The Boom Hierarchy

Alexander Bunkenburg*
Computing Science Department, University of Glasgow,
Glasgow, Scotland

Abstract

The Boom Hierarchy is the family of data structures *tree, list, bag, set*. By combining their properties in other ways, more data structures can be made, like mobiles. The paper defines the data structures of this extended Boom Hierarchy and shows how the functions *reduce, map, and filter* are applied to them.

1 Introduction

The Boom Hierarchy is the family of data structures *tree, list, bag, set*, to be used with the higher-order Squiggol functions *reduce, map, filter*.
Example
The term that filters the odd numbers from the list [1..10], and adds up their squares is

$$+/ \circ \mathsf{sqr}^* \circ \mathsf{odd} \triangleleft \;\; . \; [1..10].$$

(Reduce /, map *, and filter ◁ are defined further down.)
end of example
In this paper the data structures are presented as free algebras. New data structures in the family (e.g. mobiles) spring from algebras with new combinations of laws. The relations between the data structures are explained and some sample data structures. No category theory required!

2 The Boom hierarchy

The hierarchy of data structures that [Mee86] attributes to H. J. Boom comprises four data structures: *tree, list, bag,* and *set*. It is a fitting coincidence that Dutch "boom" means "tree", and *trees* are in the hierarchy. English "boom" meaning "pole" reminds of *lists* in the same way.
A data structure value is

- either [ ], the empty value containing no elements,
- or $[a]$, the singleton containing one element,
- or $l +\!\!+ r$, the join of two values.

*Supported by a postgraduate research studentship from the Science and Engineering Research Council

For nonempty data structures [ ] is excluded. *This notation is used for all data structures, not just lists.* Hopefully cutting away syntactic differences will expose the semantic similarities and differences between data structures more clearly.

Each data structure is the free algebra of its binary operation $+\!\!+$. The algebras (and therefore the data structures) differ in the laws they satisfy. More laws to an algebra mean less structural information in the data structure. The four laws of a binary operation that we'll consider are:

$$\begin{array}{rcll} a \otimes 1_\otimes = & a & = 1_\otimes \otimes a & \text{UNIT} \\ (a \otimes b) \otimes c & = & a \otimes (b \otimes c) & \text{ASSOC} \\ a \otimes b & = & b \otimes a & \text{COMM} \\ a \otimes a & = & a & \text{IDEM} \end{array}$$

Let's talk about these properties in shorthand. A binary operation has properties $a_1a_2a_3a_4$ means that if a_i is 1 then it satisfies the i'th property, and if a_i is 0, then it doesn't. Addition for instance has properties 1110; it has a unit, is associative and commutative, but not idempotent. The join operations of the data structures *tree, list, bag, set* have properties 1000, 1100, 1110, and 1111.

Each set of properties specifies a variety of algebras. The properties 1100 are the variety monoid for example.

Adding a law to an algebra can be thought of as partitioning the carrier of the algebra into equivalence classes induced by that law, and regarding each class as one element. Partitioning all lists by commutativity puts the lists $[1, 2], [2, 1]$ into the same class, the class representing the bag $[1, 2]$. Each class will in general have many elements, therefore mapping an element to its class is a function, but mapping a class to one of its elements involves the choice of which element to take. Let's call data structure A *higher* than data structure B if A's laws are a subset of B's laws. The shorthand makes this relation obvious: The data structure with variety $a_1a_2a_3a_4$ is higher than the one with variety $b_1b_2b_3b_4$ if each $a_i \leq b_i$. Downward mappings (from lists to bags for instance) are easy, and can safely be implicit, but upward mappings (from sets to bags for instance) involve a choice, and therefore have to be written.

Figure 1 gives examples how different values of a higher data structure are translated downwards to the same value in a lower data structure. Two useful choices for upward translations are there too: *setToBag* takes a set to the bag that contains each element of the set once, and *sort* takes a a bag to the list with the same elements in order (*sort* obviously only makes sense if the type of the elements is ordered).

Figure 2 is a picture of the Boom hierarchy data structures. There's a free algebra (and therefore a data structure) for all 16 possible combinations of the above four laws. Each data structure is identified by the properties of its join-operation, and by name. The abbreviations "n." and "id." stand for "nonempty" and "idempotent". The original Boom hierarchy data structures plus *mobiles* are the diamond with tail on the left. Their nonempty versions form a diamond with tail in the middle. On the right the idempotent versions repeat the two diamonds. Their tails *idempotent bags* are just *sets*. The idempotent structures (1 in fourth position) exist; but *sets* seem the only useful ones. However guaranteed non-empty structures (0 in first position) are useful. If all edges of the *higher-than* relation are drawn, the picture becomes a 4d-cube.

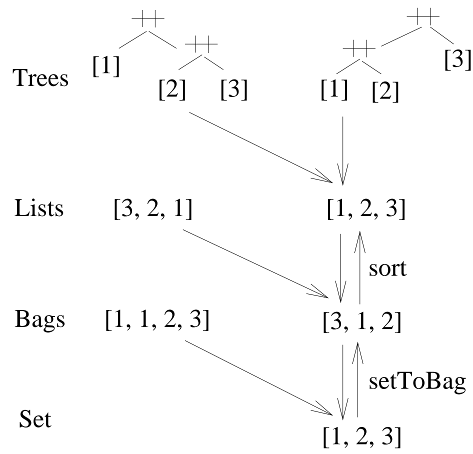

Figure 1: Some translations

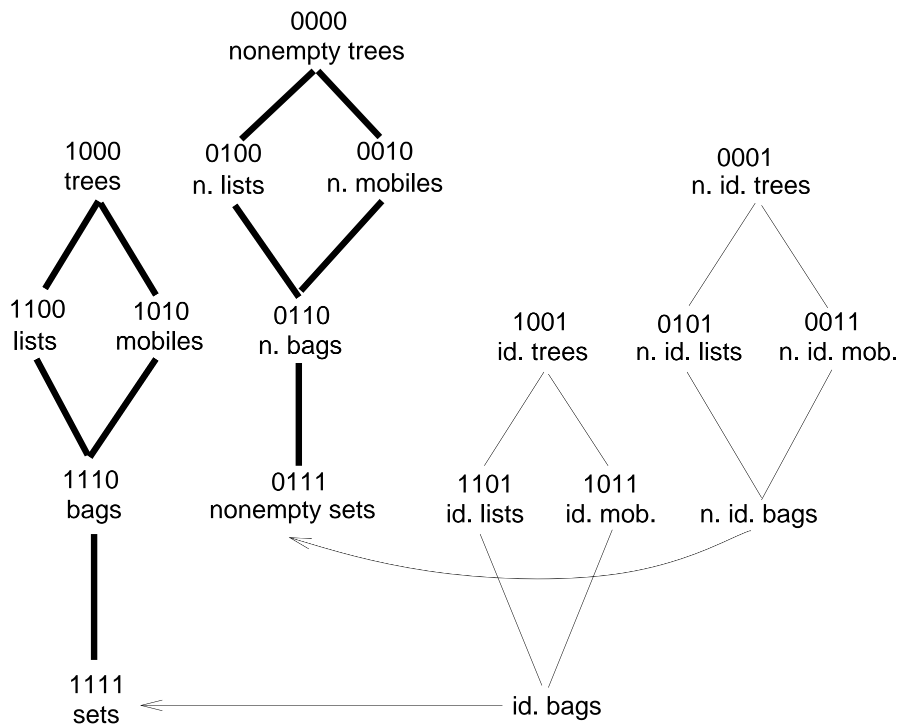

Figure 2: The Boom hierarchy

3 Reduce, map, filter

There are a couple of useful functions that can be applied to the Boom hierarchy data structures. The functions are *reduce* $/$, *map* $*$, and *filter* $\triangleleft$. Reduce is also called "fold". Filter removes elements from a structure, map applies a function to all elements of a structure, and reduce combines the elements in a structure.

The definition of reduce is:

$$\begin{aligned} \oplus/[\,] &= 1_\oplus \\ \oplus/[a] &= a \\ \oplus/(x +\!\!\!+\, y) &= \oplus/x \;\oplus\; \oplus/y. \end{aligned}$$

The unit of operation $\otimes$ is denoted $1_\otimes$. The function $\oplus/$ is a homomorphism from $+\!\!\!+$ to $\oplus$ if $\oplus$ satisfies the laws of $+\!\!\!+$ and not well-defined otherwise. That is,

- if $+\!\!\!+$ has a unit, then $\oplus$ must have a unit,
- if $+\!\!\!+$ is associative, then $\oplus$ must be associative,
- if $+\!\!\!+$ is commutative, then $\oplus$ must be commutative,
- if $+\!\!\!+$ is idempotent, then $\oplus$ must be idempotent.

Let's abbreviate "homomorphism from $+\!\!\!+$ to $\oplus$" by "$+\!\!\!+ \to \oplus$ homomorphism". The definition of map is:

$$\begin{aligned} f^*[\,] &= [\,] \\ f^*[a] &= [f.a] \\ f^*(x +\!\!\!+\, y) &= f^*x \;+\!\!\!+\; f^*y. \end{aligned}$$

The function f^* is a $+\!\!\!+ \to +\!\!\!+$ homomorphism. Map and reduce are crucial in defining homomorphisms from data structures. Indeed any homomorphism from a free algebra (a data structure) can be defined as a composition of a map and a reduce. This is called the "homomorphism lemma" [Bir86], or the "Universal property of free algebras". A given $+\!\!\!+ \to \oplus$ homomorphism h satisfying:

$$\begin{aligned} h.[\,] &= 1_\oplus \\ h.[a] &= f.a \\ h.(x +\!\!\!+\, y) &= h.x \oplus h.y \end{aligned}$$

can be written as $h = \oplus/ \circ f^*$. Reduce captures the way that $[\,]$ and $+\!\!\!+$ are replaced by $1_\oplus$ and $\oplus$, and map captures how h acts on singletons.

Filter $\triangleleft$ is defined on any possibly-empty data structures. Its definition is:

$$\begin{aligned} p \triangleleft [\,] &= [\,] \\ p \triangleleft [a] &= [a] \text{ if } p.a \text{ else } [\,] \\ p \triangleleft (x +\!\!\!+\, y) &= p \triangleleft x \;+\!\!\!+\; p \triangleleft y. \end{aligned}$$

The function $p\triangleleft$ is an $+\!\!\!+ \to +\!\!\!+$ homomorphism. Filter is convenient, but can be eliminated using map (see "trading" later).

4 Some data structures

Many useful functions from data structures are defined as homomorphisms from the join function of the data structure to another binary function. Therefore in the following most data structures are presented together with binary functions that have the same properties as the join of the data structure, and homomorphisms to them. We'll look at the four usual ones $tree, list, bags, set$, and at a couple of stranger ones.

4.1 1000 Trees

A tree is either the empty tree $[\,]$ containing no elements, or a singleton tree $[a]$ containing one element, or a join of two trees $r +\!\!+\, l$. Tree-join has unit $[\,]$, but it is not associative, not commutative, and not idempotent. These trees are binary trees with elements in their leaves.

4.2 1100 Lists

The organisational information that is lost in going from trees to lists is the shape of the tree. The law that is added to the algebra is associativity of $+\!\!+$.

Another operation with properties 1100 is square matrix multiplication of $n \times n$-matrices over complex numbers, where n is fixed. There is a square unit matrix I_n, and matrix multiplication is associative, but not commutative or idempotent. So Π defined over lists of matrices by:

$$\begin{aligned} \Pi.[\,] &= I_n \\ \Pi.[A] &= A \\ \Pi.(l +\!\!+\, r) &= \Pi.l \; mult \; \Pi.r \end{aligned}$$

is a homomorphism from lists to matrix multiplication, but not from any lower data structure. Using map and reduce we can write $\Pi = mult/ \circ \mathsf{id}^{\,*}$, which is just $mult/$. Composition of n-space transformations like translation, scaling, and rotation has properties 1100 too (and not surprisingly can be modelled by square matrix multiplication).

For lists of characters there is a more convenient notation: “ ” is the empty list, and “abc” is the list [‘a’,‘b’,‘c’].

4.3 1110 Bags

Bags are like lists that have lost their order, and like mobiles that have lost their shape. Bag join is commutative, but not idempotent. A bag is not ordered, but can contain an element once, twice, or any natural number of times. The number of times an element is contained in a bag is called the element's *frequency* in the bag.

The classic function with properties 1110 is addition. It is used to define three important homomorphisms from bags, namely $size, \Sigma$, and $freq.x$. They return the number of elements in a bag, their sum, and the frequency of a given x in a bag. Their definitions are:

$$size = +/ \circ (1 \ll)^*$$

$$\begin{aligned}\Sigma &= +/\\ freq.x &= +/\circ f^*,\\ &\quad \text{where } f.a = 1 \text{ if } a = x \text{ else } 0\end{aligned}$$

Obviously Σ only applies to bags of numbers. The function $\ll$ is called "first" and defined by $a \ll b = a$, and $(1 \ll)$ above is the function $(\lambda x : 1 \ll x)$. The function "second" is written $\gg$.

4.4 1111 Sets

Sets are like bags, but they have lost the notion of containing an element a particular number of times (the element's frequency). The law that is added to the algebra in going from bags to sets is idempotency of join $+\!\!+$.

Binary operations with properties 1111 are logical disjunction and conjunction, and max, min (if they have units). Disjunction is used to define three homomorphisms from sets, namely existential quantification $\exists$, $nonEmpty$, and the element-function $x \in$ for a given x. Their definitions are:

$$\begin{aligned}\exists &= \vee/\\ nonEmpty &= \vee/\circ(\mathsf{true} \ll)^*\\ (x \in) &= \vee/\circ f^*,\\ &\quad \text{where } f.a = (x = a).\end{aligned}$$

Universal quantification is defined similarly as a $+\!\!+ \to \wedge$ homomorphism.

4.5 0010 Nonempty mobiles

Another possible data structure is this: join $+\!\!+$ has no unit $[\,]$, but it is not associative, and is not commutative or idempotent. One could think of these structures as mobiles[1]. They are like trees that can rotate. Any subtree can also rotate, independently of the rest of the tree. In going from trees to mobiles, the structures lose their sense of left and right.

This binary function of natural numbers has properties 0010:

$$n \oplus m = (n \,\mathsf{max}\, m) + 1.$$

It is not associative, but commutative. The function $depth$ is a homomorphism from nonempty mobiles to $\oplus$:

$$depth = \oplus/\circ(1 \ll)^*$$

The function $depth$ is also a homomorphism from 0000 nonempty *trees*, because they are higher than 0010 nonempty mobiles, but not from possibly-empty mobiles, because $\oplus$ has no unit. However, its definition can be adjusted to have unit 0:

$$n \oplus m = (n \,\mathsf{max}\, m) \text{ if } (n \,\mathsf{min}\, m = 0) \text{ else } (n \,\mathsf{max}\, m) + 1,$$

and then $depth = \oplus/\circ(1 \ll)^*$ is defined on possibly-empty mobiles too, with $depth.[\,] = 0$.

[1] Mobiles were invented by the Alexander Calder (1989 - 1976) [Lip76]. They consist of objects suspended by threads from wires. Marcel Duchamp named them "mobiles" because the objects move in the wind. Calder called his *fixed* sculptures "stabiles".

4.6 0101 Nonempty idempotent lists

An idempotent list is like a normal list, but disregards equal adjacent sublists. So for idempotent lists we have:

"banana" = "bana" = "bbbabanananana".

For lists $as \mathbin{+\!\!+} bs = cs$ has a unique solution for as, given bs and cs. That is not so for idempotent lists. The properties 0101 seem a strange combination, but there are functions that satisfy them: the pair projections $\ll$ and $\gg$. They have no (left *and* right) units, are associative, $a \ll (b \ll c) = (a \ll b) \ll c$, are not commutative, but they are idempotent, $a \ll a = a$. The well-known functions *head* and *last* are homomorphisms from nonempty idempotent lists to the pair-projections:

$$\begin{aligned} head &= \ll / \\ last &= \gg /. \end{aligned}$$

Nonempty idempotent lists can be thought of as nonempty sets with two member-selecting functions. Since nonempty lists are higher than idempotent nonempty lists, *head* can be applied to them too.

5 Conclusion and related work

The Boom hierarchy comprises more than four data structures. In a Boom data structure any of the four properties UNIT, ASSOC, COMM, and IDEM can be present or absent, therefore there are 16 data structures (of varying practicality). Any data structure can be translated to one with a superset of properties, because it then loses structural information, but it can not be translated to one with a subset of properties, because it then would gain structural information, and there is a choice about what information should be added.

The Boom hierarchy and Squiggol are first presented in [Mee86] and [Bir86], and from then in many publications of the "Dutch School", mostly based on category theory. In [Bac89] they are compared to the Eindhoven quantifier expression notation. [Hoo92] presents the Boom hierarchy for relations, and [Jeu92] in a category context.

References

[Bac89] Roland Backhouse. An exploration of the Bird-Meertens Formalism. *International Summerschool on Constructive Algorithmics, Ameland 1989*, September 1989.

[Bir86] Richard S. Bird. An introduction to the theory of lists. In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design*, volume F36 of *NATO ASI Series*. Springer Verlag, 1986.

[Hoo92] Paul F. Hoogendijk. *(Relational) Programming Laws in Boom Hierarchy of Types*, volume 669 of *Lecture Notes in Computing Science*. Springer, June/July 1992.

[Jeu92] Johan Jeuring. Theories for algorithm calculation. *Lecture Notes of the STOP Summerschool on Constructive Algorithmics*, 1, September 1992.

[Lip76] Jean Lipman. *Calder's Universe*. Harrison House, New York, 1976.

[Mee86] Lambert Meertens. Algorithmics - towards programming as a mathematical activity. *Mathematics and Computer Science*, 1, 1986. CWI Monographs (J. W. de Bakker, M. Hazewinkel, J. K. Lenstra, eds.) North Holland, Puhl. Co.