Mathematics has always had problems with *Nothing*. The problem at first was that no one thought to symbolize it. After mathematicians finally did, the problem was that they couldn't stop. Programmers inherited these sundry synonyms and augmented them with algorithmic kinds of *Nothings*. Such a variegated crop of vacuities is costly in both disciplines not just computationally, but especially conceptually: disparate kinds of *Nothings* inhibit us from unifying disparate kinds of numbers. We can unify by reverting to the primordial economy of just one *Nothing* in *all* disciplines, achieved by following a few simple principles.

Fixing *Nothing*

Gary Harper

Mathematics, for the entire several thousand years of its existence, has always had problems with *Nothing*, called *sifr* by the ancient Arabs.[1] The problem for the first few thousand years was that no one thought to symbolize it. Why, as children sometimes ask, would *Nothing* need to be symbolized?—you can just leave an empty space, as was often done. Anyway, if you symbolize *Nothing* it becomes *Something* doesn't it? Well no, it merely becomes *represented by Something*, perplexing perhaps, but conceivable. The *Something* representing *Nothing* eventually settled on "0", whose vacuous interior connotes emptiness—that symbol is essentially an empty box. After it arose, *sifr* slowly became a name for *it*, rather than *Nothing*.

Such transfer of meaning became permanent when *sifr* was in due course transliterated into various alphabetic languages as *zero*. When a foreign word is borrowed for a particular purpose, it becomes specialized to that purpose and loses its original meaning. Most people now think that zero is a number, *and only a number*; few know that "*zero*" originally meant *Nothing*, a concept transcending the narrow realm of number.

Both "zero" and "0" are are linguistic maps, and *Nothing* had been their tenuous common (no) territory during the last half of the first half of the Common Era. Now however, "zero" the *word* and "0" the *symbol* denote 0 the *number*—***Nothing has****, ahem,* ***disappeared***, which testifies to the real perplexity of representing it with *Something*. After brief experience with that, humankind reverted to familiar *Something* representing *Something Else*. And that has been the problem for the last few hundred years: absent *Nothing*, recent mathematicians have generated diverse kinds of empty boxes, with little understanding that, ***under summary discipline*** (coming up)*, they all denote the same (no) thing*.

Such vacuous redundancy is computationally costly, of course, but the problem is more subtle and pernicious than that. Disparate kinds of *Nothings* inhibit unification of disparate kinds of numbers. Such unification can make most computations more efficacious, not just vacuous ones. Unification has been further inhibited by computer languages, which augment the various mathematical vacuities with algorithmic kinds. Most programmers believe, and most mathematicians believe, that their numerous *Nothings* have nuances that make relevant distinctions. That may be so within the current disjointed computational hodgepodge, but it will not be so after…

Unifying vacuous mathematics

The mathematician who did most to get various *Nothing*s started, accidentally, was William Rowan Hamilton. He became fascinated in the early 1830s by the way complex numbers can articulate the geometry of a plane; but he was vexed by the counter-intuitive nature of $\sqrt{-1}$. To remove vexation, he introduced *couples*:

> In the THEORY OF SINGLE NUMBERS, the symbol $\sqrt{-1}$ is *absurd*, and denotes an IMPOSSIBLE EXTRACTION, or a merely IMAGINARY NUMBER; but in the THEORY OF COUPLES, the same symbol $\sqrt{-1}$ is *significant*, and denotes a POSSIBLE EXTRACTION, or a REAL COUPLE, namely, (as we have just now seen) the *principle square-root of the couple* $(-1, 0)$.[2]

His couples implicitly treat 1 and $\sqrt{-1}$ as perpendicular unit coordinate vectors in a plane. This ploy has become commonplace, but it seemed radical to Hamilton because planar coordinates allowed him to divert attention away from the IMPOSSIBLE EXTRACTION of raw unpackaged IMAGINARY $\sqrt{-1}$ and focus instead on a POSSIBLE EXTRACTION on a packaged pair of REAL coordinates.

Hamilton became so enamored with his focus-diverting packaging that he wanted to extend it to REALED *triplets*, able to articulate physical space in a similar way. After spending a decade rassling with that idea he finally realized, in late 1843, that triplets won't work, but *quaternions*[3] will. With that success he hoped to generalize to *polyplets*, a precursor of our modern *tuples*. He discarded such packaged coordinates before they become popular—the apparent dimensional incongruity between his new 4-algebra and its slowly unveiled 3-geometry persuaded him to revert to *explicit addition* of three scaled coordinate vectors.

Few subsequent mathematicians had such qualms. Most began to embrace packaged coordinates, thereby embracing the various kinds of *Nothing* Hamilton had inadvertently introduced. His couple $(0, 0)$ is a *pair of Nothings*: no 1 and no $\sqrt{-1}$. Is two *Nothings* more, or less, vacuous than one? That question seldom occurs to mathematicians because $(0, 0)$ is, to them, a *2-tuple of zeros*, ***syntactically*** different from one zero. The 3-tuple of zeros, $(0, 0, 0)$, is syntactically different from both of them, and so on, which generates endlessly many vacuities. And that is only the vacuous beginning: there are also endlessly many vacuous matrices, tensors, and other emptinesses we shall peer at.

They are all *syntactically* different but *semantically* the same under addition and multiplication: addition with a vacuity has no effect; multiplication with one vanishes. The catch is that not just any old vacuity can add or multiply with any old element—the kind of vacuity has to match its kind of element. This conundrum arose because mathematics has diverted attention from the *contents* of its packages to their *form*—mathematics has become mostly about *maps*, little about *territory*.

Symbol maps in mathematics have peculiarities that have acquired significance absent in the territories they had originally represented. It is as if some astronauts had taken three different maps of San Francisco when they went on a tour of the galaxy; and their progeny, far from Earth, discovered them and considered them three different cities. After all, the Pacific Highway is red in one map, blue in another, and grey in the other. If a map were a city, *rather than a representation of a city*, then the astro-children would really have three

cities with different-colored highways. It is hard to imagine real children making a blunder like that.

That kind of blunder requires Hamilton-class sophistication: *intentionally ignore the territory* like he did. If a tuple of zeros were *Nothing*, *rather than a representation of Nothing*, then there would be an infinite number of *Nothings*; and that blunder is made constantly in mathematics. Its antidote is to explicitly distinguish map from territory.

We humans, for good evolutionary reasons (survival),[4] seldom do so. However, when territory has mostly evaporated, as it has in mathematics, then it becomes crucial to consider what the maps had originally represented. Tuples, matrices, tensors and other boxes of scalars had represented *geometric* territory but their packaging peculiarities are absent therein. Fortunately, the aforementioned ***summary discipline*** provides a straightforward way to discard such irrelevances:

Use explicit addition to package algebraic summary.

This is our first and most important unifying principle. It is what Hamilton reverted to, so let us call it *Hamilton's Reversion* to round out *Hamilton's Principle*, *Hamilton's Characteristic Function*, *Hamilton's Equations of Motion*, the *Hamiltonian*, the *Hamilton-Jacobi Equation*, *Hamiltonian Mechanics* … etc. Hamilton's Reversion unifies by restoring lost algebraic relations, thereby recovering some of the geometric territory.

A tuple like (x, y, z) is an *implicit* algebraic summary because, in Analytic Geometry for example, it *tacitly* means $x\mathrm{i} + y\mathrm{j} + z\mathrm{k}$, where i, j, k are the usual coordinate vectors (minted of course by Hamilton, for use in his quaternions). Those vectors constitute the most obvious territory lost by the tuple; but also lost, more importantly, is the addition connecting them.

Tuples are seductive because they seem like a succinct way to hide tedious coordinate addition. Tuples, in fact, are a succinct way to *implement* such addition in software; and they work okay within the algebra if you are adding homogenates like i, j, k. But what if you want to add disparates, say a scalar and a vector, as Hamilton wanted to?[5]

Many mathematicians would say, '*Well, you just can't. They are different kinds of things. Vector spaces only allow vectors to add with vectors, scalars with scalars. Study the axioms.*' Here you see the pernicious influence of tuple use at its worst: traditional tuple homogeneity, and segregation by tuple length, have inhibited most of us for over a century from adding inhomogeneous things, an operation that is actually *very* expressive.

There is excellent reason to add a scalar and a vector. The scalar can represent time; the vector can represent space. That was Hamilton's original idea (and one reason he repented of using tuples). The summary package of both of them can represent spacetime, something now routinely done in Geometric Algebra. That language uses mixed addition to package things; and exiles tuples, matrices and other such boxes to the implementation. Such discipline restores lost algebraic relations and permits just one *Nothing*. It shall be enlightening to see why there can be just one, and what its properties are (not).

First, consider the tuple (0, 0, 0). The implicit addition therein would be explicitly expressed as $0\mathrm{i} + 0\mathrm{j} + 0\mathrm{k}$, which the rules of addition immediately condense to $0 + 0 + 0 = 0$. That is obvious even to a child, who would correctly conclude that $(0, 0, 0) = 0$, at least *semantically*. This equation asserts that the vector represented by the 3-tuple has vanished,

really vanished. Many mathematicians would consider it a crude sophomoric blunder owing to its *syntactic* mismatch across the equals sign. Are you beginning to see how pernicious tuples can be?

If not, consider mixed addition. Under such addition, it is essential that those things that vanish ***really vanish***, and do not leave an irrelevant residue. Such residue would prevent dimension from being well defined. To see this, suppose you use explicit addition to package a scalar, 7 say, a vector v, and a *B*ivector *B* to form the M*ultinumber*…

$$\underline{\mathrm{M}} = 7 + \mathrm{v} + B$$

M's dimension, since this sum cannot further coalesce, is clearly composite. In fact…

$$\dim(\underline{\mathrm{M}}) = \{0, 1, 2\}$$

…where 0 is the *geometric dimension* of the scalar (perplexed by that vacuity?—we shall soon rassle with it), 1 is the dimension of the vector, and 2 is the dimension of the bivector. Curly braces indicate a *list* of dimensions, necessary for mixed addition.

Now suppose you augment M with vector u, symbolized as M + u. What is the dimension of this new package? On first thought the augmenting vector would merely augment dimension, meaning that dim(M + u) would be {0, 1, 2, 1}. On second thought that can't be right because vector u added with vector v almost always coalesces to a single vector. Hence, dim(M + u) would almost always be the original dimension of M, namely {0, 1, 2}. Here you see that, *to define dimension well*, the dimension operator must operate on a *minimal form*. Which brings up our second unifying principle, ***minimal discipline***:

Keep algebraic summary in minimal form.

That is why I repeated "*almost always*" twice. If u were equal to –v, then the vector sum (v + u) would vanish under minimal discipline, causing dim(M + u) to produce an even more minimal {0, 2} dimension. Notice that minimal discipline *automatically precludes zero tuples* because they are not in minimal form, which for them is 0. In fact, minimal discipline effectively precludes summary tuples of any fixed length because any zero entry that arises would put the tuple in non-minimal form. If all of this seems obvious, then you are one of those fortunates who has not yet been indoctrinated with twenty-first century mathematical practice.

Standard practice asserts that vector u added with vector v ***always*** produces another vector—no *almost* about it. That vector may sometimes be the "zero vector" (0, 0, 0), *which is, pedantically, still a vector*. Standard pedantry would require dim(M + u) to ***always*** be {0, 1, 2}, even if the vector sum happened to be the "zero vector". That "vector", as an obdurate 3-tuple, *fails to vanish*. (How could it vanish?—its tuple has lost the addition that would annihilate it.) A zero-vector's failure to vanish is whimsy, like the grin that remains after Cheshire Cat has vanished. Sadly, it has become *formal* whimsy, which prevents dimension from being well defined.

Under standard practice zero is astonishingly reluctant to vanish. For example, suppose you were to add –v, –*B* and –7 to M in that order. What is the dimension of this new package? If you think it would be {0}, then you, enlightened tho you may have become, are nonetheless still suffering from the deleterious effects of tuple use. {0} is the dimension of what remains after the first two additions, namely 7: the vector and bivector

have vanished. When –7 is finally added, *that scalar also vanishes*, leaving a dimension of {}, the dimension of 0, namely *no possible dimension*.

Standard practice, when overridden by standard fudges, does occasionally allow vectors to vanish—"become zero"—but it ***never*** allows scalars to do so. Rather, they also "become zero", *which is always considered a scalar, a "real number"* (because it long ago became an obdurate *number*, rather than *Nothing*, as it had begun).

When dimension becomes well defined, zero is never a real number! *And it is never any other kind of number either*. Numbers have properties, like dimension for example, or magnitude, or direction… Zero has no properties at all. ***Zero***, when properly understood as *Nothing*, ***transcends numbers***—many things other than numbers can vanish, like sets for example, or matrices, textual strings, bananas, ignorance about zero … on and on.

Of course bananas and ignorance lack the formalities of geometric numbers, dimension in particular. Keeping that well defined is an obvious goal of minimal discipline. This is *very* seldom done in current practice. To keep dimension well defined, you need an operation that *formally generates dimension* from a primitive one. Such an operation was introduced by Hermann Grassmann in his 1844 magnum opus, *Lineal Extension Theory*.[6] At that time he called it *extension* or the *outer product*. Later, in his 1862 attempt to appeal to mathematicians, he renamed it as the more grammar-focused *combinatorial product*.

The *syntax* of that product has made its way into mainstream mathematics via Cartan's exterior calculus, but its diverse dimension-augmenting *semantics* has not; so few mathematicians have acquired well-defined multi-dimension. Among those who have, even fewer have acquired the distinction between the {0} dimension of a scalar and the {} dimension of 0.

That distinction, I know from long experience trying to explain it, baffles many people, even exceptionally smart ones—even aficionados of Geometric Algebra still have trouble with it.[7] The main difficulty seems to be our thousand-year loss of zero-as-*Nothing*, which has provided little motivation to *establish formal context* for what has vanished.

The curly braces in "dim(7) = {0}" provide a formal context for the *Nothing* within. The particular context there is *dimension*, induced by the dim() operator. A zero *dimension* refers to a scalar; and means that it has no locus like a vector has, like a bivector has… Among geometric elements, a scalar is unique not only in lacking locus, but also in lacking ability to change dimension under multiplication. All of that *lack* is encoded by lack of geometric dimension, which is the fundamental property a scalar has *Nothing* of.[8] Here you see a ***contextual*** *narrowing of Nothing*, which brings up our next unifying principle:

Use context to narrow *Nothing*.

Context has already been observed narrowing *Nothing*—that is what Hamilton's couples had done accidentally. Unfortunately, they violate summary discipline. By contrast, the embraced context produced by the dimension operator is merely a list, not an algebraic summary. Such context, to focus exclusively on dimension, is necessarily narrower than the generic algebraic context in which it arises. However, it is all too easy to narrow *Nothing* needlessly, as Hamilton did, as most everyone else still does. Which brings up our final unifying principle:

Never narrow *Nothing* ***needlessly.***

An immediate consequence of these principles is that there can be only one *Nothing*. This has always been *semantically* obvious: *Nothing* has no properties by which it could be classified into various kinds. Now it is *syntactically* obvious: when *Nothing* becomes distinguished solely by context, then there can be only one.

For that purpose, let us use an unmistakable empty box, □, and assert that it now reverts to its original trans-numeric meaning, *sifr*. To try to enforce that, suppose we call it *nada*, a pithy word more widely understood than *sifr*. Let us see how *nada* would work in other contexts.

The other context for *nada* most familiar in mathematics is Set Theory. In that context *nada* is the so-called *empty set*, symbolized as ∅, whose duplicity would contravene a singular *Nothing*. Can we replace that symbol with universal □? Yes, operational context *implicitly* disambiguates it like so: when *nada* is engaged by addition or multiplication it means *no numeric quantity* (*no-number*); when *nada* is engaged by union or intersection it means *no set elements* (*no-set*, not even an "empty" one. If that startles you, it is further testimony to how pervasive our loss of *no-thing* has become).

There is no need to *explicitly* distinguish, either conceptually or algebraically, between no-number and no-set. In either case exactly what had vanished is no longer relevant: it's gone, it left no grin. In either case algebraic discipline allows only one *nada*; and in either case that vacuity either gets ignored or annihilates. Best of all, it obeys our final two unifying principles: its operational context automatically narrows it, but not needlessly.

The use of a single □-as-*nada*, when followed faithfully, does generate some strangeness. Set Theory also uses {} to mean *nada*, another duplicitous vacuity that needs to be replaced by universal □. This works fine: □ successfully unifies the two superfluous vacuities ∅ and {}. However, {} had *also* been the dimension of *nada*, another vacuity that needs to be replaced by □. Wouldn't such replacement obliterate important distinctions? Actually not—it amounts to discarding a no-longer-relevant box, but that would be confusing unless carefully expressed. Symbolically, the replacement would generate these equations:

$$\dim(\square) = \square$$

$$\dim(7) = \{\square\}$$

The context in which nada finds itself successfully narrows it. In this case *exactly-what-had-vanished* ***is*** relevant: The un-embraced □ says that *nada* has *no context for dimension*; the embraced {□} says that a scalar *has no dimension*; even tho, *as a number*, it does have context for one.

It is all too easy to read these equations like so: 'The dimension of *nada* equals *nada*; and the dimension of 7 equals *nada*'. Such rendering blatantly disregards context, a violation of the unifying principles. To properly regard context, avoid saying syntactic 'equals *nada*' and instead say semantic 'no *whatever-there-is-nada-of*' as just explained. By applying such colloquial discipline to the four unifying principles, you will have successfully unified the vacuous part of mathematics. With a much-appreciated quirk:

As you might expect, the unifying principles sometimes conflict. The prime example is the widely celebrated positional notation for numbers. Consider, as a typical example, the number 90038. On the one hand, minimal summary discipline would require this to become $9*ten^4 + 3*ten + 8$. On the other hand, the positions of zero provide a contextual narrowing for *Nothing*; and *that narrowing is not needless*.

Quite the contrary: positional context is necessary for efficient computation, as math students get told over and over again. It engendered, they are told, a flowering of mathematics. That is true; it did. What is *not* true is what they also get told over and over: the invention of ***number*** zero enabled positional notation. It did not.

It was the invention of a ***symbol*** for *Nothing* that actually enabled positional notation. That symbol's subsequent conversion into a *number* was a naive perversion. To see this, express 90038 as 9□□38. Is □ a number therein? No. Does it need to be a number? No, *it merely needs to engage with numbers*. What about 9□□38 itself—is that a number? Yes, or rather it *represents* one just as well as 90038 does. Better actually: the positions of trans-numeric *nada* exhibit *what is being ignored*, namely ten^3 and ten^2. That same service is provided by "number" zero, but without that same clarity about dimension.

Here you see a second unusual case in which *exactly-what-had-vanished* ***is*** relevant: A *numeral* had vanished in a certain position. Altho that context does conflict with summary discipline, it is needed to enable radix summary to be ***tacitly*** *expressed in a readily computable form*. This is such an advantage that it trumps summary discipline—it is a tradeoff of good design.[9]

Having resolved this conflict, we are now ready to explicitly engage no-number *nada* with numbers, thereby…

Unifying vacuous computation

This can be achieved by starting over and designing a new programming language, *UniNada* shall we say, disciplined by our new unifying principles. For its comparison with typical languages, let us use the recently ascendant app-programming ones, Objective-C and JavaScript.

Both of those languages have 0 augmented by tuple vacuities in the form of arrays [0], [0, 0], [0, 0, 0], … which may themselves be part of other vacuous arrays. Objective-C has, in addition, the following vacuities: false, NO, nil, Nil, NULL, null. JavaScript has these: false, null, undefined, "".[10] Most of these vacuities are euphemisms for 0—euphonious either to the programmer or the code generator. The collection of all of them represents what this paper calls *nada*, to which we shall try to condense them. The first step is to implement the most important unifying principle: *Use explicit addition to package algebraic summary*.

It is remarkable that no general-purpose programming language has done that for disparate items yet. To emphasize such atypical inhomogeneity, let us call such a package a "*mixed sum*", or *mixSum* for short. To see how easy a mixSum would be to implement, start by constructing a typical one, M say, whose underline underscores its diverse Multiplicity.

(Travel advisory: the short section that follows confronts hackneyed university mathematics with *exceedingly* elementary arithmetic. It is not intended to teach arithmetic; you already know that I'm guessing. Rather, it is intended to depose sterile concepts held at the highest level of mathematics. If the following arithmetic seems too obvious or tedious, skip to where it begins doing its deposition duties, introduced by… "Typically, mixSums will be used…". Or just skip to *Amalgamating algorithmic vacuities*.)

We begin in the tradition of sacred literature and great epics, with *nada*, like so: $\underline{M}$ = *nada*. Then we add an apple: $\underline{M}$ + apple = *nada* + apple = apple. This prompts our most fundamental rule of mixSummary: For any generic mixSum $\underline{X}$,

$$\underline{X} + \square = \underline{X}$$

So *nada* gets *ignored* under summary, something small children understand. *Ignorance*, they might surmise, is the essence of *nada*. Even in those special binary operations in which *nada* annihilates, it turns out to be ignoring the other argument: *nada* either ignores or gets ignored.

Now add an orange: $\underline{M}$ + orange = apple + orange. To that package you can add a banana giving $\underline{M}$ = apple + orange + banana. All of this may seem obvious, but there is an implementation detail that should be emphasized: mixSums are not bound by *enclosure*, as done in arrays, key-value pairs, sets, and virtually all other programming collections; rather they are bound by *explicit addition*, as just displayed.

Such addition does not function like the *passive separating punctuation* (commas, semicolons, spaces, …) found in traditional collections. Instead it functions as *active uniting punctuation* obeying certain coalescing rules. The fundamental rule just displayed, for example, coalesces by ignoring *nada* whenever it gets added to anything. Such rules, when properly disciplined to minimize, may actually make a mixSum *shrink* when it is *added to*; something a passively separated collection would never do.

The evolution of our package so far suggests two more rules, namely indifference to order and grouping. Together, they ensure that *the* sum is *unique* regardless of how it had been achieved. These rules are familiar for homogeneous sums, but need to be rechecked for our unfamiliar *mixSums*:

Clearly we could have combined an orange with an apple rather than an apple with an orange; we would have arrived at the same package. Similarly, we could have combined a package of an apple and orange with a banana, rather than an apple with a package of an orange and a banana. So generic mixSums $\underline{X}, \underline{Y}, \underline{Z}$ do indeed obey the following rules:

$$\underline{X} + \underline{Y} = \underline{Y} + \underline{X}$$

$$(\underline{X} + \underline{Y}) + \underline{Z} = \underline{X} + (\underline{Y} + \underline{Z})$$

These rules are taught in elementary school, but the example that just engendered them blatantly violates the most fundamental rule taught there: *never add apples and oranges*. Why not?—we just checked that it obeys the rules. Here is why not: the apples-and-oranges rule is a naive version of a professor's scalars-and-vectors rule. The children who become professors dignify the naive rule, as required by their profession; and then present it in academic vector attire to aspiring elementary teachers, who remove its black gown and pass it along to their trusting young students, who become professors …

Now add an orange and take away a banana, giving $\underline{M}$ = apple + 2*orange. Yes, we have no bananas;[11] but we do have something not yet seen, namely multiple-addition. The banana loss is handled by this generic rule:

$$\underline{X} + -\underline{X} = \square$$

When you subtract a mixSum from itself you are left with *nada*, of course. The slight syntactic intricacy here arises from subtraction being expressed as addition of a negative, necessary because addition, not subtraction, is the *packager* in a mixSum. Now for the multiple-addition:

When an element is added to a mixSum that already contains it, that element becomes a multiple-element, a 2*orange in this case.[12] When the multiplicity of such addition is *n* you get *n**orange, in which the "*" symbol denotes *multiple-addition*. To clearly communicate its meaning, let us call the addition that produced it *(n)plicity-addition*.

But what about *partial*-addition, *half*-addition say? Well a (*two*)plicity-addition of a (*half*)plicity-addition of an orange would produce a full orange, but in two equal pieces. So a (*half*)plicity-addition alone produces half an orange, symbolized as (1/2)*orange. A (1/*n*) plicity-addition would produce only (1/*n*)th of an orange. In the limit, a (*nada*)plicity-addition would of course produce *nada*, symbolized as…

$$\square * \underline{X} = \square$$

This rule displays the prototype binary operation for which *nada* ignores the other argument, thereby annihilating it. It completes the *nada*-rules for a mixSum, but there are further rules for multiple-addition. That operation has, unsought, just generated a new binary operation, scalar multiplication, whose rules for scalars *r* and *s* you have seen since you were a child. What you may not have seen is how they interact with generic multinumbers $\underline{X}$ and $\underline{Y}$.

$$1 * \underline{X} = \underline{X}$$

$$\underline{X} * /\underline{X} = 1$$

$$s * \underline{X} = \underline{X} * s$$

$$r * (s * \underline{X}) = (r * s) * \underline{X}$$

$$s * (\underline{X} + \underline{Y}) = s * \underline{X} + s * \underline{Y}$$

The slight syntactic intricacy here is that the *reciprocal* of $\underline{X}$, $/\underline{X}$, when multiplied with $\underline{X}$ produces *one* in the same way that the *negative* of $\underline{X}$, $-\underline{X}$, when *added* with $\underline{X}$ produces *nada*. A reciprocal is possessed by every scalar, vector, bivector, and every other geometric element; but not by *nada*, a bivector plus a nonparallel vector, an apple, and most other mixSums. So this rule applies only to those mixSums that do have a reciprocal.

A *scalar* is exemplary because *it **always** has a reciprocal* that multiplies it to ***another scalar***, the unit one. There is no need for the constant nag to not divide by *number* zero. *Zero is not a number*, and its ignoring of the other argument precludes it from ever multiplying to any kind of number, let alone the unit scalar. A bivector plus a nonparallel vector *is* a number, a *geometric* multinumber, but its dimensional properties preclude it

from ever multiplying to a scalar, let alone the unit one. An apple cannot multiply to a scalar either, so it too lacks a reciprocal.

Now that the rules for a mixSummary have been assembled, the next unifying principle requires them to generate a minimal form. That is the responsibility of the addition operator, which must wield the rules so they always merge elements as much as possible. Which may sometimes merge them to *nada*.

So is *nada* itself a mixSum? No!—*nada* transcends even transcendent mixSums. That is the whole point of this paper. *Nada* is the *universal Nothing*, useful whenever *anything* has vanished: numbers, sets, strings, arrays, structs, objects … whatever. After they vanish, no properties remain by which they could be distinguished as *Anything*. Such an utterly vanished *nada* shall be needed by many operations other than those for mixSums, which merely provide familiar examples of its use.

To proceed to a few such examples, some knowledge about possible implementation of a mixSum might be useful. Each element must convey two pieces of information: its *kind* and its *multiplicity*. This can be done with classical *key-value* pairs, for which there are now elegant and efficient implementations. To derive mixSums from such pairs, all that needs to be done further is to express them backward in *value-key* form, like *2*orange* for example, and then discipline them with minimal addition. That will bind value*keys into mixSum packages of arbitrary length, and dispense with the need for traditional enclosing punctuation.

Typically, mixSums will be used in various specialized domains. The domain developed so far has been a whimsical mixed-fruit domain, but you may more often want a vector-space domain, or a complex-number domain, a geometric-algebra domain, and so on. If so, it is useful to provide mixSums with an explicit *basis*, namely a list of those kinds of elements that can occur in that domain, such as {i, j, k}, or perhaps {apple, orange, banana}. Such a basis is not required, but when available it can enable error checking and even better efficiency. Basis-specialized mixSums may be deployed with either *restrictions* to their rules, or *enhancements*.

The most familiar example of a *restricted* mixSum domain is the vector space on the basis {i, j, k}. Its purpose is to generate any possible scaled sum of those vectors. The built-in rules of a mixSum do that automatically by producing, for example, 3*i + 4*j + 5*k. However, from a traditional perspective the rules do too much because they allow scalars to add with vectors. Scalars became empowered to do so after they had unexpectedly arisen from multiple-addition. The mixed nature of that addition allows them, *as elements of the domain*, to participate in it. So *scalars are an intrinsic part of any basis* in a mixSum.

This means that 1 is implicitly present in a mixSum basis. Hence, to get a traditional vector space, you not only have to provide mixSums with a vector basis, you also have to explicitly exclude 1 from that basis, which precludes scalars from adding to vectors.

Such explicit restriction is a stark contrast to the *implicit dimensional segregation* traditionally used to prevent scalars from adding to vectors. Under that subterfuge, scalars (0-D elements) and vectors (1-D elements) have separate sets of rules, and can only interact in a limited way. In essence, scalars must sit at the back of the bus, like so:

There is an addition of scalars and a separate addition of vectors, *each obeying duplicate sets of rules*, including all of the rules about zero. For such tacit dimensional segregation to work, there must be a "zero scalar" and a duplicate "zero vector". There is a multiplication of scalars by a scalar, and a separate multiplication of vectors by a scalar, *also obeying duplicate sets of rules*. The only mixSum rules not duplicated are those that would allow a scalar to add to a vector, which is how that is prevented.

Now suppose you want to increase the expressive power of your vector space by allowing scalars to add to vectors. This would be easy if you had implemented it as a restricted mixSum: just remove the restriction. If you had instead implemented it by implicit dimensional segregation, as usual, then you would have to duplicate the mixSum rules that allow a scalar to add to a vector. But then you would have *completely* duplicate sets of addition and multiple-addition rules for scalars and vectors, whose duplicity would therefor make no relevant distinction.

Most important to this paper, you would have duplicate zeros, whose duplicity would now be ignored. That duplicity could, in fact, have been ignored all along if those zeros had been *un-dimensioned*, as *nada* actually is. Un-dimensioned zeros would have caused their duplicate sets of rules to coalesce to just one set. This would work fine even under a traditional segregated vector space: exactly what had vanished within it is even less relevant than what had vanished among arithmetic and set algebra. That *exactly-what-had-vanished* is currently deemed relevant in a vector space is another needlessly narrowing historical accident. It arose from mis-focus on the peculiarities of symbol maps rather than on the territory they represent.

In summary, a traditional vector space is just a *restricted* mixSum on a vector basis. Conversely, a mixSum on a vector basis is an *unrestricted* vector space in which scalars may expressively add with vectors. Now for examples of *enhanced* mixSums:

Perhaps the most familiar example is complex numbers on the basis {i}, dubbed the *imaginary number* by Descartes because i^2 mysteriously equals –1. That product is the *enhancement* to the mixSum rules. That enhancement aside, this kind of mixSum is the simplest possible *unrestricted* vector space—it has just one explicit element in its basis, namely i, and it is not restricted from scaled addition with the implicit element 1.

Such uncommon magnanimity about scalars is not due to any uncommon enlightenment about disparate addition. Quite the contrary in fact: since i is able to add with scalars, i itself is commonly thought to be some kind of scalar! Like zero is commonly thought. In fact "*scalar*" i is the same kind of naive perversion that "*number*" zero is.

The idea of an imaginary "scalar" fails to seem bizarre to modern mathematicians because it confers three benefits. First, it avoids the traditional requirement for segregated rules, one set for reals, another set for imaginaries. The summary rules for complex numbers are just the rules for a mixSum, unduplicated, which are nothing more than the rules of arithmetic. Second, if i is a "scalar" then there can be no separate imaginary zero—there can be just one "scalar" zero. Third, having two similar "scalars" (rather than two dissimilar coordinate vectors) has caused them to now be packaged by explicit addition, rather than Hamilton's couples. These genuine benefits would have been better achieved by enlightenment about mixed addition, rather than ignorance about dimension.

Ignorance about dimension is the reason a "scalar" i is truly bizarre. When dimension becomes well defined, a genuine scalar like 7 acquires dimension {0}. When that happens the Hamiltonian *planar* imaginary i acquires dimension {2}, which removes its mystique: Under multiplication it rotates a vector by a quarter-turn—*an orthogonal turn*—so its square is a half-turn, namely multiplication by –1.[13] There is also a *volume* imaginary having dimension {3} which performs a spatially extended kind of ortho-turn, whose square is also a half-turn, –1.

In fact there are endlessly many ever-more-intricate *hyper-volume* ortho-turners. Half of them backtrack, meaning that they undo themselves and square to 1. Half of them proceed onward and square to –1. Can such high-dimensioned "imaginaries" be generated by grammatical fiat, the way our so-called scalar "imaginary" had been generated in the mid 1500s?

Let's try: enhance a *deliberately unrestricted* {i, j, k} vector space in the same way the *de-facto unrestricted* {i} "complex" vector space had been enhanced: cavalierly set $i^2 = j^2 = k^2$ to –1. That's what Hamilton did in 1843 (*with no clue what it might mean!*). This *really* enhances the mixSums on that basis because they now generate the entirety of traditional vector algebra including the dot product, the cross product, the triple product, and so on, as Hamilton slowly discovered to his continued astonishment and delight. That algebra[14] belatedly acquired just one 0 after Hamilton's Reversion to explicit addition; so things within it can *really vanish*.

However, it hasn't moved us to higher-dimensioned imaginaries. The {i, j, k} basis elements are still *Hamiltonian Planar Imaginaries* (shall we say), namely bivectors that he misinterpreted as vectors. He was able to get away with that because his imaginaries had moved up off his *couple*-plane into volume-space where they became the orthogonal complements of vectors. So our bold grammatical fiat has merely moved *Hamilton's Unwitting Bivectors* up off a plane into the next-higher space where there is room for three independent ones.

To actually generate a higher-dimensioned imaginary, enhance unrestricted mixSums on the {i, j, k} basis with Clifford's *geometric product*,[15] rather than with cavalier negative squares. That generates the free Geometric Algebra of physical space in which ijk is the aforementioned *volume* "imaginary", dimension {3}. The basis elements {i, j, k} are now genuine vectors, dimension {1}. Their pairwise products, ij, jk, ki constitute three *witting* bivectors, dimension {2}. These bivectors square to –1, as does trivector ijk. By contrast, the *witting* i, j, k vectors here all square to 1. Those vectors are the orthogonal complements of bivectors jk, ki, and ij. Hence, the even sub-algebra here (scalar plus 3 bivectors) is Hamilton's quaternion algebra; and the even sub-algebra on the {i, j} basis (scalar plus 1 bivector) is his couple algebra, the usual "complex" algebra. The full algebra on {i, j, k}, like other mixSummaries, lets things—***now well-dimensioned things***—*really vanish*.

Readers wishing to see how these disparate, well-dimensioned numbers arise and interact might spend many delightful years studying Geometric Algebra if my experience is any indication. However, this paper only tidies up their vanishing, finally amalgamated to *nada* by the rules for mixSums. To finish tidying we need to get on with…

Amalgamating algorithmic vacuities

There is really only one computational vacuity at the hardware level, namely bit-pattern 0, as low-level programmers understood from the beginning. That vacuity is most meaningfully expressed as trans-numeric *nada* because it is used in both numeric and non-numeric contexts. Such diverse contexts have caused it to acquire various different high-level names needlessly. The history of Objective-C provides a cautionary example.

At the hardware level, bit-patterns are used for both logical and addressing operations. The bit-pattern for 0 is exceptional in three ways: 1) It is the pattern arising from self-subtraction. 2) When it arises, the zero flag is set. 3) It is hard-coded into the silicon so that registers can be cleared. These facts combine to make hardware 0 ideal for representing trans-numeric *nada*.

The first fact makes 0 ideal for representing *numeric nada* because it generates all of the *nada*-rules for a mixSum, like so: self-subtraction vanishes; addition with 0 gets ignored; multiplication with 0 vanishes.

As for *non-numeric nada*, the zero flag distinguishes the 0 bit-pattern from all others; so 0 can make the distinction between profuse *Something* and meager *Nothing*. Let us call that the *some–none distinction* to set it apart from the ***much narrower*** *true–false distinction*, scrutinized shortly.

The third fact, 0-cleared registers, reinforces the *some–none* distinction in two ways: First, a 0-cleared data register allows memory to be initialized to a known empty state, meaning that it contains *Nothing*. Second, a 0-cleared address register starts the operating system at a standard entry. That address is not valid for applications, so it can serve as their canonical representative for a non-address, meaning that it addresses *Nothing*. The zero flag makes detecting it trivial.

All of this was well known to the designers of C, who created a language with a genuinely trans-numeric *nada*, represented by the 0 bit-pattern. It is fair to say that they, at least, had recovered for 0 the singular *Nothing*ness it had lost a thousand-some years ago. In particular, branching was based on the broadly informative *some–none* distinction.

Unfortunately, this non-numeric use of "***number***" zero to make decisions seemed so strange to high-level programmers that they devised their own constants for branching: true = 1 and false = 0. That subterfuge preserved comfortable prose distinctions.

The use of ad hoc true and false constants to make branching decisions became so prevalent that it was eventually sanctioned in the C99 standard. The now-sanctioned presence of those constants often causes naive C programmers to coerce all decisions into a *true-or-false* form, which gets complied busily as a 1-or-0 distinction. That is then actually computed as a *some–none* distinction via the zero flag, for which 1, *and only* 1, represents *Something*. This is clearly inefficient, especially in inner loops; so compliers have evolved that can sometimes reduce prolix *true–false* distinctions to concise *some-none* form. Other artifices can sometimes undo other problems caused by reflexive rejection of non-numeric 0.

Rejection of non-numeric 0 eventually spread to pointers for which the macro constant NULL was defined. Like false, it is also set to 0. That 0 indicates, as stated, that this pointer does not point to anything.

Objective-C arose before these various vacuous constants had officially insinuated their way into C; and its creators defined their own peculiar vacuities, also set to 0. NO is Objective-C's version of false, and nil is its somewhat more specialized version of NULL. The specialization is that nil indicates no *object* is being pointed at. An object requires a Class to generate it, which is *also* an object in Objective-C; so Nil was defined to indicate that no Class-object is being pointed at. And of course with objects available to point at there arose the need for a vacuous one, for which null was defined. It is, in essence, an object wrapper around 0—a nested empty box.

With the exception of that wrapper, an Objective-C programmer can substitute 0 for every one of the aforementioned vacuities—they are all mere euphemisms for it. Doing so is frowned on (or even balked at by paternalistic compliers, even tho compilation would be faster) because **zero *is a number darnnit***, not to be used in non-numeric contexts. Actually **zero**, as I hope to have persuaded you, ***transcends numbers*** and should be used in any context in which things can vanish. For that reason, the best possible euphemism for it might be *nada* because that word encompasses all the others: zero, false, NO, nil, Nil, NULL, null. To emphasize this, let us henceforth use □-*nada* rather than 0-*zero*.

Typically, the operational context in which *nada* finds itself will automatically narrow it. This is true for all of the just-listed euphemisms except null, meaning a vacuous object. It requires extra-operational distinction from nil and Nil, meaning no pointer to an object. This is similar to the distinction needed between the dimension of □ and the dimension of a scalar; and can be made in a similar way: set nil = Nil = □ and null = {□}.[16] The lack of braces for nil and Nil indicates that they point to no object (they have *no context* for an object); the braces for null indicate that this object contains *no thing*. This tactic preserves a universal □-as-*nada*, and allows its context, whether operational or explicit or both, to make all narrowing distinctions.

A similar analysis applies to JavaScript with this exception: the empty string "" is *falsy*,[17] meaning that it is considered vacuous under a *some–none* distinction. It is another of those superfluous vacuities that could easily be set to □. That would require defining how trans-numeric □ acts under string concatenation: it gets *ignored* left and right of course. Such unification can't be done in JavaScript because the plus sign is used not only for number addition, but also for string concatenation—the stringy-ness of "" is required to determine which operation to perform.

This illustrates that the design of *UniNada* should give the plus sign only the semantics defined by the addition rules for a mixSum. Concatenation has different semantics (it doesn't commute, for starters) requiring a different symbol, which would automatically set up an operational context for its use of *nada*.

Careful distinction by *context for Nothing*, rather than *kind of Nothing*, is essential for any programming language hoping to reclaim for 0 the transcendent singular *Nothing*ness it had lost so long ago. The early C language can serve as an embryonic exemplar.

Never forget Sturgeon's Law: ***Nothing is always absolutely so***.[18]

[1] An internet search for "history of zero" provides a rich source of information. The sources I found most informative include Ahmed Boucenna, *Origin of the Numerals, Zero Concept*; and Robert Kaplan, *The Nothing that Is, A Natural History of Zero*. The well-written Wikipedia article *0 (Number)* provides other sources.

[2] **Crowe**, Michael J. *A History of Vector Analysis* p25. This work is magisterial, and is my favorite reference on the evolution of vector ideas.

[3] Why didn't he call them *quadruplets* in conformity with his *triplets* and *polyplets*? I now believe, after studying his *quaternion* expositions, that he must have felt *quadruplets* lacked the gravitas befitting such a momentous concept.

[4] For example, one of our distant grandfathers encountering a saber-tooth tiger did not think "Hmm… a looming image containing long white descenders—what does that represent?"…chomp. Chew. Burp. Rather he thought "Tyger!"…dodge, and maybe escape. Readers wanting to pursue this may read Gregory Bateson, who harped about how evolution has predisposed us to conflate map with territory; and how important not doing so has become.

[5] **Crowe** p31-32. Hamilton did add a scalar and a vector, starting from the day of his invention of quaternions, but it perplexed him and his colleagues for many years.

[6] This is usually translated as "*Linear Extension Theory*". Unfortunately, that title misleads contemporary mathematicians, who now understand *linear* to mean distributive across addition and scaling. That was not Grassmann's intent, even tho many of his operations are indeed *linear* in that sense. Instead his *Lineale* referred to *lines*; and was intended to distinguish this book from a planned *Angular Extension Theory* that he never wrote. Lloyd Kannenberg provided this clarification.

[7] The Geometric Algebra community uses a single kind of *Nothing*. However, the other kinds are still sanctioned, but just ignored. For example… "There is good reason to regard the zero vector as one and the same *number* as the zero scalar." or… "The element 0 *must be allowed to have any appropriate grade* [dimension]." (My emphasis.)

By merely ignoring vacuous distinctions, rather than rejecting them outright, that community fails to arrive at the trans-numeric aspect of *Nothing*. To emphasize that aspect, let me repeat: *zero is not a number—it must not be allowed to have any grade whatsoever.*

[8] Did you think it was points, not scalars, that are 0-dimensional? That was Euclid's idea, but it doesn't work with an operation that formally generates dimension. Points have locus, the most primordial kind, so *their dimension is the primitive one*, namely {**1**}, from which other dimensions arise. Grassmann was first to understand this, in the 1840s. If you would like to acquire his remarkable insights, the *Geometric Algebra* page at gary-harper.com/ can get you started.

[9] Positional context of tuples enables a weaker kind of computational efficacy, but at the great cost just discussed. For tuples, the considerable benefits of summary discipline *strongly* trump the slight benefit of positional context. That context can, anyway, still be maintained by well-ordered notation and careful implementation. Again, it is a question of good design.

[10] NaN is also considered a vacuity in JavaScript for branching purposes, but it is not a vacuity in the sense used in this paper—it denotes multiple *Somethings*, numerically invalid ones.

[11] This apparent non sequitur contrasts languages: The "yes" would occur in Spanish to affirm the antecedent "You have no bananas?" That "yes" would be a "no" in English to deny the subsequent presence of bananas, an unusual construct among languages. Such linguistic inconsistency about *Nothing* underscores its truly perplexing nature.

[12] English requires "s" for plural. However, that is an pointless embellishment for a generic mixSum; and often a pointless embellishment even for English, as authors, I mean *an author*, like me know, I mean *knows*, all too well.

[13] This rotation is only indirectly related to the well-known rotation in the usual Argand diagram of complex numbers. That diagram is a peculiar *model* of them in which a "vector" actually has even dimension {0, 2}. By contrast, multiplication by a bivector, dimension {2}, rotates a genuine geometric vector, dimension {1}. Moreover, it does so independently of basis and position: the vector need not be anchored to the origin as it does in an Argand diagram, nor is the "imaginary" bivector compressed onto the vertical axis.

[14] That algebra has a rich literature. The most accessible expositions, as Gibbs and Heaviside both testified, are those written by Peter Tait, Hamilton's quaternion evangelizer. Hamilton's own expositions are obscure and academic—even his colleagues were baffled. Lord Kelvin and Sir George Airy, in fact, were appalled.

[15] William Kingdon Clifford unified Grassmann's inner and outer products into the *geometric product* in which Grassmann's products are its even and odd parts. His seminal paper on this is *On the classification of geometric algebras*, found in his *Mathematical Papers*, p397. Also relevant are *On the hypotheses which lie at the bases of geometry*, p55; *Preliminary sketch of biquaternions*, p181; *Applications of Grassmann's extensive algebra*, p266. These papers were all written in the 1870s. Clifford-the-geometer was a rare enthusiast of Grassmann, unlike Hilbert-the-mathematician and most others, who considered him bafflingly innovative but woefully non-rigorous.

[16] Operational context would automatically distinguish null = {□} from dim(7) = {□}.

[17] *Falsy* hasn't made its way into dictionaries yet, except in old ones where it refers to padding enhancing a woman's figure. Its recent coinage refers to those many constants in JavaScript that mean *Nothing*, which drives home just how inexorable zero's loss of *Nothing* has become.

[18] Theodore Sturgeon, in *The Claustrophile*, 1956. Sturgeon's more-well-known *Revelation* is that "*Ninety percent of everything is crap*", deployed to defend science fiction against that claim about it specifically. Not crap is Sturgeon's classic science-fiction novella, *More than Human*, a prescient anticipation of the transcendent power of internet-enhanced *networks of disparate humans* acting in concert.