Connectors and Variables

Linas Vepstas

21 August 2020*

Abstract

Almost all classical work on logic and symbolic reasoning, and indeed, much of mathematics, including lambda calculus and term algebras, are built on the intuitive foundation of "variables" and "functions" that map between them. Modern work on linguistics and category theory indicates an alternative viewpoint: that of "connectors". In this view, the act of replacing a variable by a value, and "plugging it into a function" (beta-reduction) is a special case of connecting a pair of connectors.

This is a short note clarifying the relationship between these two concepts. Noteworthy is that the concept of connectors is "more general", and that, as a mathematical framework, it is far less explored.

As a footnote, it is noted that the use of connectors (and the implied sheaf theory) appears to have impact on a number of philosophical arguments, ranging from mereology to postmodernism(!) This is a surprise. (The author acknowledges that it sounds perhaps cranky to place such claims in the abstract, but the notions are heart-felt.)

Introduction

In elementary mathematics, it is commonplace to speak of a variable, such as x, and functions of that variable, such as $f(x)$. The power of this abstraction is that variables can be assigned a value, and that value can be "plugged into" a function. For example, one asserts that $x = 42$ and then ponders $f(42)$. This act of "plugging in" is formally called "beta reduction".

This process of "plugging in" can be compared to the process of "connecting", for example, connecting together two jigsaw-puzzle pieces. When mating together connectors, one typically has a matched pair that can be mated: a plug that can be plugged into a socket. It should be clear that beta reduction is a special case of this: that $f(x)$ is a "socket" into which values can be plugged into.

This observation seems nearly trivial; it is so painfully obvious that one wonders what more can be said. Yet there are tremendous consequences that arise from this, consequences that are in some sense equally trivial and obvious. That something is

*Revised version; Earlier version: 2 May 2020

"obvious" is perhaps an indicator that it is important. This note dwells on these "obvious" ideas, and demonstrates their importance. The problem with "obviousness" is that it frames all manner of thinking: it is Aristotle's "formal cause", applied to thinking. It is provides a normative framework in which all conversation takes place. Design and thinking happen in the foreground; there is a "figure" and "ground", and anything that is obvious falls into the background, becomes implicit, and serves as a barrier to further thought and examination. Thus, the impetus for this note.

The starting point of this note is that, when one thinks of variables and functions, one is inevitably lead to the notion of "directed acyclic graphs" (DAGs). That is, plugging variables into functions, and those into other functions invariably leads to a DAG, because the sense of direction of "plugging in" must be preserved: one cannot plug a variable into a value; one cannot plug a function into a value; the plugging is always directed. One is a receptacle, the other is not. By contrast, that act of connecting together connectors has no such constraint: a jigsaw puzzle is not a DAG; there is no jigsaw piece "at the top", and none "at the bottom". Although the connectors in a jigsaw puzzle have a polarity, the sense of direction that they offer is immaterial to the final construction.

The desired conclusion is that by approaching knowledge representation as an assemblage of connections, rather than as a unidirectional network of inferences, one can gain considerable power in working with the common problems and stumbling blocks of AI and AGI, including problems of planning, inference, constraint satisfaction, combinatorial explosion and the "frame problem".

This note begins with a long review of everywhere that the notion of variables and functions appear, or, more generally, of arrows and directed graphs. This may seem pointless, but serves only to underscore their ubiquity. This is followed by a section that develops the proposed alternative. Along the way, assorted commentary is made with regards to outstanding philosophical problems, as befits any exploration that belabors the "obvious".

Arrows

Arrows, and usually accompanying DAGs are everywhere. They really are pervasive in descriptions of mathematics, in symbolic AI and knowledge representation, and even underpin vast tracts of philosophy. This is, again, so potently "obvious" that an explicit review is called for.

In philosophy:

Here's a hint of how pervasively important the concept of variables and values can be. Some philosophers use these notions to anchor the idea of "objects" or "things" that "exist":

"An object is anything that can be the value of a variable, that is, anything we can talk about using pronouns, that is, anything." (Van Inwagen 2002, 180)

This conception of an object then leads to confused discussions about the identity of indiscernibles, and the importance of location (space-time) in mereology. The notion

of connectors can be used as an alternate foundation for the conception of "objects", "identity" and "location". Specifically, one can instead define objects as "things that can participate in relationships".

In term algebra:

The theory of term algebras is a specific branch of mathematics that deals with the abstract process of performing generic algebraic manipulations, as any mathematician or engineer might do when working with pencil and paper. It is important for the theory of computation, because much what we do with computers is to manipulate symbols; term algebras provide a coherent vocabulary of ideas for thinking about algorithms that manipulate symbols. This is, in turn, important for both logic, and for symbolic AI, since reasoning engines and algorithmic theorem provers are built on such symbolic manipulations.

Consider a typical definition of a term algebra. It consists of:

- A set of constants $c_0, c_1, \cdots$
- A set of variables $x, y, z, \cdots$
- A set of n-ary function symbols $f(x_1, \cdots, x_n)$

The terms of a term algebra are then anything that can be constructed from the above, by recursively "plugging in". That the resulting terms always have the structure of a DAG should be intuitively obvious.

In type theory, it is common to assign types to the different constants, variables and function symbols; this does not alter the result that the terms are DAGs.

In set theory:

In (well-founded) (naive) set theory, there is effectively just one function symbol: the set, and only one constant: the empty set. There are no variables, in the sense that proper sets do not contain variables in them. Of course, in the definition of the axioms (and axiom schemas!) that define set theory, and in the articulation of the theory itself, it is impossible to avoid variables; its just that variables are not set elements. By definition, "well-foundedness" disallows infinitely recursive sets. Thus, effectively, all finite sets in set theory have the shape of a DAG (they form a partial order).

In computer science:

Lambda calculus can be thought of as having arbitrary constants and variables, but only one function symbol: the lambda. It is effectively a theory of linear strings (sequences) of symbols arranged in order, on a line. Insofar as lambda expressions can be beta-reduced, they form a DAG. Lambda calculus is manifestly finite: it is very unusual to study countable or uncountable limits of lambda calculus. In general, lambda calculus is always assumed to be well-founded, so that the expressions are always DAGs. An exception to this is chem-lambda (Buliga 2003), which redefines lambda as having input and output connectors, thus allowing looping connections to be made.

The general typed lambda calculus famously corresponds to computer programs, this is the Curry-Howard correspondence. Infinite loops can be thought of in two ways in computing. One way is as a directed graph containing a loop: the arrows denote a function that recursively calls back to itself. The state transitions are described by points (for states) and arrows (denoting state transitions), arranged such that one (or more!) collections of arrows can be traced in a circular loop. The other way of thinking of infinite loops is as recursive structures: one unrolls the loop so that it becomes an infinitely-long unterminating sequence of arrows. Roughly speaking, infinite loops correspond to ill-founded sets. They are the "gunk" of philosophy.

The moral of the story that is being presented here is that almost all of computer science is founded on a theory of DAGs. Even when loops are allowed, the graphs are formed from edges that are directed: the edges are always arrows. This is not the same as a general theory of connectors connecting things together.

In topology:

The infinite binary tree provides a simple example of unrolling a pair of interconnected loops. Axiomatically, a binary tree is constructed from a pair of left-right elements, and each element can be another pair, or the termination symbol. In this sense, both the left and right elements can loop back onto themselves; when the loops are unrolled, the tree becomes infinite DAG (rather than a small graph containing a pair of loops). The endpoints of a binary tree form the Cantor set; there are an uncountable infinity of them. These are the "points" of point-set topology.

One can also have a point-free topology, built on top of lattice theory. Rather than considering points as mereological simples, one instead focuses on the directedness of set inclusion. Starting with the notion of a partially-ordered set, the axioms of lattice theory allow one to concentrate on meets and joins, filters and ideals, frames and locales, without once bringing up the notion of a point. Yet, in the end, partial orders are founded on the idea of direction, of inclusion, with set theoretic notions of the same providing much of the intuitive grounding. There is a DAG, even if discussion of terminal elements is avoided.

In category theory:

The simply typed lambda calculus is important, as it is the "internal language" of "Cartesian closed categories". The Cartesian-closed categories naturally describe "tuples" of "things": that is, things that can be placed in an ordered list (a "Cartesian product"). The "things" are presumed to be discernible items[1] that have an identity. Indiscernibles are not described by Cartesian closed categories; for this, needs symmetric monoidal categories. Tensors and tensor algebras, vector spaces and Hilbert spaces famously belong to this category; the internal language is "linear logic". Yet, the tensor category is dagger-compact: it has a left-right symmetry to it that constrains structure.

A pending critique of neural nets is that they are founded on vector spaces (and thus are part of the tensor category, which is symmetric) whereas natural language is

[1] One could say "objects" instead of "items", as that is the appropriate term for category theory. It is, however, useful in the present case to use the slightly more vague term "items".

described by monoidal categories that are manifestly not symmetric. This observation has been pursued elsewhere, in various writings by this author.

Category theory itself is a theory of dots and arrows. Every dot has an arrow back to itself (thus, a loop), and arrows can be composed: if there is a sequence of two arrows connecting three dots, then there always is an arrow, going in the same sense, between the two endpoints. This is all that a category is. Deep theorems follow, such as the existence of all limits, given only spans and equalizers! Category theory is surprising in it's power.

The moral of the story that is being presented here is that almost all of computer science is founded on a theory of DAGs. Even when loops are allowed, the graphs are formed from edges that are directed: the edges are always arrows. This is not the same as a general theory of connectors connecting things together.

In model theory:

Model theory takes the idea of a term algebra, and supplements it with relations. Relations are n-ary predicates $P(t_1, \cdots, t_n)$ of terms t_k from the term algebra. A predicate is either true, or it is false: it either holds or does not. The most famous predicate is that of equality; the "theory of pure equality" is a term algebra supplemented with a predicate defining equality.

Predicates are useless unless they can be combined using the operators of logic: "and", "or", "not", "there exists", "for all". Curiously enough, the rules for combining these operators again form a DAG: the sigma-pi hierarchy. Combining terms with predicates seems nearly enough to serve as a foundation for all mathematics. The sigma-pi hierarchy, taken on set theory, is extremely powerful. Just the first few levels: first-order logic and second-order logic are enough to meet the concerns of most classical mathematics. Even things that are "larger" than set theory can be represented in this hierarchy: this is the content of the Yoneda lemma.

The moral of the story is as before: when viewed from the meta-perspective, DAGs appear to be sufficient to describe all of mathematics. It is hardly a surprise that they are part of the ground, the background, and not a part of the figure. They are a part of the under-pinnings of the meta-mathematics, and are almost never overtly discussed.

In relational algebras:

Removing the term algebra but keeping the relations results in a "relational algebra", with examples of relations being "is-a", "has-a", "part-of". Limiting oneself to a single part-whole relationship, one arrives at mereology as a possible alternative to set theory as a foundation for mathematics. But one also arrives the the notion of "gunk" in philosophy (that, roughly speaking, infinite recursion is possible, when considering the structure of physical reality), as well as a number of puzzles dating to Ancient Greece (the Ship of Theseus: what happens when parts are replaced? The Statue and the Lump of Clay: what happens when parts are rearranged?). That there are such puzzles is perhaps endemic to the desire to apply part-whole relationships, and the adduced DAG partial orders as a fundamental ground on which to build philosophical discussions.

In knowledge representation:

A practical application of relational algebras can be found in the field of knowledge representation. Practical systems embodying relational algebras include key-value databases and SQL databases. The "key" is a named slot, the "value" is the item that is attached to the key (that occupies the "slot" named by the key). Hierarchical relations can be built by placing keys into slots: thus, for example, the Unix file-system is effectively a key-value database. On the surface, SQL databases seem to be quite different, having a tabular structure, consisting of a fixed schema (a fixed n-ary predicate), the instances of which are rows in a table. Yet it is famously the case that SQL and key-value databases are categorial opposites to one-another: if one takes the arrow relationships in one, and reverses the sense of the arrows, one obtains the other (Meyer 2011).

Actual data-sets occurring in knowledge representation include WordNet, which provides synonym, holonym and meronym relations. The urge to solve the "grounding problem" leads to "upper ontologies", such as SUMO, which asserts that things are objects, and classifies types of things. Ontologies themselves force the user to make distinctions between intensional and extensional membership, intensional and extensional inheritance. For example, one can assert that dogs are a kind of animal (intensional) and then assert that "Fluffy" is an example of a dog (extensional - belongs to the class of dogs).

Ontologies provide only a handful of such relationships (is-a, has-a, part-of). To generalize to a greater number of relations, one arrives at the notion of "semantic triples": these are effectively labeled arrows: the two endpoints (head and tail) of the arrow, and a label for the relationship type.

As before, the presence of arrows implies ordering relationships, with concomitant notions of transitivity, (anti-)reflexivity, (anti-)symmetry, even when they may be a good bit more fuzzy and vague, as in the domain of "common sense". As Spock might say, human gut feels are not logical.

Graphs

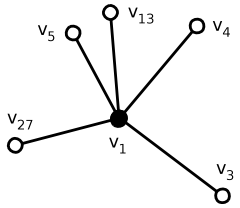
Unlike the examples above, graph theory offers a different possibility: the undirected edge. Unfortunately, this is again "painfully obvious", and we'll have to slog through a lot of rather obvious observations to make anything resembling progress.

The canonical definition of a graph is that of two sets: a set V of vertices and a set E of edges. An edge $e_k \in E$ is a pair $e_k = \{v_i, v_j\}$ of vertices, with each $v_i \in V$. Although this definition is adequate for most mathematical applications, it is severely deficient for algorithmic applications. The biggest issue is that of non-locality of data. If one merely stuffs the vertices and edges into a table, then a graph walk in general appears to look like a uniformly distributed random hop through those tables: terrible for modern computers which rely on locality, so that caches can speed data access performance.

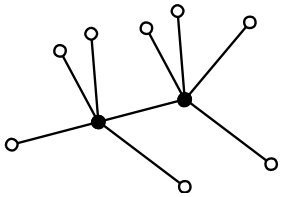
This section develops an alternative representation for graphs, and argues that it is foundational in nature. It addresses not only some of the practical problems in the DAG-view of the world (of the noosphere) but also provides an alternative to some of the philosophical foundations of the same.

Seeds

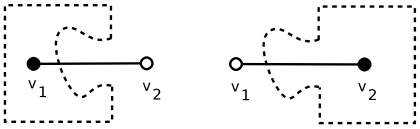
The problem of locality can be (partly) solved by defining graphs in a different way: in terms of vertexes, and the (half-)edges that attach to each vertex. In this case, one creates a table of pairs $(v_i, \{e_{i1}, e_{i2}, \cdots, e_{ik}\})$ of a vertex v_i and the set of all edges landing on that vertex. It is convenient to think of the edges as "half-edges", as it is enough to identify the remote end to which they connect; the local end is implicit: it is just the vertex v_i at the "center". As this is obvious, it can be belabored with a diagram.

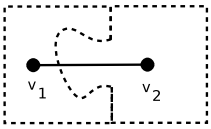


The black dot represents the vertex; the open circles represent the unconnected half-edges. As this vaguely resembles a burr, this general shape will be referred to as a "seed", or sometimes a "germ" in what follows. Connecting these proceeds in the "obvious" fashion, and so a pair of connected seeds is shown below:



The unconnected dots, the half-edges, can be called connectors, in which the jigsaw-puzzle piece analogy comes into full force. Another "obvious" diagram illustrates this:





The explicit polarity of the mating, despite the fact that the resulting edge is an undirected edge, serves several purposes. Foremost, it helps avoid mating errors: if the connectors do not have opposite polarity, they cannot attach. Secondly, they can be used to establish directionality, when that is needed. As the previous section noted, directed edges are everywhere; this did not mean to imply that they are useless! The collection of tabs and sockets can come in a large variety of shapes. These shapes correspond to the "types" of "type theory". Most importantly, the concept of vertices adorned with connectors allows one to define a concept of a grammar.

Sections and Compostionality

The burrs/seeds/puzzle-pieces, as presented above, are obviously "compositional", in that there is an obvious way to connect them together. Once connected, the resulting structure is again of the same form: some network of vertices connected with edges, and some unconnected half-edges sticking out of it. Furthermore, it should be clear that the order in which pieces are assembled is immaterial: one obtains the same figure, no matter what the sequence.

In philosophy:

The paradoxes of mereology were touched on in the previous section: the difficulty of defining part-whole relationships in a consistent manner, the problem of the existence or non-existence of "gunk", and even the puzzles from Ancient Greece (the Ship of Theseus, *etc*.) One proposed solution to these riddles is the notion of "restricted composition" (see Wikipedia). The idea is that "things" are still composed of "other things", but the manner in which these can be composed is restricted. The body of ideas is very nearly identical to the concept of sheaves. Quoting directly from the Wikipedia article on mereology, we can can learn that these are:

- Contact—that a complex object Y is composed of X's if and only if the X's are in contact. *viz.* seeds attach only if they can be connected.
- Fastenation—the X's compose a complex Y if and only if the X's are fastened. *viz.* seeds attach only if the connectors allow mating. Once mated, they are fastened together.
- Cohesion—the X's compose a complex Y if and only if the X's cohere (cannot be pulled apart or moved in relation to each other without breaking). The point of connectors is that they are not meant to be pulled apart, once mated. Insofar as the above abstract definition of a sheaf failed to appeal to and forced order of the connectors, or to the length of the edges, the concept of "movement" remains incomplete[2]
- Fusion—the X's compose a complex Y if and only if the X's are fused (fusion is when the X's are joined together such that there is no boundary). Connectors, once connected, serve no further purpose, and are discarded. What mattered was the bond that was made. In this sense, the bonds are the fusion of connectors.[3]
- Brutal Composition—"It's just the way things are." There is no true, nontrivial, and finitely long answer.

[2]This can be repaired in a variety of ways. In practical applications, it is convenient to order the edges, and to give them weights. These weights can sometimes act as distances, e.g. in N-dimensional space. Furthermore, the branch of mathematics known as "algebraic topology" is very highly developed, and can provide very explicit and concrete statements about the structure of space. For example, the dimensionality of the embedding of the network is given by the degree of the largest simplicial complex in the network.

[3]This is reminiscent of Derridas statement that "deconstruction is not analysis". One cannot break up meaning into atomic parts; there are no self-sufficient atomic parts of meaning. Meaning derives only from the fusion of words into text and into language.

The last point is perhaps the most interesting notion coming out of restricted composition. It is not an ingredient, it is rather a consequence. If one abandons the explicit DAGs of partial orders, one then also abandons the notion of logical predicates and truth assignments. One no longer arranges logical connectives into a tree structure, or even into Horn clauses. The network, rather, represents things "as they are". This is not to deny predicate logic, or Boolean satisfiability: these are tremendously useful concepts. Rather, this is to avoid the symbol grounding problem (ref Wikipedia, again). By discarding the arrow from "signifier" to "significand", one can no longer peer into the gaping chasm of the question "what does this symbol mean?"

This is very much in line with theories of postmodernism and deconstruction evident in the works of Derrida,[4] Deleuze, Lyotard. There is no difference between appearance and true form; the difference is undecidable. The constructed network of connections and relations is all there is.

In the following sections, the sheaf construction will be used as a tool to create A(G)I representations of reality. Whether the constructed network is an accurate representation of reality is undecidable, and this is true even in a narrow formal sense. Famously, it is known that the question of whether two different presentations of a (group-theoretical) group refer to the same group is undecidable, in the computational sense: there is no algorithm, guaranteed to terminate in finite time, that is capable of making this decision. This is the undecidability problem in group theory. That this result carries over into the fullness of a network built from a sheaf should come as no surprise: the presentation of a group is little more than a certain collection of labeled vertexes and edges (the group elements and the group operations) and decidability requires the graph-rewriting of these into a different form. There are no graph rewrite rules that can always bring two different presentations into normal form. Oddly enough, such "received wisdom" from mathematics can be used as a foundational cornerstone for post-modernism. This is unexpected.[5]

A more explicit bridge can be constructed via linguistics, in the following.

Tensors

Much of what has been said above can be made slightly more precise with the aid of mathematical notation. This section reviews the concept of a tensor, starting from its most narrow form, and broadening it to a general setting. It is hoped that the reader already knows what a tensor is, as otherwise, most of this may seem pointless. The explanation begins very simply, at the freshman level; do not be mislead by the simplicity, there are a few tricks in here.

[4] The différance of Derrida can be seen as a rejection of the linguistic equivalent of mereological nihilism: there is no infinite regression of looking up the meanings of words in a dictionary.

[5] Despite this, sheaves are still somehow fundamentally "structuralist", not post-structuralist. The bonds are relational, in the end. There is always still a predicate: either an edge exists, or it doesn't. To every graph there is a corresponding adjacency matrix, populated with zeroes and ones. Fortunately, one can wriggle ones way out if this, by assigning numerical weights to edges, making some probable and some improbable. Whether true and false can be replaced by a subobject classifier in this context is perhaps a step too far, but perhaps also an interesting step.

Four properties of tensors will be exposed and articulated. These may seem strange and idiosyncratic, if you already know what a tensor is; but communicating the strangeness is why we write.

- A means of storing data in a particular form or shape.
- The tensor product as a kind of concatenation and "forgetting".
- The inner product as a kind of "plugging together of connectors".
- Types and fibers.

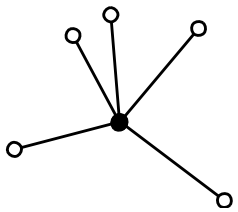
These properties can be taken in contrast to conventional expositions, which emphasizes multi-linearity and and behavior under change-of-basis. The change in focus is required to build the abstract notion of a tensor.

Tensors as databases

The conventional definition of a tensor in science and engineering starts with the introduction of the concept of a scalar s: a single number usually taken to be a real number or a complex number or even, abstractly, taken from some ring R. A vector $v = [v_1, v_2, \cdots, v_n]$ is a sequence of numbers. A matrix $M = [M] = M_{ij} = M(i,j)$ is a square of numbers, indexed by rows and columns i, j. The equality signs here just suggest different ways (different notations) of writing the same thing. A 3-tensor $T_{ijk} = T(i,j,k)$ is a cube of numbers; a 4-tensor $T_{ijkm} = T(i,j,k,m)$ is a 4-dimensional "hypercube" of numbers, and so on.

That tensors can be used to "store data" sounds a bit silly, or perhaps painfully obvious, given the conventional definition. None-the-less, tensors are a form of "database", where "data" (the number) has been "stored" at a "location" determined by the indexes (which are taken as ordinal numbers). This becomes a slightly more interesting observation when one realizes that computer-science has a small armada of similar devices for storing data: arrays and vectors and lists and lookup tables and the like. In compsci, considerations such as speed, space, accessibility, mutability become important, and strongly affect algorithms. That a tensor has the form of a database becomes even more interesting when one ponders how to store extremely sparse tensors: those whose entries are mostly zero, with a scattering of non-zero entries. Consider a matrix with one million rows and one million columns: this has $10^6 \times 10^6 = 10^{12}$ locations; were we to splat a 64-bit floating-point number down at each location, this would require 8×10^{12} bytes of storage: 8 terabytes. If all of those locations were zero, except for possibly for 10^6 here or there ... that would be an insane waste of RAM: storing a million non-zero numbers requires only 8 megabytes, not 8 terabytes.

For sparse tensors, the preferred storage mechanism takes the form of a "seed", already illustrated in a figure above. Lets repeat it:



The black dot at the center is the "data" (the number). The white dots are the "tensor indexes" – the ordinal-number valued tuple identifying the location of the data. The figure shows five connectors, and thus an entry in a 5-tensor.

Tensor products

We begin with the conventional definition of the tensor product, before generalizing it in a later section.

The tensor product is conventionally denoted with the "otimes" symbol $\otimes$. This intimidating symbol is used to emphasize the multi-linearity of the tensor product. That is, if u, v and w are vectors, and a and b are constant numbers, then

$$(au + bv) \otimes w = au \otimes w + bv \otimes w$$

for the left side of the product, and likewise for the right: that this works on both sides is what makes it multi-linear. The $\otimes$ symbol is required because the Cartesian product symbol $\times$ does not work, it is not multi-linear. Consider, for example, the two ordered pairs $u \times w = (u, w)$ and $v \times w = (v, w)$. How can we multiply a scalar times an ordered pair? Conventionally, one re-scales all of the components. That is, $a.(u, w) = (a.u, a.w)$ but this fails spectacularly in terms of multi-linearity:

$$a.(u,w) + b.(v,w) = (a.u, a.w) + (b.v, b.w) = (au + bv, aw + bw) \neq (au + bv, w)$$

The tensor product $\otimes$ is not the Cartesian product $\times$.

Tensor products as equivalence

The tensor product can be constructed from the Cartesian product by declaring an equivalence. From the multi-linearity property above, it's clear that we wish to say that $au \times w$ is the same thing as $u \times aw$. That is, we already have defined the tensor product so that $au \otimes w = u \otimes aw$ as strict equality; yet this is patently false for ordinary ordered pairs.

The procedure to rectify this situation is to introduce a new notion of equivalence. Using the symbol $\sim$ to denote "the same as", one writes $au \times w \sim u \times aw$. The usual laws of algebra should apply, so subtraction can be used to bring everything over to one side: $au \times w - u \times aw \sim 0$. That is, the difference of these two ordered pairs is the "same as" zero. Switching notation for ordered pairs, one may write $(au, w) - (u, aw) \sim 0$. The algebra is meant to behave "as expected", and so

$$(au, w) - (u, aw) = (au - u, w - aw) = ((a-1)u, (1-a)w) \sim 0$$

But $a-1$ is just a number $c = a-1$, so the above says

$$(cu, -cw) \sim 0$$

for all numbers c. Such ordered pairs are "equivalent" to zero.

The formal way to write the tensor product is as a quotient over this equivalence. This is conventionally written as

$$U \otimes W = U \times W / \sim$$

which is meant to denote the set of all equivalent ordered pairs. In set notation,

$$u \otimes w = \{\text{all pairs } (s,t) \text{ such that } (s,t) = (u,w) + (cu, -cw) \text{ for some const } c\}$$

This is to be read "the set of all pairs to which we have added the equivalent of zero". Formally, the set on the right is called "coset". By treating all members of this coset as "equivalent", we "forget" their identity and uniqueness (stemming from their origins as Cartesian pairs), and treat them as being all the same. Choosing any one exemplar from the coset will do; it is a form of "forgetting" of differences, or an "erasure" of origins. It is a denial of identity politics, it is a certain racial homogenization, it is a democratic notion that "all are created equal".

Tensor products as concatenation

The above was needlessly complicated. There is a wildly simpler way of saying the same thing, which, remarkably, arrives at the same place. Let $F(x,y,z)$ be a function of variables x, y, z which are understood to be ordinal numbers, that is, integers. Without any further restrictions (for the moment, as we ignore change-of-basis, for now), the function $F(x,y,z)$ is a tensor, and the x, y, z are the tensor indexes. Given another function aka tensor $G(s,t)$, the tensor product is simple the tensor

$$T(x,y,z,s,t) = F(x,y,z)\, G(s,t)$$

where the product is simply the scalar product. This seems almost trivial in it's definition – how hard can a scalar product be? Its just the ordinary multiplication of numbers. This seems effectively trivial, but it hides a bit of trickery: there's a sleight-of-hand. The tensor on the left-hand side is written as $T(x,y,z,s,t)$ and not as $T((x,y,z),(s,t))$. If we look at (x,y,z) and (s,t) as two ordered lists, then (x,y,z,s,t) is the concatenation of those lists. It is NOT the Cartesian product of them!

As always, let's belabor the painfully obvious:

$$(x,y,z,s,t) \neq (x,y,z) \times (s,t) = ((x,y,z),(s,t))$$

The right side is a list. The left side is a list of lists. List concatenation "erases" the nested parenthesis that appear on the right-hand-side. It "forgets" where the indexes

came from. Indeed, it might have been the case that the list was the result of concatenating (x,y) with (z,s,t) – we simply don't know, we forgot this bit of information. We "erased" it, the origins have been "democratized": we only have a list (x,y,z,s,t) but know not whence it came.

To illustrate the correspondence, list concatenation can also be written with an equivalence principle. Using the symbol $\sim$ to denote "the same as", one writes

$$(x,y,z)\times(s,t)\sim(x,y)\times(z,s,t)$$

Using the equivalence $\sim$ to write a quotient space of concatenated products requires the development of some additional notion. This is partly undertaken in the next section.

This works for conventional tensors because F,G and T are considered to be maps to "numbers", elements of a field or maybe a ring. The multiplication of numbers (members of a field or ring) is also "forgetful": when we write "42", we don't know if it was constructed from the product of 6 and 7, or from the product of 2 and 21. It could have been either of these, or yet more. To put it differently, if $F(a,b,c)=6$ and $G(d,e)=7$ for fixed constants a,b,c,d,e, and we construct the product $T(a,b,c,d,e)=F(a,b,c)\,G(d,e)$, we no longer know where the 42 came from. It might have come from $T(a,b,c,d,e)=H(a,b)\,K(c,d,e)$ where $H(a,b)=2$ and $K(c,d,e)=21$. The forgetfulness of list concatenation goes hand-in-hand with the forgetfulness of multiplication in fields and rings. Tensors work because the exploit both of these properties.

Cartesian products and lambda calculus

The simply-typed lambda calculus is famously the internal language of Cartesian-closed categories. Let's take a moment to unpack that statement. The treatment here is a bit informal; our goal is not to teach category theory. Consider a class S of symbols. We call it a class because it could be a finite set, or an infinite set; it may be uncountable, or it may be so horridly structured that it cannot be expressed as a set. For the purposes of comp sci, things are always finite, co calling it a class S is merely conventional. The elements of S are "symbols" because, for the purposes here, symbols, and the connotation that they "stand for something", is an important property.

The Cartesian product of elements of S is an ordered list: we already wrote above that $a\times b\times c=(a,b,c)$ is two different ways of writing the same thing. It is convenient to drop the commas, and write $(a,b,c)=(a\,b\,c)$ as is standard in Lisp-dialect programming languages. Of course, one can likewise construct lists-of-lists, and so on. Consider now the collection (class) of all such nested lists-of-lists-of-lists... A typical exemplar might look like

$$(a\,b\ (c\,d)\ e\ (f\ g\ (h\,(j\,(k\,(m))))))$$

whence the acronym LISP - "Lots of InsidiouS Parenthesis" comes from. Here, the letters were taken to be symbols drawn from the class S. It is, however, convenient to introduce the notion of variables; for these, we write $x,y,z,\cdots$ as always, by convention. One can then consider the class of lists with embeded variables in them. OK, but things really get interesting when one then considers replacing variables by values. To do this,

the special notation of λ is introduced, and one conventionally writes

$$(\lambda x.A)\, B \to A\,[x := B]$$

where both A and B are lists (possibly containing variables), and the expression on the left-hand side of the arrow is a lambda-binding of the variable x such that any occurrence of the variable x in the list A is to be replaced by the list B. By convention, that is what the expression on the right hand side of the arrow is supposed to mean. The arrow itself is meant to denote beta-reduction, the actual act of plugging in or substitution of the variable x by the thing B that is to be plugged in.

Beta reduction takes actual effort and work, it is a computation problem, and vastly complex schemes have been developed to perform beta reduction rapidly. From the comp-sci perspective, it is a highly non-trivial process, no matter how obvious it may seem from the short expression above.

That's it. That's the effective definition of simply-typed lambda calculus. It is "simply typed" because all members of the class S are taken to be of the same type. There were only four ingredients in the construction:

- The Cartesian product, which allows the construction of lists (and of lists of lists ...).
- The use of variables as "placeholders".
- The use of a special symbol λ which is used to call out or bind or name a specific variable.
- The performance of substitution, named "beta-reduction", for historical reasons.

There is, of course, much more that can be said about lambda calculus; this hardly scratches the surface. But it does show the centrality of the Cartesian product to the construction.

It also helps highlight just how different list concatenation is from the Cartesian product of lists. This is a source of tremendous confusion for students of engineering and science, and so (as always) it is worth belaboring here. By convention, one is introduced to the notion of $\mathbb{R}^n$ as the n-dimensional Cartesian space. It is a coordinate space – the points of Cartesian space are labeled by n coordinates, taken to be real numbers, *i.e.* taken as elements of $\mathbb{R}$. By convention the Cartesian product of such spaces is $\mathbb{R}^m \times \mathbb{R}^n \cong \mathbb{R}^{m+n}$ where the $\cong$ symbol denotes isomorphism. By convention, this works as list concatenation: if one has spatial coordinates (x,y,z) in the 3D space $\mathbb{R}^3$ and the spatial coordinates $(s,t) \in \mathbb{R}^2$, then, "of course", $(x,y,z,s,t) \in \mathbb{R}^5 = \mathbb{R} \times \mathbb{R} \times \mathbb{R} \times \mathbb{R} \times \mathbb{R}$. Notationally, this is a bit unfortunate for our purposes, as it seems to be a form of list concatenation. Even worse, the right hand side is not a tensor product! Are you confused yet? Posed as a riddle, the solution to the riddle is to realize that when one writes a tensor T_{ijklm} of five indeces, these indeces are ordinal numbers: integers. They are not real-number-valued coordinates of Euclidean space. The value of T might be real, but the indexes are not. The abundance of notations - parenthesis of various shapes, and commas and what-not, can be a bit of a trap.

Tensor product, in general

We now have enough machinery developed to allow a general definition of the tensor product. Lets return to the form

$$T(x,y,z,s,t) = F(x,y,z)\,G(s,t)$$

as a simple product of $F(x,y,z)$ and $G(s,t)$. This time, we allow F and G to be "anything with a forgetful product" – not numbers, but anything which can be "multiplied" together, in some forgetful way, so that the origin of the factors is no longer identifiable. We had been previously vague as to whether the x,y,z,s,t were variables or constants; we may proceed with this vagueness, except that now we generalize them not to be ordinals, but to be members of some class S. The ordinal property of the indexes is not relevant, here.

Replacing numbers by things that can be forgetfully multiplied discards the addition of tensors. This, however, is an important property that we want to maintain. This can be done by going "meta-mathematical" in the conventional sense, and using the disjunctive-or to combine generalized tensors. The disjunctive-or is a "menu choice": it says "pick this or pick that, pick at least one, but don't pick both". If we have generalized tensors T and U, we can no longer write $T+U$ because there is no addition, but we can still combine them with disjunctive choice: I can present you with $T \vee U$ and demand that you pick either T or U. By convention, the disjunctive-or is denoted with $\vee$.

One can still use the $\otimes$ symbol to write the tensor product, or one can omit it entirely, as was done above, when we wrote $T = FG$. It is often useful keep a tensor product symbol, but, as we've generalized, it's conventional to use a slightly different symbol: the ampersand &, and so write $T = F\&G$. We now have two symbols: & and $\vee$ and can ask what the algebra of these symbols is. Note very carefully that it is NOT the Boolean algebra. We have one distributive property, but not the other. So,

$$(u \vee v)\,\&w = (u\&w) \vee (v\&w)$$

is the alternate (generalized) form of the conventional distributive property

$$(u+v)\otimes w = u\otimes w + v\otimes w$$

However, the other one, that would have made things Boolean, does not hold:

$$(u\&v) \vee w \neq (u\&w) \vee (v\&w)$$

This is hardly a surprise, since it is also the case that

$$(u\otimes v) + w \neq u\otimes w + v\otimes w$$

which is once again obvious and trivial: tensor products do not form a Boolean algebra.

Tensor logic

To round out the algebraic constructions for the tensor product, we can do the same thing that was done to obtain (simply-typed) lambda calculus:

- Define the algebra of & and $\vee$ over a class of symbols S
- Introduce variables as "placeholders".
- Employ the symbol λ to bind or call out a specific variable.
- Enable beta-reduction.

This glosses over the need for alpha-conversion and possibly eta-reduction. I will call this algebraic system "tensor logic", as I am not aware of any conventional name for it. It is similar to "linear logic", but is not the same, as it is missing an important ingredient, that of conjugation.

Inner product

Tensors become interesting in engineering and science only after the addition of one more ingredient: the inner product. The inner product allows tensors to be combined by "contracting tensor indeces". For example, given tensors A_{ijk} and B_{lm} one might consider the contraction

$$C_{ijm} = A_{ijk}B_{km} = \sum_k A_{ijk}B_{km}$$

where the middle expression uses the so-called "Einstein convention" of summing repeated indexes, whereas on the right the sum is explicit. For the moment, the distinction between covariant and contra-variant indeces will be ignored; as any discussion of change-of-basis has been ignored, it remains appropriate to also ignore issues of contra/covariance.

The prototypical inner product is that of two vectors, say $\vec{a} = [a_1, a_2, \cdots, a_n]$ and $\vec{b} = [b_1, b_2, \cdots, b_n]$. One writes:

$$i\left(\vec{a}, \vec{b}\right) = \vec{a} \cdot \vec{b} = a_1 b_1 + a_2 b_2 + \cdots + a_n b_n$$

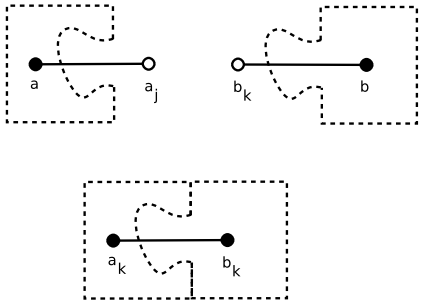
For the general case, one needs only to convert notation:

$$i\left(\vec{a}, \vec{b}\right) = a_1 \& b_1 \vee a_2 \& b_2 \vee \cdots \vee a_n \& b_n$$

which clearly has the same form despite a vastly different interpretation of the symbols.

Inner product as connection

Diagrammatically, the contraction of tensor indeces was earlier depicted with a puzzle-shaped diagram, repeated below, with slightly different labeling.



The diagram is slightly awkward here; the upper row is meant to depict the two vectors, with naked, uncontracted indexes. The lower part depicts a scalar, with Einstein-convention repeated indexes that are summed over. The two different mating puzzle-tab shapes hint at a hidden contra/covariance. They are also meant to indicate compatible types. That is to say, the type of a_j is compatible with the type of b_k, and so can be multiplied: so, they might both be numbers, or both be something else, as long as they can compatibly be multiplied with each-other using the product & between them. We return to types shortly. The point of the diagram is to re-assert the claim that the inner product is just a form of "connecting things together", this time made notationally explicit.

Beta reduction as inner product

Earlier, also it was claimed that beta-reduction was a form of connecting things together. How can this be? To recap: when one plugs in 42 for x in $f(x)$ to get $f(42)$, one is "connecting" 42 to $f(x)$. This is intuitively obvious, and one can even intuit how the jigsaw-puzzle diagram captures this idea. Yet it is not (yet) obviously an inner product.

To demonstrate that it is, one may appeal to the most basic type theory to elucidate. Here, 42 is an exemplar of the set of "integers", and $f(x)$ is a "function that takes integers". They can mate together because the types are compatible. We can write an integer as

$$1 \vee 2 \vee 3 \vee \cdots$$

which is a menu choice - the disjoint union, again. It says that "you should choose either 1 or choose 2 or choose 3 ... but choose at least one, and chose no more than one". Likewise, the result of plugging in is

$$f(1) \vee f(2) \vee f(3) \vee \cdots$$

This begins to look like an inner product; the resemblance can be completed by writing it as

$$f_1\&1 \vee f_2\&2 \vee f_3\&3 \vee \cdots$$

where, of course, $f_1\&1 = f(1)$ is how one writes "the value of f at one can be obtained by mating together the value one, as a jigsaw-puzzle-tab, with f_1 as the corresponding matching jigsaw-puzzle mate". With the revised notation, its now clear that beta-reduction is a form of the inner product.

To write it this way, there was a slight abuse of the notation &. Previously, it was defined as the "product of two things of the same type"; yet here it is used to denote "the mating of matching types". It is convenient to read it both ways; for the moment we sweep this under the rug, along with the co/contravariance issue. They are not entirely unrelated.

Types and fibers

Until now

Applications

In linguistics:

Why, link-grammar, of course. its a grammar. Its a categorial grammar. Type theory. Pre-group grammar. Stay and Baez rosetta stone. And

Sheaf Axioms:

It obeys the sheaf axioms.

In AI:

Inference, theorem proving, planning, constraint satisfaction.

The planning aspect is that one can arrange a collection of unconnected connectors at one end (the starting state) and another at the other end (the goal state) and build a bridge between the two.

frame problem, combinatorial explosion.

Conclusions:

There is no such thing as non-fiction; all writing is fundamentally fictional at it's root. The only way to be non-fictional is to be quarks and space-time, the universe itself.

The above is a fancy way of saying that analytic philosophy is blarney, and that the quest for symbolic AI is doomed. One cannot unambiguously assign speech-act labels to text, or symbols to meaning. The problem lies beyond mere "fuzziness" or "probability", Bayesian or otherwise.

Yet, despite these problems, there is a way forward in relationships, without insisting on cause and effect, premise and inference, (Bayesian) prior and deduction. Directionality can be useful, but it is an infelicity, a trap with yawning abysses of foundational problems, not the least of which is the concept of "well-foundedness" itself. Restricted compositionality is more powerful and more general.

AtomSpace

The AtomSpace is a graph database intended to provide generic support for knowledge representation, logical inference, and symbolic AI reasoning. It provides a large variety of operators for constructing terms and defining relations, as well as a fairly comprehensive type system for assigning types to variables and working with signatures of functions and terms. In this sense, it incorporates many features that are commonly accepted in quotidian computer science. There is some question about how the existing AtomSpace implementation should inter-operate with the idea of connectors and bonds. The goal of this section is to review the existing type system, and how it can interface with a connector-and-bond system.

The Atomese Type System

The existing Atomese type system is an extensible system of types usually defined at compile time. Although new types can be added at run-time, it is usually useful to have a corresponding object-oriented class definition for each type, so that the types can "do things" (be evaluatable, perform numeric operations, perform i/o operations, etc.) As a result, the type hierarchy is somewhat limited and fairly rigid. Although there are more than 100 different types in use in the current system, only a handful of these are relevant to the current discussion. A quick sketch of these is given below.

The core concept underlying Atomese is the ATOM: a globally-unique immutable structure which can be referenced and can be used as an anchor for mutable, varying VALUEs. All ATOMs derive from two basic types: the NODE and the LINK. The NODE is simply a (UTF-8) string. All NODEs having the same string name are in fact the same NODE. Very crudely, one can think of NODEs as named vertexes of a graph. The LINK corresponds to a hyper-edge: a LINK can contain zero, one, two or more ATOMs; it is a possibly-empty list of ATOMs. LINKs to not have any other properties; in particular, they have no string name. Thus, crudely, one may think of a LINK containing exactly two NODEs as an ordinary graph edge. Like NODEs, LINKs are globally unique: there can only ever be one LINK containing a given list of ATOMs. It is very convenient to think of a LINK as a vertex internal to a tree; the NODEs are the leafs of the tree. The trees are necessarily acyclic, and finite; it is not possible to construct a LINK that contains itself.

The AtomSpace is a container for ATOMs. As each ATOM is effectively a tree, one can think of it as a forest-of-trees. However, since each ATOM is globally unique (within an AtomSpace), a more accurate conceptual visualization is that of a rhizome or woolen felt, as different trees typically share the same branches. In the following, s-expressions will be used to write down trees.

VariableNode

Declares the name of a variable. For example, `(VariableNode "x")`.

TypedVariableLink

Associates a type declaration with a variable. For example, `(TypedVariable (Variable "x") (Type "ConceptNode"))`. This states that the the variable is necessarily of type "CONCEPT", with CONCEPTNODE being one of the many predefined types in the system. The TYPENODE is just a NODE whose string name must be the string representation of a simple type.

SignatureLink

A mechanism to declare or construct a compound or complex type. For example, `(Signature (EvaluationLink (PredicateNode "foo") (ListLink (Type "Concept") (Type "Concept"))))` constructs a type consisting of two CONCEPTs, and given a fixed label of "foo". The specific meaning of EVALUATIONLINK, LISTLINK and PREDICATENODE are of no particular concern here, they're merely examples of some of the predefined types in the Atomese system.

Other type constructors

The system contains several other type constructors, including those for type union and type intersection.

Bibliography

- Marius Buliga (2003). "Artificial chemistry experiments with chemlambda, lambda calculus, interaction combinators." https://arxiv.org/abs/2003.14332
- Erik Meijer and Gavin Bierman (2011). "A co-Relational Model of Data for Large Shared Data Banks." *ACM Databases* Volume 9, issue 3.
- Van Inwagen, P. (2002). "The Number of Things." *Philosophical Issues* 12 pp.176–196.