit{iff}\ \ \exists (s, v) : [\![S \triangleleft P]\!]A\ \wedge\ \mathsf{size}_A(s, v) = n$$

The costing function $[-]$ is recursively defined over the syntax of terms used to define container operations op which is a subset of λ-terms (Proposition 1):

Proposition 1. Container operations op (mapping from container values to container values) are defined in this paper in terms of the following sub-grammar of λ-calculus terms t extended with

position terms p, shape terms s, and abstract functions f which are either parameters to op or input valuations v:

$$t ::= (s,t) \mid x \mid (\lambda x_p.t) \mid p \mid f\ t \mid (\lambda x_p.t)\ p \tag{2}$$

That is $\mathsf{op}(s,v) = t$ where terms t are either container values (s,t), variables, λ-terms abstracting over positions (where position variables are denote x_p), position terms, β-redexes on abstract functions or valuations f, or β-redexes of a λ-term with a position.

Definition 4 (Cost model)**.** The cost function $[-] : t \to \mathbb{N}$ assigns a cost to container operations.

The cost for input valuations or any (universally quantified) function f that parameterises a container operation op is kept abstract in terms of $[f]_n$. This provides a flexible cost semantics that can be instantiated for particular abstract-machine models via instantiations of $[-]_n$. The only assumption we place on $[-]_n$ is that it is monotonic on its input size. The costing is defined:

$$\begin{aligned} [(s,t)] &= \textstyle\sum_{\forall p : P\,s} [t\ p] \\ [x] &= 0 \\ [\lambda x_p.t] &= 0 \\ [p] &= 0 \\ [f\ t] &= 1 + [f]_{|t|} + [t] \\ [(\lambda x_p.t)\ p] &= 1 + [t[p/x_p]] \end{aligned} \tag{3}$$

The first clause gives a cost analysis of a container value $(s,t) : \Sigma s : S.(P\,s \to A)$ which enumerates every position for the shape s, summing the cost of evaluating the container valuation t at each position. Therefore, given a container value (s,v) of size $n = |P\,s|$ (number of positions at the present shape), if $\forall p : P\,s.\,[v\ p] \in \mathcal{O}(f)$ it follows that the complexity of evaluating the container is $[(s,v)] \in \mathcal{O}(nf)$.

Variables, position terms, and naked λ-terms cost nothing. Thus complexity measures in this paper do not take into account the different sizes of positions, which are instead uniformly treated as having a constant unit size $|p| = 1$.

The application of an abstract function or valuation f to a term t is defined using the parameter costing $[-]_n$. This connects the concrete costing function to the abstract computation-time, where the cost of application is $[f]_{|t|}$ (parameterised by the size of its argument) plus the evaluation cost of t and plus 1 to mark the computation step involved in the β-reduction of the application. A similar costing is used by Wadler [26, §3] for strict timing analysis, where the cost of function application is the cost of the function body plus one, plus the cost of the arguments.

The cost of a β-redex with a position argument has the cost of one plus the cost of the reduced term (with syntactic substitution $[p/x_p]$ of the position p for variable x_p). This matches the idea that positions are of constant size and incur only a constant cost when substituted.

Counting β-reductions is known to be an imprecise cost model for the λ-calculus as the number of syntactic substitutions required is not constant and the size of substituted terms is not accounted for (see the discussion in [6]). However, in the above definition, the abstract costing $[-]_n$ provides the main cost and can be instantiated with some more precise abstract machine model. The counting of β-reductions is not used ubiquitously in the costing, only to account for applications to position constants or applications of abstract (parameter) functions. The key point of the costing $[-]$ is to allow the cost of a valuation to be scaled by the number of elements in the container. As shall be noted throughout, the counting of β-reductions incurred by $f\,t$ or $(\lambda x.t)\,p$ ends up being subsumed by the costs of the abstract functions f being scaled by a function of the input size n.

Definition 5 (Structural sizes)**.** For nested data structures, the notation $n[\leq m]$ denotes a container value of size n whose elements are bounded above by size m. This is similar to the notation of Skillicorn and Cai who write the size of a nested data structure as a list of elements $[n, m]$ meaning the outer layer has size n and the inner elements of size at most m [22].

Similarly, $n[m \leq]$ denotes a container value of size n whose elements are bounded below by size m (*i.e.*, at least of size m).

Proposition 2. Consider an extensional equality $f \equiv g$ between two program fragments f and g (that is, they have the same output for the same input). If $[f]_n \in \Omega([g]_n)$ (or equivalently $[g]_n \in \mathcal{O}([f]_n)$) then the equality can be oriented as a rewrite rule from left to right as $f \rightsquigarrow g$ which improves the asymptotic complexity of the program. Thus this rewrite rule may provide a program optimisation.

Section 3 and Section 4 study functor and comonad structures on containers. In both, we consider derived instances of this structure, where the definition is given explicitly, and abstract instances of the structure, where the definition is unknown. For both functors and comonads, we characterise what it means to be derived and abstract. In the case of abstract container comonads, we can still reason about complexity based on the axioms of the comonad structure.

3. Functors

We consider containers which have the additional structure of a functor. The morphism mapping of a container functor captures the common programming pattern of applying a function to every element in a container data type, generalising the *map* combinator for lists.

Definition 6 (Derived container functor)**.** Every container $S \triangleleft P$ induces an (endo)functor $[\![S \triangleleft P]\!] : \mathcal{C} \to \mathcal{C}$ (for some base category $\mathcal{C}$), defined on objects and morphisms respectively by:

- $[\![S \triangleleft P]\!]A = \Sigma s : S.(P\,s \to A)$ (as in Definition 2);
- $[\![S \triangleleft P]\!]\,f\ (s, v) = (s, f \circ v)$ where $f : A \to B$, $s : S$ and $v : P\,s \to A$.

Thus, the morphism mapping is the post-composition of the morphism $f : A \to B$ with the valuation, giving a container value $[\![S \triangleleft P]\!]\,B$ of transformed elements.

Note that the composition $f \circ v$ can be equivalently expressed as $\lambda p.f\,(v\,p)$, *i.e.*, a term in the restricted subset of the λ-calculus used to construct valuations (see Proposition 1).

Theorem 1 (Container functor cost)**.** For a container $S \triangleleft P$, the morphism mapping of its functor $\mathsf{F} = [\![S \triangleleft P]\!]$ has complexity:

$$[\mathsf{F}\,f]_{n[\leq m]} \in \mathcal{O}(n[f]_m + nQ_n)$$

for all $f : A \to B$ and for some cost function Q_n of the input size n which characterises the time to compute valuations at each position (the *indexing* time). Thus, the execution time of a morphism mapping is linear in n times the cost of f (the transformation) plus the cost of indexing.

Proof. Following from the definition of container functors (Definition 6) and the cost function (Definition 4, p. 4), then for some $(s, v) : [\![S \triangleleft P]\!]\,A$ where $n = |P\,s|$ and $\forall p : Ps.\ |v\,p| \leq m$ (*i.e.*, elements have size of at most m), then:

$$\begin{aligned}
[\mathsf{F}\,f\,(s, v)] = [(s, f \circ v)] = [(s, \lambda p.f\ (v\,p))] &= \sum_{\forall p : Ps} ([\lambda p.f\ (v\,p)]) \\
&= \sum_{\forall p : Ps} ([f\ (v\,p)] + 1) \\
&= \sum_{\forall p : Ps} ([f]_{|v\,p|} + [v\,p] + 2) \\
\{since\ |p| = 1\} \quad &= \sum_{\forall p : Ps} ([f]_{|v\,p|} + [v]_1 + 3) \\
&= n[f]_{|v\,p|} + n[v]_1 + 3n \\
\{since\ |v\,p| \leq m\} \quad &\leq n[f]_m + n[v]_1 + 3n
\end{aligned}$$

$$\begin{aligned}
\{assuming\ [f]_m > 0\ and\ [v]_1 > 0\} \quad &\Rightarrow \quad [\mathsf{F}\,f\,(s, v)] \in \mathcal{O}(n[f]_m + n[v]_1) \\
\{let\ Q_n > 0\ be\ upper\text{-}bound\ cost\ of\ [v]_1\} \quad &\therefore [\mathsf{F}\,f]_{n[\leq m]]} \in \mathcal{O}(n[f]_m + nQ_n)
\end{aligned}$$

Note, whilst $[v]_1$ is parameterised by constant input size 1, its cost depends also on the container value size n since $v : P\, s \to A$, hence the term Q is parameterised by the input size n. □

Remark 1. The cost to reduce the term $(\lambda p.(f\,(v\,p))\,p$ in the resulting valuation is given as $3n$ by our costing function. However, this cost is subsumed by the costs $n[f]_m$ and $n[v]_1$.

Corollary 1. Theorem 1 assumes that the elements of the input container have a well-defined size which is at most m. If there is no such size defined for the elements, the theorem (and its proof) still holds but with $m = 1$, *i.e.*, $[\mathsf{F}\, f]_n \in \mathcal{O}(n[f]_1 + nQ_n)$.

The result of Theorem 1 is straightforward from the definition and well-known, at least tacitly, for concrete instances of containers such as lists, trees, and other inductive types (where Q_n is constant). However, in the context of an optimising compiler, the implementation of a container functor may be unknown, rather than derived as in Definition 6. Whilst a container functor may be extensionally equivalent to that of Definition 6, it may differ intensionally. For example, an inefficient implementation for lists might use a linear-time lookup for each element (based on its index), rather than the usual recursive definition; or, lists with repeated elements might be stored more efficiently with a run-length encoding style implementation. However, we can still derive useful complexity bounds even in this abstract case, with some restriction on what it means to be an abstract finite container functor.

Definition 7. An endofunctor $\mathsf{F} : \mathcal{C} \to \mathcal{C}$ is an *abstract finite container functor* if there exists a finite container $S \triangleleft P$ such that $\mathsf{F} \cong [\![S \triangleleft P]\!]$, that is there exists a natural isomorphism, with $\alpha_A : \mathsf{F}A \to [\![S \triangleleft P]\!]A$ and inverse $\alpha_A^{-1} : [\![S \triangleleft P]\!]A \to \mathsf{F}A$.

Remark 2. An abstract finite container value has a definition of size via the isomorphism to some concrete finite container. That is, $\mathsf{size}'_A : \mathsf{F}A \to \mathbb{N}$ is given by $\mathsf{size}'_A = \mathsf{size}_A \circ \alpha_A$.

We first show a lower bound for such abstract container functors in the worst case.

Proposition 3. Let F be an abstract finite container functor, isomorphic to $[\![S \triangleleft P]\!]$. Assuming that the costing $[\mathsf{F}\, f]_n$ is defined, then the morphism mapping of F has worst-case lower-bound complexity $[\mathsf{F}\, f]_n \in \Omega(n[f]_1)$, where the input size n is calculated by size'_A.

Proof. The isomorphism to a concrete container explains the extensional behaviour of the abstract container functor; by the isomorphism $F \cong [\![S \triangleleft P]\!]$ it follows that $\mathsf{F}\, f = \alpha_B^{-1} \circ [\![S \triangleleft P]\!] f \circ \alpha_A$ for some $f : A \to B$. By the definition of the morphism mapping of concrete container functors $[\![S \triangleleft P]\!]\, f\, (s, v) = (s, f \circ v)$, it follows that $\mathsf{F}f$ extensionally preserves the size of the incoming container and applies f to every element in a container value. Thus, $[\mathsf{F}\, f]_n$ is bounded below by $n[f]_1$ in the worst case. □

Remark 3. The above proposition is stated as a *worst-case* lower bound. This accounts for functor implementations with a tighter lower bound complexity for some inputs by utilising features of the underlying elements. For example, a compact representation could be provided for lists with shared or repeated elements. A worst case scenario for such an implementation would be a list with no common elements, inducing the linear lower bound.

A lower-bound complexity result is interesting, but not particularly useful for optimisation purposes: given two extensionally equivalent programs $f \equiv g$ if we only know two lower-bounds $[f]_n \in \Omega(t(n))$ and $[g]_n \in \Omega(t'(n))$ we cannot determine whether either $f \rightsquigarrow g$ or $g \rightsquigarrow f$ provides any improvement; upper bounds are much more useful. However, in the abstract setting we can only give an upper bound parameterised by representations of additional unknown costs.

Theorem 2 (Abstract container functor cost)**.** Let F be an abstract finite container functor. Its morphism mapping has upper-bound complexity, where for all $f : A \to B$

$$[\mathsf{F}\, f]_{n[\leq m]} \in \mathcal{O}(n[f]_m + nQ_n + R_n)$$

where Q_n is a function of the input size n representing the time to access each element in the container (the time to evaluate a valuation). The term R_n is a function of n and acts as a *catch all*: we cannot rule out that possibility of some pathological implementations with redundant expensive computation whose complexity far outweighs the $(n[f]_m + nQ_n)$ term.

If there is no well-defined notion of size for elements A, then $m = 1$ (see Corollary 1, p. 6).

Proof. This result builds on the lower-bound complexity of F (Proposition 3) which was based on the extensional behaviour of F via its isomorphism to a concrete container. An upper bound must be at least as big as the lower bound, hence the first part of complexity bound here is $n[f]_m$.

In the isomorphic concrete container, accessing an element has constant time but this may not be the case in F. Instead, the cost of accessing each element is accounted for by the term nQ_n (since there must be at least n accesses by the earlier argument). Finally, the factor R_n provides the opportunity for any additional wasteful computation to dominate the linear-time portion. □

Remark 4. The above theorem is quite weak: the R_n factor means we do not have a concrete upper bound. However, this still provides a bound that can be used to reason about optimisations. (A similar situation occurs with comonads in the next section). We can analyse two cases: when R_n is dominant, and when it is not, and show that an optimisation does not get asymptotically worse even if R_n is dominant.

Optimisation 1 (Functor optimisation)**.** For an abstract or derived container functor F, the axiom $\mathsf{F}id \equiv id_{\mathsf{F}}$ can be oriented as an optimising rewrite rule $\mathsf{F}id \rightsquigarrow id_{\mathsf{F}}$.

Proof. Straightforward for either derived or abstract container functors. Let $id_{\mathsf{F}}(s, v) = (s, v)$, which in our model has cost:

$$[id_{\mathsf{F}}\ (s,v)] = [(s,v)] = \sum_{\forall p : Ps} [v\, p] = \sum_{\forall p : Ps} (1 + [v]_{|p|} + [p]) = \sum_{\forall p : Ps} (1 + [v]_1) = n + n[v]_1$$

Thus, let $Q_n = [v]_1$ then identity has complexity $[id_{\mathsf{F}}]_{n[\leq m]} \in \mathcal{O}(nQ_n)$ and $[\mathsf{F}id]_{n[\leq m]} \in \mathcal{O}(n[f]_m + nQ_n + R_n)$ where $R_n = 0$ in the case of derived functors. Therefore, regardless of whether R_n dominates or not, the rewriting $\mathsf{F}id \rightsquigarrow id_{\mathsf{F}}$ is an optimisation since it either reduces asymptotic complexity (if $[f]_m$ dominates Q_n) or maintains it. □

Remark 5. One might expect the cost of the identity function to be constant, *i.e.*, $(\lambda x.x) \in \mathcal{O}(1)$. However, we make complexity claims relative to our cost model of container operations. The model essentially includes an evaluation of a container value at every position (in the current shape), thus identity on a container value is not constant, but requires the resulting container to be evaluated.

The remaining functor axiom $\mathsf{F}(g \circ f) = \mathsf{F}g \circ \mathsf{F}f$ is exploited in the classic deforestation transformation $\mathsf{F}g \circ \mathsf{F}f \rightsquigarrow \mathsf{F}(g \circ f)$ [27]. Performance is improved by eliminating the intermediate data structure between the two morphism mappings and performing one traversal instead of two. However, this rewrite is not justified via the results here since the complexity of each term is the same; asymptotic bounds cannot distinguish n from $2n$. To justify this axiom (in a general setting) requires a more fine-grained (non-asymptotic) costing analysis outside the scope of this paper.

4. Container comonads

The morphism mapping of the functor construction for containers captures the programming pattern of an element-wise traversal, applying a transformation $f : A \to B$ to each element. Comonads generalise this element-wise traversal by allowing a transformation to depend not just on a single element, but on an element and its neighbours. The following compares the signatures of the morphism mapping for a functor (left) with the *extension* operation of a comonad (right):

$$\textit{functor}\ \frac{f : A \to B}{\mathsf{F}f : \mathsf{F}A \to \mathsf{F}B} \qquad \textit{comonad}\ \frac{g : \mathsf{F}A \to B}{g^{\dagger} : \mathsf{F}A \to \mathsf{F}B}$$

On the left, the functor applies the morphism f *pointwise*, where f computes a B value from a single A value. On the right, extension applies the morphism g *contextwise*, where g computes a B value from possibly multiple A values from the "context" provided by $\mathsf{F}A$. That is, $\mathsf{F}A$ is more than just a container functor, it is a container functor with a notion of context, such as a pointer to a particular element. Thus, g can access more than just a single element, and is therefore described as a *context-dependent computation* (see [18]). For example, the kernel function of a Gaussian blur takes the mean of an element at the current context and its immediate neighbours.

Definition 8 (Comonads)**.** For an endofunctor F, a comonad is a triple $(\mathsf{F}, \varepsilon, (-)^\dagger)$ which comprises

- a natural transformation $\varepsilon_A : \mathsf{F}A \to A$, called the *counit*;
- an *extension* operation, (uniquely) mapping morphisms $f : \mathsf{F}A \to B$ to morphisms $f^\dagger : \mathsf{F}A \to \mathsf{F}B$ satisfying the axioms:

$$\text{[C1]}\ \ \varepsilon_A^\dagger \equiv id_{\mathsf{F}A} \qquad \text{[C2]}\ \ \varepsilon_B \circ f^\dagger \equiv f \qquad \text{[C3]}\ \ g^\dagger \circ f^\dagger \equiv (g \circ f^\dagger)^\dagger$$

Extension lifts (or, *extends*) a morphism $f : \mathsf{F}A \to B$ which models a computation localised to an incoming context $\mathsf{F}A$, to a global operation $f^\dagger : \mathsf{F}A \to \mathsf{F}B$ by applying f at every possible context within the incoming $\mathsf{F}\,A$ value. For the Gaussian blur example, extension would apply the kernel function to the array *focussed* at every possible index in the array. The counit $\varepsilon_A : \mathsf{F}A \to A$ defines the notion of a *current context* at which an element is located and projected out by ε. This acts as the identity for extension [C1], [C2]. The third axiom [C3] defines associativity of $(-)^\dagger$.

Example 4 (Non-empty list comonad)**.** Non-empty lists have a comonad structure, where ε takes the head of the list and extension applies the parameter function $f : [A] \to B$ to successive sublists of the input list, defined:

$$\varepsilon\ xs = head\ xs \qquad f^\dagger\ xs = \begin{cases} [f\ [x]] & xs = [x] \\ (f\ xs) : f^\dagger(tail\ xs) & otherwise \end{cases}$$

Thus, the "context" at each element of the list is its suffix, to which extension applies f. The requirement that the lists are non-empty is such that ε is total.

We focus first on the axiom [C3] $g^\dagger \circ f^\dagger \equiv (g \circ f^\dagger)^\dagger$ which reassociates extension. The nested use of extension on the right-hand side suggests the possibility of a quadratic difference in complexity compared with the left, which has no such nesting. This is exactly the situation described in the introductory example (Section 1). This kind of nesting can easily arise during program construction and can lead to significant performance issues. Therefore, it is preferable to automatically eliminate this nesting (by a compiler) given a guarantee that it is always an improvement. We infer the complexity difference between the two sides of [C3] from the comonad axioms and show that the rewrite $(g \circ f^\dagger)^\dagger \rightsquigarrow (g^\dagger \circ f^\dagger)$ is indeed an optimisation.

Proposition 4 (Shape preservation)**.** The extension operation of a comonad is *shape preserving* [18, 16], similarly to the morphism mapping part of a functor. That is, for an abstract notion of shape given by $\mathsf{shape}_A : \mathsf{F}A \to \mathsf{F}1$ defined as the lifting of the terminal morphism $\mathsf{shape}_A = \mathsf{F}!_A$ (also used by Jay and Cockett [14]), then for all $f : \mathsf{F}A \to B$:

$$\mathsf{shape}_B \circ f^\dagger = \mathsf{shape}_A$$

Thus, the comonadic extension of any morphism $f : \mathsf{F}A \to B$ preserves the shape of the input value in its output.

Proof. By [C1] $\varepsilon_A^\dagger \equiv id_{\mathsf{F}A}$, for all $f : \mathsf{F}A \to B$ then:

$$\mathsf{shape}_B \circ f^\dagger \overset{def}{\equiv} \mathsf{F}!_B \circ f^\dagger \overset{ext}{\equiv} (!_B \circ f)^\dagger \overset{!_A}{\equiv} (!_A \circ \varepsilon_A)^\dagger \overset{ext}{\equiv} \mathsf{F}!_A \circ \varepsilon_A^\dagger \overset{[C1]}{\equiv} \mathsf{shape}_A$$

where the step (*ext*) follows from $\mathsf{F}f \equiv (f \circ \varepsilon)^\dagger$ (proof not shown) and [C2],[C3], and $!_A$ is the universal morphism of terminal objects. □

Corollary 2 (Size preservation)**.** For a finite container comonad, the extension operation $(-)^\dagger$ is *size* preserving, that is:

$$\mathsf{size}_B \circ f^\dagger = \mathsf{size}_A$$

Proof. Since $\mathsf{shape}_A = \mathsf{F}!_A$, by naturality of size_A it follows that $\mathsf{size}_1 \circ \mathsf{shape}_A = \mathsf{size}_A$. Combining this with Proposition 4 gives size preservation:

$$\mathsf{size}_B \circ f^\dagger \overset{natur.}{\equiv} \mathsf{size}_1 \circ \mathsf{shape}_B \circ f^\dagger \overset{Prop\ 4}{\equiv} \mathsf{size}_1 \circ \mathsf{shape}_A \overset{natur.}{\equiv} \mathsf{size}_A$$ □

The size preservation of comonadic extension is a useful property for deriving complexity bounds on extension and has the further corollary about cost:

Corollary 3. For a finite container comonad on F and for any morphisms $f : \mathsf{F}A \to B$ and $g : B \to C$ then the following property holds for the execution time:

$$[g \circ f^\dagger]_n = [g]_n + [f^\dagger]_n$$

That is, since $f^\dagger$ is size preserving, the output size of $f^\dagger$ is the same as the input and therefore the execution time of the composition can be decomposed with $[g]$ parameterised by size n.

Similarly to functors in the last section, both derived and abstract structures are considered in this section. Comonads can be derived from containers with some additional structure.

4.1. Directed containers

The subclass of containers known as *directed containers* has additional structure on shapes and positions which induces a container comonad [3]. The additional structure describes a system of *subshapes* related to each position in a shape.

Definition 9 (Directed containers)**.** A container $S \triangleleft P$ is called *directed* if it has the following additional operations for manipulating shapes and positions [3, 4]:

- $\mathsf{o} : \Pi\{s : S\}.\, P\, s$ – a *root* position for every shape (the shape parameter s is given implicitly);[1]
- $\downarrow$: $\Pi s : S.\, P\, s \to S$ – computes the *subshape* at a particular position;
- $\oplus : \Pi\{s : S\}.\, \Pi p : P\, s.P(s \downarrow p) \to P\, s$ – given a position p' in a subshape starting at p, then $p \oplus p'$ gives the position of p' within the parent shape.

These satisfy the following two equations on shapes [S1-2] and three positional equations [P1-3]:

[S1] $s \downarrow \mathsf{o} = s$

[S2] $s \downarrow (p \oplus p') = (s \downarrow p) \downarrow p'$

[P1] $p \oplus \mathsf{o} = p$

[P2] $\mathsf{o} \oplus p = p$

[P3] $(p \oplus p') \oplus p'' = p \oplus (p' \oplus p'')$

[1] The implicit parameters of an operation are marked by surrounding them in braces {...}.

The shape parameters have been kept implicit in the above axioms, but [P3], for example, can be rendered with its shape arguments explicit as $(p \oplus\{s\}\, p') \oplus\{s\}\, p'' = p \oplus\{s\}\, (p' \oplus\{s \downarrow p\}\, p'')$.

Modulo the dependent types, rules [P1-3] are essentially that $\oplus$ and o form a monoid over positions, and [S1-2] that $\downarrow$ is a right-monoid action with the set of shapes.

Definition 10 (Derived container comonad)**.** Every directed container $(S \triangleleft P, \downarrow, \mathsf{o}, \oplus)$ induces a comonad for the functor $[\![S \triangleleft P]\!] : \mathcal{C} \to \mathcal{C}$ with counit and extension:

$$\varepsilon(s, v) = v\ \mathsf{o}\{s\}$$
$$f^\dagger(s, v) = (s, \lambda p.\, f\ (s \downarrow p,\ \lambda p'.\, v\ (p \oplus\{s\}\ p')))$$

The counit operation ε takes the valuation at the root position o. The extension $f^\dagger$ on a container value (s, v) produces a container where at each position p the morphism $f : \mathsf{F}A \to B$ is applied to a sub-container of shape $s \downarrow p$ whose valuation takes elements at positions offset by $p \oplus\{s\}$.

Example 5 (Directed list container and its comonad)**.** Example 1 (p. 2) defined the list container as $\mathbb{N} \triangleleft \mathsf{Fin}$ where shapes are list lengths. Non-empty lists are a container with $\mathbb{N}_{>0} \triangleleft \mathsf{Fin}$ and a directed container with $s \downarrow p = s - p$ (the subshape is the (length of the) suffix) and $\mathsf{o}\{s\} = 0$ (the head element) and $p \oplus p' = p + p'$ (the global position of p' in the subshape starting at p is its position in the entire list).

This produces a non-empty list comonad structure which is extensionally equivalent to the non-empty list comonad given in Example 4.

Theorem 3 (Derived container comonad cost)**.** For a directed container $(S \triangleleft P, \downarrow, \mathsf{o}, \oplus)$, the derived comonad has extension $(-)^\dagger$ with the upper-bound complexity:

$$[f^\dagger]_n \in \mathcal{O}(n[f]_m + nmQ_m)$$

where $m = \mathsf{max}_{p:P\,s}|P\,(s \downarrow p)|$ is the largest subshape from s for some input $(s, v) : [\![S \triangleleft P]\!]\, A$ and Q_m is the cost of applying valuations v (parameterised on the maximum subshape size m).

Therefore the cost of applying $f^\dagger$ to a container of size n is bounded above by n times the cost of f on input size m (the size of largest possible subcontainer that f is applied to), plus nm times the valuation cost (the cost of retrieving values from the container in each application of f).

Proof. By the derived container comonad definition (Def. 10). Let $n = |P\,s|$ and $Q_m = [v]_1$, then:

$$\begin{aligned}
[f^\dagger(s, v)] &= \sum_{\forall p:Ps} (1 + [f\ (s \downarrow p, \lambda p'.v\ (p \oplus\{s\}\ p'))]) \\
&= \sum_{\forall p:Ps} (1 + [f]_{|P(s\downarrow p)|} + \sum_{\forall p':P(s\downarrow p)} (1 + [v\ (p \oplus\{s\}\ p')])) \\
\{\textit{since } |p| = 1\} \qquad &= \sum_{\forall p:Ps} (1 + [f]_{|P(s\downarrow p)|} + \sum_{\forall p':P(s\downarrow p)} (2 + [v]_1)) \\
&= n + n[f]_{|P(s\downarrow p)|} + n \sum_{\forall p':P(s\downarrow p)} (2 + [v]_1) \\
\{m = \mathsf{max}_{p:P\,s}|P\,(s \downarrow p)|\} \qquad &\leq n + n[f]_m + nm(2 + [v]_1)
\end{aligned}$$

$$\{Q_m = [v]_1 \wedge Q_m > 0\}\ \therefore [f^\dagger]_n \in \mathcal{O}(n[f]_m + nmQ_m)$$

Since positions are of constant size, we treat the cost of $\oplus$ as constant also in the above proof. □

Remark 6. In the above theorem and proof, the maximum size of the subshapes m is $\geq n$ by the axiom [C2] $\varepsilon \circ f^\dagger \equiv f$ instantiated with $f = \mathsf{size}$ giving $\varepsilon_{\mathbb{N}} \circ \mathsf{size}^\dagger_A = \mathsf{size}_A$ *i.e.*, the size of the subshape at the current context is equal to the size of the original container (this is exposed in the directed container structure by axiom [S1]). Therefore, as m is the size of the largest subshape, and the subshape at the root position is the size of n, it follows that $m \geq n$.

The above theorem provides the basis to show that [C3] can be oriented as an optimisation:

Optimisation 2. For a derived container comonad, axiom [C3] can be oriented as $(g \circ f^\dagger)^\dagger \rightsquigarrow g^\dagger \circ f^\dagger$ guaranteeing an asymptotic improvement or not making the complexity worse.

Proof. Assume input (s, v). Let $n = |P\, s|$, $m_1 = \mathsf{max}_{p:P\, s}|P\,(s \downarrow p)|$ (largest subshape size) and $m_2 = \mathsf{max}_{p:P\, s, p':P(s\downarrow p)}|P\,((s \downarrow p) \downarrow p')|$ (largest sub-subshape size). By Theorem 3 and Corollary 3:

$$\begin{aligned}
[g^\dagger \circ f^\dagger]_n &= [g^\dagger]_n + [f^\dagger]_n \\
&\in \mathcal{O}(n[g]_{m_1} + nm_1Q_{m_1} + n[f]_{m_1} + nm_1Q_{m_1}) \\
[(g \circ f^\dagger)^\dagger]_n &\in \mathcal{O}(n[g \circ f^\dagger]_{m_1} + nm_1Q_{m_1}) \\
&\in \mathcal{O}(n([g]_{m_1} + m_1[f]_{m_2} + m_1m_2Q_{m_2}) + nm_1Q_{m_1}) \\
&\in \mathcal{O}(n[g]_{m_1} + nm_1Q_{m_1} + nm_1[f]_{m_2} + nm_1m_2Q_{m_2})
\end{aligned}$$

Each bound shares a common summand $n[g]_{m_1} + nm_1Q_{m_1}$ and a common factor n on the remaining terms. It therefore remains to prove $([f]_{m_1} + m_1Q_{m_1}) \in \mathcal{O}(m_1[f]_{m_2} + m_1m_2Q_{m_2})$ (the remaining distinct subterms).

This follows from the property that $m_1 = m_2$, that is the maximum subshape size is equal to the maximum sub-subshape size, which follows from the directed container axioms:

1. First, consider a general property on functions: let X and Y be sets and $a : X \to Y$ and $b : Y \to Z$ be functions where X and Y are finite and a is surjective.

   Then: $\{b(y) : \forall y \in Y\} = \{b(a(x)) : \forall x \in X\}$ since a maps to its entire co-domain Y.

2. From axiom [P1], it follows that $\oplus\{s\}$ is surjective, mapping to all values in its co-domain $P\, s$. That is, $\forall p \in (P\, s)\ \exists p_1 \in (P\, s)$ and $\exists p_2 \in (P\,(s \downarrow p))$ such that $p_1 \oplus\{s\}\, p_2 = p$, where $p_1 = p$ and $p_2 = \mathsf{o}$ by axiom [P1] $p \oplus \mathsf{o} = p$.

3. Instantiate the general property (1) with $X = \Pi p : P\, s.P\,(s \downarrow p)$ (which is finite due to our restriction to finite containers and hence finite sets of positions), $Y = P\, s$ (also finite) and $Z = \mathbb{N}$, with $a = \oplus s$ (which is surjective (2)) and $b\, p = |P\,(s \downarrow p)|$ (the cardinality of positions at subshape s from position p). Thus:

   $$\{|P\,(s \downarrow p)| : \forall p \in P\, s\} = \{|P\,(s \downarrow (p_1 \oplus \{s\}p_2)| : \forall p_1 \in P\, s, \forall p_2 \in P(s \downarrow p)\}$$

   which are both finite sets by the finiteness of X and Y.

4. From axiom [S2], $\forall\{s, p_1 : Ps, p_2 : P(s \downarrow p_1)\}\,.\, s \downarrow (p_1 \oplus p_2) = (s \downarrow p_1) \downarrow p_2$ it follows that:

   $$\{|P\,(s \downarrow p)| : \forall p \in P\, s\} = \{|P\,((s \downarrow p_1) \downarrow p_2)| : \forall p_1 \in P\, s, \forall p_2 \in P(s \downarrow p)\}$$

5. Let $m_1 = \mathsf{max}_{p:P\, s}|(s \downarrow p)|$ and $m_2 = \mathsf{max}_{p_1:P\, s, p_2:P(s\downarrow p_1)}|P\,((s \downarrow p_1) \downarrow p_2)|$ then from (3) and (4) it follows that $m_1 = m_2$:

   $$m_1 = \mathsf{max}\{|P\,(s \downarrow p)| : \forall p \in P\, s\} = \mathsf{max}\{|P\,((s \downarrow p_1) \downarrow p_2)| : \forall p_1 \in P\, s, \forall p_2 \in P(s \downarrow p)\} = m_2$$

Therefore the largest subshape size m_1 equals the largest sub-subshape size m_2 and therefore $([f]_{m_1} + m_1Q_{m_1}) \in \mathcal{O}(m_1[f]_{m_1} + m_1m_1Q_{m_1})$. Overall, this gives bounds for either side of [C3]:

$$\begin{aligned}
[g^\dagger \circ f^\dagger]_n &\in \mathcal{O}(n[g]_{m_1} + nm_1Q_{m_1} + n[f]_{m_1} + nm_1Q_{m_1}) \\
[(g \circ f^\dagger)^\dagger]_n &\in \mathcal{O}(n[g]_{m_1} + nm_1Q_{m_1} + nm_1[f]_{m_1} + nm_1^2Q_{m_1})
\end{aligned}$$

Regardless of whether Q or $[f]$ dominates then $[g^\dagger \circ f^\dagger]_n \in \mathcal{O}[(g \circ f^\dagger)^\dagger]_n$ justifying the rewrite. □

4.2. Abstract container comonads

For abstract (non-derived) container comonads, this section shows their complexity is similar to those of the derived container comonads. But how do we characterise abstract container comonads in order to have enough structure to derive the complexity bounds?

Definition 11 (Abstract finite container comonad)**.** A comonad structure $(\mathsf{F}, \varepsilon, (-)^\dagger)$ is an *abstract finite container comonad* if there exists a finite container $S \lhd P$ such that $\mathsf{F} \cong [\![S \lhd P]\!]$.

The above definition is essentially that of an abstract finite container functor (Definition 7) but where F is a comonad. This is all that is needed to derive the following complexity bound; an isomorphic *directed* container structure on $S \lhd P$ (and thus a comonad on $[\![S \lhd P]\!]$) is not required, just that there is an isomorphic container $S \lhd P$.

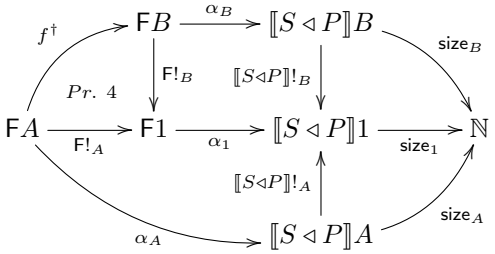
Theorem 4 (Abstract container comonad cost)**.** Let F be an abstract finite container with comonad structure $(\mathsf{F}, \varepsilon, (-)^\dagger)$ and a container $S \lhd P$ such that $\alpha : \mathsf{F} \cong [\![S \lhd P]\!]$. The size of abstract container values is given by $\mathsf{size}'_A : \mathsf{F}A \to \mathbb{N} = \mathsf{size}_A \circ \alpha_A$. The extension $(-)^\dagger$ then has the upper bound (assuming the cost $[f^\dagger]_n$ is indeed defined):

$$[f^\dagger]_n \in \mathcal{O}(n[f]_m + nmQ_n + R_n)$$

with $m \geq n$ where m represents the size of the largest subcontainer passed to f, Q_n represents costs for constructing subcontainers and valuation cost, and R_n covers pathological implementations.

Proof. The proof resembles that of Theorem 2 for abstract functors with the parameterisation of the complexity bound by Q_n and R_n. In contrast with the complexity bounds of abstract container functors (Proposition 3, p. 6), this result relies more on the general axioms of the structure on F.

1. By shape preservation of comonads (Proposition 4), the size of a container computed by the extension $f^\dagger$ is the size of the input container, *i.e.*, $\mathsf{size}'_B \circ f^\dagger = \mathsf{size}'_A$ by the following diagram (recall $\mathsf{size}'_A = \mathsf{size}_A \circ \alpha_A$), where the unlabelled commuting squares are naturality properties:



2. By the comonad axiom [C2] $\varepsilon_{\mathbb{N}} \circ \mathsf{size}'^{\dagger}_A = \mathsf{size}'_A$, the size of the subcontainer computed for the current context is the size of the input container (Remark 6, p. 10). Thus the maximum size m of a subcontainer computed by extension $(-)^\dagger$ is at least n, *i.e.* $m \geq n$.

3. The comonad axiom [C1] $\varepsilon^\dagger_A = id_{\mathsf{F}A}$ implies that for some input container $x : \mathsf{F}\,A$ and function $f : \mathsf{F}\,A \to B$ then $f^\dagger\, x : \mathsf{F}\,B$ applies f to n subcontainers of x where each subcontainer has each element of x at its current context.

   This follows by a contradiction: assume that $f^\dagger$ does not apply f to n subcontainers. Therefore, to compute an output container of size n (from point (1) above) of element type B, the operation $f^\dagger$ must copy some of the results from applying f. Therefore, applying $\varepsilon^\dagger_A$ to an input container with distinct elements A would produce an output container with duplicate elements. Therefore, $\varepsilon^\dagger_A \neq id_{\mathsf{F}A}$ and the initial assumption must be false.

   Combined with point (2), the execution time is therefore bounded above by $n[f]_m$ where m is the maximum size of subcontainers, plus nmQ_n to account for constructing subcontainers for each application of f and accessing elements (the valuation cost).

4. Finally, the term R_n accounts for arbitrary wasteful implementations. □

Optimisation 3. For any container comonad, axiom [C3] can be oriented as $(g \circ f^\dagger)^\dagger \rightsquigarrow g^\dagger \circ f^\dagger$ guaranteeing an asymptotic improvement or not making the complexity worse (in the case where Q_n or R_n dominate all other terms).

Proof. Essentially the same as the proof for Optimisation 2 (p. 11), modulo the additional term R_n which appears in both complexity terms, but with an additional linear factor for the (more costly) $(g \circ f^\dagger)^\dagger$. Thus, $[g^\dagger \circ f^\dagger]_n \in \mathcal{O}([(g \circ f^\dagger)^\dagger]_n)$. □

4.3. Optimisations for counit

The computational cost of the counit operation $\varepsilon_A : \mathsf{F}A \to A$ has yet to be considered. Its axioms [C1] $\varepsilon^\dagger \equiv id$ and [C2] $\varepsilon \circ f^\dagger \equiv f$ are clearly amenable to orientation as optimising rewrites. This is formalised briefly here, using similar approaches to the above.

Remark 7. Not much can be said about the computational complexity of ε as a function of its input. For a directed container $(S \triangleleft P, \downarrow, \mathsf{o}, \oplus)$, the derived comonad has counit $\varepsilon(s, v) = v\,\mathsf{o}\{s\}$. The execution time of ε is then the time taken to access the root position from the container value, $[\varepsilon]_n \in \mathcal{O}(Q_n)$, where $Q_n = [v]_1$ is unknown since it depends on the valuation function of the input container value. The same is true for the abstract case where there is no further information. However, axioms [C1] and [C2] can still be shown to be optimisations.

Optimisation 4 (Comonad [C1])**.** For all derived or abstract container comonads, axiom [C1] can be oriented as a rewrite $\varepsilon^\dagger \rightsquigarrow id$ providing an optimisation.

Proof. For derived container comonads $[\varepsilon^\dagger] \in \mathcal{O}(n[\varepsilon]_m + nmQ_n)$ and for abstract container comonads $[\varepsilon^\dagger] \in \mathcal{O}(n[\varepsilon]_m + nmQ'_n + R_n)$. Since $[id_F]_n \in \mathcal{O}(nQ_n)$ (see proof of Optimisation 1, p. 7) this rewrite is guaranteed to preserve asymptotic complexity or improve the complexity (if either $m > 1$ or $[\varepsilon]_m$ has non-constant complexity, or R_n dominates). □

Optimisation 5 (Comonad [C2])**.** For all derived or abstract container comonads, axiom [C2] can be oriented as a rewrite $\varepsilon \circ f^\dagger \rightsquigarrow f$ providing an optimisation

Proof. Let (s, v) be an input container value, with $n = |P\,s|$.

For derived container comonads $[\varepsilon \circ f^\dagger]_n \in \mathcal{O}(n[f]_m + nmQ_n + [\varepsilon]_n)$ where m is the size of the maximum subshape (Theorem 3, p. 10). By [S1] it follows that $m \geq n$ since the current shape is a subshape of itself (at the root position) and therefore the maximum subshape size must be at least n. Thus $[f]_n \in \mathcal{O}(n[f]_m)$ and therefore $[f]_n \in \mathcal{O}(n[f]_m + nmQ_n + [\varepsilon]_n)$.

For abstract container comonads, $[\varepsilon \circ f^\dagger]_n \in \mathcal{O}(n[f]_m + nmQ'_n + R_n + [\varepsilon]_n)$ (Theorem 4, p. 12). By similar reasoning to the derived case, the [S1] axiom corresponds to the general [C2] comonad axiom $\varepsilon \circ \mathsf{size}^\dagger = \mathsf{size}$, (see Remark 6, p. 10) that is, the size of the subshape at the current context is the size of the input container. Thus $m \geq n$ and it follows that $[f]_n \in \mathcal{O}(n[f]_m + nmQ'_n + R_n + [\varepsilon]_n)$.

Subsequently, in both the derived and abstract cases $\varepsilon \circ f^\dagger \rightsquigarrow f$ is an optimisation. □

4.4. Summary

The results of this section showed complexity bounds for derived and abstract container comonads, providing the proof that each of the comonad axioms can be oriented to give an optimisation. The rewrite $(g \circ f^\dagger)^\dagger \rightsquigarrow (g^\dagger \circ f^\dagger)$ (Optimisation 2 and 3) corresponds to the flattening transformation discussed in the introduction (Section 1), moving from a quadratic to linear program. We revisit this example concretely using the formalism of this paper.

Example 6 (Arrays)**.** The introductory example of this paper showed two imperative array programs with the same extensional behaviour, but differing asymptotic behaviours:

```
for i = 1..n
   for u = 1..n
      B(u) = f(A, u)
   C(i) = g(B, i)
```
$\in \mathcal{O}(n^2[\mathtt{f}] + n[\mathtt{g}])$

```
for i = 1..n
   B(i) = f(A, i)
for i = 1..n
   C(i) = g(B, i)
```
$\in \mathcal{O}(n[\mathtt{f}] + n[\mathtt{g}])$

where `A`, `B`, and `C` are one-dimensional arrays and `f` and `g` are pure functions. This example can be explained in the framework of this paper because the above traversals are instances of the extension operation of an *array comonad* [17] which has a *cursor*– a position which identifies or *focusses* on a particular element in the container (see [4, §4.5]). In the above loops, the iteration variables `i` and `u` act as cursors to the arrays. For example, in the left program, `u` is used as the cursor for `A` and `i` is used as the cursor for `B`.

One-dimensional arrays with a cursor are captured by the container:

$$\mathsf{Array} = \Sigma(l : \mathbb{N})(c : \mathsf{Fin}\, l) \lhd (\lambda(l, c).\mathsf{Fin}\, l)$$

Shapes are a dependent-sum of a length $l : \mathbb{N}$ and a *cursor position* (written as c) drawn from the finite set $\mathsf{Fin}\, l$. Positions for shapes with length l are drawn from the finite set $\mathsf{Fin}\, l$. The generated container data type[2] is then $[\![\mathsf{Array}]\!]A = \Sigma(l : \mathbb{N}, c : \mathsf{Fin}\, l).(\mathsf{Fin}\, l \to A)$, that is, a dependent-sum of the length l of the array, the cursor c into the array, and the valuation map from positions within the length to values.

The Array container also has the following directed structure:

$\mathsf{o}\{(l, c)\}$	$= c$	*the root position is marked by the cursor*
$(l, c) \downarrow p$	$= (l, p)$	*the subshape at position p has same length, but with cursor p*
$p \oplus p'$	$= p'$	*the offset of position p to p' is just the cursor p'*

Inlining the above definitions, the derived extension operation of the comonad for this directed container is defined as:

$$f^\dagger((l, c), v) = ((l, c),\ \lambda p.\ f\ ((l, p), v))$$

That is, extension $(-)^\dagger$ produces an array with the same size and cursor as the input array, where elements of the new array are computed by applying f to arrays $((l, p), v)$ which are the original array "refocussed" at each position p by setting p as the cursor. This produces the following behaviour captured by the imperative code:

```
for p = 0..(l-1)
   B(p) = f(A, p)
```

where `A` corresponds to the valuation and `p` to the position which is used as the cursor within the array (of length `l`). The output array `B` corresponds to the new valuation. Thus, the two array programs in the introductory example correspond to two comonadic array container computations: $(\mathtt{g} \circ \mathtt{f}^\dagger)^\dagger$ and $\mathtt{g}^\dagger \circ \mathtt{f}^\dagger$ respectively.[3]

Applying Theorem 3 on the derived container comonad cost and Corollary 3 on the cost of post-composing functions after comonadic extension, we recover the complexity results shown in

[2] Following Definition 2 exactly, the data type is $[\![\mathsf{Array}]\!]A = \Sigma(s : \Sigma(l : \mathbb{N})(\mathsf{Fin}\, l))((\lambda(l, c).\mathsf{Fin}\, l)s \to A)$. The text provides an (isomorphic) simplification.

[3] Whilst the elements of $\mathsf{Fin}\, l$ range from 0 to $l - 1$, it is easy to translate these to match the index space of the original example (with indices drawn from 1 to n, where $l = n$ here).

the introduction (with some additional precision). By the above definition of the array comonad, the size of largest subshape $m = \mathsf{max}_{p:P\,s}|P\,(s \downarrow p)| = |P\,s| = n$ since subshaping $\downarrow$ just changes the position of a cursor rather than changing the size of an array. Thus, assuming that the valuation cost is constant $Q_n = 1$, we derive the complexity bounds:

$$\begin{aligned}(\mathtt{g} \circ \mathtt{f}^\dagger)^\dagger &\in \mathcal{O}(n[\mathtt{g} \circ \mathtt{f}^\dagger]_n + n^2 Q_n) \\ &\in \mathcal{O}(n([\mathtt{g}]_n + n[\mathtt{f}]_n) + n^2 Q_n) + n^2 Q_n) \\ &\in \mathcal{O}(n\mathtt{g}_n + n^2[\mathtt{f}]_n + n^3 Q_n + n^2 Q_n) \\ &\in \mathcal{O}(n[\mathtt{g}]_n + n^2[\mathtt{f}]_n + n^3)\end{aligned} \qquad \begin{aligned}\mathtt{g}^\dagger \circ \mathtt{f}^\dagger &\in \mathcal{O}(n[\mathtt{g}]_n + n^2 Q_n + n[\mathtt{f}]_n + n^2 Q_n) \\ &\in \mathcal{O}(n[\mathtt{g}]_n + n[\mathtt{f}]_n + n^2)\end{aligned}$$

On the right, the extension traversals are sequentially composed, whereas on the left the extension of $\mathtt{f}$ is nested within the traversal for the extension of $\mathtt{g}$, leading to the quadratic factor on the cost of $\mathtt{f}$. Applying Optimisation 3 (p. 13) to this program rewrites $(\mathtt{g} \circ \mathtt{f}^\dagger)^\dagger$ to $\mathtt{g}^\dagger \circ \mathtt{f}^\dagger$, thus reducing the asymptotic complexity. The nested array-traversal optimisation discussed in the introduction is therefore an example of structuring a program via a comonad, and optimising by applying the [C3] axiom as a rewrite rule.

In a programming, the kind of nesting seen in this example is more likely to occur when composing library functions than when writing straight-line imperative code by hand. Indeed, existing comonadic libraries sometimes suffer from complexity problems due to overly nested operators, for example in the comonadic notation for Haskell which dualises **do**-notation [18] and as noted in the work of Foner on fixed-points over comonadic extension and comonadic arrays [7]. By adding a [C3] rewrite rule into the compiler, this situation can be remedied.

5. Related work

Related to the notion of containers (à la Abbott *et al.*) are *shapely datatypes* [12, 14], which were shown to be a subset of containers with finite cardinals [1]. Shapely data types have a *shape* morphism $sh : FX \to F1$ (which can be given by $F!_A$) and *data* morphism $data : FX \to [X]$, mapping to a list of values. A shapely data type (or functor) is then the pullback of these two morphisms. Size can be determined from the data morphism using the length of the list, *e.g.*, $size = length \circ data$. Thus, our results for abstract constructions can also be applied also to shapely functors.

Various works have provided a cost analysis for algorithmic skeletons in the context of parallel programming *e.g.* [15, 10, 22, 13], which have similar motivation to this work: reasoning about performance abstractly to inform program optimisation. Riely and Prins prove that the flattening transformation (for nested parallelism) provides an improvement in their costing [20]. Size inference for concrete list data structures has been used for optimising compilation [13].

The work of Hayashi leverages a concrete cost-model for parallel architectures (accounting for memory model, communication cost) in the context of a vector-based set of algorithmic skeletons [10]. There are many possible cost models available (Hayashi's thesis gives a survey of cost models and different cost analysis techniques [10]). However, many of these costings are specialised to particular data structures, *e.g.*, lists or vectors, and focus more on the distribution and communication cost, rather than a general computational complexity. The analysis here is much more abstract, further divorced from any implementation (sequential or parallel). More abstract approaches have costed parallel programs over shapely functors [12]. The work of Jay considers map, fold, and zip constructs in the more abstract setting [13]. The present paper broadens this to the larger class of containers and studied the comonad pattern, which has not been given a complexity analysis in the literature.

Skillicorn and Cai provide a cost-model for Bird-Meertens algebra-of-programming style [22]. They define the (sequential) cost of the *map* operation on lists as $t(map\ f)_n = nt(f)_n$, and of

concat (*i.e.*, multiplication of the list monad) as $t(concat)_n = n-1$. Thus, the associative axiom of the list monad $t(concat \circ (map\ concat))_n = n(n-1) + n - 1 = n^2 - 1$ versus $t(concat \circ concat)_n = n - 1 + n - 1 = 2(n-1)$. This justifies the reassociating transformation of a list monad. Again, a particular data structure is studied.

6. Discussion and conclusion

Infinite containers. We have considered models of finite, discrete containers in this paper, since our asymptotic bounds are defined for finite cardinals. Can similar reasoning principles be used to explain asymptotic behaviour of infinite containers? One potential avenue for infinite but *discrete* containers is to define bounds in terms of ordinals.

Example 2 defined the infinite stream container $\{*\} \lhd (const\ \mathbb{N})$. The asymptotic behaviour might be reasoned about via ordinals, where ω (linear, infinite) and ω^2 (quadratic, infinite) can be distinguished. This provides a way to explain the overhead incurred when nesting the extension operation for stream comonads.

Refining bounds. The complexity bounds for comonadic extension appealed to the notion of the maximum size for the subshapes of a container. Many examples of directed containers however exhibit the following property:

$$\forall s : S, p : P\,s\ .\ |P\,(s \downarrow p)| \leq |P\,s|$$

That is, for all shapes, the size of each of its subshapes is no larger than the size of its parent shape. If this property holds for a directed container, the complexity of $f^\dagger$ can be refined from $[f^\dagger]_n \in \mathcal{O}(n[f]_m + nQ_n)$ where $m = \mathsf{max}_{p:P\,s}|P\,(s \downarrow p)|$ (largest subshape size) to $[f^\dagger]_n \in \mathcal{O}(n[f]_n + nQ_n)$ (note the n subscript rather than m) thus obviating the additional calculation and reasoning behind m. It may be possible to prove this result from just the directed container axioms, but a proof has yet to be found, even though all reasonable examples satisfy it. This is future work.

The abstract term R_n was introduced to capture the cost of any potential wasteful computation involved in the operations of an abstract container functor or comonad. In a simply- (or dependently-) typed λ-calculus model, the possibility of this additional wasteful computation cannot be ruled out from looking at the types and axioms of the structures alone. However, under a linear typing discipline, such waste is constrained to constant overheads. Exploring linear and bounded-linear variants of the definitions here to give tighter bounds is further work.

In practice, reasonable container functor and comonad implementations do not have any additional cost that is not accounted for by the valuation costing Q_n. Thus $R_n = 0$ in standard, common cases. Furthermore, the valuation cost Q_n is typically bounded above by n, corresponding to the time to index elements out of a container of size n. The author's doctoral dissertation provides a catalogue of different container comonads [16, §3.2], all exhibiting $R_n = 0$ and $Q_n \in \mathcal{O}(n)$.

Monads. Ahman *et al.* sketched the additional container structure required to generate container monads [3], and later gave the full details of the requisite container algebra [24]. Monads can be seen as generalising a functor's traversal in a dual way to comonads, capturing *local substitution* of elements for substructures (akin to a *scatter* [17]). This is a frequent pattern in container programming. Further work is to derive complexity bounds for container monads following the scheme of this paper. Initial investigation suggests that deriving the complexity information is straightforward and leads to related bounds and results. For example, the axiom $(g^* \circ f)^* = g^* \circ f^*$ for the Kleisli extension of a monad $(-)^*$ can be oriented left-to-right as an optimising rewrite. This matches the well-known problem of nested monadic computations, addressed in various approaches for reifying a computation tree to reassociate monadic computations, *e.g.*, [11, 25].

Category theory models of complexity. Various works seek to give a category theoretic account of complexity classes, algorithms, and wastefulness (inefficiency), *e.g.*, [28, 5]. An interesting future direction would be to consider containers on the category of *timed sets*, giving a measure of the execution time of morphisms and complexity measures from within the model, rather than externally via a cost function as done here. This is further work.

Relatedly, it may be possible that the account of execution cost and complexity for comonads here (and in the future for monads) could be reduced into more basic structures such as adjunctions, and studied from a categorical perspective. This is further work.

Concluding remarks. The analysis in the paper provides a foundation for program optimisation for programs structured in terms of containers, functors, and comonads. These are already common programming idioms in functional programming. When combined with user-defined rewrite rules, this provides a powerful technique for suggesting optimisations. Furthermore, by instantiating the execution time function $[-]_n$ the parameterised cost model $[-]$ provides a way to specialise the results to particular precise, concrete abstract machine models.

It should be pointed out that, whilst these optimisations are proven to not make a program asymptotically worse, this is still a fairly weak statement. It may be the case that such rewrites interact poorly with other transformations in a compiler, leading to worse performance. As always with optimising compilation, care must be taken.

Containers are a useful syntax and abstraction for exploring generic programming interfaces for common data types. There are still plenty of avenues for further exploration. This paper demonstrated that container abstractions can be used to reason about efficiency, which has not appeared before, and for which there is much more work to be done.

Acknowledgments. Thank you to the anonymous reviewers of this special issue who greatly improved the precision of this work with their comments. Thanks also to Stephen Dolan for helpful comments on an early version, Matthew Anderson, Michael Gale, Tomas Petricek and Radu Grigore for discussion, comments by Alan Mycroft on an early draft, and Marcin Jurdinski for listening to my initial idea on a bus in Riga. Thanks also to the participants at DICE'2015 for their feedback on an extended abstract of this work. This work was partially supported by EPSRC EP/K011715/1 (whilst the author was a Research Associate at Imperial College) and EPSRC EP/M026124/1.

[1] M. Abbott, T. Altenkirch, and N. Ghani. Categories of containers. In *Foundations of Software Science and Computation Structures*, pages 23–38. Springer, 2003.

[2] M. Abbott, T. Altenkirch, and N. Ghani. Containers: Constructing Strictly Positive Types. *Theoretical Computer Science*, 342(1):3–27, 2005.

[3] D. Ahman, J. Chapman, and T. Uustalu. When is a container a comonad? *Foundations of Software Science and Computational Structures*, pages 74–88, 2012.

[4] D. Ahman, J. Chapman, and T. Uustalu. When is a container a comonad? (extended version). *Logical Methods in Computer Science (LMCS)*, 10(3), 2014.

[5] R. Cockett, J. Díaz-Boïls, J. Gallagher, and P. Hrubeš. Timed sets, functional complexity, and computability. *Electronic Notes in Theoretical Computer Science*, 286:117–137, 2012.

[6] U. Dal Lago and S. Martini. An invariant cost model for the lambda calculus. In *Logical Approaches to Computational Barriers*, pages 105–114. Springer, 2006.

[7] Kenneth Foner. Functional pearl: getting a quick fix on comonads. In *ACM SIGPLAN Notices*, volume 50, pages 106–117. ACM, 2015.

[8] J.-Y. Girard. Normal functors, power series and λ-calculus. *Annals of pure and applied logic*, 37(2):129–177, 1988.

[9] J. Havel and A. Herout. Rendering Pipeline Modelled by Category Theory. In *GraVisMa 2010 workshop proceedings*, pages 101–105. University of West Bohemia in Pilsen, 2010.

[10] Y. Hayashi. Shape-based cost analysis of skeletal parallel programs. 2001.

[11] R. Hinze. Kan extensions for program optimisation or: Art and Dan explain an old trick. In *Mathematics of Program Construction*, pages 324–362. Springer, 2012.

[12] C Barry Jay. A semantics for shape. *Science of computer programming*, 25(2):251–283, 1995.

[13] C Barry Jay. Costing parallel programs as a function of shapes. *Science of Computer Programming*, 37(1):207–224, 2000.

[14] C Barry Jay and J Robin B Cockett. Shapely types and shape polymorphism. In *Programming Languages and Systems – ESOP'94*, pages 302–316. Springer, 1994.

[15] R. Lechtchinsky, M. Chakravarty, and G. Keller. Costing nested array codes. *Parallel Processing Letters*, 12(02):249–266, 2002.

[16] D. Orchard. Programming contextual computations. Technical Report UCAM-CL-TR-854, University of Cambridge, 2014. PhD dissertation, `http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-854.pdf`.

[17] D. Orchard, M. Bolingbroke, and A. Mycroft. Ypnos: declarative, parallel structured grid programming. In *DAMP '10: Proceedings of the 5th ACM SIGPLAN workshop on Declarative aspects of multicore programming*, pages 15–24, New York, NY, USA, 2010. ACM.

[18] D. Orchard and A. Mycroft. A Notation for Comonads. In *IFL '12: Implementation and Application of Functional Languages, Revised Selected Papers*, volume 8241, 2012.

[19] S. Peyton Jones, A. Tolmach, and T. Hoare. Playing by the rules: rewriting as a practical optimisation technique in GHC. In *Haskell Workshop*, volume 1, pages 203–233, 2001.

[20] J. Riely and J. Prins. Flattening is an improvement. In *Static Analysis*, pages 360–376. Springer, 2000.

[21] D. Skillicorn. Models for practical parallel computation. *International Journal of Parallel Programming*, 20(2):133–158, 1991.

[22] D. Skillicorn and W. Cai. A cost calculus for parallel functional programming. *Journal of Parallel and Distributed Computing*, 28(1):65–83, 1995.

[23] T. Uustalu and V. Vene. Comonadic Notions of Computation. *Electron. Notes Theor. Comput. Sci.*, 203(5):263–284, 2008.

[24] Tarmo Uustalu. Container combinatorics: Monads and lax monoidal functors. In *International Conference on Topics in Theoretical Computer Science*, pages 91–105. Springer, 2017.

[25] J. Voigtländer. Asymptotic improvement of computations over free monads. In *Mathematics of Program Construction*, pages 388–403. Springer, 2008.

[26] P. Wadler. Strictness analysis aids time analysis. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 119–132. ACM, 1988.

[27] P. Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical computer science*, 73(2):231–248, 1990.

[28] N. S Yanofsky. Towards a Definition of an Algorithm. *Journal of Logic and Computation*, 21(2):253–286, 2011.