Pervasive Simplicity:
Investigations of the 'Primary Algebra' of *Laws of Form*

George Burnett-Stuart

To the memory of my sister Joanna (1960-2020)

§1 Introduction

This paper takes as its starting-point a talk given at the LoF50 Conference in Liverpool, in August 2019, 'Lessons from the Markable Mark'; then it runs off after an algorithm for simplifying algebraic expressions. This new project was sparked off by some remarks made by William Bricken during his talk at the conference [1]. It is probable that the algorithm arrived at in this paper bears a strong resemblance to algorithms already developed by Bricken but not yet published. I conclude by describing some applications of the algorithm to the solution of recreational logic problems.

'The Markable Mark' is my *Laws of Form*-based website, to be found at `www.markability.net`. The website consists essentially of three main parts:

- An introduction to the Primary Arithmetic by way of a fable or fairy tale about a woodcutter, who each day has to cut down the trees that have been marked in the night by his line manager, the mysterious Forester. The woodcutter has to figure out, on pain of something horrible happening to him, or his family, or his animals, which trees are marked for destruction (or not, as the case may be). If a tree has a circle painted on it it has to be chopped down. A second circle, beside the first one, seems to make no difference – the tree is still marked; but a second circle *inside* the first one turns out to *cancel* the mark: a tree with two concentric circles painted on it is to be spared. The mark is an empty circle: a circle, that is, whose interior is unmarked. And so it goes on...
- A presentation of the Primary Algebra, including an interactive Flash program (which readers may like to try, if they haven't already) in which one can practise the legal moves in the game of Primary Algebra
- A section on what I call the Calculus of Circles, Letters and Subscripts, which is a way of complicating the Primary Algebra (or 'Calculus of Circles and Letters') in such a way that it can do the work of the 'first order' logic of predicates and quantifiers (things like 'for all x there exists a y such that such-and-such a relation holds between x and y', and that sort of thing).

In this paper we're not going to be concerned with the subscripts, or the woodcutter, just with the Primary Algebra. Now in *Laws of Form* [2] the Primary Algebra is introduced as 'a calculus taken out of the calculus', that is, out of the Primary Arithmetic, essentially by letting letters stand for 'unknown values', so each letter is a sort of place-holder for a value (mark or for no-mark) – with the proviso that if the same letter crops up more than once, in an expression, it stands for the same value everywhere, as in the ordinary numerical algebra that we learn at school. But here I want to emphasize that this game of algebra, even if it 'arises' in the way just described, can stand on its own feet, existing independently on its own terms.

To bring out this point I will tend to refer to it as the 'Calculus of Circles and Letters', or CL-calculus, for short. The essential thing is that the letters are just themselves, not 'standing for' something else. They don't actually *have to* be letters – we'll see what their essential properties have to be, as we go along. The original idea of my talk was to be a sort of experiment: if we deny ourselves the obvious, ready-to-hand interpretation of what this CL-calculus is all about, and try just to look at it 'without prejudice', perhaps other meanings will emerge. (The fact that the Primary Algebra can 'stand on its own feet' is of course part of the argument in *Laws of Form*, particularly Chapters 9 and 10, on Completeness and Independence. I'm just emphasizing it.)

How then does the 'game' of CL-calculus work? We work with 'CL-expressions', defined as follows: a) any expression has a 'skeleton', consisting of a set of non-intersecting, perfectly continent circles, some of which may be contained in others; and b) the skeleton may then be sprinkled with letters, with each letter falling in one of the spaces between the circles; repeats of the same letter are allowed. Note that there may be no letters, no circles, or indeed nothing at all ('the void expression').

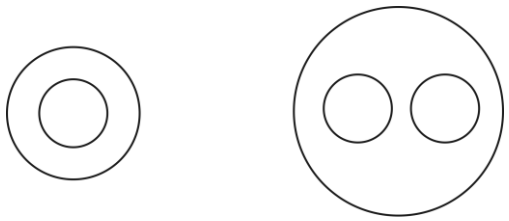
Next, we add in the Elementary Moves, from which more complicated moves can be built up. In order to explain the moves, it is convenient to adopt the convention that an upper-case letter, such as A, or E, stands for 'any CL-expression'. Also, the permitted transformations can take place anywhere *inside* a larger expression: we decree that, in such a case, the resulting transformation of the larger expression is also a permitted one. Finally, an equals sign is taken to mean that the form on the left of the equation can be changed into the form on the right, or *vice versa*. Both directions are permitted.

In *Laws of Form*, the Elementary Moves (also called 'initial equations') are given as follows:

J1 ('Position'): ((P) P) =

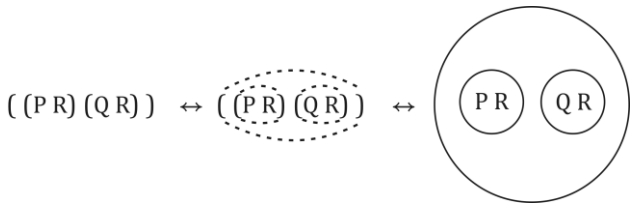
J2 ('Transposition'): ((PR) (QR)) = ((P) (Q)) R

We note that the skeletons of the various expressions in **J1** and **J2**, respectively, which should really be patterns of circles, namely



(J1) (J2)

are represented in our equations by means of brackets, with each matching pair of brackets representing a circle – as if the two brackets in each pair were connected up by imaginary arcs above and below the line of text:



The development of the algebra, out of the initials **J1** and **J2**, is covered in *Laws of Form*. Alternatively, the whole thing can be based on a single Elementary Move, which we will denote **H** (short for Huntington):

H: A = ((A) B) ((A) (B))

or on the single equation **R** (for Robbins):

R: A = ((A B) (A (B)))

The developments out of **H** (or out of **R**) are given by Kauffman[3]. The equivalence of the various systems is proved by the fact that **H** can be shown to be a consequence of **J1** and **J2**; and conversely, both **J1** and **J2** can be shown to follow from **H** (with similar statements applying to **R**).

But our concern will be with a third system, due to William Bricken[4]. The 'Bricken initials' are the ones I like. There are more of them – three, rather than two, or one – but they have the virtue of simplicity. I first took up with them because their simplicity made them more suitable than the alternatives, to build into the Flash program on my website (the so-called 'Algebraic Play Area'). But their appeal goes beyond mere convenience, as Bricken himself has always insisted. Hopefully, what follows will bear this out.

So here are Bricken's three Initial Equations; or six Elementary Moves – as the moves can go in either direction. (I have added my own personal names for the moves.)

B1 A = ((A)) (→ wrap ← unwrap)

B2 A ( ) = ( ) (→ delete ← undelete)

B3 A (B) = A (A B) (→ copy ← uncopy).

By way of commentary:

- Wrap: a double enclosure (with nothing in the annular gap between the two enclosing circles) is the same as no enclosure at all
- Delete: when a space is marked by an empty circle, anything else can be inserted into, or removed from the same space – as if the mark shines so brightly, that anything else going on in the same space, any coming or going, is invisible, makes no difference. Any other sub-expression in the same space is effectively void-equivalent
- Copy: a sub-expression or 'form' A can be copied at will to the interior of any circle occupying the same space as A – no matter what else is already in that interior space. Conversely, if a form in some space is duplicated in the next space out, then the copy in the inner space can be deleted

The 'copy' move is susceptible of immediate generalization. Suppose we are given an expression of the form

A (B (C))) .

By successive application of copy and uncopy we can build up the following transformation:

A (B (C))) = A (A B (C))) [by copy, level 0 to level -1]

= A (A B (A C))) [by copy, level-1 to level -2]

= A (B (A C))) [by uncopy, from level -1, with reference to level 0].

Thus A has been copied, not to the next space down, but to a space two levels down (or further 'in'). By reversing the steps, a form in a deeper space, if duplicated two levels up, can be deleted or 'uncopied' from the deeper space.

Clearly, there is nothing special here about the number two: essentially the same argument would apply to copying a form into *any* space interior to a given space, at any depth; and to uncopying a form if duplicated in any higher space.

We may say (following Bricken) that the form in the outer space PERVADES all the spaces contained in that outer space, and has the seemingly supernatural ability to appear out of thin air in any of those spaces, or else to disappear out of them. It is as if it is, or is not, everywhere. As you get used to the idea of pervasion, you may get into the way of thinking that the outer form should be 'understood' to be present in all the inner spaces, there is just no need to write it in explicitly.

One last point about the generality of 'copy': a form can be copied into its own space. We see this as follows:

A = A (( )) [a bit of void has been 'wrapped']

= A ((A)) [A copied to the second level down]

= A A [the copy of A unwrapped].

But there is no way A can be copied to any location that is *exterior* to the space in which A lies.

One nice thing about the Bricken axioms is the way that the Copy axiom encapsulates the essential purpose or characteristic of the *letters* in this Calculus of Circles and Letters. (We could say that Wrap and Delete encapsulate the essential characteristics of the circles.) Letters are the kind of thing of which we may say 'Ah yes, this one is *the same* as that', and be quite clear about what we mean. This is the meaning of Charles Peirce's famous 'type/token distinction'. There is a type called 'the letter x', and all occurrences of that letter, in printed books or magazines, say, or in CL-expressions, are tokens or instances of that type. Two tokens are 'the same' if they instantiate the same type. Indeed, in some applications of the CL-calculus, as we shall see, there aren't enough letters to stand for all the tokens that are needed. At this point it becomes apparent that the CL-calculus is really a calculus of circles and tokens. The tokens can be strings of letters (or words) or mixtures of letters and numbers – or they could be many other things. All that is needed is that every token is an instance of a definite type, in such a way that it is always decidable whether or not two tokens are 'the same'. Rather than a calculus of Circles and Letters, one could talk of a calculus of Containment and Sameness.

In the next sections we shall see how our three Elementary Moves can be combined and compounded to effect more elaborate transformations.

§2 Some Five-Finger exercises in the primary algebra

To start with, let's set ourselves the task of proving that the *Laws of Form* initials, **J1** and **J2**, and the Huntington initial, **H**, follow from the Bricken initials. For **J1**, we start with the form ((P) P) , and apply uncopy, delete, and unwrap in turn:

((P) P) = (( ) P) [uncopy from level -2 to level -1]

= (( )) [delete]

= [unwrap].

In reverse, a bit of void is wrapped up; in level -1 the glare of the empty circle allows the expression P to come into being next to it; then P is then copied into level -2.

That wasn't too bad! **J2** is a somewhat harder nut to crack. It is one of those cases where we get on better if, rather than plunging straight in, we first pause to prove a 'lemma' (as these bits of scaffolding, or 'ancillary results', are called by mathematicians). My private name for this minor result is Black Dog (I give the reason in a footnote[1], but you should pay no attention to this, as logic is not our subject); Bricken calls it 'subsumption'.

Lemma (Black Dog): For all expressions A and B,

(A) = (A) (AB) .

Proof: We start with (A), then we wrap up a bit of the void in the space outside (A). Then, in the glare of the empty circle at level -1, we allow the expression B to come into being:

(A) = (A) (( )) = (A) (( ) B).

Now, the empty circle having done its bit, we want to get rid of it, or, better, replace it by A. How might we do that? It turns out to be surprisingly easy: we just have to copy (A) into the empty circle:

(A) (( ) B) = (A) (((A)) B) = (A) (AB),

where the last step is just an unwrap. ***Q.E.D.***

It goes without saying that we could do the transformation in reverse, by reversing each step; but there is a more direct way of doing it that is not without interest. Given (A) (AB), we start by copying (A) into the interior of (AB):

(A) (AB) = (A) ((A) AB).

Then we uncopy A from level -2 to level -1, giving an empty circle, which then annihilates AB:

(A) ((A) AB) = (A) (( ) AB) = (A) (( )) = (A).

How could we describe what has happened? Looking at it in a biological way, we could say that the form (A) has *killed* the form (AB). Its first move was to send a copy of itself through the outer

[1] In logic, to say 'it's not a dog, and it's not a black dog' is to say no more than 'it's not a dog'. If we let d stand for 'it's a dog', and b stand for 'it's black', and if we let bracketing stand for negation, we can then say that (d) (bd) says no more than (d). It says no less, either, so, in sum, (d) = (d)(bd).

membrane of (AB); from then on a relentless, rather sinister mechanism led to the annihilation of the victim.

We're now ready to go onto **J2**. We start with the form ((PR) (QR)). How are we going to get that R out, from level -2, into the outer space, level 0? We do it by a sort of conjuring trick. Watch carefully! It is as if we 'imagine' that R has got out to where we want it to be. We start by wrapping a bit of the void outside the original expression; then, as we've now done a couple of times, we use the glare of the empty circle in level -1. But this time, what we bring into being is not plain R, but R in a circle. So:

((PR) (QR)) = ((PR) (QR)) (( )) = ((PR) (QR)) (( ) (R)).

Now, if we could only get rid of the empty circle from the right-hand factor, that factor would turn into R, doubly wrapped, equivalent to R. So, as in the proof of Black Dog, we do what we can, which is to copy the left hand factor into that empty circle:

((PR) (QR)) (( ) (R))= ((PR) (QR)) ((((PR) (QR))) (R)) [copy]

= ((PR) (QR)) ((PR) (QR) (R)) [unwrap]

Now, by Black Dog, (R) can set about killing, first (PR), then (QR) (all going on inside the second factor), giving us

((PR) (QR)) ((R)) = ((PR) (QR)) R [unwrap]

= ((P) (Q)) R [uncopy, twice]

We're done: **J2** is proved. As for **H**, we get at this via the following

Splitting Lemma: For any expressions E and B,

(E) = (EB) (E (B)).

Proof: We begin with a double application of Black Dog:

(E) = (E) (E (B)) [Black Dog]

= (E) (EB) (E (B)) [Black Dog again].

We now let the two extruded expressions (the black dog and the not-black dog, as it were) turn back on their parent (E) and attack it, in concert:

(E) (EB) (E (B)) = (E (EB) (E (B))) (EB) (E (B)) [copy, twice]

= (E (B) ((B))) (EB) (E (B)) [uncopy, twice]

= (E (B) ( )) (EB) (E (B)) [uncopy]

= (EB) (E (B)) [delete, unwrap] ***Q.E.D.***

Between the two of them[2], (EB) and (E (B)) were able to kill off (E). If we now substitute (A) for E, in the splitting lemma, and unwrap, we obtain

A = ((A)) = ((A)B) ((A) (B)),

which is **H**. On the other hand, by simply circling each side of the splitting lemma equation, and changing E to A, we get

A = ((A)) = ((AB) (A (B))),

which is the 'Robbins initial', **R**.

§3 The Power of Life and Death

We return to the matter of 'killing'. We have said that (A) can 'kill' any expression of the form (AB). But there is another side to it: (A) can *engender* (AB) as well as destroy it; in other words (A) has power of life *and* death over all expressions of the form (AB) (including the case when B is the void expression). More generally, if E and F are expressions such that E can kill F (implying that E can also engender F) I am going to say that 'E has power of life and death over F'. For short, I will express this state of affairs (or relation between E and F) by the statement

$$E \geq F ;$$

and when I've had enough of 'power of life and death', I'll just say that E *dominates* F.

By what has been said,

$$E \geq F \text{ if and only if } E = E\,F.$$

There is more to this relation than meets the eye, as we see in the following composite theorem:

Theorem (Dominance). Let A, B, C, E, F, G, X be CL-expressions, then:

i. (Transposition) If $E \geq F$ then $(F) \geq (E)$
ii. If $E \geq F$ and $F \geq E$, then $E = F$
iii. For all E, $(\ ) \geq E$
iv. For all E, $E \geq\ $.
v. For all A and B, $AB \geq A$
vi. In particular, $A \geq A$
vii. If $E \geq F$, then, for all X, $EX \geq FX$
viii. (Transitivity) If $E \geq F$ and $F \geq G$, then $E \geq G$
ix. (Criterion for dominance) $E \geq F$ if and only if (E (F)) = .
x. (Peircean see-saw) Consider the two expressions A(B(C .. (LX) ..)) and A(B(C .. (L) ..)), where the dots indicate that there may be more nested circles and intervening expressions between C and L. Then in case X lies at even depth, the first expression (with X in place) has power over the second (with X removed); and *vice versa* if X lies at odd depth.

[2] In logic the interpretation would be: if it's not a black dog, and not a not-black dog, then it's not a dog at all.

Proof: The proof of the first clause is perhaps the most interesting. This says, if E kills F, then circled F kills circled E. So we see what happens when we send (F) in to attack (E):

(F) (E) = (F) ((F) E) [copy]

= (F) ((FE) E) [copy]

= (F) ((E) E) [E kills F]

= (F) (( ) E) [uncopy]

= (F) [delete, unwrap].

There is something deeply satisfying here, I feel, in the way the tables are turned, so the very fact that E can kill F means that (F) has the power to kill (E). There is something else to note. In order to as it were unleash the power of E to kill F, we had to copy E from level -1 into level -2, and then uncopy it out again, after the deed was done. But the from the 'pervasion' point of view, it is unnecessary to explicitly copy E into the inner space, we already know that it has the ability to pop up in any inner space whenever it wants to. But this means that its power of life and death over another expression (e.g. F) also extends to any inner space. In other words, *the power of life and death is also pervasive*. In this particular case, E can kill F from outside the space in which F lies. So a shortened proof of i) would read:

(F) (E) = (F) ((F) E) [copy]

= (F) (( ) E) [E kills F]

= (F) [delete, unwrap].

ii) has a very nice proof, perhaps the archetypal 'one line proof':

E = EF = F.

Here the first step uses E ≥ F (so E engenders F), the second, F ≥ E (so F kills E).

iii) is a restatement of the axiomatic **B2**. The empty circle is all-powerful.

iv) No expression is altered by the juxtaposition of more void. The void is the doormat of the universe, universally dominated; the receptive, the ever available, the Yin.

v) This follows from Black Dog, by transposition (that is, by (i) above). Or, more directly:

AB = AB A [A copied into its own space]

vi) Just put B = in (v).

vii) E ≥ F means E = EF, so EX = EF X = EF XX = EX FX, in other words, EX ≥ FX.

viii) We have E = EF and F = FG, therefore E = EF = EFG = EG, as required.

ix) First, suppose E ≥ F, so E = EF. Then (E (F)) = (EF (F)) = (EF ( )) = . Conversely, if (E (F)) is void-equivalent, then

E = E (E (F)) [premise]

= E ((F)) [uncopy]

= EF [unwrap]

x) We use v) and vii) together with transposition (i). For any L and X, LX ≥ L. Therefore, by transposition, (L) ≥ (LX), and by vii) K (L) ≥ K (LX). By transposition, (K (LX)) ≥ (K (L)), and by vii) J (K (LX)) ≥ J (K (L)). Continue the steps as often as is required. ***Q.E.D.***

Coda

In the notes to *Laws of Form* ([4], notes to Chapter Six), Spencer Brown makes reference to a work *Elements of Theology* by the 4th century Neo-Platonist Proclus, in particular to the terms *proódos* (procession) and *epistrophe* (reversion), which play a fundamental role in the work. Spencer Brown likens the two directions of the Copy equation – that is, Copy and Uncopy – to procession and reversion, respectively. I propose that the analogy be extended to the powers of life and death. If an expression A dominates an expression B, then A can produce B out of itself, either in the same space as A, or in any space pervaded by A, at will. This seems to me to correspond rather nicely to Proclus' notion of procession. And this production is never irrevocable, it can always be as it were 'undone' – this is reversion. Proclus interprets cause and effect in these terms: the effect proceeds from the cause, and can always return into it. This is surely a good way of thinking about cause and effect – one that enlarges our habitual, overly mechanical, 21st century notion. The One, from which everything proceeds, corresponds, of course, to the empty circle O ; and the state of total privation, from which nothing can proceed, to .

§4 The normal form of an Expression

Given an expression E, the Huntington equation, **H**, allows us to express it as

E = ((E) B) ((E) (B)),

where B is any expression. Now suppose that the letters which appear in E are x, y, z... Then we can use the expression consisting of the single letter x, for the expression B in the equation, giving us

E = ((E) x) ((E) (x)).

Inside the first factor, ((E) x), the letter x pervades E to all depths, implying that all occurrences of x in E can be uncopied: in other words, E can be replaced by the expression E_x, where E_x is obtained from E by removing every occurrence of x – or, to put it another way, by substituting the void expression for every occurrence of x in E.

In the second factor, we can remove every occurrence of (x) from E. At first sight, this may not allow us to do very much, as x may not occur very often, in E, in the form (x). But actually, this is no obstacle, because we can replace every occurrence of x in E by the 'wrapped' form ((x)), and every such form certainly contains an occurrence of (x), which can be removed, leaving an empty circle ( ). This amounts to replacing every occurrence of x, in E, by an empty circle. Let's call the expression, so obtained, $E_{(x)}$.

We now have:

$$E = ((E_x\ )\ x)\ ((E_{(x)}\ )\ (x)).$$

This is an important result in its own right, leading to a useful theorem, which we will call the Two-Case Theorem:

Theorem (Two-Case): Let E and F be two expressions, and let E_x, F_x be the expressions that result when x is everywhere replaced by the void, and $E_{(x)}$, $F_{(x)}$, those that result when each occurrence of x is replaced by the mark. Then E is equivalent to F if and only if equivalence holds both between E_x and F_x, and between $E_{(x)}$ and $F_{(x)}$.

Proof: The 'if' part is proved by the sequence

$$E = ((E_x\ )\ x)\ ((E_{(x)}\ )\ (x)) = ((F_x\ )\ x)\ ((F_{(x)}\ )\ (x)) = F.$$

For the 'only if' part, we note that the same steps which transform E into F can be repeated with the void everywhere, from beginning to end, substituted for x: starting with E_x, the steps will lead inevitably to the result F_x. Similarly, with the empty circle substituted for x: starting with $E_{(x)}$, the steps will lead to $F_{(x)}$. ***Q.E.D.***

Corollary: If both E_x and $E_{(x)}$ are void-equivalent, then so is E; and conversely.

Proof: this is a special case of the theorem, with F = .

Now E_x and $E_{(x)}$ no longer contain the letter x, but they may very well still contain the letters y, z, and so on. We now turn our attention to the next remaining letter, which we may assume to be y. Both of the factors on the right hand side of the equation just preceding the last theorem can be split using the splitting lemma of §2:

$$E = ((E_x\ )\ xy\ )\ \ ((E_x\ )\ x(y)\ )\ \ ((E_{(x)}\ )\ (x)y\ )\ ((E_{(x)}\ )\ (x)(y)\ );$$

then we can do pervasive uncopying of y and (y), respectively, to obtain

$$E = ((E_{xy}\ )\ xy\ )\ \ ((E_{x(y)}\ )\ x(y)\ )\ \ ((E_{(x)y}\ )\ (x)y\ )\ ((E_{(x)(y)}\ )\ (x)(y)\ ).$$

Here $E_{(x)y}$, for example, is obtained from E by substituting an empty circle for every occurrence of x, and a void for every occurrence of y. $E_{(x)y}$ (and the other similar expressions) have now been purged of x's and y's, but may well still contain other letters, such as z, and so on, to w, say. Then we can split again, with respect to z, and so on. Eventually E will be turned into a product of 2^n factors, where n is the number of letters. But if there are no letters left, each of the 'reduced' expressions, such as

$$E_{xyz..w},$$

(where w is supposed to be the last letter) must contain *no letters at all*: in other words, it will be an expression made up of circles alone, and therefore equivalent either to the void or to an empty circle. In the first case the *factor* gets scrubbed out (destroyed, as it were, by the empty circle inside it); in the second case the circled reduced expression is void-equivalent and can be scrubbed out, leaving a factor of the form (xyz..w), or ((x)yz.. w), and so forth, with some of the individual letters circled, others not.

In sum, by the end of the splitting operation E will have been transformed into a product of the form

$$(F_1)(F_2)(F_3)..\ (F_m),$$

where m is a number not greater than 2^n, and each F_i is a product of all the letters, but with some circled, others not. We call this the 'normal form' of E.

Suppose we call any product of all the letters, with some circled, others not, a 'valuation string'. The number of distinct possible valuation strings is 2^n, because each one contains n letters, each one of which may be circled or not. The set $\{F_1, F_2, F_3 .. F_m\}$ is a subset of the set of all possible valuation strings. Whether a particular valuation string $\mathcal{v}$ belongs to the set, or not, depends on whether the reduced expression $E_{\mathcal{v}}$ evaluates to mark or no-mark. For example,

$(\mathcal{v}_1)$ = (x y (z) a (b) c)

will be a factor in the fully split normal form of E, if and only if the reduced expression

$E_{xy(z)a(b)c}$

evaluates to the mark.

Recall the definition of this reduced expression: it is what you get if, starting with the original E, you replace every occurrence of x, or y, or a, or c by a void, and every occurrence of z or b by a mark ( ). (We're here assuming there are no more letters in the original E, other than the six mentioned). There is another way of looking at this. The expression E can be regarded as a 'function' of a set of unknown values, or 'variables', each such unknown corresponding to one of the letters occurring in E (in this case, one of x, y, z, a, b or c). Each time definite values are assigned to the unknowns, E itself takes on a definite value, void or mark.

Now $E_{xy(z)a(b)c}$ can be interpreted as *the value taken by the function E* when the values void, void, mark, void, mark, void are assigned to the unknowns x, y, x, a, b and c, respectively. An alternative notation would be

$E_{xy(z)a(b)c}$ = E(_ , _ , O , _ , O , _).

Now the factor (x y (z) a (b) c) of the normal form may itself be regarded as a function of x, y, z, a, b and c. This particular function can be characterized rather easily: it takes the value mark when x, y, z, a, b, c are assigned the values _ ,_ , O, _ , O, _ , respectively; and takes the value no-mark *in all other cases*. It is thus analogous to the Dirac delta-function, $\delta(\mathbf{x} - \mathbf{x_0})$, part of the tool-kit of physicists and Fourier analysts, which takes the value zero everywhere except at the point $\mathbf{x} = \mathbf{x_0}$.

The normal form thus exhibits E as a 'product of delta-functions'. Evidently, it will take the value mark precisely when one of the delta-functions takes the value mark. (We may note that it is never the case that more than one of them takes the value mark at the same time).

It follows, from all that has been said (in particular, from the criterion for any one of the set of possible delta-functions to be present in the normal form), that E and its normal form are identical as functions[3].

Moreover, if two expressions E and F are identical as functions, then they must be equivalent as CL-expressions; that is to say, it must be possible to transform E into F using the Elementary Moves. We

[3] This also follows from the more general principle that if E can be transformed into F by means of stringing together Elementary Moves, then E and F must be identical as functions. This is because none of the elementary moves can make any alteration to an expression, considered as a function. (To see this, you need to look at each of the three elementary moves in turn.)

see this as follows: let both E and F be transformed into their normal forms. Each of these transformations is reversible. But if E and F are identical as functions, their normal forms must be identical. So E can be transformed into F as follows: 1) transform E into its normal form 2) reverse-transform the normal form back into F.

Laws of Form expresses this intimate relation between the Primary Algebra or 'CL-Calculus' and the functional interpretation of CL-expressions by saying that the Primary Algebra gives a 'complete' account of the properties of the Primary Arithmetic: meaning that every equation that is a theorem of the arithmetic, is a demonstrable consequence in the algebra. Our approach has been slightly different. We have taken the Algebra as a given; then, by way of the splitting lemma, we have found that any CL-expression has a normal form; and at this point the interpretation of a CL-expression as a function is almost forced upon us – certainly, it is strongly suggested. Then, almost immediately, we see that if two expressions are identical as functions, they must be CL-equivalent (that is, interconvertible by allowed moves). This opens the door to *applications* of the CL-calculus, such as the ones described in §7 below.

§5 Pervasive Simplification

Let's return to the Splitting Lemma, writing it the wrong way round (as a 'Combination Lemma', in other words)

(a b) (a (b)) = (a).

Now one way of getting from the relatively complicated expression on the left, to the simple expression on the right, is to reverse each of the steps in our proof of the splitting lemma (§2). But when we look at this in detail we see that it involves quite of lot of wrapping and undeleting and copying, in other words, taking steps in the 'wrong', that is, complicating direction, in order to achieve the ultimate objective of simplification. Might it be possible to achieve the same result in a more 'unidirectional' fashion? This would be a 'direct' as opposed to a subtle, roundabout, ingenious method of simplification. For a first attempt in this direction, have a look at this sequence of moves:

(a b) (a (b)) = (a b) (a (a b))	[a copied into (b)]
= (a b) (a)	[(a b) uncopied from the second factor]
= (a)	[reverse Black Dog].

A slightly different explanation of the last step would be to say that (a) has killed (a b), which is possible because (a) has power of life and death over (a b), or, in symbols, (a) ≥ (a b). But now have another look at what the first and second steps accomplished between them:

(a b) (a (b)) → (a b) (a).

The term (b) in the second factor has been eliminated – killed, in other words. How has this come about? We could say that it has to do with the fact that the sub-expression (a b) is pervasively present through the whole expression, and that the term a (in the second factor) is pervasively present through the whole of the second factor. Suppose the term (a b) had always been inside the second factor, so we started with

((a b) a (b)).

Then (a b) and a between them could still have killed (b) by essentially the same copy/uncopy moves as before:

((a b) a (b)) → ((a b) a (a b)) → ((a b) a).

The outer circle plays no part here; we could equally write

(a b) a (b) → (a b) a (a b) → (a b) a,

which amounts to a proof that

(a b) a ≥ (b).

Thus the key fact is that (a b) and a *between them* dominate (b) (even though neither of them dominates (b) by itself). This could be exploited, *wherever* (a b) and a were in the original expression (so long as they were in a position to pervade the space of (b)); for example, if the expression looked like this:

A ((a b) B (C (D a (E (F (b)))))) .

Here (a b) and a are still pervasively present in the space containing F and (b) and so can do their work of killing (b), giving us

A((a b) B (C (D a (E (F (b)))))) = A((a b) B (C (D a (E (F)))))).

More generally, in an expression of the form

A (B (C (D (E (F X))))),

the influence of all the expressions A, B, C, D, E and F can be brought to bear (by pervasion) on the expression X; and if, between them, they have power over X, then X can be 'disappeared' so

A (B (C (D (E (F X))))) → A (B (C (D (E (F))))).

If, alternatively, they had power over (X) rather than X, then X in the original expression could be double-wrapped, making ((X)), then the (X) removed, leaving the empty circle ( ). Thus if ABCDEF has power over (X), X can be replaced by ( ). In the former case, when ABCDEF has power over X, X can be replaced by the void .

We get our first glimpse of a possible general method for simplifying expressions: we focus our attention on a particular sub-expression of the whole (X in the example above), then consider all the sub-expressions (in higher or equal spaces) which *pervade the space of X* (in the example, these are the expressions A, B, C, D, E and F). Then we see whether or not the expressions A, B, C, D, E and F between them dominate X. If they do, then X can be eliminated; alternatively, if they dominate (X), then X can be replaced by a mark. In both cases, we have effected a simplification of the whole expression; and having dealt with X we can go onto another sub-expression Y.

The question, whether A, B, C, D, E and F between them dominate X can be restated (by part (ix) of the Dominance Theorem, §3) as the question whether the 'test expression'

(A B C D E F (X))

is or is not void-equivalent. Similarly, the criterion for dominance of (X) is

(A B C D E F X) = .

A question that might occur to the reader is: How often can we expect either of these conditions to be satisfied? If seldom or never, our 'method of simplification' is never going to be much use. Now, in all honesty, I have no answer to the question, as it stands; all I have is a piece of evidence, that is, I think, quite encouraging. I will put it in the form of a theorem:

Theorem ('Elimination'): Suppose that a CL-expression P is *void-equivalent*, and let X be any sub-expression of P. Suppose that the sub-expressions which pervade the space of X are A, B, C .. F. Then, if X has even depth, the expressions A, B, ..F between them dominate X, so that P is equivalent to P with X replaced by ; if X has odd depth, the same expressions dominate (X), and P is equivalent to P with X replaced by ( ).

Proof. Suppose X has even depth in the expression

P = A(B(C....(F X) ...)),

and consider the 'test expression' that tests for dominance

T = (A B C....F (X)).

Given that P is void-equivalent, we can write

T= (A B C....F (X) P) = (A B C....F (X) A(B(C....(F X) ...))).

Now A, B, C .. F can all be stripped (by pervasive uncopy) out of the P-part of the expression inside the outer circle. Given that X is at even depth, we will be left with X enclosed by an even number of circles, equivalent to plain X. So

T = (A B C....F (X) X) = (A B C....F ( ) X) = ,

therefore X is indeed dominated by ABC..F. A similar proof will apply if X lies at odd depth. ***Q.E.D.***

Let's try an example. The following expression is known to be void-equivalent:

E = (((r p) (r **q**)) (r ((p) (q)))).

(This has the form E = (A (B)), with A= ((rp)(rq)), B= r ((p)(q)); but A and B are known to be equivalent, by **J2**, therefore A ≥ B, and E must be void-equivalent.) For our sub-expression X, that we hope to eliminate, let's focus on the first instance of the letter q (picked out in bold type). We note that this q is at depth 3, so it is (q) that we expect to be dominated. The pervading expressions (the 'uncles' – that is, the siblings of the father, grandfather, great-grandfather, etc. – of q) are r, from the same space as q; (r p), from the next space up; and (r ((p)(q))), from the next step up again. So the expected dominance relation is

r (r p) (r ((p)(q))) ≥ (q)

with criterion

(r (r p) (r ((p)(q))) q) = .

Uncopying the q from the deeper space, this becomes

(r (r p) (r ((p)( ))) q) = (r (r p) (r) q) = (r (r p) ( ) q) = (( )) = ,

confirming the dominance relation, and confirming that our target expression (the bold q) can be replaced by a mark. So E becomes

E = (((r p)(r ( ))) (r ((p)(q)))) = (r p (r ((p)(q)))).

Now, it is pretty obvious that E is void-equivalent, but just for practice let's launch an attack on the remaining instance of the letter q. This q is at depth 4, so we expect q itself to be dominated by its 'uncles', in this case, (p), r, and r p. So we expect to find

(p) r r p ≥ q,

and this is definitely true because the left-hand side is equivalent to the omnipotent mark, which dominates everything. So q can be eliminated, and E becomes

E = (r p (r)),

at which point it is easy to see that any of the remaining letters can be eliminated (the first r, or the p, replaced by a mark; the second r, by a void) leading straight to the void.

In sum, if E is void-equivalent, then it can be directly reduced to the void by 'attacking' its letters one at a time. The mode of attack is our souped-up version of uncopy or 'kill', where we gather up the pervading expressions or 'uncles' and bring them all to bear on the letter under attack.

Given an expression F of which we don't know whether or not it is void-equivalent, it follows from the Elimination Theorem that one way of finding out is to 'attack' one letter at a time. If ever the elimination of a letter fails, then we know that F is *not* void-equivalent. If it always succeeds, then eventually F will be reduced to the void.

So now we know that this method of simplification (attack on one letter at a time) is bound to work on a void-equivalent expression, and will eventually reduce the expression to the void. As for more general expressions, we don't know whether it will work or not: but it seems worth giving it a try.

Will we miss anything by concentrating our attention on eliminating letters, rather than larger sub-expressions? (There is a definite advantage to be had by attacking single letters, as we shall see below). The answer is No, as is shown by the following rather nice theorem, closely related to the Elimination Theorem:

Theorem. Suppose that a sub-expression X of a larger expression is dominated by an expression U (which will usually be the compound of the pervading 'uncle' expressions of X). Let Y be a sub-expression of X, and suppose that X has the form

X = A (B (.. (F Y) ..)).

Without loss of generality we may assume that Y lies at even depth in X. (If not, let Y be wrapped up, that is, replaced by ((Y)), and consider instead the sub-expression (Y), which *does* lie at even depth.) Then, in the presence of U, Y is dominated by its uncles and can therefore be voided (replaced by no-mark). (By 'in the presence of U' we mean that the expression U is pervasively present within X.)

Proof. The premise is U = UX , and what we seek to prove is

U X = U X_Y ,

where X_Y stands for X with Y replaced by void. Now,

(U A B .. F (Y)) = (UX A B ..F (Y)) [by the premise]

= (U A (B (.. (F Y) ..)) A B ..F (Y)) [form of X]

= (U Y A B ..F (Y)) [uncopy, and unwrap, using even depth of Y]

= . [uncopy, delete, unwrap].

Thus Y is dominated by the aggregate of the expressions pervading its space, and can therefore be voided in the given context. In other words,

U X = U X_Y .

Q.E.D.

A shorter version of the proof could be given, as follows: in the presence of U, X is 'effectively' void-equivalent. Therefore the elimination theorem can be applied to any of its sub-expressions. Q.E.D.

In the above, Y could be taken to be a simple letter, and we see that if X can be eliminated as a whole, it can also be eliminated one letter at a time.

Finally (in this section) we return to the advantage of trying to eliminate letters rather than nested sub-expressions. The criterion for U to dominate X is (U(X)) = . So to prove dominance, we have to prove the void-equivalence of an expression almost as complex as our original expression. But if X is a single letter, x, say, the criterion becomes simpler, as we see in the following lemma:

Lemma. U dominates x if and only if $U_{(x)}$ = ( ); similarly, U dominates (x) if and only if U_x = ( ).

Proof. i) the 'only if' part: we are given that U=Ux. In any equation, if we replace x by a value (mark or no-mark) throughout, the equation remains valid. So, replacing x, in the equation just given, by the mark, we have

$U_{(x)}$ = $U_{(x)}$ ( ) = ( ).

ii) the 'if' part: given that $U_{(x)}$ = ( ), we have

U (x) = $U_{(x)}$ (x) [(x) uncopied from U]

= ( ) (x) [premise]

= ( ) [delete].

Therefore (U (x)) = , and U ≥ x. The proof of the 'similarly' part goes along exactly similar lines. ***Q.E.D.***

We could, alternatively, prove the lemma by use of the 'Two Case' Theorem. By the Two Case Theorem, an expression E is void equivalent if and only if both E_x and $E_{(x)}$ are void-equivalent. Applying this to the expression E = (U (x)), we find

E_x = , $E_{(x)}$= ($U_{(x)}$).

Therefore (U(x)) is void-equivalent if and only if ($U_{(x)}$) is void-equivalent, in other words, if and only if $U_{(x)}$ is mark-equivalent. *Q.E.D.*

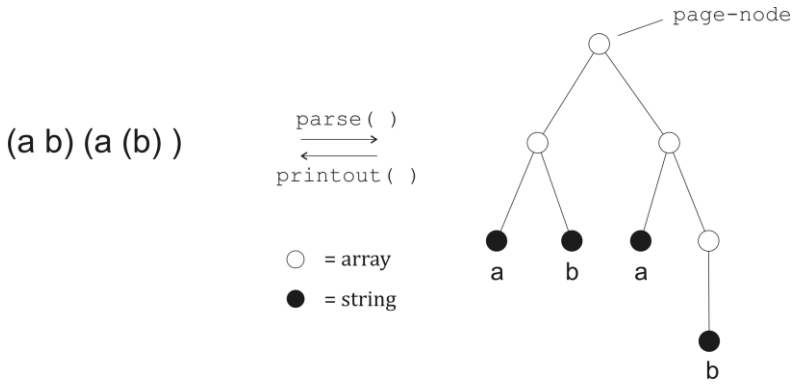
Thus the expression which needs to be tested is ($U_{(x)}$), or (U_x), both of which are simpler than the original expression, in that the letter x has been eliminated from them. This is crucial for turning our method into a recursive algorithm.

§6 An algorithm for simplification

We here give a brief account of an algorithm for simplification, based on the ideas of the last section. The algorithm has been realized in Javascript and can be tried out at the web-page: 'Simplification Algorithm for CL-expressions' [5].

The algorithm takes as input a character-string representing a CL-expression. As usual, the circles are represented by bracket-pairs; the 'letters' can actually be strings if required, and could be described more accurately as 'tokens'. If single letters are intended to be interpreted as tokens, they must be separated by spaces.

The first task of the program is to turn the input string into something more meaningful for the purposes of algebraic manipulation. This something is a 'tree', in the sense in which the word is used in Graph Theory: think of it as a family tree, each of whose nodes is either an array or a string. (For present purposes, a string does not count as an array.) The only exception is the node at the top of the tree, which we call the page-node: the page-node is necessarily an array, not a string. Apart from the page-node, the array-nodes correspond to the circles or bracket-pairs in our CL-expression; the string-nodes, to the tokens. The page-node (an array) contains the strings and (possibly occupied) arrays that make up our expression at the top level – as a page contains various tokens and (possibly occupied) circles in the usual pencil-and-paper representation of an expression. Any of the arrays (including the page-node) can be empty. Then the node has no descendants. Nodes which are strings also have no descendants. Otherwise, the nodes descending from an array-node represent the contents of the array. An example of the correspondence between expression and tree is given in the following figure:



The part of the program which performs this task of converting a string into a tree we call the 'parse' function. Another function, called 'printout', performs the inverse task of turning a tree (of the specified type) into a printable (or at least exhibitable) string; array-nodes are represented by bracket-pairs ( ), with their descendants (if any) between the brackets. Again the exception is the page-node: in this one case the brackets are omitted. So if the page is empty, it is represented by a space .

The next task is done by a function ‘substitute’. This takes as arguments a tree E, a token x, and a value v. The value v is 0 or 1, corresponding to no-mark or mark (note that the numeral zero is not to be confused with the empty circle O). The function goes through E, from left to right, as it were, and whenever it comes across a token it inspects it to see if it is an instance of x (so, strictly speaking, x is playing the role of ‘type’ rather than token). If it is, and if v= 0, then the token is removed from the tree. Its parent node loses a child; the parent-array loses a member. If, on the other hand, the token is an x, and v=1, then the token is replaced by an empty array. At the same time as doing one or other of these substitutions, whenever it finds an x, the function also looks out for opportunities to ‘delete’ or ‘unwrap’. In the first case, if it finds an empty array (or indeed creates one, in the case v=1), it immediately deletes all the siblings of the empty array-node (i.e. all the other nodes, if any, which have the same parent). In the second case, whenever it finds an array-node α (not the page-node) whose only child β is also an array-node, it does the equivalent of removing the two concentric circles. That is to say, the children (if any) of the only-child array β become instead the children of β’s grandparent, the parent of α. (The exception is when α is the page-node, for then it has no parent, and unwrap cannot take place.) As for α and β, they simply drop out of the picture.

If the token-type is set to ‘null’, then ‘substitute’ just goes through the tree doing any deleting or unwrapping that it can, so the expression is left purified of empty circles (except at page-level) and of concentric circle-pairs with nothing in between the circles.

Now for the main function, ‘simplify’, which takes a tree as input. This goes through the tree, from left to right, and whenever it finds a token, it tries to eliminate it. For this purpose, as the ‘focus’ of the function moves through the tree it maintains an expression called ‘context’ or C which consists of all the ‘uncle-expressions’ which pervade the space of the focus. When the focus falls on a token, x, for example, it discovers whether x is dominated by the context. This is a matter of testing the expression (C(x)), or rather ($C_{(x)}$), for void-equivalence. ($C_{(x)}$) is derived from (C) by using the ‘substitute’ function. Then ($C_{(x)}$) is tested for void-equivalence by passing it back into ‘simplify’ – recursively – and seeing what comes out at the other end. By the Elimination Theorem, if ($C_{(x)}$) is indeed void-equivalent, what comes out of the other end will be the void expression. If anything else comes out, ($C_{(x)}$) will be judged to be non-void. In the first case, when ($C_{(x)}$) = , x can then be removed (that is to say, its string can be removed from the tree). In the second case, an attempt can then be made on (x). If (C_x) turns out to be void, x can be replaced by an empty array. Otherwise, x has to be left as it is, and the focus moves on to the next token; and so on, until the focus emerges from the right-hand side of the tree. As with ‘substitute’, the ‘simplify’ function cleanses the tree of empty arrays and double circles as it goes along.

And that, essentially, is all there is to it. In the actual program, we use a specialized function ‘TVE’ to test a tree for void equivalence, rather than just passing it into ‘simplify’. As explained in the last section, if a tree is actually void-equivalent, then every attempt at elimination of a token, or circled token (whichever is at even level) must be successful. Just one failure is enough to prove that the input tree is not void-equivalent. So this streamlined function ‘TVE’ terminates itself more quickly, on average, than ‘simplify’. The longer it spends inspecting the input tree, the more likely the outcome will be successful – that is, that the answer will be ‘Yes, this tree is void-equivalent’, meaning that the token, or circled token, under attack, can be removed.

Hopefully, this account of the workings of the algorithm will help the interested reader to make sense of the source code of the web-page [5] (see also Appendix A.)

§7 Applications of the Elimination Algorithm

Here are some strings to try inputting (use copy and paste):

- `(a b) (a (b))`
- `(  ((r p)(r q))  ( r ((p)(q)) )  )`
- `(a b) (a(b)) ((a)b) ((a)(b))`
- `(a b c) (a(b) c) ((a)b c) ((a)(b)c) (a b(c)) (a(b)(c)) ((a)b(c)) ((a)(b)(c))`

The first two expressions are examples from §5, above. The last two both simplify to the mark, or empty circle. It is fun to remove one or more of the factors from these input expressions, and see what comes out. This next expression is taken from Bricken's paper [6] on *Lewis Carroll's Five Liars Problem*. To solve the problem, one must find a set of values such that this expression, regarded as a function of the variables a1, a2, b1, b2, etc. , takes the value ( ). With the expression as it stands, this is a near-impossible task; after being put through the algorithm, it becomes tractable.

- `(((a1 (((b1 b2)((b1)(b2)) (d1 d2)((d1)(d2))) (((b1 b2)((b1)(b2)))((d1 d2)((d1)(d2)))))) ((a1)((((b1 b2)((b1)(b2)) (d1 d2)((d1)(d2))) (((b1 b2)((b1)(b2)))((d1 d2)((d1)(d2)))))))) ((a2 ((c1 c2 e1 e2) ((c1 c2)(e1 e2)))) ((a2)(((c1 c2 e1 e2) ((c1 c2)(e1 e2)))))) ((b1 (((a1 a2)((a1)(a2)) (c1 c2)((c1)(c2))) (((a1 a2)((a1)(a2)))((c1 c2)((c1)(c2)))))) ((b1)((((a1 a2)((a1)(a2)) (c1 c2)((c1)(c2))) (((a1 a2)((a1)(a2)))((c1 c2)((c1)(c2))))))) ((b2 ((d1 d2 (e1)(e2)) ((d1 d2)((e1)(e2))))) ((b2)(((d1 d2 (e1)(e2)) ((d1 d2)((e1)(e2))))))) ((c1 (((a1)(a2)(d1)(d2)) (((d1)(d2)) ((a1)(a2))))) ((c1)((((a1)(a2)(d1)(d2)) (((d1)(d2)) ((a1)(a2))))))) ((c2 (((b1 b2)((b1)(b2)) e1 e2) (((b1 b2)((b1)(b2)))(e1 e2)))) ((c2)(((b1 b2)((b1)(b2)) e1 e2) (((b1 b2)((b1)(b2)))(e1 e2)))))) ((d1 ((a1 a2 e1 e2) ((a1 a2)(e1 e2)))) ((d1)(((a1 a2 e1 e2) ((a1 a2)(e1 e2)))))) ((d2 ((b1 b2 (c1)(c2)) ((b1 b2)((c1)(c2))))) ((d2)(((b1 b2 (c1)(c2)) ((b1 b2)((c1)(c2))))))) ((e1 (((a1)(a2)(b1)(b2)) (((a1)(a2))((b1)(b2))))) ((e1)((((a1)(a2)(b1)(b2)) (((a1)(a2))((b1)(b2))))))) ((e2 (((c1 c2)((c1)(c2)) (d1 d2)((d1)(d2))) (((c1 c2)((c1)(c2)))((d1 d2)((d1)(d2)))))) ((e2)((((c1 c2)((c1)(c2)) (d1 d2)((d1)(d2))) (((c1 c2)((c1)(c2)))((d1 d2)((d1)(d2)))))))))`

An easy way to interpret the simplified expression is to put it through the normalizing algorithm to which a link can be found at [5]. To understand what has been achieved here, refer to Bricken's paper[6].

Pigeons in Holes

A standard problem in the theory of Boolean Computation is to prove the Pigeonhole Principle (also known as Dirichlet's *Schubfachprinzip* or Drawer Principle) for a definite number of pigeons [7]. The principle says that if m pigeons have n holes to roost in, and if m>n, then at least one hole will have to be shared by more than one pigeon. We can turn this into a circle-and-token problem as follows. For simplicity, suppose we have 4 pigeons and 3 holes. We associate the pigeons with the letters a, b, c and d; the holes with the numbers 1, 2 and 3. Then the token b2, say, is to be regarded as a variable associated with the question whether or not pigeon b goes to sleep in hole 2. We make the choice that b2 takes the value O if pigeon b does indeed occupy hole 2; the value , if not.

Then the assertion that pigeon b has to sleep somewhere can be translated into the assertion that the expression

```
(b1 b2 b3)
```

must take the value void. (One at least of b1, b2 and b3 must take the value O.) On the other hand, to assert that if hole 2 is occupied by pigeon b, then it cannot be occupied by any other pigeon, is to assert that

```
( (b2) (a2 c2 d2))
```

must also take the value void. If b2= , the condition is satisfied; if b2=O, the condition becomes

```
a2 c2 d2 =  ,
```

implying that a2, c2 and d2 all severally take the value . If we want to include the condition that if pigeon b is in hole 2, then it cannot be in another hole, we replace this last condition by

```
( (b2) (a2 c2 d2 b1 b3) ) =  .
```

The condition for there to be an allocation of all 4 pigeons to the 3 holes, such that every pigeon has just one hole, and no hole has more than one pigeon becomes the condition that the following expression is void-equivalent

```
E =   (a1 a2 a3)(b1 b2 b3)(c1 c2 c3)(d1 d2 d3)
      ((a1 )(b1 c1 d1 a2 a3 ))
      ((a2 )(b2 c2 d2 a3 a1 ))
      ((a3 )(b3 c3 d3 a1 a2 ))
      ((b1 )(c1 d1 a1 b2 b3 ))
      ((b2 )(c2 d2 a2 b3 b1 ))
      ((b3 )(c3 d3 a3 b1 b2 ))
      ((c1 )(d1 a1 b1 c2 c3 ))
      ((c2 )(d2 a2 b2 c3 c1 ))
      ((c3 )(d3 a3 b3 c1 c2 ))
      ((d1 )(a1 b1 c1 d2 d3 ))
      ((d2 )(a2 b2 c2 d3 d1 ))
      ((d3 )(a3 b3 c3 d1 d2 ))
```

Can values be substituted for the variables in such a way as to satisfy the condition? Well, let's put it through the simplification algorithm. We find that E is equivalent to ( ) : it is mark-equivalent. No matter what values are substituted for the variables a1, a2, a3, b1, .. etc., E can never take the value void. Therefore the pigeon-allocation cannot be done. The Pigeonhole Principle has been proved, for m=4, n=3, by circle-and-token means.

It is tedious to construct the expressions for other values of m and n, by hand, so we have a program to do it for us (also linked to [5]). This actually constructs the expression F, equal to (E), rather than E, so a solution to the allocation-problem corresponds to F = ( ). When m>n, we find F = , so there is no solution. When m is less than or equal to n, on the other hand, solutions are possible, as indicated by the simplified expression. (If it had been void-equivalent, the algorithm would have discovered that fact.) To see what the solutions are explicitly, copy the simplified expression and put it through the normalizer (linked to [5]).

The standard case of m=n+1 becomes notoriously complex and time-consuming as n increases. This can be verified by putting in various values of n (up to no more than 9, I suggest) and noting the time taken by the simplification algorithm to prove that the n+1 pigeons cannot be fitted into the n holes.

Sudoku

Sudoku is a well-known puzzle, to be found in many daily newspapers, and elsewhere. A Sudoku solution consists of a 9x9 grid, each square holding a single-digit number between 1 and 9, such that no number appears more than once in any row or column of the grid, nor in any of the 9 3x3 sub-grids into which the large grid is divided by thicker lines.

We can translate this into 'logic' as follows. We work with $9^3 = 729$ tokens or variables of the form

```
aijk
```

with each integer i, j, k running from 1 to 9. Here i indicates a row, j, a column, k, a digit. The value taken by the variable corresponds to the question whether or not the cell situated in row i, column j, contains the digit k. We choose that the value 'mark' corresponds to 'Yes'. For example, if a853 is mark then the cell situated in row 8, column 5 is filled in with the digit 3.

Now we can translate the conditions for a solution. First of all, each cell must be filled in with something or other, so

```
(aij1 aij2 aij3 aij4 aij5 aij6 aij7 aij8 aij9)
```

must take the value , for each i and j. Secondly, if cell ij contains 3, say, then it can't contain any other digit, so

```
((aij3) (aij1 aij2 aij4 aij5 aij6 aij7 aij8 aij9)) =   .
```

Thirdly, if cell 57, for example, contains 3, then no other cell in the same row can contain 3:

```
((a573)(a513 a523 a533 a543 a553 a563 a583 a593)) =  ;
```

and the same for any other cell in the same column:

```
((a573)(a173 a273 a373 a473 a673 a773 a873 a973)) =  ;
```

and the same for any other cell in the same 3x3 box:

```
((a573)(a473 a483 a493 a583 a593 a673 a683 a693)) =  .
```

These last three conditions can of course be combined into one big condition:

```
((a573)(a513 a523 a533 a543 a553 a563 a583 a593 a173 a273 a373 a473
a673 a773 a873 a973 a473 a483 a493 a583 a593 a673 a683 a693)) =  .
```

There are 729 of these conditions, one for each of the variables. So the complete 'solution-expression' is 81 of the 'each cell must have something in it' expressions, 729 of the 'if this number then no other' conditions, and 729 of the 'no repetition' conditions, all placed side by side. For a solution, this enormous expression must take the value void.

There are many Sudoku solutions. We get an interesting puzzle by adding in several clues. These are cells already filled with certain digits. The art of setting a Sudoku puzzle is to choose the clues so that they lead to a unique solution; so all the other solutions are incompatible with the clues, in other words. If one of the clues is 'cell 49 contains the digit 1', to us that means that it is given that a491 takes the value mark; or, otherwise expressed, that

```
(a491) =  .
```

If we add the expression (a491) onto the end of the general solution-expression, then, if we can make the new expression void, we have a solution such that cell 49 contains 1, in other words, a solution that is compatible with the clue. In a good puzzle (with only one solution) there will be at least 17 clues, which means that we tack at least 17 expressions of the form (aijk) – usually about 25 of them – onto the end of the general solution-expression, in order to express the conditions of the puzzle.

The Sudoku page (again linked to [5]) does the donkey-work of constructing the expression corresponding to a given puzzle. In principle, we could plug this very large expression into the simplification algorithm, and see what came out. This turns out to be too much of a mouthful for our algorithm to chew. So we do a bit of streamlining of the algorithm, to make the task a bit easier.

Note that each clue, (a491), for example, reacts quite strongly with the general solution-condition. One way of putting it is that each occurrence of (a491) can be uncopied from the solution condition; alternatively, each occurrence of the token a491 can be replaced by a mark. So, for instance, the 'cell must have something' condition

```
(a491 a492 a493 a494 a495 a496 a497 a498 a499)
```

is vitiated: it disappears. The big 'no repetition' condition associated with a491 resolves as follows

```
((a491)(a411 a421 a431 a441 a451 a461 a471 a481 a191 a291 a391 a591
a691 a791 a891 a991 a471 a481 a571 a581 a591 a671 a681 a691))

→ a411 a421 a431 a441 a451 a461 a471 a481 a191 a291 a391 a591 a691
a791 a891 a991 a471 a481 a571 a581 a591 a671 a681 a691.
```

In effect, we have a whole lot of new clues, negative ones this time, telling us that the digit in cell 41 isn't 1, whatever else it is, and so on.

These new clues then react back on the solution-expression, each one vitiating several of the 'no repeat' conditions. Thus the clues give rise to a great reduction of the solution-expression, reducing it to more manageable proportions. This process of reduction I call 'washing'. To wash an expression, you search for any tokens, or circled tokens, floating around at page-level, uncopy them from the rest of the expression, then put them into a separate compartment. They can play no role in the further simplification of the expression, and can be brought back into the expression when the simplification of the trimmed-down expression has been completed.

Our Sudoku-solving program washes the puzzle-expression before putting it into the simplification algorithm. It also makes use of washing whenever the test-for-void-equivalence function TVE is called. We are in effect testing whether an expression Cx or C(x) is mark-equivalent. We have always used the fact that x or (x) can be uncopied from C in order to make the task easier, but the same goes for any other tokens that are present at the top level in C. So as a preliminary to testing Cx or C(x) for mark-equivalence, the expression can be washed and thus made smaller.

With this modification, and somewhat to our surprise, we have found that our Elimination Algorithm can indeed cope with most well-set Sudoku problems and come up with the answer, usually in under a minute, often in a few seconds. It has become one of life's pleasures to spend three quarters of an hour (or more) cracking one's head on a 'fiendish' Sudoku, then to hand it over to the algorithm to make mincemeat of it! There are in fact much speedier ways of getting a computer to solve Sudoku, by 'brute force' – essentially by trying out all possibilities for each cell in succession, and back-tracking

from blind alleys. The charm, if I may so put it, of the present approach is that it gets to the solution by a sort of 'reasoning' – the way Sudokus are meant to be done. Admittedly, the style of reasoning is not quite the same as the human style of reasoning!

Logic Puzzles

A similar approach can be taken to a popular kind of logic puzzle, sometimes called a 'logic grid puzzle'. Here we have a number of individuals each of whom has several attributes or properties, each with a certain value, e.g. name, age, home town, favourite dish. Again, none of the values can be shared by more than one individual, and certain clues are given, e.g. Brian is 24, Nigel lives in Newark and likes macaroni cheese, and so on. The problem is to work out the attribute-values of all the individuals.

Let's take an example, and see how we can deal with it using circles and tokens. Here are the clues:

> The man from Burnley likes fish and chips, but is not ginger-haired
> Ron has brown hair
> The chap who likes steak and kidney pudding is from Bolton
> Reg is not blonde, nor is the tripe-lover
> Don is not from Burnley
> ---
> Who lives in Burnley and who lives in Bolton?

We start by listing the property values occurring in the problem, and arranging them in a table:

	A	B	C	D
	Name	**Home town**	**Favourite food**	**Hair-colour**
1	Ron	Burnley	Fish and chips	Ginger
2	Reg	Bolton	Steak and kidney	Brown
3	Don	?	Tripe	Blonde

It seems that we are dealing with three individuals. Only two place-names are mentioned, but that doesn't matter. In each column the values are listed in the order they are mentioned. The fact that Ron, Burnley, Fish and Chips, Ginger appear in the same row does not imply that they are associated. Indeed, the true associations are what we mean to discover. The letters along the top of the table, and the numbers down the left-hand side enable us to refer to any cell of the table in a concise way.

We assign a token to each possible association between cells. Thus B2D1, for example, corresponds to the potential association of cells B2 and D1. (To avoid duplication, we require that the two letters in a token appear in alphabetical order.) If the token, regarded as a variable, takes the value 'mark', then the link is realized or 'true'; if the value is void, then the link is unrealized or false. (We could have made the opposite choice.) So we can restate the first clue as follows:

```
(B1C1) B1D1 =  .
```

For the expression on the left to be void-equivalent, B1C1 must take the value mark, and B1D1 must take the value void. In other words, B1 and C1 must be associated, B1 and D1 not. The whole set of clues becomes

```
(B1C1) B1D1 (A1D2) (B2C2) A2D3 C3D3 A3B1 =   .
```

We now have to consider the general conditions of the puzzle. Each cell in the first column (say) has to be associated to a cell in the second, third, and fourth columns. So, for example,

```
(A1B1 A1B2 A1B3) =  ,

(A1C1 A1C2 A1C3) =  ,

(A1D1 A1D2 A1D3) =  .
```

But if A1 is associated to B1 then it cannot be associated to B2 or B3; nor can any other A-cell be linked to B1:

```
((A1B1)(A1B2 A1B3 A2B1 A3B1)) =   .
```

Finally, links are transitive. For example, if A1 is linked to B3, and B3 to D2, then A1 must be linked to D2. We express this condition as follows:

```
((A1B3)(B3D2) A1D2) =  .
```

In the case under consideration (4 letters, 3 numbers) there will be 18 conditions of the first type, 54 of the second, and 81 of the third[4]. Again, it would be tedious to write them all out by hand, but we can let a computer do the work. As each condition says that a certain expression takes the value 'void', we can express the aggregate of all the conditions by asserting that the giant expression, obtained by juxtaposing all the little ones (including the clues), takes the value void. Then we see what we get by simplifying the giant expression.

To try this out, go to Logic Puzzle web-page (linked to [5]), select 4 attributes and 3 individuals, and paste in the clue-expression as given above. A second, less trivial example of a puzzle is given in Appendix B.

[4] The cyclic permutations of the transitivity condition between A1B3, B3D2, A1D2, namely (A1B3 (B3D2) (A1D2)) and ((A1B3)B3D2 (A1D2)), turn out to be logically redundant and can be omitted.

Conclusion

The mechanical simplifier is certainly fun to use, and seems to be effective on a range of recreational logic problems. It may well have serious applications as well (as suggested by Bricken's paper [1]). It is worth remarking that to apply the CL-calculus to puzzles there is no need to go via a description in terms of Boolean Algebra as such. The unknown values have always been mark and no mark, not True and False. Usually, there is no need to speak in terms of truth and falsity; marks correspond directly to conditions, not to the judgement that such-and-such a statement of condition is true or false. This confirms one's feeling that with *Laws of Form* one is working at a more basic level, closer to the ground of things, than one is with the logic of truth and falsity.

I am not sure if I am any nearer to the original goal of my talk at LoF50, which was to gain an intuitive and direct understanding of the Calculus of Circles and Letters, without reference to its being 'taken out of' the more primitive Calculus of Circles, as an algebra taken out of an arithmetic. Perhaps its reframing as the Calculus of Containment and Sameness is a small step in the right direction; and Bricken's concept of Pervasion, and its corollary, the pervasiveness of dominance, with its rich consequences, is surely another clue to the mystery.

References

[1] William Bricken, 'Boundary Logic: The Design of Computation', 2019,
http://iconicmath.com/new/lof50presentation/lof50presentation-190731/

[2] G Spencer Brown, *Laws of Form*, Allen and Unwin, London, 1969

[3] Louis H Kauffman, 'Robbins Algebra', May 1990,
http://homepages.math.uic.edu/~kauffman/RobbinsAlgebra.pdf

[4] William Bricken, 'Distribution is not axiomatic', October 1986,
http://www.wbricken.com/pdfs/01bm/02logic/01transforms/04distribution.pdf

[5] George Burnett-Stuart, 'Simplification Algorithm for CL-Expressions', 2020,
http://www.markability.net/pervade

[6] William Bricken, 'Lewis Carroll's Five Liars Puzzle', June, 2009,
http://iconicmath.com/mypdfs/bl-fiveliars.090625.pdf

[7] William Bricken, 'Computational Complexity And Boundary Logic', January 2002
http://iconicmath.com/mypdfs/bl-complexity.020121.pdf

Appendix A: Javascript code for the Simplification Algorithm

We give here the Javascript part of the source code for the web-page at [5].

```
<script>

function parse_whole(expr)    //expr is a string, returns a tree
{
        var i;  //global within parse_whole; records progress through expr

        function parse()
        {
                var tree=new Array();
                while (i<expr.length)
                {
                        var ch=expr[i];
                        switch(ch)
                        {
                                case '(':       i++; tree.push(parse());      break;
                                case ')':       i++; return tree;
                                case ' ':       i++; break;
                                default:        tree.push(token()); break;
                        }
                }
                return tree;
        }

        function token()       //returns a token which is a string not containing a space
        {
                var tok='';
                tok+=expr[i];
                i++;
                while (i<expr.length)
                        switch (expr[i])
                        {
                                case '(':
                                case ')':
                                case ' ': return tok;
                                default: tok += expr[i]; i++;
                        }
                return tok;            //in event end of input string
        }

        function bracket_error()
        {
                var level=0;
                for (i=0;i<expr.length;i++)
                {
                        switch(expr[i])
                        {
                                case '(': level++; break;
                                case ')': level--; break;
                        }
                        if (level<0) return true;
                }
                return (level>0);
        }

        if (bracket_error()) return null;
        else
        {
                i=0;
                return parse();
```

```
        }
}


function printout(tree)       //returns a string
{
        var str='';
        for (var i=0; i<tree.length;i++)
            {
               if (typeof(tree[i])==='string') str+= tree[i]+' ';
               else
               {
                       str+='(';
                       str+=printout(tree[i]);
                       str+=')';
                }
            }
        return str;
}

function copy(tree)           //used in both simplify() and TVE()
{
        var newtree;
        if (typeof(tree)==='string') return (newtree=tree);
        newtree= new Array();
        for (var i=0; i<tree.length;i++) newtree.push(copy(tree[i]));
        return newtree;
}

function substitute(node,ch,value)

/*-------------------------------------------------------------------
        Returns the 'value' of a circled array,
        0 if the array contains a mark,
        1 if it is empty,
        2 if the array has one element, which is an array (ripe for Unwrap),
        null otherwise.
        Each occurrence of ch to be replaced by value (=0 or 1, corresponding to _ or O)
-------------------------------------------------------------------*/

{
        if (typeof(node)==='string')
        {
                if (node==ch) {return value;}
                else {return null;}
        }
        else
        {
                for (var i=0;i<node.length;i++)
                {
                        switch(substitute(node[i],ch,value))
                        {
                                case 0:        //child is VOID-EQUIVALENT
                                        node.splice(i,1);     //remove this child from the array
                                        i--;
                                        break;
                                case 1:        //array contains a MARK
                                        return 0;  //returned value is _
                                        break;
                                case 2:        //UNWRAP
                                        var grandchild=node[i][0];
                                        node.splice(i,1);
                                        for (var j=0;j<grandchild.length;j++)
                                                node.splice(i+j,0,grandchild[j]);
                                        i = i-1+grandchild.length;
                                        break;
                        }
```

```
                }
                if (node.length==0) return 1;
                else if ((node.length==1)&&(typeof(node[0])!=='string')) return 2;
                //leads to UNWRAP
                else return null;
        }
}



function TVE(expr)     //Test for Void-Equivalence
{
        if (reduce(expr, [], 0)==1) return 0;
        else return 1;

        function reduce(e,context,level)
        {
                if (typeof(e)==='string')
                {
                        var test = [copy(context)];
                        var j = (level-1)%2;
                        switch(substitute(test,e,1-j))
                        {
                                case 0: return null;
                                case 1: return j;
                                case 2:
                                case null:     if (reduce(test,[],0)==1) return j;
                                //in other words, if TVE(test)==0
                                        else return null;
                        }
                }
                else
                {
                        for (var i=0;i<e.length;i++)
                        {
                                var c=context.concat(e);
                                c.splice(context.length+i,1);
                                switch( reduce(e[i], c, level+1))
                                {
                                        case 0: e.splice(i,1); i--; break;
                                        case 1: return 0;
                                        case null:             // letter could not be eliminated
                                                return null;
                                }
                        }
                        if (e.length==0) return 1;
                        else return null;
                }
        }       //end of reduce

}

function simplify(node, context)
{
        if (typeof(node)==='string')
        {
                var test;
                for (var j=0;j<2;j++)
                {
                        test=[copy(context)];
                        switch(substitute(test,node,1-j))
                        {
                                case 1: return j;
                                case 2:
                                case null: if (TVE(test)==0) return j;
                        }
```

```
                }
                return null;
        }
        else
        {
                for (var i=0;i<node.length;i++)
                {
                        var c=context.concat(node);
                        c.splice(context.length+i,1);
                        switch( simplify(node[i],c))
                        {
                                case 0: node.splice(i,1); i--; break;
                                case 1: return 0;
                                case 2:         //UNWRAP
                                                var grandson=node[i][0];
                                                node.splice(i,1);
                                                for (var j=0;j<grandson.length;j++)
                                                        node.splice(i+j,0,grandson[j]);
                                                i+=-1+grandson.length;
                                                break;
                        }
                }
                if (node.length==0) return 1;
                else if ((node.length==1)&&(typeof(node[0])!=='string')) return 2;
                //leads to UNWRAP
                else return null;
        }
}

function process()
{
        let estr=document.getElementById("expr").value;
        document.getElementById("1").innerHTML = estr;
        document.getElementById("2").innerHTML = "";
        document.getElementById("3").innerHTML = "";
        let f=parse_whole(estr);

        if (f==null) document.getElementById("2").innerHTML = 'UNBALANCED BRACKETS';
        else
        {
                if (simplify(f,[])==0) f=[[]];
                document.getElementById("3").innerHTML = printout(f)+"<br><br>OK";
        }
}
</script>
```

Appendix B: A second example of a Logic Puzzle

Five friends from the Liverpool area were all born in the same year, in successive months from May to September. Each one has a favourite band from the 80's. We are given the following:

- Beattie is 2 months older than the girl who likes Duran Duran
- Cathy has her birthday 2 months earlier than the girl from Maghull
- The Clash-fancier is a month older than the one who likes Frankie goes to Hollywood, and two months older than the girl from Aintree
- The girl from Chorley was born two months before Debbie, but one month after the one who likes U2
- Ally is 2 months older than the one who likes Adam and the Ants, who is not from Aintree
- Ellie is from Crosby

The question is: Which band does Beattie like?

Here's the table of property-values:

	A	B	C	D
	Name	**Home town**	**Favourite band**	**Birthday**
1	Beattie	Maghull	Duran Duran	May
2	Cathy	Aintree	The Clash	June
3	Debbie	Chorley	Frankie/Hollywood	July
4	Ally	Crosby	U2	August
5	Ellie	?	Adam and the Ants	September

We can express the clues as follows:

```
A1=C1-2     A2=B1-2     C2=C3-1=B2-2      B3=A3-2=C4+1      A4=C5-2
B2C5  (A5B4)
```

If the month associated with A1 is two months before the month associated with C1, then A1 cannot be linked to August or September; and A1=May implies C1=July, A1=June implies C1=August, and A1=July implies C1=September. We can translate all this as follows:

```
A1D4 A1D5 ((A1D1)C1D3) ((A1D2)C1D4) ((A1D3)C1D5) =   .
```

Treating the 2nd, 3rd, 4th and 5th clues in the same way, we obtain for the whole set of clues:

```
A1D4 A1D5 ((A1D1)C1D3) ((A1D2)C1D4) ((A1D3)C1D5)

A2D4 A2D5 ((A2D1)B1D3) ((A2D2)B1D4) ((A2D3)B1D5)

C2D4 C2D5 ((C2D1)((C3D2)(B2D3))) ((C2D2)((C3D3)(B2D4)))
((C2D3)((C3D4)(B2D5)))

C4D3 C4D4 C4D5 ((C4D1)((B3D2)(A3D4))) ((C4D2)((B3D3)(A3D5)))

A4D4 A4D5 ((A4D1) C5D3) ((A4D2) C5D4) ((A4D3) C5D5)

B2C5 (A5B4) =  .
```

Now paste the clue-expression into the Logic Puzzle Solver (with 4 attributes, 5 individuals). With luck, and after a pause of a few seconds, you will ascertain that the band liked by Beattie is The Clash. And she comes from Chorley.