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

May 11, 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-

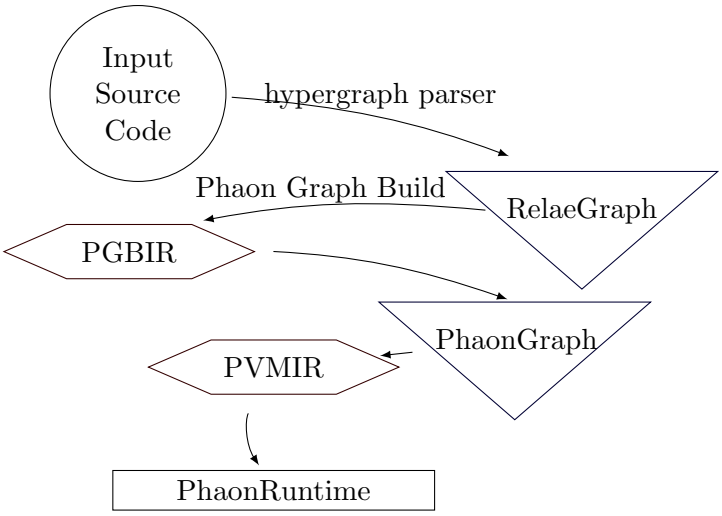

Figure 1: M1

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.

The remainder of this section will examine some of the code and techniques associated with these different processing step, which hopefully can be proxies for investigation of techniques related to hypergraphs in general.

General-Purpose Hypergraph Parsers The parser outlined here I will term "general-purpose" because, athough it outputs hypergraphs, it assumes that the languages being processed are not consciously designed for hypergraphs. Instead, hypergraphs are useful because they provide greater flexibility in capturing features of the input language, which helps for designing formal (e.g., sripting, query, or markup) languages to a desired level of expressiveness and contest-sensitivity. In particular, the parsing capabilities and space of parsable grammars is expanded relative to mechanisms like BNF grammars. (As a practical matter, a further benefit of methods such as I will outline is that they can be realized in a normal, e.g. C++ code library rather than needing a separate code-generator as in LEXX/YACC.)

I dub the actual C++ parsing library (and corresponding hypergraph reprecentation) "RelaeGraph" (for "Regular Expression Labelling and Annotation Engine"). RelaeGraph grammars are mostly comprised of rules that each pair a regular expression (a formula describing patterns in the input stream) with code that is executed in the event of a pattern match. This code, in turn, typically calls functions to modify the output hypergraph by inserting a new hypernode and/or hyperedge. A common senario is that the text matched by a rule's regular-expression pattern forms the basis of a hypernode, while the specific rule that matched (and, for context-sensitive languages, the contexts in effect for the match) determines how a new hyperedge, connecting the new hypernode to the existing hypergraph, is labeled and annotated (here annotations are added details supplementing a basic vocabulary of labels).

It is interesting to consider the connections between regular-expression oriented grammars and graphs in general: a single regular-expression pattern-match can be

pictured as part of an edge, with the matched text becoming a node, and the successful rule (via a name or identifier) supplying an edge label. The current parsing context then determines a site where the new node and edge are added to an output graph under construction. This picture is very approximate; in pratice, further processing is necessary to establish correct node content, edge labels, and parsing context. But it shows an intuitive parallel between parsing theory and hypergraphs: parsers identify pieces of input text as anchored into grammatical contexts, with relations between textual elements and surrounding (or distant) text stipulated by syntactic forms. Analogously, hypernodes and hyperedges are tied to the surrounding material in their hypergraphs according to different patterns and contexts. The extra detail offered via hypergraphs permits contextual details to be explicated more precisely and compactly. Generalizing from graphs to hypergraphcs is then analogous (and, in practice, an implementation for) generalizing from contest-free grammars to more nuanced, context-sensitive grammars.

In a context-senitive grammar, not every rule-match will directly modify an output graph; instead, the result of a match might be to alter the parsing context. For example, Listing 1 shows code sampled from the data set that works with string literals in script code, via a distinct string-literal parsing context (the code provisioned directly to a rule-match, as at (1), delegates most logic to a "graph builder" class that acts as an intermediary between the parsing-specific code and the class which directly modifies the output hypergraph; cf. (3)). The current parsing context is tracked by a specific `parse_context_` object which is modified by the graph builder (the parsing engine accesses that object as well and only attempts rule-matches for rules described as active for the current context).[4] Overall then the parsing response operations are split between three main classes: one which defines rules and match-callbacks themselves; one which is principally responsible for maintaining parse context and flags and bridging the other two layers; and one which performs most graph-transformation operations.

Listing 1: Contextual Parsing for String Literals

```
add_rule( flags_all_(parse_context ,inside_string_literal),
  run_context,  "string-literal-character",
  " (?: [^\"\\\\]+ ) | (?: (?:\\\\++)(?!\") ) ",
 [&]
{
 QString str = p.match_text();
 graph_build.add_to_string_literal(str);
});

add_rule( flags_all_(parse_context ,inside_string_literal), (2)
 run_context,  "string-literal-maybe-end",
 " \\\\*\" ",
 [&]
{
 // depends on how many back-slahes ...
 QString str = p.match_text();
 if(str.size() % 2) // odd mans even backslashes ...
 {
  if(str.size() > 1) (1)
  {
   str.chop(1);
   graph_build.add_to_string_literal(str);
  }
  graph_build.process_string_literal();
  parse_context.flags.inside_string_literal = false;
 }
 else
 {
  graph_build.add_to_string_literal(str);
 }
});

add_rule( run_context, "string-literal-start",
 " \" ",
  [&]
{
 graph_build.string_literal_start();
});
... (3)
void RE_Graph_Build::process_string_literal()
{
 caon_ptr<RE_Token> token = new RE_Token(string_literal_acc_);
 string_literal_acc_.clear();
 token->flags.is_string_literal = true;
 add_run_token(*token);
}
```

Of these three parsing layers, one layer "faces" the input text directly — the layer that defines grammar rules and match-callbacks. Another layer "faces" the hypergraph output directly — it adds (and occasionally rearranges) hypernodes and hyperedges (the third layer acts as an intermediary). Separating these layers establishes a separation of concerns so that the grammar-specific layer can focus on recognizing parsing structures for an expressive, flexible input language, without itself attending to graph operations. Simutaneously, the components which do moiy hypergraphs can be developed in isolation from the syntax of the language being parsed. Listing 2 samples code logically following Listing 1 wherein a new hypernode (such as, a string literal

[4] To be precise, context is set both by the named contexts that may be dynamically declared in a grammar and by boolean flags which assert finer-grained contexts with conjunction, disjunction, and negation operations. A full context declaration can look like at (2) in Listing 1: `flags_all_(parse_context ,inside_string_literal), run_context`.

Listing 2: Hypergraph Modification Code

```
void RE_Markup_Position::add_token_node(
  caon_ptr<RE_Node> token_node)
{
...
 if(flags.active_type_indicator_node)
 {
  flags.active_type_indicator_node = false;
  current_node_ << fr_/qy_.Type_Indicator >> token_node;
  if(token)
  {
   token->flags.has_type_indicator = true;
  }
  return;
 }
...
 check_add_implied_call_entry();

 caon_ptr<RE_Node> prior_current_node = nullptr;

 const RE_Connectors* pConnector;
 const RE_Connectors* aConnector = nullptr;
 caon_ptr<RE_Node> aConnector_src_node = nullptr;

 switch(position_state_)
 {
...
 const RE_Connectors& connector = *pConnector;

 switch(position_state_)
 {
 case Position_States::Root:
  {
  caon_ptr<RE_Node> block_entry_node =
    insert_block_entry_node(rq_.Run_Block_Entry);
  caon_ptr<RE_Block_Entry> rbe =
    block_entry_node->re_block_entry();
  rbe->flags.file_default = true;
...
```


token) is inserted. The relevant C++ class offers a variety of insertion and related graph-modification routines, without paying attention to the rationale for choosing each particular kind of modification at each particular moment. The middle "Graph Build" layer then receives calls from the parser about rule-matches and has a range of graph-modification routines it can select to handle each match (often based on contextual data tracked at the middle layer). So the first layer is reponsible for generating fine-grained notifications about input patterns to the second layer, while the third layer is reponsible for presenting a flexible set of operations from which the second layer selects the proper graph-modification for each match and context.

While this kind of compiler architecture is not intrinsically dependent on hypergraph forms at the output level, using hypergraphs augments the benefits in how such a pipeline is organized. Hypergraphs, compared to other kinds of parser output (like Abstract Syntax Trees or Directed Acyclic Graphs), have fewer structural requirements and restrictions and therefore offer greater leeway in how information about input text is preerved in the output structure. This accordingly offers greater flexibility in systematizing the input language's syntax and semantics. For example, consider the goal to support special built-in datatypes with special language-level syntax, such as embedding XML directly in source code, or "sugaring" compound expressions within concise and readable forms (along the lines of how LINQ — Language Integrated Query — packages complex method-calls into a fluid syntax resembling SQL). In a context-sensitive hypergraph grammar such language features can be realized by defining a parsing context for the non-stanard syntax and defining special node-types, edge-labels, or nested graphs to hold data parsed from input when the special contexts are in effect. The resuting data, which may have complex internal structure, can be inserte into the overall graph as a single node and/or edge, taking abvantage of hypergraphs' capability to work with different scales of organization. Languages designed against the backdrop of this kind of parsing engine can flexibly evolve to support new syntactic special cases and constructs, in part because a hypergraph output format is a basically free-form tableau for stashing details and contexts observed in input text.

With that said, the hypergraphs built directly from rule-matches will still closely reflect the input text and may need substantial processing to identify the proper information and semantics embedded in the input. Insofar as the input is script-like computer source code, this semantic interpretation includes details such as assigning data types to input tokens and mapping script-level function calls to binary structures that can be used at runtime to effectuate corresponding C or C++ calls. The requisite semantic analysis depends on traversing and drawing information from hypergraphs produced via the parsing stages, but those graphs may need to be restructured so as to make the interpretive analysis — rather than the initial parsing — most convenient. Such intergraph transforms and analytics lie outside the context of parsing itself, so I will discuss these next.

Hypergraph Intermediate Representation In the demo code, the graphs produced during the parsing phase I just outlined are subject to several rounds of modification. The first stages of this follow-up are performed in the context of the initial hypergraphs, and invove changes that can be tacked on to those graphs without

substantial ateration of their structure. For instance, many source-code tokens can be assigned a provisional type just by examining the token's character content (e.g., numeric literals start with a decimal digit, and string literals are enclosed in quotes). Also, certain code segments can be necessarily associated with function calls and can therefore be pre-emptively accommodated with calls to some internal functions even at this early stage. For example, the demo language implemented in the data set has a grammar feature where a token preceded (without space) by a comma — as in the (type-declaration) line ,fnd ::: Fn_Doc*; — is usually the introduction of a new lexically-scoped symbol, so it can trigger a pre-emptive call to a method that keeps track of source-code scopes (the quoted line also implies that the symbol Fn_Doc designates a type, so a function may be call registering this as a known type).

Listing 3: Hypergraph Intermediate Representation

```
(pgb::make_root_node :|-!>last_block_pre_entry_node|)
(pgb::make_token_node :|>_@&prn| :|-->entry-node|)
(pgb::add_block_entry_node :|<!>last_block_pre_entry_node|
   :|<->entry-node|)
...
(pgb::make_channel_fuxe_entry_node :|/--result| :|/--u4|
   :|-->cfx-node|)
(pgb::make_token_node :|>_$#+| :|-->entry-node|)
...
(pgb::add_channel_entry_token :|<->channel-seq| :|<$>lambda|
   :|>_$78| :|-->channel-seq|)
(pgb::add_channel_token :|<->channel-seq| :|>_$12|
   :|-->channel-seq|)
(pgb::copy_value :|<!>last_expression_entry_node|
   :|-->channel-seq|)
(pgb::add_channel_continue_token :|<->channel-seq| :|>_$3|
   :|-->channel-seq|)
```

Once this provisional anaysis has run its course, the system then undertakes a more extreme reworking where the original hypergraph is morphed into a new graph with a different vocabulary and structural conventions. Where as the earlier graph was built to prioritize integration with a hypergraph parser, the new graph will be built to prioritize integration with a language runtime. In the demo, the first graph produces the second by emitting a special kind of graph-serialization file. This indirection is useful for testing, development, and exposition because the intermediate file can be studied as a (relatively human-readable) artifact in its own right. It also allows me to review the general subject of a hypergraph serialization (or initialization) format, which will be the focus of the next few paragraphs.

One way to conceptualize hypergraph serialization is to treat the graphs as built up from an empty graph via a finite series of steps. A serialization format can then be designed on the premise of expressing each construction-step as a distinct instruction, and then serializing a whole graph as the sequence of all its implied steps. Listing 3 samples a file in the demo's format, which is based on Common Lisp to leave open the possibility of these files being read by a Common Lisp evaluator for pre-processing. Here though the intermediate language is parsed directly in C++ — unlike the parsers discussed earlier, which aim to work with flexible scripting languages, an intermediate hypergraph representation can be narrowly targeted to machine-generated hypergraph stages and needs only simplistic parsers. For this language, each form (adopting the term from Lisp's name for a single expression) generally correspons to a single C++ method. For development, the steps described via intermediate code can be analogously implemented as C++ code directly (the demo includes project files for running an example of that direct construction in lieu of intermediate files). Accordingly, designing a serialization format for hypergraphs can be closely tied to the design of a (say, C++) library for building graphs via a set of primitive operations.

The demo format employs a notion of "multigraph tokens", which are string representations of nodes' values along with a set of token kinds. An encoding system allows the kind to be notated via a signal embedded in the token string, yielding a node-reprensentation which can work across different varieties of hypergraphs (facilitating graphs' inter-translation). Listing **??** shows some code involved with routing input text to graph-build operations. Unlike the earlier parse strategy, the simple input language does not call for complex contextual logic to determine the correct operation; instead, the requisite operation is named explicitly in the input, so the parsing callback mostly reduces to a long dispatch table. Care does have to be taken, however, to correctly anticipate which kinds of tokens are expected on the input in light of the graph-manipulation method that will be called when the tokens are parsed (some tokens hold raw data; other refer to preexisting nodes that are given names in

Listing 4: Running Hypergraph IR

```
void PGB_IR_Run::run_line(QString fn, QMultiMap<MG_Token_Kinds,
  QPair<MG_Token, int>>& mgtm)
{
 PGB_Methods md = parse_pgb_method(fn);

 switch (md)
 {
 case PGB_Methods::make_root_node:
  {
   caon_ptr<PHR_Graph_Node>* tr = get_target(mgtm);
   if(tr)
     *tr = graph_build_.make_root_node();
  };
  break;
...
 case PGB_Methods::signature:
  {
   caon_ptr<PHR_Graph_Node> n = get_arg(mgtm);
   if(caon_ptr<PHR_Graph_Signature> s = n->phr_graph_signature())
   {
    QList<MG_Token> mgts = get_signature_tokens(mgtm);
    s->add_tokens(mgts);
   }
  };
  break;
...
 case PGB_Methods::make_signature_node:
  {
   caon_ptr<PHR_Graph_Node>* tr = get_target(mgtm);
   caon_ptr<PHR_Graph_Node> n = get_arg(mgtm);
   if(tr)
     *tr = graph_build_.make_signature_node(n);
   else
     graph_build_.make_signature_node(n);
  };
  break;
...
 case PGB_Methods::push_expression_entry:
  graph_build_.push_expression_entry();
...
 }
}

void PGB_IR_Run::run_line(QString line)
{
 QMultiMap<MG_Token_Kinds, QPair<MG_Token, int>> mgtm;
 QString fn = PGB_IR_Build::parse_line(line, mgtm);
 run_line(fn, mgtm);
}
```

the input text).

The intermediate language designed via thexe techniques is effective from the point of view of machine-generating code to construct a new hypergraph from some prior data (including other hypergraphs). Rather than having components generate the intermediate code directly, the demo library includes a C++ class which builds the IR via a method interface. Listing 5 shows code using this interface, treating the IR output as a vector of (Lisp-style) forms and inserting forms at the end (or occasionally in the middle) of the vector (the methods for adding a form mostly have the same names as the methods which the forms themselves cause to be called, so programming via the IR intermediary feels like programming with the target graph builder directly).

In this section I discussed the first hypergraph — built via input from a scripting language — as an "output" graph; but more precisely the digital output of the first several processing stages is a text file in the intermediate language I have summarized. This discussion therefore follows the pipeline up until the second "generation" of hypergraph is initialized, from the IR built off of the first graph. Explaining this second graph further, and the remaining steps toward a working language runtime engine, requires a discussion of *channels* and so will be picked up in the next section.

I'll remark to conclude the current section that I believe the kinds of components and workflow outlined here — not only the graph libraries but the supporting tools and code-generators — are indicative of the sorts of infrastructural elements that would have to be standardized for a common hypergraph ecosystem modeled on the Semantic Web, for example. Replacing RDF with an explicit hypergraph paradigm is not only a project of writing code libraries that structure data in a standardized hypergraph format; it is also a matter of complementing such libraries with additional components that send data into and pull data from hypergraphs via standardized languages, as well as via a standardized interface to connect hypergraphs with parsers targeting more general-purpose languages. While this is more code that has to be written (and agreed upon, if we're envisioning a communally-adopted platform), it also means that standardizing the supporting formats is a reasonable proxy for standardizing the hypergraph models themselves. After all, graph markup languages and similar tools currently exist; the limitation is they do not properly enable the full spectrum of structures and data that would characterize a genuine hypergraph ecosystem. But, consequently, the formal contrast between fully general hypergraph models and the limited features currently supported by (e.g.) graph serialization languages can be explicated by enumerating — via a new hypergraph serialization protocol — all the desired constructions in the more general paradigm.

3 Channels and Edge Diversification

Graph representations have obvious applications to modeling computer code and computations. Intuitively, consider procedures as nodes; edges incoming to or outgoing from nodes then may represent the value passed into or returned from procedures. In order to be a reasonable code representation, however, this model has to capture further details via edge-annotations, grouping nodes and/or edges into higher-level units, or similar hypergraph constructions.

As I argued last section, hypergraphs are useful target for general-purpose programming language parsers because these general languages tend to have relatively complex grammars and special syntactic contexts. While programming languages in a theoretical sense may have an expected translation to Abstract Syntax Trees, real-world languages embrace syntactic forms that complicate this abstract picture. Earier I cited C#'s LINQ as an example of an intra-linguistic special syntax; other examples include embedded XML in Scala; attribute markings in C# and JAVA; template metaprogramming in C++; regular-expression syntax in languages like PERL and RUBY; and so forth. I have argued that a hypergraph (rather than graph or tree) paradigm is a more natural foundation for representing code in these practically evolving languages, which are designed around semantic expressiveness rather than an *a priori* commitment to theoretical compilation models.

As I will focus on in this section, similar comments apply to the semantics of procedures and function-calls. The "mathematical" picture of procedures as functions mapping inputs to outputs is only approximately accurate in the context of modern programming languages. One issue is that the distinction between input and output parameters is imprecise, given that input parameters can be modifiable references. A common C++ idiom, for example, is to pass a non-constant reference instead of returning relatively large data structures — mutable references are technically an input rather than output value, even if the intent is for a called procedure to supply data for the mutable input (this technique avoids extra copying of data). Another complication is that procedures can have multiple sorts of input and output: e.g., throwing an exception rather than returning a value. Likewise, Object-Oriented languages' "message receivers" are input parameters with distinguished properties as conmpared to normal inputs.

Since programming notation visually distinguishes procedure inputs from outputs, it is easy to think of the input/output distinction as mostly syntactic. The significant differences, however, are more semantic. Consider a C++ function which returns an integer: it is a compiler error to return from such a function without suppying a value; also, the called procedure has no access to the memory which would hold the return value. By contrast, a function which takes a non-constant integer reference is not forced by the compiler to modify that value, nor prohibited from accessing the input's value. This points to the practical differences between returning values and altering input references, but it also indicates that inputs can be treated as *de facto* outputs by adhering to usage conventions — a reference mimics the behavior of a return value to the extent that procedures initialize the reference on all execution paths, and refrain from using any value which the reference might have before the procedure starts. Generalizing this point, the badic contrast between input and output can be seen as a semantic matter of usage protocol, rather than a syntactic contrast between passing valued to a function vs. accessing a function's returned result.

Of course, input and output are usually both distinguished and interconnected by how outputs are used as inputs to other functions. With respect to procedures which take one parameter, the general function-composition operator $\mathfrak{f}\circ\mathfrak{g}$ takes the output of $\mathfrak{g}$ as the input to $\mathfrak{f}$. We could of course define a different operator where $\mathfrak{g}$ and then $\mathfrak{f}$ both run, taking the same value (potentially a reference-type value that $\mathfrak{g}$ could potentially modify before $\mathfrak{f}$ runs). The former (composition) operator establishes an inevitable dependence of $\mathfrak{f}$ on $\mathfrak{g}$: $\mathfrak{f}$ needs $\mathfrak{g}$'s output for an input value. The second case is less obvious — in a sequential call like $\mathfrak{f};\mathfrak{g}$ ($\mathfrak{g}$ runs then $\mathfrak{f}$ runs — I notate the relation with an inverted semi-colon to highlight the parallels with $\mathfrak{f}\circ\mathfrak{g}$, but the analogous case in practice would be $\mathfrak{g};\mathfrak{f}$), there may or may not be a $\mathfrak{g}$ side-effect that infuences $\mathfrak{f}$ (even if they are passed the same inputs). So inputs and outputs are distinguished by how output-to-input links — outputs from one procedure becoming inputs to another — have certain semantic properties in the context of functional composition, and by extension the relations between precedures which run in sequence. Again, though, this call for a semantic analysis which looks beyond languages' surface syntax

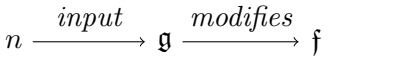

Figure 2: figGF

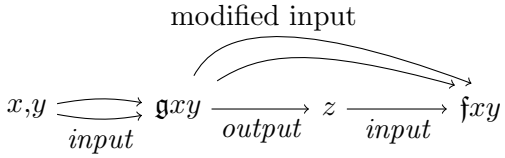

Figure 3: figGFComplex

for function calls, and how usages of return valued are notated.

Mutable references and procedures-with-side-effects expose limits in the representational expressivity of graphs (even hypergraphs) for modelling inter-procedure relationships. Consider a case such as represented in Figure 4 , where $\mathfrak{g}$ modifies input n that is then also input to $\mathfrak{f}$. Here $\mathfrak{g}$'s effect on $\mathfrak{f}$'s input is marked by a $\mathfrak{g}$-to-$\mathfrak{f}$ arrow, but this directed edge implies a different semantic relation than $\mathfrak{g}$'s output being $\mathfrak{f}$'s input (as in $\mathfrak{f}\circ\mathfrak{g}$). The specific difference can be visualized in a scenario like Figure **??**, based on code such as written below the graph. The situation here is that $\mathfrak{f}$ both uses $\mathfrak{g}$'s output return (via the symbol z the value is assigned to) and also the values held by x and y which are altered by $\mathfrak{g}$. I notate these dependencies by grouping the $\mathfrak{g}$-to-$\mathfrak{f}$ edges through x and y separately from the edge with z — the latter edge conforms to the relation of a procedure taking another procedure's output as input (although here the relation is indirect through a lexically-scoped symbol), but the former edges reflect a different inter-procedure pattern. The point of this example is that separating input and output edges is not enough to express program-flow semantics in anything but pure functional languages — even given the "hypergraph categories" strategy of grouping input an output values into hypernodes.

For situations like these I propose a notion of *channels*, which are higher-scale groupings of edges analogous to how hyperedges involve higher-scale groupings of nodes. In Figure **??**, for instance, the x and y edges can be considered a separate channel than the z edge, because x and y bridge $\mathfrak{f}$ and $\mathfrak{g}$ according to one sort of program-flow semantics (with side effects and mutable inputs passed to sequential procedures) while z follows a different pattern (with output values and lexical symbols). The shaded box in the figure shows the core role of channels, grouping together edges that have some particular connection which could not be expressly identified by other means (like shared annotations) and also isolating edges and edge-groups from other elements (here, isolating the x-and-y pairing from the z edges).

This case illustrates the basic premise of channels as distinguished and/or edge-groupings, and also one modelling domain where I contend channels can be applied; specifically, representing details in computer code (in a general hypergraph representation of source code) such as program flow. To discuss channels as graph elements further, I will start by examining channels in the specific computer-code context, which calls for a detour review of certain semantic structures central to the semantic of computer programs.

Typed Values and "Careers" It is commonly accepted that a theory of types — with at least some details adopted from mathematical type theory — is a necessary component of a rigorous treatment of computer software and programming languages. However, we should not neglect to keep in mind the specific realities of programming environments that complicate such basic notions as types' instances, the nature of entities that we should deem to be bearers of types, the status of special (non-)values like `null` or "Not A Number", and related details that contrast programming "types" from mathematical sets or "spaces" in various incarnations (sheaves, topoi, etc.).

In particular, the "instances" of types (at least in the programming-language context) are not particularly suited to the intuitions of "members" of sets, or "points" of spaces. One reason is that there is no fixed theory of types' extension, or "set" of possible instances, in a given computing enbironment. For example, types without a fixed size in memory — say, lists of numbers — are constrained be a computer's available memory. We cannot be certain which lists are small enough that a given environment can hold them in memory, so that they are available as typed values for a program. Conceptually, then, types are very different from sets of values.

More significantly, types' instances do not have a kind of abstract self-sufficiency like points in a mathematical space. Consider how computational procedures deal with "values", the things that are representatives of types.

Most values which procedures encounter are produced by other procedures, either passed as parameters from calling procedures to called procedures, or returned as outputs from called procedures to calling procedures.

Of course, programs execute in a world of surrounding data — on computer drives and in memory, plus "cyberphysical" data from devices hooked up to computers (at a minimum, devices crafted to be part of a computer set-up, like mice and keypads; plus often "smart" devices which software accesses to manage an oversee). From a programmer's point of view, this data is accessed by calling specific functions, albeit ones which (at the lowest level) programmers themselves do not implement (viz., the procedures exposed in device "drivers"). Analogously, many values are presented directy in source-code literals; but we should think of these values as likewise yielded from procedures that read source tokens or strings and interpret the intended type-spefic value (e.g., converting the character-string "123" to the corresponding integer). Typically these "reader" procedures occur behind-the-scenes; although, which helps demonstrate the point that source literals involve implicit function-calls, some languages support "user-defined" literals with nonstandard syntax — consider Lisp "reader macros" and C++'s `operator""`.

Consequently, analysis of programs and computer code should proceed from the perspective that all typed values are introduced into a running program as results of procedure-calls. We therefore have extra detailing in any context where we would talk about "typed values". The situation could be analogously compared to settings like approximation theory — consider an analysis according to which a spatial magnitude, say, is not just a real number but the result of a measurement operation. By extension, quantities that on purely mathematical terms might be treated as points in spaces such as the real number line are "packed" into *observations* which can be sites carrying further information, such as an approximation factor. My point is not that computer programs fail to work with exact values — in a typical case a single typed value does not need an approximation "window" unless the software is manifestly interacting with physical systems where measurement uncertainty is taken into consideration. However, approximation as "extra structure" is a useful analogy for the added detail latent in programming theory given that typed values are not "freestanding" mathematical entities but are always packaged into digital structures that make them available for software procedures.

For sake of discussion, I will use the term *carriers* to describe the structures that "package" typed values. When a procedure returns a value, said value is embedded in a carrier which "transports" it to other procedures. The "things" exchanged between proceures are actually, then, carriers rather than values — we should picture carriers travelling on "routes" laid out in source-code hypergraphs. Carriers, in particular, are more general than values. Consider a function which, on one occasion, throws an exception rather than returning a value. The carrier arranged to hold the return value is therefore empty. Carriers, then, can be in a *state* other than that of holing a valid, initialized value. Some programming language accommodate these cases via special `null` or `Nothing` values, but these are still constructs inside the language's type system. Thinking in terms of carriers, instead, removes certain "states" from the type system: an empty or pre-initialized carrier does not have some special "empty" value; it does not have a value at all. Similar comments apply to carriers which hold values which were once valid but are no longer so, like deleted pointers. Any carrier potentially has a "career" which starts with no value then is initialized with a value (and maybe gets updated over time); and may then "retire" and cease to hold a legitimate value. Only in the middle phase of this trajectory does the carrier's range of possible states correspond with potential instantiations of a type (I assume here that each carrier is associated with one canonical type).

Carriers and Categories of Types . In some contexts it is useful to examine a type system $\mathbb{T}$ via a category $\mathcal{T}$ of types (such as, the category $\mathfrak{H}$ of Haskell types). Carriers are an extra structure which rest "on top of" types, calling for the foundation (including Category-theoretic treatment if applicable) to be reconsidered. First, the Haskell-like interpretation of $\mathcal{T}$ morphsims should be redefined. On the usual account, $\mathfrak{t}_1 \to \mathfrak{t}_2$ morphisms are functions mapping type $\mathfrak{t}_1$ to $\mathfrak{t}_2$. This picture is complicated by procedures deviating from pure functions: as I discussed earlier in this section, the input/output discussion is complex and does not precisely map to the mathematical model of morphisms between Category-objects. Instead, inter-type morphisms (to the degree that these are idetified as categorial morphisms establishing type systems as categories) should be restricted to

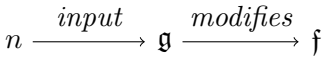

Figure 4: figGF

specialized semantic associations between types, such as casts from integers to floating-point values (hence, morphisms between integer types to floating-point types).

Moreover, reasonable type systems have a nontrivial collection of endofunctors marking inter-type associations. A good example is the "unreferencing" operator which recovers the underlying type $\mathfrak{t}$ from a relerence type $\mathfrak{t}\&$. This operator naturally translates to a function $\mathfrak{t}\& \to \mathfrak{t}$ that retrieves the value held by a reference. It is reasonable to treat this function as a $\mathfrak{t}\& \to \mathfrak{t}$ morphism.

In a general case, though, implemented procedures should be treated apart from such basic morphisms that tie a type sytem together: a general procedure which takes a $\mathfrak{t}_1$ input and outputs $\mathfrak{t}_2$ is not a $\mathfrak{t}_1 \to \mathfrak{t}_2$ morphism but a more complex structure operating on carriers. For each input and output parameter there is a carrier accessible in the environment of the procedure (its program code and its runtime environment) that provides the procedure acess to an underlying value. The specific access can be modeled around different sorts of carriers — carriers holding return values should allow procedures to "write upon" (initialize or modify) their values but not read them (conformant to the expected semantics of "procedural output"). In this theory, the output carrier from one procedure is distinguished from the input carrier for another procedure, even in a case where one's output is directly used as the other's input. To illustrate, Figure **??** shows a modified version of Figure 4 where the $\mathfrak{g}$-to-$\mathfrak{f}$ edge is annotated by a pair of carriers (output and input differentiated by solid vs. double borders). Similar to Link Grammar (see the last section), a single edge bears a double-annotation to indicate that the edge depends on some notion of compatibility: in particular, output-to-input carriers can only be linked if they hold the same type (or at least interconvertible types). As such, a double-annotated edge captures a situation where the value in one carrier is transferred into a second carrier — potentially with a type-cast or, e.g. in C++, a behind-the-scenes call to a copy-constructor. I will refer to such a value-transfer as a *handoff*. A $\mathfrak{g}$-to-$\mathfrak{f}$ (hyper-)edge therefore expresses added details because its annotations convey properties of carriers whose state and compatibility determine the value-transfers that enable inter-procedure relations, such as one procedure calling and/or using the output from a different procedure.

In lieu of mostly theorizing type categories, then, we can also consider a category $\mathcal{CS}$ of carrier *states*, which generalize typed values. A morphism in $\mathcal{CS}$ represents either a change in state or a mapping between values. When a carrier $\mathfrak{c}$ is initialized, it transitions from being in a state with no value to being in a state where it holds an instance of its type (when a carrier is already initialized its state can change by repacing its held value with another value; or by becoming "retired"). A $\mathfrak{c}_1$-to-$\mathfrak{c}_2$ "handoff" then corresponds to a morphism in $\mathfrak{c}_2$'s state wherein $\mathfrak{c}_2$ (perhaps transitions out out a pre-initialized state and) acauires a concrete value derived from $\mathfrak{c}_1$'s. This case — I'll call it a "handoff-induced morphism" — plays a foundational role in analysis via carriers and hypergraphs analogous to $\mathfrak{t}_1 \to \mathfrak{t}_2$ morphisms in conventional category-theoretic program semantics.

After accounting for this change in perspective, many concepts of hypergraph categories appear to carry over to this theory. Rather than morphisms in a type category $\mathcal{T}$, (hyper-)nodes in this case represent procedures, and their surrounding edges represent (via annotations) carriers accessible (as inputs or outputs) to the procedure's implementation. Morphisms are represented "within" these edges insofar as a procedure being called induces a change in state for carriers along the procedure's incoming edges (likewise, returning values induces state-changes among outgoing edges).

In practice, procedures call other procedures. A procedure's input carriers can be augmented by a "lexical scope"; a repository of local carriers (syntactically speaking, lexically-scoped source-code symbols) which start as uninitialized but can hold outputs from other procedures called from the current procedure's implementation body. Each implementation is then a sequence of further procedure call wherein the calling procedure assembles the requisite local carriers to provide inputs to the called procedure (via handoffs). The calling procedure must also provision carriers to hold the called procedure's output. In effect, the calling procedure must orchestrate a suite of handoffs among its own carriers and the those of the called procedure. The semantics of this multi-faceted sum of carrier-transfers can be represented

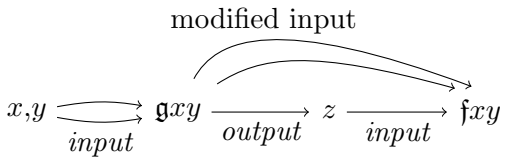

Figure 5: figGFComplexc

within a source-code graph; and so, the structures and semantics of these hypergrah representations should be codified in a manner which renders (a de-facto Ontology) the patterns of aggregate carrier handoffs).

To illustrate, Figure **??**, modifies Figure **??** so as to show carriers and handoffs, analogous to Figure **??** adding carriers to Figure 4. Each case where one procedure calls another can be modelled via a graph which groups together the carriers (seen in the context of the calling procedure) that are packaged together to enable the call; and concordantly the links between these carriers and those in the called procedure's signature and implementation which embody handoffs executed in the transition in program flow between the calling procedure to the called procedure, and back.

Carriers and Channels The theory of channels comes into effect insofar as we combine this carriers and handoffs model with my earlier discussion of differentiating input and output varieties. For instance, inputs may be Object-Oriented message-receivers or normal parameters; outputs may be thrown exceptions or normal returns. In the hypergraph framework I just outlined, hyperedges carry double-annotations to mark the pairs of carriers involved in a handoff within a procedure-call. Grouping carriers based on their common semantics — separating the this object from non-distinguished parameters, let's say — involves a further level of organization, grouping hypereges into channels. Taken together, then, the overall hypergraph system exhibits two generalizations from simpler code-graphs: double-annotations on edges (based on carrier theory) and the use of channels to reflect higher-level programming semantic, such as the distinct status of method-receivers as non-standard input and of exceptions as non-standard output.[5]

To develop this unified theory, we can consider a *channel* as (semantically) a (possibly empty) collection of (one or more, if not empty) carriers — the corresponding hypergraph representation is a channel grouping edges whose annotations refer to these carriers. Each channel is then associated with a specifis "protocol", an idiosyncratic method for procedures to interact with carriers (and by extension with one another). For example, the procol for regular input and output channels is that one output carrier's value can be directly "handed off to" an input carrier, as in $\mathfrak{f}\circ\mathfrak{g}$-like composition. In many languages a procedure can have at most one output value, and also at most one "message receiver" (this) object. Accordingly, a procedure-call can bundle the output channels from multiple other procedures to populate some further called procedure's input parameters. Since there is only one " this ", two methods can be combined (modulo type compatibility) by inserting the prior method's output as the next method's message-receiver (handing off the output value to the next this carrier), giving rise to "method chaining" (such call-sequences can continue over multiple steps). Meanwhile *exceptions* cannot be treated as normal values (you cannot in most cases hand off between an "exception channel" and normal carriers). Instead, procedures which want to "catch" exceptions need special structures, like catch-blocks, which provide the atypical carriers that can actually hold exception values (and so receive exception-channel handoffs.

All of these are customary coding structures and design patterns which regulate how modern programming languages handle features like method-calls and exceptions. These patterns, which I am theorizing here according to channels' semantic protocols, are not primarily grounded in a language's type system but rather in regulations on procedural flow and program organization that are orthogonal to type restrinctions. I contend that a theory of channels can unify the regulations enforced by strong typing (e.g. not calling procedures with incompatible arguments vis-à-vis expected types) and those due to inter-channel semantic protocols. What the two semantic regimes have in common is that they ultimately maniest as rules on carrier-transers — both carriers' types and the kinds of channels to which they respectively belong influence whether a purported handoff (needed by a call between procedures) is permissible; if not, the code declaring the relevant procedure-call should not be accepted (it should not compile). So, enumerating channel-semantic protocols is a way to formally model

[5] My chapter in [**?**] includes examples of other more "exotic" kinds of channels as well.

what inter-procedure relations are legal accoring to the program-flow and organizational principles of the implementation language, analogous to how procedure calls are evaluated for compliance with the language's type system. The syntactic rules for how procedure-calls are notated are driven in part by inter-channel protocols (for example, using a dot to separate a `this` object from a method-name allows method-chains to be compactly written), but a rigorous model should take syntactic details as epiphenomena — the important details for hypergraph source-code models is the kinds of channels recognized by a language for inter-procedure communication, and the protocols associated with various channel-kind pairs governing how and when carriers in one channel may be linked by handoffs to carriers on other channels (of the same or other kinds).

One benefit of the channel-based framework I have presented here is that it provides an intuitive breakdown of procedure-calls into smaller steps. A procedure call generally requires a structure involving multiple carriers to be present in the context of the calling procedure (while, on the called side, the procedure's signature provisions a set of carriers that receive values from the calling context). Preparation for the call therefore entails building up a set of carriers and establishing their respective links to carriers on the receiving end. It is possible to divide this process into a series of minimal sets, each involving operations on a single carrier. Via this interpretation, graphs modeling computer code with channels an carriers can be translated into sequences of minimal carrier-related instructions, which in turn can be executed as a kind of virtual machine. The scripting environment I am using in this paper for demonstration — which in the last chapter I covered mostly from a grammar and parsing angle — therefore culminates in data structures constituting an Intermediate Representation used by a runtime engine to translate the compiled code to internal (mostly C++) function-calls. This involves two software components — an algorithm to walk through code graphs to generate Virtual Machine code and then a runtime library to execute the code thereby generated. I will briefly summarize both components as case-studies in how hypergraph code representations can be transformd into concrete execution environments.

A Hypergraph-Based Virtual Machine According to channel theory, an operational execution environment is, first an foremost, a "virtual machine" which defines and operates on carriers, packages them into channels, and then uses bundles of channels as specifications for procedure-calls. When *creating* virtual machine code, we can then proceed by stepping through graph elements representing channels and carrier, emitting data as needed to reconstruct channels via minimal operations. When *running* virtual machine code, the important step is packaging groups of channels into a single data structure (for each distinct procedure call) and then mapping the resulting structure to a binary array which mimics the ABI (Application Binary Interface) of target procedures — so as to produce function-calls via the virtual machine which, at the binary level, appear to be the result of compiled (e.g., C++) functions.

Listing 6 depicts part of a virtual machine file generated by the demo from a script-like source file. This kind of file (which I call PVMIR, for "Phaon Virtual Machine") is the final output of the compiler pipeline described in this and the last section. The basic premise behind this format is that every procedure call can be associated with a *channel package*, each of whose channels is a stack of carriers. It is straightforward to generate PVMIR code tracing a series of instructions to recreate a channel package held as a structure (in the demo, a C++ object) in memory. Moreover, the "Phaon Graph" format is designed to represent information associated with channels and carriers; so generating PVMIR (at least on the level of individual procedure calls) is facilitated by constructing graphs in the "Phaon" format to begin with. I use the term "RelaeGraph" for graphs generated from "relae" parsers; in the overall pipeline then the transformation from Relae-Graphs to Phaon-Graphs is the important step which enables the PVMIR generator to run mechanically.

As desribed, PVMIR plays a limited role insofar as it is generated from channel-package objects — yielding PVMIR files — and then sometime later a virtual machine runs the PVMIR code to recreate those objects at runtime. In this role PVMIR just allows the channel-package objects to be encoded in a text file for later use. However, PVMIR is more than only a serialization format for channel packages. The extra roles for PVMIR come into play when defining source-code-level procedures (which includes code blocks, e.g. `if−then−else` alternatives, which are implemented as anonymous procedures). Because PVMIR represents a seauence of minimal

carrier-related operations, certain such sequences may be isolated as separate subroutines, which in turn can be assigned unique-identifier and called within the virtual-machine runtime as "virtual" functions.

The PVMIR format relies on the fact that each minimal virtual-machine operation has a similar form — actually every instruction either takes no arguments or a single string argument. As such the input text can be compiled into an array of pairs comprised of a pointer-to-member-function and a (possibly empty) argument (see Listing 7). The code at (1) converts a string name to a method-pointer and appends to an instruction list indexed by the name of the procedure currently being compiled; while (2) shows the analogous runtime logic that loads a compiled instruction-list, which may then be executed as a called procedure. In this milieu the channel packages whose structure laid out the setup for a procedure-call here become itemized into the creation of a local, stack-driven context where the called procedure can finally be realized, as a virtual-machine subroutine.

The final step for a Phaon Virtual Machine is to "reduce" channel packages to emulate the normal C++ ABI. The PhaonGraph system is intended to support "exotic" channel packages, with channel-semantic protocols that depart from the basic inputs and returns (and also `this` objects and exceptions) from mainstream programming langages. The specialized protocols can provide a more sophisticated basis for static or runtime analysis of Phaon-structured source code representations. However, a Phaon environment eventually needs to defer to C++ functions to provide underlying functionality, which means that packages based on exotic protocols must be mapped to simpler packages shaped around C++: at most one `this` and one return value; a block to catch exceptions if necessary; and any remaining parameters in a conventional input channel. Mimicking a C++ ABI also requires that the byte-length of input and output parameters is anticipated, based on the signature of a called C++ function.

So far I have talked about channel *packages* which are structured aroud points in source-code where procedures are called. The PVM system has a parallel notion of channel *bundles* which are associated with functions' signatures — at runtime, PVM can access C++ functions which are described ahead of time in terms of channel-bundles.[6] When a PVM instance starts up, the predefined bundles are analyzed to produce a record of functions' signatures from the perspective of the C++ ABI. The chief task of this analysis is to sort C++ functions into equivalence classes, where two functions are equivalent when they can be called via identical function-pointer and argument-array binary layouts. In effect, functions are equivalent if their argument lists are equally-sized arrays of parameters which are pairwise equal in byte-length. In such a case the two functions may be jointly represented by `void*` function-pointers and called from the same site in the PVM engine. The low-level details are sampled in Listing 7: here (1) marks a point where the engine switches over numeric codes standing for certain equivalence classes; and (2) shows how multiple target functions, once the equivalnce class is identified (and the argument array initialized accordingly), may be accessed via a single function-pointer.

Further details about reducing channel-packages around the C++ ABI are not directly concerned with graph representations and so are outside the scope of this paper, so with this overview I will wrap up my enumeration of stages in the PVM pipeline. I will comment, however, that the PVM engine's binary manipulation of function-pointers and argument-arrays enables the runtime to work with complex structures derived from non-stanard channel packages. The end result is that PVM provides an execution engine that works with sophisticated hypergraph representations of source code: whereas a more conventional architecture might assume that compilers will eventually reduce source input to relatively simple AST-style data, PVM assumes that the Virtual Machine runtime is operating on instruction-sequences that preserve the more complex, hypergraph-driven code-representation semantics. In this sense PVM suggests the kind of compiler and Virtual Machine architecture that could be appropriate in general for the overall project of standardizing hypergraph code models and leveraging them directly in a runtime execution engine.

[6] Separately, PVM can also call Qt-specific methods according to Qt's internal reflection mechanism, the Meta-Object Protocol.

4 Conclusion

My analysis in this paper has focused on four themes: converging on a general-purpose hypergraph model via a standized hypergraph serialization format at an Intermediate Representation level; generalizing hypergraphs via node, edge, and annotation diversification — in particular *procedural* hypegraphs whose hypernodes can have a data-collections interface, along with edge-diversification into channels (especially in the context of code-representation) and annotation-diversification into double-annotated edges representing carrier-transfers; hypergraph parsers to initialize hypergraphs with these extra features from source code; and strategies for implementing virtual machines that can offer an execution environment for the resulting hypergraphs.

Listing 5: Hypergraph IR Builder

```
void RPI_Stage_Form::write_unmediated(QTextStream& qts,
  caon_ptr<RPI_Stage_Form> prior)
{
...
 RPI_Assignment_Info* rai = get_parent_assignmnt_info();
 RPI_Stage_Element_Kinds last_kind =
   RPI_Stage_Element_Kinds::N_A;
 caon_ptr<RPI_Stage_Form> prior_form = nullptr;
 for(RPI_Stage_Element& rse : inner_elements_)
 {
  QString rset = rse.text();
  switch (rse.kind())
  {
...
  case RPI_Stage_Element_Kinds::S1_FGround_Symbol:
  case RPI_Stage_Element_Kinds::FGround_Symbol:
   if(rset.startsWith('#'))
     pgb_(step_forms_).make_token_node(rset.prepend('$'),
     "&fsym-node") = Purpose_Codes::Make_Token_Node_FSym;
   else
     pgb_(step_forms_).make_token_node(rset.prepend('@'),
     "&fsym-node") = Purpose_Codes::Make_Token_Node_FSym;
   if( flags.is_block_entry_statment
      || flags.is_inferred_block_entry_statment )
   {
    if(rai)
    {
     pgb_(step_forms_).make_statement_info_node(
       rai->text().prepend('@'), ":result",
       assignment_info_.encode_ikind().prepend(':'),
       "&si-node");
     pgb_(step_forms_).add_block_entry_node(
       "!last_block_pre_entry_node",
       "&fsym-node", "!last_block_entry_node", "&si-node");
    }
    else
      pgb_(step_forms_).add_block_entry_node(
      "!last_block_pre_entry_node",
      "&fsym-node", "!last_block_entry_node");
   }
...
   pgb_(step_forms_).copy_value("&fsym-node",
     "!last_expression_entry_node");
   pgb_(step_forms_).copy_value("&fsym-node", "&channel-seq");
   break;
...
  case RPI_Stage_Element_Kinds::Form:
   {
    caon_ptr<RPI_Stage_Form> f = rse.form();
    if(f->instruction("kb::write-anon-fdef"))
    {
     f->write_unmediated(qts, nullptr);
     if(!f->step_forms().isEmpty())
     {
      pgb_(step_forms_).make_block_info_node("&bin");
...
      pgb_(step_forms_).push_block_entry();
      step_forms_.append(f->step_forms());
      pgb_(step_forms_).pop_block_entry();
     }
    }
    else
    {
     f->write_unmediated(qts, nullptr);
...
    }
    prior_form = f;
   }
   break;
  default:
   continue; // skip last_kind assignment ...
  }
  last_kind = rse.kind();
 }
...
}
```

Listing 6: Example PVMIR Code

```
 .; generate_from_fn_node ;.
push_carrier_stack $ fuxe ;.
hold_type_by_name $ fbase ;.
push_carrier_symbol $ &prn ;.
 .; args ;.
push_carrier_stack $ lambda ;.

push_unwind_scope $ 1 result ;.

 .; unwind_scope: 1 ;.

 .; generate_from_fn_node ;.
push_carrier_stack $ fuxe ;.
hold_type_by_name $ fbase ;.
push_carrier_raw_value $ #+ ;.
 .; args ;.
push_carrier_stack $ lambda ;.
hold_type_by_name $ u4 ;.
push_carrier_raw_value $ 3 ;.

push_unwind_scope $ 1 result ;.
```

Listing 7: PVMIR to Member-Function Pointers

```
void PhaonIR::read_line(QString inst, QString arg)
{
 static QMap<QString, void(PhaonIR::*)(QString)> static_map {{
  { "push_carrier_stack", &push_carrier_stack },
  { "hold_type_by_name", &hold_type_by_name },
  { "anchor_without_channel_group", &anchor_without_channel_group },
  { "finalize_signature", &finalize_signature },
  { "push_carrier_symbol", &push_carrier_symbol },
  { "push_carrier_stack", &push_carrier_stack },
  { "push_carrier_raw_value", &push_carrier_raw_value },
...
 }};

 auto it = static_map.find(inst);
 if(it != static_map.end())
 {
  line_ops_[current_source_fn_name_].push_back(
    {new QString(arg), fn_u{.fn1=it.value()}});          (2)
 }
}

void PhaonIR::run_lines(QString source_fn)
{
 auto it = line_ops_.find(source_fn);
 if(it == line_ops_.end())
   return;
 const QList<QPair<QString*, fn_u>>& lines = it.value();   (1)
 for(auto& pr: lines)
 {
  if(pr.first)
    (this->*(pr.second.fn1))(*pr.first);
  else
    (this->*(pr.second.fn0))();
 }
}

void PhaonIR::run_callable_value(QString source_fn)
{
 run_state_stack_.push({current_source_function_scope_,
   program_stack_, current_carrier_stack_,
   held_channel_group_, sp_map_});

 init_current_source_function_scope(source_fn);
 init_program_stack();
 current_carrier_stack_ = nullptr;
 held_channel_group_ = nullptr;
 sp_map_ = new QMap<QPair<Unwind_Scope_Index,
   PHR_Channel_Semantic_Protocol*>, PHR_Carrier_Stack*>;
 run_lines(source_fn);
...
 Run_State rs = run_state_stack_.pop();
 current_source_function_scope_ = rs._source_function_scope;
 program_stack_ = rs._program_stack;
...
}
```

Listing 8: PVM Runtime — Function Equivalence Classes

```
void PHR_Command_Runtime_Router::proceed_s0_2(PHR_Function_Vector* pfv, void** pResult)
{
 int byte_code = 0;

 s0_fn1_p_p_type fn = nullptr;
 bool sr;

 int mc = 0;
 int bc = 0;

 if(pfv)
 {
  if(void* fnp = pfv->match_against_codes({6002, 7002}, mc, bc, &result_type_object_))
  {
   fn = (s0_fn1_p_p_type) fnp;
   byte_code = bc;
   sr = mc == 6002;
  }
  else byte_code = 0;
 }
...
 if(fn)
 {
  proceed_s0<2, s0_fn1_p_p_type>(pResult, fn, byte_code, sr, false);
 }
...
}
...
template<>
void PHR_Command_Runtime_Router::proceed_s0_r<2, s0_fn1_p_p_type>(QVector<quint64>& args, void*& result,
  s0_fn1_p_p_type fn, int byte_code)
{
 switch(byte_code)
 {
 case 944: tybc_<2>::run<944, s0_fn1_p_p_type>(args, result, fn); break;
 case 984: tybc_<2>::run<984, s0_fn1_p_p_type>(args, result, fn); break;
 case 948: tybc_<2>::run<948, s0_fn1_p_p_type>(args, result, fn); break;
 default:
 case 988: tybc_<2>::run<988, s0_fn1_p_p_type>(args, result, fn); break;
 }
}
...
struct tybc_<2>
{
 template<int byc, typename fn_type>
 static void run(QVector<quint64>& args, void*& result,
  fn_type fn)
 {
  result = ( (typename tybc<byc>::_r) fn)
    ( (typename tybc<byc>::a0) *((quint64*) args[0]),
    (typename tybc<byc>::a1) *((quint64*) args[1]));
 }
}
```

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