Complexity bounds from abstract categorical models

Extended abstract*

Dominic Orchard
Imperial College London

Abstract

Common category theory notions are useful for abstracting both semantics and programs, simplifying definitions and aiding reasoning. We argue that categorical descriptions can also be used to reason about program complexity. We calculate complexity bounds from the axioms of abstract category theory structures commonly used in programming and semantics, which imply opportunities for optimisation. We study *functors* and *comonads* in the context of programming with (finite) containers. Due to the abstract interface provided by the language of category theory, the inferred implicit complexity bounds and subsequent optimisations they imply are implementation agnostic (machine free).

1 Introduction

Consider the standard higher-order *map* function on lists, with polymorphic type $map : \forall\alpha, \beta.(\alpha \to \beta) \to [\alpha] \to [\beta]$, where $map\, f$ applies f to every element of the input list exactly once to build the result list of the same length, *e.g.*, $map\ (*2)\ [1, 2, 3, 4] = [2, 4, 6, 8]$. This behaviour is captured by two axioms $map\, id \equiv id$ and $map\,(g \circ f) \equiv map\, g \circ map\, f$. In this paper, we show that from just the axioms and parametricity of *map* we can infer its lower-bound complexity (over finite inputs). Using the notation $[f]_n$ for the execution time of a unary function f on some input list of size n, the lower-bound asymptotic complexity of *map*'s execution time is:

$$[map\, f]_n \in \Omega(n[f]_1) \tag{1}$$

i.e., *map* at least visits each element of the input list, applying f at each. We explain a generalisation of this result for the class of functors modelling discrete, finite *containers* (*e.g.*, lists, arrays, trees, graphs), which might be used for a language semantics or for structuring programs directly. The above lower bound is not particularly useful on its own, but can be used to turn program equalities into rewrite rules that make asymptotic improvements when combined with complexity bounds of other related functions and equations involving *map*.

More formally, given an equality $f \equiv g$ between two program fragments f and g, if $[f]_n \in \Omega([g]_n)$ (or $[g]_n \in \mathcal{O}([f]_n)$) 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.

Context This work sprang from modelling and optimising numerical algorithms abstracted over the underlying data structures. We study functors since they capture traversals over a data structure (*e.g.*, transforming every element of an array) [5]. We also study comonads since they generalise a functor's traversal, capturing *local transformations* which may depend on an element and its neighbours [6, 3], *e.g.*, convolutions, fluid simulations, and the Game of Life. We briefly introduce this view of

*Presented at DICE'15. Draft last updated 11^{th} April 2015

comonads below. Section 2 and Section 3 then consider optimisations on functor and comonad operations respectively.

A model of finite containers We abstract finite containers and useful data processing operations over them via the following structures:

- a base category of computation $\mathbb{C}$, whose objects we essentially consider to be program types and morphisms as (total) functions;
- a functor $\mathsf{F} : \mathbb{C} \to \mathbb{C}$ modelling a container data type (by the object mapping) and element-wise traversal (by the morphism mapping), satisfying the functor axioms:

$$\text{[F1]}\ \ \mathsf{F}id \equiv id \qquad \text{[F2]}\ \ \mathsf{F}(g \circ f) \equiv \mathsf{F}g \circ \mathsf{F}f$$

- a natural transformation $\mathsf{size}_A : \mathsf{F}A \to \mathbb{N}$, *i.e.*, for every object $x : \mathsf{F}A$ there is a finite cardinal counting the number of A elements in x;
- a natural transformation $\eta_A : A \to \mathsf{F}A$ modelling constant *promotion* (*e.g.*, mapping from a scalar to a vector);
- a comonad structure $(\mathsf{F}, \varepsilon, (-)^\dagger)$ which comprises
  - the *counit* natural transformation $\varepsilon_A : \mathsf{F}A \to A$
  - the *extension* operation, mapping a morphism $f : \mathsf{F}A \to B$ to morphism $f^\dagger : \mathsf{F}A \to \mathsf{F}B$ satisfying the axioms:

$$\text{[C1]}\ \ \varepsilon^\dagger \equiv id \qquad \text{[C2]}\ \ \varepsilon \circ f^\dagger \equiv f \qquad \text{[C3]}\ \ g^\dagger \circ f^\dagger \equiv (g \circ f^\dagger)^\dagger$$

Comonads generalise the element traversal of functors by allowing a transformation to depend not just on a single element, but on an element and its neighbours. Compare the signatures of morphism mapping and comonadic extension:

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

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, it is a container 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 [3]). For example, the kernel function of a Gaussian blur takes the mean of an element at the current context and its immediate neighbours.

Extension lifts (extends) a morphism $f : \mathsf{F}A \to B$ modelling 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 at every possible index in an array. The counit operation $\varepsilon_A : \mathsf{F}A \to A$ defines the notion of a *current context*, at which there is an element which is projected out. This acts as the identity for extension (see [C1]).

The notion of a *container* can be formalised as parametric data types that contain only strictly positive occurrences of the parameter type. An alternate characterisation is that containers comprise a set of *shapes*, a mapping from shapes to *positions*, and a mapping from positions to values [1]. We consider such data types which have additional monoidal structure on positions (allowing a form of 'navigation' through

the data type) which characterises these containers as comonads [2]. We consider those that have a finite set of positions (finite size). We elide the details for brevity, but example functors/comonads which the reader may call to mind include lists, trees, and arrays, each paired with a cursor index which points to an element within.

Structured size A final word on notation. The lower-bound complexity of *map* is conservative. It assumes the execution time of f is independent of the element sizes in the input. Consider however nested *map*s, *e.g.* $map\ (map\ (*2))\ [[1,2,3],[4,5],[6,7]] = [[2,4,6],[8,10],[12,14]]$ where each element in the input list is a list of at least size two.

We thus define a notation for sizes of structured data (*structural sizes*). A *regular* (two-layer, or two-dimensional) structure with size n and elements all of size m has the size term $n[m]$, equivalent to an overall size nm. An *irregular* (two-layer) structure with size n and element size *bounded below* by m has structural size term $n[\Omega(m)]$, *i.e.*, the size of inner elements has lower bound m.

We thus generalise our lower bound on *map* to: $[map\, f]_{n[\Omega(m)]} \in \Omega(n[f]_m)$.

2 Functors and promotion

Functors generalise the notion of the list data type along with its *map* function to arbitrary data types F with an analogous notion of *mapping* a function over the elements of a F value. The lower-bound complexity of *map* (1) applies analogously.

Proposition 1. For any discretely finite container F, the morphism mapping operation has lower bound complexity $[\mathsf{F}f]_{n[\Omega(m)]} \in \Omega(n[f]_m)$.

Proof. (sketch)

1. Naturality of size for the functor F means $\mathsf{size}_B \circ \mathsf{F}f = \mathsf{size}_A$ for all $f : A \to B$. Thus, the output size of $\mathsf{F}f$ is that of its input size.

2. By functor law [F1] $\mathsf{F}id \equiv id$, every element of the $\mathsf{F}A$ is visited and has f applied (thus, *e.g.* f is not just applied to one element which is then copied n times);

3. Parametericity in A, B of the morphism mapping $f : A \to B$ to $\mathsf{F}f : \mathsf{F}A \to \mathsf{F}B$ implies $\mathsf{F}f \not\equiv id$, since every A must be transformed to a B. Parametricity also implies we cannot decide the equality $f = id$, so a constant time implementation cannot be interposed for $map\, id$.

$\therefore \mathsf{F}f$ at least applies f to every element of its input, hence $[\mathsf{F}f]_{n[\Omega(m)]} \in \Omega(n[f]_m)$. □

Promotion The promotion operation $\eta_A : A \to \mathsf{F}A$ may, for example, map a value to a singleton list or lift a value to a constant vector. Since η is natural, it is fully parametrically polymorphic in its parameter type and therefore cannot deconstruct or inspect its parameter. The complexity of η is thus independent of its input size, *i.e.*,

$$[\eta]_n \in \mathcal{O}(1) \quad \text{and} \quad [\eta]_n \in \Omega(1) \tag{2}$$

Furthermore, this means the resulting $\mathsf{F}A$ value is of a constant size, created from copies of the input. The complexity of $\mathsf{F}f \circ \eta_A$ is therefore solely dependent on the complexity of f since only a constant sized container is constructed by η. That is:

$$\exists m > 0.\ [\mathsf{F}f \circ \eta]_n \in \Omega(m[f]_n) \quad \therefore\ [\mathsf{F}f \circ \eta]_n \in \Omega([f]_n) \tag{3}$$

where $[f]$ is parameterised by input size n since η does not interfere with the input. The two sides of the naturality axiom for η ($\mathsf{F}f \circ \eta \equiv \eta \circ f$) therefore have the same lower-bound complexities, and the upper-bound complexity of the right-hand side is completely determined by the complexity of f:

$$[\mathsf{F}f \circ \eta]_n \in \Omega([f]_n) \qquad [\eta \circ f]_n \in \Omega([f]_n) \in \mathcal{O}([f]_n) \tag{4}$$

This does not provide enough information for guaranteeing any asymptotic optimisation by using naturality as a rewrite. There is however a constant factor improvement by m (the size of the container created by η) due to (3), and in practice the optimisation $\mathsf{F}f \circ \eta \rightsquigarrow \eta \circ f$ is prudent. Indeed, this transformation formed the basis for an optimisation step used in an automatic vectoriser for Haskell [4].

3 Comonads

Recall that the extension operation of a container comonad can be seen as generalising the traversal provided by its functor (Section 1). We apply similar techniques to the previous section to infer optimising rewrite rules for extension.

We mainly consider 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 kind of nesting can easily arise during program construction and can lead to significant performance issues. Therefore, it would be preferable if it could be automatically eliminated (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. Firstly, we establish a lower-bound for extension in a similar way to functors.

Proposition 2. For any comonad over a finite container F, *extension* $(-)^\dagger$ has lower-bound complexity $[f^\dagger]_n \in \Omega(n[f]_1)$.

Proof. (Sketch) By [C1] $\varepsilon^\dagger \equiv id$ it follows that extension is *size preserving*. That is, for any $f : \mathsf{F}A \to B$ then $\mathsf{size}_{\mathsf{F}B} \circ f^\dagger = \mathsf{size}_{\mathsf{F}A}$ as follows (we abbreviate size to s):

$$\mathsf{s} \circ f^\dagger \overset{nat}{\equiv} \mathsf{s} \circ \mathsf{F}!_B \circ f^\dagger \overset{ext}{\equiv} \mathsf{s} \circ (!_B \circ f)^\dagger \overset{!_A}{\equiv} \mathsf{s} \circ (!_A \circ \varepsilon)^\dagger \overset{ext}{\equiv} \mathsf{s} \circ \mathsf{F}!_A \circ \varepsilon^\dagger \overset{nat}{\equiv} \mathsf{s} \circ \varepsilon^\dagger \overset{[C1]}{\equiv} \mathsf{s}$$

where the step *ext* follows from $\mathsf{F}f \equiv (f \circ \varepsilon)^\dagger$ (proof not shown), $!_A$ is terminality, and *nat* is size naturality. Thus the size of the input is the size of the output after extension ([3] has a related proof). Further, [C1] coupled with parametricity of $(-)^\dagger$ implies that f is applied n times, to compute each new B element. $\therefore [f^\dagger]_n \in \Omega(n[f]_1)$. □

Corollary 1. Size preservation for comonads implies $[g \circ f^\dagger]_n \in \mathcal{O}([g]_n + [f^\dagger]_n)$. That is, the upper-bound complexity of post-composing a morphism g with the extension $f^\dagger$ is the sum of their complexities each parameterised by the input size n since (by size preservation of extension) g is passed a container of size n.

The bound of Proposition 2 does not provide enough information to orient [C3] rule as an optimising rewrite rule. However, following similar reasoning, a general upper-bound can be given, revealing the quadratic difference between the two sides of [C3].

Proposition 3. There exists terms P_n and $Q_n \geq 1$, parameterised by n, such that:

$$[f^\dagger]_n \in \mathcal{O}(P_n + n\,Q_n[f]_n) \tag{5}$$

where P_n accounts for time traversing the container to reach the leaves (the elements) and Q_n accounts for any extraneous applications of f beyond the linear (in n) use.

Proof. (sketch) By [C2] $\varepsilon \circ f^\dagger \equiv f$ it holds that $\varepsilon_{\mathbb{N}} \circ \mathsf{size}^\dagger_A \equiv \mathsf{size}_A$. This means the size of a container passed to a morphism being extended is the same size as that of the parameter container at the current context. By [C3], this generalises to every context, *i.e.*, extension is size preserving at all contexts, by considering a morphism $\mathsf{sum} : \mathsf{F}\mathbb{N} \to \mathbb{N}$ which sums natura number elements in a container. Then [C3] requires:

$$(\mathsf{sum} \circ \mathsf{size}^\dagger_A)^\dagger \quad \overset{[C3]}{=} \quad \mathsf{sum}^\dagger \circ \mathsf{size}^\dagger_A$$

This axiom is violated if size passes larger that n contexts to its parameter function.

Thus, the upper-bound complexity of extension of f is in the order of traversing to the leaves P_n (a function on n) summed with at least n applications of f (Proposition 2) to a container of at most size n (by the above size preservation argument). □

This upper bound is quite general (with P_n, Q_n) but it allows the following result:

Proposition 4. 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 (if, for example, term P_n dominates all other terms).

Proof. Following from Corollary 1 and Proposition 3 then:

$$\begin{aligned}
[g^\dagger \circ f^\dagger]_n &\in \mathcal{O}(P_n + nQ_n[g]_n + nQ_n[f]_n) \\
[(g \circ f^\dagger)^\dagger]_n &\in \mathcal{O}(P_n + nQ_n([g]_n + P_n + nQ_n[f]_n)) \\
&\in \mathcal{O}(P_n + nQ_n[g]_n + (nQ_n)^2[f]_n + nQ_nP_n)
\end{aligned}$$

□

Note on linearity Linear types could be used to constrain the number of times extension applies its parameter function, giving a more precise upper bound. For example, in a language with bounded linear types and size-indexed containers, $(-)^\dagger$ might be embedded with the type signature:

$$(-)^\dagger : !_n(!_1\mathsf{F}_m A \to B) \to (\mathsf{F}_n A \to \mathsf{F}_n B)$$

(where $!_1\mathsf{F}_m A \to B$ is equivalent to $\mathsf{F}A \multimap B$) constraining the upper-bound complexity of extend to $[f^\dagger]_n \in \mathcal{O}(P_n + n[f]_n)$. This is a reasonable constraint following from the size preservation property of comonads: we only need to apply f to each element.

Conclusions By simple arguments following from the axioms of categorical structures, and parametricity, we derived *complexity bounds for free*, which allowed us to oriented equalities as rewrite rules that provide optimisations (or do not make things worse). These complexity bounds are implicit and implementation agnostic. Further work is to fully formalise the proof sketches here, making use of parametricity theorems and formal characterisation of containers (such as by Abbott et al. [1]).

Acknowledgments Thanks to Stephen Dolan for many helpful comments, Matthew Anderson, Michael Gale and Tomas Petricek for discussion, comments by Alan Mycroft on an early draft, and Marcin Jurdinski for listening to my initial idea on a bus. This work was partially supported by EPSRC EP/K011715/1.

References

[1] M. Abbott, T. Altenkirch, and N. Ghani. Containers: constructing strictly positive types. *Theoretical Computer Science*, 342(1):3–27, 2005.

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

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

[4] Leaf Petersen, Dominic Orchard, and Neal Glew. Automatic SIMD Vectorization for Haskell. In *Proceedings of the 18th ACM SIGPLAN international conference on Functional programming*, pages 25–36. ACM, 2013.

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

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