Multilayer Category-Theoretic Knowledge Representation of Regulatory Documents

Sumin Leem
University of Calgary
Calgary, AB, Canada
sumin.leem@ucalgary.ca

Kristine Bauer
University of Calgary
Calgary, AB, Canada
bauerk@ucalgary.ca

Nathaniel Osgood
University of Saskatchewan
Saskatoon, SK, Canada
osgood@cs.usask.ca

Regulatory documents are not organized as isolated rules, but as interconnected systems of provisions, definitions, exceptions, and cross-references. Modeling explicit logical structure, including relevance, dependency, and order of evaluation, can help organize regulatory documents for computation across multiple levels of applicability and evaluation. In this paper, we propose a three-layer category-theoretic model for regulatory documents: a versioning layer, a layer of document structure and rule traversal, and a layer of local rule semantics. This framework gives a knowledge representation of regulatory documents using Spivak and Kent's ontology logs (ologs) [7] that supports structured reasoning. We illustrate the three-layer framework in the setting of building codes to demonstrate how these layers can be instantiated in a concrete regulatory domain.

1 Introduction

Legal documents are an important part of our daily life. Documents such as contracts, laws and safety codes are essential to understanding our rights and responsibilities as we engage in activities which involve other people. However, many of these documents are highly technical, involve a great deal of legal jargon and are extremely lengthy. These features are barriers which make it difficult for the average person to be able to find the information that they need in the document.

An obvious idea, given the proliferation of applications of artificial intelligence, is to use a computer-assisted approach to navigating legal documents. However, large language models (LLMs) are prone to hallucination and other errors. For example, in a recent study about the use of artificial intelligence use in legal documentation [4], not only were general-purpose chatbots such as GPT-4 observed to hallucinate case-law with alarming frequency but even "hallucination free" special purpose AI research tools hallucinated or made other errors between 17% and 33% of the time. For legal documents which govern life and death situations, such as safety codes, any amount of hallucination is unacceptable and can risk human health and safety. However, humans also make errors and AI tools can be used to reduce these. In an ideal world, we would be able to use automated tools with 100% accuracy and minimal room for human error.

An alternative, conventional, approach would be to seek to encode the rules of such legal documents directly into a relational database, so as to support user queries regarding compliance. While this approach would sidestep many of the issues that limit the effectiveness of LLMs, it would suffer other serious shortcomings. The hard-coding of the rules into queries would stymie multi-purpose use of such rules, for example for reasoning not only in support of assessing compliance, but also for finding alternative project parameters that would allow for compliance. Such hard-coding would further offer little benefit a myriad of potential valuable supporting tools for working with the code, for example, to identify logical satisfiability challenges associated with the regulatory code, to visualize the logical structure of that regulatory code, or to identify transformations of the presentation of that code that would leave

Submitted to:
Applied Category Theory 2026

invariant the underlying logic but ease communication. While such supporting tools could be separately built, doing so risks the duplication of the logic of such rules in several places. Such hard-coding of regulatory rules into queries would further render laborious the evolution of the query-based system as the regulatory code evolves through successive incremental updates.

To more effectively and sustainably address this problem, we set out to provide a way to model highly structured documents which would result in a knowledge graph representation of the document itself. Such a knowledge graph could be used to automate the process of finding information in the document without risk of hallucination, since there would be no information in the knowledge graph which is not in the document itself. Our approach is to model the internal structure of rules in regulatory documents, rather than treating the text as unstructured prose. This supports consistent rule representation, dependency handling, provides a natural foundation for traceable reasoning, and graceful incremental updating in parallel with the incremental updates characteristic of regulatory documents.

We focused on building codes as an example of a highly structured document which is representative of the challenge we described: building codes contain a lot of jargon, are difficult to read, yet impact human lives and the environment, making proper understanding extremely important for safety and effective and efficient operation of the real estate sector. There are several approaches to automated building code compliance checks in the literature. For example, [8, 5] both automate the pipeline of compliance checks using a floor plan or other Building Information Models (BIM), which organize information according to the elements of the building plan itself. The approach presented here focuses on mathematically characterizing and using the structure of the building code. This work benefits from the fact that building codes are often broken into interconnected, labelled parts whose labels convey semantic relationships between the different units. The goal of our work is to provide a chatbot or other system which would allow a builder to query the building code, e.g. by asking "does my project require additional exits?".

We construct a knowledge representation system using ontology logs, or ologs, from [7]. An olog can be used to produce a graph which aids in natural language processing. However, an olog is a categorical construction which has several advantages over other methods of modeling language. Ologs are defined in Section 2. One advantage of using ologs in highly structured, legal documents such as building codes is that ologs are functorial by their nature. An instance of an olog is a functor from the olog to the category of sets which populates each vertex and directed edge. This means that the olog can be an especially useful tool in version control. With minimal changes, the entire knowledge graph can be updated quickly and efficiently to reflect the new version.

In this paper, we explain how we used a multi-layer approach to extract a knowledge graph for a building code. The three layers of ologs we use in our algorithm are (1) a versioning layer, whose function is to select the correct version of a document (such as a building code in the right jurisdiction or whose effective date indicates its relevance to a project), (2) a document structure layer, which organizes the document into sections and governs how various parts of the document are prioritized or related to one another, and finally (3) the rule semantics layer, which uses ologs to model sentence structure and the meaning of the rules in the document. Each layer determines the information in the next layer using a functional relationship which makes versioning easy to manage.

This work forms part of a broader project that remains under development. The long-term aim is to pair the olog-based knowledge representation developed here with an algorithm that can search the resulting graph to answer user queries about a building project. Viewed in this way, the present paper represents an initial step toward a broader structured reasoning framework for complex regulatory documents, combining machine-readable knowledge representation, structured reasoning, and AI-assisted encoding of logical structure.

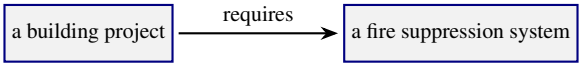
This paper is organized as follows. In Section 2, we provide some necessary backgroud on ologs summarizing the relevant portions of [7]. In Section 3, we describe our construction of the multilayered olog approach in detail. In Section 4, we describe a detailed example of how we applied our techniques to the British Columbia Building Code document.

Acknowledgements

Sumin Leem gratefully acknowledges support from the Mitacs Elevate program. The authors also thank Clause Technology and Sogol Ghattan for their support of this work, Reza Honarpisheh for his domain expertise and assistance with the case study, and Dr. Chris Dutchyn for his guidance regarding Z3. Kristine Bauer would like to thank the Pacific Institute for the Mathematical Sciences for their support of the Math to Power Industry Program, from which this project originally stemmed.

2 Preliminaries: Ologs

A knowledge graph is a graph database that interconnects data. Knowledge graphs are a kind of concept map which include all of the important data about a concept as well as the way that data relates. Using knowledge graphs is an effective way to enable computers to understand and process human knowledge. A knowledge graph has nodes and edges in which each node represents an entity and each edge represents a relation. The edges in a knowledge graph are directed. Knowledge graphs appear frequently in computer science literature especially pertaining to machine learning, artificial intelligence or natural language processing (see, e.g. [2] for a survey of the uses of knowledge graphs). Typically, the directed edges represent verbs in a sentence and the direction of the arrow is from the subject of the sentence to the object. For example:



The knowledge graphs that we will use are produced using a concept called an *olog*. An ontology log, or olog, is a framework for understanding a particular object of study, such as building codes. Ologs were invented by D. Spivak and R. Kent [7] as a way of keeping track of the data of an ontology, or study of a particular object or concept, in a way which can be both graphically represented and easily translated into code. In brief, an olog consists of three parts.

- *Types* are collections of things that all have something in common. Typically, a type is a broad descriptor for a set of objects or entities. Types are represented by nodes in a graphical representation.
- An *aspect* represents a relation between types. Linguistically, aspects are usually transitive verbs. Aspects are represented by directed arrows in a graphical representation.
- A path in an olog is an alternating sequences of types and aspects which begin and end with a type. These can be translated into sentences. A *fact* is a declaration that two paths between two given types correspond to sentences which have the same meaning. These are represented by commuting diagrams in a graph.

An olog is a category, O, whose objects are types, whose morphisms are aspects, and whose commuting diagrams (or equality of parallel morphisms) are facts. The underlying graph, $|O|$, of the category O

can be used to produce a knowledge graph. To attach meaning to the category, O, we must populate the objects and arrows of O with sets and functions corresponding to the types and aspects of O. Let **Set** be the category of sets and functions.

Definition 2.1 *[7, Definition 3.2.3] An* instance *of O is a functor $I_O : O \to$ **Set**. We denote an instance of an olog by (O, I_O).*

The underlying graph of the subcatgory of **Set** produced by the image of I_O is a knowledge graph in the traditional sense. A distinction between traditional knowledge graphs and ologs is that an olog formalizes the fact that two distinct sentences may have the same meaning using facts. Commuting diagrams in the category O model facts in a populated instance of the olog because the functor I_O will preserve commuting diagrams. Ologs are related to database schemas and vice-versa since both of these concepts are modeled by small categories [6].[1]

One reason that using an olog is very useful in modeling highly structured regulatory text is that regulatory texts very frequently contain lists of requirements and exceptions for a rule to be met. These lists and exceptions can be modeled using coproducts and products.

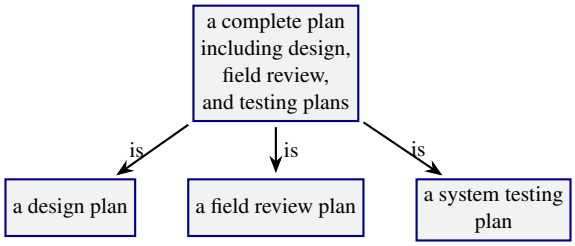
Example 2.2 *This example serves to demonstrate how categorical structures such as coproducts and pullbacks can be used to model regulatory code. The National Building Code of Canada Alberta Edition has the following requirement (amongst many) for obtaining a building permit:*

> *Unless deemed by the authority having jurisdiction to be a minor alteration to an existing system, all plans produced for an automatic fire suppression system referred to in Sentence 2.4.1.3.(1) shall be authenticated by a registered professional. [1]*

At the heart of this part of the building code, there is a requirement that **all automatic fire suppression system plans shall be authenticated**. *However, the requirement contains more information: Sentence 2.4.1.3.(1) details that there are three specific criteria which must be included in the "plan", and the sentence allows for an exception in the case that the jurisdiction deems the plans to be part of a minor alteration. In practice, someone who wanted to comply with this regulation would need to present plans which included the* **union** *of the requirements in Sentence 2.4.1.3.(1) for each plan, and which* **excluded** *plans deemed to be minor alterations.*

In general, a *requirement* in an olog in its simplest form is an aspect, which is an edge in the associated knowledge graph. It is no surprise that categorical limits and colimits play an important role in ologs. Specifically, products are used to model situations in which several requirements must be met simultaneouly, the coproducts will be used to model a situation in which one or another requirement must be met.

In Example 2.2, the "plans" must include three additional pieces of information contained in Sentence 2.4.1.3.(1) of the building code: a design plan, a field review plan, and a system testing plan. The graphical representation of the olog corresponding to the idea that the complete plan comprises these three elements is:

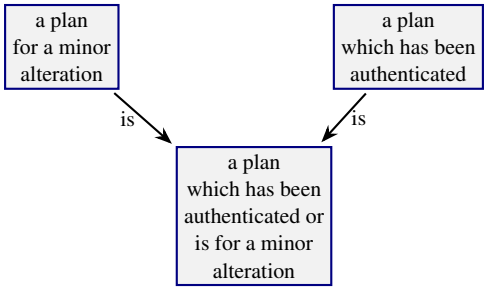


[1] Unlike ologs, database schemas often use attributed data. Attributed data is outside the scope of this paper because it is not a feature of ologs, so we set this aside.

The three conditions comprise what might be called a *complete* plan, that is, a plan which contains each of the specified pieces of information at once. The graphical representation for this is the usual way of representing a product as a diagram, but with information at the nodes and arrows conveying the same meaning as what is in the building code.

On the other hand, the keyword “unless” in Example 2.2 indicates that there are conditions which must not be met at the same time. In this case, either a fire suppression system plan must part of a minor alteration, or the fire suppression plan must be authenticated by a registered professional.

In the category of sets, the disjoint union is the categorical coproduct, which can be used to model the kind of exclusion of minor alterations mentioned in Example 2.2 since it corresponds to the exclusive “or":



Here, we envision that it is impossible for a plan for a minor alteration to be authenticated by a registered professional. If this is possible, then this coproduct diagram could be replaced by a pushout where an initial node “a plan for a minor alteration which has been authenticated” could be included. Taken together, the plans in the fragment of the olog in this graphical representation would satisfy the requirement that they be authenticated *unless* the plans are for a minor alteration.

The structure of coproducts and products or more general limits and colimits demonstrated in these examples allows us to encode complex ideas using diagrams which have fairly simple nodes. Breaking complex ideas into simpler types and aspects is an advantage of using limits and colimits in ologs in general.

In the next sections, we will see how to use ologs in a multilayered approach to analyze regulatory texts.

3 Multilayer Operational Categorical Model

In this section, we introduce a multilayer operational categorical model for regulatory documents. We envision a context in which a user wishes to find specific information about a project that is governed by the regulations in the document. The layers are organized by the different questions that must be resolved to evaluate compliance with the regulation: which regulatory document applies, which parts of the document govern the project, and which local rule logic must ultimately be encoded.

The first layer is the versioning layer, which determines the unique version of the document which applies to the situation at hand. This is determined by a user’s perspective and includes contextual clues such as the jurisdiction or time frame which apply to a project. The second is the document structure layer, which organizes the applicable document into subsections and governs how relevant portions are traversed or prioritized. The third is the rule semantics layer. In this part, the semantics of the smallest atomic parts of a document (usually sentences) are modeled using ologs as in Example 2.2. These ologs encode the compliance logic of the rules in the document.

Accordingly, a query is processed through three layers in sequence. It is first matched to an appropriate version of a regulatory document, then routed through the relevant document structure, and finally evaluated against the corresponding rule semantics. In this way, the decomposition is not only descriptive, but can capture the order in which compliance reasoning proceeds.

The remainder of the section develops these layers in turn. Section 3.1 introduces the versioning layer. Section 3.2 explains how that layer determines an appropriate document structure and the associated operating logic. Section 3.3 then describes how local rule semantics are assigned to sentences within the document structure layer.

3.1 Versioning

A regulation may admit different expressions in different regulatory documents depending on factors such as the time frame, jurisdiction, and other applicability conditions. The versioning olog, O_V, captures these factors before considering the organization of any rules within the document. We refer to O_V as the versioning olog because it represents the conditions that determine the regulatory framework that is in force.

An applicability condition is a condition used to determine which version of a regulatory document is applicable to a given query. Depending on the regulatory setting, such applicability conditions may include jurisdiction, effective date, or other contextual qualifiers. This layer is important in regulatory settings because the correct regulatory document is not always determined by the regulation name alone. For example, when a law is amended, the applicable law may depend on the date of the relevant event or decision rather than simply on the latest version of the text. Making these conditions explicit supports traceability in how a query is evaluated.

To construct O_V, we begin with a type for the project (the subject of the query) and a type for the regulatory document versions. For each applicability condition, we add aspects to represent the fact that a given project satisfying an applicability condition is governed by a specific version of a regulatory document which is in force whenever that applicability condition is satisfied. In practice, we include enough applicability conditions to ensure that a regulatory document is uniquely determined by the conditions. Figure 1 displays the fact corresponding to a single applicability condition, while, in general, O_V con-

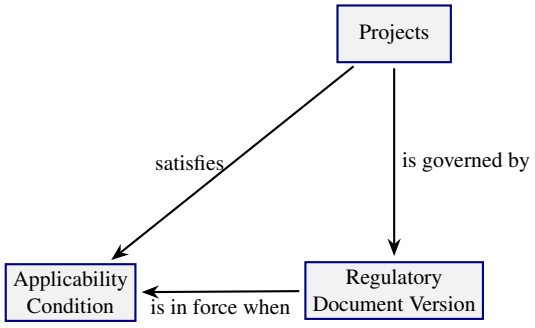


Figure 1: A fact in O_V with a single applicability condition

tains one such fact for each applicability condition relevant to the regulatory setting under consideration. An example of this construction specific to building codes can be found in Figure 2 in Section 4.

An instance, $I_V : O_V \to \mathbf{Set}$, populates each of the nodes in O_V with projects satisfying parameters including applicability conditions which we may have in mind in our context, regulatory document versions which we wish to consider and applicability conditions relevant to our situation which satisfy a

fact corresponding to each applicability condition. Given a query about a project, P in $I_V(Projects)$, we obtain a unique regulatory document D by evaluating $I_V(\text{is governed by})(P)$. This function depends on the instance (O_V, I_V) of our olog, and the value of D is determined only by the choice of project, P, once the olog is instantiated.

Proposition 3.1 *The above choice of types and aspects defines an olog O_V.*

Once the olog is instantiated and a project is selected, the applicable document edition can be determined in this way, yielding the valid context needed to proceed. The analysis then proceeds to the document structure layer for the corresponding document version.

3.2 Document Structure

In the previous layer, given a project, we determined a unique regulatory document that governs the project. This layer models the structure of that document. The document structure includes not only the hierarchical organization of the document (e.g. sections, subsections, etc.) but also relations between parts of the structure, such as cross-references, links between sections that uses a definition from another, and exception-relations. It therefore captures how parts are arranged and related to one another within the document, rather than the content of the parts themselves. For example, this layer may record that one provision is contained within another, that one provision refers to another, or that one provision functions as an exception to another. These relations help determine which parts of the document must be considered before the content of any particular provision is analyzed in detail. In practice, we use this to construct an olog which may only apply to the parts of the document related to a specific query about a project. This is a resource-saving step which makes it possible to construct knowledge graphs which are manageable and realistic in the final layer.

Let D be a regulatory document version associated to a project P obtained from (O_V, I_V) in the versioning layer. To construct the olog $O_{V,D}$ for the document structure layer, we model the hierarchical organization of the document and the references between its provisions in different ways. Hierarchical containment relations, such as a subsection belonging to a section, are naturally represented by an aspect in the form of a directed edge from the smaller part to the larger part indicating the containment. References between provisions need not be represented by functions, since a given provision may refer to more than one other provision. Rather than encoding such a relation using a single aspect, we use a new type consisting of pairs whose instances are ordered pairs (A,B) with the interpretation that A refers to B. Aspects which originate from this type are projections that recover the source provision and the target provision.

We include a type for compliance topics. An instance of this type is an index that lists specific articles related to the topic of a given query about whether or not a project complies with the regulations. See Section 4 for further details.

We also insist that there is a lowest level of structure in the hierarchical organization of the document. Consider the subcategory consisting of the different provisions and the hierarchical containment between them. We insist that this subcategory has an initial object T_D. This type will be used in the next layer to analyze the semantics of rules in the document.

This constructs an olog $O_{V,D}$ which depends only on the version of the regulatory document, D, determined by a choice of project in the previous layer. The next proposition follows from the construction outlined here.

Proposition 3.2 *Let O_V be the versioning olog from Section 3.1 and let (O_V, I_V) be an instance. Each*

instance of a project, P, in (O_V, I_V) determines an associated document structure olog $O_{V,D}$ where D is the regulatory document version determined by P.

Hence, the document structure layer is a family $\{O_{V,D}\}$ indexed by the projects of an instance (O_V, I_V) of the versioning olog. A particular version of a regulatory document, D, determines an instance $I_{V,D}$ of $O_{V,D}$ by sending the types of $O_{V,D}$ to their set of instances in D, e.g, to the set of sections of D.

This assignment $D \mapsto O_{V,D}$ means we may be view $\{O_{V,D}\}$ as a set-indexed family of small categories, indexed by the set of versions of regulatory documents. An advantage of this assignment is that it can easily accommodate variation in document structure across different regulatory documents. For example, the structural organization of Canadian criminal law may differ substantially from that of Alberta building codes. Fully declaring a document's structure in this layer adds to transparency.

3.3 Rule Semantics

The rule semantics layer encodes and stores the actual rules in the document. These rules are the elements of an instance of the initial object T_D from the document structure layer. Each element of the set of smallest hierarchical structures $I_{V,D}(T_D)$ is a rule. For example, if the smallest hierarchical structure is the sentences of the document, D, then the rules of D are the sentences of D and each one can be modeled using an olog as in Section 2.

This layer includes two closely related components; (1) rules from the document and (2) implicit type constraints. The first is the explicitly stated rule content carried by each instance of T_D in $(O_{V,D}, I_{V,D})$. The second is a type-semantic component, consisting of implicit constraints on the variables appearing in the rules. Although such type semantics could be separated into an additional layer in a more elaborate architecture, we group them here because they support semantics for rule evaluation.

Each $\tau \in I_{V,D}(T_D)$ is assigned a local rule olog $O_{R,\tau}$. The role of $O_{R,\tau}$ is to represent the explicitly stated compliance logic in τ: the conditions under which the provision applies, the resulting requirement or consequence, and the logical relations among these components. In regulatory text, this local logic often includes conjunctions, disjunctions, thresholds, and exception-like structure as demonstrated in Section 2.

Each $O_{R,\tau}$ is determined by a choice of τ in $I_{V,D}(T_D)$. That is, there is a function

$$\Phi_{V,D,R} : I_{V,D}(T_D) \to \mathbf{Cat}_{small}, \qquad \tau \mapsto O_{R,\tau},$$

sending each instance τ of type T_D to its associated rule olog $O_{R,\tau}$, which is an object of the category of small categories. This map defines a set-indexed family of rule ologs, indexed by instances of the designated rule-bearing type T_D. Choosing a different T_D in the document structure layer will change the domain of τ, and therefore change the overall structure of this layer. This is why we insist that T_D is initial: it is uniquely determined by the category $O_{V,D}$.

Proposition 3.3 *Let $(O_{V,D}, I_{V,D})$ be an instance of the document structure olog for the document D, and let T_D be the initial object of $O_{V,D}$. The function $\Phi_{V,D,R}$ defines a family of local rule ologs indexed by instances of T_D.*

This locality is important both mathematically and operationally. Mathematically, it makes the rule layer a family of small categories indexed by instances of the type T_D in the document structure. Operationally, it allows rule evaluation to proceed in a modular way: once we identify the relevant provision in $O_{V,D}$, the corresponding $O_{R,\tau}$ provides the immediate semantic object on which compliance checking is performed. This modular structure is also compatible with downstream formal evaluation methods,

since each local rule olog can in principle be translated into a logical representation and checked independently once the relevant inputs are fixed. In particular, satisfiability modulo theory (SMT)-based evaluation is a candidate when rule satisfaction must be checked under partial or incomplete information. The same modularity also supports maintainability, since changes in one provision can often be reflected by modifying the corresponding local rule olog without reconstructing the entire semantic layer.

3.3.1 Implicit Type and Domain Constraints

Encoding only the rule explicitly stated in a sentence is not sufficient for machine-readable representation. Regulatory rules typically rely on supporting semantic constraints: (1) **implicit type constraints** that human readers are expected to know from common sense and (2) **domain constraints** that are defined elsewhere in the document. Thus, in addition to the rules from the document, local rule evaluation requires semantic structure attached to the variables and terms occurring in that rule. For example, a human would know that a distance cannot be negative, but a machine does not automatically recognize that constraint. In addition, when there is a defined term such as an “egress window", which is a window bigger than a certain size and meeting several conditions, those conditions for defined terms must also be represented in this layer.

4 Case Study: Building codes

In this section, we illustrate the application of the multilayer model of Section 3 to building codes.

Building codes are regulatory documents that specify requirements for building construction and renovation. Different codes may apply in different jurisdictions, and each code typically has multiple versions because regulations are revised over time.

Construction of O_V: For building codes, the role of an instance of the versioning layer (O_V, I_V) is to determine which regulatory regime and which version of the regime apply to a given project, P. When considering building code compliance, applicability is determined primarily by jurisdiction and application date. Jurisdiction identifies the governing authority for the project. For example, projects within British Columbia are generally governed by the British Columbia building code (BCBC). The application date identifies the relevant version of the code, since building code requirements change over time and transition rules may affect which version applies. Thus, a renovation project is not necessarily assessed against the latest code, but against the code version legally applicable to the project. Figure 2 shows an example olog O_V. **How O_V works:** Now, suppose that P is a renovation project located in

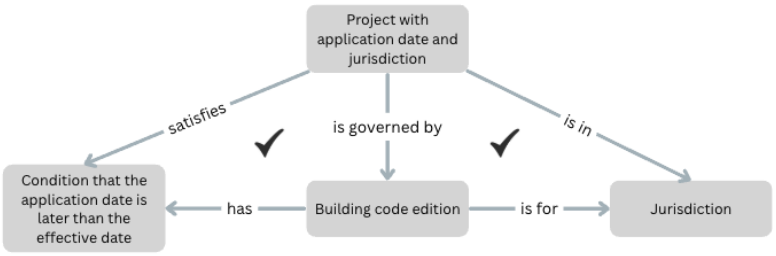


Figure 2: An example of versioning and perspective olog O_V for building codes

British Columbia, Canada, with application date March 31, 2026. By evaluating $I_V(\text{is governed by})(P)$, we obtain the document $D = \text{BCBC 2024}$, which is the applicable building code edition for P.

Construction of $O_{V,D}$: Since we obtained D (which is BCBC 2024) from P and (O_V, I_V), the document structure olog $O_{V,D}$ takes the hierarchical organization of BCBC 2024 as its structural backbone, including types such as Division, Part, Section, Subsection, Article, Sentence, Clause, and Subclause. As mentioned in Section 3.2, $O_{V,D}$ is constructed with two modeling choices: (1) we attach compliance topics at the Article level for query-based retrieval, since the article provides a useful intermediate unit: more specific than a subsection, while still broad enough to collect related provisions under a common local context, and (2) we take the Sentence to be the designated rule-bearing type T_D, since in building codes sentences often provide a sufficiently local unit for representing an individual rule or condition. Figure 3 provides a part of $O_{V,D}$ that captures hierarchical structure of the document and shows how

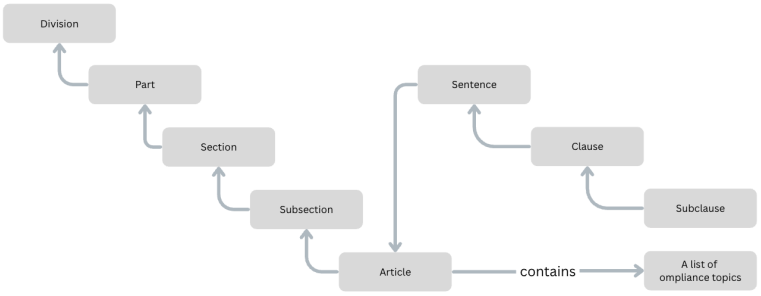


Figure 3: Types for hierarchical structure and the type for a list of compliance topics

each article is connected to a list of compliance topics. All unnamed aspects are “belongs to" (omitted for visual simplicity). A list of compliance topics can be connected with different types if one chooses a different type for the query-based retrieval. Figure 3 does not include supplementary informational objects, such as Note and Table, for simplicity. We choose to ignore the provisions called “Clause” and “Subclause” since these often contain sentence fragments which are not modeled well by the ologs described in Section 2. Thus, the initial object is $T_D = \text{Sentence}$.

In addition to the structure shown in Figure 3, $O_{V,D}$ contains types representing references between provisions, which are used to determine the order of rule evaluation. In our case study, the primary reference types are exception, compliance, and informational reference; $\text{Exception}(A,B)$ means A is an exception of B, $\text{Compliance}(A,B)$ means A must comply with B, and $\text{InfoRef}(A,B)$ means A refers B for information. Since the rules will be incoporated in the Sentence type, the source provision of each reference-pair is a sentence, while the target provision may belong to any structural type. These reference-pair types and their associated source and target aspects are omitted from Figure 3 for simplicity.

How $O_{V,D}$ works: Consider a question Q: “For a suite renovation, how many means of egress are required?" The question contains one compliance topic: number of means of egress, which can be used as a query to retrieve relevant articles. First, we search for the query in a list of compliance topics and retrieve the indices of relevant articles. Second, we retrieve all sentences in the retrieved articles. Among retrieved sentences, the order of evaluation can be determined by the reference-pairs involving those sentences. In our traversal convention, for a reference-pair $\text{Exception}(A,B)$ or $\text{Compliance}(A,B)$, A is evaluated before B, while for a pair type $\text{InfoRef}(A,B)$, B is consulted as an information source for evaluating A rather than being evaluated as a separate rule.

Construction of $O_{R,\tau}$: Once the applicable sentences and the order of evaluation is determined by $O_{V,D}$, we consider each retrieved sentence τ and its assigned olog $O_{R,\tau}(= \Phi_{V,D,R}(\tau))$ can be used to answer the question Q. The following is a simplified example of τ that is relevant to the question Q.

Example 4.1 τ*: Except for dwelling units, a minimum of 2 egress doorways from the suite shall be provided for every suite that is used in a floor area that is not sprinklered throughout, and the area of a room or suite is more than 200 m*^{2} *or the travel distance within the suite to the nearest egress doorway is more than 25 m.*

This sentence τ determines when a suite is required to have two means of egress. To construct $O_{R,\tau}$ from τ, we first separate the consequence of τ from its conditions, and then organize those conditions according to the logical structure expressed in the sentence.

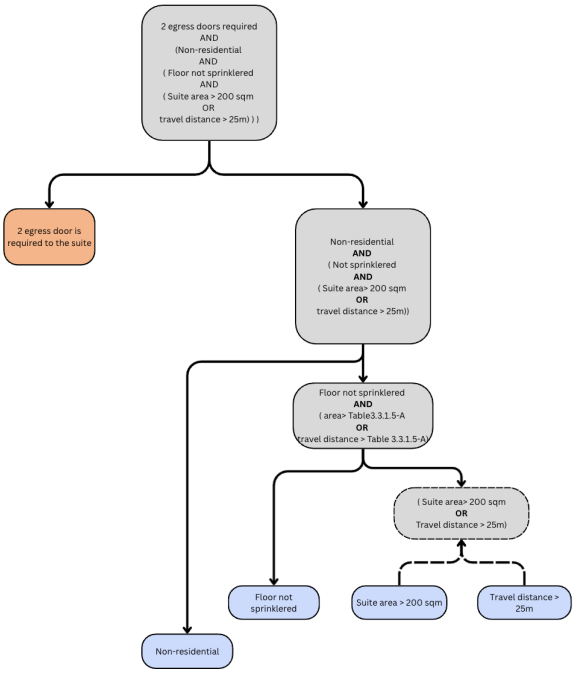


Figure 4: An example of $O_{R,\tau}$

Figure 4 shows how $O_{R,\tau}$ is created based on the conditions in τ. Blue-colored types are the conditions, and the orange-colored type is the type related to a compliance topic of τ. Gray types are pushouts and pullbacks. At the top level, we obtain a type representing the complete triggering condition for the requirement expressed in τ. Note that here labels such as "travel distance > 25m" are used as shorthand for condition-types, namely the type of travel distances whose value exceeds the indicated threshold.

For each orange- or blue-colored type in $O_{R,\tau}$, there is a corresponding project parameter, which may carry implicit typing information and domain constraints. For example, suite area and travel distance are modeled as non-negative real numbers. Some parameters also encode the suite's major occupancy classification under the BCBC, such as Groups A-1, A-2, A-3, A-4, B, C, D, E, F-1, F-2, and F-3. Here, Group C denotes residential occupancy, so a non-residential suite is one whose occupancy classification is not C.

Figure 5 shows how this implicit typing and domain constraints can expand a type in $O_{R,\tau}$ by making explicit the underlying semantic structure required for this interpretation, including the quantity type, value domain, and unit. **How $O_{R,\tau}$ works**: Given those parameters, compliance is checked by determining whether the relevant populated instance satisfies $O_{R,\tau}$. If the applicability conditions are satisfied but the required consequence is not, then the project fails to satisfy $O_{R,\tau}$ and is non-compliant.

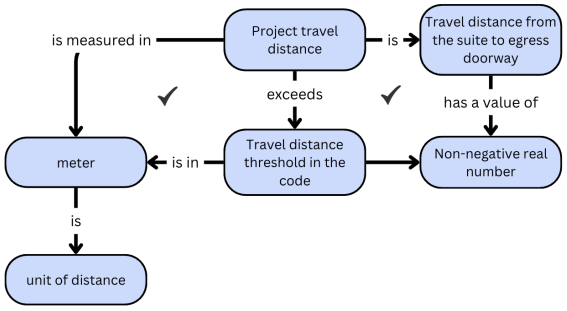


Figure 5: Expansion of “Travel distance $>$ 25m"

5 Conclusion and Future Work

The work described herein has been supported by a set of spike prototypes offering proof-of-concept implementations for particular aspects of the approach. We seek to build on the success of that work to turn this into a cohesive working algorithm that would allow a builder to easily query the building code. We remark here on several lines of development that we are exploring.

This work exhibits a cross-cutting need for categorical support. To address that, we are evaluating the tradeoffs between building atop two platforms from the Topos Institute. The first option would leverage Catlab's excellent support for computing using Attributed C-Sets. This option would employ the classic 1-categorical implementation of ologs laid out in [7], including the presence of attributes valued in Julia types. An alternative option would instead build atop the more recent CatCoLab, and its support for double theories, and models thereof using lax double functors. While CatCoLab's existing support for ologs is 1-categorical in character, this option would leave open the potential of eventually drawing on the double-categorical approach to ologs seen in [3]. That double-categorical approach is notable for providing support for data instances valued in relations (and thus offering not just functions, but also relations, between sets), for cartesian equipment structure that supports data operations, and—via tabulators—for encoding propositions as types. However, support for this powerful approach to double-categorical ologs would entail substantial extensions to CatCoLab.

A distinct implementation avenue for this work relates to the ability of Satisfiability Modulo Theories (SMTs) to assess the satisfiability of complex logical constraints, and to identify the values of free variables that successfully meet the satisfiability criteria. Once the framework described above identifies the conditions relevant to a query, and characterizes them logically using the constructions above, many users will likely seek to understand if their particular building aims satisfy those criteria. To the degree that user aims exhibit flexibility, they may further have interest in understanding which, if any, variations of their visions for a project would successfully comply with the relevant building code in place. SMTs offer a well-explored and performance-optimized approach for reasoning about satisfiability of logical constraints, including in conducting searches to identify and report particular assignments of values to variables that would successfully meet such constraint. We have successfully explored at a preliminary level the use of the popular Z3 solver for reasoning about the sort of logical constraints that arise in areas of the building code. Despite some important limitations—including those involving the handling of real-valued attributes—the prototyping suggests great potential for uses of SMT solvers to support the user interface. That potential is further underscored by the ready availability of rich libraries for some SMT solvers, including Z3.

This work has additionally explored use of another important class of contemporary technologies: LLMs. While hallucinations and other limits render LLMs unreliable for reasoning concerning building codes, they offer a powerful tool for facilitating the translation of voluminous building codes into the logical structures of the rule semantics layer (explored in Section 3.3), and aiding in updating such structures in light of incremental building code updates. The building codes for many Canadian jurisdiction are massive, commonly running over 2000 pages. Our experiments suggest that LLMs do offer considerable utility in offering a rough, first-cut encoding of the structure of building code text into the logical structures described above.

References

[1] Canadian Commission on Building and Fire Codes. National building code – 2023 alberta edition, volume 2. Technical Report Report Number-1234, National Research Council of Canada, Canada, 2023.

[2] Shaoxiong Ji, Shirui Pan, Erik Cambria, Pekka Marttinen, and Philip S. Yu. A survey on knowledge graphs: Representation, acquisition, and applications. *IEEE Transactions on Neural Networks and Learning Systems*, 33(2):494–514, 2022.

[3] Michael Lambert and Evan Patterson. Representing knowledge and querying data using double-functorial semantics. *arXiv preprint arXiv:2403.19884*, 2024.

[4] Varun Magesh, Faiz Surani, Matthew Dahl, Mirac Suzgun, Christopher D. Manning, and Daniel E. Ho. Hallucination-free? assessing the reliability of leading ai legal research tools. *Journal of Empirical Legal Studies*, 22(2):216–242, 2025.

[5] J. Peng and X. Liu. Automated code compliance checking research based on bim and knowledge graph. *Scientific Reports*, 13:7065, 2023.

[6] David I. Spivak. *Category Theory for the Sciences*. MIT Press, Cambridge, Massachusetts, 2014.

[7] David I. Spivak and Robert E. Kent. Ologs: A categorical framework for knowledge representation. *PLOS ONE*, 7(1):1–1, 01 2012.

[8] Summayah Waseem. Automating ontology-based construction code compliance with bim and linked building data (lbd). Master's thesis, California State University, Northridge, Northridge, January 2025. Statement of responsibility: by Summayah Waseem; Committee members: Jeffrey Wiegley and Maged Elaasar.