Connectors and Variables

Linas Vepstas

17 December 2021*

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 versions: 2 May 2020; 21 August 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)[1]

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

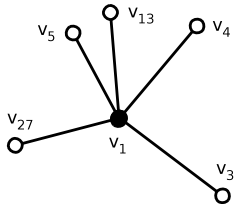
Unlike the examples above, graph theory offers a different possibility: the undirected edge. Unfortunately, this is again "painfully obvious", and we'll have to slog through a lot of rather obvious observations to make anything resembling progress.

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.

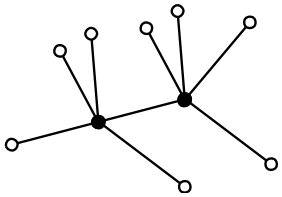
This section develops an alternative representation for graphs, and argues that it is foundational in nature. It addresses not only some of the practical problems in the DAG-view of the world (of the noosphere) but also provides an alternative to some of the philosophical foundations of the same.

Seeds

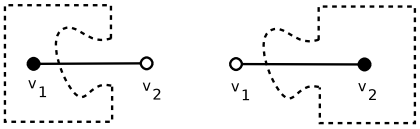
The problem of locality can be (partly) solved by defining graphs in a different way: in terms of vertexes, and the (half-)edges that attach to each vertex. In this case, one creates a table of pairs $(v_i, \{e_{i1}, e_{i2}, \cdots, e_{ik}\})$ of a vertex v_i and the set of all edges landing on that 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.

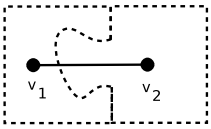


The black dot represents the vertex; the open circles represent the unconnected half-edges. As this vaguely resembles a burr, this general shape will be referred to as a "seed", or sometimes a "germ" in what follows. Connecting these proceeds in the "obvious" fashion, and so a pair of connected seeds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

Polarity (Sexuality)

The above presentation of the concept of jigsaw pieces is misleading in one important sense, which should be clarified. The jigsaw pieces, as drawn, are explicitly heterosexual (bipolar), in that there are explicit male and female connectors. In general, this need not be the case: one can have monopolar (asexual) connectors, or tri-sexual connectors, or connectors of arbitrary complexity.

This is worth articulating. In the previous diagram, there was a slot and a tab, which can be indicated as $-$ and $+$. They had an explicit shape, which we can call "type A". A different shape might be called "type B". The connection rules are such that $A+$ can only connect to $A-$ and never to $B+$ or $B-$. But also, $A+$ cannot connect to other $A+$'s. This "makes sense", and is a desirable property if one is using jigsaw pieces to model beta reduction: a variable of type A can only be beta-reduced with a term of type A. That is, variables are always $-$ and those things that can be "plugged into" variables are always $+$.

There are other possibilities. The monosexual (monopolar) case would be a connector direction of $*$ with a rule that any $A*$ can connect to any other connector of type $A*$ but never to a connector of type $B*$. There is no real analog of this in term algebras: such algebras always make a sharp distinction between variables and constants; yet the monopolar case fails to distinguish these. Connecting two connectors together is still like beta reduction; but the sense of direction is lost.

An example of a trisexual connector would require connectivity rules similar to the following. There are three direction types, #, \$ and % and the rules are that \$ can connect to # and \$ can connect to % but # cannot connect to %. Nor can \$ connect to \$. (Confused? Look at the US keyboard layout: these three keys are in order, with \$ in the middle. The rule is "neighbors can connect, but not otherwise"). There are still the connector-shape types $A, B, \cdots$ and so we extrapolate that $A\$$ can connect to $A\#$ and to $A\%$ but not to $B\$$ or to $B\#$ or $B\%$.

Neither the monosexual nor the trisexual cases are easy to illustrate in terms of pretty hand-drawn pictures of jigsaw pieces. The strict idea of jigsaw pieces does break down.

Representation of Sexuality

Wandering a bit off-topic, it is worth pointing out that there is a representation theory of the multi-polar direction types in terms of monopolar and bipolar direction types, together with strings of connector-types.

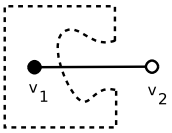
This is illustrated here. For the connector strings, lower-case letters will be used. For # write the string $r*-$ and for $ write $rs+$ and for % write $*s-$. Now the mating rules are these: the wild-card $*$ matches any lower-case letter; otherwise lower-case letters must match exactly. As before, $+$ can only pair with $-$. Thus, $r*-$ can mate with $rs+$ because the r matches, the $*$ matches anything, and the $+$ and $-$ will mate. Likewise, $rs+$ can mate with $*s-$. However, $r*-$ and $*s-$ cannot mate: although the wild-cards allow this, the poles do not.

This remark was necessary, since Link Grammar[2, 3] uses this style for indicating connectivity. Capital letters must match exactly; strings of lower-case letters with embedded wild-cards match is the strings match, and the poles match. Thus, the theory can be understood as having bipolar connection directions, with complex string-matching rules, or as simple upper-case types, and multi-sexual connector directions. As per usual, a rigorous mathematical treatment of this representation theory may reveal subtleties beyond what has been said here. Or maybe not. That this string representation is sufficient to represent any kind of sexual mating remains an unproven conjecture at this point.

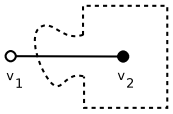
In proof theory:

A simple variant of the idea of a jigsaw piece can be found in proof theory, where it is called a "context" or a "formula occurrence". See, for example Troelstra & Schwichtenberg, "Basic Proof Theory"[4] section 1.1.4. A "context" is a "formula" with a "hole cut out of it". The hole is called a "placeholder" or a "special propositional variable", sometimes denoted with a star-character. The star is outside of the language of predicate logic; it's in the meta-language. A "formula occurrence" is a subformula together with a context. Roughly speaking, it is a subformula cut out of a formula, leaving a hole in it's place. That pairing, of a subformula and the place it was cut from, is the "formula occurrence".

Presumably it is self-evident that this resembles the pair of unconnected jigsaw pieces in the diagram above. Perhaps not, so we belabor it. Here is the context:

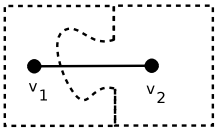


The formula is v_1 and the hole is v_2. Here is the subformula:



The "actual" subformula is v_2 but we've drawn a little more, here: the v_1 is the shadow of the formula from which the subformula as cut.

These two pieces together constitute the original formula:

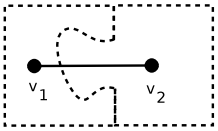


Drawn assembled like this, this is just the formula. Drawn disassembled gives the formula occurrence.

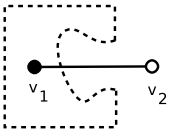
In lambda calculus

The sketch of contexts and formula occurrences in proof theory resembles the concept of eta expansion in lambda calculus. In lambda calculus, if f is an expression with some free variable in it, then $\lambda x.f(x)$ is it's eta expansion. Roughly speaking, this reads as the operation: "cut out the free variable x from f, leaving a hole, and then mark the location of that hole with a bound variable x, binding that variable with lambda, so that we know where that hole is."

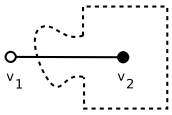
Presumably it is self-evident that eta expansion can be understood as jigsaw connectors. But then again, it is worth belaboring this point. So here is a diagram showing the expression f



More precisely, the above shows the expression f as v_1 and with v_2 being some free variable inside of it. Disconnecting this gives the expression $f(x)$ below:



In the above, the v_1 is the expression f, with the v_2 explicitly providing the location of the x. What is the lambda? Well it is just this, below:

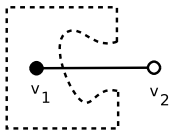


It is tempting to say that the above is just λx, but it is actually just a bit more: it is λx with the shadow of f as v_1 still visible. The eta expansion $\lambda x.f(x)$ is just the two jigsaw pieces, disassembled.

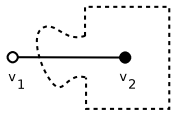
In type theory

The phrase "shadow of" in the above text is perhaps disconcerting. This is where type theory can come to the rescue. The "shadow" is the type. The matching of the type

of the variable, to the type of what can be plugged into the variable, is what the shape of the connector implies. So, again, her e is $f(x)$ below, where v_2 is understood as the free variable x being of type A:

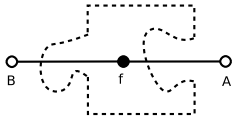


Here is the bound variable λx which is again of type A:



That is, x itself is just v_2, and it takes the half-edge to indicate that it's type is A. The connector itself is the type.

Again, this may seem strange and foreign, which might be clarified with a more complex example. The type constructor $A \to B$ constructs functions. it can be understood as a lambda expression, in the sense that given a term b of type B, containing a free variable x of type A, then $\lambda x.b$ is a term of type $A \to B$. How is this to be represented in terms of jigsaws? With the diagram below:



This time, the connectors have been explicitly labeled with the types. The above is explicitly an instance of the type $A \to B$. The open circle with the capital A indicates that anything that is of type A can be connected there; connecting-together is beta-reduction. The open circle labeled with B denotes that this jigsaw (labeled f) can be plugged into anything that is expecting type B.

The above exercise hopefully shows how connectors can be understood as types. However, the theory of jigsaw pieces is not type theory. The type constructor $A \to B$ is explicitly directional, the jigsaw pieces are not: a jigsaw piece that is flipped, rotated, inverted is still a jigsaw piece; functions are always directional. Reversing the arrow gives an inverse function, but an inverse function is still directional, just going the other way. This is in part because type theory is closely tied to the internal language of category theory, which is itself a theory of dots and arrows. By contrast, the jigsaw conception provided here is derived from a graph theory where the directionality of the edges is immaterial, or of secondary importance.

In philosophy:

The paradoxes of mereology were touched on in the section on relational algebras. For an extended discussion of the philosophical issues, see the text titled "Mereology and Postmodernism", in "mereology.pdf" in this directory.

Tensors

Much of what has been said above can be made slightly more precise with the aid of mathematical notation. This is provided in the chapter on tensors; see "tensors.pdf" in this directory.

Types and fibers

TODO: talk about how vectors can have different indexes of different types on them. A prototypical example are the quark wave functions, which have a color index, a flavor index, a spin index and a momentum index.

Follow outline given in email.

Sheaf Axioms

It obeys the sheaf axioms. See "sheaf-axioms.pdf" in this directory.

Learning

Searching, parsing and learning are closely related. See "learning.pdf" in this directory.

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

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.

Inductive logic programming

See, for example, the Wikipedia article on this.[2] The learning of sheaf structure is a generalization of "inductive logic programming", where the specific axioms of logic have been abstracted away. More precisely, logic has an explicit directionality to all of it's rules and axioms; sheaves can not only capture this directional, but they also provide a non-directional generalization. XXX TODO: this entire paragraph needs to be extended to a full-sized paper.

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

[2] See https://en.wikipedia.org/wiki/Inductive_logic_programming.

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.

References

[1] P. Van Inwagen, "The Number of Things", *Philosophical Issues*, 12, 2002, pp. pp.176–196.

[2] Daniel Sleator and Davy Temperley., *Parsing English with a Link Grammar*, Tech. rep., Carnegie Mellon University Computer Science technical report CMU-CS-91-196, 1991, URL http://arxiv.org/pdf/cmp-lg/9508004.

[3] Daniel D. Sleator and Davy Temperley, "Parsing English with a Link Grammar", in *Proc. Third International Workshop on Parsing Technologies*, 1993, pp. 277–292, URL http://www.cs.cmu.edu/afs/cs.cmu.edu/project/link/pub/www/papers/ps/LG-IWPT93.ps.

[4] A. S. Troelstra and H Schwichtenberg, *Basic Proof Theory, Second Edition*, Cambrdige University Press, 2000.