HyperGraphDB: A Generalized Graph Database

Borislav Iordanov

Kobrix Software, Inc.
http://www.kobrix.com

Abstract. We present HyperGraphDB, a novel graph database based on generalized hypergraphs where hyperedges can contain other hyperedges. This generalization automatically reifies every entity expressed in the database thus removing many of the usual difficulties in dealing with higher-order relationships. An open two-layered architecture of the data organization yields a highly customizable system where specific domain representations can be optimized while remaining within a uniform conceptual framework. HyperGraphDB is an embedded, transactional database designed as a universal data model for highly complex, large scale knowledge representation applications such as found in artificial intelligence, bioinformatics and natural language processing.

Keywords: hypergraph, database, knowledge representation, semantic web, distributed.

1 Introduction

While never reaching widespread industry acceptance, there has been an extensive body of work on graph databases, much of it in the 90s. Various data models were proposed, frequently coupled with a complex object representation as a natural, practical application of graph storage. More recently, several developments have contributed to a renewed interest in graph databases: large-scale networks research, social networks, bioinformatics as well as the semantic web and related standards. Part of that interest is due to the massive amounts of graph-oriented data (e.g. social networks) and part of it to the inherent complexity of the information that needs to be represented (semantics and knowledge management). This body of work has been thoroughly reviewed in [1]. In this paper, we present the implementation of a generalized hypergraph model independently proposed by Harold Boley [4] and Ben Goertzel [5]. The model allows for n-ary hyperedges that can also point to other edges, and we add an extensible type tower [6] to the mix. Two generalizations of graphs related to our work have been the Hypernode model [2] and the GROOVY model [3], both focused specifically on representing objects and object schemas. While hypergraphs have been extensively used as an analytical tool in database research, we are not aware of any other implementation of general hypergraphs as a native database.

H.T. Shen et al. (Eds.): WAIM 2010 Workshops, LNCS 6185, pp. 25–36, 2010.
© Springer-Verlag Berlin Heidelberg 2010

The main contributions of HyperGraphDB[1] lies in the power of its structure-rich, reflexive data model, the dynamic schema enforced by an extensible type system, and open storage architecture allowing domain specific optimizations. It must be noted however that the representational flexibility also leads to performance gains when fully exploited. While a hypergraph can be represented as a regular graph, frequently the opposite is also true in a non-trivial way. Many graphs will have repetitive structural patterns stemming from the restrictions of the classical graph model. Such patterns can be abstracted via a hypergraph resentation, leading to much fewer nodes and database operations. For example, flow graphs where edges represent multi-way input-output connections can be stored much more compactly using a hypergraph-based model.

Furthermore, reducing the complexity of a representation is not the only benefit of a hypergraph model. As illustrated in the context of cellular networks, "transformation to a graph representation is usually possible but may imply a loss of information that can lead to wrong interpretations afterward" ([7]). In fact, biological networks are replete with multilateral relationships that find a direct expression as hypergraphs. Another example of how the ability to represent higher-order relations can improve algorithms can be found in [8], where the authors present a learning algorithm for Markov Logic Networks based on what they call "hypergraph lifting" which amounts to working on higher-order relations.

One common criticism of RDF ([9]) stores is the limited expressiveness of binary predicates, a problem solved by HyperGraphDB's n-ary relationships. Two other prominent issues are contextuality (scoping) and reification. A popular solution of the scoping problem has been proposed in the form of *Named Graphs* ([11]). Reification is represented through a standardized reification vocabulary, by transforming an RDF graph into a reified form ([10]). In this transformation a single triplet yields 4 triplets, which is unnatural, breaks algorithms relying on the original representation, and suffers from both time and space inefficiencies. A similar example comes from the Topic Maps standard [12] where reification must be explicitly added as well, albeit without the need to modify the original representation. Those and other considerations from semantic web research disappear or find natural solutions in the model implemented by HyperGraphDB.

The paper is organized as follows: first, we review some variations of hypergraphs and describe the particular one adopted by HyperGraphDB. In subsequent sections, we detail the system's architecture: storage model, typing, indexing, querying and its P2P distribution framework. Lastly, we describe in some detail one particular application in natural language processing.

2 Hypergraphs and the HyperGraphDB Model

The standard mathematical definition of a hypergraph is a family of sets over a universal set of vertices V, or equivalently an undirected graph where an edge

[1] The system is open-source software, freely available at `http://code.google.com/p/hypergraphdb`.While the current implementation is in the Java programming language, we note that the architecture is host-language-agnostic and we make very few references to Java constructs in the exposition below.

can connect any number (> 0) of vertices. Such hypergraphs are studied in the context of set combinatorics, where they are also called finite set systems [13]. In representational problems directed hypergraphs have also been proposed, where a hyperedge acquires an orientation in one of two ways: (1) partitioning it into a head and a tail set yields several interesting applications as reported in [14]; or (2) treating the edge as a tuple (ordered, with repetition) yields a very powerful representational language [16].

In basic set theory a hypergraph essentially defines an incidence structure over the universe of vertices V. Such a hypergraph is isomorphic to a bipartite graph where one set represents the hypergraph's vertices and the other its hyperedges. If one includes hyperedges in the vertex universe as well, a set theoretic formalization becomes harder because the foundation axiom no longer holds. Such hypergraphs are isomorphic to general (i.e. allowing cycles) directed graphs and they have been studied from a set-theoretic perspective by P. Azel [17]. But in a computational setting such generalized hypergraphs are a much more natural construct as they directly allow the recursive construction of logical relationships while easily avoiding dangerous circularity in a manner similar to Russel's type theory. This is the model adopted by HyperGraphDB. The representational power of higher-order n-ary relationships is the main motivation behind its development.

In the HyperGraphDB data model, the basic representational unit is called an *atom*. Each atom has an associated tuple of atoms called its *target set*. The size of the target set is called the atom's *arity*. Atoms of arity 0 are called *nodes* and atoms of arity > 0 are called *links*. The *incidence set* of an atom x is the set of atoms that have x as a member of their target set (i.e. the set of links pointing to x). The use of tuples instead of sets for an atom's target set is not the only possible choice; both can be supported, but we have focused on the former as the most practical by far.

Moreover, each atom has an associated strongly typed *value*. Values are of arbitrary *types*, and types themselves are atoms (see below for a detailed discussion of the type system). Atoms and values represent two orthogonal aspects of the information content stored in a HyperGraphDB instance. Atoms are the semantic entities that form the hypergraph structure, while values are typed data that may be structured or not. The value of an atom can be changed or replaced with a value of a different type while preserving an identical linkage structure. Several atoms may share the same value. On the other hand, values are immutable entities.[2]

Note that this model does not need to impose any particular semantics on the data being stored. It is a semi-structured, general purpose model incorporating graph, relational, and object-oriented aspects. The HyperGraphDB database schema is a type system that can evolve dynamically, where data integrity constraints can be enforced by type implementations. As exemplified by RDF, such a

[2] One could directly overwrite a value in storage or change its runtime representation and HyperGraphDB has no control over that, but there are no means in the API to modify a value.

flexible architecture is called for on the open web, where fixed database schemas can easily break.

3 Two-Layered Architecture

A key aspect of HyperGraphDB's architecture are the two levels of data organization which we now describe. We note that the actual physical storage is mostly irrelevant with the notable requirement that an efficient key-value indexing must be available, such as the BerkeleyDB storage system [18] which is currently being used.

The two-layered architecture defines a primitive hypergraph storage abstraction, the primitive storage layer and a model layer. The primitive storage layer is partitioned into two associative arrays (i.e. key-value stores):

$LinkStore : ID \rightarrow List < ID >$
$DataStore : ID \rightarrow List < byte >$

In other words, the primitive layer consists of a graph of pure identifiers and raw data. Each identifier maps either to a tuple of identifiers or to a plain byte array. The default representation of identifiers is a cryptographically strong type 4 UUID [19]. This makes it easier to manage them in a large distributed environment while virtually eliminating the chance of collision - each data peer can make up new IDs without a central authority. This layer is augmented with a set of indices some of which form an essential part of the data organization while others can be added by users or extension implementations. The main requirement from the key-value store perspective is that the indices support multiple ordered values per single key.

The model layer is where the hypergraph atom abstraction lives, together with the type system, caching, indexing and query facilities. The model layer is implemented by formalizing the layout of the primitive storage layer in the following way:

$AtomID \rightarrow [TypeID, ValueID, TargetID, ..., TargetID]$
$TypeID \rightarrow AtomID$
$TargetID \rightarrow AtomID$
$ValueID \rightarrow List < ID > | List < byte >$

In other words, each identity is taken to represent either a hypergraph atom, or the value of an atom. In the former case, the atom is stored as an identity tuple where the first argument is the (identity of the) type of the atom, the second its value, and the rest of the arguments form the target set of the atom. In the latter case, the value can be directly stored as a byte array or it can be an aggregate stored as an identity tuple as well. The layout of atom identities is managed by the core framework while the layout of data values is delegated to type implementations. Note that ValueIDs also form a recursive structure enabling the storage of arbitrarily complex structures that are orthogonal to the graph atom abstraction. The $TypeID \rightarrow AtomID$ production will be detailed in the next section.

The core indices needed for the efficient implementation of the model layer are:

$IncidenceIndex : UUID \rightarrow SortedSet < UUID >$
$TypeIndex : UUID \rightarrow SortedSet < UUID >$
$ValueIndex : UUID \rightarrow SortedSet < UUID >$

The IncidenceIndex maps a hypergraph atom to the set of all links pointing to it. The TypeIndex maps a hypergraph type atom to the set of all its instance atoms. The ValueIndex maps (the ID of) a top-level value structure to the set of atoms carrying that value as a payload.

4 Typing

HyperGraphDB has the ability to store arbitrary types of data as atoms in a programming-language-neutral way. Specifically, as an embedded database it maps data values in the host language to and from permanent storage. Therefore, it must cover typing constructs found in various languages in a way that integrates seamlessly with a given language's runtime environment. This is achieved via a general, extensible typing mechanism that interacts very closely with the storage layer, based on the foundational notion of a *type* in computer science.

In computational type theory, types are formalized by saying what can be done with them. A defining notion is equality: when are two elements of a given type equal? Another fundamental concept is compounding (or constructing) new types out of simpler ones. To make a new type, one needs type constructors, that is types of types. Finally, the notion of sub-typing, which is akin to subsumption or substitutability, is usually introduced independently. On the other hand, equality can be conceived as bi-directional subsumption: two instances of the same type are equal iff they subsume each other. The essence of types has been nicely summed up by the logician Jean-Yves Girard as "...plugging instructions. A term of a given type T is both something that can be plugged somewhere as well as a plug with free, typed variable" ([20]). This duality, together with the considerations above lead to the following minimalistic design:

- A type is an atom capable of storing, constructing and removing runtime representations of its instances to and from the primitive storage layer.
- A type is capable, given two of its instances, to tell whether one of them can be substituted for the other (subsumption relation).

In other words, types are just atoms assigned a special role that allows the system to maintain the assocation between types and instances as well as super-subtype relationships. This is formalized in the following Java interface that each type atom must implement:

```
public interface HGAtomType
{
  Object make(ID valueId,
              List<ID> targetSet,
              Set<ID> incidenceSet);
```

```
 ID store(Object instance); // return valueId for runtime instance
  void release(ID valueId);
  boolean subsumes(Object general, Object specific);
}
```

In addition to the `subsumes` relation, the interface simply defines CRUD methods needed to manage values in the storage layer. Because values are by definition immutable, there is no update method. Note that the `make` method takes both the target and incidence sets as extra parameters (besides the identifier of the value in primitive storage). Thus, the representation of an atom is a function of both those sets. This means that an atom can be annotated with various relationships that define what it is at runtime. The `make` method makes a type into an object factory. A type constructor is simply a type atom that returns an *HGAtomType* instance from its make method.

Now, the type system is bootstrapped by a set of (configurable) predefined types covering standard simple values such as numbers and strings as well some standard type constructors for record structures, lists and maps. Such predefined types all have the same type called *Top*, which is its own type. For example, the type constructor for record structures is managing record types which in turn manage concrete records. A compound type such as a record or a list can store its components by relying recursively on other HyperGraphDB types. That is, *HGAtomType* implementations can handle values both at the atom level and the pure (composite) value level.

Record-style structures with named parts are so common that we have defined an abstract interface for them called *HGCompositeType* that views complex values as multidimensional structures where each dimension is identified by a name and has an associated *HGProjection* implementation which is able to manipulate a value along that dimension. Such types that deal with complex structures are free to split them into separate identities in storage (i.e. as value links in the low-level graph) and provide functions like indexing, querying or reference counting for their parts. Alternatively, they can store them as compact blobs accessible only at the atom level.

In sum, types in HyperGraphDB aggregate many aspects commonly found in type systems in various computer languages. They play the role of object factories like classes in object-oriented languages. They are classes in the set theoretic sense, having a well-defined extension. They also define the structural properties of atoms and hence are intensional. Finally, they play a semantic role in constraining atom usage and enforcing certain consistency and integrity properties as typical in both typing and database systems. The combination of those aspects enable HyperGraphDB to naturally integrate many different formalisms without the need to single out any one in particular. Meta-models such as Sowa's conceptual graphs, RDF, combinatory logic expressions, object-orientation and even the standard relational model can be implemented "on their own terms". Furthermore, it allows for storage customization and optimization all the way to the very primitive values.

5 Indexing

Keeping in the spirit of the reflexive, open architecture of HyperGraphDB, indexing facilities are also an extensible feature. The implicit indexing of atoms described in section 3 is not optional as it is essential for an efficient support of the model layer semantics. For example, the incidence set indexing is crucial for the performance of both graph traversals and in general for querying the graph structure. Additional indices are being maintained at both the primitive and model layers.

At the primitive layer custom indices are employed by type implementations. Such indices are obtained directly from the storage layer and managed internally by a type. For instance, the default implementation of primitive types maintains an index between the primitive data and its value identifiers, thus enforcing value sharing between atoms at the storage level with reference counts. [3] Value sharing and reference counting is not done by default for complex types because there is no universal way to implement it efficiently unless the notion of primary key (i.e. a given dimension or a combination thereof) is introduced by implementing an appropriate complex type constructor.

At the model layer, indices can work both on the graph structure and value structure of atoms. However, they are always associated with atom types. This association is less constraining than it seems since an index will also apply to all sub-types of the type it is associated with. Indices are registered with the system by implementing the *HGIndexer* interface which must produce produce a key given an atom. *HGIndexer* instances are stored as atoms as well, and can thus have associated persistent meta-data for query planning, application algorithms etc. Some of the predefined indexers are listed in the following table:

ByPartIndexer	Indexes atoms with compound types along a given dimension.
ByTargetIndexer	Indexes atoms by a specific target (a position in the target tuple).
CompositeIndexer	Combines any two indexers into a single one. This allows to essentially precompute and maintain an arbitrary join.
LinkIndexer	Indexes atoms by their full target tuple.
TargetToTargetIndexer	Given a link, index one of its target by another.

Though we have not done that in the current implementation, an indexer could detect and index local graph structures to be used in more sophisticated graph mining algorithms.

6 Querying

Querying a graph database may take one of several forms. A graph traversal is a form of query yielding a sequence of atoms. Retrieving a set of not necessarily

[3] This is not necessary conceptually or technically; it is transparent at the model layer and could be changed in case different performance characteristics are required.

linked atoms matching some predicate, as in relational databases, is yet another kind. Finally, a third prominent category is pattern matching of graph structures.

Traversals are defined in terms of iterator APIs. Evidently, the more general hypergraph model entails a corresponding generalization of traversals.[4] There are several aspects to that generalization, some of which are already outlined in [16]. Firstly, the notion of node adjacency is generalized. Adjacent atoms are found as in a standard graph, by examining the links pointing to a given atom, i.e. its incidence set. Given an atom x_i and a set of typed link tuples pointing to it, one might want to examine only a subset of those links. Then given a particular link of interest $l = [x_1, ..., x_i, ..., x_n]$, the adjacent atoms could be a subset of $\{x_{i+1}...x_n\} or \{x_{i-1}...x_1\}$ depending on both atom types and properties and the direction of the traversal. Furthermore, in a case of uniform structural patterns in the graph, one might be interested in a notion of adjacency where more than on link is involved - that is, y is adjacent to x if it is reachable in a certain way. All of those cases are simply resolved at the API level by providing the traversal algorithm (breadth-first or depth-dirst) with an *adjacency list generator* - an interface that returns a list of atoms adjacent to a given atom. The second generalization of traversals adds an additional dimension to the process by allowing one to follow incident links as well as adjacent atoms. While handled equally trivially at the API, such hyper-traversals model a conceptually different phenomenon: jumps in abstractions levels within the representational structure. Because such jumps are highly domain dependent, they haven't been formalized within the system. It must be noted that both flat and hyper-traversals depend on the incidence index and the efficient caching of incidence sets.

Set-oriented queries are implemented through a simple, but efficient querying API where data is loaded on demand with lazy join, bi-directional (whenever possible) cursors. Query expressions are built out of a set of primitives[5]:

$eq(x), lt(x), eq("name", x), \ldots$	Compare atom's avalue.
$type(TypeID)$	Type of atom is $TypeID$.
$typePlus(TypeID)$	Type of atom is a subtype of $TypeID$.
$subsumes(AtomID)$	Atom is more general than $AtomID$.
$target(LinkID)$	Atom belongs to the target set of $LinkID$.
$incident(TargetID)$	Atom points to $TargetID$.
$link(ID_1, \ldots, ID_n)$	Atom's target set includes $\{ID_1, \ldots, ID_n\}$.
$orderedLink(ID_1, \ldots, ID_n)$	Atom is a tuple of the given form.
$arity(n)$	Atom's arity is n.

as well as the standard *and*, *or* and *not* operators. Much of the querying activities revolve around performing join operations on the various indices available for the

[4] In hypergraphs as finite set system, traversals are generalized to the notion of transversals which are sets having non-empty intersection with every edge of the graph. Computing minimal transversals has applications in various combinatorics algorithms, but HyperGraphDB has been mainly applied to tuple-oriented representational problems and instead provides facilities for computing directed n-ary traversals.

[5] For brevity, we list only the most fundamental ones.

query. Small to moderately sized atom incidence sets are cached as ordered ID arrays, which makes scanning and intersecting them extremely efficient. Wholes or portions of larger storage level indices get cached at the storage layer as well, but remain in B-Tree form. Joins are mostly performed on pairs of sorted sets (usually of UUIDs) via a zig-zag algorithm, when the sets support key-based random access, or a merge algorithm, when they don't. Intersections are implemented as bi-directional iterators, which allows for backtracking during more sophisticated search algorithms.

Because of the generality of the HyperGraphDB data model, querying for patterns would most conveniently be done through a special purpose language in the style of [21]. Such a query language, accounting for the various structural possibilities of typed atoms with directed or undirected hyperedges and complex values, is currently a work in progress.

7 Peer-to-Peer Distribution

Data distribution in HyperGraphDB is implemented at the model layer by using an agent-based, peer-to-peer framework. Algorithms are developed using communication protocols built using the Agent Communication Language (ACL) FIPA standard [22]. ACL is based in turn on speech act theory [23] and it defines a set of primitive communication performatives such as *propose*, *accept*, *inform*, *request*, *query* etc. Each agent maintains a set of conversations implementing dialog workflows (usually simple state machines) with a set of peers. All activities are asynchronous where incoming messages are dispatched by a scheduler and processed in a thread pool.

The P2P architecture was motivated by the most common use case of HyperGraphDB as a distributed knowledge representation engine. It is assumed that total availability of all knowledge at any particular location will not be possible, or in many cases not even desirable. For example, an eventually consistent data replication schema is based on the following algorithm:

1. Upon startup each agent broadcasts interest in certain atoms (defined by a boolean predicate) to its peers by sending a *subscribe* performative.
2. Each peer then listens to atom events from the database and upon addition, update or removal of an atom, it notifies interested peers by sending an *inform* communication act.
3. To ensure consistency, local transactions are linearly ordered by a version number and logged so that they can eventually reach all interested peers.
4. A peer that receives a notification about an atom transaction must acknowledge it and decide whether to enact the same transaction locally or not. There is no two-phase commit: once an event is acknowledged by a receiver, the sender can assume that it was processed.

This approach pushes the balance of replication vs. partitioning of data to the user level. We felt this was the appropriate choice since simple partitioning schemas, such as by type or by value, are easily configured on top of the base

algorithm while application-dependent graph structures can only be distributed meaningfully at the domain level. A fully transparent alternative that attempts to adapt the precise location of atoms based on local usage and meta-data would result in unpredictable and frequent remote queries, in addition to requiring a significant amount of additional local storage.

8 Example Application

One area where HyperGraphDB has been successfully applied is Natural Language Processing (NLP). We developed a distributed NLP processing system called Disko where the representational power of the data model considerably simplified the task. Disko uses a relatively standard NLP pipeline where a document is split into paragraphs and sentences, then each sentence is parsed into a set of syntactic dependency relations. Finally semantic analysis is applied to those relations yielding a logical representation of their meaning.

In this application, several domains are naturally represented independently while interacting in a clean and meaningfull way. First, a manually curated OWL ontology is stored as a hypergraph representing OWL classes as types that create runtime instances of OWL individuals based on their graph linkage. The OWL classes themselves are typed by an $OWLTypeConstructor$. Prior to parsing, an entity detection algorithm maps named entities (such as people and organizations) into individuals of that ontology. Second, the WordNet lexical database [24] is represented by storing synonym sets as undirected links between word atoms and semantic relationships between those sets (e.g. part-whole) as 2nd order directed links. A parser based on a dependency grammar [25] combined with a postprocessing relational extractor [26] yields strongly typed syntactic relationships between word atoms while a WSD (Word Sense Disambiguation) phase yields semantic relationships between synonym sets, again 2nd order, n-ary links. Further semantic analysis is performed by Prolog rules operating directly on the database instance via a mapping of logic terms to HyperGraphDB atoms, yielding further semantic relations. Both syntactic and semantic relations are of varying arity ranging from 1 (e.g. verb tense) to sometimes 4 (e.g. `event(buy, time, buyer, seller, object)`). The tree-like structure of the document is also recorded in HyperGraphDB with scoping parent-child binary links between (a) the document and its paragraphs, (b) a paragraph and its sentences, (c) a sentence and each linguistic relationship inferred from it. Finally, an implementation of the dataflow-based programming model ([27]) also relies on HyperGraphDB to store its distributed dataflow network.

A semantic search based on the NLP pipeline output makes heavy use of HyperGraphDB's indexing capabilities. First, a user question is translated into a set of relationships using the same pipeline. Ontological and lexical entities are quickly resolved by various $ByPartIndexer$s on their object-oriented representation. Non-trivial relationships (the ones with $arity > 1$) between those entities are matched against the database using $orderedLink(x_1, x_2, ...)$ queries. Such queries are translated to joins between the incidence sets of x_1 through

x_n and possibly including $ByTargetIndexer$s on some particularly large types of relationships. The matches thus accumulated are grouped according to their sentence/paragraph scopes, where the scopes themselves are efficiently resolved via a $TargetToTargetIndexer$, which is usually applied when one wants to efficiently navigate a tree sub-graph of a large hypergraph. The groups are scored and sorted into a search result set.

Note that both the ability of links to hold more than two targets and to point to other atoms is essential in this representation. Furthermore, the type system simplifies the programming effort by (1) making the mapping between persistent data and runtime representation transparent and (2) enforcing schema constraints on the representation. An approach relying on most other database models (including ODBMs, RDBMs, RDF stores, classical graph stores etc.) would force a contrived mapping of the several disjoint domains involved to a representation motivated solely by storage.

9 Conclusion

The data model adopted by HyperGraphDB generalizes classical graphs along several dimensions (reified n-ary links, strong typing) and as such it doesn't lend itself to a pointer structure based storage layout. On the other hand, an open layered architecture and an extensive type system allow the model to easily subsume many different formalisms without necessarily incurring a performance penalty in the process. Further development of the query language for the HyperGraphDB model, bindings for other programming languages such as C++, support for nested graphs and distributed algorithms based on the established foundation will hopefully fuel the adoption of this data model also in domains outside AI/NLP/Semantic Web research.

References

1. Angles, R., Gutierrez, C.: Survey of Graph Database Models. ACM Computing Surveys 40(1), Article 1 (2008)
2. Levene, M., Poulovassilis, A.: The Hypernode model and its associated query language. In: Proceedings of the 5th Jerusalem Conference on Information technology, pp. 520–530. IEEE Computer Society Press, Los Alamitos (1990)
3. Levene, M., Poulovassilis, A.: An object-oriented data model formalised through hypergraphs. Data Knowl. Eng. 6(3), 205–224 (1991)
4. Boley, H.: Directed recursive labelnode hypergraphs: A new representation-language. Artificial Intelligence 9(1) (1977)
5. Goertzel, B.: Patterns, Hypergraphs & Embodied General Intelligence. In: IEEE World Congress on Computational Intelligence, Vancouver, BC, Canada (2006)
6. Sheard, Tim: Languages of the Future. In: Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages and Applications, Vancouver, British Columbia, Canada, (2004)
7. Klamt, S., Haus Utz-Uwe, Theis, F: Hypergraphs and cellular networks. PLoS Comput. Biol. 5(5) (2009)

8. Kok, S., Domingos, P.: Learning markov logic network structure via hypergraph lifting. In: Proceedings of the 26th Annual International Conference on Machine Learning, ICML 2009, pp. 505–512. ACM, New York (2009)
9. Resource Description Framework, http://www.w3.org/RDF/
10. RDF Semantics, http://www.w3.org/TR/rdf-mt/
11. Carroll, J.J., Bizer, C., Hayes, P., Stickler, P.: Named graphs, provenance and trust. In: Proceedings of the 14th international conference on World Wide Web, WWW 2005, pp. 613–622. ACM, New York (2005)
12. Topic Maps, http://www.isotopicmaps.org/
13. Berge, C.: Hypergraphs: combinatorics of finite sets. North-Holland, Amsterdam (1989)
14. Gallo, G., Long, G., Pallottino, S., Nguyen, S.: Directed hypergraphs and applications. Discrete Applied Mathematics 42(2-3), 177–201 (1993)
15. XML Schema, http://www.w3.org/XML/Schema
16. Boley, H.: Declarative operations on nets. In: Lehmann, F. (ed.) Semantic Networks in Artificial Intelligence, pp. 601–637. Pergamon Press, Oxford (1992)
17. Aczel, P.: Non-well-founded sets. CSLI Lecture Notes, vol. 14. Stanford University, Center for the Study of Language and Information, Stanford, CA (1988)
18. Olson, M.A., Bostic, K., Seltzer, M.: Berkeley DB. In: Proceedings of the Annual Conference on USENIX Annual Technical Conference, p. 43. USENIX Association, Berkeley (1999)
19. IETF UUID draft specification, http://www.opengroup.org/dce/info/draft-leach-uuids-guids-01.txt
20. Girard, J.-Y., Lafont, Y., Taylor, P.: Proofs and Types. Cambridge University Press, Cambridge (1989)
21. SPARQL Query Language for RDF, http://www.w3.org/TR/rdf-sparql-query/
22. FIPA Agent Communication Language Specification, http://www.fipa.org/repository/aclspecs.html
23. Searle, J.: Speech Acts. Cambridge University Press, Cambridge (1969)
24. Miller, G.A.: WordNet: A Lexical Database for English. Communications of the ACM 38(11), 39–41 (1995)
25. Sleator, D., Temperley, D.: Parsing English with a Link Grammar. Carnegie Mellon University Computer Science technical report CMU-CS-91-196 (1991)
26. RelEx Dependency Relationship Extractor, http://opencog.org/wiki/RelEx
27. Morrison, J.P.: Flow-Based Programming: A New Approach to Application Development. Van Nostrand Reinhold, New York (1994)