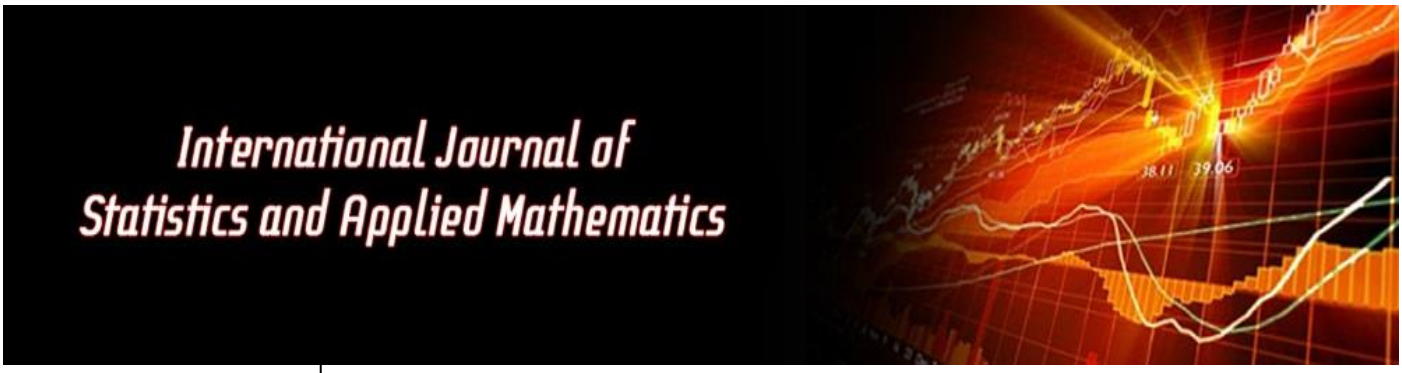
International Journal of Statistics and Applied Mathematics 2020; 5(5): 114-117



ISSN: 2456-1452
Maths 2020; 5(5): 114-117

www.mathsjournal.com
Received: 20-08-2020
Accepted: 23-10-2020

Dr. Daya Shankar Pratap
Research Scholar, Department of Mathematics, JP University, Chapra, Bihar, India

Corresponding Author:
Dr. Daya Shankar Pratap
Research Scholar, Department of Mathematics, JP University, Chapra, Bihar, India

Analysis of data structures of unified algebra

Dr. Daya Shankar Pratap

Abstract
For contrast, a set represents a collection of objects in a package or container. The contents of a set is a bunch. These vague descriptions are made precise as follows. A data structure is a collection, or aggregate, of data. The kinds of structuring we consider are packaging and indexing. These two kinds of structure give us four data structures.

1. unpackaged, unindexed: bunch
2. packaged, unindexed: set
3. unpackaged, indexed: string
4. packaged, indexed: list

Any binary or number is an elementary bunch, or element.

Keywords: Data structures of unified algebra

Introduction

The number 2 is an elementary bunch, or synonymously, an element. Indeed, every expression is a bunch expression, though not all are elementary.
If A and B are bunches, then

A, B "A union B"
A‘ B "A intersection B"

are bunches,

¢A "size of A"

is a number, and

A: B "A is in B", "A is included in B"

is a binary expression.
The size of a bunch is the number of elements it includes. Elements are bunches of size 1.

¢2 = 1
¢(0, 2, 5, 9) = 4

Here are three quick examples of bunch inclusion.

2: 0, 2, 5, 9
2: 2
2, 9: 0, 2, 5, 9

The first says that 2 is in the bunch consisting of 0, 2, 5, 9. The second says that 2 is in the bunch consisting of only 2. Note that we do not say "a bunch contains its elements", but rather "a bunch consists of its elements". The third example says that both 2 and 9 are in 0, 2, 5, 9, or in other words, the bunch 2, 9 is included in the bunch 0, 2, 5, 9.
I earlier made the statement "We must never use an expression to express more than one value; to do so would be a serious error called inconsistency." I now amend that statement to say "We must never use an elementary expression to express more than one value." Bunch expressions can indeed represent more than one value; that is their purpose, and we do not call it "inconsistency".
Here are the bunch laws. In these laws, x and y are elements (elementary bunches), and A, B, and C are arbitrary bunches.

(x: y) = (x = y)	elementary law
(x: A, B) = (x: A) ∨ (x: B)	compound law
A, A = A	idempotence
A, B = B, A	symmetry
A, (B, C) = (A, B), C	associativity
A‘ A = A	idempotence
A‘ B = B‘ A	symmetry
A‘ (B‘ C) = (A‘ B)‘ C	associativity
(A, B: C) = (A: C) ∧ (B: C)	
(A: B‘ C) = (A: B) ∧ (A: C)	
A: A, B	generalization
A‘ B: A	specialization
A: A	reflexivity
(A: B) ∧ (B: A) = (A = B)	antisymmetry
(A: B) ∧ (B: C) ≤ (A: C)	transitivity
¢x = 1	
∉(A, B) + ∉(A‘ B) = ¢A + ¢B	
– (x: A) ≤ (∉(A‘ x) = 0)	
(A: B) ≤ (∉A ≤ ∉B)	

For other laws.

Here are several bunches that are useful enough to be named:

null	empty	
bin	the binaries includes	a, ⊥
nat	the naturals	includes 0, 1, 2 and others
int	the integers	includes –2, –1, 0, 1, 2 and others
rat	the rationals	includes –1, 0, 2/3 and others
real	the reals	

We define them formally in a moment.

The operators, ‘ ∉: ≠ < > apply to bunch operands according to the axioms already presented. Other operators can be applied to bunches by applying them to the elements of the bunch. For example,

–null = null	base
–(A, B) = –A, –B	distribution or factoring
A + null = null	base
A + (B, C) = A + B, A + C	distribution or factoring

This makes it easy to express the positive naturals (nat + 1), the even naturals (nat × 2), the squares (nat2), the powers of two (2nat), and many other things.

We define the empty bunch, null, by the laws

null: A
(¢A = 0) = (A = null)

The bunch bin is defined by the law bin = a, ⊥.

The bunch nat is defined by two laws.

0, nat + 1: nat	construction
(0, B + 1: B) ≤ (nat: B)	induction

The first, construction, says that 0, 1, 2, and so on, are in nat. The second, induction, says that nothing else is in nat by saying that of all the bunches satisfying the construction law, nat is the smallest. Now that we have nat, we can define int and rat as follows:

int = nat, –nat
rat = int/(nat + 1)

The law defining real will be given later in the section titled “Limits”.

We also use the notation

m...n “m to n”

where m is integer, and n is integer or binary, and m ≤ n. This notation means the bunch m, m + 1, m + 2, up to but not including n. The asymmetric notation is a reminder that the left end is included but the right end is excluded. Here are its laws:

(x: m...n) = (x: int) ∧ (m ≤ x < n)
¢(m...n) = n – m

And here are three examples:

0...3 = 0, 1, 2
0...0 = null
0...a = nat

Sets

Let A be any bunch (anything). Then

{A} “set containing A”

is a set. Thus {null} is the empty set, and the set containing the first three natural numbers is expressed as {0, 1, 2} or as {0...3}, and {nat} is the set of natural numbers. All sets are elements; not all bunches are elements; that is the difference between sets and bunches. We can form the bunch 1, {3, 7} consisting of two elements, and from it the set {1, {3, 7}} containing two elements, and in that way we build a structure of nested sets. Set formation has an inverse. If S is any set, then

~S “contents of S”

is its contents. For example,

~{0, 1} = 0, 1

Now that we have bunches, the laws of sets are very easily stated.

{~S} = S	set formation
~{A} = A	contents
{A} + A	structure
#{A} = ¢A	size
(A ∈ {B}) = (A: B)	element
({A} ≤ {B}) = (A: B)	subset
({A} ∈ B) = (A: B)	power
{A} ∨ {B} = {A, B}	union
{A} ∧ {B} = {A ‘ B}	intersection
({A} = {B}) = (A = B)	equation

Note that the element, subset, and power laws are all just bunch inclusion.

Strings

Just as bunches and sets are, respectively, unpackaged and packaged collections, so strings and lists are, respectively, unpackaged and packaged sequences. There are sets of sets, and lists of lists, but there are neither bunches of bunches nor strings of strings.

The simplest string is

nil the empty string

Any binary, number, set (and later also list and function) is a one-item string, or item. For example, the number 2 is a one-item string, or item. A bunch of items is also an item. Strings are catenated (joined) together by semicolons to make longer strings. For example, 4; 2; 4; 6 is a four-item string. The length of a string is the number of items, and is obtained by the $ operator.

$(4; 2; 4; 6) = 4

The index of an item is the number of items that precede it. In other words, indexing is from 0. An index is not an arbitrary label, but a measure of how much has gone before. Your life begins at year 0, a highway begins at mile 0, and so on. We refer to the items in a string as “item 0”, “item 1”, “item 2”, and so on; we never say “the third item” due to the possible

confusion between item 2 and item 3. We obtain an item of a string by subscripting. For example,

(3; 5; 7; 9)2 = 7

In general, S_n is item n of string S. We can even pick out a whole string of items, as in the following example.

$(3; 5; 7; 9)_{2; 1; 2} = 7; 5; 7$

Strings can be compared for equality and order. To be equal, strings must be of equal length, and have equal items at each index. The order of two strings is determined by the items at the first index where they differ. For example,

3; 6; 4; 7 < 3; 7; 2

If there is no index where they differ, the shorter string comes before the longer one.

3; 6; 4 < 3; 6; 4; 7

This ordering is known as lexicographic order; it is the ordering used in dictionaries.

If i is an item, and S and T are strings, then

nil	the empty string
i	an item
S; T	"S catenate T"
ST	"S sub T"
S ∧ T	"S min T"
S ∨ T	"S max T"

are strings, and

$S	"length of S"

is a natural number or a, and

S = T	"S equals T"
S + T	"S differs from T"
S < T	"S is less than T"
S > T	"S is greater than T"
S ≤ T	"S is at most T"
S ≥ T	"S is at least T"

are binary.

Here are the laws of string algebra. In these laws S, T, and U are strings, and i and j are items.

nil; S = S; nil = S	S_{null} = null
S; (T; U) = (S; T); U	$S_{T, U} = S_T, S_U$
$nil = 0	$S_{\{T\}} = \{ST\}$
$i = 1	S_{nil} = nil
$(S; T) = $S + $T	$S_{T; U} = S_T; S_U$
($S < a) ≤ ((S; i; T)$S = i)	$S_{(TU)} = (S_T)_U$

($S < a) ≤ ((I < j) = (S; i; T < S; j; U)) ($S < a) ≤ (nil ≤ S < S; i; T)

($S < a) ≤ ((I = j) = (S; i; T = S; j; T))

We also use the notation x;..y "x to y" (same pronunciation as x,..y) where x is an integer, and y is an integer or binary, and x ≤ y. As in the similar bunch notation, x is included and y excluded, so that

$(x;..y) = y – x

Here are the laws.

x;..x = nil
x;..x + 1 = x
(x;..y) ; (y;..z) = x;..z

String catenation distributes over bunch union:

A; null; B = null base
(A, B); (C, D) = (A; C), (A; D), (B; C), (B; D) distribution or factoring

So a string of bunches is equal to a bunch of strings. Thus, for example,

0; 1; 2: nat; 1; (0,..10)

because 0: nat and 1: 1 and 2: 0...10.

Our main purpose in presenting string algebra is as a stepping stone to the presentation of list algebra.

Lists

A list is a packaged string. For example,

[0; 1; 2]

is a list of three items. List brackets [ ] distribute over bunch union.

[null] = null	base
[A, B] = [A], [B]	distribution or factoring

Because of the distribution we can say

[0; 1; 2]: [nat; 1; (0...10)]

On the left of the colon we have a list of integers; on the right we have a list of bunches, or equivalently, a bunch of lists.

Let S be a string, L and M be lists, n be a natural number, and i be an item. Then

[S]	"list containing S"
L M	"L M" or "L composed with M"
L + M	"L catenate M"
n →i | L	"n maps to i otherwise L"
L ∧ M	"L min M"
L ∨ M	"L max M"

are lists,

L n	"L n" or "L index n"

is an item,

~L	"contents of L"

is a string,

#L	"length of L"

is a natural number or binary, and

L = M	"L equals M"
L ≠ M	"L differs from M"
L < M	"L is less than M"
L > M	"L is greater than M"
L ≤ M	"L is at most M"
L ≥ M	"L is at least M"

are binary.

Parentheses may be used around any expression, so we may write L(n). If the index is not simple, we will have to enclose it in parentheses. When there is no danger of confusion, we may write Ln without a space between, but when we use multicharacter names, we must put a space between.

The contents of a list is the string of items it contains.

~[3; 5; 7; 4] = 3; 5; 7; 4

The length of a list is the number of items it contains.

#[3; 5; 7; 4] = 4

List indexes, like string indexes, start at 0. An item can be selected from a list by juxtaposing (placing next to each other) a list and an index.

[3; 5; 7; 4] 2 = 7

A list of indexes gives a list of selected items. For example,

[3; 5; 7; 4] [2; 1; 2] = [7; 5; 7]

This is called "list composition". List catenation is written with a small raised plus sign+.

[3; 5; 7; 4] + [2; 1; 2] = [3; 5; 7; 4; 2; 1; 2]

The notation n→i | L gives us a list just like L except that item n is i.

2→22 | [10;..15] = [10; 11; 22; 13; 14]
2→22 | 3→33 | [10;..15] = [10; 11; 22; 33; 14]

Let L = [10;..15]. Then

2→L3 | 3→L2 | L = [10; 11; 13; 12; 14]

The order operators $< \leq > \geq$ apply to lists; the order is lexicographic, just like string order.

Here are the laws. Let S and T be strings, and let i and j be items.

[S] + S = [S] = [S]	structure, contents, and formation
#[S] = $S	length
[S] + [T] = [S; T]	catenation
[S] T = S_T	indexing
$[S_T] = S_{[T]}$	
[S] [T] = $[S_T]$	composition
($S < a) ≤ (($S) → i | [S; j; T] = [S; i; T])	modification
([S] = [T]) = (S = T)	equation
([S] < [T]) = (S < T)	order

Let L, M, and N be lists, and n be natural. Then

(L M) n = L (M n)	
(L M) N = L (M N)	associativity
L (M + N) = L M + L N	distributivity or factoring

When a list is indexed by a structure, the result will have the same structure. Here is a fancy example. Let L = [10; 11; 12]. Then

L [0, {1, [2; 1]; 0}] = [L 0, {L 1, [L 2; L 1]; L 0}] = [10, {11, [12; 11]; 10}]

Lists can be items in a list. For example, let

A = [[6; 3; 7; 0] ;
[4; 9; 2; 5] ;
[1; 5; 8; 3]]

Then A is a 2-dimensional array, or more particularly, a 3 × 4 array. Indexing A with one index gives a list

A 1 = [4; 9; 2; 5]

which can then be indexed again to give a number.

Conclusion

When we apply a formalism to describe and reason about some phenomena, we may find that it works quite well for a certain range of observations, but less well outside that range. In this formalism, when D represents a finite class of objects and Bx is a binary expression, we find that ∨⟨x: D → Bx⟩ is quite useful for saying "There exists an object, let's call it x, in the bunch of objects represented by D, such that B is true of x." But when D represents an infinite bunch of objects, ∨⟨x: D → Bx⟩ differs slightly from the traditional mathematical idea of existence. One way to resolve the discrepancy is to redesign the algebra, sacrificing simplicity and elegance, to attempt to fit the traditional mathematical idea of existence more closely. Another way to resolve the discrepancy is to part from the traditional mathematical idea of existence in order to fit the algebra, or even abolish the idea of mathematical existence altogether. I prefer abolishment.

References

1. Grundy J. Transformational Hierarchical Reasoning. The Computer Journal 1996;39(4):291-302.
2. Hehner ECR. From Boolean Algebra to Unified Algebra. The Mathematical Intelligencer, Springer 2004;26(2):3-19. www.cs.toronto.edu/~ hehner/BAUA.pdf
3. Hehner ECR. A Practical Theory of Programming. Springer 1993. www.cs.toronto.edu/~hehner/aPToP
4. Hoare CAR. A Couple of Novelties in the Propositional Calculus. Zeitschrift für Mathematische Logik und Grundlagen der Mathematik 1985;31(2):173-8.