Every Typed Value is a Hypernode: A generalized hypergraph representation for data structures and computer code

May 8, 2019

Abstract

GHG

—

Computational and scientific data representation intrinsically balances two competing priorities: "semantic" expressiveness and computational tractability. On the one hand, representations should not obscure important details: the formal requirements on representational validity should not force representations into structures that necessitate the elimination of meaningful information. On the other hand, conversely, representations should have enough structural consistency that they are amenable to analysis and transformations driven by formal algorithms.

I have just left a lot of loose ends: in particular, these comments need to give some meaningful definition to "representation". There are many media wherein "data" can be represented: via graphics (e.g., charts, "maps" in the cartographical sense, or "graphs" in the sense of plotted visualizations of mathematical functions), via printed documents (like the logarithmic tables or astronomic records of early modern science), or via mathematical equations and formulae (if a mathematical theory correctly predicts and quantifies empirical data — e.g., fitting trajectories to elliptic or parabolic shapes — then the numeric structures of the theory are a proxy representation of the corresponding data). Moreover, aside from these relatively formal modes of representation, we have the capacity to indirectly describe information via natural language.

The modern age has a further notion of representation: *digital* representations which leverage the capabilities of computer storage and processing to create persistent data repositories, with the possibility of manipulating data via computer programming and displaying data via computer software. These digital representations incorporate features of the older representational media I alluded to: like Natural Language digital data often emanates from a textual representation, with data structures initialized from special "languages" like XML or JSON. As with structured "printed" documents (of the astronomical-table or even grocery-shopping-list variety), digital representations often build off of a rigid structural template, like a two-dimensional table with rows and columns, or a hierarchical document whose elements can be nested documents. And as with mathematical representations, digital representations can be analyzed as instances of spaces or structures with certain algebraic or syntactic properties and requirements. It is often possible to mathematically describe the full space of structures which conform to a given representational protocol, or the full space of transformations that can modify a given data structure while staying consistent with its enforced protocols. Also, each data structure potentially has some notion of aggregateness: of having different "parts", of being able to focus on one part at a time, and to "move" focus from one part to another. Given these qualities, digital representations extend and integrate the affordances of print, graphical, and linguistic media — but they do so in an electronic environment that permits digital archives to be accessed via computer software. With sotfware as their access point, digital representations can inherit the formal properties of the software that uses them, so we can think of digital representations as formal structures themselves, amenable to something like a mathematical analysis.

In short, digital representations have several general criteria: *validity* (a formal model of conformant vs. in-

valid structures[1]); *traversability* (a notion of parthood and iterating or refocusing among parts), and, let's say, *atomicity* (a notion of unitary parts that can be represented as integral wholes). These criteria ensure that digitally represented data structures can work within software and networking representations: information is presented to software users by displaying atomic units (textually or graphically) and traversing through data structures to fill in, via application code, a visual tableau presenting compound data, with different unitary displays visually separated and often organized into coherent visual patterns, like the grid-pattern of a spreadsheet. Meanwhile, atomic units can be textually encoded, and aggregate structure likewise notated through syntactic conventions that preserve atoms' boundaries, yielding textual serializations of data structures that can be shared between computing environments, allowing information to be copied and distributed. Finally, digital representations of data structures can be marshalled into different binary forms — encoded in byte- and bit-patterns — to enable both persistent storage in a database and "runtime" presence as binary data that may be accessed by software applications.

I take the time to lay out these basic principles because I want to emphasize the different contexts where digital representations can be found: in particular, textual encoding and serialization; graphical/interactive displays for software users; application runtimes; and long-term database storage. A given representation will morph and mutate to accommodate these different contexts. Moreover, these contexts correspond to distinct technical specializations: database engineers have a different perspective on data structures than network engineers designing protocols for encoding and exchanging data between network endpoints; and application designers focused on optimal human-computer-interaction understand digital representations as visual and interactive phenomena, whereas application *developers* need to focus on how to properly encode data structures for binary runtimes. These various perspectives all influence the theory and technology behind digital data representation — a successful representational paradigm needs to adapt to the operatoonal requirements of engineers in each of these disicplines.

[1] An invalid graph might be a case where an edge has no incident nodes; an invalid XML structure is one without a single root element or (insofar as such a structure would be representable) with mismatched tags, and so forth)

Additionally, insofar as the point of digital archives is to encode empirical, "real-world" data, a proper representational protocol needs to promote a synergy between the information as humans understand it and the data structures recognized by the technology. For instance, if a published data set shares scientific data, it should be represented in ways that preserve scientifically significant details — any derivations, descriptions, or observations which are intrinsic to the science's methodology, theory, assumptions, and experimental results. The data needs to be structured according to the "semantics of the science", so that the scientific background can be reconstructed along with future use of the data, even after a circuitous journey through different contexts, like through a database and over a network to a scientific-software application (maye years later). As formal models of data semantics have become more rigorous — e.g. in our century with Ontologies and the "Semantic Web" — this "semantic engineering" has become a further technical perspective needing consideration in the design and evaluation of digital representations.

Over the decades, many digital representation strategies and "layouts" have been envisioned, from tabular structures in the manner of Relational Databases, to tree-like documents whose morphology is inspired by markup languages, to structurally looser variations on row/-column arrangements like multi-dimensional key-value spaces or "Big Column" and other "NoSQL"-database-inspired articulations. These various paradigms are subject to selective pressures based on how well they meet the different engineering needs I have identified. But the multiplicity of requirements complicates the "competition" between representational strategies: a paradigm optimized for one context (e.g., persistent databases) is not necessarily optimal for another (e.g., application and GUI development). As a result, developers and computer scientists must continue to explore and collaborate on new paradigms which work better across contexts.

In the plurality of representational paradigms, a significant evolution has been the emergent popularity of structures based on graphs. The predominant influence on this line of development has been the Semantic Web and the specific graph architecture codified by "Web Ontologies", although other graph-representation models (such as Conceptual Graph Semantics) were also candidates for potential technological "entrenchnment" before Semantic Web formats like RDF and OWL became

standardized.[2] More recently, scientists have proposed generalizations and/or alternatives to the Semantic Web based on hypergraphs in lieu of ordinary (directed, labeled) graphs.

As might be ascertained from hypergraphs being the focus of this paper, I believe that representational paradigms based on hypergraphs can be superior to other formats and need to be studied with an integrated, generalized set of theories and computer tools. I have introduced the topic of hypergraph data structures via general issues in digital representation in part to rest claims of hypergraphs' merit on explicit criteria. In particular, I believe hypergraphs can adapt to the different technological contexts prerequisute to a decentralized digital ecosystem — databases, networking, application implementation, visual/interactive software design, and semantic expressiveness — more effeectively than other paradigms, like regular graphs or SQL-style tables.

I suspect other scientists and engineers have similar intuitions, because there has been a recent uptick in research on hypergraphs in various disciplines, such as Category Theory, Information Management, Artificial Intelligence, and Natural Language Processing. Compared to the Semantic Web, however, there is a noticeable lack of standard tools or formats for expressing hypergraph data or sharing hypergraph structures across multiple applications and environments. Whereas RDF and OWL are definitively associated with the Semantic Web, so that supporting these standards is a basic entry point for Semantic Web technologies, there is no comparable consensus on the underlying theory of hypergraph data in general.

This situation may be explained in part by subtle differences on what the word "hypergraph" is supposed to mean in different settings, so the overal terrain of hypergraphs is divided into distinct mathematical models, often without a rigorous theory of their interrelationships. Another problem is perhaps a failure to appreciate how hypergraphs are structurally different from ordinary graphs, so that hypergraph-shaped data may be imprecisely treated as a mere variation upon or a special structuration within ordinary graphs. For example, Semantic Web structures such as RDF "bags" and "collections" introduce a kind of hierarchical organization to Semantic Web graphs — qualifying as a form of hypergraph — but these protocols are not described as enabling a transition from a networking framework based on directed labeled graphs to one based on hypergraphs. Instead, hypergraphs become *de facto* embedded within ordinary graphs, exploiting the representational flexibility which also makes the Semantic Web suitable for spreadsheets, XML, and other structures that are not graphs at a *prima facie* level. The problem is that while hypergraphs can be *encoded* in ordinary graphs with a suitable labeling convention, their distinct structural advantages are lost as explicit architectural features once hypergraphs are "encoded" in conventional Semantic Web formats.

As I will outine in the first section, there are at least some half-dozen different structures that may certainly be described as hypergraphs, generalizing various underlying graph models (or indeed, even if we restrict attention to labeled directed graphs). Within this space of possibilities, recent computer-science-related) research into hypergraphs seems to emphasize two different topics: first, the representational potential for hypergraphs as a general morphology for encoding structured information-resources (so as a general tactic for data warehousing and modeling); and, second, the specific possibilities of using hypergraphs to model computer code and computational processes. In the second subject-area, hypergraphs are studied as a medium for expressing details about computer algorithms as a way to reason about executable computer code as a structured system, and perhaps even to design execution engines (virtual machines to run computer code that is in a suitable representation). This latter research sometimes appears to present a mathematical picture of hypergraphs as an abstract model of computing procedures and evaluations — a kind of graph-based interpretation of the lambda calculus — but sometimes is also marshaled into implemented execution environments, as in the OpenCog and lmnTal frameworks.

My goal in this paper is to consider what a unified hypergraph paradigm can look like — how the different strands which in their own way embody hypergraph structures can be woven together into a multi-purpose whole. My emphasis is on computer implementations rather than mathematics — that is, I will not present

[2] Why RDF/OWL and not, say, Conceptual Graphs, or the hybrid Object-Database/Graph-Database models studied in the 1990s, became canonized in the Semantic Web, is an interesting question but perhaps of mostly historical interest insofar as Hypergaphs can unify each of these paradigms.

axioms or theorems formalizing different sorts of hypergraphs, though I ackowledge that such descriptions are possible. Instead, my aim is to describe what might constitute a general software library or toolkit that can adapt to different hypergraph models and use-cases. This paper is accompanied by a "data set" which involves a library (written in C++) for creating and manipulating hypergraphs of different varieties, and also a "virtual machine" for modeling and realizing computatational procedures via hypergraph structures. The point of the accompanying code is to demonstrate that a generalized hypergraph representation is possible, and that in addition to use-cases for modeling data structures such a representation can be used at the core of a virtual processor. The code includes simple tests and "scripts" that can be executed via the provided virtual-machine code.

Studying hypergraphs from a computational (and "implementational") perspective — not just as mathematical objects — introduces useful details that can add depth to our overall understanding of hypergraphs. For example, one question is how hypergraphs can be initialized — how software systems can build up hypergraph structures, piece by piece. This is related to the question of proper serialization formats for hypergraphs. Given some specific hypergraph data aggregate $\mathcal{H}$, it is important to have a textual encoding where $\mathcal{H}$ can be mapped to a character-string, shared as a document, and then reconstructed with the same hypergraph structure. This raises the question of when and whether two hypergraphs are properly isomorphic — such that serializing and then deserializing a hypergraph yields "the same" hypergraph — and also the problem of validating and parsing textual representations of hypergraphs. The theory of parsing *serializations* of hypergraphs — textual encodings constructed according to a protocol wherein their grammar and morphology are suitable to expressing and then rebuilding hypergraph structures — then becomes an extension of hypergraph theory itself.

In addition to creating hypergraphs via textual representations in the manner of markup languages (like XML or RDF), we can also consider the incremental accumulation of hypergraph structures via minimal units of transformation, providing a kind of "Intermediate Representation" embodying the form of a hypergraph via a sequence of effectively (at least on one scale of analysis) atomic operations. For any variety of hypergraph, then, we can consider which is a suitable Intermediate Representation for conformant spaces — in particular, which set of primitive options can, iteratively, construct any representative of the space of instantations of a given kind of hypergraph. Insofar as a software framework seeks to work with several hypergraph varieties, we can consider how several different such operation-sets can be combined.

Moreover, we can consider both serialization and Intermediate Representation as parallel tactics for building hypergraphs. An Intermediate Representation language can be used to carry out transformations between hypergraphs — producing IR code based on the first hypergraph from which the second can be populated. In conjunction with serialization and parsers, a hypergraph can then be realized via several stages, with one form of hypergraph used as an intermediary because it has a convenient grammar and works with a parsing engine; from that preliminary graph a new graph can be created via IR code. Grammars, serializaion and deserialization, and IR thereby become part of the formal architecture defining a hypergraph ecosystem and inter-graph transformations (analogous to the trio of XML parsers, XML transforms, and the XML Document Object Model).

Continuing the XML analogy, note that XML is not just a serialization specification: the full XML technology defined requirements on a series of tools operating on XML data, not just XML files: tools for traversing XML documents (including a programming interface for how traversal options are to be operationalized, e.g. via Object-Oriented methods like parentNode and nextSibling); for parsing XML files into traversable data structures (including requirements on the traversals which the derived structurees — the so-called "post-processing infoset" — must support); and for inter-document mapping (via XSLT or XML-FO). The Document Object Model and post-processing infoset is the structural core of the XML format, even more than any surface-level syntax (the grammar for tags, attributes, and so forth). Analogous specifictions would have to be developed for a generalized Hypergraph representation.

The code discussed here does not purport to provide a mature or decisive implementation of a general-purpose hypergraph ecosystem, but rather to demonstrate by example what components may comprise such a framework and how they may interoperate. The code includes hypr-

graph models but also the preliminary and intermediary structures that can connct hypergraphs to a surrounding computational infrastructure — parsers, Intermediate Representation, application-integration logic (such as mapping hypergraph nodes to application-specific data types), and a "virtual machine" for realizing hypergraphs as executable code.

The remainder of this paper will focus on three subjects: delimitating the proper range of hypergraph models; execution models and the "virtual machine"; and a review of formal structurs (like grammars and IR formats) which are structurally different than hypergraphs but can help tie hypergraph representations togther into a unified ecosystem. I will also, in conclusion, sketch arguments to the effect that hypergraphs offer a plausible foundation for reasoning about structured/computation data and date types in general. I have spoken only informally about hypergraphs themselves, taking for granted that we have a general picture of what hypergraphs are, but this now demands more rigorous treatment. There are actually several different research and engineering communities that talk about hypergraphs, in each case describing some sort of generalization of regular (often labeled and/or directed) graphs, but these generalizations fall into different kinds. Hence there are several different varieties of hypergraph, and I will try to outline a general theory of hypergraphs in these various forms.

1 Varieties of Hypergraphs

Any notion of hypergraphs contrasts with an underlying graph model, such that some element treated as singular or unstructurd in regular graphs becomes a multiplcity or compound structure in the hypergraph. So for example, an edge generalizes to a hyperedge with more than two incident nodes (which means, for directed graphs, more than one source and/or target node). Likewise, nodes might generalize to complex structures containing other nodes (including, potentially, nested graphs). The "elements" of a graph are nodes and edges, but also (for labeled/weighted and/or directed graphs) things like labels, weights (such as probability metrics associated with edges), and directions (in the distinction between incoming and outgoing edges incident to each node). Potentially, each of these elements can be transformed from a simple unit to a plural structure, a process I will call "diversification". That is, a hypergraph emerges from a graph by "diversifying" some elements, rendering as multiplicities what had preciously been a single entity — nodes become node-sets, edges become grouped into larger aggregates, labels generalize to complex structures (which we can call "annotations"), etc. Different avenues of diversification give rise to different varieties of hypergraphs, which I will review in the next several paragraphs.

Hyperedges Arguably the most common model of hypergraphs involves generalizing edges to hyperedges, which (potentially) connect more than two nodes. Directed hyperedges have a "source" node-set and a "target" node-set, either of which can (potentially) have more than one node. Note that this is actually a form of node-diversification — there is still (in this kind of hypergraph) just one edge at a time, but its incident node set (or source and target node-sets) are sets and not single nodes. Another way of looking at directed hyperedges is to see source and target node-sets as integral complex parts, or "hypernodes". So a (directed) hypergraph with hyperedges can also be seen as generalizing ordinary graphs by replacing nodes with hypernodes.

Recursive Graphs Whereas hyperedges embody a relatively simple node-diversification — nodes replaced by node-sets — so-called "recursive" graphs allow compound nodes (hypernodes) to contain entire nested graphs. Edges in this case can still connect two hypernodes as in ordinary graphs, but the hypernodes internally contain other graphs, with their own nodes and edges.

Hypergraph Categories and Link Grammar Since *labeled* graphs are an important model for computational models, we can also consider generalizations of labels to be compound structures rather than numeric or string labels (or, as in the Semantic Web, "predicate" terms, drawn from an Ontology, in "Subject-Predicate-Object" triples). Compound labels (which I will generically call "annotations") are encountered (sometimes implicitly) in several different branches of mathematics and other fields. In particular, compound annotations can represent the rules, justifications, or "compatibility" which allows two nodes to be connected. In this case the

annotation may contain information about both incident nodes. This general phenomenon (I will mention further examples below) can be called a "diversification" on *labels*, transforming from labels as single units to labels as multi-part records.

Channels Hyperedges, which connect multiple nodes, are still generally seen as single edges (in contrast to multigraphs which allow multiple edges between two nodes). Analogous to the grouping of nodes into hypernodes, we can also consider structures where several different edges are unified into a larger totality, which I will call a *channel*. In the canonical case, a directed graph can group edges into composites (according to more fine-grained criteria than just distinguishing incoming and outgoing edges) which share a source or target node. The set of incoming and/or outgoing edges to each node may be partitioned into distinct "channels", so that at one scale of consideration the network structure can be analyzed via channels rather than via single edges. For a concrete example, consider graphs used to model Object-Oriented programming languages, where a single node can represent a single function call. The incoming edges are then "input parameters", and outgoing edges are procedural results or outputs. In the Object-Oriented paradigm, however, input paramters are organized into two groups: in addition to any number of "conventional" arguments, there is a single this or self object which has a distinct semantic status (vis-à-vis name resolution, function visibility, and polymorphism). This calls, under the graph model, for splitting incoming edges into two "channels", one representing regular parameters and a separate channel for the distinguished or "receiver" object-value.

In each of these formulations, what makes a graph "hyper" is the presence of supplemental information "attached" to parts of an underlying (labeled, directed) graph. As a concrete example, consider a case from linguistics — specifically, morphosyntactic agreement between gramatically linked words, which involves details matched between both "ends" of a word-to-word "link" (part-of-speech, plural/singular, gender, case/declension, etc.). According to the theory known as "link grammar", words are associated with "connectors" (a related useful terminology, derived more in a Cognitive-Linguistic context, holds that words carry "expectations" which must be matched by other words they could connect to). The word *many*, for instance, as in *many dogs*, carries an implicit expectation to be paired with a plural noun. The actual syntactic connection — as would be embodied by a graph-edge when a graph formation is employed to model parse structures — therefore depends on both the expectations on one word in a pair (whichever acts as a modifier, like *many*) and the "lexicomorphic" details of its "partner" ("dog", as a lexical item, being a noun, and *dogs* being in plural form).

In Link Grammar terminology, both the expectations on one word and the lexical and morphological state of a second are called "connectors"; a proper linkage between two words is then a *connection*. For each connection there is accordingly two sets of relevant information, which might be regarded as a generalization on edge-labeling wherein edges could have two or more labels. Furthermore, the assertion of multi-part annotations on edges permits edges themselves to be grouped and categorized: aside from several dozen recognized link varieties between words (which can be seen as conventional edge-labels drawn from a taxonomy, consistent with ordinary labeled graphs), edge-annotations in this framework mark patterns of semantic and syntactic agreement in force between word pairs (not just foundational grammatic matching, like gender and number — singular/plural — but more nuanced compatibility at the boundary between syntax and semantics, such as the stipulation that a noun in a locative position must have some semantic interpretation as a place or destination). Insofar as these agreement-patterns carry over to other word-pairs, annotations mark linguistic criteria that tie together multiple edges in the guise of signals that a specific parse-graph (out of the space of possible graphs that could be formed from a sentence's word set) is correct.

Ordinary directed graphs (not necessarily hypergraphs) already have some sense of grouping edges together, insofar as incoming and outgoing edges are distinguished; but this indirect association between edges does not internally yield a concordant grouping of the nodes at the sources of edges all pointing to one target node (or analogously the targets of one source node). In the theory of hypergraph *categories*, hypernodes come into play insofar as representation calls for the nodes "across" incoming/outgoing edges to be grouped together; we distinguish an *incoming node-set* from an *outgoing node-*

set:

> The term hypergraph category was introduced recently [Fon15, Kis], in reference to the fact that these special commutative Frobenius monoids provide precisely the structure required for their string diagrams to be directed graphs with 'hyperedges': edges connecting any number of inputs to any number of outputs. ... We then think of morphisms in a hypergraph category as hyperedges [5, p. 13].

For a general mapping of categories to graphs where edges represent morphisms, this represents a generalization on the notion of *edges* themselves. Suppose morphisms are intended to model computational procedures in a general sense (say, as morphisms in an ambient program state). Because procedures can have any number (even zero) of both inputs and outputs, this implies a generalization wherein directed edges can have zero, one, or multiple source (and respectively target) nodes. A corresponding hypergraph form is one where hyperedges have source- and target- hypernodes, but each hypernode models variant-sized sets of further "inner" nodes (or "hyponodes"); here the empty set can be a hypernode with no hyponodes. Elsewhere, apparently an equivalent structure is called a "trivial system":

> Monoidal categories admit an elegant and powerful graphical notation [wherin] an object A is denoted by a wire [and] [a] morphism $f : A \to B$ is represented by a box. The trivial system I is the empty diagram. Morphisms $\mu : I \to A$ and $\nu : A \to I$ ... are referred to as **states** and **effects** [3, p. 4].

The system I — which we can also see as a hypernode with an empty (hypo-)node set — may embody a procedure which has no internal algorithmic or calculational structure (at least relative to the domain of analysis where we might represent computer code). In Cyberphysical Systems, a function which just produces a value (with no input and no intermediate computations) can also be called an *observation*, perhaps a direct reading from a physical *sensor* (accordingly, as in the above excerpt, a *state*). Dually, a procedure which performs no evident calculation and produces no output value, but has a cyberphysical *effect*, can be called an "actuation", potentially connected to a cyberphysical *actuator* (an example of a sensor would be a thermostat, and an example of an actuator would be a device which can activate/deactivate a furnace and/or cooling system).

In these examples I have mentioned hypergraphs in a linguistic (Link Grammar) a mathematical (hypergraph categories) context. Both of these have some parallels insofar as a core motivation is to generalize and add structure to edges, either freeing edges fron constraints on node-arity (even allowing edges to be "unattached", or supplying edges with structured (potentially multi-part) annotation.

A somewhat different conception of hypergraphs is found in database systems such as HypergraphDB. Graphs in that context express what have been called *recursive* graphs, wherein a hypernode "contains" or "designates" its own graph. Thebasic idea is that for each (hyper-)node we can associate a separate (sub-)graph. This can actually work two ways, yielding a distinction between (I'll say) *nested* and *cross-referencing* graphs. In a *cross-referencing* graph, subgraphs or other collections of graph elements (nodes, edges, and/or annotations) can be given unique identifiers or designations and, as a data point, associated with a separate node. Consider a case where nodes refer to typed values from a general-purpose type sytem; insofar as subgraphs themselves may be represented as typed values, a node could reference a subgraph by analogy to any other value (textual, numeric, nominal/enumerative data, etc.). Here the (hyper)node does not "contain" but *references* a subgraph; the added structure involves subgraphs themselves being incorporated into the universe of values which nodes nay quantify over. Conversely, *nested* hypergraphs model hypernodes which have other graphs "inside" them, thereby creating an ordering among nodes (we can talk of nodes at one level belonging to graphs which are contained in nodes at a higher level). Such constructions may or may not allow edges across nodes at different levels.

Note that nested hypergraphs can be seen as a special case of cross-referencing graphs, where each hypernode $\mathfrak{n}$ is given an index i, with the restriction that when $\mathfrak{n}$ designates (e.g. via its corresponding typed value) a subgraph $\mathfrak{s}$, all of $\mathfrak{s}$'s hypernodes have index $i-1$. Cross-referencing graphs, in turn, can be seen as special cases of an overall space of hypergraphs wherein hypernodes are paired with typed values from some suitable expressive type system $\mathbb{T}$. If $\mathbb{T}$ includes higher-order types

— especially, lists and other "collections" types which become concrete types in conjunction with another type (as in *list* qua generic becomes the concrete type *list of integers*), then hypernodes acquire aggregate structure in part by acquiring values whose types encompass multiple other values. I will use the term *procedural* hypergraphs to discuss structures that model node diversification via mapping hypernodes to collections-types (with the possibility for hypernodes to "expand" or "contract" as values are inserted into or removed from the collection).

Cross-referencing also potentially introduces a variant derivation of hypergraphs which proceeds by accumulating graph elements (nodes, edges, and labels or annotations) into higher-scale posits, rather than defining inner structures on elements. Specifically, as a complimentary operation to diversification, consider "aggregation" of graph elements: the option to take a set of (hyper)nodes, (hyper)edges, and/or annotations as a typed value which can then be assigned to a (hyper)node. In such a manner, higher-level structures can be notated with respect to graphs, which is one way to model phenomena such as *contexts* — the kind of multi-scale patterns that are considered endemic to practical domains like the Semantic Web. While it is informally acknowledged that a single-level interpretation of the Semantic Web is misleading — the Semantic Web is not an undifferentiated mesh of connections, but rather an aggregation of data from many sources, which implies the existence of localization, contextualization, and other "emergent" structure — there is no defininitive protocol for actually representing this emergenet structure. This issue, in turn, is one of arguments for hypergraphs in lieu of ordinary graphs as general-purpose data representations.

The multi-scale, contextualized nature of the Semantic Web also points toward a conceptual duality in *how* graphs represent data. On the one hand, graph structures — especially in the case of the Semantic Web, which builds off of internet technology in general — represent *relationships* between points or structures of data in some sense; in familiar web terms, the relata linked by graph edges are often "resources", designated by unique web addresses. A more theoretical model might take the information "residing" at graph nodes as typed values. But in either case a given node may stand in f for an aggregate of information — a single web resources may contain a theoretically unlimited supply of data, and a typed value can be of a list or tuple type intlernally containing its own body of information. Consequently, the full stock of data embodied in a graph may not lie primarily in the graph structure itself, but rather distributed among its nodes (that is, among notes' associated data).

Conversely, graph structures (with suitable semantic specifications) are also considered to be media for serializing arbitrary data structures, which implies representing all details, at all levels of hierarchical organization, via graph structures. Insofar as nodes embody their own information spaces, such internal structuration must then be mapped to their own graphs, as part of a workflow to project arbitrary structured data onto a canonical format. The theoretical correlary to this idea would be that node data (via associated typed values) has its own internal representation; that it is, every typed value has a corresponding graph structure that may be "contained" within a higher-scale node.

Different varieties of hypergraph forms complicate this picture because the structuring elements of hypergraphs include aggregate data within nodes as well as the space of edges and incidence relations. Given the structures I have referred to as "procedural" hypergraphs, nodes can encompass multiple internal values so long as they have a suitably well-defined internal structure. We can analyze these possibilities by defining, for each hypernode, an "interface" or list of operations available for updating hypernodes' associated values. Which operations are proper depends on a hypernode's type: hypernodes associated with a single unstructured value should have one basic update operation, while nodes with list-like types would have operations to insert (and remove) values at different positions.

Separate and apart from operations modifying hypernodes' values, there are also conventional graph operations — adding and removing nodes and edges. In combination, the graph-oriented and node-oriented operations present a variegated interface for manipulating procedural hypergraphs. Such an interface then serves as a rigorous characterization of the overall hypergraph model — the structuring elements expressed by enumerating graphs' transformation operations represent the particular features of each specific hypergraph variety. In the case of procedural hypergraphs, many of these transformations are not graph-related per se but derive from hypernodes' collections or tuple types. I contend that this is a useful property of procedural hypergraphs for reasons

I alluded to in the introduction — hypergraphs (or at least the data represented with them) need to work in a variety of computational contexts. The relatively unstructured form of graph data is not always appropriate from one context to another; the list-of-values or value-tuple structures embodied as hypernode data may be more consistent with internal representations in database or language-runtime engines, for example. Procedural hypergraphs are appropriately flexible in that some data is modeled at the graph level proper while other data is modeled as lists, tuples, and similar data structures within individual nodes.

In many practical contexts graph structures are not implicitly used at all; the importance of Semantic Web-style representation is for intermediary structures, where information is routed ampong different environments (database, applications, serialization, and so forth). Transformations between hypergraphs can be a central process in generic transformations between data structures proper to different contexts. In effect, where there is a general need for data tranforms in routing between contexts — e.g., database to application runtime — we can hope to capture most of the relevant transform logic as specifically mappings between hypergraphs, with each hypergraph possessing a structure optimized for being initialized from one context (or for generating data used in another context). This progression may involve restructuring wherein information modeled at the inter-hypernode level — which we can call *hypernode* data — tends to be migrated to the level *inside* hypernodes, which we can call *hyponode* data. In broad outline, hypergraph transforms can progress from relatively informal structures (with a preponderance of information expressed as *hypernode* data) to more constrained structures — mapping hypernode to hyponode data, i.e., mapping data from structures *between* hypernodes to those *within* hypernodes — where the hyponode data is regulated by hypernodes' types.

In this section I identified different additions through which graph structures generalize to hypergraphs; a general-purpose hypergraph engine would need to support each of these variations, which entails enabling the complete repository of transform-operations applicable to different hypergraph varieties. This includes generalizing edges to hyperedges by "diversifying" nodes to encompass mutiple values; representing nodes' internal structure in terms of data structures such as lists, tuples, and nested graphs; and generalizing edge-labels to annotations which may have multiple parts. I have not yet discussed in detail the possibility of grouping hyperedges into higher-level structures which I have called "channels", but I will return to those details in a later section.

For the remainder of this paper, I will attend especially to hypergraphs modeling (and subsequently executing) computer code, because conctructing a working runtime engine which runs source code, via hypergraph intermediaries, demonstrates a variety of concepts applicable to hypergraphs in general. I will discuss a workflow connecting parsers, a runtime "virtual machine", intermediate representations for hypergraphs, and inter-graph transforms. Collectively this workflow implicates many of the capabilities which would be requisite for a general-purpose hypergraph software ecosystem.

Interested readers who would like to observe a concrete unfolding of this workflow are invited to download the code base accompanying this paper, where readers can examine the operations of parsers and code generators working with a built-in, general-purpose hypergraph library. The downloadable dataset is fully integrated with Qt Creator, a C++ Integrated Development Environment, and has no further dependencies (assuming users have a working Qt and C++ compiler; Qt is a popular C++ application-development framework). The dataset includes instructions for experimenting with the hypergraph library via Qt Creator and examining runtime structures by executing demonstration scripts in Debug mode.

2 Hypergraph Parsers and Intermediate Representation

Having pointed out the features desired for a general-purpose hypergraph framework, the next problem is to implement a hypergraph library supporting these various features. One dimension of this problem is the proper in-memory representation of hypergraph structures and hypergraph elements (hypernodes, hyperedges, and so forth) — to implement the datatypes and procedures requisite for hypergraphs as software artifacts. I will not emphasize these internal details directly, except for making implementation-related observations when relevant

to other contexts (the accompanying code demonstrates one approach to building an in-memory hypergraph library, though I make no claim that the implementation is optimized for speed or to "scale up" to large graphs; my priority was instead to demonstrate hypergraph semantics, representing multiple hypergraph varieties, as expressively as possible).

A second stage in the implementation process, which I will examine here in greater detail, involves transformations and initialization of hypergraphs — creating hypergraphs from other sources (such as text files) and using hypergraphs to initialize other kinds of data structures, either directly or indirectly (e.g., via code generators). This dimension encompasses the overall capabilities allowing software to use hypergraphs for their own data representations and/or as bridge structures for sharing data between applications and/or components. In bridge cases, the strategies for building hypergraphs *from* other kinds of data, and then building other kinds of data from hypergraphs, are equally significant to techniques for modeling hypergraphs themselves.

On the *input* side, hypergraphs may be initialized via several routes. The most direct path is to program function calls that can modify (and incrementally build up) hypergraphs directly. If hypergraphs are expressed via a cohort of C++ classes (for graph, node, edge, etc.), that means using C++ method calls to perform operations like adding a node or adding an edge between nodes. Less directly, hypergraphs can be initialized from text files, given a standard format for textually encoding hypergraphs. For conventional (non hyper-) graphs, standard formats include RDF, GRAPHML, and Notation-3.[3] Analogous standards for hypergraphs would be more complex because of the extra structure involved. Nevertheless, the serialization can be expressly organized to facilitate hypergraph initialization; in essence, the grammar can be standardized to unambiguously map test encodings to low-level function-calls via which hypergraph are built directly. A still less direct input strategy would involve more flexible input languages, designed for ease of use from the point of view of people composing or reading serializations. Such encodings then require more complex grammars and parsers for the step of mapping input test to the intended hypergraph structures.

[3]Some graph formats support certain hypergraph features, but not the full spectrum of features I outlined in the last section

Similar alternatives apply to *output*, in the sense of initializing other structures from hypergraph data. Output structures can be compiled directly via function-calls, or indirectly via software-generated text files, which in turn coul have a more low-level or more high-level format. Suppose hypergraphs are employed within publishing software to generate XML and LaTeX output. This can happen mostly at the software level: there are many "XML builder" libraries allowing XML documents to be constructed via function calls, rather than by actually producing XML code. Alternativey, the software can act as a code generator, expressly composing XML or LaTeX code by creating the requisite character strings — a process which can be complicate by the languages' syntactic requirements, such as the composition of XML tags and LaTeX commands with the requisite braces and brackets. Compared to higher-level markup builders, code generation can be difficult because software has to take responsibility for every character in the output — every closing brace, bracket, quote, and so on. In between these two alternatives — using builder libraries and explicit code generation — is the possibility of generating code in alternative formats that are optimized for machine processing, and consequently bypass some of the complications attending to generating normal (say) XML or LaTeX. For instance, event-driven or "SAX" parsers understand XML as a sequence of "events", like *open-tag*, *close-tag*, *annotation*, or "character data". Some projects adopt this principle to generate XML from non-XML sources (see for example [10], [11]): any data source can be treated as an XML front-end if there is a processor that can map the data to an XML event-stream. On such a basis developers can design encoding languages that model XML event-streams explicitly, as an alternative to normal XML syntax. Similar techniques work with other formats, including hypergraphs (as I will discuss below).

Whatever the format, generating output from hypergraphs is closely tied to the issue of traversing or navigating within hypergraphs. In this context it is useful to consider a hypergraph as a kind of "space", with a notion of "points" or "locations". Suppose a particular graph element (hypernode, hyperedge, annotation, and values which may be contained in them) can be singled out as "foreground". Technology then needs some notation for moving from one foregrounded element to another — a hypernode to an incident edge, or instance. In con-

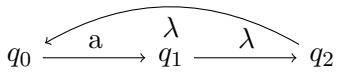

Figure 1: M1

junction with event-driven output models as mentioned last paragraph, hypergraph-traversal yields productive output algorithms so long as each successive foreground element can be mapped to one or more output events.

For this paper I will examine these and related concepts in the specific context of hypergraph execution engine. Specifically, consider the challenge of building a programming language base on hypergraphs as internal representations. In this case the input format is not tailored to hypergraphs, but rather to a grammar for source code which reflects the language's high-level paradigms (Object Orientation, Functional Programming, etc.). Likewise, the output format is not a relatively high-level markup language like XML, but rather something like a virtual machine or intermediate language which can translate to low-level (e.g., C or C++) function-calls. The pipeline demonstrated here also includes two different hypergraph formats, along with a textual serialization to marshall between them. Accordingly, the system also demonstrates initialization of hypergraphs from a hypergraph-specific intermediate representation.

All told, the language engine reviewed here has a pipeline covering five or six steps, outlined in Figure 1 . The process begins by parsing source code (designed as a vaguely C-like scripting language) yielding one style of hypergraph, with graph manipulation functions tied to the grammar's production rules. Those hypergraphs (after some preliminary analysis) are then traversed to produce an intermediate representation from which a second genre of hypergraphs is initialized, one better-suited for generating executable instructions. This second hypergraph is then traversed in turn, outputting code in a special-purpose "virtual machine" language. As a final step, this last code is then evaluated by mapping instruction to relatively low-level (mostly C++) function-calls.

References

1 Benjamin Adams and Martin Raubal, *A Metric Conceptual Space Algebra*.
https://pdfs.semanticscholar.org/521a/cbab9658df27acd9f40bba2b9445f75d681c.pdf

2 Benjamin Adams and Martin Raubal, *Conceptual Space Markup Language (CSML): Towards the Cognitive Semantic Web*.
http://idwebhost-202-147.ethz.ch/Publications/RefConferences/ICSC_2009_AdamsRaubal_Camera-FINAL.pdf

3 Bob Coecke, *et. al.*, *Interacting Conceptual Spaces I: Grammatical Composition of Concepts*.
https://arxiv.org/pdf/1703.08314.pdf

4 Brendan Fong, *Decorated Cospans* https://arxiv.org/abs/1502.00872

5 Brendan Fong, *The Algebra of Open and Interconnected Systems* https://arxiv.org/pdf/1609.05382.pdf

6 Peter Gärdenfors and Frank Zenker, *Theory Change as Dimensional Change: Conceptual Spaces Applied to the Dynamics of Empirical Theories*. *Synthese 190(6)*, pp. 1039-1058, 2013. http://lup.lub.lu.se/record/1775234

7 Michael Anthony Smith and Jeremy Gibbons, *Unifying Theories of Objects*
http://www.cs.ox.ac.uk/jeremy.gibbons/publications/uto.pdf

8 David Spivak and Robert Kent, *Ologs: A Categorical Framework for Knowledge Representation*
https://journals.plos.org/plosone/article/file?id=10.1371/journal.pone.0024274&type=printable

9 Gregor Strle, *Semantics Within: The Representation of Meaning Through Conceptual Spaces*. Univ. of Novi Gorici, dissertation, 2012.

10 Baltasar Tranc'on y Widemann, *et. al.*, *Automized Generation of Typed Syntax Trees via XML*
http://pizza.cs.ucl.ac.uk/xse01/ready/10.pdf

11 Martin Westhead, *BinX – The Binary XML Description Language* http://buphy.bu.edu/~brower/SciDAC/doc/BinXIntro2.pdf