Knowledge Representation and Reasoning with Domain Information System (DIS)

KNOWLEDGE REPRESENTATION AND REASONING WITH DOMAIN INFORMATION SYSTEM (DIS)

BY
ALICIA MARINACHE[1], M.A.Sc. (Software Engineering)
McMaster University, Hamilton, ON, Canada

A THESIS
SUBMITTED TO THE DEPARTMENT OF COMPUTING AND SOFTWARE
AND THE SCHOOL OF GRADUATE STUDIES
OF MCMASTER UNIVERSITY
IN PARTIAL FULFILMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY IN ENGINEERING

Doctor of Philosophy in Engineering(2025) McMaster University
(Computing and Software) Hamilton, Ontario, Canada

TITLE:	Knowledge Representation and Reasoning with Domain Information System (DIS)
AUTHOR:	Alicia Marinache[3] M.A.Sc. (Software Engineering) McMaster University, Hamilton, ON, Canada
CO-SUPERVISORS:	Dr. Ridha Khedri, McMaster Univeristy Dr. Wendy MacCaull, St Francis Xavier University
NUMBER OF PAGES:	xi, 258

To Florin, for his continuous support
To William, for keeping me on my toes and letting me see the world through his eyes
To my family and friends, for keeping me sane

Abstract

Ontology engineering lacks a systematic, data-driven methodology, often requiring manual, ad hoc processes that struggle to integrate structured datasets with conceptual domain knowledge. Traditional approaches, particularly those based on Description Logic (DL), prioritise top-down taxonomic modelling, making it difficult to align with structured data sources that follow different relational paradigms. This disconnect leads to complex mapping efforts and possible semantic inconsistencies. To address these challenges, we propose a data-driven methodology grounded in the Domain Information System (DIS) formalism, and designed to align domain conceptualisation with existing structured datasets from the outset.

Acknowledgements

I am deeply grateful to my co-supervisors, Dr. Ridha Khedri and Dr. Wendy MacCaul, whose unwavering support, guidance, and patience throughout my graduate studies have been instrumental in the completion of this work. Their belief in me, even during the most difficult phases, made this journey possible.

I extend my sincere thanks to Dr. Andrew LeClair for the many thoughtful conversations, critical insights, and encouragement. His ability to challenge my thinking while offering steadfast support provided both clarity and momentum when I needed them most.

To my family, thank you. To my husband, for embracing the significance of my return to academia and walking beside me with constant encouragement; to my son, whose humour and sharp perspective brought joy and balance to this experience; and to my extended family and friends, who never failed to lift me up with kindness, laughter, and support.

This would not have been possible without you - thank you for being there every step of the way.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The process of extracting meaningful information from the available data and then generating new knowledge from it is known as Knowledge Representation and Reasoning (KRR) (Kendall and McGuinness, 2019). For most organisations, the KRR process is a necessary, albeit time-consuming operation. Currently, there are large volumes of data stored in various sources, formats, and with varying degrees of consistency. The speed with which data is flowing is becoming a challenge in various fields such as astronomy (AstroML, 2021; Zhang and Zhao, 2015; Nativi *et al.*, 2015) and marine research (Baumann *et al.*, 2016; Leadbetter *et al.*, 2016; Nativi *et al.*, 2015), where a large volume of sensor data is constantly being recorded. Thus, the KRR process should be automated and timely, and should enable a clear and systematic methodology to build knowledge systems, i.e., the datasets, each of their application domains, and the relationships between them (Kendall and McGuinness, 2019).

Ontologies, described by Guarino *et al.* (2009) as "explicit specifications of a conceptualisation of a domain", have been used as a means to represent the domain of application of knowledge systems. The *conceptualisation* refers to the process of abstracting the domain, by identifying its relevant concepts, along with the relations between them (Kendall and McGuinness, 2019). The *explicit* refers to the process of ensuring that the concepts identified are clearly defined. In general, we subscribe to Gruber's intuitive understanding of an ontology, therefore, in our work, the terms *ontology*, *knowledge system*, and *knowledge representation (of a domain)* are interchangeable.

Ontology engineering facilitates the discovery of implicit knowledge that is not explicitly represented, and is rather embedded within the conceptual domain. Ontology engineering plays a crucial role in KRR, providing structured, machine-readable models of domain knowledge. This process is crucial because it allows for a deeper

understanding of the domain. Thus, it becomes paramount that the ontology is well designed and easily understood by its users and that it offers support for effective reasoning engines. The field of ontology engineering has only recently seen the development of methodological frameworks for ontology engineering (Tudorache, 2020). Among the most prominent formal approaches in this space is DL, which has emerged as the primary formalism for representing domain knowledge (Le Clair *et al.*, 2022). A typical DL ontology consists of three components: a Terminological Box (T-Box), which defines concepts and their hierarchical relations, an Assertional Box (A-Box), which contains assertions about individual instances, and a Rule Box (R-Box), which describes relations between individuals and concepts, as well as properties of these relations. Over the years, a rich ecosystem of DL fragments and optimised reasoning engines has emerged, each tailored to specific reasoning needs, ranging from efficient T-Box reasoning for large conceptual hierarchies (Baader *et al.*, 2022), to scalability in A-Box reasoning for ontologies with numerous individual instances (Tahrat *et al.*, 2020), and handling complex R-Box relations and rules (Jackermeier *et al.*, 2023).

1.1 Motivation

While DL provides a well-defined and logically robust foundation for ontology representation, certain practical limitations emerge in data-centric settings. In our work, we focus on two issues that arise in the current landscape of ontology engineering: the lack of systematic, semi-automated methodology for ontology engineering, and the disconnect between ontologies and structured data sources (Kotis *et al.*, 2020; De Giacomo *et al.*, 2018).

Building knowledge systems remains largely ad hoc, following no clear process (Kotis *et al.*, 2020). Recently, we have seen the development of methodological frameworks for ontology engineering, which we discuss in more detail in Section 2.2.2. Existing ontology engineering methodologies are manual, often employing a top-down conceptual modelling strategy. This approach requires domain experts to define conceptual structures first and map data to them later, often resulting in misalignment (De Giacomo *et al.*, 2018).

DL-based and other traditional ontology engineering approaches focus on abstracting domain knowledge through T-Box conceptualisation. In this approach, the foundational relation `isA` is used to define the hierarchical subsumption (e.g., "a sparrow `isA` a bird"). This taxonomic approach emphasises classification and specialisation, i.e., grouping entities under broader categories based on shared properties. In contrast, relational datasets are governed by the `partOf` relation, used to define component relations (e.g., "a wing is `partOf` a bird"). This mereological approach emphasises

the compositional structure. Mapping structured datasets to taxonomic ontologies requires reconciling fundamentally different relational paradigms, often through intermediate representations or transformations (De Giacomo *et al.*, 2018).

Without an integrated approach that incorporates structured data from the outset, ontologies risk being detached from real-world datasets and becoming difficult to maintain and scale. These observations do not detract from the strengths of DL, instead motivating the need for complementary approaches that place structured data at the core of the modelling process.

In our work, we focus on structured, organised data, represented as a set of tuples. We believe that this approach is sufficient, since organisations are in the process of converting their unstructured data into a more structured format (de Haan *et al.*, 2024; Hong, 2016; Najafabadi *et al.*, 2015). While unstructured data currently accounts for up to 90% of enterprise information, it is often underutilised, with only a fraction of organisations able to analyse it effectively. As a result, considerable effort is directed toward extracting structured features from unstructured sources, such as metadata, summaries, and sentiment scores, to make them suitable for downstream analysis de Haan *et al.* (2024). This transformation process reflects an industry-wide shift toward structuring unstructured data, reinforcing our decision to build our research around structured data as a practical and tractable modelling choice.

These limitations in data-driven integration and formalisation motivate the need for a new formalism, one that places structured data at the core of the modelling process, while remaining amenable to rigorous reasoning. This motivates the central question of this thesis.

It is our belief that the right approach is to provide a clear (semi-)automated process for ontology engineering. The specification of domain-knowledge representation needs to employ a modular and adaptable structure. To address the challenges of mapping data to existing ontologies, reduce the complexity of integration, and maintain consistency between domain knowledge and data, ontology engineering must follow a systematic process that integrates directly with structured datasets. This research builds on DIS (Marinache, 2016), a previously introduced formal specification for knowledge systems, by formalising its syntax and semantics, developing a methodology for the construction of a DIS guided by existing data, and enabling reasoning through its implementation in a higher order logic setting.

1.2 Problem Statement

This thesis aims to advance the field of ontology engineering by developing a clear, formal methodology for constructing knowledge systems that are tightly aligned with structured data. Specifically, it formalises the DIS, a framework that embodies theories of data, information, and domain knowledge through mathematical structures. The goal is not only to define a repeatable and partially automated process for ontology construction, but also to provide a foundation that supports reasoning tasks, such as classification and consistency checking.

In particular, this thesis aims to make formal ontology engineering more accessible by abstracting away the underlying mathematical complexity of DIS. Through the use of formal semantics and automated reasoning, DIS hides these foundational structures from the ontology designer, while preserving the rigour needed for verifiable reasoning. By allowing for seamless alignment of existing domain data with domain-level conceptual views, DIS facilitates the systematic generation, evolution, and verification of knowledge in a variety of application domains.

1.3 Objectives and Methodology

This section defines the three main objectives and outlines the methodology used to address them. The objectives of the thesis reflect the need to formalise this framework, evaluate its theoretical foundations, and demonstrate its practical capabilities through implementation and reasoning tasks. The work progresses through three phases: engineering process design, formal specification, and reasoning.

Objective 1: Develop a systematic ontology engineering methodology

The first objective focuses on designing a process that enables the construction of ontologies from structured datasets and expert domain knowledge. This process draws on principles of modularity, separation of concerns, and formal specification. Existing ontology engineering methods are reviewed to identify limitations and extract reusable design principles. These principles are then applied to the development of a representative case study that illustrates the construction of a DIS instance from real-world data.

To improve accessibility and reproducibility, the process incorporates partial automation. A set of generic templates is introduced to formalise recurring structures in DIS specifications. This approach reduces the effort of manual specification while preserving semantic correctness. Together, these elements define a structured approach to

engineering ontologies from data-rich sources.

Objective 2: Define the semantics and formal specification of the DIS theory

The second objective establishes the formal foundations of DIS by defining its syntax, semantics, and implementation. DIS integrates two types of formal knowledge: a data theory grounded in cylindric algebra, and a conceptual theory based on Boolean lattices and graph structures. These components are linked through a mapping mechanism that aligns structured data with abstract concepts. The resulting system provides a unified view of information at both the syntactic and semantic levels.

This theoretical foundation is formalised in a higher-order logic proof assistant that supports modular specification and machine-checked verification, making it suitable for encoding and validating the algebraic and relational components of DIS. The formal specification serves as a foundation for correctness, modularity, and extensibility across domains.

Objective 3: Explore reasoning capabilities within the DIS framework

The third objective investigates whether standard reasoning tasks, such as classification, satisfiability, and subsumption, can be carried out within the DIS framework. These tasks are defined over the formal structures established in the previous objective and implemented in generic Higher-Order Logic proof assistant Isabelle (Isabelle/HOL). Reasoning is performed at both the data and conceptual levels, demonstrating the framework's support for semantically coherent inference.

This objective evaluates the expressivity and tractability of DIS in a formal reasoning setting. By applying reasoning tasks to DIS specifications and verifying their correctness through interactive theorem proving, the framework is shown to support practical and formally sound knowledge inference over structured data.

1.4 Contributions

This section shows that the objectives outlined in Section 1.3 are achieved, leading to the following contributions:

1.4.1 Formalised the syntax and semantics of the DIS framework as a unified system with algebraic and relational foundations (Chapter 4)

This thesis formalises the previously proposed DIS by defining its core syntax and semantics as two unified and interrelated algebraic theories: a data theory for structured datasets and a domain representation theory for conceptual abstractions. The *data theory*, referred to as the Domain Data View (DDV) is built over structured datasets and formalised using the language of cylindric algebra. It supports data sorts, the universe of discourse, and operators that support data-level abstraction and manipulation. The *domain representation theory*, referred to as the Domain Ontology (DOnt), captures the minimal conceptual structure needed to interpret a dataset. It is modelled as a Boolean lattice built over atomic concepts derived from data attributes, and it is enriched with a monoid of concepts and a family of rooted graphs, allowing for modular and structured conceptual modelling.

These two theories are formally integrated through a mapping operator that aligns data and ontological structure, ensuring that all reasoning is grounded in the relevant data context. The definition of *datascape concepts* provides a formal mechanism to describe the conceptual footprint of the data. Together, these components form a unified formalism with algebraic and relational foundations that links domain knowledge with its associated domain data in a semantically transparent and context-aware manner.

Unlike traditional knowledge systems, which often assume a fixed ontology and require complex mappings to align with data (Xiao *et al.*, 2018), DIS offers a fully integrated structure where the domain layer is based on the data layer, and the development of both layers is guided by the existing structured data, using compatible formal theories. This enables users to define knowledge systems that are both modular and data-grounded from the start, with a guaranteed conceptual alignment that eliminates mismatches between the domain data and domain knowledge theories, making the system immediately usable and adaptable without deep logical expertise.

1.4.2 Proposed a methodology to generate the theory necessary to reason on a given data set and its domain knowledge (Section 4.2)

This research introduces a methodology for constructing a DIS instance from structured datasets, enabling the systematic generation of the data theory, the domain representation theory, and their formal integration. The process demonstrates the

ability of the DIS to support and integrate multiple domains of application, and to adapt to evolving information.

The methodology is guided by formal design principles derived from the well-formed underlying theory, ensuring that the resulting specifications represent both the data and its conceptual context in a structured, modular, and semantically aligned manner.

Unlike most existing frameworks, where ontologies are constructed manually and semantic constraints are validated afterward, each DIS instance is a model of the formally defined DIS theory, inheriting correctness, modularity, and semantic alignment by design. While DL-based reasoning tools validate logical entailments within a fixed ontology, they do not ensure that the ontology itself conforms to a higher-order semantic specification (Baader *et al.*, 2003), as such constraints are typically defined separately from the data and not structurally enforced during construction. In contrast, the DIS framework guarantees that every instance is both semantically coherent and data-aligned from the outset. For users, this enables the generation of tailored ontologies that reflect their datasets directly, without requiring further mapping or reconciliation. Furthermore, because rooted graphs in DIS are constructed over data-derived concepts, the domain knowledge includes only those concepts that are grounded in actual data, ensuring conceptual relevance and eliminating speculative modelling.

1.4.3 Encoded DIS theory within Isabelle/HOL (Chapter 5)

The DIS framework is fully formalised in the Isabelle/HOL proof assistant. Each component is implemented using Isabelle's locale system, and leveraging its rich type-theoretic and algebraic foundation. This approach enables the modular construction of DIS instances, higher-order abstraction, and machine-checked reasoning of both structural and semantic properties. Proofs are developed interactively, combining user-guided proof strategies with the automated tactics in Isabelle to balance rigour and tractability.

While many KRR frameworks provide automated reasoning capabilities, they are typically specified in First Order Logic (FOL) and often rely on specialised fragments and custom reasoners with fixed inference strategies (Baader *et al.*, 2003). DIS is embedded directly in a formal proof assistant, giving users the ability to define and verify system behaviour down to the logical level. This integration ensures that correctness properties are not assumed but proven, allowing users to build domain-specific knowledge systems with full traceability and confidence, especially in domains where verification is essential.

1.4.4 Proposed templates that ease and automate the process of building a DIS instance (Chapter 6)

The methodology is extended through the implementation of generic templates and a parsing mechanism that automate the generation of DIS instances from structured datasets. These templates significantly reduce the manual specification effort and improve usability for domain experts. The approach supports the automated construction of the core theories for both the data and knowledge representation layers as Isabelle/HOL specifications. Manual input is limited to two tasks: defining the mapping between the data sorts and atomic concepts, and specifying the rooted graphs for the domain ontology. With this minimal involvement of experts, DIS bridges the gap between formal methods and practical application, enabling the development of a scalable and maintainable knowledge system.

In most existing frameworks, ontology construction is a bespoke process, dependent on domain experts, and difficult to scale. DIS changes this by introducing a template-driven mechanism that allows domain experts to generate entire DIS instances from existing data sets. For users, this effectively lowers the entry threshold, enabling them to rapidly construct formal knowledge systems utilising intuitive, declarative templates without requiring comprehension of the underlying logic or proof constructs.

1.4.5 Proposed a reasoning framework in HOL for validating domain-specific conjectures (Chapter 7)

This research explores how standard reasoning tasks, such as satisfiability, classification, and subsumption, can be defined and executed within the DIS formalisation in Isabelle/HOL. It shows that, despite the theoretical undecidability of higher-order logic, these tasks remain tractable and useful in practice for validating structured, domain-specific knowledge.

Reasoning in DL frameworks is grounded in carefully designed fragments of FOL that ensure decidability and tractable inference, particularly over concept hierarchies and instance checking tasks. While this makes DL highly effective for ontology-centric reasoning, it also imposes structural and semantic restrictions that limit expressivity, particularly in contexts requiring alignment with structured data, complex mappings, or domain-specific algebraic semantics.

In contrast, DIS is formalised within a higher order logic framework, not to replace decidable reasoning tasks, but to support a broader class of reasoning scenarios that integrate both data-level and concept-level inference within a unified structure. While

Higher Order Logic (HOL) is undecidable in general, many reasoning tasks within the DIS remain tractable in practice. By leveraging Isabelle/HOL, DIS supports reasoning that is semantically precise, structurally modular, and not limited by the syntactic boundaries of FOL fragments. This enables not only the verification of traditional ontological properties, such as consistency or classification, but also the construction and formal validation of DIS instances. For users, this means they can express complex domain inferences, without needing to refactor their models to fit narrow fragment profiles, and with the assurance that correctness can be verified interactively and incrementally.

1.5 Related Publications

- Marinache, A., Khedri, R., LeClair, A., and MacCaull, W. (2021). DIS: A data-centred knowledge representation formalism. In *2021 Reconciling Data Analytics, Automation, Privacy, and Security: A Big Data Challenge (RDAAPS)*, pages 1–8. IEEE

This paper presents a detailed case study to highlight the features of DIS. Several datasets and their respective domain knowledge conceptualisations are integrated to illustrate reasoning tasks requiring data-grounded and domain-related answers to user queries. The case study highlights the ability of DIS to handle information evolution. The paper addresses two open issues in ontologies: the lack of clear and explicit guidelines for ontology construction and the prohibitive cost of adapting and reusing existing ontologies.

- Marinache, A., Khedri, R., and MacCaull, W. (2025a). Bridging data and knowledge: A roadmap from Domain Information Systems (DIS) Theory to Practical Reasoning. To be Submitted for publication

This paper presents the Isabelle/HOL implementation of the theory of DIS, together with a case study. The main purpose is not to provide an exhaustive reasoning analysis, rather to showcase the reasoning capabilities of a DIS. Reasoning tasks are explored from the perspective of the DIS theory and are accompanied by their Isabelle/HOL implementation. In addition, we briefly describe the (partially) automated process of engineering a DIS, using the DISEL tool (Wang *et al.*, 2022).

- Marinache, A., Khedri, R., and MacCaull, W. (2019). A Data-Centered Framework for Domain Knowledge Representation. Technical report, McMaster University

This technical report introduces the DIS, the formal framework for designing ontologies from structured data on which our work is based. DIS integrates a domain representation and a data view, connected through a mapping operator that aligns individual data elements with corresponding concepts. The approach uses Boolean lattices and rooted graphs for modelling conceptual knowledge, and cylindric algebra for modelling the domain data view. The report also compares DIS with existing approaches to data and ontology evolution, demonstrating its robustness under information change.

- Marinache, A., Khedri, R., and MacCaull, W. (2025b). Domain Information System (DIS): From theory to semantics. Technical Report CAS-25-02-RK, McMaster University

This companion technical report formalises the semantics of the DIS framework by constructing a mathematical interpretation of its core components and proving it satisfies the axioms of the DIS theory. The report rigorously defines the two foundational layers: data (DDV) and conceptual (DOnt), with the mapping operator linking them. This semantic foundation ensures internal coherence and supports formally verified reasoning across data and conceptual layers.

- Marinache, A., Khedri, R., and MacCaull, W. (2025c). Domain Information System (DIS): Specification and automation. Technical Report CAS-25-01-RK, McMaster University

This technical report presents the formal specification and partial automation of the DIS. The report formalises the DIS architecture in Isabelle/HOL, where the data and conceptual layers are specified as distinct theories, linked via the mapping operator. It also introduces a template-driven mechanism for generating DIS instances and demonstrates its applicability through the Wine DIS case study. This work establishes a semi-automated, formally grounded process for building verifiable, data-aligned ontologies.

- Le Clair, A., Marinache, A., El Ghalayini, H., MacCaull, W., and Khedri, R. (2022). A review on ontology modularization techniques: a multi dimensional perspective. *IEEE Transactions on Knowledge and Data Engineering*

This survey examines various techniques for modularising ontologies, regardless of the formalism used in their construction. It includes a discussion on emerging research areas, particularly the development of modularisation techniques that integrate both logical and graphical approaches. The findings of this paper serve as the foundation for the literature review conducted in this thesis.

- LeClair, A., Khedri, R., and Marinache, A. (2019). Toward Measuring Knowledge Loss due to Ontology Modularization. In *Proceedings of the 11th International Joint Conference on Knowledge Discovery, Knowledge Engineering and Knowledge Management - Volume 2: KEOD*, pages 172–184. INSTICC, SciTePress

In this paper, the theory of DIS is used to formalise the view traversal, a graphical modularisation technique, based on the relational structure of its Domain Ontology (DOnt) component. The Boolean lattice underlying the DOnt allows for the formalisation of knowledge loss due to view traversal modularisation. This DIS modularisation technique addresses the challenges associated with a dynamic domain, which is enriched with data and multiple independent agents.

- LeClair, A., Khedri, R., and Marinache, A. (2020). Formalizing graphical modularization approaches for ontologies and the knowledge loss. In *International Joint Conference on Knowledge Discovery, Knowledge Engineering, and Knowledge Management*, pages 388–412. Springer

In this paper, the view traversal modularisation technique is extended with an algebraic approach. Using the fact that a Boolean lattice is isomorphic to a Boolean algebra, this paper introduces the definition of a module in DIS as a principal ideal subalegbra module, showing the similarity between the graphical approach and the algebraic approach in DIS.

1.6 Thesis Outline

The remainder of this thesis is organised as follows.
Chapter 2 surveys existing mathematical approaches to KRR and identifies their limitations.
Chapter 3 introduces the required mathematical background, along with the details of the DIS theory.
Chapter 4 discusses the semantics of the new DIS theory and introduces a detailed case study that highlights the characteristics of DIS.
Chapter 5 presents the DIS specified as a theory in the higher-order logic generic proof assistant Isabelle/HOL.
Chapter 6 introduces automation elements for building a DIS.
Chapter 7 explores standard reasoning tasks and exemplifies how they can be applied on a DIS.
Chapter 8 presents the conclusions of the research and outlines directions for future work based on the results of this work.

Chapter 2

Literature Review

In this chapter, we review the literature for existing approaches on the formalisation of knowledge representation, and on generating knowledge from existing data. In Section 2.1, we explore existing formalisms for knowledge representation. In Section 2.2, we identify design criteria for building better ontologies. In Section 2.3, we present existing approaches to knowledge generation and a summary of existing reasoning support in ontologies. In Section 2.4, we look at the existing higher order logic provers.

2.1 Knowledge Representation Formalisms

Due to the rapid increase in the size and diversity of the generated data, knowledge representation has become increasingly important in a variety of fields, such as data analysis, natural language processing, machine learning, and more (Kendall and McGuinness, 2019). Knowledge can be classified in multiple ways, such as *explicit* (i.e., knowledge that can be expressed, documented, and shared), *implicit* (i.e., knowledge that exists within, and can be inferred from the explicit knowledge), *tacit* (i.e., knowledge that cannot be easily expressed, as it is related to personal experience and beliefs), *situational* (i.e., knowledge about situations in a given context or domain), *procedural* (i.e., knowledge about actions or procedures that are valid in a domain or independent of a domain), *conceptual* (i.e., knowledge about the facts and relations in a domain), or *contextual* (i.e., knowledge pertinent or valid in a context or domain) (De Jong and Ferguson-Hessler, 1996). In this work, we focus on the explicit representation of conceptual knowledge, referred to as domain knowledge, within a given application context. While the literature offers a variety of classification schemes for knowledge representation formalisms (Grimm, 2009; Patel and Jain, 2018), we organise our discussion around three principal categories: relational structures (Section 2.1.1), algebraic structures (Section 2.1.2), and logic-based formalisms (Section2.1.3). These categories were selected because they reflect the

dominant paradigms used in formalising, manipulating, and reasoning about conceptual knowledge in structured systems, and they form the theoretical foundation of the DIS. For each of the structures in these categories, we aim to understand how they capture information and what kind of information they are able to capture (their expressivity), as well as if they can assume an open-world approach. In an open-world approach, what is not known to be true about the domain is simply assumed to be not known. In contrast, in a closed-world approach, what is not known to be true about the domain is assumed to be false (and it is called *negative information*).

2.1.1 Relational Structures

Relational structures are mathematical constructs consisting of a set and a collection of relations defined over that set. When representing knowledge through relational structures, the world is described as a set of concepts and individuals, with relations between them. The relational structures are usually graph-based, with concepts and individuals represented as vertices of the graph, and the relations as its edges. The main advantages of relational structures are their intuitiveness and simplicity. However, in general, graph-based structures capture only binary relations. In addition, reasoning is usually done through procedures that manipulate the structures. Although structural manipulation is easy to understand, it is more limited than logical approaches (as discussed in Section 2.3). In the remainder of this section, we explore three relational structures: formal concept analysis, semantic networks, and conceptual graphs.

Formal Concept Analysis (FCA) (Ganter *et al.*, 2019) is rooted in lattice theory and abstracts conceptual hierarchies from sets of individuals, describing their attributes. A *formal concept* is an ordered pair of the form (*extent*, *intent*). Within the context of FCA, the *extent* consists of a set of individuals that share the attributes of the intent. The *intent* consists of a set of attributes that are common to all individuals of the corresponding extent. FCA provides support for creating ontologies from datasets, using the relationships between attributes and individuals in a given dataset, to create abstract concepts that describe the domain. At the same time, it enables a mathematical characterisation of the hierarchy of concepts, called the concept lattice. The FCA theory has a clear link to relational data structures, in that each dataset may be represented by a concept lattice. However, FCA strictly models the `isA` relation, limiting expressivity to the subsumption reation, and embeds data directly into the lattice structure, which results in a closed-world assumption (CWA) (Ganter *et al.*, 2019).

Semantic Networks (Sowa, 2014) are another example of relational structures. They

represent knowledge in a given domain as directed graphs, where concepts are represented as nodes, and relations as edges. In contrast to FCA, in which only the `isA` relation can be modeled, semantic networks can represent any binary relations. In semantic networks, a main drawback is the lack of a clear definition of generic nodes. Generic nodes represent either abstract concepts, linked to individuals through the `typeOf` relation, or sets of abstract concepts, linked to individuals through the `memberOf` relation. Despite their name, semantic networks lack formal semantics (Lenzerini *et al.*, 2004). There is no distinction made between different kinds of edge in the graph, edges can represent either properties of individuals and concepts, or relation between concepts (as classes of individuals) and individuals. The need to be more expressive, while using semantic networks-like formalisms, led to the development of DL, which we describe in Section 2.1.3.

Conceptual Graphs (CG) (Van Harmelen *et al.*, 2008) are graph representations of natural language semantics, based on the semantic networks. They contain two types of nodes: concepts and conceptual relations. The directed edge links connect a conceptual relation to its input and output concept. In contrast to FCA and semantic networks, conceptual graphs can represent n-ary relations. In conceptual graphs there are no edges that can connect concepts directly to concepts (or relations to relations). The formalism has the same expressive power as FOL, meaning that some reasoning tasks, including validity and subsumption, are undecidable (Van Harmelen *et al.*, 2008).

2.1.2 Algebraic Structures

Algebraic structures are mathematical structures consisting of a set and a collection of operators defined on it. One of the advantages of using algebraic structures to represent knowledge is the fact that most of the structures are theoretically mature, i.e., they have existed for a long time, and there is strong support for their usage. Capturing the domain knowledge through structures that are well established enables the use (or allows for the extension) of automated theorem provers for reasoning on these structures. In this section, we look in more detail at some algebraic structures that capture and represent domain knowledge, such as algebras of relations, and information algebra.

The development of algebras of relations as formal systems for manipulating relations came as a natural extension of relational structures. In the algebras of relations, the elements of the carrier set are represented as relations. In addition to the advantage described above, these algebras offer a high-level description of the data and enable the development of rules for rewriting, querying, and optimising low-level access. We

review two types of algebra of relations (Hirsch, 2007; Sayed Ahmed, 2022): Relational Algebra (RA)s and Cylindric Algebra (CA). RA underlies the semantics of relational databases and supports operations such as selection, projection, and join over n-ary relations. In contrast, CAs were introduced by Tarski *et al.* (1971) to capture the semantics of first-order logic, and model relations along with indexed variables and quantification, enabling reasoning about logical structure. While both frameworks operate over n-ary relations, CAs are more expressive, and are more suitable for representing higher-order relational systems.

RAs represent uniform data, assuming a closed-world. CAs, through the cylindrification operator, offer support for nonuniform structured data, and assume an open-world. Both kinds of algebra offer support for representing object-attribute relations (i.e., the structured data in the domain of knowledge). At the same time, neither structure offers support to represent other relations, such as the abstract notion of concepts (defined as classes of objects) or the non-structural relations between concepts. By non-structural relations, we understand relations not given by the structured data (as defined in Section 3.3.2), such as complex definitions of concepts. For example, the statements "concept Person *hasA* attributes Name and Age" and "individual John *isA* (concept) Person" describe structural relations. In contrast, the statement "A (concept) Toddler is a Person whose age is in between 2 and 4" is a non-structural relation.

Information Algebra (IA) (Kohlas and Schmid, 2014) proposes a theoretical foundation for information processing. It is a mathematical structure built on well-established theories: a lattice of frames, which provides the structure and context of organising information, and a semigroup of information, which represents the individual data associated with specific questions within those frames. Each frame contains a set of specific questions that are related to a particular piece of information within the domain. The questions are organised in a partial order, based on their level of detail, called *granularity*. This hierarchical structure helps with combining the information pieces or projecting them to different levels of granularity. IAs are not concerned with the internal structure of a frame, but only with its granularity. While IAs can be viewed as a generalisation of RAs, by extending them to support more abstract operations such as combination and projection over general information domains, they introduce certain limitations. In particular, IAs represent relations primarily through a partial ordering that reflects the granularity of information frames, and do not capture richer relational structures within the domain. Furthermore, IAs operate under a CWA, which can limit their flexibility when modelling open or evolving data contexts.

2.1.3 Logic-Based Formalisms

Logic-based formalisms are mathematical frameworks consisting of a formal language, a set of axioms, and inference rules. They are designed to represent information by capturing the structure and semantics of a given domain in a precise and clear manner, typically through the use of symbolic logic. In doing so, logic-based formalisms offer support for automated reasoning in information systems. In addition, as discussed in Sections 2.1.1 and 2.1.2, most of the relational and algebraic structures lack the ability to represent non-structural relations within the domain of application. This shortcoming is addressed by the use of logic-based languages. To illustrate key approaches to logic-based knowledge representation, we focus on formalisms that exemplify complementary design goals, such as object-oriented structuring, interoperability across systems, and ontology-centred modelling. Specifically, we examine Frame Logic (F-Logic), Knowledge Interchange Format (KIF), and DL. These three formalisms have been influential in both foundational and applied settings, and together they span a diverse range of perspectives on how knowledge can be formally specified, exchanged, and reasoned over.

F-Logic (Angele *et al.*, 2009) is a logic-based formalism that introduces object-oriented features such as objects, inheritance, and meta-level properties. This approach provides a flexible and expressive framework for modelling knowledge. In F-Logic, concepts, relations, and individuals are modelled as terms. Frames, which are fundamental data structures in F-Logic, encapsulate generic situations within a specific domain of application. Frames organise knowledge hierarchically, facilitating efficient reasoning and taxonomy management. Frame-based formalisms, of which F-Logic is a member, use these constructs to organise frames systematically, enabling effective knowledge representation and reasoning (Bhatia *et al.*, 2019). F-Logic provides a concise and straightforward syntax coupled with well-defined semantics, enabling intuitive reasoning about objects and their properties (Bhatia *et al.*, 2019). In addition to specifying the relations between concepts and/or individuals, F-Logic offers support to define general rules. Such rules may be used to represent the declarative knowledge from the facts captured in the knowledge system (Ekaputra *et al.*, 2017). Thus, most ontological constructs can be directly mapped to F-Logic. This makes the underlying language of F-Logic more expressive than some World Wide Web Consortium (W3C) (W3C, 2025) standard-based ontology languages, such as Resource Description Framework (RDF), Resource Description Framework Schema (RDFS), Web Ontology Language (OWL), or SPARQL Protocol and RDF Query Language (SPARQL) (Ekaputra *et al.*, 2017).

KIF (Genesereth *et al.*, 1992) is a FOL-based formalism that provides a standardised syntax for representing knowledge, thus enabling the sharing of knowledge between

different systems and tools. The knowledge engineers can specify the ontology in the system that is most appropriate for their needs, along with a translator to and from KIF. This approach allows the knowledge engineers to share knowledge across systems. In addition, KIF can be used for directly representing the domain knowledge. KIF describes the domain using four types of constants: object, function, relation, and logical constants. Syntactically, there is no difference between these types of constants, and they can be used anywhere the language syntax allows the use of constants. Thus, in contrast to other FOL-based languages, the underlying KIF language allows meta-statements, by using formulae as terms in other formulae, a process known as *reification*. However, this flexibility can lead to ambiguities in interpretation. Different implementations might handle the same KIF expression in slightly different ways, which can result in inconsistencies when sharing knowledge between systems. Based on FOL, KIF inherits most of the expressive power of FOL. Thus, due to the undecidability of FOL, KIF offers limited support for fully automated reasoning (Schneider and Šimkus, 2020).

DL (Baader *et al.*, 2003) is a family of knowledge representation formalisms based on FOL, providing a formal framework for knowledge engineering and reasoning about a domain of application. While FOL is undecidable, DL fragments are predicate-based decidable FOL fragments in which concepts are represented as unary predicates, relations as binary predicates, and individuals as terms. In DL, concepts denote sets of individuals. While binary operators (such as join $\sqcup$) are used to construct complex concepts, the concepts constructed in such a manner are not considered binary predicates (Krötzsch *et al.*, 2012). An ontology built using DL is generally comprised of two components: the T-Box, which defines the concepts and roles (relations between concepts) within the domain, and the A-Box, which contains assertions about individuals, including their membership in concepts and their participation in role relations. The information in an A-Box is usually viewed as being incomplete, thus DL reasoners use the open-world assumption.

In the past decade, DL is regarded as the standard in knowledge representation, mainly due to its formal foundation and the balance it strikes between expressivity and decidability. This has led to the development of numerous DL fragments (Le Clair *et al.*, 2022). The main disadvantage of knowledge systems based on DL is their monolithic structure: the data is captured inside the A-Box, giving it a static aspect in a dynamic world. Traditional DL reasoning algorithms might encounter efficiency issues when handling large volumes of instance data in the A-Box, which can impact the scalability and performance of the reasoning process (Westhofen *et al.*, 2022). As A-Boxs grow larger and more complex, reasoning tasks, such as classification or instance retrieval, can become become increasingly computationally expensive and

time-consuming. While research continues to address these challenges, it often focuses on defining new DL fragments tailored to specific reasoning scenarios, rather than providing general-purpose solutions (Zombori, 2008; Lukácsy and Szeredi, 2009). Another common drawback of logic-based formalisms is the inherent trade-off between how richly a language can express a domain and how computationally difficult it is to reason over representations built with that language (Van Harmelen *et al.*, 2008; Matentzoglu *et al.*, 2015). This has led to the current state of the DL-based language family, where various applications and situations prompt the creation of new languages tailored to specific parameters and optimised reasoning engines. This situation has the advantage of tailoring the reasoning for more expressive and scalable fragments, an approach that provides flexibility and customised solutions. From a usability perspective, however, handling multiple DL fragments may result in increased complexity in system design and usage, challenges with interoperability, and higher maintenance overhead. This requires careful consideration of the balance between expressiveness, scalability, and usability. While glsdl offers a clean semantic foundation, it often lacks support for modular development. Once an ontology is constructed, modifications to the concept structure or data schema tend to be global, requiring significant effort to preserve consistency. This presents a challenge for systems that require frequent updates or flexible reuse of components.

Formal representational power alone is insufficient. Knowledge representation formalisms provide the theoretical backbone for representing both data and conceptual structures, and their effective application requires systematic design methodologies. The next section surveys design criteria and engineering perspectives in ontology development.

2.2 Design Perspectives on Knowledge Management

The challenges mentioned in Section 2.1 point to a broader design need: knowledge systems must support modularity not only at the conceptual level but also in their underlying representations and reasoning infrastructure. The field of knowledge engineering has evolved to meet the demand for more effective and efficient information systems, with a significant focus on improving their underlying Knowledge Base (KB) (Mizoguchi, 2019). Part of the research in this field is focused on establishing methodologies and frameworks to support knowledge management, with its various operations on ontologies, such as building, sharing, maintaining, and extending information systems. Although some researchers and domain experts have looked at these tasks from an immediate, short-term perspective, others have approached the engineering aspect of ontologies in a more formal manner. However, the field remains as fragmented as the diverse array of DLs it employs. Researchers often develop

specific methodologies tailored to particular projects, leading to a lack of consensus on standardised foundational practices for ontology design and development (Tudorache, 2020). Furthermore, despite the increasing importance of collaboration in many domains, ontology engineering methodologies have predominantly emphasised non-collaborative approaches. This focus seems counterintuitive in an era where collaborative efforts are crucial to integrating diverse sources of expertise and knowledge (Sattar *et al.*, 2020).

In our work, we adopt an engineering perspective, treating the development of knowledge systems as a systematic, repeatable process that can be (partially) automated. This section is dedicated to identifying the methods and principles that enable the development of information systems to become an engineering activity. In Section 2.2.1, we review the existing literature in order to identify design criteria used in ontology engineering. In Section 2.2.2, we explore existing methodologies for ontology development.

2.2.1 Design Criteria for Ontology Engineering

In a seminal work by Gruber (Gruber, 1995), we find the first mention of design criteria for ontologies, such as *clarity, coherence, extendibility, minimal encoding bias,* and *minimal ontological commitment*. *Clarity* refers to the ability of the ontology to effectively communicate the meaning of its terms, by providing objective, correct, and complete definitions for the ontological terms. The property of *coherence* is defined in terms of consistency, i.e., the axioms of the ontology are logically consistent. This requires that the inferences or logical conclusions derived from the ontology are consistent with the ontological definitions. *Extendibility* refers to the ability to add new definitions and axioms to the ontology without having to change existing ones. *Minimal encoding bias* speaks about specifying the conceptualisation independently of any particular notation, language, or implementation. Finally, *minimal ontological commitment* refers to making as few ontological commitments (definitions, axioms, etc.) as possible about the modelled world.

Madni *et al.* (2001) build an ontology based on the following four design principles: *neutrality, extensibility, complementarity,* and *interoperability*. *Neutrality* is defined as notation- and implementation-independent, corresponding to Gruber's *minimal encoding bias* criteria. *Extensibility* refers to the ability to easily extend the ontology to various application domains, as well as its compatibility with other relevant ontologies in the same domain. Extensibility and Gruber's extendibility are related, as both refer to the ability of the ontology to be extended. *Complementarity* refers to the the observation that multiple perspectives are needed to model various aspects of the

world, and the ontology should allow for this variety of perspectives. *Interoperability* is taken in the general engineering sense, as the ability to easily integrate with other domain ontologies. These last two design principles have no direct correspondence in Gruber's list of design criteria.

By the early 2000s, the field has developed a wealth of ontology examples, yet (Stojanovic, 2004), highlights the inefficiencies and potential errors in constructing ontologies from scratch. To address these challenges, the concept of *modularisation* emerges, advocating that ontologies are built from smaller, well-defined modules that can be managed independently. The criteria considered for the modularisation process are borrowed from the software engineering field, and include *cohesion* and *coupling*. *Cohesion* refers to the (usually semantic) similarities between the concepts in one module, and *coupling* refers to the degree of interdependence between different modules or components within an ontology. Just as in software engineering, ontology modularisation aims to provide high cohesion (i.e., stronger similarity between the concepts of a module) and low coupling (i.e., fewer relations connecting concepts between modules).

Mizoguchi (2003) remark that an ontology is similar to the notion of class hierarchy from the Object-Oriented (OO) paradigm. From that, it is immediate that the process of designing ontologies should employ software engineering design principles, such as separation of concerns, and low coupling. In addition, they mention representation-independence as a key property for ontologies, along with semantic interoperability, or the ability to interpret and translate the metadata used in the semantic web. Thus, an ontology is regarded as "a theory of content", enabling the domain experts to share knowledge across application domains.

According to (Burton-Jones *et al.*, 2005), an ontology can be evaluated with respect to four qualities: *syntactic, semantic, pragmatic,* and *social*. *Syntactic* quality measures the formal style an ontology is written in. *Semantic* quality refers to the correctness of an ontology, or the absence of contradictory statements. The *pragmatic* quality refers to the usefulness of the ontology. The *social* quality refers to the strength of its connections to other ontologies and domains, such as the number of links made to it, and the number of times it is accessed. The first two ontology evaluation criteria correspond to two of the criteria defined by Gruber in (Gruber, 1995), clarity and coherence.

Smith (2006) presents a number of principles that should guide the design of a "good ontology." These principles are based on the ISO Standard 15926 - Integration of life-cycle data for process plants including oil and gas production facilities (ISO 15926).

Some of the more important principles are: the principle of *intelligibility*, which can be directly linked to Gruber's clarity quality; and the principle of *reusing* available resources, which can be linked to the modularisation property. In addition, the principle of *terminological coherence* states that, in a "good ontology", a concept cannot have two semantically different definitions. Finally, the principle of *non-circularity* states that, in a "good ontology", there should be made a distinction between defined and primitive terms, and that a "good ontology" should not allow circular definitions.

Finally, Sattar *et al.* (2020) provide a list of sixteen criteria that an ontology engineering method is evaluated upon. Some of these criteria have roots in engineering processes, such as reusability, maintainability, extensibility, documentation support, and merging and modularisation. Others are strictly related to the process of knowledge engineering and ontology building, such as conceptualisation and instantiation.

2.2.2 Methodologies for Ontology Engineering

Some of the early methodologies, such as METHONTOLOGY (Fernández-López *et al.*, 1997) or On-To-Knowledge Methodology (OTKM) (Sure *et al.*, 2004) focused mostly on developing a process for building (and reusing) ontologies (Gómez-Pérez *et al.*, 2006). METHONTOLOGY adapted the software development life-cycle for the ontology development process, while OTKM uses two orthogonal processes, with feedback loops. The first one is the usual process for knowledge use and evaluation, and the second is a knowledge meta process for introducing the ontology into an enterprise, as well as for maintaining it. Thus, the life-cycle model proposed by METHONTOLOGY is slightly more rigid than the one defined in OTKM. Neither of these methodologies mention any design criteria to be observed while developing ontologies.

De Nicola *et al.* (2009) propose a new methodology for building ontologies to check the quality of the resulting ontologies. The methodology is using four of the criteria discussed in Section 2.2.1, namely the syntactic, semantic, pragmatic, and social criteria. By using a formalism such as DL or OWL to specify ontologies, the authors argue that syntactic quality is automatically achieved. The semantic quality is verified by checking the consistency of the ontology, using a reasoner. The pragmatic quality is separated into three more characteristics: *fidelity*, *relevance*, and *completeness*. In our work, through the use of the rooted graphs, we consider the pragmatic quality of relevance. This quality is discussed in more details in Section 3.3.3. The other pragmatic qualities, along with the social qualities can be verified only in strict reference to the ontology requirements, by verifying their sources, their correct implementation, and their coverage. As such, they are not considered in our work.

In recent years, as the number of ontologies has increased, the field of knowledge engineering methodologies has re-focused on more agile, lightweight approaches. The NeOn methodology (Suárez-Figueroa *et al.*, 2012) is focused on identifying scenarios for building, sharing, reusing, and re-engineering knowledge resources. In a way, the NeOn methodology adapts the idea of design patterns in software engineering to the field of knowledge engineering. The scenarios are effectively used as ontology engineering design patterns. The NeOn methodology offers two life-cycle models: the rigid, sequential waterfall model, along with an iterative, incremental cyclic model.

The Unified Process for ONtology building (UPON Lite) methodology (De Nicola and Missikoff, 2016) introduces a more agile ontology engineering approach. UPON Lite seeks to ensure the reusability of existing ontology resources and the adaptability of its methodology across various industries and applications. In addition, the methodology places a high value on its ability to be used collaboratively. The collaboration is done almost exclusively by domain experts, without the need to involve the ontology engineers until the very last step of the process, which is the formalisation of the ontology into a standard language. This methodology also offers support to produce meaningful supporting documentation.

Jaskolka *et al.* (2015) propose an architectural design framework for ontology engineering. By applying the proposed design architecture, the resulting ontology exhibits several qualities, with *separation of concerns* as the most notable. The *separation of concerns* refers to dividing a system into distinct, manageable parts that handle specific aspects of functionality. By adhering to this quality, one obtains modular, flexible systems, thus more maintainable ontologies. Additionally, the authors highlight the importance of tools that automatically generate documentation for post-building activities, including the reuse, extension, and maintenance of ontologies.

Complementing this architectural perspective, the SEmi-Automatic Design Of Ontologies (SEADOO) (Grüninger *et al.*, 2023) methodology further demonstrates how semi-automated support tools can enhance ontology engineering. SEADOO integrates formal mathematical theories, such as those from the COLORE repository, with user-guided specification via positive and negative examples to generate logically sound axioms. Its model transformation and semantic alignment features promote modularity and reuse, aligning with the separation of concerns principle. By combining formal axiom generation with practical tool support, SEADOO highlights the methodological value of bridging formal rigor and user-driven design, particularly for developing ontologies that are maintainable, extensible, and grounded in reusable formal patterns.

2.3 Knowledge Generation

In this section, we explore knowledge generation in existing knowledge systems. We are interested mainly in what knowledge generation is and what are some of its standard reasoning tasks. Given the breadth of the reasoning and knowledge generation field, our discussion focuses specifically on DL-based reasoning tasks, which are grounded in FOL but designed to support decidable and tractable inference in knowledge systems. The notation used is also that pertaining to DL. We assume the reader has basic knowledge of it (Baader *et al.*, 2003).

As discussed in Chapter 1, in knowledge systems, the process of manipulating explicit knowledge to discover new (implicit) knowledge is commonly understood as reasoning. Antoniou *et al.* (2018) and Schneider and Šimkus (2020) discuss standard reasoning tasks that a knowledge system should be able to perform, such as *KB-satisfiability*, *consistency checking*, *concept satisfiability*, *subsumption*, *instance checking*, and *classification*, along with more advanced reasoning tasks such as *query answering*, *module extraction*, and *forgetting*. Some of the reasoning tasks are T-Box-related, such as KB-satisfiability, subsumption, and classification, while others are data-related tasks (involving both the T-Box and the A-Box), such as consistency checking, instance checking, and query answering.

Given a KB Σ, two concepts C and D, and an individual a, the standard tasks correspond to the following questions. KB-satisfiability answers the question "does the KB admit a model?", while consistency checking answers the question "is the instance data (the A-Box) consistent with the schema implied by the T-Box and within itself?" Concept satisfiability answers the question "does there exist at least one model of the KB in which C admits a non-empty extension?" Subsumption answers the question "is C more general than D?" Instance checking answers the question "is a an instance of C in every model of the KB?", while classification answers the question "based on the characteristics of an individual a, what concepts is a an instance of?" Query answering refers to the process of retrieving relevant information or offering solutions derived from the structure and content of the ontology, in response to user-generated questions. Module extraction refers to the process of identifying and extracting a module from an ontology. The module retains relevant, specific information required for a particular task or query. Finally, forgetting refers to the deliberate elimination of certain knowledge from an ontology. This is typically done in order to manage the size, relevance, or confidentiality of the ontology.

As described in Section 2.1, in a sense DL has become a standard for knowledge representation through ontologies. What makes DL so desirable for reasoning is that

its numerous fragments are decidable fragments of FOL. This allows for the creation of automated tools that can perform reasoning tasks and provide definite answers in finite time. The quality of these answers is contingent upon the correctness, comprehensiveness, and accuracy of the ontology itself. Automated proof systems can be evaluated using a number of metrics, including decidability and complexity (in space, time, or both). In the remainder of this section, we review the main reasoning approaches in DL and their metrics regarding the reasoning tasks above.

One category of reasoning approaches are syntax-based algorithms that solve the subsumption task by comparing the structure of concept expressions. The structural algorithms are studied for their computational properties (Brachman and Levesque, 2004), and are implemented in a number of KB systems, such as BACK (Quantz and Kindermann, 1990), LOOM (MacGregor, 1994), or CLASSIC (Borgida and Patel-Schneider, 1993). The underlying approach of these tools is to rewrite concepts in normal form, then compare their structure. If certain constructors are allowed in the language, the structural algorithms are incomplete w.r.t. the FOL semantics (Donini *et al.*, 1996). The most obvious example is a concept defined as $C \sqcup \neg C$. Semantically it is equivalent to $\top$, and subsumes every concept, i.e., for any concept D in the KB system, $D \sqsubseteq \top$. However, if neither $D \sqsubseteq C$ nor $D \sqsubseteq \neg C$ are satisfied, then by structural comparison (i.e., by checking if subsumption is satisfied on either branch C or $\neg C$) it cannot be discovered that $D \sqsubseteq C \sqcup \neg C$. Due to their incompleteness, syntax-based reasoning approaches are not useful, and their field was abandoned in the late 90s.

After the syntax-based approaches, another category emerged in the field: semantic-based (or logic-based) approaches. Baader *et al.* (2003) show that three of the standard reasoning tasks, namely concept satisfiability, subsumption, and instance checking, can be reduced to KB-satisfiability in linear time. Currently, common methods for solving standard reasoning tasks are tableaux (Horrocks, 1998), hypertableaux (Motik *et al.*, 2009), and resolution (Motik, 2009). The tableaux calculus is proved to be a sound and complete way to to solve the satisfiability problem. Due to the close relationship between DL and both propositional logic and propositional dynamic logic, domains where tableaux algorithms have proven effective (Baader *et al.*, 2003), DL reasoning often employs tableaux-like algorithms. This approach was applied to various fragments of DL, such as $\mathcal{ALC}$, $\mathcal{ALCN}$, and $\mathcal{ALCQ}$ (Baader *et al.*, 2003). The main advantage of tableaux algorithms is that they are complete and are (often) of optimal complexity.

Most ontology reasoners are built around DL fragments and rely on tableaux or resolution calculus. Matentzoglu *et al.* (2015) give an overview of current DL-based

(specifically OWL-based) reasoners. The survey presents more than thirty reasoners, each reasoner covering a specific DL fragment (or set of fragments), supporting different levels of language expressivity and a variety of reasoning tasks. In terms of the reasoning tasks covered, most of the reasoners presented in (Matentzoglu *et al.*, 2015) support the standard T-Box-related satisfiability reasoning tasks and only a few support data-related tasks.

In terms of the underlying calculi used, reasoners reviewed in (Matentzoglu *et al.*, 2015) are classified under three main categories: consequence-, model construction-, and rewriting-based. Reasoners in the first category focus on adding new consequences to a given KB, and they mostly use resolution algorithms. The consequence-based reasoners are mainly dealing with tractable DL fragments, such as $\mathcal{EL}$ fragments, which provide high efficiency of reasoning over simple and large KBs (large number of concepts, small number of relations). The model construction-based reasoners focus on building models based on a given KB, and/or checking the KB consistency, and they mostly use tableaux and hypertableaux algorithms. They are generally used with highly expressive fragments, such as extensions of $\mathcal{ALC}$ fragments. Rewriting-based reasoners are used to expand a given KB to be used for other reasoning tasks such as answering queries. This category of reasoners is focused more on the data aspect of the KB and is commonly used for query rewriting. Many of the surveyed reasoners make use of a hybrid approach, in which they combine algorithms from multiple categories.

In terms of effectiveness, tableaux algorithms can handle more expressive DL fragments, while resolution algorithms are limited to less expressive fragments (Baader *et al.*, 2003). However, tableaux algorithms are limited to ontologies whose concepts are not cyclic, while resolution is effective on highly cyclic ontologies. By cyclic ontologies we understand ontologies that contain one or more cyclic relations, or cyclic concept definitions. In terms of complexity, tableaux algorithms use more memory space and time than resolution procedures (Song *et al.*, 2011). It is no surprise that most current reasoners focus on algorithms supporting tractable fragments DL, which ensures that the algorithms are both sound and complete (Matentzoglu *et al.*, 2015).

2.4 Existing Higher Order Logic Theorem Provers

To support this need for expressive yet tractable reasoning frameworks, we turn to theorem provers, which offer the foundational infrastructure required for formalising complex knowledge systems. The formalisation of the DIS, discussed in more detail in Chapter 4, requires a proof assistant capable of expressing and reasoning over both algebraically rich structures and semantically aligned mappings between data

and conceptual representations. Given the complexity of DIS, which encompasses structured data views (based on cylindric algebra), ontological structures (based on Boolean lattices and graphs), and an integration layer via a mapping operator, a HOL-based Interactive Theorem Prover (ITP) is required to support its formal specification and reasoning capabilities.

In this section, we review existing higher-order theorem provers to determine their suitability for our purposes. Nawaz *et al.* (2019) evaluate several theorem provers currently in use, based on different criteria. We leverage this survey to identify a suitable reasoning engine for our work, focusing on well-documented, widely recognised higher-order proving engines still in use. We focus our review on three widely used interactive theorem provers: HOL, Rocq (formerly known as the Coq Proof Assistant), and Isabelle, selected for their foundational significance, logical diversity, and widespread use in formal verification and proof engineering. These systems represent three major foundations in interactive theorem proving: simple type theory (HOL), constructive dependent type theory (Rocq), and a hybrid declarative framework with strong automation (Isabelle). Together, they provide a representative and mature landscape for formalising structured knowledge systems like DIS.

A theorem prover can be categorised as an Automated Theorem Prover (ATP) or an ITP (also called *proof assistant*). Although both are reasoning tools used to verify proofs and establish the correctness of statements, the ATPs are fully automated. In contrast, the ITPs collaborate with users, allowing users to guide the proof construction process within a formal framework, and to write, check, and verify proofs interactively. Given the practical constraints of full automation, ITPs are more suitable for formalising complex theorems (and theories) in mathematics (Nawaz *et al.*, 2019; Harrison *et al.*, 2014). The theorem provers we consider are all ITPs.

The three ITPs we review are rooted in Logic of Computable Functions (LCF) (Gordon *et al.*, 1979), a theorem prover for Scott's Logic of Computable Functions (Scott, 1993), and rely on two key ideas (Harrison, 2009). First, any proof is performed by a sequence of inferences on a small set of axioms (primitives). Thus, provided the original set of axioms is correct, any results based on it should be correct as well. This is achieved by making the theorems a special abstract type, and using the constructors of this type as the inference rules of the system. Second, the entire reasoning engine is embedded within a powerful functional (strongly typed and high-level) programming language, Meta Language (ML), used to program new inference rules. The programming environment of ITPs uses the abstract type *theorem* to ensure that new rules can be reduced to primitives (original axioms). This basic approach is applicable to

any logic, therefore many of the higher order logic ITPs discussed below are descendants of LCF.

The ITPs we reviewed satisfy the de Bruijn criterion (Barendregt and Wiedijk, 2005), which requires that the proof assistant generates proof objects that can be verified by a small proof-checking kernel. In addition, all three proof assistants have implemented *hammers*, which are general reasoners over (large) formal proof libraries. Hammers leverage data from previous proof attempts and maintain a small, trusted base of proofs used by proof assistants. They typically resolve proof goals through brute-force methods (Ringer *et al.*, 2019).

2.4.1 HOL

HOL (2012) is a LCF-style ITP for higher-order logic. Highly automated, the proof assistant offers an environment in which proof techniques and strategies can be implemented, verified, and refined (Kunčar and Popescu, 2015). It also provides an oracle mechanism that accesses external (and efficient) Boolean Satisfiability (SAT) solvers, such as Satisfiability Modulo Theory (SMT) and Binary Decision Diagram (BDD) solvers. HOL is built using the meta-language ML awhich gives it a high degree of programmability. It is used for implementing a variety of tasks, such as combinations of deductions and property checking. Since it is LCF-based, the proof objects of HOL are (usually) ephemeral, i.e., they are built and checked step by step (Ringer *et al.*, 2019). This allows for a lower memory print. In addition, the time to generate and check proof objects is directly proportional to the proof object size.

2.4.2 Rocq

ROCQ (2025) is an LCF-style formal proof management system. Based on an expressive language that integrates higher-order logic with functional programming language, Rocq offers interactive proof methods, decision procedures, and extensibility for user-defined proof strategies. It is commonly used for certifying properties of programming languages and formalising mathematical theories, making them machine-readable. Rocq allows the user to translate certified programs into functional programming languages such as Objective Caml (OCaml), Haskell, or Scheme. In contrast to HOL, in Rocq proof objects are produced in full, which may create a higher memory print. Additionally, the time required to check proofs in Rocq may not be directly proportional to the size of the proof object (Ringer *et al.*, 2019).

2.4.3 Isabelle

Isabelle (2025) is an LCF-style generic proof assistant, with a declarative proof style that allows for a natural proof text, easily understood by both humans and computers. Isabelle is the answer to a common problem in computer science, that of a need for proof support for many particular logics in a common, reusable, generic manner (Harrison *et al.*, 2014). Its primary use is the formal development of mathematical proofs, including formal verification of hardware or certification of programming languages and protocols. Compared to other ITPs, Isabelle offers extensive built-in support through a comprehensive theory library that includes foundations for number theory, algebra, and set theory. Isabelle's logical architecture is compatible with a variety of formal calculi, enabling flexible and rapid prototyping of deductive systems. Users can extend it with their own proof procedures and theory packages, using ML. Similar to HOL, Isabelle does not produce full proofs, it rather builds a proof piece by piece. Similar to Rocq, Isabelle allows the extraction of executable specifications into functional languages code (OCaml, Haskell, Scheme).

2.4.4 Other Higher Order Logic Provers

Prototype Verification System (PVS) (PVS, 2023) is a formal specification and verification automated system. Its theorem prover is designed for applications with an expressive classical type theory and powerful automation. It was one of the first automated prover that showed one does not need to choose between expressivity of logic and power of the reasoning engine (Harrison *et al.*, 2014). PhoX (PhoX, 2024) is an extensible higher order logic proof assistant. It is highly user-interactive (not fully automated) tool, the user giving an initial goal and guiding the proof through subgoals. NuPRL (NuPRL, 2014) is another LCF-style ITP, developed originally to create programs in a rigorous, mathematical manner, by interactive refinement. Ωmega (Benzmüller *et al.*, 1997) introduces a unified technique of combining several proof tools, including proof planning.

2.5 Conclusion

Over the past decade, the field of knowledge representation has seen a proliferation of formalisms, particularly in the areas of graph-based and logic-based approaches. Graph-based formalisms, such as semantic networks and conceptual graphs, emphasise structural expressiveness, while logic-based formalisms, such as DL and F-Logic, offer well-defined semantics for reasoning. DL has emerged as the standard for ontology specification and reasoning due to its decidable fragments, while F-Logic remains

valued for its object-oriented modelling capabilities and integration with Semantic Web Services. However, limitations remain: F-Logic is generally undecidable and assumes a closed-world view, while DL formalisms, although more tractable, often require rigid and static ontological structures that complicate data evolution and integration. These limitations highlight the need for more flexible, semantically grounded approaches to knowledge representation, particularly those capable of accommodating structural changes and supporting data-ontology alignment. As modern knowledge systems increasingly interact with evolving and heterogeneous datasets, knowledge representation formalisms must evolve to balance expressivity, modularity, and adaptability.

To support these evolving needs, algebraic structures offer a foundation for representing and reasoning over both data and conceptual knowledge. RAs have historically provided the formal basis for manipulating data in relational databases, offering a well-understood framework for manipulating n-ary relations. However, RAs lack features such as variable abstraction, quantification, and equality, which are essential for modelling logic-based systems. CAs, originally developed to formalise FOL, extend RAs by introducing operations such as cylindrification, which addresses existential quantification, and diagonal elements, which addresses equality. These features make CA strictly more expressive than RA and better suited for modelling structured data in logic-based environments. Complementing this data-oriented expressiveness, algebraic structures such as Boolean lattices, concept monoids, and rooted graphs serve as effective tools for modelling conceptual knowledge. These structures enable modular composition and abstraction at the knowledge level, supporting the design of complex ontologies. Together, these algebraic foundations provide a rich formal toolkit for building knowledge systems that are both logically expressive and structurally modular, properties increasingly essential in modern, dynamic knowledge engineering environments.

Beyond formal expressiveness, a strong engineering foundation is equally critical. The literature highlights essential properties of well-designed ontologies, such as coherence, clarity, extensibility, modularity, and reusability. From an engineering standpoint, achieving modularity requires knowledge systems to maintain low coupling and a clear separation of concerns. This principle also applies to the integration of data: embedding data directly into ontologies is not practical. Instead, a modern engineering perspective advocates for decoupling data from its domain knowledge, while ensuring the two remain semantically aligned. Despite the existence of various ontology engineering methodologies, the field remains underdeveloped in terms of formal process and collaborative tooling. The interdisciplinary nature of ontology development calls for greater methodological support for collaboration, especially among

domain experts. As knowledge representation increasingly intersects with dynamic and cross-domain systems, the need for collaborative, modular engineering approaches becomes even more critical.

The ability to generate new knowledge through reasoning remains central to the utility of formal systems. A wide range of reasoning tasks, such as consistency checking, satisfiability, classification, and query answering, form the backbone of logical inference in DL-based systems. The evolution of reasoning approaches, from syntax-based methods to more robust semantic-based algorithms, highlights the ongoing quest to enhance the efficiency and versatility of reasoning engines. However, DL reasoning is typically tied to specific fragments, each with its own tailored algorithmic support. This fragmentation limits the adaptability and maintainability of reasoning systems. A more general and unified approach to reasoning could reduce the need for specialised tools and make reasoning frameworks more scalable, maintainable, and accessible across domains. This direction calls for logical frameworks that are expressive, semantically transparent, and modular enough to support practical reasoning.

In this context, ITPs play a crucial role. Unlike automated provers that prioritise speed over expressivity, ITPs allow for the formalisation of complex mathematical structures and domain theories through user-guided proof development. In modern knowledge engineering, where formal rigour must coexist with usability and extensibility, ITPs offer a practical and scalable path toward trustworthy knowledge system design. DIS includes reasoning about sets of sets, which require the expressive foundation of HOL. This becomes especially important when specifying mapping operators between the data sorts and concept atoms and proving their properties. This form of reasoning is inherently higher-order, as it deals not just with objects in the domain but also with properties and operations over them. In addition, the modularity of the DIS architecture, where components like the DDV, the DOnt, and their integration are developed as independent yet composable theories, benefits from the structured theory development capabilities found in HOL-based ITPs.
Among the most mature ITPs are HOL, Rocq, and Isabelle, each grounded in the LCF architecture that ensures soundness through a small trusted kernel. Isabelle, in particular, stands out for its support for modular theory development, its integration of multiple mathematical foundations, and its readable, declarative proof language. Its locale system and natural deduction environment are particularly well-suited for formalising multi-layered knowledge systems that combine algebraic and graph-based reasoning, and enables the clean separation of assumptions, contexts, and reusable theory fragments. Isabelle's interactive interface and human-readable proofs facilitate collaboration with domain experts, bridging the gap between formal methods and applied knowledge engineering. These features make it well-suited for formalising

structured, algebraically grounded knowledge systems and for supporting collaboration with domain experts.

This chapter reviewed foundational formalisms, engineering methodologies, reasoning approaches, and tool support relevant to the development of structured knowledge systems. The limitations of current approaches, particularly in terms of modularity, data integration, and reasoning scalability, motivate the need for a unified formalism. The next chapter introduces the mathematical structures and concepts used in this work, including DIS, a framework that addresses these gaps by combining algebraic semantics, engineering principles, and formal reasoning in a cohesive model.

Chapter 3

Mathematical Background

In this chapter, we provide the mathematical background needed to make this thesis self-contained. In Section 3.1, we present the foundational elements of mathematical structures. In Section 3.2, we introduce the mathematical background that is essential for understanding Domain Information System (DIS), which is a relatively new formalism. In Section 3.3, we present DIS, a novel data-centered knowledge representation formalism that is the focus of this research. In Section3.4, we present algebraic specifications, to support the semantics of DIS, as well as its specification in Isabelle/HOL.

Throughout the thesis, we adopt the uniform linear notation employed in Isabelle, for quantified terms and set comprehension expressions. The general structure of a quantified term is written as $(\star R\, x.\ P\, x)$, where $\star$ denotes the quantifier, x is the bound (or quantified) variable, $R\, x$ specifies the range or domain restriction, and $P\, x$ represents the body of the quantified statement. The expression $\exists R\, x.\ P\, x$ is understood as $\exists x.\ (R\, x \wedge P\, x)$, while the expression $\forall R\, x.\ P\, x$ is understood as $\forall x.\ (R\, x \implies P\, x)$. When it is clear what the range is, we use the simplified version $(\star x.\ P\, x)$. Note that throughout this work, the symbols $\wedge, \vee, \neg$ stand for logical AND, OR, and NOT, respectively. In addition, the symbol $!\exists$ is understood as *exists a unique*.

The general form for set comprehension is $\{E\, x \mid x\ .\ P\, x\}$, where x is the dummy variable, $E\, x$ is the expression that describes the elements of the set (in terms of the given variable), and $P\, x$ is the predicate representing the restriction placed on the variable. In some cases, within the set comprehension predicate $P\, x$ we may need to use other dummy variables that are not used inside the $E\, x$ expression. Such a set comprehension may look like $\{E\, x \mid x\, y\, z\ .\ P\, x\, y\, z\}$ and it translates to "the values of $E\, x$ s.t. $\forall x.\ \exists y\, z.\ P\, x\, y\, z$".

3.1 Mathematical Structures

In (Marker, 2000), a homogeneous (or single sorted) mathematical structure is described with the use of a *signature*, which includes function, relation, and constant symbols, each associated with a fixed arity. This framework is a specialisation of heterogeneous (or many-sorted) structures, where multiple distinct sorts are considered simultaneously.
A many-sorted signature Σ specifies:

- a finite set of sort symbols $\mathcal{S} = \{S_1, S_2, \ldots, S_k\}$
- a set of function symbols, each with a designated list of input sorts and one output sort, e.g., $f : S_{i_1} \times S_{i_2} \times \cdots \times S_{i_n} \to S_j$, called a function of arity n or an n-ary function
- a set of relation symbols over tuples of sorts, e.g., $R \subseteq S_{i_1} \times S_{i_2} \times \cdots \times S_{i_n}$, called a relation of arity n or an n-ary relation
- a set of constant symbols associated with specific sorts, e.g., $c : S_i$, called a function of arity 0 or a nullary function

In the homogeneous case, the set of sorts contains exactly one sort, S, called the underlying set. Then a function symbol is specified as $f : S^{n_f}$, with n_f the arity of f, and a relation symbol is specified as $R \subseteq S^{n_R}$, with n_R the arity of R. For clarity and ease of presentation, the remainder of this section focusses on formal definitions in the homogeneous case, which can be straightforwardly generalised to the heterogeneous setting.

Definition 3.1.1. (Marker (2000)) Let $\mathcal{F}$ be a set of function symbols f with positive integers n_f the arity of each $f \in \mathcal{F}$, $\mathcal{R}$ a set of relation symbols R with positive integers n_R the arity for each $R \in \mathcal{R}$, and $\mathcal{C}$ a set of constant symbols. Then $\Sigma = \big(\mathcal{F},\ \mathcal{R},\ \mathcal{C}\big)$ is called a *signature*. □

For example, the signature of monoids is defined as $\Sigma_m = \big(\{(+, 2)\},\ \varnothing,\ \{0\}\big)$, and the signature of boolean algebras is defined as $\Sigma_b = \big(\{(+, 2), (-, 2), (\cdot, 2)\},\ \varnothing,\ \{0, 1\}\big)$. In general, the signatures omit the arity, thus, the two signatures above are simplified to $\Sigma_m = \big(\{+\},\ \varnothing,\ \{0\}\big)$ and $\Sigma_b = \big(\{+, -, \cdot\},\ \varnothing,\ \{0, 1\}\big)$, where $+, -, \cdot$ are binary function symbols and $0, 1$ are constants.

Definition 3.1.2 (Marker (2000))**.** Let Σ be a signature and let M be a non-empty set, called the universe or domain. The *interpretation* for each $f \in \mathcal{F}$ is a function

$f^{\mathcal{M}} : M^{n_f} \to M$, the interpretation for for each each $R \in \mathcal{R}$ is a relation $R^{\mathcal{M}} \subset M^{n_R}$, and the interpretation for each $c \in \mathcal{C}$ is a constant $c^{\mathcal{M}} \in M$.
Then $\mathcal{M} = \big(M, \{f^{\mathcal{M}}\}_{f\in\mathcal{F}}, \{R^{\mathcal{M}}\}_{R\in\mathcal{R}}, \{c^{\mathcal{M}}\}_{c\in\mathcal{C}}\big)$ is called a Σ-*structure*. □

For example, the structure $\mathcal{M}_m = \big(\mathcal{N}, \{+\}, \varnothing, \{0\}\big)$ is called the monoid of natural numbers under addition and the structure $\mathcal{M}'_m = \big(\mathcal{N}, \{\cdot\}, \varnothing, \{1\}\big)$ is the monoid of natural numbers under multiplication. In the former, the binary operator $+$ is interpreted as $+^{\mathcal{M}_m} \equiv +$ (i.e., the addition over the natural numbers), and the constant 0 as $0^{\mathcal{M}_m} \equiv 0$ (i.e., the natural number 0). In the latter, $+^{\mathcal{M}'_m} \equiv \cdot$ and $0^{\mathcal{M}'_m} \equiv 1$.
In a mathematical structure, any of the sets $\mathcal{F}$, $\mathcal{R}$, $\mathcal{C}$ may be empty. A mathematical structure is called a *relational structure* if $\mathcal{F} = \varnothing$. A mathematical structure is called an *algebraic structure* if $\mathcal{R} = \varnothing$. Then, the Σ-structure $\mathcal{M} = \big(M, \{f^{\mathcal{M}}\}_{f\in\mathcal{F}}, \{c^{\mathcal{M}}\}_{c\in\mathcal{C}}\big)$ is called a Σ-Algebra.

For example, the monoid of natural numbers under addition is an algebraic structure and its notation is simplified to $\mathcal{M}_m = \big(\mathcal{N}, +, 0\big)$. In Sections 3.2.2 and 3.2.4 we give other examples of relational and algebraic structures, respectively.

3.2 Mathematical Background

In this section, we present foundational mathematical elements, such as graphs (Section 3.2.1), homogeneous relations and their representations (Section 3.2.2), lattices and their algebraic properties (Section 3.2.3), algebraic structures (Section 3.2.4), and Tarsky's cylindric algebra (Section 3.2.5).

3.2.1 Graphs

The following definitions are borrowed from (Gross *et al.*, 2018). A *graph* $G = \big(V, E\big)$ is a mathematical structure consisting of two sets V and E. The elements of V are called vertices, and the elements of E are called edges. Each edge has a set of one or two vertices associated to it, which are called its endpoints. A *path* from vertex v_0 to vertex v_n is an alternating sequence of vertices and edges $P = \langle v_0, e_1, v_1, e_2, \ldots, v_{n-1}, e_n, v_n\rangle$, where $v_0, \ldots, v_n \in V$ and $e_1, \ldots, e_n \in E$, s.t. the endpoints of e_i are v_{i-1}, v_i and there are no repeated edges or vertices (except possibly the initial and final vertices).

A *directed edge* is an edge on which one of its endpoints is designated as the tail, and the other endpoint is designated as the head. The direction of the edge is given by the ordered pair $(tail, head)$. A graph with directed edges is called a *directed graph*

or *digraph*. A *rooted directed graph* is a digraph with a distinguished node r, such that there is a directed path from r to any node other than r. A *self-loop* is an edge that joins a single endpoint to itself. A path starting and ending at the same vertex is a *cycle* (except for self-loops). An *acyclic* graph has no cycles and no self-loops.

3.2.2 Homogeneous Relations

The definitions and results that follow are adapted from (Davey and Priestly, 1990). A relation captures the manner in which elements are connected or associated. When all related elements belong to the same set, the relation is said to be *homogenous*. This work focusses exclusively on homogeneous binary relations.

Let C be a set. A *(binary homogenous) relation* R on C is defined as a subset of $C \times C$, written as $R \subseteq C \times C$. Relations are sets of tuples, where a *tuple* is understood to be an ordered pair. Thus all set operations are available for use. On C, there are defines a number of distinguished relations, such as the *empty relation*, denoted by $O = \varnothing$ (or O_C when it is not clear what the underlying set is); the *universal relation*, denoted by $U = C \times C$ (or U_C for clarity); and the *identity relation*, denoted by $\mathbb{I} = \{(x, x) \mid x \in C\}$ (or $\mathbb{I}_C$ for clarity).

Let C be a set and $R \subseteq C \times C$ an associated homogeneous relation. R can be represented as a directed graph $G = \big(C, R\big)$, where the vertices of the graphs are the elements of the underlying set C, and the edges of the graph are represented by the tuples of R. Let $R, S \subseteq C \times C$ be relations. Their *product* $R;S \subseteq C \times C$ is defined as $R;S = \{(x, z) \mid x\ y\ z\ .\ y \in C \wedge (x, y) \in R \wedge (y, z) \in S\}$, and it is called the *composition* of relations. When it is clear from the context, the composition is also denoted by RS. The powers of R are written as R^2, R^3, etc. The composition of relations is associative, and its identity element is $\mathbb{I}$.

Let R be a binary relation on C. We say that R is *reflexive* iff $\mathbb{I} \subset R$. This is equivalent to $R = R \cup \mathbb{I}$ or $\forall x.\ (x, x) \in R$. We say that R is *symmetric* iff $\forall x\ y.\ (x, y) \in R \implies (y, x) \in R$. R is *antisymmetric* iff $\forall x\ y.\ (x, y) \in R \wedge (y, x) \in R \implies x = y$. Finally, we say that R is *transitive* iff $R^2 \subseteq R$, which is equivalent to $\forall x\ y\ z.\ (x, y) \in R \wedge (y, z) \in R \implies (x, z) \in R$. We define the *transitive closure* of R (denoted by R^+) as the minimal relation that contains R and is transitive, i.e., $R^+ \stackrel{\text{def}}{=} \bigcap\{H \mid R \subseteq H \wedge H \text{ is transitive}\}$.

3.2.3 Lattices

The following definitions and results are taken from (Davey and Priestly, 1990). Let P be a set. A *partial order* on P is a binary relation on P that is reflexive, antisymmetric, and transitive. A set P equipped with a partial order $\leqslant$ is called a *partially ordered set* (or a *poset*), written as $\big(P, \leqslant\big)$. When it is clear from the context, we say simply that "P is a poset". Two posets are *order-isomorphic* if they have the same size and there is an order-preserving mapping between them (Ciesielski, 1997). Let P be a poset and let $x, y \in P$. We say x is *covered by* y (or y *covers* x), and write $x \prec y$ or $y \succ x$, if $x < y \wedge \forall z \in P.\ x \leqslant z < y \implies z = x$.

Given a poset P, a new ordered set P^d (the *dual of* P, given by $\big(P, \geqslant\big)$) can be created by defining $x \leqslant y$ to hold in poset P^d iff $y \leqslant x$ holds in poset P. Given a statement Φ about ordered sets that is true in all ordered sets, then the dual statement Φ^d is true in all ordered sets, and it is obtained by replacing all original occurrences of $\leqslant$ by $\geqslant$ and all original occurrences of $\geqslant$ by $\leqslant$. P is said to have a *bottom* element if there exists $\bot \in P$ s.t. $\bot \leqslant x$, for all $x \in P$. Dually, P is said to have a *top* element if there exists $\top \in P$ s.t. $x \leqslant \top$ for any $x \in P$. For example, if P is the power set of a set X, with the subset relation $\subseteq$ denoting ordering relation, then $\top(\mathcal{P}(X)) = X$ and $\bot(\mathcal{P}(X)) = \varnothing$. When they exist, it is easy to show that the top and bottom elements are unique. Let P be a poset and let $S \subseteq P$. We call $a \in S$ a *minimal* element of S iff $\forall x \in S.\ x \leqslant a \implies a = x$ (dual: *maximal* element). We call $a \in S$ the *least element* of S iff $\forall x \in S.\ a \leqslant x$ (dual: the *greatest element*). An element $x \in P$ is an *upper bound* of S if $\forall s \in S.\ s \leqslant x$ (dual: *lower bound*). The set of all upper bounds for S is denoted by S^u (read "S upper") and it is written as $S^u = \{x \in P \mid \forall s \in S.\ s \leqslant x\}$ (dual: "S lower", S^l).

If S^u has a least element, x, then x is called the *least upper bound* or *supremum* of S, $sup(S)$ (dual: *greatest lower bound*, *infimum*, $inf(S)$). When $S = P$, if $\top$ of P exists, $P^u = \{\top\}$ and $sup(P) = \top$. Dually, if $\bot$ of P exists, $P^l = \{\bot\}$ and $inf(P) = \bot$. When $S = \varnothing$, $\varnothing^u = P$ and, if $\bot$ of P exists, $sup(\varnothing) = \bot$. Dually, if $\top$ of P exists, $inf(\varnothing) = \top$. If $sup\{x, y\}$ exists, we write it as $x \sqcup y$ or x *join* y. Dually, if $inf\{x, y\}$ exists, we write it as $x \sqcap y$ or x *meet* y. Similarly, we write $\bigsqcup S$ (the *join* of S) for $sup(S)$ and $\bigsqcap S$ (the *meet* of S) for $inf(S)$, respectively.

In a poset P, the least upper bound, $x \sqcup y$ of any two elements, $\{x, y\}$ may fail to exist because x and y have no common upper bound or they have no least upper bound. For example, let $P = \{\varnothing, \{a\}, \{b\}, \{c\}, \{a, b\}\}$, with $\subseteq$ the ordering relation. The partial order is represented in Figure 3.1. We observe that $\{a\} \sqcap \{b\} = \varnothing$, and similarly, $\{a\} \sqcap \{c\} = \varnothing$, $\{b\} \sqcap \{c\} = \varnothing$, $\{a, b\} \sqcap \{a\} = \{a\}$, $\{a, b\} \sqcap \{b\} = \{b\}$, and $\{a, b\} \sqcap \{c\} = \varnothing$. Thus, any pair of elements in P has a meet. We observe that the

$\{a, b\}$ pair has a join in P, $\{a\} \sqcup \{b\} = \{a, b\} \in P$. However, the join of the $\{a\}, \{c\}$ pair is not an element of P, $\{a\} \sqcup \{c\} = \{a, c\} \notin P$. Thus, not all pairs in P have a join.

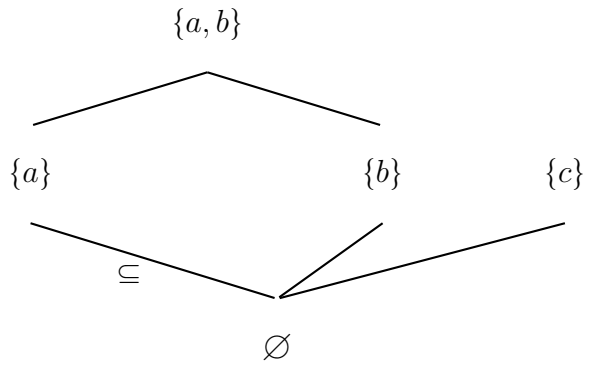

Figure 3.1: Poset

Let P be a non-empty poset. P is called a *lattice* if for all $x, y \in P$ both $(x \sqcap y)$ and $(x \sqcup y)$ exist. P is called a *complete lattice* if for all $S \subseteq P$ both $\bigsqcup S$ and $\bigsqcap S$ exist. The poset P depicted in Figure 3.1 is not a lattice, the meet and join do not exist for all pairs of elements of P.

Let L be a lattice. L is said to be *distributive* if it satisfies the *distributive law*, as follows: $\forall a, b, c \in L.\ a \sqcap (b \sqcup c) = (a \sqcap b) \sqcup (a \sqcap c)$. Let L be a lattice with a top $\top$ and a bottom $\bot$. For every $a \in L$, we say $b \in L$ is a *complement* of a if $a \sqcap b = \bot$ and $a \sqcup b = \top$. If a has a unique complement, we denote this complement by a'.

Definition 3.2.1 (Davey and Priestly (1990))**.** Let L be a lattice. L is called a *Boolean lattice* if L is distributive, L has a top and a bottom, and $\forall a \in L.\ !\exists a' \in L.\ a'$ is complement of a. □

Definition 3.2.2 (Davey and Priestly (1990))**.** Let L be a lattice with bottom element $\bot$. Then $a \in L$ is called an *atom* if it covers the bottom element, i.e., $\bot \prec a$. The set of all atoms in L is denoted by $\mathcal{A}(L)$. □

Corollary 3.2.1 (Davey and Priestly (1990))**.** Let L be a finite lattice. Then the following statements are easilyy shown to be equivalent:

(i) L is a Boolean lattice

(ii) L is order-isomorphic to the powerset of its atoms

(iii) $\forall a \in L.\ !\exists a' \in L.\ a'$ is complement of a □

In Figure 3.2.3, we show an example of the Boolean lattice that is order-isomorphic to the poset $\big(\mathcal{P}(\{a, b, c\}), \subseteq\big)$.

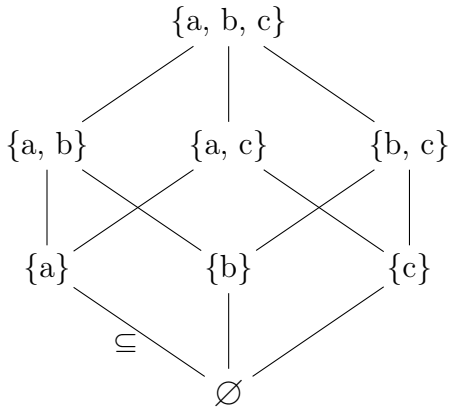

Figure 3.2: Boolean lattice for $\mathcal{P}(\{a, b, c\})$

3.2.4 Algebraic Structures

The following definitions and results are borrowed from (Birkhoff and MacLane, 1941). Let B be a set of elements, and let there be a binary operation $\diamond$ on B. The operation $\diamond$ is said to be *idempotent* iff $\forall x \in B.\ x \diamond x = x$. The operation $\diamond$ is said to be *commutative* iff $\forall x, y \in B.\ x \diamond y = y \diamond x$. The operation $\diamond$ is said to be *associative* iff $\forall x, y, z \in B.\ x \diamond (y \diamond z) = (x \diamond y) \diamond z$.

Two binary operations $\diamond$ and $\star$ on B are said to satisfy the *absorption law* iff $\forall x, y \in B.\ x \diamond (x \star y) = x \star (x \diamond y) = x$. Operator $\diamond$ is said to *distribute over* operator $\star$ iff $\forall x, y, z \in B.\ x \diamond (y \star z) = (x \diamond y) \star (x \diamond z)$. $\diamond$ and $\star$ are said to be *mutually distributive* iff they distribute over each other. On a set B equipped with a binary operation $\diamond$, a distinguished element $e \in B$ is called *identity element* if $\forall x \in B.\ x \diamond e = e \diamond x = x$.

A *semigroup* is an algebraic structure, $\big(B, \diamond\big)$, where B is a set of elements and $\diamond$ is an associative binary operation. The semigroup is called *commutative* if the operator $\diamond$ is commutative. A *monoid* is an algebraic structure $\big(B, \diamond, e\big)$, where $\big(B, \diamond\big)$ is a semigroup and e is the identity element for $\diamond$. The monoid is called *commutative* if $\diamond$ is commutative.

Definition 3.2.3 (Birkhoff and MacLane (1941))**.** Let B be a set equipped with binary operators $+$ and $\cdot$, unary operator $-$ (called *complement*), and distinguished

elements 0 and 1 . The algebraic structure $\mathcal{B} = \big(B, +, \cdot, -, 0, 1\big)$ is called a*Boolean algebra* if the following axioms are satisfied:

(B1) $\big(B, +, 0\big)$ is a commutative monoid

(B2) $\big(B, \cdot, 1\big)$ is a commutative monoid

(B3) $+$ and $\cdot$ are mutually distributive;

(B4) $\forall x \in B.\ x + -x = 1 \wedge x \cdot -x = 0$ □

Let $\mathcal{B}$ be a Boolean algebra. Its distinguished elements 0 and 1 are called *annihilators* for $+$ and $\cdot$, respectively, i.e., $\forall x \in B.\ x \cdot 0 = 0$ and $\forall x \in B.\ x + 1 = 1$. Let $x, y \in B$ be any two elements of $\mathcal{B}$. Then a non-strict ordering relation, denoted by $\leqslant$, is defined in the natural way, as follows:

$$x \leqslant y \equiv x + y = y \tag{3.1}$$

With this understanding, the $+, \cdot$ operators of the Boolean algebra $\mathcal{B}$ are the equivalent of the join and meet operators on the lattice $(B, \leqslant)$, respectively. Therefore, the lattice $(B, \leqslant)$ is a Boolean lattice, and it is isomorphic to the Boolean algebra $\mathcal{B}$ (Sankappanavar and Burris, 1981). In the remainder of this thesis we use the terms Boolean algebra and Boolean lattice interchangeably.

3.2.5 Cylindric Algebras

Cylindric algebra was introduced by Alfred Tarski in (Tarski *et al.*, 1971) as an extension of Boolean algebra, which provides the algebraic basis for propositional logic. To capture the additional expressive power required for first-order logic with equality, Tarski enriched the Boolean framework by incorporating a cylindrification operator to model quantification, and diagonal elements to represent equality. This construction resulted in an algebraic system capable of representing first-order logic in a purely mathematical form, making it accessible to those without a background in formal logic.

In our work we are interested only in the cylindrification operator, hence the focus of this mathematical background is on diagonal-free cylindric algebras. Throughout the remainder of this work, when we say *cylindric algebra*, we refer to a diagonal-free cylindric algebra.

Definition 3.2.4 (Tarski *et al.* (1971))**.** Let $\big(B, +, \cdot, -, 0, 1\big)$ be a Boolean algebra and $\kappa, \alpha \in \mathbb{N}$. The algebraic structure $\mathcal{B} = \big(B, +, \cdot, -, 0, 1, \{\mathbf{c}_\kappa\}_{\kappa<\alpha}\big)$ is called a *diagonal-free cylindric algebra of dimension* α if the following axioms are satisfied for any $x, y \in B$, and any $\kappa, \lambda \in \mathbb{N}$ with $\kappa, \lambda < \alpha$:

(C1) $\mathbf{c}_\kappa 0 = 0$

(C2) $x \leqslant \mathbf{c}_\kappa x$

(C3) $\mathbf{c}_\kappa(x \cdot \mathbf{c}_\kappa y) = \mathbf{c}_\kappa x \cdot \mathbf{c}_\kappa y$

(C4) $\mathbf{c}_\kappa \mathbf{c}_\lambda x = \mathbf{c}_\lambda \mathbf{c}_\kappa x$ □

Axiom (C1) expresses the normality of the $\mathbf{c}_\kappa$ operators, where the normality of an operator indicates its adherence to standard properties or behaviours within a particular mathematical framework. Axiom (C2) expresses the generalisation property of the $\mathbf{c}_\kappa$ operators, stating that any element is included in its cylindrification. Axiom (C3) closely parallels the modular law, which, in lattice theory, describes a relationship between the join and meet operations of a lattice. Axiom (C4) expresses the commutativity of the $\mathbf{c}_\kappa$ operators. For additional information on cylindric algebras, we refer the reader to Appendix D, Section D.2.

Cylindric algebras provide a natural framework for modelling information systems, where elements of the Boolean algebra B represent individual pieces of information, the operator $\cdot$ corresponds to information combination, and the 0 element denotes the absence of information. Within this interpretation, Axiom (C1) guarantees that applying cylindrification to the absence of information results in no information. Axiom (C2) reflects the expansive nature of cylindrification: applying it to any piece of information yields a result that includes, rather than restricts, the original content.

3.3 Domain Information System

The DIS is a novel data-centered framework for knowledge representation. It consists of a domain knowledge representation, called the domain ontology $\mathcal{O}$, a domain data representation, called the domain data view $\mathcal{A}$, and an operator τ that maps $\mathcal{A}$ to $\mathcal{O}$. The domain ontology is further refined (i.e., structured) into three components: a monoid of concepts, a Boolean lattice of concepts, and a family of rooted graphs. The data view is modeled through the use of cylindric algebra (Tarski *et al.*, 1971). The two components of a DIS, the DOnt and the DDV, loosely correspond to the T-Box and A-Box of a DL ontology. In Section 3.3.1, we outline the clear separation of the

data and domain knowledge within a DIS. In Section 3.3.2, we present a overview of the intuitive understanding a DIS construction. In Section 3.3.3, we describe the DIS formalism as an algebraic theory.

3.3.1 Domain Information System: Data vs Domain Knowledge

We demonstrate the clear separation of the data and the domain knowledge in a DIS, based on a classic Kantian approach of the world. In the "Critique of Pure Reason" (Kant, 1908), the central idea is to challenge the notion that our understanding of phenomena and objects comes solely from pure reasoning. Instead, it proposes that our understanding occurs through the use of categories, described in the *Analytic of Concepts* section as "fundamental concepts of an object in general". The categories are pure (or fundamental) principles of understanding. They are not derived from experience, they are a priori structures of thought that shape and organise perceptions. We read in (Kant, 1908, Page 227):

> But the elements for all a priori cognitions, even for arbitrary and absurd fantasies, cannot indeed be borrowed from experience (for then they would not be a priori cognitions), but must always contain the pure a priori conditions of a possible experience and of an object of it, for otherwise not only would nothing at all be thought through them, but also without data they would not even be able to arise in thinking at all.

In Kantian terms, data is that what is observable, quantifiable, and experiential in the world around us. In this context, a dataset represents structured, organised information that is accessible for analysis through cognitive abilities. Kant distinguishes between two such abilities: the ability to receive sensations from external objects, shaping our experiences through empirical intuition, and the capacity to actively process and relate this intuitive data using concepts, known as understanding. This process involves forming judgments about the information gathered. Thus, once experienced, the world around us is represented and analysed through a more abstract, objective lens, using categories.

In a DIS, these two approaches are separated into the two main components of the information system. The DOnt component offers a view of the domain concerned with how the objects behave, in the sense of what their structure is and what relationships exist between them. Thus, the objects of the world are grouped into categories, which we call *objective concepts* or, on short, *concepts*. The DDV component offers a view of the domain that is concerned only with data: its structure and the means to combine pieces of data. We slice the data schema into individual attributes, and refine

the notion of *sort* to denote these slices. At the DIS level, the objective concepts, as defined within the DOnt, assume a new perspective as data-driven concepts, which we call *datascape concepts*. We borrow the *datascape* term from the field of architecture. In the architectural field, the term datascape refers to a dynamic, multidimensional data-driven map that assists and guides the urban designer in new developments or the creation of public policies (Amoroso, 2010). In the field of knowledge representation, this term refers to a landscape or visualisation of data, and it emphasises the process of converting data into conceptual representations that make it easier to understand complex information.

3.3.2 Intuitive Understanding

In developing the new DIS formalism, our focus is on representing structured and semi-structured data in a unified way. The data elements may have uniform structure, as in the records in a dataset that share the same schema, or variable structure, as in entries of log files that may differ in length or content. To accommodate both cases, we interpret the data elements within a Cartesian space, which is the set of all possible ordered pairs formed from the elements of two or more sets (Ben-Ari, 2012). Specifically, the Cartesian space can be seen as a collection of ordered tuples, each representing a combination of data elements from different sets, facilitating structured representation and manipulation of data.

In a DIS, the concepts in the domain ontology originate from two sources: (i) the data, and (ii) the domain of application. The concepts in the domain ontology can be composed to form new concepts. We understand the composition as the Cartesian product of concepts. Similar to how the information in a dataset record is not dependant on the order of attributes in a dataset, we require the composition operator to be commutative (up to an isomorphism), associative, and idempotent. The commutativity and associativy properties ensure that the order in which concepts are composed is not relevant. The idempotency property ensures that by composing a concept with itself the system does not create a new concept, or new knowledge from the existing knowledge in the system.

As discussed in Section 2.1, in current approaches to connect an ontology to existing data, mapping the two components to each other is a time consuming task, as it needs to bridge the conceptual gap between the two layers (Xiao *et al.*, 2018). To avoid this issue, the core component of the domain ontology $\mathcal{O}$ of DIS is built using the Cartesian `partOf` relation. This relation defines how individual elements or subtuples (parts) within a tuple relate to the whole tuple. Essentially, for a given tuple, the Cartesian `partOf` relation identifies which elements or combinations of elements

are considered parts of the whole structure. A concept with no sub-parts is called an *atomic* concept or an *atom*. The construction of a DIS starts from an existing dataset that guides the design of both the Cylindric algebra of the DDV and the Boolean lattice of the DOnt. In the DDV, the attributes of the dataset schema are used to build the Boolean algebra that is the foundation of the Cylndric algebra. These attributes correspond one-to-one to a subset of atomic concepts in the DOnt. The Boolean lattice of the DOnt is freely generated from this subset of atoms, using the `partOf` relation. This makes the Boolean algebra of the DDV isomorphic to the Boolean lattice of the DOnt, and the task of mapping the data (i.e., the DDV component of the DIS) to the ontology (i.e., the DOnt component of the DIS) becomes trivial.

3.3.3 DIS Formalisation

Before giving the formal definitions of the components of a DIS, we define a rooted graph in the context of DIS. On a set of concepts C, Let $C_i \subseteq C$, $R_i \subseteq C_i \times C_i$, and $t_i \in C_i$. A rooted graph at t_i, defined as $G_{t_i} = \big(C_i, R_i, t_i\big)$, is a connected directed graph of concepts in C_i with a unique *root* at t_i. Note that in graph theory, the root serves as the origin vertex of a directed graph. We abuse the notion of root by reversing the direction of the paths. Then, in DIS, a root is the vertex where all paths of the graph end. Formally, for a rooted graph $G_{t_i} = \big(C_i, R_i, t_i\big)$, its root is given by $t_i \in C_i$ s.t. $\forall k \in C_i.\ k = t_i \vee (k, t_i) \in {R_i}^+$, where ${R_i}^+$ is the transitive closure of R_i.

Definition 3.3.1 (Domain Ontology)**.** Let $\mathcal{C} = \big(C, \oplus, \mathbf{e}_C\big)$ be a commutative idempotent monoid. Let $\sqsubseteq_C$ be the natural order on $\mathcal{C}$, defined as $\forall c, d \in C.\ c \sqsubseteq_C d \iff c \oplus d = d$. Let $\mathcal{L} = \big(L, \sqsubseteq_C\big)$ be a free Boolean lattice, generated from a set of atoms in C. Let I be a finite set of indices, and $\mathcal{G} = \{G_{t_i}\}_{t_i \in L, i \in I}$ a set of graphs rooted in L. A *domain ontology* is the structure $\mathcal{O} \stackrel{\text{def}}{=} \big(\mathcal{C}, \mathcal{L}, \mathcal{G}\big)$. □

Datasets with records of different lengths can be modeled by diagonal-free cylindric algebras, thus, the DIS domain data view component is formalised using a diagonal-free cylindric algebra. In the domain data view component, the dimensions of the cylindric algebra are given by the attributes of the considered dataset. Thus, the cylindrification operators $\mathbf{c}_\kappa$ are indexed by the elements of a set $\mathcal{U}$ isomorphic to the atoms of L, where L is the carrier set of the Boolean lattice $\mathcal{L}$ of the domain ontology, as described in Definition 3.3.1.

Definition 3.3.2 (Domain Data View associated with an ontology)**.** Let $\mathcal{O} = \big(\mathcal{C}, \mathcal{L}, \mathcal{G}\big)$ be a domain ontology. A *domain data view* associated with $\mathcal{O}$ is a diagonal-free

cylindric algebra $\mathcal{A} = \big(A, +, \star, -, 0_A, 1_A, \{\mathbf{c}_\kappa\}_{\kappa \in \mathcal{U}}\big)$, where $\mathcal{U}$ represents the finite set of attributes of the considered dataset. □

For the properties of a cylindric algebra, we refer the reader to (Tarski *et al.*, 1971). The elements of A are understood as n-dimensional objects (i.e., a set of n-dimensional points), and the Boolean operators of $\mathcal{A}$ (i.e., $+, \star, -$) create new solids on A (Tarski *et al.*, 1971). The cylindrification operator on the i-th dimension can be understood as a projection of the solid on the remaining $(k-1)$-dimensional space, which is then extended to the whole "cylinder" along the removed dimension. For a graphic representation of cylindrification in a two-dimensional space, please refer to Figure D.1, in Appendix D. In a three-dimensional space, let $a = \{(v_x, v_y, v_z)\} \subseteq X \times Y \times Z$ be an element of A. The cylindrification of a with respect to the third dimension Z is denoted by $\mathbf{c}_Z(a)$ and represents the generalisation of a over the Z-dimension. Formally, this results in the set $\mathbf{c}_Z(a) = \{(v_z, v_y, z) \mid z \in Z\}$. $\mathbf{c}_Z(a)$ corresponds to the cylinder over the base point (v_x, v_y), extending freely along the Z-axis. This operation captures the idea that we abstract away (or "forget") the specific Z-value of a, generalising the tuple by allowing any possible value in the Z domain for that coordinate.

Definition 3.3.3 (Domain Information System)**.** Let $\mathcal{O}$ be a domain ontology, $\mathcal{A}$ its associated domain data view, and $\tau : A \to L$ a mapping that relates the elements of A to the elements of the Boolean lattice in $\mathcal{O}$. A *Domain Information System (DIS)* is the structure $\mathcal{I} = (\mathcal{O}, \mathcal{A}, \tau)$ for which the following axioms hold, for any $a, b \in A$:

- $\tau(0_A) = \mathbf{e}_C$
- $\tau(1_A) = \top_{\mathcal{L}}$
- $\tau(a + b) = \tau(a) \oplus \tau(b)$ □

For a complete list of the DIS axioms, we refer the reader to Appendix D, Section D.1. The components of DIS are analogous to those proposed in (Calvanese and Franconi, 2018); the domain data view corresponds to the Data Box (D-Box), the Boolean lattice $\mathcal{L}$ to the A-Box, and the family of rooted graphs $\mathcal{G}$ (plus the domain expert-defined axioms) corresponds to the T-Box. Using the τ operator, the domain data view is mapped to the Boolean lattice. Through the use of the rooted graphs, the domain expert captures concepts related to the data view. In addition, multiple DISs, each built from a specific data view, may be required to integrate knowledge from several application domains.

3.4 Algebraic Specifications

In this section, we outline the formal framework necessary for defining the DIS language, specifying an ontology, and detailing the elements of knowledge generation within it. In order to define a language, we need to describe both its syntax and the rules that show how the language functions. This algebraic specifications framework is based on the notions of signature Σ and Σ-structure, discussed in Section 3.1. The following definitions and results are borrowed from (Hatcher and Hebert, 1993; Ehrig and Mahr, 1985).

Definition 3.4.1 (Hatcher and Hebert (1993))**.** Let $\Sigma = \big(\mathcal{F},\ \mathcal{R},\ \mathcal{C}\big)\{n_f\}_{f\in\mathcal{F}}\{n_R\}_{R\in\mathcal{R}}$ be a signature and let X be a set of variables. $T_X(\Sigma)$ denotes the *set of X-terms of type* Σ, defined inductively as follows:

- $X \cup \mathcal{C} \subseteq T_X(\Sigma)$ (basic terms)
- $\forall t_1, t_2, \ldots, t_n \in T_X(\Sigma), f \in \mathcal{F}.\ f(t_1, t_2, \ldots, t_{n_f}) \in T_X(\Sigma)$ (composite terms)
- There are no other terms in $T_X(\Sigma)$.

$\Phi(\Sigma)$ denotes the *set of formulas of type* Σ, defined inductively as follows:

- $\forall t_1, t_2, \ldots, t_{n_R} \in T_X(\Sigma), R \in \mathcal{R}.\ R(t_1, t_2, \ldots, t_{n_R}) \in \Phi(\Sigma)$ (atomic formulas)
- Let $\square \in \{\wedge, \vee, \rightarrow, \leftrightarrow\}$ be a logical operator and $Q \in \{\forall, \exists\}$ a quantifier.
  $\forall \phi, \psi \in \Phi(\Sigma), x \in X.\ \neg\phi \in \Phi(\Sigma) \wedge \phi \square \psi \in \Phi(\Sigma) \wedge Qx.\ \phi(x) \in \Phi(\Sigma)$
- There are no other formulas in $\Phi(\Sigma)$. □

Given a signature Σ and a set of variables X, $FV(t)$ denotes the set *free variables in a term* $t \in T_X(\Sigma)$, defined inductively by:

- $\forall x \in X.\ FV(x) = \{x\}$
- $\forall c \in \mathcal{C}.\ FV(x) = \varnothing$
- $\forall t_1, \ldots, t_{n_f} \in T_X(\Sigma), f \in \mathcal{F}.\ FV(f(t_1, \ldots, t_{n_f})) = \bigcup_{i\in\{1,\ldots,n_f\}} FV(t_i)$

A term t with no free variables, i.e., $FV(t) = \varnothing$, is called a *closed term*.

In a formula, the variables that occur in the scope of a quantifier are called *bound variables*. In a given formula ϕ any occurrence of a variable not bound by a quantifier is a free variable in ϕ. In a formula, a variable may have both free and bound occurrences. E.g., in the formula $\phi \rightarrow \psi$, where $\phi \equiv \forall x.\ P\,x\,y$, and $\psi \equiv Q\,x$, variable x is bound in ϕ, while it is free in ψ, and variable y is free in ϕ. A formula with no free variables is called *closed formula*.

Definition 3.4.2 (Ehrig and Mahr (1985))**.** Let Σ be a signature, X a set of variables, and L, R X-terms of type Σ, $L, R \in T_X(\Sigma)$. The triple $e = (X, L, R)$ is called an *equation* w.r.t. Σ. □

For example, given a signature Σ with a binary function f, and $x, y \in X$, we express that f is commutative through the equation $f\ x\ y = f\ y\ x$.

Definition 3.4.3 (Algebraic Specifications (Ehrig and Mahr, 1985))**.** Let Σ be a signature and E a set of equations w.r.t. Σ. The tuple $\mathcal{S} = (\Sigma, E)$ is called an *algebraic specification*. A Σ-algebra that satisfies all equations in E is called an $\mathcal{S}$-algebra. □

Extending this definition, a *specification* is the tuple $SPEC = \big(\Sigma, E, \Phi\big)$, where E is a set of equations and Φ a set of formulas w.r.t. signature Σ. A Σ-structure $\mathcal{M}$ of specification *SPEC* must satisfy all equations in E and all formulas in Φ. We introduce the notion of *parameterised specification*, which is an algebraic specification with a distinguished formal parameter part. The use of formal parameters allows the creation of modular and reusable specifications.

Definition 3.4.4 ((Ehrig and Mahr, 1985))**.** Let $\mathcal{S}_0 = (\Sigma_0, E_0), \mathcal{S}_1 = (\Sigma_1, E_1)$ be two algebraic specifications. We say that $\mathcal{S}_0$ is a *subspecification* of $\mathcal{S}_1$ (and Σ_0 is a *subsignature* of Σ_1) if the following hold:

- $\Sigma_0 \subseteq \Sigma_1$, i.e., all symbols in Σ_0 appear in Σ_1
- $\mathcal{F}_0 \subseteq \mathcal{F}_1$, i.e., all function symbols in Σ_0 appear in Σ_1
- $E_0 \subseteq E_1$, i.e., all equations in $\mathcal{S}_0$ appear in $\mathcal{S}_1$.

A *parametrised specification* is a pair of specifications $PS = (\mathcal{S}_p, \mathcal{S}_t)$, such that $\mathcal{S}_p$ is a subspecification of $\mathcal{S}_t$. The specification $\mathcal{S}_p = (\Sigma_p, E_p)$ is called the *formal parameter* and $\mathcal{S}_t = (\Sigma_t, E_t)$ is called the *target* specification or the body of PS. □

3.5 Conclusion

This chapter presents the mathematical foundations underpinning the DIS theory. We review key structures such as Boolean lattices, cylindric algebras, and rooted graphs, which together support both the data and conceptual components of DIS. These structures enable a formal representation of domain knowledge and its associated data, with particular attention to modularity, abstraction, and formal reasoning.

We then introduce the DIS itself, a novel, data-centered knowledge representation formalism that separates structured data (captured in the Domain Data View (DDV)) from domain knowledge (captured in the Domain Ontology (DOnt)). The DDV is formalised using diagonal-free cylindric algebra, which provides a powerful abstraction for reasoning over data. The DOnt, in turn, is formalised using Boolean lattices and concept monoids, enabling compositional reasoning about domain knowledge. By treating these components independently and linking them through a formal mapping operator, DIS supports modular, semantically coherent integration of data and knowledge. These components are connected algebraically, forming a coherent and expressive system in which both data and knowledge can be specified, reasoned about, and evolved in tandem. The mapping operator enables automated alignment between datasets and domain concepts, alleviating the manual burden of traditional ontology engineering approaches.

Finally, we introduce the algebraic specification language, which will be used to define and reason about DIS in Isabelle/HOL, in Chapter 5. This foundation prepares us for the next steps, in which we define the DIS syntax, specify its semantics , and implement reasoning tasks in a machine-verifiable and modular way, in Chapters 4, 5. The use of parameterised and compositional specifications ensures that DIS can scale across multiple application domains, each grounded in their own data schema and conceptual structures.

Chapter 4

Semantics of Domain Information System

In this chapter, we explore the DIS framework in more depth, detailing the construction and integration of multiple DISs.

Model theory (Hodges *et al.*, 1993) studies the interpretation of formal and informal languages, by associating them with set-theoretic structures. Within these structures, the elements of the universe of discourse are grouped into classes, commonly referred to as *sorts*. A sort specifies a category of objects, and the expression $t : S$ denotes that an object t belongs to sort S. The interpretation of this relationship is set-theoretic: given an interpretation t^I, S^I, the expression $t : S$ holds if and only if $t^I \in S^I$. This set-based interpretation of sorts is widely used in algebraic specification and abstract data type theory (Ehrig *et al.*, 1982), where sorts represent data domains, independent of any operational behaviour. Sorts are thus purely classificatory, they group objects according to abstract criteria, without implying how the objects behave or interact. In contrast, a *type* is concerned not just with membership but with the behaviour and operations applicable to the objects (Harper, 2016). Types define what can be done with values of a certain kind, and how they interact under defined functions or constraints. For example, the sort *Integer* represents all whole numbers, while the type *GroupElement* includes integers equipped with addition and inverses satisfying group axioms.

In the context of this work, it is also useful to distinguish sorts from concepts. While a sort classifies objects at the data level (i.e., in the domain data view), a concept refers to a more abstract categorisation relevant at the domain knowledge level. For instance, a value may be of sort *Integer* and also belong to the concept *Age*, where

the latter captures contextual or semantic information beyond the structural classification, such as representing a temporal property of an entity (e.g., a person or wine), being measured in years, possibly being bounded (e.g., non-negative), and participating in domain-specific constraints (e.g., legal drinking age, wine maturity classifications).

In Section 3.3.1, we have detailed how the DDV is used to represent organised, data-driven information depicting the phenomena reality within a specific domain. Considering the domain of application from this perspective, the attributes of the data are grouped in sorts, as we are not concerned with the behaviour of objects within the domain of application, only with their structure. The DDV includes components such as the universe (i.e., a set of sorts), s-values, s-datums, and s-datas. These components are defined and discussed in more detail in Section 4.3.1. In contrast, the objective perspective of the world is captured through the DOnt, which consists of concepts and the relationships between them. We designate the DDV as the "data-driven" or "evidence-based" reality, while the DOnt signifies the "objective" reality. The DIS encompasses both these realities.

In Section 4.1, we present the DIS language and its syntax. In Section 4.2, we introduce the example used to illustrate the construction of a DIS. In Sections 4.3, 4.4, and 4.5, we detail and illustrate the model for each DIS component: its domain data view, domain ontology, and mapping operator, respectively. Using the DIS syntax and semantics, in Section 4.6, we detail a way to capture refined knowledge within a DIS. Finally, in Section 4.7, we discuss the main advantages of using DIS to formalise domain knowledge, as well as its limitations.

4.1 Domain Information System: Syntax

The DIS is a heterogeneous theory, and its alphabet is based on the set $S = \{C, L, A, \mathcal{U}\}$ of domains, where C is the domain associated with the concepts contained in the DOnt component, L the domain associated with the concepts contained in the Boolean lattice, A the domain associated with the elements of the DDV component of the DIS, and $\mathcal{U}$ the domain associated with the set of sorts of the DDV. The concepts in L have a direct link to the data in the corresponding DDV (through the mapping operator τ), while the other concepts in C are linked to their datasets through external DISs. For this reason, we consider L and C to be different domains, despite the fact that $L \subseteq C$.

Definition 4.1.1 (DIS Signature)**.** Let $\mathcal{I} = (\mathcal{O}, \mathcal{A}, \tau)$ be a Domain Information System. The signature Σ_{DIS} is given by the tuple $\Sigma_{DIS} = (\mathcal{F}, \mathcal{R})$, where

$\mathcal{F} = \{\oplus, \otimes, \ominus, \mathbf{e}_C, \top_{\mathcal{L}}, +, \star, -, \mathbf{0}, \mathbf{1}, \tau, cyl\}$ is the set of function symbols and $\mathcal{R} = \{\sqsubseteq_C, \leqslant\} \cup \{R_i\}_{i \in I}$ is the set of relation symbols, with I is a set of indeces. The arity mappings $r_{\mathcal{F}}, r_{\mathcal{R}}$ are defined as follows:

$$\begin{array}{lll} r_{\mathcal{F}}(\oplus) = (C.C, C) & r_{\mathcal{F}}(\otimes) = (L.L, L) & r_{\mathcal{F}}(\tau) = (A, L) \\ r_{\mathcal{F}}(+) = r_{\mathcal{F}}(\star) = (A.A, A) & r_{\mathcal{F}}(cyl) = (A.\mathcal{U}, A) & r_{\mathcal{R}}(\sqsubseteq_C) = (L, L) \\ r_{\mathcal{F}}(\mathbf{e}_C) = r_{\mathcal{F}}(\top_{\mathcal{L}}) = (e, L) & r_{\mathcal{F}}(\ominus) = (L, L) & r_{\mathcal{R}}(R_i) = (C, C) \\ r_{\mathcal{F}}(\mathbf{0}) = r_{\mathcal{F}}(\mathbf{1}) = (e, A) & r_{\mathcal{F}}(-) = (A, A) & r_{\mathcal{R}}(\leqslant) = (A, A) \end{array}$$

□

For ease of reading, we denote $cyl(\kappa, a)$ by $\mathbf{c}_\kappa a$. In addition, we define three countable sets of variables: (i) $\mathcal{X}^C$, the set of variables of C that we denote by k, k_1, k_2, etc.; (ii) $\mathcal{X}^L$, the set of variables of L or $\mathcal{U}$ that we denote by $\kappa, \lambda, \kappa_1, \kappa_2$, etc.; and (iii) $\mathcal{X}^A$, the set of variables of A that we denote by a, b, a_1, a_2, etc. Let $\mathcal{X} = \{\mathcal{X}^s\}_{s \in S}$ be the S-indexed set. The non-logical symbols of the DIS-based language are provided by the functions $f \in \mathcal{F}$, relations $R \in \mathcal{R}$, and variables in $\mathcal{X}$. In addition to the S-indexed alphabet Σ_{DIS}, the language contains the following symbols: (i) parenthesis and brackets; (ii) a relational symbol ; that denotes the composition of relations; and (iii) logical symbols $\wedge, \vee, \neg, \implies, \forall, \exists, True,$ and $False$.

The DIS-based expressions (terms and formulae) are built inductively over this language, as described in Section 3.4. The theory of a DIS is the S-indexed alphabet Σ_{DIS} described above, together with the axioms obtained from Definition 3.3.3.

In Section 3.3.3, we have shown that by using a DIS, a domain expert is able to capture the structural information about the domain of application. Additional domain knowledge concepts can be introduced into the DIS, by defining composite concepts or by defining new datascape concepts, as described in Sections 4.4, and 4.6, respectively. These concepts are defined as algebraic terms constructed over the language of the Σ_{DIS}. During the reasoning process, the concept definitions will be treated as axioms, thus they will be called *axioms* throughout this work.

4.2 Domain Information System: A Running Example

As discussed in Section 3.3.3, the DIS is a bottom-up data-centred formalism for knowledge representation. The construction of a DIS model (called simply a DIS when it is clear from the context) is guided by an existing dataset. The dataset is

used to model both the DDV component and the core component of the DOnt (i.e., its Boolean lattice). Using the mapping operator τ, the Boolean algebra component of the DDV is mapped to the Boolean lattice component of the DOnt. This part of a DIS construction can be automated. To enrich the DOnt component, the domain expert adds other concepts related to the concepts of the Boolean lattice, thus related to the data through relations from the domain of application. They are captured in the rooted graphs of the DOnt. This part of construction can be semi-automated, by using templates, as discussed and illustrated in Chapter 6.

Both Boolean constructions are of manageable size. This is because in practice, in a normalised database, the number of attributes in a dataset schema is kept low, rarely exceeding ten (Vassiliadis *et al.*, 2015). Thus, the number of atoms in the Boolean algebra at the DDV level, as well as the number of atoms in the Boolean lattice at the DOnt level rarely exceeds ten, putting the number of elements in the Boolean construction below 1,024, for most cases.

Throughout this chapter, interspersed with the DIS model, we illustrate the construction and integration of a number of DISs for the Film and TV domain of application. We use public data available on the IMDb website (IMDb, 2020) and the Rotten Tomatoes website (accessed through the Open Movie Database (OMDb) (OMDbAPI, 2019)), and offer three perspectives (or views) on the application domain. The first perspective V_M, is the Media view, guided by the IMDb Titles dataset ('Titles.basic' in the IMDb database). The IMDb Titles dataset contains general data about movies and TV shows, such as title, genre, year, and more. The second perspective, V_P, is the People view, guided by the IMDb Names dataset and the IMDb Principals dataset ('Names.basic', 'Titles.Principals', respectively in the IMDb database). The first dataset contains details about people who work in the Film and TV industry, such as name and birth date. The second dataset relates people to titles through the main credits. Finally, the third perspective, V_R, is the Reviews view, guided by the OMDb dataset, which contains both critics' and fans' reviews on titles.

A partial view of the IMDb Titles dataset is presented in Table 4.1. It contains attributes for the unique identifier of a record (*tconst*), the title of the movie or series (*title*), and more. An entry '\N' indicates an empty value.

4.2.1 The Competency Questions

A domain expert would build a Film and TV domain DIS with the goal of finding information (that is not provided explicitly through the data) about movies (using in the V_M view), their credits (V_P view), or their ratings (V_R view). In order to

tconst	titleType	title	originalTitle	startYear	endYear	runtime	genre
tt0944947	tvSeries	Game of Thrones	Game of Thrones	2011	2019	57	Fantasy
tt0377092	movie	Mean Girls	Mean Girls	2004	\N	\N	Comedy
tt0087800	movie	A Nightmare on Elm Street	A Nightmare on Elm Street	1984	\N	91	Horror
tt0076759	movie	Star Wars IV: A New Hope	Star Wars	1977	\N	121	Adventure
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

Table 4.1: IMDb Titles dataset: Partial View

limit its scope, the construction of the DIS is guided by a number of Competency Questions (CQ)s, listed below. We show the view(s) associated with each CQ within parentheses:

CQ1 (V_M) What movies are slashers?

CQ2 (V_M, V_P) What slashers are a critically acclaimed?

CQ3 (V_P) Are there any infamous fresh media?

CQ4 (V_M, V_R) Who are the actors credited in the "Ghostbusters 2016" movie?

Some of the notions used in the CQs above (such as *slasher*, *infamous*, *fresh*) cannot be found in the given data schema and cannot be made from a simple combination of data attributes. These notions are defined by the domain expert, as datascape concepts. The process of defining new datascape concepts is described in more detail in Section 4.6. Thus, the CQs using them, i.e., CQ1, CQ2, and CQ3, cannot be answered through a simple database query.

In addition, some of the CQs (such as CQ2 and CQ4) use more than one view. The framework enables the integration of multiple DISs, to capture and use knowledge from multiple application domains (or views). In Figure 4.1, we illustrate the DIS integration, with DIS $\mathcal{I}_M = (\mathcal{O}_M, \mathcal{A}_M, \tau_M)$ depicted in red, and DIS $\mathcal{I}_R = (\mathcal{O}_R, \mathcal{A}_R, \tau_R)$ depicted in blue. The overlap shows concepts common to both $\mathcal{I}_M$ and $\mathcal{I}_R$ (i.e., *Fresh*), or defined by the domain expert to be equivalent (i.e., $tconst \equiv imdbID$ and $\bot_{\mathcal{L}_M} \equiv \bot_{\mathcal{L}_R}$).

4.3 DIS Model: Domain Data View Component

Since the construction of a DIS is a bottom-up process, we give the model of the DIS theory in the same manner, starting from the DDV, and working up to the DOnt structure, and the mapping operator.

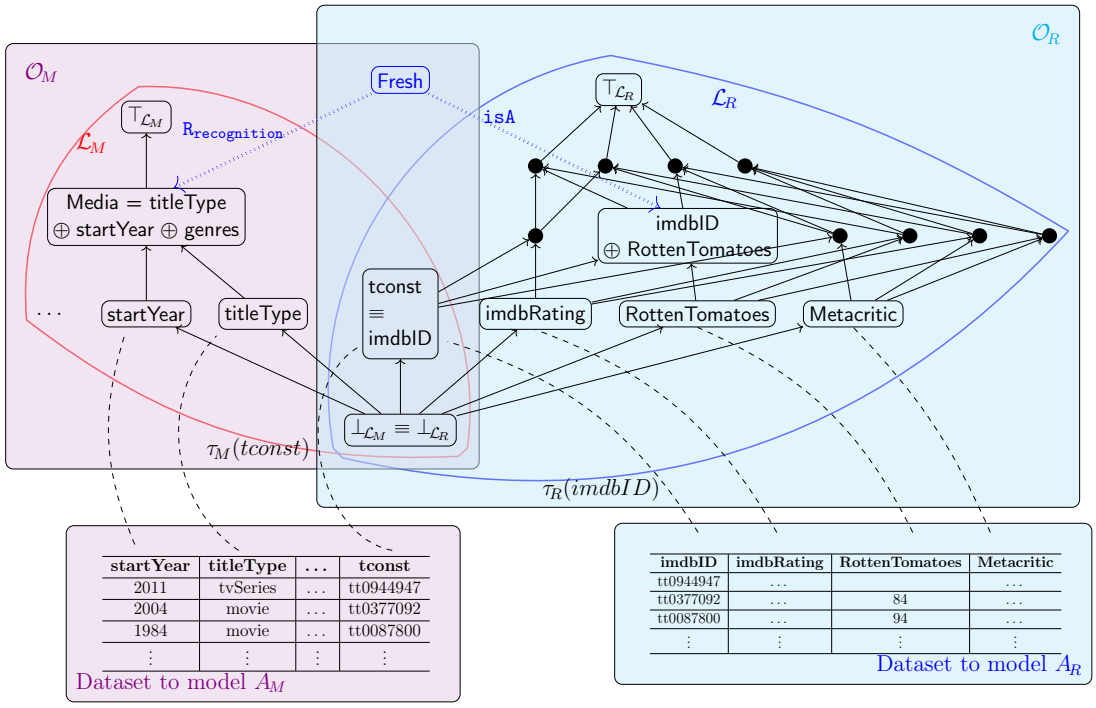

Figure 4.1: Integrating Multiple DISs

In this Section, we detail the proposed model for the domain data view. In Section 4.3.1, we present the foundational elements of the model, along the interpretation of the DDV carrier set. In Section 4.3.2, we follow with an illustration of these elements using the example provided in Section 4.2. In Section 4.3.3, we discuss a set of helper data operators and present their properties. In Section 4.3.4, we present the interpretation of the DDV operators, and we illustrate the application of the cylindrification operator(s) using the example. Finally, we show that the proposed structure is a model for a diagonal-free cylindric algebra, thus it is a model for the DDV component of a DIS.

4.3.1 Foundational Elements

We start the interpretation of the DDV from a finite set of sorts. These sorts do not represent the abstract sort symbols of the formal signature Σ_{DIS} defined in Section 4.1. The sorts each correspond to an attribute of the dataset, with each sort interpreted as a finite set of elements associated with that attribute. Note that while two sorts may have an equivalent interpretation (e.g., the sorts corresponding to StartYear and EndYear have the same elements), they are semantically considered two different sorts and they are interpreted, in Section 4.4, as types with specific meaning, rules,

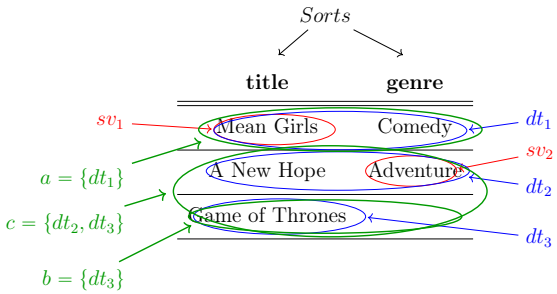

Figure 4.2: Film & TV Domain Ontology dataset example

and operations. The set of sorts in a DDV is called *the universe*, and is denoted by $\mathcal{U} = \{S_1, S_2, \ldots, S_n\}$, with each $S_i \in \mathcal{U}$ corresponds to exactly one attribute of the dataset, s.t. $S_i = \{v\}$, where the elements v represent the values of the corresponding attribute. For the remainder of this work, unless otherwise specified, we refer to this interpretation of the DIS components, and when we use s for a sort in $\mathcal{U}$, we understand its interpretation (i.e., a set of values). The building blocks of the cylindric algebra are ordered pairs consisting of a sort and a value of that sort. We call such a pair a *sorted value*, in short, *s-value*. Note that two s-values with the same value and different sorts, e.g., $(StartYear, 2000)$ and $(EndYear, 2000)$, are not equivalent and represent two semantically different values. We take a nonempty set of s-values such that there is at most one s-value representing each sort and we call it a *sorted datum* or, in short, *s-datum*. We call a set of s-datums a *sorted data*, in short, *s-data*. We refer to the set of sorts that appear in an s-data as its *structure*. The elements of the carrier set of the cylindric algebra A are interpreted as s-datas, i.e., each element of A is a set of s-datums. We denote sort variables by s, s_1, etc., and elements of sorts (or values) by v, v_1, etc. Unless otherwise mentioned, we denote the sort corresponding to the attribute 'attr' of the datatset by S_{attr}. We denote s-values by sv, sv_1, sv_a, etc., s-datums by dt, dt_1, dt_a, etc., and s-datas by a, b, x, y, a_1, x_1, etc.

For the cylindric algebra $\mathcal{A}$, we offer two models that differ only in the origin of the elements of the sorts. The first approach is grounded in the existing data, and we call it the *evidence-based* approach. The second approach is grounded in the information provided by the domain expert, and we call it the *expert-based* approach. When using the evidence-based approach, the elements of the sorts are provided exclusively by the current evidence (i.e., existing data). The data is considered clean, and the sorts are generated from the values found in the data. When using the expert-based approach, the elements of the sorts are captured by the domain expert, from the domain knowledge. In this approach, the data may not be clean, in the sense that values found

in the data may not be found in the expert-provided sorts. Thus, the sorts may be used to eliminate (or correct) the s-datas containing such s-values. While, in our work, we focus on the evidence-based approach, in this section we also present examples of evaluations of various elements of the DIS using the expert-based approach.

Take the partial dataset example of the Film and TV Domain Ontology, pictured in Figure 4.2. The universe is defined by $\mathcal{U} = \{S_t, S_g\}$, where the sorts S_t, S_g correspond to the attributes *title* and *genre*, respectively, i.e., $S_t = \{Mean\,Girls, A\,New\,Hope, Game\,of\,Thrones\}$ and $S_g = \{Comedy, Adventure\}$. Some s-values are $sv_1 = (S_t, Mean\,Girls)$ and $sv_2 = (S_g, Adventure)$. Some s-datums are $dt_1 = \{(S_t, Mean\,Girls), (S_g, Comedy)\}$, $dt_2 = \{(S_t, A\,New\,Hope), (S_g, Adventure)\}$, $dt_3 = \{(S_t, Game\,of\,Thrones)\}$. Some s-datas are $a = \{dt_1\}$, $b = \{dt_3\}$, and $c = \{dt_2, dt_3\}$. Based on the evidence provided, the genre of a movie can only be *Comedy* or *Adventure*. However, a domain expert may be aware of movies that have a different genre, such as *Fantasy*. The information provided in s-data b is incomplete, as it does not contain an s-value for the S_g category. The two approaches open up the world in a different manner. In the evidence-based approach, the title *GameofThrones* can be classified either as a *Comedy* or an *Adventure* movie, as there is no evidence for the existence of another genre. In the expert-based approach, the domain expert can provide the sort $S_g = \{Comedy, Adventure, Fantasy\}$. Thus, the title *GameofThrones* could, in addition, be classified as a *Fantasy* movie, even if there is no evidence for such a genre in the existing dataset. On the other hand, any s-data that contains the s-value $sv = (Game\,of\,Thrones, Epic\,Fantasy)$ is not part of the set of possible s-datums and it may be cleaned. This can be done either by simply removing all such s-datas from the dataset, or by correcting the S_g s-value in it to match the sort values, based on the domain expert knowledge. Using the latter approach, the s-value may be corrected to $sv = (Game\,of\,Thrones, Fantasy)$.

The Domain Data View model(s) are specified by the mathematical structure $\mathcal{M}_{\mathcal{A}} = \big(A_{D,\mathcal{U}}, +, \star, -, \mathbf{0}, \mathbf{1}, \{\mathbf{c}_\kappa\}_{\kappa\in\mathcal{U}}\big)$. The carrier set of the cylindric algebra $(A_{D,\mathcal{U}})$ is freely generated by all the operators over a set of elements, called the *generating set*. The elements of the generating set are called *generators*. The generating set is, in turn, generated from the universe $\mathcal{U}$. The set of s-datums D, which represents the existing dataset, may be used to create the universe $\mathcal{U}$ and we detail the process below. When it is clear from the context what we refer to, we denote the carrier set by A. For a given universe $\mathcal{U}$, let SV be the set of all possible s-values, and SD the set of all possible s-datums or the generating set. We show next how these sets are generated.

The set of s-values is defined as

$$SV \stackrel{\text{def}}{=} \bigcup_{s \in \mathcal{U}} \{(s, v) \mid v \,.\, v \in s\}$$

For ease of readability, we define a helper operator, $\bowtie$. The operator $\bowtie$ denotes a special product (similar to the Cartesian product) on a subset of sorts, $\bowtie : \mathcal{P}(\mathcal{U}) \rightarrow \mathcal{P}(SV)$. When it is clear from the context, we write $\bowtie T$ instead of $\bowtie(T)$. The $\bowtie$ operator is defined by induction, as follows, for any $s \in \mathcal{U}, T \subseteq \mathcal{U}, s \notin T$:

$$\begin{aligned} \bowtie \{\} &= \{\} \\ \bowtie (T \cup \{s\}) &= \bowtie T \cup \big\{dt \cup \{(s, v)\} \mid dt, v \,.\, dt \in \bowtie T \wedge v \in s\big\} \end{aligned} \tag{4.1}$$

The set of s-datums is defined as $SD \stackrel{\text{def}}{=} \bowtie \mathcal{U}$ and it is the generating set of the cylindric algebra. Finally, the carrier set of the cylindric algebra is defined as $A_{D,\mathcal{U}} = \mathcal{P}(SD)$. Note that in this model, due to the definition of A and the powerset axiom $(x \in \mathcal{P}(S) \iff x \subseteq S)$, it is immediate that

$$a \in A \iff a \subseteq \bowtie \mathcal{U} \tag{4.2}$$

$$s \notin T \implies \bowtie(\{s\}) \nsubseteq \bowtie T \;. \tag{4.3}$$

With both the evidence-based and expert-based approaches, the sets above are generated the same way. First all s-values are generated from the sorts in $\mathcal{U}$ to form SV, next all the possible s-datum combinations are generated from the s-values in SV to form SD, and then all the possible s-datas are generated from the s-datums in SD to form $A_{D,\mathcal{U}}$. The main difference between the two approaches is the origin of the elements of the sorts. In the evidence-based approach, the sorts are built from the existing set of s-datums, D. In contrast, in the expert-based approach, the elements of the sorts are specified by the expert domain through the universe $\mathcal{U}$ and independently of D. Note that in the evidence-based approach, $D \subseteq A^{\mathcal{E}}$, while in the expert-based approach, D may not be a subset of A. While the names of the sorts of D always coincide with the sorts of the universe $\mathcal{U}$, the content of the sorts may differ.

4.3.2 Illustration

Taking the example of Figure 4.2, we show how to build the generating set. We annotate with $^{\mathcal{E}}$ and $^{\mathcal{X}}$ the sets in the evidence-based approach and expert-based approach, respectively. In the evidence-based approach, the sorts are defined by the set of values found in the data, i.e., ${S_t}^{\mathcal{E}} = \{\mathit{Mean\,Girls}, \mathit{A\,New\,Hope}, \mathit{Game\,of\,Thrones}\}$ and ${S_g}^{\mathcal{E}} = \{\mathit{Comedy}, \mathit{Adventure}\}$. Then ${SV}^{\mathcal{E}} = \{(S_t, \mathit{Mean\,Girls}),$

$(S_t, A\,New\,Hope), (S_t, Game\,of\,Thrones), (S_g, Comedy), (S_g, Adventure)\}$,
$SD^{\mathcal{E}} = \big\{\{\}, \{(S_t, Mean\,Girls)\}, \{(S_t, A\,New\,Hope)\}, \{(S_t, Game\,of\,Thrones)\}$,
$\{(S_g, Comedy)\}, \{(S_g, Adventure)\}, \{(S_t, Mean\,Girls), (S_g, Comedy)\}$,
$\{(S_t, Mean\,Girls), (S_g, Adventure)\}, \{(S_t, A\,New\,Hope), (S_g, Comedy)\}$,
$\{(S_t, A\,New\,Hope), (S_g, Adventure)\}, \{(S_t, Game\,of\,Thrones), (S_g, Comedy)\}$,
$\{(S_t, Game\,of\,Thrones), (S_g, Adventure)\}\big\}$, and $A^{\mathcal{E}} = \mathcal{P}(SD^{\mathcal{E}})$.

With the expert-based approach, while the process to construct the three sets is identical, what may differ is the content of the sorts. With this approach, the domain expert infuses their knowledge of the domain into the content of the sorts. Possible evaluations for this approach are ${S_t}^{\mathcal{X}} = {S_t}^{\mathcal{E}} \cup \{A\,Nightmare\,on\,Elm\,Street\}$ and ${S_g}^{\mathcal{X}} = {S_g}^{\mathcal{E}} \cup \{Fantasy\}$. Thus, $SV^{\mathcal{X}} = SV^{\mathcal{E}} \cup \{(S_t, A\,Nightmare\,on\,Elm\,Street), (S_g, Fantasy)\}$,
$SD^{\mathcal{X}} = SD^{\mathcal{E}} \cup \big\{\{(S_t, A\,Nightmare\,on\,Elm\,Street)\}, \{(S_g, Fantasy)\}$,
$\{(S_t, A\,Nightmare\,on\,Elm\,Street), (S_g, Comedy)\}$,
$\{(S_t, A\,Nightmare\,on\,Elm\,Street), (S_g, Adventure)\}$,
$\{(S_t, A\,Nightmare\,on\,Elm\,Street), (S_g, Fantasy)\}$,
$\{(S_t, Mean\,Girls), (S_g, Fantasy)\}, \{(S_t, A\,New\,Hope), (S_g, Fantasy)\}$,
$\{(S_t, Game\,of\,Thrones), (S_g, Fantasy)\}\big\}$, and $A^{\mathcal{X}} = \mathcal{P}(SD^{\mathcal{X}})$.

4.3.3 Operators on Data and their Properties

To support data manipulation and reasoning within a structured domain, we define a series of helper operators that act on sorted data elements (i.e.,s-datums and s-datas). The first such operator is used to detect whether a given sort is present in an s-datum, that is, whether an s-value of a particular sort appears within a given s-datum. This is a fundamental check, as many subsequent operations rely on knowing whether a sort is active within a given s-datum. Throughout this work, we adopt infix notation for all helper operators to improve readability and maintain consistency in expressions. Formally, for a given sort $\kappa \in \mathcal{U}$ and an s-datum $dt \in SD$, we say that "κ is part of dt" if there exists a value $v \in \kappa$ s.t.κ s-value in dt. We denote this by $\kappa \mathrel{\overline{\in}} dt$. By construction of s-datums, the following property holds for any $dt \in SD, \kappa \in \mathcal{U}$, and $v \in \kappa$:

$$\kappa \mathrel{\overline{\in}} dt \iff \exists v \in \kappa.\ dt = \big(dt\big\downarrow_{\kappa} \cup \{(\kappa, v)\}\big) \tag{4.4}$$

The next two operators remove specific s-values from a given s-datum or s-data. The first such operator is called *s-datum-reduction-by-sort* and maps an s-datum and a sort to a new s-datum in which the s-value corresponding to the given sort is removed (if present). Its application to a specific s-datum $dt \in SD$ and sort $\kappa \in \mathcal{U}$ is called the *κ-reduction of dt*, and the resulting s-datum is referred to as *κ-reduced dt*. The s-datum-reduction-by-sort operator is denoted by $\big\downarrow : SD \times \mathcal{U} \to SD$, and is defined

for any $dt \in SD$, $\kappa \in \mathcal{U}$ as follows:

$$dt\big\downarrow_{\kappa} \stackrel{\text{def}}{=} \{(s, v) \mid s, v \,.\, (s, v) \in dt \wedge s \neq \kappa\} \tag{4.5}$$

Similarly, the second operator is called *s-data-reduction-by-sort* and maps an s-data and a sort to a new s-data in which all s-values corresponding to the given sort are removed from all s-datums in the s-data. Its application to a specific s-data $a \in A$ and sort $\kappa \in \mathcal{U}$ is called the *κ-reduction of a*, and the resulting s-data is referred to as *κ-reduced a*. The s-data-reduction-by-sort operator is denoted by $\Downarrow : A \times \mathcal{U} \to A$ and is defined for any $a \in A$, $\kappa \in \mathcal{U}$ as follows:

$$a\Big\Downarrow_{\kappa} \stackrel{\text{def}}{=} \{dt\big\downarrow_{\kappa} \mid dt \,.\, dt \in a\} \tag{4.6}$$

The final two operators extend a given s-datum or s-data with all the s-values specific to a given sort, only if the sort is present in the given s-datum or in any s-datum of the given s-data. The first such operator is called *s-datum-extension-by-sort*, its application to a specific s-datum $dt \in SD$ and sort $\kappa \in \mathcal{U}$ is called the *κ-extension of dt*, and the resulting s-data is referred to as *κ-extended dt*. The s-datum-extension-by-sort operator is denoted by $\uparrow : SD \times \mathcal{U} \to A$, and is defined for any $dt \in SD$. $\kappa \in \mathcal{U}$ as follows:

$$dt\big\uparrow^{\kappa} = \begin{cases} \left\{ \left(dt\big\downarrow_{\kappa} \cup \{(\kappa, v)\}\right) \mid v \,.\, v \in \kappa \right\} & \text{if } \kappa \,\overline{\in}\, dt \\ \{dt\} & \text{otherwise} \end{cases} \tag{4.7}$$

Similarly, the second extension operator is called *s-data-extension-by-sort*, its application to a specific s-data $a \in A$ and sort $\kappa \in \mathcal{U}$ is called the *κ-extension of a*, and the resulting s-data is referred to as *κ-extended a*. The s-data-extension-by-sort operator is denoted by $\Uparrow : A \times \mathcal{U} \to A$, and is defined for any $a \in A$, $\kappa \in \mathcal{U}$ as follows:

$$a\Big\Uparrow^{\kappa} \stackrel{\text{def}}{=} \bigcup_{dt \in a} dt\big\uparrow^{\kappa} \tag{4.8}$$

Note the use of κ-reduced s-datum $dt\big\downarrow_{\kappa}$ in the definition (4.7) of the s-datum-extension-by-sort operator. This ensures that all resulting s-datums are well formed, which means that each s-datum contains exactly one κ s-value. There exists already a κ s-value (κ, v) in the original s-datum dt, and if the κ sort contains more than one value, say $v' \in \kappa, v \neq v'$, then naively extending dt with all κ s-values would lead to malformed s-datum. Specifically, the resulting s-datums would contains multiple s-values for the sort κ, such as (κ, v) and (κ, v'). To prevent this, we first apply the

κ-reduction on dt to remove any existing κ s-values. The extension is then safely applied to the reduced s-datum, guaranteeing that each extended s-datum includes exactly one κ s-value and remains well formed.

We take the example of Figure 4.2, with $dt_1\big\downarrow_{S_g} = \{(S_t, \mathit{Mean\,Girls})\}$, $dt_2\big\downarrow_{S_g} = \{(S_t, \mathit{A\,New\,Hope})\}$, $dt_3\big\downarrow_{S_g} = \{(S_t, \mathit{Game\,of\,Thrones})\} = dt_3$, $a = \{dt_1\}$, $b = \{dt_3\}$, and $c = \{dt_2, dt_3\}$. Reducing dt_1 and dt_2 on S_g has the effect of removing their respective S_g s-value. At the same time, reducing dt_3 on S_g has no effect, since the s-datum dt_3 has no S_g s-value. Similarly, $a\Big\Downarrow_{S_g} = \big\{dt_1\big\downarrow_{S_g}\big\} = \big\{\{(S_t, \mathit{Mean\,Girls})\}\big\}$, $b\Big\Downarrow_{S_g} = \big\{\{(S_t, \mathit{Game\,of\,Thrones})\}\big\} = b$, and $c\Big\Downarrow_{S_g} = \big\{\{(S_t, \mathit{A\,New\,Hope})\}, \{(S_t, \mathit{Game\,of\,Thrones})\}\big\}$. Reducing a and c on S_g has the effect of removing the S_g s-values from all the contained s-datums. However, reducing b on S_g has no effect, as the s-datum contained in b had no S_g s-value to start with. In general, an s-datum (or s-data) that has been reduced on a certain dimension may have its structure changed from the original s-datum (or s-data). In the example above, the dt_1, dt_2, a, and c reduced on S_g all have their structure modified. dt_3 and b remain unchanged in structure (and content).

In contrast, extending an s-datum or s-data on a given sort may only modify the content of the original elements, and not its structure. In the evidence-based approach, extending the s-datums and s-datas on the S_g dimension produces the following s-datas:

$$
\begin{aligned}
dt_1\big\uparrow^{S_g} &= \big\{\{(S_t, \mathit{Mean\,Girls}), (S_g, \mathit{Comedy})\}, \{(S_t, \mathit{Mean\,Girls}), (S_g, \mathit{Adventure})\}\big\}\\
dt_2\big\uparrow^{S_g} &= \big\{\{(S_t, \mathit{A\,New\,Hope}), (S_g, \mathit{Comedy})\}, \{(S_t, \mathit{A\,New\,Hope}), (S_g, \mathit{Adventure})\}\big\}\\
dt_3\big\uparrow^{S_g} &= \big\{\{(S_t, \mathit{Game\,of\,Thrones})\}\big\}\\
a\Big\Uparrow^{S_g} &= dt_1\big\uparrow^{S_g}\\
b\Big\Uparrow^{S_g} &= dt_3\big\uparrow^{S_g}\\
c\Big\Uparrow^{S_g} &= \big\{\{(S_t, \mathit{A\,New\,Hope}), (S_g, \mathit{Comedy})\}, \{(S_t, \mathit{A\,New\,Hope}), (S_g, \mathit{Adventure})\},\\
&\qquad \{(S_t, \mathit{Game\,of\,Thrones})\}\big\}
\end{aligned}
$$

The structure of the s-datas a and c includes the S_g sort. Thus, by extending a and c on S_g we obtain s-datas with the same structure as the original s-data, and extended content. The two s-datas now contain s-datums for which the original S_g s-value has been replaced with all the possible s-values of the S_g sort. In contrast, the structure of the s-data b does not include the S_g sort. Thus, extending b on S_g has no effect (in either structure or content) to the original s-data.

We present below properties that the operators $\bowtie$, $\downarrow$, $\Downarrow$ and $\Uparrow$ exhibit. The operator $\uparrow$ behaves exactly like $\Uparrow$, as the extension of an s-datum dt can be seen as the extension of the singleton s-data containing only dt. Thus, any properties of $\Uparrow$ hold for $\uparrow$. The detailed proofs for all these properties are provided in Appendix A, Section A.1.

Proposition 4.3.1. Let $\mathcal{U}$ be a DIS universe. For any $s \in \mathcal{U}, T \subseteq \mathcal{U}$:

1. $s \notin T \implies \bowtie\{s\} \cap \bowtie T = \{\}$
2. $\forall v \in s, dt \in SD.\ s \notin T \implies \big(dt \cup \{(s, v)\}\big) \notin \bowtie T$ □

Proposition 4.3.2. Let $\mathcal{U}$ be a DIS universe and $\bowtie$ the special product operator defined in 4.1. The $\bowtie$ operator is montone, i.e., for any $T, T' \subseteq \mathcal{U}$ the following statement is true: $T' \subseteq T \implies \bowtie T' \subseteq \bowtie T$

Proposition 4.3.3. Let $\mathcal{U}$ be a DIS universe and $\bowtie$ the special product operator defined in 4.1. For any $s \in \mathcal{U}, T \subseteq \mathcal{U}$, $s \in T \implies \bowtie\{s\} \cap \bowtie T = \bowtie\{s\}$. □

Proposition 4.3.4. Let $\mathcal{U}$ be a DIS universe and $\bowtie$ the special product operator defined in 4.1. The $\bowtie$ operator distributes over intersection, i.e., for any $T_1, T_2 \subseteq \mathcal{U}$, $\bowtie T_1 \cap \bowtie T_2 = \bowtie (T_1 \cap T_2)$. □

Proposition 4.3.5. Let $\mathcal{U}$ be a DIS universe. For any $\kappa, \lambda \in \mathcal{U}$ and $v \in \kappa$

$$\{(\kappa, v)\}\big\downarrow_{\lambda} = \begin{cases} \{(\kappa, v)\} & \text{if } \kappa \neq \lambda \\ \{\} & \text{otherwise} \end{cases}$$

□

Intuitively, for a given s-data $a \in A$ and $\kappa \in \mathcal{U}$, if κ is part of all the s-datums in a, then all the s-datums in the κ-extended a contain (exactly) one κ s-value. For any $a \in A$, none of the s-datums in the κ-reduced a contain a κ s-value, thus the κ-reduction and κ-extension of a are disjoint. This property is formalised in Proposition 4.3.6.

Proposition 4.3.6. Let $\mathcal{A}$ be a DDV, with $\mathcal{U}$ its universe and A its carrier set. For any $\kappa \in \mathcal{U}$ and $a \in A$ the following holds: $(\forall dt \in a.\ \kappa \overline{\in} dt) \implies a\Big\Downarrow_{\kappa} \cap a\Big\Uparrow^{\kappa} = \{\}$. □

The operator $\downarrow$ distributes over both union and intersection. The operators $\Downarrow$ and $\Uparrow$ distribute over union; however, they do not distribute over intersection. These properties are formalised in Proposition 4.3.7.

Proposition 4.3.7. Let $\mathcal{A}$ be a DDV, with $\mathcal{U}$ its universe, SD its set of s-datums, and A its carrier set. For any $\kappa \in \mathcal{U}$, $dt_1, dt_2 \in SD$, and $a, b \in A$, the following properties hold:

1. $(dt_1 \cup dt_2)\big\downarrow_\kappa = dt_1\big\downarrow_\kappa \cup dt_2\big\downarrow_\kappa$
2. $(dt_1 \cap dt_2)\big\downarrow_\kappa = dt_1\big\downarrow_\kappa \cap dt_2\big\downarrow_\kappa$
3. $(a \cup b)\Big\Downarrow_\kappa = a\Big\Downarrow_\kappa \cup b\Big\Downarrow_\kappa$
4. $(a \cap b)\Big\Downarrow_\kappa \subseteq a\Big\Downarrow_\kappa \cap b\Big\Downarrow_\kappa$
5. $(a \cup b)\Big\Uparrow^\kappa = a\Big\Uparrow^\kappa \cup b\Big\Uparrow^\kappa$
6. $(a \cap b)\Big\Uparrow^\kappa \subseteq a\Big\Uparrow^\kappa \cap b\Big\Uparrow^\kappa$ □

Intuitively, for a fixed sort $\kappa \in \mathcal{U}$, repeatedly applying the reduction or extension operator with respect to κ on any s-datum (or any s-data) yields the same result as applying it once. In this sense, both operators are idempotent with respect to κ: once all κ s-values have been removed (in the case of reduction) or added (in the case of extension), further applications of the same operator w.r.t. the same sort have no additional effect. This property is formalised in Proposition 4.3.8.

Proposition 4.3.8. Let $\mathcal{A}$ be a DDV, with $\mathcal{U}$ its universe, SD its set of s-datums, and A its carrier set. For any $\kappa \in \mathcal{U}$, $dt \in SD$, and $a \in A$, the data operators are idempotent w.r.t. to κ, i.e.,:

1. $(dt\big\downarrow_\kappa)\big\downarrow_\kappa = dt\big\downarrow_\kappa$
2. $(dt\big\uparrow^\kappa)\Big\Uparrow^\kappa = dt\big\uparrow^\kappa$
3. $(a\Big\Downarrow_\kappa)\Big\Downarrow_\kappa = a\Big\Downarrow_\kappa$
4. $(a\Big\Uparrow^\kappa)\Big\Uparrow^\kappa = a\Big\Uparrow^\kappa$ □

Similarly, the reduction and extension operators are commutative with respect to sorts: for any two sorts $\kappa, \lambda \in \mathcal{U}$, applying κ-reduction (or extension), followed by λ-reduction (or extension) yields the same result as applying them in the opposite order. This reflects the fact that removing or adding values of different sorts does not interfere with one another, so the order of operations is irrelevant. These properties are formalised in Proposition 4.3.8.

Proposition 4.3.9. Let $\mathcal{A}$ be a DDV, with $\mathcal{U}$ its universe, SD its set of s-datums, and A its carrier set. For any $\kappa, \lambda \in \mathcal{U}$, $dt \in SD$, and $a \in A$, the following equations hold:

1. $(dt\big\downarrow_\kappa)\big\downarrow_\lambda = (dt\big\downarrow_\lambda)\big\downarrow_\kappa$
2. $(a\Big\Downarrow_\kappa)\Big\Downarrow_\lambda = (a\Big\Downarrow_\lambda)\Big\Downarrow_\kappa$

3. $(a\Uparrow^{\kappa})\Uparrow^{\lambda} = (a\Uparrow^{\lambda})\Uparrow^{\kappa}$ □

With these notations, we observe that, for a given $\kappa \in \mathcal{U}$, any s-data $a \in A$ has two disjoint components. The first component contains s-datums that do not include any κ s-values, and it is called κ-*floored* a. The second component contains s-datums that include a κ s-value, and it is called κ-*raised* a. We denote these two components as follows, for any $\kappa \in \mathcal{U}$ and $a \in A$:

$$\lfloor a \rfloor_{\kappa} = \{dt \mid dt \in a \wedge \neg\exists v \in \kappa.\ (\kappa, v) \in dt\} \tag{4.9}$$

$$\lceil a \rceil^{\kappa} = \{dt \mid dt \in a \wedge \exists v \in \kappa.\ (\kappa, v) \in dt\} \tag{4.10}$$

Together, these subsets define a partition of a w.r.t. κ, as stated in the following proposition:

Proposition 4.3.10. Let $\mathcal{A}$ be a DDV, with $\mathcal{U}$ its universe and A its carrier set. For any $\kappa \in \mathcal{U}$ and $a \in A$, the following properties hold:

1. $\lfloor a \rfloor_{\kappa} \cup \lceil a \rceil^{\kappa} = a$ (completeness: each s-datum in a is either κ-floored or κ-raised)
2. $\lceil a \rceil^{\kappa} \cap \lfloor a \rfloor_{\kappa} = \{\}$ (disjointness: no s-datum in a belongs to both components)) □

For any sort $\kappa \in \mathcal{U}$ and s-data $a \in A$, the κ-reduced a and κ-floored a s-datas consist of s-datums that do not contain κ s-values. In other words, any $dt \in a\Downarrow_{\kappa}$ or $dt \in \lfloor a \rfloor_{\kappa}$, satisfies the condition $\lfloor \kappa \overline{\in} dt$. In contrast, the κ-assigned component of a includes only those s-datums that contain one κ s-value, i.e., for any $dt \in \lceil a \rceil^{\kappa}$, $\kappa \overline{\in} dt$. The κ-extended of a s-data differs in that it may include a mix of s-datums: some that contain a κ s-value, and others that contain no κ s-value. Proposition 4.3.11 formalises the interaction between the κ-floored and κ-raised components of an s-data a and the results of applying the κ-extension and κ-reduction operators to a. Together, these properties show how the raised and floored components behave under κ-based modifications and how they relate to the overall structure of a. Specifically, Equations 1 and 2 show that both the κ-floored and κ-raised components are preserved under intersection with the κ-extended a. Equation 3 reflects that κ-extension is idempotent w.r.t. membership, i.e., intersecting a with its κ-extension yields the original s-data. Equation 4 highlights the disjointness of the κ-raised and κ-reduced s-datas, i.e., no s-datum can be simultaneously in κ-raised a and κ-reduced a. Equations 5 and 6 illustrate that a κ-reduced s-data contains exactly the κ-floored component of the original s-data, showing a structural equivalence between the effect of κ-reduction and the notion of κ-flooring. Together, these properties establish that κ-flooring and κ-raising are not just syntactic components of an s-data but are deeply aligned with the algebraic behaviour of the reduction and extension operators. They form a partitioning structure that is stable under κ-modification, which will later support reasoning about data transformations.

Proposition 4.3.11. Let $\mathcal{A}$ be a DDV, with $\mathcal{U}$ its universe and A its carrier set. For any $\kappa \in \mathcal{U}$ and $a \in A$, the following equations hold:

1. $\lfloor a \rfloor_\kappa \cap a\Big\Uparrow^\kappa = \lfloor a \rfloor_\kappa$
2. $\lceil a \rceil^\kappa \cap a\Big\Uparrow^\kappa = \lceil a \rceil^\kappa$
3. $a \cap a\Big\Uparrow^\kappa = a$
4. $\lceil a \rceil^\kappa \cap a\Big\Downarrow_\kappa = \{\}$
5. $\lfloor a \rfloor_\kappa \cap a\Big\Downarrow_\kappa = \lfloor a \rfloor_\kappa$
6. $a \cap a\Big\Downarrow_\kappa = \lfloor a \rfloor_\kappa$ □

The disjointness properties established previously for a single s-data extend naturally to any pair of s-datas, $a, b \in A$. Intuitively, the κ-raised component of a is disjoint from both the κ-floored b and κ-reduced b, for any $\kappa \in \mathcal{U}$. This reflects the structural separation enforced by κ-based decomposition: if an s-datum belongs to the κ-raised component of one s-data, it cannot simultaneously belong to the κ-flooring or κ-reduction of another. However, the interaction between the κ-floored components of one s-data the κ-extension of another is more nuanced. These properties are formalised in Proposition 4.3.12.

Proposition 4.3.12. Let $\mathcal{A}$ be a DDV, with $\mathcal{U}$ its universe and A its carrier set. For any $\kappa \in \mathcal{U}$ and $a, b \in A$, the following equations hold:

1. $\lceil a \rceil^\kappa \cap \lfloor b \rfloor_\kappa = \{\}$
2. $\lceil a \rceil^\kappa \cap b\Big\Downarrow_\kappa = \{\}$
3. $\lfloor a \rfloor_\kappa \cap b\Big\Uparrow^\kappa = \lfloor a \rfloor_\kappa \cap \lfloor b \rfloor_\kappa$ □

4.3.4 Operators Interpretation

The operators of the structure $\mathcal{M}_\mathcal{A}$ are defined using set operators. The cylindrification operator on an s-data is defined as the extension of the s-data. For any $\kappa \in \mathcal{U}$

and $a, b \in A$:

$$a + b \stackrel{\text{def}}{=} a \cup b \tag{4.11}$$

$$a \star b \stackrel{\text{def}}{=} a \cap b \tag{4.12}$$

$$\mathbf{0} \stackrel{\text{def}}{=} \{\} \tag{4.13}$$

$$\mathbf{1} \stackrel{\text{def}}{=} \bowtie \mathcal{U} \tag{4.14}$$

$$-a \stackrel{\text{def}}{=} \mathbf{1} \backslash a \tag{4.15}$$

$$\mathbf{c}_\kappa a \stackrel{\text{def}}{=} a \Uparrow^\kappa \tag{4.16}$$

The cylindrification operator on an s-data over the sort κ extends the s-value corresponding to the sort κ with all the values in the sort. Thus, application of the cylindrification operator is evaluated differently in the two approaches. Taking the s-datas a, b, from the example in Figure 4.2, we show the result of applying the cylindrification operator on them over the sort S_g. In the evidence-based approach,

$$\mathbf{c}^{\mathcal{E}}_{S_g} a = \big\{\{(S_t, \mathit{Mean\,Girls}), (S_g, \mathit{Comedy})\}, \{(S_t, \mathit{Mean\,Girls}), (S_g, \mathit{Adventure})\}\big\}$$
$$\mathbf{c}^{\mathcal{E}}_{S_g} b = \big\{\{(S_t, \mathit{Game\,of\,Thrones})\}\big\}$$

In contrast, using the expert-based approach, the application of the cylindrification operator on a and b evaluates to $\mathbf{c}^{\mathcal{X}}_{S_g} a = \mathbf{c}^{\mathcal{E}}_{S_g} a \cup \big\{\{(S_t, \mathit{Mean\,Girls}), (S_g, \mathit{Fantasy})\}\big\}$ and $\mathbf{c}^{\mathcal{X}}_{S_g} b = \mathbf{c}^{\mathcal{E}}_{S_g} b$, respectively.

In either approach, since a contains S_g s-value(s), its cylindrification on S_g extends the s-data only in content, by adding s-datum(s) with the same structure (as defined in Section 4.3.1) as that of a. In the evidence-based approach, a is extended with the $\{(S_t, \mathit{Mean\,Girls}), (S_g, \mathit{Adventure})\}$ s-datum, and in the expert-based approach, a is further extended with the $\{(S_t, \mathit{Mean\,Girls}), (S_g, \mathit{Fantasy})\}$ s-datum. In contrast, in either approach as b contains no S_g s-value(s), its cylindrification does not change its content or structure. Throughout the remainder of this work, all the examples use the evidence-based approach, as the difference between the evidence-based and expert-based approaches is the way the content of the sorts are provided. For simplicity, we assume that the data (the DIS is based on) is clean and that the content of the sorts originates from the data (i.e., the values of each attribute in the dataset).

Within this interpretation of a DDV, we can show a few properties of the cylindrification operator, as follows:

Proposition 4.3.13. Let $\mathcal{A}$ be a DDV, with $\mathcal{U}$ its universe and A its carrier set. The

cylindrification operator distributes over the operator, i.e., for any $a, b \in A$ and any $\kappa \in \mathcal{U}$, $\mathbf{c}_\kappa(a + b) = \mathbf{c}_\kappa a + \mathbf{c}_\kappa b$. □

Proposition 4.3.14. Let $\mathcal{A}$ be a DDV, with $\mathcal{U}$ its universe and A its carrier set. The cylindrification operator is idempotent w.r.t. to κ, i.e., for any $a \in A$ and any $\kappa \in \mathcal{U}$, $\mathbf{c}_\kappa(\mathbf{c}_\kappa a) = \mathbf{c}_\kappa a$. □

In Appendix A, Section A.2, we prove that the $\mathcal{M}_\mathcal{A}$ structure satisfies the axioms of a diagonal-free cylindric algebra, by verifying each condition from Definition 3.2.4. Specifically, we show that the two binary operators ($+$ and $\star$) are commutative, associative, and distribute over each other. We show that the axioms for identity and annihilation w.r.t. $\mathbf{0}$ and $\mathbf{1}$, as well as the complement axioms are satisfied, thus establishing that $\mathcal{M}_\mathcal{A}$ is a Boolean algebra (i.e., the first condition of a cylindric algebra). In addition, we prove that the four axioms for a diagonal-free cylindric algebra axioms are satisfied. Therefore, the structure $\mathcal{M}_\mathcal{A}$ qualifies as a model of diagonal-free cylindric algebra.

4.4 DIS Model: Domain Ontology Component

As discussed in Section 3.3.1, at the DOnt level, we are not interested in the objects of the domain (i.e., the data), we are only interested in grouping them into categories. A basic category is called *atom* and a set of atoms is called *concept*. A singleton concept (containing exactly one atom) is called an *atomic concept*. If it is clear from the context, an atomic concept may be called *atom* as well. We denote atoms by at, at_1, etc. With this understanding, the operators of the monoid of concepts and the Boolean lattice are interpreted using the set operators.

In general, given a set of concepts C and a concept $k \in C$, where $k = \{at_1, at_2, \ldots, at_n\}$, concept k represents objects that share the same atomic structure . Here, the atomic structure of a concept refers to the set of atomic attributes (atoms) that characterise it. Each atom corresponds to a minimal, indivisible property or attribute within the domain of discourse. The composition operator $\oplus$ allows the formation of new concepts by composing existing concepts, i.e., by combining the atomic structures of concepts. When we write $k \stackrel{\text{def}}{=} k_1 \oplus k_2$ we mean that "Concept k is defined as the Cartesian construction of the atomic structures of concepts k_1 and k_2" (i.e., the set of atoms in k is the union of the atoms in k_1 and k_2). Consequently, a concept k_1 is considered a `partOf` another concept k, denoted by $k_1 \sqsubseteq_C k$, if the atomic structure of k_1 is a Cartesian projection of that of k (i.e., if the set of atoms defining k_1 is a subset of the atoms defining concept k). The empty concept $\mathbf{e}_C$ is understood as a concept with no atoms.

We define the composition operator $\oplus$ and the empty concept $\mathbf{e}_C$ as follows, for any $k_1, k_2 \in C$:

$$k_1 \oplus k_2 \stackrel{\text{def}}{=} k_1 \cup k_2 \tag{4.17}$$

$$\mathbf{e}_C \stackrel{\text{def}}{=} \{\} \tag{4.18}$$

The `partOf` relation is defined in the classic way, using the composition operator, for any $k_1, k_2 \in C$:

$$k_1 \sqsubseteq_C k_2 \stackrel{\text{def}}{=} k_1 \oplus k_2 = k_2 \tag{4.19}$$

Take the example in Figure 4.3, discussed in more details below. In it, concept *Listing* is defined as the composition of atomic concepts *titleType* and *genre*. This means that from a structural point of view, an individual of the concept *Listing* contains information for both *titleType* and *genre*. It is immediate that *titleType* is `partOf` *Listing*, i.e., $titleType \sqsubseteq_C Listing$.

The set of concepts $C_{At_\mathcal{C}}$ is freely generated by these two operators (composition and empty concept), over a finite set of atoms $At_\mathcal{C}$, and it can be shown that $C_{At_\mathcal{C}} = \mathcal{P}(At_\mathcal{C})$. By definition, the composition operator is commutative, associative, and idempotent, and the empty concept is its neutral element. Thus, $\mathcal{M}_\mathcal{C} = \left(C_{At_\mathcal{C}}, \oplus, \mathbf{e}_C\right)$ is a commutative idempotent monoid.

For the next component of the DOnt, the Boolean lattice, we start from a finite set of atoms $At_\mathcal{L}$, s.t. there is a one-to-one correspondence between the sorts of the universe in the DDV model and the atoms of $At_\mathcal{L}$. We ensure that $At_\mathcal{L} \subseteq At_\mathcal{C}$. As discussed in Section 3.2.4, a Boolean lattice is isomorphic to a Boolean algebra (Sankappanavar and Burris, 1981), thus we use the algebraic definition of the lattice $\mathcal{L}$. Let $\mathcal{M}_\mathcal{L} = \left(L_{At_\mathcal{L}}, \oplus, \otimes, \ominus, \mathbf{e}_C, \top_\mathcal{L}\right)$.

For ease of readability, we use the following naming convention: the atoms in $At_\mathcal{L}$ are denoted by κ_{at}, and there is a corresponding atomic concept $\kappa \in L, \kappa = \{\kappa_{at}\}$. The carrier set of the Boolean lattice is defined by $L_{At_\mathcal{L}} \stackrel{\text{def}}{=} \mathcal{P}(At_\mathcal{L})$, and is referred as simply by L when it is clear from the context what we refer to. By construction, the carrier set L contains the empty concept $\mathbf{e}_C$. The composition operator of the Boolean lattice is a restriction of the composition operator of the monoid of concepts, $\oplus\colon L \times L \to L$, from Definition 4.17. The other three Boolean lattice operators are

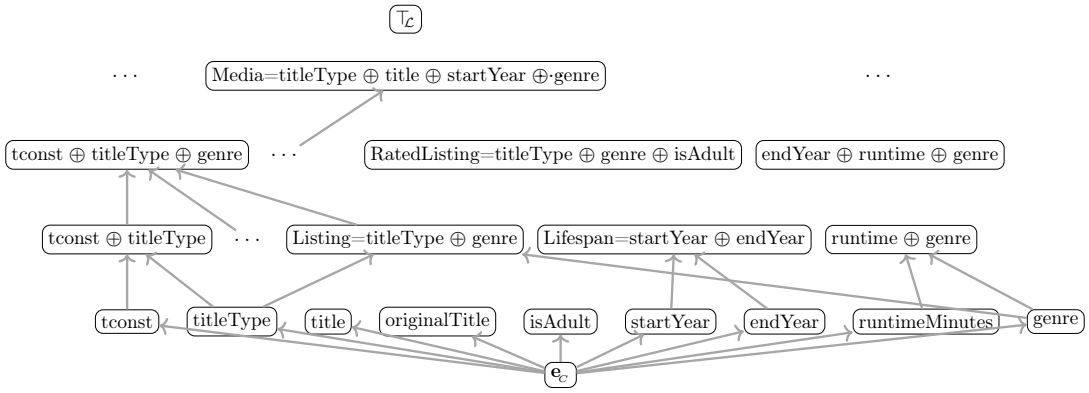

Figure 4.3: Media DIS Boolean Lattice (a partial view)

defined using set operators, for any $\kappa, \kappa_1, \kappa_2 \in L$:

$$\kappa_1 \otimes \kappa_2 \stackrel{\text{def}}{=} \kappa_1 \cap \kappa_2 \tag{4.20}$$

$$\top_{\mathcal{L}} \stackrel{\text{def}}{=} At_{\mathcal{L}} \tag{4.21}$$

$$\ominus\kappa \stackrel{\text{def}}{=} \top_{\mathcal{L}} \backslash \kappa \tag{4.22}$$

In Figure 4.3 we present a partial view of the Media DIS Boolean lattice. The atoms of the lattice correspond one-to-one to the sorts of the domain data view. Thus, in our example, the atoms of the lattice are specified by the set $At_{\mathcal{L}_M} = \{tconst_{at}, titleType_{at}, title_{at}, originalTitle_{at}, isAdult_{at}, startYear_{at}, endYear_{at}, runtimeMinutes_{at}, genre_{at}\}\}$. The carrier set of the Boolean lattice is described by $L = \mathcal{P}(\{tconst, titleType, title, originalTitle, isAdult, startYear, endYear, runtimeMinutes, genre\})$.

In Appendix A, Section A.3, we prove that the $\mathcal{M}_{\mathcal{L}}$ structure satisfies the axioms of a Boolean algebra. Specifically, we show that the two binary operators ($\oplus$ and $\otimes$) are commutative, associative, and distribute over each other. We show that the the axioms for identity and annihilation w.r.t. $\mathbf{e}_C$ and $\top_{\mathcal{L}}$ are satisfied, as well as the complement axiom hold. Therefpre, the structure $\mathcal{M}_{\mathcal{L}}$ qualifies as a model of Boolean algebra.

For a better understanding of the domain of application, the domain ontology must be enriched with more than the concepts of its Boolean lattice. To add concepts to the DOnt, the domain expert may (i) name composite concepts in the Boolean lattice or (ii) define other concepts related to the Boolean lattice concepts, i.e., the non-Boolean lattice concepts of the rooted graphs. We further discuss both methods below. In additon, the domain expert may add datascape concepts to the DIS, a

process detailed in Section 4.6.

Using the first method, the domain expert can define composite lattice concepts that are significant for the domain of application. These concepts are explicitly related to the data in the DDV. For instance, in Figure 4.3, there are new named concepts that are defined by composing other lattice concepts, such as:

$$Listing = titleType \oplus genre$$
$$Media = titleType \oplus title \oplus startYear \oplus genre$$

The remaining composite concepts in the Boolean lattice, such as $tconst \oplus titleType$, exist and may be used by the reasoning process. They are not described here with a specific identifier.

Using the second method, the domain expert can capture domain concepts that may not be explicitly described in the DDV (i.e., in the existing data). The domain expert captures these concepts through the process of building rooted graphs, which further integrates other DISs to access additional domain data views and their datascapes. In this section, we refer to *datascape* as abstract representations that link s-datas to conceptual views defined in the domain ontology. A detailed formalisation of datascapes is provided later in Section 4.6. With this informal understanding in place, we can now examine an example that illustrates how concepts from two domains interact, and how datascape concepts support the alignment between structured data and domain knowledge.

In Figure 4.4, the concepts outside the V_M Boolean lattice cannot be defined using the elements of the DDV, as their data resides in the V_R domain. Thus, within the V_M domain, they are captured as atomic concepts. If needed, during the reasoning process, these concepts are instantiated through their respective datascape concepts within the V_R domain.

The graph pictured in Figure 4.4 is formed over the $\texttt{R}_{\texttt{recognition}}$ relation, depicted with a double line. The vertices of this graph (except for the root) are concepts in the V_R domain, and they describe the recognition level of the *Media*, such as *Acclaimed* or *Infamous*. Within the V_R domain, these concepts can be further refined, by defining their corresponding datascape concepts. This method shows similarities to how `isA` relations are captured in current taxonomic ontologies (Di Pinto *et al.*, 2019). Within the V_M domain, the relation between the concept *Media* and its datascape concepts *Acclaimed* and *Infamous* is described in the classic way, as a taxonomic graph. This taxonomic relation $\texttt{isA}_{\texttt{recognition}}$ is depicted with a double dotted line. The relation

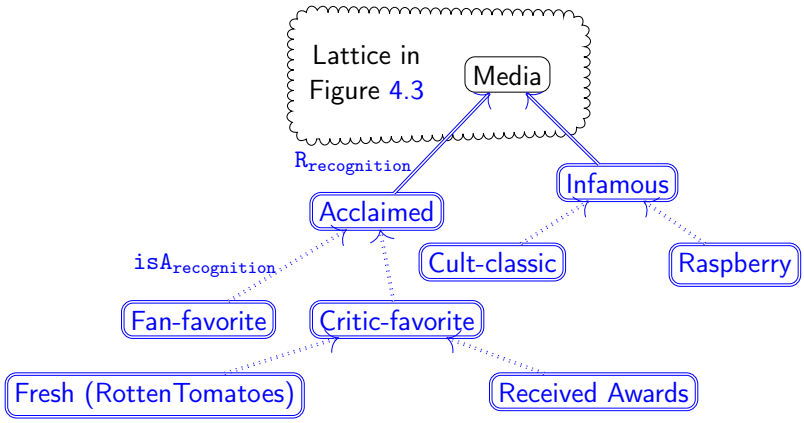

Figure 4.4: Media Rooted Graph for $\mathtt{R}_{\mathtt{recognition}}$ relation

of the rooted graph is understood as $\mathtt{R}'_{\mathtt{recognition}} = \mathtt{R}_{\mathtt{recognition}};\mathtt{isA}_{\mathtt{recognition}}$. Note that while this method is depicted as the taxonomic `isA` relation within the V_M DIS, there is no need for such a rooted graph within the V_R DIS. This is because the `isA` relation is defined through the use of datascape concepts.

Let $\mathcal{M}_{\mathcal{G}} = \{G_{t_i}\}_{t_i \in L, i \in I}$ be the interpretation of the family of rooted graphs, where $G_{t_i} = \big(C_i, R_i, t_i\big), C_i \subseteq C, R_i \subseteq C_i \times C_i, t_i \in L$. By construction, $\mathcal{M}_{\mathcal{G}}$ qualifies as a model for a family of rooted graphs. Thus, the structure $\mathcal{M}_{\mathcal{O}} = \big(\mathcal{M}_{\mathcal{C}}, \mathcal{M}_{\mathcal{L}}, \mathcal{M}_{\mathcal{G}}\big)$ qualifies as a model for the domain ontology $\mathcal{O}$.

4.5 DIS Model: Mapping Operator Component

The link between the DDV and the DOnt is specified through the mapping operator τ, which maps an s-data in the DDV to a concept in the Boolean lattice of the DOnt. The sorts of the universe $\mathcal{U}$ correspond to atoms of the Boolean lattice. We say that the *structure*, or more precisely, the conceptual type, of an s-data refers to the set of domain atomic concepts corresponding to the sorts present in its s-values. We remind the reader that, as described in Sections 4.3 and 4.4, given an attribute 'attr' of the dataset, we denote by S_{attr} , $attr_{at}$, and $attr$ its corresponding sort, atom, and atomic concept, respectively.

Given an s-data $a \in A$, we denote by S_a the set of sorts corresponding to all the s-values in a. Then, $\tau(a)$ is defined by the composition of the atomic concepts corresponding to the elements of S_a. In the following, we detail the mapping process, the formal definition of the mapping operator τ, as well as its properties.

To support the formal definition of the mapping operator, we define a series of helper operators. On s-values, we define two postfix operators, $.sort : SV \to \mathcal{U}$, with $sv.sort = s$, and $.val : SV \to \mathcal{U}$, with $sv.val = v$, for any $sv \in SV, sv = (s, v)$. We define a helper mapping operator $\eta : \mathcal{U} \to L$, $\eta(S_{attr}) = attr$, which does a one-to-one mapping from the sorts of the DDV to the atomic concepts of the Boolean lattice in the DOnt. Finally, the mapping operator $\tau : A \to L$ is defined, for any $a \in A$, as follows:

$$\tau(a) = \{\eta(sv.sort) \mid sv, dt \,.\, sv \in dt \wedge dt \in a\} \tag{4.23}$$

Taking the example in Figure 4.3, with $a \in A$ to be the s-data $a = \Big\{\{(S_{titleType}, \text{`}movie\text{'}), (S_{title}, \text{`}MeanGirls\text{'}), (S_{startYear}, 2004), (S_{genre}, \text{`}Comedy\text{'})\}\Big\}$, then $\tau(a) = titleType \oplus title \oplus startYear \oplus genre$ or $\tau(a) = Media$. For any s-data $a \in A$ and $\kappa \in L$ s.t. $\tau(a) = \kappa$, we say that "a is of type κ".

In the remainder of this section, we present properties of the mapping operator. The detailed proofs for these properties are provided in Appendix A, Section A.4. Two such properties of the mapping operator are related to the s-datum-extension-by-sort and s-datum-reduction-by-sort data operators, $\Uparrow$, and $\Downarrow$, respectively, and are formalised in Proposition 4.5.1.

Proposition 4.5.1. Let $\mathcal{I}$ be a DIS, with $\mathcal{U}$ the universe and A the carrier set of its DDV, and η, τ its mapping operators. For any $\kappa \in \mathcal{U}$ and $a \in A$, the following properties hold:

1. $\exists dt \in a.\ \kappa \overline{\in} dt \implies \eta(\kappa) \sqsubseteq_C \tau(a\Uparrow^{\kappa})$
2. $\eta(\kappa) \not\sqsubseteq_C \tau(a\Downarrow_{\kappa})$ □

The DDV and the DOnt both are built on top of the same core mathematical structure (a Boolean algebra), thus the mapping operator τ acts as a morphism between the two Boolean algebras. While, from a mathematical perspective, it would be convenient that the mapping operator is a homomorphism between the Boolean algebra of the DDV and the one of the DOnt, not all properties hold. Intuitively, when we add a new s-value (of some sort) to an s-data, its structure may be extended, thus the concept corresponding to the added s-value must be added to the structure (or type) of the s-data. Thus, given a DIS $\mathcal{I}$, its τ operator preserves the composition, zero, and one, for any $a, b \in A$, reflected in the mapping operator axioms of the DIS:

$$(A15)\ \tau(\mathbf{0}) = \mathbf{e}_C$$
$$(A16)\ \tau(\mathbf{1}) = \top_{\mathcal{L}}$$
$$(A17)\ \tau(a + b) = \tau(a) \oplus \tau(b)$$

In contrast, the mapping operator does not preserve the complement and the $\star$ operators. In the world of data (i.e., at the DDV level), the complement of an s-data a is obtained by "hiding" (i.e., removing them from the $\mathbf{1}$, Definition 4.14) the s-datums of a. This means that the s-datums in the complement $-a$ may contain s-values of the same sort as the ones in a. Thus the sorts found in the complement of a may still be a part of the complement. On the other hand, in the world of concepts (i.e., at the DOnt level), the complement of a concept k is obtained by "hiding" (i.e., removing them from the $\top_{\mathcal{L}}$) of all the atomic concepts of k. The complement concept $\ominus k$ contains only atomic concepts that are found in k. If $\tau(a) = k$, this may not translate into $\tau(-a) = \ominus k$. The level of granularity is different at the two levels, and the mapping operator does not preserve the complement operator $-$. Since in a Boolean algebra the $\star$ is defined in terms of $+$ and $-$, the mapping operator does not preserve the $\star$ operator either. The properties of the mapping operator are formalised in Proposition 4.5.2.

Proposition 4.5.2. Let $\mathcal{I}$ be a DIS, with $\mathcal{U}$ the universe and A the carrier set of its DDV, and τ its mapping operator. For any $a, b \in A$, and $T \subseteq \mathcal{U}$:

1. $(a = \mathbf{0} \vee a = \mathbf{1}) \implies \tau(-a) = \ominus\tau(a)$
2. $\tau(a) \otimes \tau(b) = \mathbf{e}_{_C} \implies \tau(a \star b) = \mathbf{e}_{_C}$
3. $\tau(a \star b) \subseteq \tau(a) \otimes \tau(b)$
4. $a = \bowtie T_a \implies \big(\tau(a) = \bigcup\{\eta(s) \mid s \,.\, s \in T_a\}\big)$
5. $\big(a = \bowtie T_a \wedge b = \bowtie T_b\big) \implies \big(\tau(a \star b) = \tau(a) \otimes \tau(b)\big)$ □

For a counterexample of the complement and $\star$ operators non-preservation, we refer the reader to Figure 4.2, with $\tau(a) = \mathit{title} \oplus \mathit{genre}$ and $\tau(b) = \mathit{title}$. Then $-a = \mathbf{1} \backslash a$, and includes s-values such as $(S_t, \mathit{A\,New\,Hope})$ or $(S_g, \mathit{Adventure})$. Then $\tau(-a) \subseteq \mathit{title} \oplus \mathit{genre}$. Since $\ominus\tau(a) = \mathbf{e}_{_C}$, it is immediate that $\tau(-a) \neq \ominus\tau(a)$. Similarly, $a \star b = \{\}$, thus $\tau(a \star b) = \mathbf{e}_{_C}$. However, $\tau(a) \otimes \tau(b) = \mathit{title}$, thus $\tau(a \star b) \neq \tau(a) \otimes \tau(b)$.

4.6 Datascape Concepts

In Section 4.4, an informal reference was made to datascape concepts as abstractions that bridge structured data and domain knowledge. In this section, we formally define datascape concepts, and describe their role within the DIS. This formalisation provides the necessary foundation for reasoning tasks that involve the dynamic instantiation of concepts based on data-level information.

The domain expert has so far built the DDV from a given dataset, created the DOnt on top of it, and linked the two through the mapping operator. To further refine the domain of application, the domain expert may constrain the Boolean lattice concepts with s-values from the DDV. We call these datascape concepts, and we denote them by $concept^D$. Within the Boolean lattice, we look at a concept $\kappa \in L$ as representing a set of s-datas of the type κ. Its datascape conceptis defined by $\kappa^D = \{a \mid \tau(a) = \kappa\}$, where τ is the mapping operator described in Section 4.5. In DL, a concept has two aspects: its intension (i.e., its formal description), and its intension (i.e., the set of individuals that satisfy this description in a given interpretation). In this context, in DIS, a concept or datascape concepts represent the logical definition (i.e., the intension), and the evaluation of a datascape concept w.r.t. an associated DDV its extension.

For example, in the Media DIS, the *Media* concept may have multiple refinements, such as the $Comedy^D$ or $Slasher^D$ datascape concepts. The former represents a *Media* concept which has "Comedy" as its genre, and the latter represents a movie that is either horror or thriller, and was released between the years 1970 and 1990. We provide the definitions of these datascape concepts as follows:

$$Comedy^D = \{a \mid a, dt \,.\, \tau(a) = Media \wedge dt \in a \wedge dt.genre = \text{“}Comedy''\} \tag{4.24}$$

$$\begin{aligned} Slasher^D = \{a \mid\ & a, dt \,.\, \tau(a) = Media \wedge dt \in a \wedge dt.titleType = \text{“}movie'' \\ & \wedge \big(dt.genre = \text{“}Horror'' \vee\ dt.genre = \text{“}Thriller''\big) \\ & \wedge\ 1970 < dt.startYear < 1990\} \end{aligned} \tag{4.25}$$

In Equations (4.24) and(4.25), the variable a is free, allowing for the datascape concepts $Comedy^D$ and $Slasher^D$ to be bound to different data views. In Section 4.6.1, we show how a can be bound to the s-datums of a specific domain data view. For ease of readability, by $dt.\kappa$ we denote the value of the κ s-value of dt, i.e., $dt.\kappa = \{v \mid (s, v) \in dt \,.\, s = \kappa\}$.

In other knowledge systems, such as DL, the main relation is the taxonomical `isA`. In a DIS, the main relation between concepts is the mereological `partOf`, described by the Boolean lattice of the DOnt. Other relations are defined through the family of rooted graphs. The `isA` relations are defined implicitly using datascape concepts. Thus, for any two datascape concepts, k_1^D and k_2^D, we say that k_1^D `isA` k_2^D if and only if $k_1^D \subseteq k_2^D$. E.g.,, from their definition, it is immediate that $Comedy^D$ `isA` $Media^D$ and $Slasher^D$ `isA` $Media^D$.

When two (or more) DISs are integrated, they may share concepts. Capturing the shared concepts is done in two ways. The first way is by explicitly designating two

concepts to be equivalent. For example, in Figure 4.1, the concepts $tconst \in L_M$ and $imdbID \in L_R$ are equivalent. We chose to depict them as the same concept. In practice, they are depicted as two different concepts, and the domain expert adds the definition of their equivalence, as follows

$$tconst = imdbID$$

The second way is through the use of rooted graphs in the first DIS, by capturing concepts from a second DIS. For the shared *Fresh* concept, we illustrate this method using Figure 4.1. In the $\mathcal{I}_M$ DIS, *Fresh* is a vertex of the rooted graph formed by the $\mathtt{R}_{\mathtt{recognition}}$ relation and root *Media*. In the $\mathcal{I}_R$ DIS, *Fresh* is a refined Boolean lattice concept, described by the following datascape concept:

$$\begin{aligned} Fresh^D = \{a \mid a, dt \,.\, \tau_R(a) = imdbID \oplus RottenTomatoes \\ \wedge\ dt \in a \wedge dt.RottenTomatoes \geqslant 70\} \end{aligned}$$

4.6.1 Instantiating Concepts

The concepts that are part of the lattice or are defined (axiomatically) as a specialisation of a lattice concept have been specified in an abstract manner, independent of their links to the data in the DDV. This approach suffices for parts of the reasoning process. For example, assume the domain expert has specified an axiom that states that "A Horror movie or TV show must have at least a PG13 rating". This axiom, along with the (abstract) Definition (4.25) for the $Slasher^D$ concept contain enough information to infer that "A *Slasher* must have at least a PG13 rating".

However, to answer any Competency Questions (CQ) that requires data, the concept involved must be instantiated using the DDV. For example, for the CQ1 query, the domain expert must instantiate concept $Slasher^D$ within a given dataview $\mathcal{A}_M$. This can be done using two sets of s-datas. In general, a datascape concept is instantiated, as detailed below, using the carrier set A of the dataview:

$$\begin{aligned} Slasher^D(A) = \{a \mid a, dt \,.\, a \in A \wedge \tau(a) = Media \wedge dt \in a \\ \wedge\ dt.titleType = \text{“}movie\text{”} \\ \wedge\ \big(dt.genre = \text{“}Horror\text{”} \vee\ dt.genre = \text{“}Thriller\text{”}\big) \\ \wedge\ 1970 < dt.startYear < 1990\} \end{aligned} \tag{4.26}$$

This method of instantiation, by linking to the carrier set of the dataview, offers the flexibility to answer the 'What could possibly be a *Slasher*, given this data?'. In some cases, the domain expert may need more precision, thus asking 'What are the existing *Slashers*, given this data?'. For the latter, the datascape concept is instantiated by

linking it to the original dataset D, and the answer is provided by $Slasher^D(D)$. The datascape concept $Slasher^D(D)$ is defined using Equation (4.26), where D replaces A.

This approach enables the domain expert to consider an abstract schema of the data view. By plugging in different data views, the DIS framework offers various instantiations of the domain ontology. To replace one data view with another one, the only constraint is that they must have the same set of attributes (i.e., their schema corresponds to identical Boolean lattices).

4.7 Discussion

The Domain Information System (DIS) formalism is well suited for structured, organised datasets, and is less ideal for mapping natural language text data. This limitation stems from the design of the Domain Data View (DDV), which is based on the schema of existing structured datasets organised as sets of tuples. Natural language data, in contrast, lacks such explicit structure and would require significant preprocessing before alignment with DIS components.

In the DIS framework, the data is separated from the domain conceptualisation through the two core components: the DDV and the Domain Ontology (DOnt). Structured data is captured at the level of sorts, s-values, s-datums, and s-datas in the DDV, while the conceptual view of the domain is represented through concepts constructed from atomic elements in the DOnt. Although the integration of data and knowledge is central to DIS, it is important to note that the conceptual view can be used independently: the DOnt component, when taken alone, constitutes a pure ontology grounded in a Boolean lattice structure. In scenarios where data is unavailable or undesirable, the DOnt allows reasoning and conceptual modelling without reliance on structured data.

The link between the DDV and the DOnt is established through a surjective helper operator that maps each sort to an atomic concept. The structure of an s-data, defined as the set of its associated sorts, is interpreted semantically through the mapping operator τ. Specifically, τ maps an s-data to a concept in the DOnt by collecting the atomic concepts corresponding to the sorts associated to an s-data, as determined by the surjective mapping. This process reflects the structure of an s-data, and also incorporates the conceptual relations captured within the DOnt, thus yielding a rich conceptual characterisation for reasoning purposes.

In this chapter, a clear and systematic process for building a DIS is presented. The

process not only enables the representation of data and knowledge, but also may assists domain experts in addressing semantic confusion between concepts with similar names (Chantrapornchai and Choksuchat, 2016). By constructing a distinct Boolean lattice for each domain of application, DIS ensures that concepts are semantically distinct unless explicitly linked via equivalence declarations, a process described in Section 4.4.

DIS shares some foundational goals with approaches such as DL, Ontology-based Data Access (OBDA), and DOGMA. While DL distinguishes between the A-Box and T-Box at the logical level, DIS offers a structural separation between the data and the its domain knowledge, supporting modular construction, partial automation, and systematic evolution. In traditional systems, query translation is costly both in terms of computational complexity, due to potentially exponential query rewritings, and in execution efficiency, as relational databases are not optimised for ontology-driven query structures (Xiao *et al.*, 2018). In contrast, in DIS the construction of the ontology is guided directly by the structured data, minimising the mismatch between the two views and avoiding this limitation.

By using the `partOf` relation to build the Boolean lattice, DIS establishes a direct and automatable correspondence between dataset attributes and atomic concepts. The mapping operator captures this link, enabling the derivation of conceptual interpretations and avoiding the need for retrospective alignment between the structured data and the ontology. In addition, both the DDV and DOnt are specified in compatible algebraic languages, which facilitates their alignment.

When addressing ontology evolution, DIS provides a more localised and systematic approach than traditional monolithic ontologies (Zablith *et al.*, 2015; Xiao *et al.*, 2018). Changes to the data content require only updates to the DDV and mappings, leaving the domain ontology unchanged. This task can be fully automated, which makes it efficient. Changes to the data schema require local modifications to the domain ontology structure. The Boolean lattice can be rebuilt automatically, and the graphs and datascape concepts affected by change can be identified automatically. Changes to the domain of application affect only the rooted graphs and datascape concepts, without altering the underlying Boolean lattice. This layered separation ensures that evolution is localised and manageable.

In addition, DIS supports modularisation and selective information hiding, an important feature for a knowledge system (Borgo and Hitzler, 2018). Through operations such as view traversal, modules can be extracted systematically, and knowledge loss can be quantified algebraically via the kernel of the module(LeClair *et al.*, 2019). This

capability supports scalable reasoning and controlled evolution of knowledge systems, addressing both efficiency and maintainability.

Chapter 5

DIS Specification

"Don't write a theorem prover. Try to use someone else's."

- L. C. Paulson, *Handbook of Logic in Computer Science*

In this chapter, we present the specification of the generic DIS theory in ITP Isabelle/HOL. In Section 5.1 we present an overview for Isabelle/HOL, to familiarise the reader with the proof assistant. In Section 5.2 we describe details of the specification of the DIS theory. In Section 5.3, we demonstrate how the DIS specification can be used to engineer concrete ontologies (i.e., instantiations of the DIS theory).

5.1 Isabelle/HOL Overview and Architecture

As discussed in Section 2.4.3, we have chosen Isabelle (Isabelle, 2025) to specify the theory of DIS. When we say Isabelle is *generic*, we mean that it is not tied or specialised to a particular logic. Isabelle provides a framework where different logical systems are implemented, such as HOL, FOL, Zermelo-Fraenkel set theory (ZF), and more. The most widespread logic currently used in Isabelle is HOL, with the Isabelle/HOL instance offering a version of classical higher-order logic theorem proving, ready for medium and large applications (Isabelle, 2025). Isabelle is *interactive* in the sense that proofs are developed incrementally, with each proof step evolving under user supervision and validation until the overall proof is completed. This is done by invoking various proof rules, a process detailed in Appendix E.4. As a *proof assistant*, Isabelle helps its user to find and maintain proofs. Its library of formal theories and proofs is under active development and constantly growing.

In recent years, Isabelle has advanced from an interactive theorem prover to an integrated programming environment, with a variety of tools that go beyond theorem

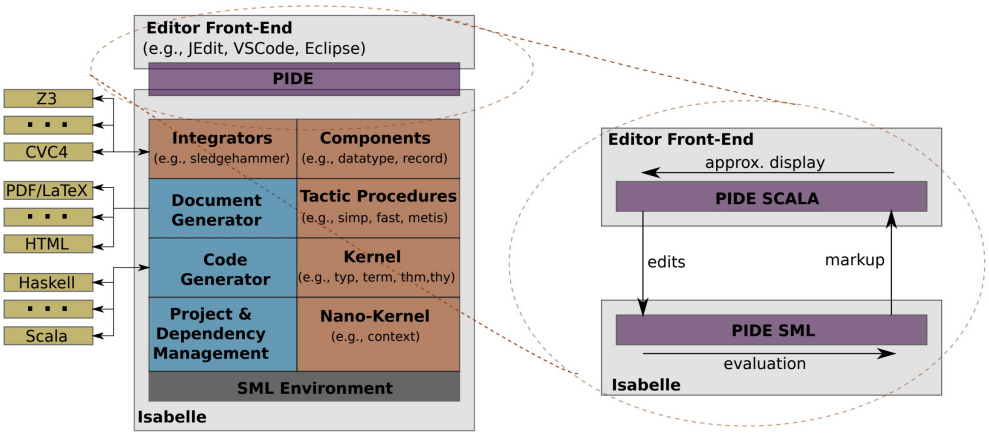

Figure 5.1: Isabelle system architecture (Brucker *et al.*, 2018)

proving. Some of the tools present in Isabelle today give its user the ability to use LCF-style generic theorem prover kernel as programming interfaces, to hierarchically organise theory documents, to incrementally process documents for interactive proof development, to process and prepare high quality documents or generate code directly from the theory modules, etc. (Wenzel and Wolff, 2007; Brucker *et al.*, 2018).

The system architecture of Isabelle is multi-layerd, built on top of the implementation language of Isabelle (i.e., Standard ML). At the top, we find the Prover IDE (PIDE), a rich user interface that offers auto-completion and error-messages while the user is editing theory documents. The system architecture of Isabelle is shown on the left-hand side of Figure 5.1, while the communication between the Isabelle framework and its PIDE is shown on the right-hand side (Brucker *et al.*, 2018). The user can access all the layers of Isabelle.
Isabelle's meta-logic $\mathcal{M}$ is an extension of a fragment of HOL (Nipkow and Roßkopf, 2021). It comes with a *propositions* type and offers three basic operators (or logical connectors): *universal quantifier*, $\bigwedge$, *implication*, $\Longrightarrow$, and *equality*, $\equiv$. The *universal quantifier* enables the notion of a fixed, but arbitrary value, not to be confused with the FOL universal quantifier. The *implication* operator expresses inference rules, and it binds to the right. For example, when we write $A \Longrightarrow B \Longrightarrow C$ (which is also expressed as $[\![A; B]\!] \Longrightarrow C$), it is equivalent to $A \Longrightarrow (B \Longrightarrow C)$, and it means "if A and B hold, then C holds". The *equality* operator expresses that two terms are definition-equivalent and can be substituted for each other. It comes with a number of axioms, such as reflexivity ($x \equiv x$), symmetry ($x \equiv y \Longrightarrow y \equiv x$), and transitivity ($x \equiv y \Longrightarrow y \equiv z \Longrightarrow x \equiv z$). In addition, meta-logic uses *lambda abstractions* λ

Rule	**Term**	**Interpretation**	**NOT**
	$f\ t\ u$	$(f\ t)\ u$	$f\ (t\ u)$
The prefix form of a function binds more strongly than anything	$f\ x + y$	$(f\ x) + y$	$f(x + y)$
Iff is expressed using equality, but equality has a higher priority	$\neg\neg P = P$	$\neg\neg(P = P)$	$(\neg\neg P) = P$
Constructs with opening delimiter and no closing one bind very weakly. E.g., *if*, *let*, *case*, $\lambda, \forall, \exists$.	$\lambda x.\ x = f$	$\lambda x.\ (x = f)$	$(\lambda x.\ x) = f$

Table 5.1: General syntactic rules of Isabelle

to represent functions and apply them to arguments within the logical framework of Isabelle. Note that the meta-level logical connectors handle the structure of proofs and statements about logical deductions, while the object-level ones are part of the logical statements reasoned about in a theory.

For ease of reference, we summarise below basic elements of Isabelle. In Table 5.1 we present basic syntactic rules for Isabelle. In general, the HOL syntax tries to follow the conventions of functional programming and mathematics. In Table 5.2, we outline a list of method proofs commonly used in Isabelle. In Table 5.3, we present the rules of natural deduction commonly used in Isabelle. They are the deduction rules for the logical connectives FOL, together with two basic natural deduction rules for modus ponens and contradiction. Logical connective rules are either *introduction* or *elimination* rules. *Introduction* rules describe how to prove that a particular logical connective or statement is true, e.g., to prove $P \wedge Q$, one has to prove separately P and Q. In contrast, the *elimination* rules describe how to deduce new facts from already established facts, e.g., knowing $P \wedge Q$, one can separately conclude P or Q, by eliminating the conjunction. In Table 5.4, we list the abbreviations most frequently used in Isabelle proofs. For more details on the syntax, natural deduction rules, and proof style of Isabelle, we refer the reader to Appendix E. For the remainder of this chapter, we assume the reader is familiar with Isabelle.

Method	**Description and Usage**
`auto`	Combines classical reasoning with simplification, and intended for proving theorems that have a lot of trivial subgoals. It proves easy subgoals and leaves the ones it cannot prove.
`best`	Legacy Isabelle method, uses best-first search, guided by a heuristic function
`blast`	A generic tableau prover, used for FOL
`fast`	Legacy Isabelle methods, uses depth-first search
`force`	Intended to completely prove the first goal. Performs an exhaustive search, thus proof attempts may take longer or diverge easily.
`rule`	A backward reasoning method that unifies the conclusion of the rule with the current subgoal
`simp`	Simplification, also known as *term rewriting*; invokes the Simplifier on the first subgoal,
`slow`	Similar to `fast`; using a broader search
`bestsimp` `fastforce` `slowsimp`	Similar to `best`, `fast`, `slow`, respectively; using the Simplifier as additional wrapper
`meson`	Implements Loveland's model elimination procedure (Loveland, 2016)
`metis`	A resolution prover

Table 5.2: Automated Proof Tactics in Isabelle

Rule	Name	Safe	Isabelle Representation
$\frac{P \quad Q}{P \wedge Q}$	conjI	Y	$[\![?P;\ ?Q]\!] \Longrightarrow ?P \wedge ?Q$
$\frac{P \wedge Q \quad [\![P;\ Q]\!] \Longrightarrow R}{R}$	conjE	Y	$[\![?P \wedge ?Q; [\![?P; ?Q]\!] \Longrightarrow R]\!] \Longrightarrow ?R$
$\frac{P}{P \vee Q} \quad \frac{Q}{P \vee Q}$	disjI1/2	N	$[\![?P]\!] \Longrightarrow ?P \vee ?Q \quad [\![?Q]\!] \Longrightarrow ?P \vee ?Q$
$\frac{P \vee Q \quad P \Longrightarrow R \quad Q \Longrightarrow R}{R}$	disjE	Y	$[\![?P \vee ?Q;\ ?P \Longrightarrow ?R;\ ?Q \Longrightarrow ?R]\!] \Longrightarrow ?R$
$\frac{P \Longrightarrow Q}{P \longrightarrow Q}$	impI	Y	$[\![?P \Longrightarrow ?Q]\!] \Longrightarrow ?P \longrightarrow ?Q$
$\frac{P \longrightarrow Q \quad P \quad Q \Longrightarrow R}{R}$	impE	N	$[\![?P \longrightarrow ?Q;\ ?P;\ ?Q \Longrightarrow ?R]\!] \Longrightarrow ?R$
$\frac{P \Longrightarrow False}{\neg P}$	notI	Y	$[\![?P \Longrightarrow False]\!] \Longrightarrow \neg ?P$
$\frac{\neg P \quad P}{Q}$	notE	N	$[\![\neg ?P;\ ?P]\!] \Longrightarrow ?Q$
$\frac{P\ a}{\forall x.\ P\ x}$	allI	Y	$[\![\bigwedge a.\ ?P\ a]\!] \Longrightarrow \forall x.\ ?P\ x$
$\frac{\forall x.\ P\ x \quad P[t/x] \Longrightarrow R}{R}$	allE	N	$[\![\forall x.\ ?P\ x;\ ?P\ ?x \Longrightarrow ?R]\!] \Longrightarrow ?R$
$\frac{P\ a}{\exists x.\ P\ x}$	exI	N	$?P\ ?a \Longrightarrow \exists x.\ ?P\ x$
$\frac{\exists x.\ P\ x \quad \begin{array}{c}[P] \\ \vdots \\ Q\end{array}}{Q}$	exE	Y	$[\![\exists x.\ ?P\ x;\ \bigwedge x.\ ?P\ x \Longrightarrow ?Q]\!] \Longrightarrow ?Q$
$\frac{P \Longrightarrow Q \quad Q \Longrightarrow P}{P = Q}$	iffI	Y	$[\![?P \Longrightarrow ?Q;\ ?Q \Longrightarrow ?P]\!] \Longrightarrow ?P = ?Q$
$\frac{P = Q \quad [\![P \longrightarrow Q; Q \longrightarrow P]\!] \Longrightarrow R}{R}$	iffE	Y	$[\![?P = ?Q; [\![?P \longrightarrow ?Q; ?Q \longrightarrow ?P]\!] \Longrightarrow R]\!] \Longrightarrow R$
$\frac{P \longrightarrow Q \quad P}{Q}$	mp	Y	$[\![?P \longrightarrow ?Q;\ ?P]\!] \Longrightarrow ?Q$
$\frac{\neg P \Longrightarrow False}{P}$	ccontr	Y	$[\![\neg ?P \Longrightarrow False]\!] \Longrightarrow ?P$

Table 5.3: Natural deduction rules in Isabelle

Abbreviation	**Stands for**
proof	**proof** (`rule` `elim−rules` `intro−rules`)
proof-	Suppress all rule matching
this	Previous proposition
then	**from this**
thus	**then show** **from this show**
hence	**then have**
with *facts*	**from** *facts* **this**
.	**by assumption**
..	**by** (`rule` `elim−rules` `intro−rules`)
...	Schematic term variable, refers to right hand side of last expression
have $\mathtt{P}_1 \ldots$ **moreover have** $\mathtt{P}_2 \ldots$ $\vdots$ **moreover have** $\mathtt{P}_n \ldots$ **ultimately show**...	**have** $\mathtt{p}_1$: $\mathtt{P}_1 \ldots$ **have** $\mathtt{p}_2$: $\mathtt{P}_2 \ldots$ $\vdots$ **have** $\mathtt{p}_n$: $\mathtt{P}_n \ldots$ **from** $\mathtt{p}_1 \ldots \mathtt{p}_n$ **show**...
?**thesis**	Current goal (the enclosing **show** or **have** statement)

Table 5.4: Isar proof abbreviations in Isabelle

5.2 DIS Specification in Isabelle

In this section, we give a quick overview of the DIS specification in Isabelle/HOL. An overview of the overall design is presented in Figure 5.2. For the remainder of this Section, we use `truetype` font to describe the names of Isabelle elements, such as locales, locale parameters, and theory modules.

As discussed in Chapter 4, a DIS has three components: the DDV, the DOnt, and the mapping operator τ, connecting the two. The DIS theory is defined within the `dis` locale, inside the `DomainInfSys` theory module. It instantiates its two components, the `domain_ontology` and `domain_data_view` locales.

The `domain_data_view` locale is defined within the `DomainDtVw` theory module. It contains the cylindrification operator definition, and the proofs that it is a model of the diagonal-free cylindric algebra (which is found in the `DFCylindricAlgebra` module). To keep the implementation modular, and following the separation of concerns engineering principle, the `domain_data_view` extends the `ddv_ba` locale, which contains the foundational Boolean algebra elements. The `ddv_ba` locale is a model of the Isabelle/HOL core theory `abstract_boolean_algebra`, which is defined within the `HOL.Boolean_Algebra` module. The `ddv_ba` locale extends the `ddv_base` locale, in which the data operators used to specify the cylindrification operator are defined. In order to do that, the `ddv_base` locale extends the `universe` locale, which is defined within the `DDVUniverse` module. The helper operator for the data operators are defined within the `universe` locale. Finally, the foundational elements of the DDV (i.e., s-value, s-datum, s-data, and sort) are all defined within the `DDVTypes` module, which is used by the `DDVUniverse` module. Note that within the Isabelle specification, the s-value, s-datum, s-data, and sort are defined as abstract types, and are denoted by `svalue`, `sdatum`, `sdata`, and `sort`, respectively.

The `domain_ontology` locale is defined within the `DomainOnt` theory module. It instantiates the three core components: the monoid of concepts `concept_monoid` (of the `ConceptMonoid` theory module), the lattice of concepts `concept_lattice` (of the `ConceptLattice` theory module), and the family of rooted graphs `family_root_graphs` (of the `ConceptRootGraph` theory module). All three modules use the abstract type `concept`, declared within the `Concept` module. The `concept_monoid` is a model for `HOL.Group.comm_monoid`. The `concept_lattice` is a dual model for `HOL.Group.comm_monoid`, once over the $\oplus$ operator, and once over the $\otimes$ operator, as well as a model for `abstract_boolean_algebra`. The `family_root_graphs` is defined as a set of `concept_root_graph`. Both locales reside within the `ConceptRootGraph` theory module. The `concept_root_graph` extends

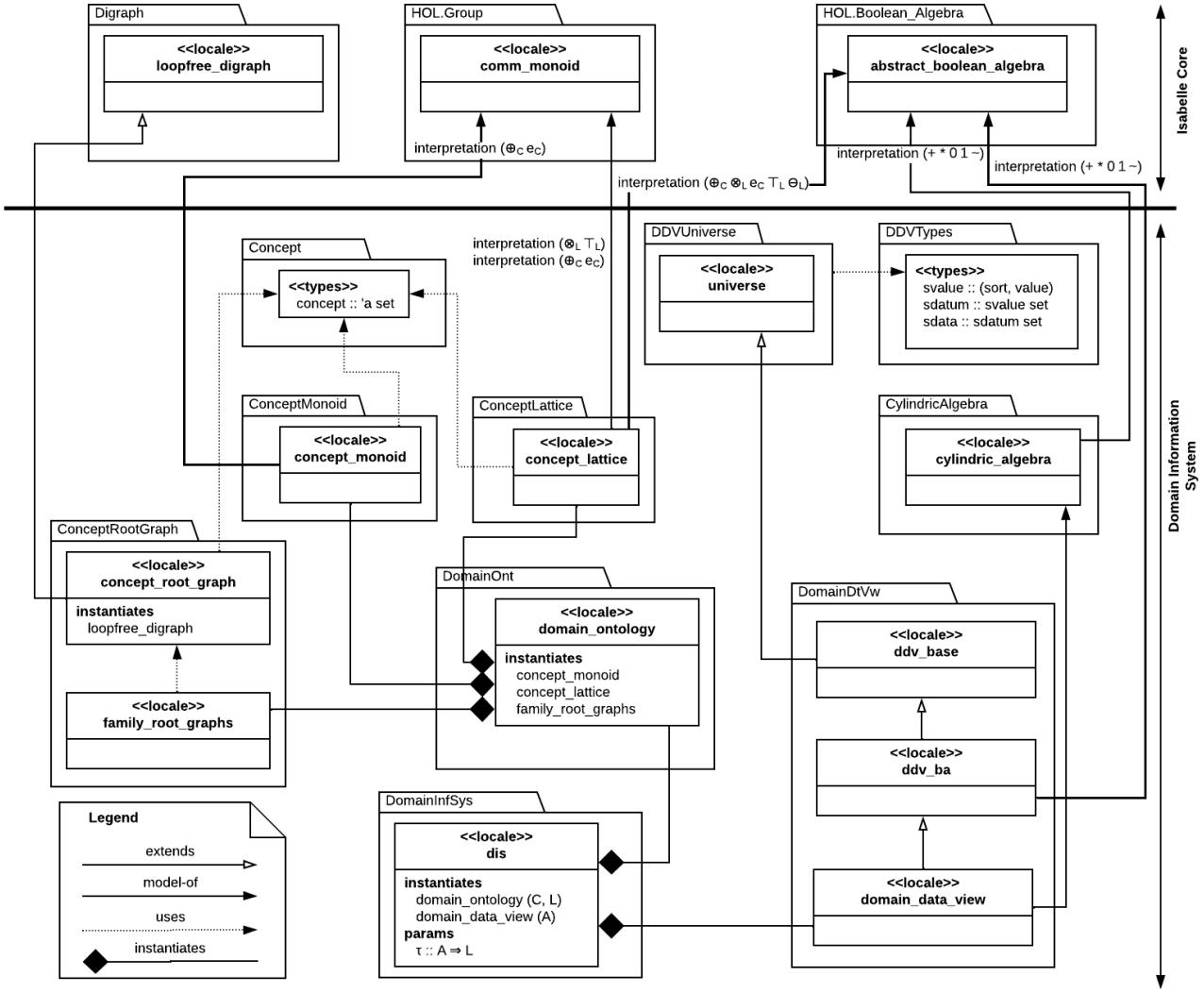

Figure 5.2: DIS Specification in Isabelle: Design Overview

the theory of acyclic directed graphs, defined in `Digraph.loopfree_digraph`, which is part of the extensive library of mathematical theories defined in Isabelle/HOL.

In this way, the DIS specification is built on top of existing classic mathematical structures, such as monoid, Boolean algebra, graph. This enables the end user to employ well-known proven results while reasoning on a concrete DIS. The specification of the DIS theory is given in Appendix B.

5.3 DIS Example: Wine Ontology

In this section we take an example from the wine domain of application to illustrate how the domain expert may specify a concrete DIS that is an instantiation of the `dis` theory. For a full illustration of the Wine DIS example, we direct the reader to Chapter 7, Section 7.2.

In Figure 5.3, we present the design overview of the Wine DIS specification. At the top, we illustrate the DIS components that are immutable. The domain expert only modifies the four components illustrated at the bottom. Within the Wine Universe component, the domain expert defines the `sort`, `svalue`, `sdatum`, `sdata`, and `concept` types. With this information, the universe $\mathcal{U}$ is then defined. Within the Wine DDV component, the domain expert defines the generator set, used to freely generate the carrier set of the Cylindric algebra. Within the Wine DOnt component, the domain expert defines the atoms of the lattice of concepts and the monoid of concepts, along with the elements of the rooted graphs (i.e., their vertices and edges).

Name	**Colour**	**Sugar**	**Body**
Merlot	Red	Dry	Full
Chardonnay	White	Dry	Medium
Vidal	Rose	Sweet	Fruity
Magliocco	Red	Dry	Full
...	...	...	...

Table 5.5: Wine dataset

The construction of the DIS is guided by the sample dataset in Table 5.5. Excerpts of the Isabelle code for the definition of sorts, universe, and the instantiation of the Wine Universe are presented below. In addition, the domain expert may create helper methods to be used for ease of reading. In the Wine DIS example, we decided to define

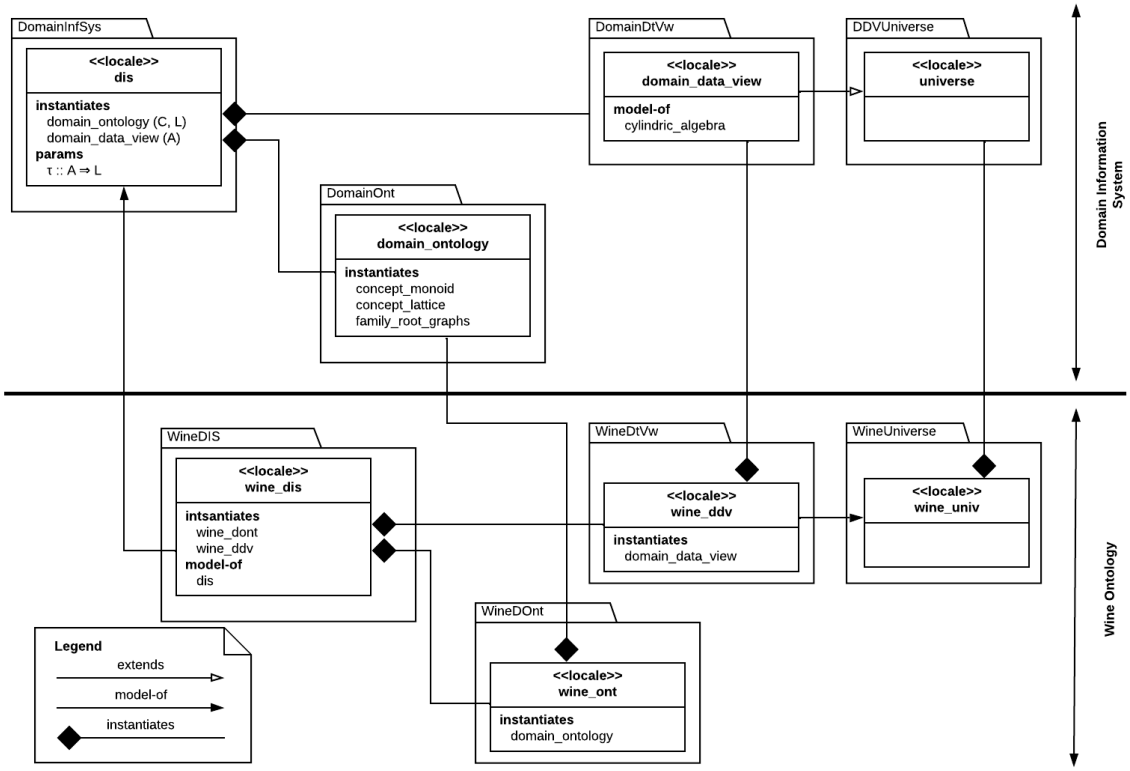

Figure 5.3: Wine DIS Specification: Design Overview

constants for some of the s-values in Wine DIS. During the construction of the other DIS modules and throughout the reasoning process, they may be used as syntactic sugar. An excerpt of these definitions is inserted in Listing 5.1.

```
definition S_W :: sort where
 "S_W = {Str ''Merlot'', Str ''Chardonay'', Str ''Vidal Blanc'', Str ''
    Magliocco''}"
 ...
 definition 𝒰 :: "sort set" where
 "𝒰 = {S_W, S_C, S_S, S_B}"
 ...
locale wine_univ =
  wine_univ: universe where 𝒰 = 𝒰 and sort2name = ws2n and name2sort =
     wn2s
for 𝒰 :: "sort set"
 and ws2n :: "sort ⇒ string"
 and wn2s :: "string ⇒ sort" +
assumes univ_def: "𝒰 = WineUniv.𝒰"
    and s2n_def: "ws2n s = s2n s"
    and n2s_def: "wn2s n = n2s n"
begin
 definition merlot :: svalue where "merlot = ⟨⟨S_W, Str ''Merlot''⟩⟩"
 ...
 definition red :: svalue where "red = ⟨⟨S_C, Str ''Red''⟩⟩"
 ...
end
```

Listing 5.1: Wine Universe Specification, excerpts

The code for the DDV module can be fully automated, and is presented in Appendix B. The specification of the DOnt module is based on the domain conceptualisation, illustrated in Figure 5.4. While the atomic concepts are guided by the existing dataset, the domain expert needs to provide the names of the concepts (and later their mapping to the sorts of the universe), as well as the rooted graphs details. We present excerpts of the specification in Listing 5.2

```
...
datatype wine = name_at | sugar_at | colour_at | body_at | producer_at |
   region_at

definition name :: "wine concept" where "name = {name_at}"
...
```

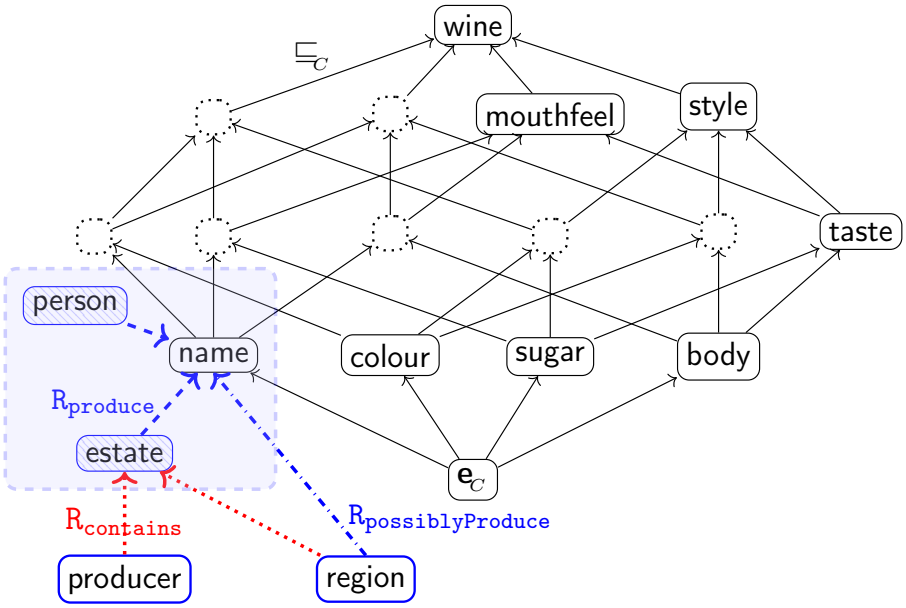

Figure 5.4: Wine Domain Ontology

```
definition wine :: "wine concept" where
    "wine = name ⊕ sugar ⊕ body ⊕ colour"
...
definition A_L :: "wine concept set" where
    A_L = {name, sugar, body, colour}"
...
definition V_r :: "wine concept set" where "V_r = {producer, region}"
definition R_r :: "wine edge set" where
    "R_r = {(region, producer), (producer, name)}"
definition G_r :: "wine rgraph" where
    "V_r = (|vertices = V_r, edges = V_r, root = name |)"
definition 𝒢 :: "wine rgraph set" where "𝒢 = {G_r}"

locale wine_dont =
 dont: domain_ontology where A_C = A_C and A_L = A_L and 𝒢 = 𝒢
for A_C :: "wine concept set"
 and A_L :: "wine concept set"
 and 𝒢 :: "wine rgraph set" +
assumes
    atom_l_def: "A_L = WineDOnt.A_L"
and atom_c_def: "A_C = WineDOnt.A_C"
```

```
    and graphs_def: "𝒢 = WineDOnt.𝒢"
begin
end (* locale wine_dont *)
```

Listing 5.2: Wine DOnt Specification, excerpts

Now the domain expert may reason on the wine DOnt structure, as illustrated in Listing 5.3

```
    lemma (in wine_dont) "producer ∉ A_L"
      by (simp add: A_L_def atom_l_def producer_def name_def sugar_def
          body_def colour_def)
```

Listing 5.3: Wine DOnt Specification, reasoning example

Finally, the domain expert creates the wine DIS. Their main job is to define the `sort2atom` mapping that is the foundation of the mapping operator τ. In addition, the domain expert may define datascape concepts, such as `whiteWine`, and use them in reasoning on the Wine DIS. We explore elements of reasoning in Chapter 7. The Wine DIS specification is illustrated in Listing 5.4.

```
theory WineDIS
  imports
    DomainInfSys
    WineDDV
    WineDOnt
begin

 locale wine_dis = wine_ddv + wine_dont +
    fixes sort2atom :: "sort ⇒ wine concept"
    assumes s2a_def: "sort2atom s = (if s = S_W then name else
                                    (if s = S_C then colour else
                                     (if s = S_B then body else
                                      (if s = S_S then sugar else e_c))))"
 begin
    interpretation dis A_C A_L 𝒢 𝒰 sort2name name2sort D sort2atom
        apply unfold_locales
        done

    definition wine_type :: "sdata ⇒ wine set" ("τ_W _" [99] 100) where
        "τ_W a = τ a"
    ...
    definition whiteWine :: "sdata set ⇒ dataconcept" where
```

```
        "whiteWine X = {a | a. a ∈ X ∧ τ_W a = wine}"
    ...
  end
end
```

Listing 5.4: Wine DIS Specification, excerpts

A large part of a concrete specification can be semi-automated in two ways. First, the atoms of the two Boolean algebras, as well as mapping operator can be lifted directly from the dataset, as described in more detail in Chapter 6. Second, rooted graphs (and implicitly the monoid of concepts) can be specified using *template*s, as discussed in Chapter 6. The input of the domain expert is needed for both stages, mainly during the definition of the mapping operator, and the rooted graphs.

5.4 Conclusion

This chapter presented the formal specification of the DIS within the Isabelle/HOL proof assistant. The generic DIS theory was modularly defined across its components: the Universe, the DDV, and the DOnt, with the DIS specification encompassing them. Each component was specified using Isabelle locales, inheriting from foundational mathematical structures such as cylindric algebras, Boolean algebras, commutative monoids, and acyclic directed graphs. This formalisation enables any instance of the DIS theory to be directly verified and reasoned about within a machine-checked environment.

To illustrate the practical application of the DIS specification, a concrete instance, the Wine DIS, was implemented using real-world dataset structures. The chapter demonstrated how key specification elements (e.g., types, sorts, concepts, and mappings) are defined, and highlighted how the modular design enables partial automation. The automation process is further details in Chapter 6. Because of its layered construction, a DIS instance can represent domain concepts at multiple levels of abstraction, enabling reasoning across both general and fine-grained conceptual views. This multi-resolution capability, grounded in a formally verified specification, provides the foundation for the reasoning tasks explored in Chapter 7.

Chapter 6

DIS Automation

In order to support the end-user, we offer a set of templates for a DIS implementation. These templates can be completed manually, by the domain expert or in a semi-automated way, by creating an input file that is supplied to a transformation tool. As the construction of the DIS is guided by the data, the generation of this input file may be partially automated. There are elements of the DIS that can be extracted from the dataset, such as the entire DDV (the sorts, the universe and its mapping operators, along with the original set of s-datas), the core of the DOnt (the atomic concepts of its Boolean lattice), and the mapping operator of the DIS. The domain expert would need to create the input for other DOnt elements, such as composite concepts, other atomic concepts (not captured in the Boolean lattice), and the rooted graphs.

In this chapter, we present these DIS templates. In Section 6.1, we detail the extended BNF used to write the DIS templates. In Section 6.2, we offer a quick overview of the templates used to implement a DIS. In Sections 6.3, 6.4, 6.5, and 6.6, we outline the production rules of the templates for the `DDVUniverse`, `DomainDtVw`, `DomainOnt`, and `DomainInfSys` modules, respectively.

6.1 Foundational Elements

In this section, we present the syntax of the DIS template, in an extended BNF described below. When describing the actual content of a template, the content is specified as a set of production rules. The production rules define how *nonterminal symbols* can be expanded into sequences of *terminal symbols* (i.e., actual content) or other nonterminal symbols. Nonterminal symbols are enclosed between a pair of angle brackets $\langle \ldots \rangle$. Optional terms are enclosed between pairs of square brackets $[\ldots]$. Terms that may appear zero or more times are superscripted with a star

*, and terms that may appear one or more times are superscripted with a plus character $^{+}$. In addition to Isabelle keywords, the DIS templates may contain *tokens* and *code delimiters*. A token is enclosed between a pair of # characters. At the time of parsing, a token is replaced with specific values. In contrast, code delimiters are removed during parsing, and they can determine either the beginning and end of a list (called *list delimiters*), or a condition to be tested (called *condition delimiters*). Code delimiters are enclosed between a pair of {! and !} symbols. List delimiters start with !`foreach #token#`! and end with !`eforeach`!. Similarly, condition delimiters start with !`if condition`! and end with !`eif`!. Any symbols or keywords not explicitly described above are part of the standard Isabelle syntax. Isabelle-specific elements, such as commands and keywords, are shown in a `truetype` font for consistency. Throughout the text, the colours used in the original code listings are omitted for clarity.

To simplify the BNF expressions throughout this section, we define a helper, called !`listgen`! that takes three arguments: a mandatory `#token#`, an optional separator `sep=sepchar`, and an optional list item `lstitem=item`. The default separator is ",", and the default list item is the token itself (the first argument). The expression !`listgen #token# sep=sepchar lstitem=item`! stands for the following template structure:

```
!foreach #token#!
    !if 'position neq first! sepchar !eif! item
!eforeach!
```

In an Isabelle theory file, the main body, which consists of formal specification and proof commands, may be augmented with informal text, such as *document comments* or *markup commands*. A document comment is enclosed between the Isabelle pair of symbols $(* \ldots *)$. After generating a document from the theory file, comments do not appear in the final PDF. The markup commands provide a structured way to insert text into the document that may be generated (by Isabelle) from a set of theory files. Each markup command takes a single argument, enclosed between special markup symbols (a pair of double angle brackets $<< \cdots >>$). The arguments of the markup commands appear in the generated PDF document.

After creating a template, the theory file may be used to generate a user-friendly PDF file. As such, after generating the template, it is recommended for the ontology expert to add sections, subsections, and comments as needed, throughout the theory file. This may be done using Isabelle markup commands such as `section`, `subsection`, and `text`. In the generated PDF, the argument of the `section` or `subsection` markup command becomes the title of the section or subsection, respectively. The argument

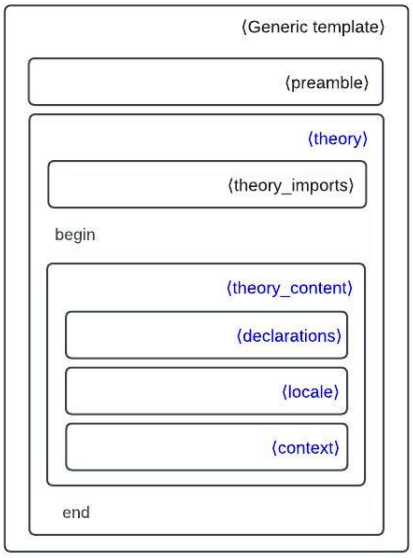

Figure 6.1: Generic DIS Template overview

of the `text` markup command becomes a detailed description in the generated PDF. Note that any time the template is regenerated, the manually added elements are removed by the template generation process.

6.2 Templates Overview

For any given DIS implementation, the ontology engineer can generate a template for each main DIS component, namely: `#TheoryName#Univ`, `#TheoryName#DDV`, `#TheoryName#DOnt`, and `#TheoryName#DIS`. All four templates follow a similar format, shown in Figure 6.1. Each nonterminal symbol of the template is a mix of Isabelle code (i.e., terminal symbols, not pictured) and a (possibly optional) set of other nonterminal symbols. The nonterminal symbols shown in blue contain other nonterminal symbols, and the nonterminal symbols shown in black contain only terminal symbols. In Isabelle, a theory can contain multiple locales. Isabelle allows the same name for both a theory and a locale, as they are objects of different kinds for the ML. To clearly differentiate between the two objects, we name a theory `#TheoryName#COMP` and a locale `#TheoryName#_comp`, following Isabelle notation standards. The `COMP` token corresponds to a DIS component, such as `Univ`, `DDV`, `DOnt`, or `DIS`, and the `comp` to its lower case version.

In general, a DIS template main production rule has two main nonterminal symbols, the ⟨*preamble_comp*⟩ and the ⟨*theory_comp*⟩. The production rule for the ⟨*preamble_comp*⟩ nonterminal contains the meta information of the document module, which is almost identical for each template, and describes the module and its content. The production rule for the ⟨*theory_comp*⟩ nonterminal contains two nonterminal symbols, ⟨*theory_imports_comp*⟩ and ⟨*theory_content_comp*⟩. The production rule for the ⟨*theory_imports_comp*⟩ nonterminal lists the theories required for the current module to run correctly and it is specific to each DIS component. The production rule for the ⟨*theory_content_comp*⟩ nonterminal contains three nonterminals: ⟨*declarations_comp*⟩, ⟨*locale_comp*⟩, and ⟨*context_comp*⟩. The production rule for the ⟨*declarations_comp*⟩ nonterminal lists basic elements of the theory, such as the sorts, generator set, or atomic concepts, etc., and composite elements, such as named concepts or rooted graphs. The production rule for the ⟨*locale_comp*⟩ nonterminal contains the definition of the main locale. It details the instantiation of the specific DIS component it represents, using elements defined within the ⟨*declarations_comp*⟩. Finally, the production rule for the ⟨*context_comp*⟩ nonterminal defines a place where the domain expert may add any other relevant information about the theory, including any specific reasoning elements. The ⟨*preamble_comp*⟩, ⟨*theory_comp*⟩, and ⟨*context_comp*⟩ nonterminals are similar to each other, regardless of the DIS component to which they belong. In Appendix C.1 we provide the production meta-rules for these nonterminals.

6.2.1 Generic Template: Preamble

The preamble section of each template follows the same generic structure. The production rule of the ⟨*preamble_comp*⟩ nonterminal symbol includes meta data about the `COMP` theory file (provided as a document comment) and descriptions of the theory file content (provided as the markup commands `section` and `text`). The values of the tokens used in the ⟨*preamble_comp*⟩ production rule must be defined by the domain expert, i.e., they cannot be automatically generated from the dataset.
The document comment (enclosed between the (∗ and ∗) symbols) includes the name of the theory, its author(s), and the date the file has been last updated. As per Isabelle requirements, the resulting theory file must be saved under the exact name `#TheoryName#COMP.thy`. For the arguments of the two markup commands it is always a good idea to provide as much detail as possible. The production rule of the preamble nonterminal symbol is defined as follows:

⟨*preamble_comp*⟩ ::=
```
(* Title:       #TheoryName#COMP.thy
   Author(s): #authorList#
   Date:       #date#
```

```
*)
section <<#TheoryName#COMP Theory>>
text <<#TheoryNameDDVDescription#>>
```

6.2.2 Generic Template: Theory

The ⟨*theory_comp*⟩ nonterminal symbol describes the current `COMP` theory. As required by the syntax of Isabelle theories, the production rule starts with the theory Isabelle keyword, followed by the name of the theory. The production rules for the ⟨*theory_comp*⟩ and ⟨*theory_content_comp*⟩ nonterminals are defined as follows:

⟨*theory_comp*⟩ ::= `theory #TheoryName#COMP`
⟨*theory_imports_comp*⟩
`begin`
⟨*theory_content_comp*⟩
`end`

⟨*theory_content_comp*⟩ ::= ⟨*declarations_comp*⟩
⟨*locale_comp*⟩
⟨*context_comp*⟩

In the next sections, we detail the production rules that are specific to each template, i.e., the ⟨*theory_imports_comp*⟩, ⟨*declarations_comp*⟩, and ⟨*locale_comp*⟩. For each nonterminal symbol, we point out the elements that can be fully automated and the ones that require input from the domain expert. The Isabelle implementation of all four templates are provided in Appendix C.

6.3 Universe Template

The universe template module, named `#TheoryName#Univ.thy`, contains details about the DDV universe. It implements the universe of the DIS, by listings the sorts of the DIS, the definition of the two helper operators, and the instance of the universe theory, as described in Section 5.2. The overview of the universe module is shown in Figure 6.2 and it follows the structure described in Section 6.2. The production rules for all nonterminals are defined in Appendix C.2.

6.3.1 Universe Template: Theory Imports

The ⟨*theory_imports_univ*⟩ nonterminal lists the theories required by the Universe theory to run correctly. As the Universe is the building block of the other DIS components, the only theory module it needs to import is the abstract `DDVUniverse`. Thus the production rule is as follows:

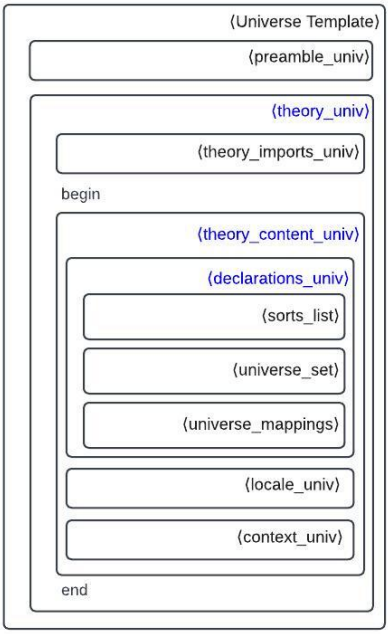

Figure 6.2: Universe Template overview

⟨*theory_imports_univ*⟩ ::= imports DDVUniverse

6.3.2 Universe Template: Declarations

The ⟨*declarations_univ*⟩ nonterminal list lists elements specific to the universe, such as the list of sorts, including the definition of the universe itself and the definition of two helper operators, mapping a sort object to its name, and vice versa. The list of sorts can be automatically lifted from the set of attributes of the dataset. The production rules of the nonterminals are defined as follows:

⟨*declarations_univ*⟩ ::= ⟨*sorts_list*⟩ ⟨*universe_set*⟩ ⟨*universe_mappings*⟩

⟨*sorts_list*⟩ ::=

```
!foreach #sort#!
definition #sort# :: sort where
  "#sort# = #sort_type# {!listgen #value#!}
!eforeach!
```

⟨*universe_set*⟩ ::=

```
definition 𝒰 :: "sort set" where
  "𝒰 = {!listgen #sort#!}
```

```
⟨universe_mappings⟩ ::=
definition #thname#_sort2name :: "sort ⇒ string" ("s2n _") where
 "(s2n s) = !foreach #sort#!
              (if s=#sort# then ''#sort#'' else
                !if 'position eq last'! '''' !eif!
             !eforeach!
             !foreach sort#! ) !eforeach!"
definition #thname#_name2sort :: "string ⇒ sort" ("n2s _") where
 "(n2s n) = !foreach #sort#!
              (if n=''#sort#'' then #sort# else
                !if 'position eq last'! {} !eif!
             !eforeach!
             !foreach sort#! ) !eforeach!"
```

6.3.3 Universe Template: Locale

Finally, in the ⟨*locale_univ*⟩, all elements come together, as the definition of the `universe` locale instantiation. The production rules for this nonterminal is defined as follows:

```
⟨locale_univ⟩   ::=
locale #TheoryName#_univ =
 #TheoryName#_univ: universe where 𝒰 = 𝒰 and
           sort2name = sort2name and name2sort = name2sort
  for 𝒰 :: "sort set"
   and sort2name :: "sort ⇒ string"
   and name2sort :: "string ⇒ sort" +
  assumes univ_def: "𝒰 = #TheoryName#Univ.𝒰"
   and s2n_def: "sort2name s = (s2n s)"
   and n2s_def: "name2sort n = (n2s n)"
begin
end
```

6.4 Domain Data View Template

The DDV component is implemented within the DDV template module, named `#TheoryName#DDV.thy`. The `universe` locale, defined in Section 6.3.3, is extended with the original dataset (i.e., the generator set), and it may be augmented with

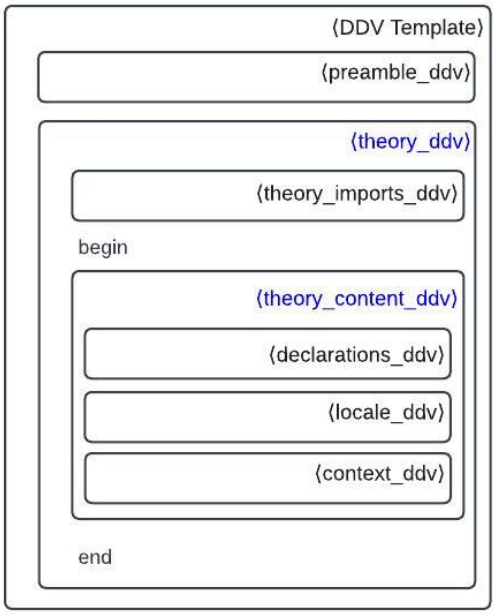

Figure 6.3: Domain Data View Template overview

reasoning tasks that do not involve the DOnt. The general overview of this template is shown in Figure 6.3. The template follows closely the structure described in Section 6.2. The production rules for all nonterminals are defined in Appendix C.3.

6.4.1 DDV Template: Theory Imports

The DDV module requires two modules to be imported. One is the previously defined `universe` module, in which the universe of the DIS has been introduced. The second one is the `DomainDtView` module, needed for its definition of the DDV locale, which is instantiated in module. Thus, the production rule for the imports nonterminal is defined as follows:

⟨*theory_imports_ddv*⟩ ::= imports
#TheoryName#Univ
DomainDtView

6.4.2 DDV Template: Declarations

Within the DDV template, there is no need for any other declarations than the generator set, or the original dataset, expressed as a set of s-datas. The generator set is passed as an argument to the instantiation of the DDV locale, described in Section 6.4.3. The production rule for the declarations nonterminal is defined as follows:

⟨*declarations_ddv*⟩ :: =

```
context #TheoryName#_univ
begin
definition #TheoryName#_generator :: "sdata set"
   where "#TheoryName#_generator = {!listgen #sdata#!}"
end
```

6.4.3 DDV Template: Locale

Finally, the bulk of the theory is defined by the ⟨*locale_ddv*⟩ nonterminal. Its production rule defines the concrete `#TheoryName#_ddv` locale as an extension of the universe locale (i.e., `#TheoryName#_univ`) and an instantiation of the generic `ddv` locale, as follows:

⟨*locale_ddv*⟩ ::=

```
locale #TheoryName#_ddv = #TheoryName#_univ +
   #TheoryName#ddv: ddv where D = D
   for D :: "sdata set" +
assumes "D = #TheoryName#_generator"
begin
end
```

6.5 Domain Ontology Template

The DOnt component of the DIS is implemented within the DOnt template module `#TheoryName#DOnt.thy`. Its main locale extends the generic `domain_ontology` locale, described in Section 5.2. The general overview of this template is shown in Figure 6.4. The production rules for all nonterminals are defined in Appendix C.4.

6.5.1 DOnt Template: Theory Imports

The locale for the DOnt an instantiation of the `domain_ontology` locale, which is defined inside the `DomainOnt` module. Thus, this is the only module required as an

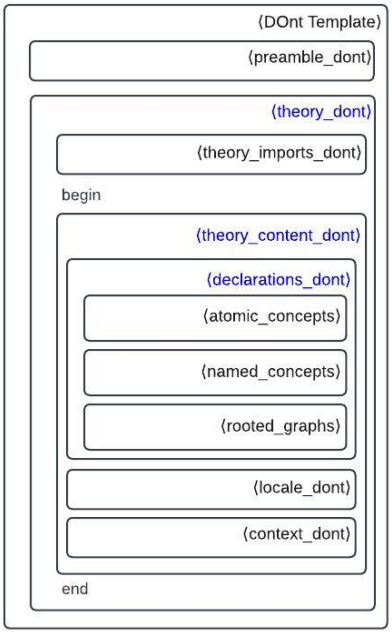

Figure 6.4: Domain Ontology Template overview

import for the DOnt module. The production rule for the imports nonterminal is defined as follows:

⟨*theory_imports_dont*⟩ ::= `imports DomainOnt`

6.5.2 DOnt Template: Declarations

The declarations nonterminal of the `#TheoryName#DOnt` module contains three nonterminals. The nonterminal ⟨*atomic_concepts*⟩ lists the set of atomic concepts used to create both the monoid of concepts and the lattice of concepts. The optional nonterminal ⟨*named_concepts*⟩ and nonterminal ⟨*rooted_graphs*⟩ outline complex elements of DIS, as described below. The production rule for the declarations nonterminal is defined as follows:

⟨*declarations_dont*⟩ ::= ⟨*atomic_concepts*⟩
[⟨*named_concepts*⟩]
⟨*rooted_graphs*⟩

The production rule for the ⟨*atomic_concepts*⟩ nonterminal lists the atomic concepts of the DOnt, as well as the set of atomic concepts used to generate the lattice of

concepts. Note that the ‖ below stands for the character | in Isabelle/HOL, to differentiate it from the BNF meta-character |. The input for the lattice of concepts atoms can be extracted (automatised) from the attributes of the dataset, and the domain expert may be involved in renaming them. The input for the other atoms used in the monoid of concepts must be provided by the domain expert. The production rule for the atomic concepts nonterminal is defined as follows:

⟨*atomic_concepts*⟩ ::=
```
datatype #thname# = !listgen #atom# sep=‖ lstitem=#atom#at!
!foreach #atom#!
definition #atom# :: "#thname# concept"
   where "#atom# = {#atom#at}"
!eforeach!
definition AL :: "#thname# concept set"
   where "AL = {!listgen #lattice_atom#!}"
```

The ⟨*named_concepts*⟩ nonterminal is defined by explicitly specifying the composition of existing concepts, whether atomic or other named concepts. The input for the ⟨*named_concepts*⟩ nonterminal is provided by the domain expert and cannot be automated from the existing dataset. The production rule for the ⟨*named_concepts*⟩ nonterminal is defined as follows:

⟨*named_concepts*⟩ ::=
```
!foreach #named_concept#!
definition #named_concept# :: "#thname# concept"
   where #named_concept# = !listgen #concept# sep=⊕!
!eforeach!
```

Finally, the ⟨*rooted_graphs*⟩ nonterminal consists of a list of rooted graphs. Each rooted graph, in turn, is defined using three terminal symbols: one for the set of vertices in the graph `#V#`, one for the set of edges (or the relations) of the graph `#R#`, and one for the graph itself `#G#`. The set of vertices consists of a list of concepts, and the set of edges consists of a list of pair of concepts. All concepts used in the rooted graphs must be defined prior to their use within the graphs definitions. They may be atomic concepts, named concepts, or unnamed composition of concepts (i.e., inline composition of concepts). The production rule for the ⟨*rooted_graphs*⟩ nonterminal is defined as follows:

⟨*rooted_graphs*⟩ ::=
```
!foreach #rooted_graph#!
definition #V# :: "#thname# concept set"
   where "#V# = {!listgen (#vertice#)!}"
definition #R# :: "#thname# edge set"
```

```
    where "#R# = {!listgen #edge# lstitem=(#tail#, #head#)!}"
definition #G# :: "#thname# rgraph"
    where "#G# = (|vertices = V, edges = R, root = #root#|)"
!eforeach!
```

6.5.3 DOnt Template: Locale

Finally, the ⟨*locale_dont*⟩ nonterminal instantiates the `domain_ontology` locale. Its production rule is defined as follows:

⟨*locale_dont*⟩ ::=
locale #TheoryName#_dont =
    dont: domain_ontology where A_C = A_C and A_L = A_L and $\mathcal{G}$ = $\mathcal{G}$
for A_C :: "#thname# concept set"
    and A_L :: "#thname# concept set"
    and $\mathcal{G}$ :: "#thname# rgraph set" +
assumes atom_l_def: "A_L = #TheoryName#DOnt.A_L"
    and atom_c_def: "A_C = A_L ∪ {!listgen #other_concept#!}"
    and graphs_def: "$\mathcal{G}$ = {!listgen #rg#}"
begin
end

6.6 Domain Information System Template

The DIS template connects the DDV and the DOnt components and offers the user a space to declare the datascape concepts and to perform any reasoning tasks that may include elements of DDV and the DOnt, along with datascape concepts. The general overview of this template is shown in Figure 6.5. The production rules for all nonterminals are defined in Appendix C.5.

6.6.1 DIS Template: Imports

The imports nonterminal lists both DDV and DOnt modules, along with the `DomainInfSys` module, in which the theory of the domain information system resides. Its production rule is as follows:

⟨*theory_imports_dis*⟩ ::= imports
                    DomainInfSys
                    #TheoryName#DDV
                    #TheoryName#DOnt

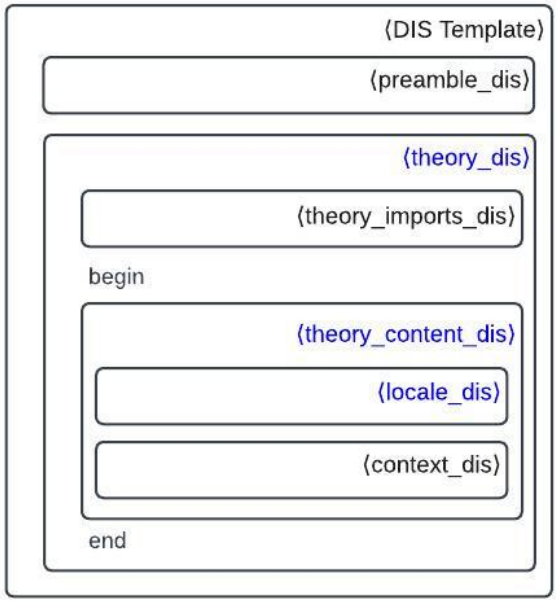

Figure 6.5: Domain Information System Template overview

6.6.2 DIS Template: Declarations

In the DIS template, the declarations nonterminal is empty.

6.6.3 DIS Template: Locale

The locale nonterminal instantiates the `dis` locale as an extension of the current `#TheoryName#_ddv` and `#TheoryName#_dont` locales. In addition, it specifies the η operator and the mapping operator, defined in Section 4.5. Finally, the locale nonterminal lists the datascape concepts. None of the elements of the `locale_dis` can be automatically lifted from the original dataset, they are all manually entered by the domain expert. The production rule for this nonterminal is defined as follows:

$\langle locale\_dis \rangle$ ::= $\langle locale\_core\_dis \rangle$
$[\langle datascape\_concept\_definition \rangle]^+$
$\langle locale\_core\_dis \rangle$ ::=

```
locale #TheoryName#_dis = #TheoryName#_ddv + #TheoryName#_dont +
fixes sort2atom :: "sort ⇒ #thname# concept"
assumes s2a_def: "sort2atom s = !foreach #sort#!
              (if s=#sort# then ''#atom#'' else
```

```
                !if 'position eq last'! '''' !eif!
            !eforeach!
            !foreach sort#! ) !eforeach!"
begin
interpretation dis 𝒜_C 𝒜_L 𝒢 𝒰 sort2name name2sort D sort2atom
    apply unfold_locales
    done
definition #thname#_type :: "sdata ⇒ #thname# set"
        ("τ_#thnamei# _" [99] 100) where
    "τ_#thnamei# a = τ a"
end

⟨datascape_concept_definition⟩ ::=
definition #name# :: "sdata set ⇒ dataconcept" where
    "#name# X = {a | a. a ∈ X ∧ ⟨condition⟩
                                ∧ τ_#thnamei# a = ⟨concept⟩}"
⟨concept⟩ ::= #atom# | #named_concept# | ⟨concept⟩⊕⟨concept⟩
```

In the last definition above, the ⟨*condition*⟩ term may be replaced by any logical condition on the concrete s-data. An example of a concrete implementation, including a datascape concept definition is provided in Section 5.3.

6.7 Conclusion

In this chapter, we focus on the automation of a DIS, and we provide a comprehensive framework for supporting domain experts and ontology engineers in implementing a domain-specific DIS. We introduce a set of templates that can be either edited manually or generated automatically by using specialised tools that transform the dataset files. The use of templates streamline the process of generating core DIS elements, such as the DDV and the Boolean lattice of the DOnt. By automating parts of the DIS engineering process, we reduce the effort and complexity of manual data-handling. This, in turn, facilitates efficient implementation of a DIS.

The DIS approach is aligned in spirit with recent efforts toward semi-automatic ontology design, such as the SEADOO project by Grüninger *et al.* 2023, which proposes techniques for generating ontologies from data models and natural language, leveraging repository-driven ontology matching. While SEADOO focuses on learning first-order axioms from structured and unstructured inputs using model transformation and user feedback, DIS offers a formalised structure in which such learned knowledge can be directly embedded and verified. Notably, SEADOO relies on

mathematical theories to construct ontologies from examples and counterexamples, paralleling the DIS model-theoretic grounding and compositional design. The key distinction is that DIS builds in this alignment from the outset, treating the data and conceptual layers as co-definable components of a structured theory rather than deriving one from the other after the fact.

The structure of the DIS theory and its use in ontology engineering creates a balance between data-driven automation and domain expert guidance. The use of templates enable partial automation of the engineering process, by extracting the core elements directly from existing datasets. At the same time, it acknowledges the role of the domain expert in specifying the objective level concepts, such as the composite concepts and the rooted graphs of the DIS. This approach ensures a practical and adaptable methodology, making the automation process both accessible and customisable. This emphasis on dual (automated and manual) strategies lays the foundation for scalable and user-friendly DIS solutions.

Chapter 7

Elements of Reasoning

As discussed in Section 2.3, various reasoning tasks are within the capabilities of a knowledge system. These tasks include consistency checking, concept satisfiability, classification, subsumption, and other forms of inference (Antoniou *et al.*, 2018; Schneider and Šimkus, 2020). In this chapter, we illustrate the reasoning tasks on the newly built Wine DIS. We look at these tasks through the lens of DIS and provide the Isabelle/HOL implementation of each task in the Wine Ontology. The scope of this section is not to be exhaustive, but rather to illustrate the capabilities of DIS in terms of reasoning.

In Section 7.1, we present a general overview of performing reasoning in a DIS. To create the foundation for performing reasoning tasks, in Section 7.2, we extend the illustrative example started in Chapter 5, Section 5.3. Then, in Sections 7.3, 7.4, 7.5, and 7.6, we present examples of consistency checking, concept satisfiability, classification and subsumption, and inference, respectively. The reasoning task samples have been checked on the Isabelle/HOL implementation of the Wine Ontology. Finally, in Section 7.7, we reflect on the current state and limitations of DIS and propose future research directions.

7.1 Reasoning in DIS

The DIS supports reasoning by formalising the relationship between structured data and domain concepts through algebraic and relational constructs. Reasoning tasks traditionally associated with DL, such as consistency checking, satisfiability, classification, and inference, are naturally framed in DIS through its modular structure: the cylindric algebra view of the structured data, the Boolean lattice of concepts, the mapping between sorts and atoms, and the relational graphs among concepts. Core algebraic properties, such as subset relations, composition, and

concept intersection, are used to model reasoning tasks internally, while exposing a simplified conceptual view to the user.

Thus, DIS enables formal reasoning over knowledge systems while hiding the underlying complexity of algebraic manipulation. In the following sections, we describe how classical reasoning tasks are adapted and interpreted within the DIS framework.

7.2 Wine Ontology, Extended Example

In Chapter 5, Section 5.3 we constructed the Wine DIS, as a part of a more complex Wine domain of application. For our reasoning example, we build two more DISs, the Producer DIS, based on the dataset presented in Table 7.1, and the Estate DIS, based on the dataset presented in Table 7.2. Taken together, we refer to the three DISs as a "Wine Domain".

Producer	**Estate**	**Grape**	**Wine Name**
Ferrocinto		Magliocco	Magliocco
Librandi	Rosaneti	Magliocco	Meganio
Inniskillin	Niagara	Vidal	Vidal Icewine
Inniskillin	Niagara	Chardonnay	Chardonnay
Inniskillin	Montague	Chardonnay	Chardonnay
Inniskillin	Montague	Merlot	Merlot
Pelee Island	Pelee Island	Chardonnay	Lola
Pelee Island	Pelee Island	Vidal	Vidal
...	...	...	...

Table 7.1: Producer dataset

The universe of Producer DIS contains four sorts, corresponding to the attributes of the dataset schema: producer $\mathtt{S}_{\mathtt{P}}$, estate $\mathtt{S}_{\mathtt{E}}$, grape $\mathtt{S}_{\mathtt{G}}$, and wine name $\mathtt{S}_{\mathtt{W}}$. If the data is clean, the ontology engineer may create one sort for $\mathtt{S}_{\mathtt{W}}$, and refer to it in both the Wine and the Producer DISs. If, instead, the domain expert wishes to use the two DISs to clean the data or they are unsure if the data is clean, each $\mathtt{S}_{\mathtt{W}}$ must be defined (as different sorts) in their respective DIS,.

The universe of the Estate DIS contains only three sorts, the producer $\mathtt{S}_{\mathtt{P}}$, estate $\mathtt{S}_{\mathtt{E}}$, and the region $\mathtt{S}_{\mathtt{R}}$. The three DISs, Wine, Producer, and Estate are linked through

Producer	Estate	Region
Ferrocinto		Calabria
Librandi	Rosaneti	Calabria
Librandi	Ponta	Calabria
Inniskillin	Niagara	Niagara
Inniskillin	Montague	Okanagan
Pelee Island	Pelee Island	Niagara
Pelee Island	Kingsville	Southern Ontario

Table 7.2: Estate dataset

the rooted graphs.

A more detailed version of the Wine Ontology is shown in Figure 7.1. Concepts originating from the Wine DOnt are based on the data in Table 5.5 and depicted in black, those originating from the Producer DOnt are based on the data in Table 7.1 and depicted in red, and concepts originating from the Estate DIS are based on the data in Table 7.2 and depicted in blue. In this example, we assume that the data is clean, i.e., the data of the common attributes coincide on the three DISs). The concepts `estate.name` and `producer.name` are shown as shared concepts between the Estate and Producer DOnts. For clarity of presentation, the concept `wine.name` is depicted separately within the Wine and Producer DOnts, although it represents a unique, common concept across the two. The relations given "horizontally", by the tuples of the Producer DDV, become rooted graphs of the Wine DOnt. Thus, the link between `producer` and `estate` translates into "`producer` $\mathtt{R}_{\mathtt{contains}}$ `estate`". Similarly, the Estate DDV relation between `region` and `estate` translates into "`region` $\mathtt{R}_{\mathtt{contains}}$ `estate`", and the Estate DDV relation between `wine` and `estate` translates into "`estate` $\mathtt{R}_{\mathtt{produces}}$ `wine`". The reasoning examples given in the remainder of this chapter are based on this understanding of the Wine domain of application. We now illustrate how classical reasoning tasks are instantiated over this extended domain.

7.3 Consistency Checking

In DL, consistency checking ensures that the KB admits a model, by answering the question "is the A-Box consistent to the schema implied by T-Box?" In DIS, consistency checking ensures that the DDV and DOnt are coherently aligned through the surjective mapping from sorts to atomic concepts and the mapping operator. At the conceptual level, consistency requires that each s-data maps to a well-formed

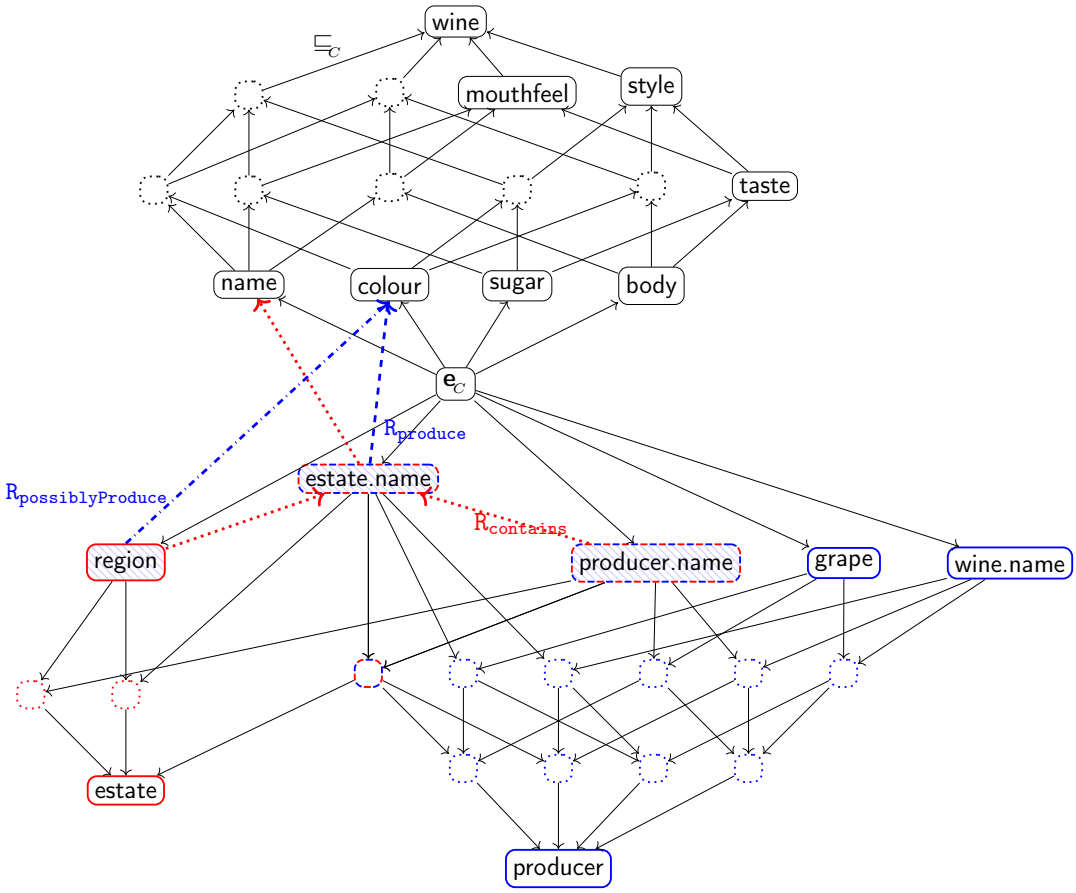

Figure 7.1: Wine Domain: Wine, Producer, and Estate Domain Ontology

concept in the ontology. This is ensured structurally: the mapping from sorts to atoms is total and surjective, and the mapping operator τ computes a unique concept for each s-data, based on its sorts.

At the system level, consistency concerns arise when aligning or integrating multiple DIS instances. The question becomes whether the sorts (and thus their corresponding atoms) are coherent across the system. Two DISs are consistent with each other if overlapping sorts are semantically and structurally equivalent, that is, they reference the same underlying concept and contain the same values. Formally, given two DISs $\mathcal{I}_1$ and $\mathcal{I}_2$, we say that two sorts $S_1 \in \mathcal{U}_1$ and $S_2 \in \mathcal{U}_2$ are consistent with each other if $S_1 = S_2$. In the Wine domain of application, this may translate into the query: "Are the sorts in the Wine DIS consistent with the ones in the Producer one?" E.g., a value for *Wine* sort may be listed in one DIS and not in the other.

Attribute consistency refers to the ability to check if the DDV is consistent w.r.t. the specific rules of the DIS. For example, in the Wine Ontology, the domain expert may capture the rule "A wine cannot be red and white at the same time", and express it through the use of two datascape concepts, as $whiteWine^D \cap redWine^D = \{\}$. The consistency check is performed by the Isabelle/HOL by proving the statement is true within the given Wine DIS. The Isabelle code is presented in Listing 7.1.

```
context wine_dis
begin
text ⟨Attribute consistency: same wine cannot be different colors in the
    generator set⟩
lemma (in wine_dis) reds_whites_int: "reds ∩ whites = {}"
  using D_def wine_generator_def reds_def whites_def
    WineDDV.wine_ddv.axioms local.univ_def
    wine_ddv.dt_in_SD wine_ddv.dt_v_in_k by blast
end
```

Listing 7.1: Elements of Reasoning: Consistency Checking

7.4 Concept Satisfiability

In DL, assessing the satisfiability of a concept C is equivalent to finding an interpretation I, s.t. $C^I \neq \varnothing$. In DIS, concept satisfiability verifies whether a given concept admits at least one instance within the structured data. Formally, a concept is satisfiable its corresponding datascape is non-empty. In DIS, satisfiability can be applied both to objective concepts, which are defined at the ontology level

(as described in Section 3.3.1), and to datascape concepts, which are constructed dynamically via constraints (as described in Section 4.6. The underlying Boolean lattice structure and relational graphs reduce satisfiability checking to set-theoretic operations. In the following we discuss in more detail the concept satisfiability of both kinds.

7.4.1 Objective Concept Satisfiability

In DIS, objective concepts are satisfiable if they admit at least one corresponding s-data in the associated DDV. This reduces to verifying whether the extension of the concept (i.e., its associated datascape concept) is non-empty. Concepts built from atoms using lattice operators are always satisfiable by construction, as their atomic constituents are built from the DDV, and there is always at least one s-data connected to the Boolean lattice concepts. Thus, for objective concepts, satisfiability checking becomes a structural validation within the Boolean lattice. Ensuring the feasibility of the concepts outside the lattice is a similar straightforward process, involving verification that they align with a Boolean lattice concept present in another DIS. Named concepts are composition of the first two types, therefore their feasibility is a straightforward process.

Within the Wine DIS example, a possible reasoning task for rooted graph concepts is answering the question "Is the `Estate` concept feasible?" As all the named concepts within the Wine DIS are defined as composition of atomic concepts only, they are inherently feasible, just as their parts (i.e., the atomic concepts) are.

7.4.2 Datascape Concept Satisfiability

Datascape concepts are instantiated over data, either the carrier set of the DDV or its subsets, such as the generator set. When instantiated over the DDV carrier set, the datascape concepts are inherently satisfiable, due to the construction of the DIS. This is because the carrier set is built to contain all possible Cartesian combinations of the s-values of the DDV universe. In contrast, for datascape concepts instantiated over restricted datasets their satisfiability must be checked explicitly. The process is lightweight: the mapping operator τ derives the concept type, and the instantiating set is used to evaluate whether any such mappings yield a match. This evaluation is internal to the system.

In the Wine domain, an example of datascape concept satisfiability translates into checking there exist at least one s-data corresponding to the $redWine^{D}$. The Isabelle

code is presented in Listing 7.2.

```
context wine_dis
begin
text ⟨Datascape concept satisfiability: The red wines ⟨reds⟩ contains at
    lest one s-data instance.⟩
lemma (in wine_dis) reds_satisfiable: "reds ≠ {}"
  using merlot_in_reds by auto
end
```

Listing 7.2: Elements of Reasoning: Concept Satifiability

Another example of concept satisfiability is checking the logical consistency of datascape concepts. For example, in the Wine Ontology, there is a rule stating that a wine can only have one colour. The domain expert defines a concept, $RedWhiteWine^{\mathcal{D}} = redWine^{\mathcal{D}} \cap whiteWine^{\mathcal{D}}$. It is immediate that the WineDIS can satisfy either the rule or the concept, and not both. Thus, within a DIS with this rule, the $RedWhiteWine^{\mathcal{D}}$ concept is not satisfiable.

7.5 Classification and Subsumption

Both the classification and subsumption reasoning tasks refer to the process of inferring concept hierarchies based on domain knowledge (Antoniou *et al.*, 2018). This hierarchy involves either relations between concepts (for subsumption) or relations between individuals and concepts (for classification). In DL, given two concepts C and D, assessing if D is subsumed by C is equivalent to checking if C is more general than D. Classification translates into checking which concepts a given individual a is an instance of.

In DIS, classification and subsumption tasks rely on the partial order structure of the Boolean lattice. Classification determines whether a given s-data belongs to the extension of a particular concept, following the mappings defined by the mapping operator. Subsumption checks whether one concept is included within another, based on subset relations among their atomic components. The partial order embedded in the concept lattice allows subsumption queries to be evaluated through algebraic comparison, while classification queries reduce to verifying membership within the mapped datascape concept.

In the Wine domain an example of subsumption is determining that "Red Wines are Wines". Addressing this query involves a two-step process: (i) defining the "Red

Wines" datascape (i.e., $redWine^D$) and (ii) verifying that $redWine^D \subseteq wine^D$. Determining whether "s-data a belongs to the concept $redWine^D$" essentially becomes a matter of membership: is $a \in redWine^D$? The Isabelle code is presented in Listing 7.3.

```
context wine_dis
begin
    text ⟨Classification: The merlot wine is classifed as a red wine.⟩
    lemma (in wine_dis) merlot_in_reds: "{merlot, red, dry, full} ∈ reds"
    using D_def wine_generator_def reds_def by auto
end
```

Listing 7.3: Elements of Reasoning: Classification

7.6 Inference Checking

Inference checking involves evaluating the logical consequences that can be drawn from the explicit knowledge represented in an ontology. Although all previous reasoning tasks qualify as inference checking, there are other queries, hypotheses, and logical consequences that do not belong clearly to these tasks. In DIS, inference checking uses the rooted graphs, concept operators, and relational operators (such as composition and transitive closure) to infer new relations. These inference processes are embedded naturally into the DIS structure. Users interact at the level of concepts and properties, while the system internally manages the logical derivations through its algebraic and relational semantics.

In wine DIS, the inference check may include wine and grape inference, wine colour inference, regional similarities, etc. For wine and grape inference, given that a wine is produced by various producers (and their estates), and that the estate belongs to a certain region, the system may infer that a certain wine belongs to a certain region. Moreover, given the relation between grape and wine, it can be inferred that the grape of a certain wine belongs to the region of the wine. If the data is not clean, another wine and grape inference may constitute the answer to the question "What kind of wine attributes may the Meganio wine have?". Although that information is not present in the Wine DDV, based on the grape from which both the `Magliocco` and `Meganio` wines are made, similar properties can be inferred for the two wines.

Another example, illustrated in Figure 7.1, shows a region-colour inference. As estates are related to colour (through the $\mathtt{R}_{\mathtt{contains}}$ relation), and region to estate (through the $\mathtt{R}_{\mathtt{contains}}$ relation), it can be inferred that at the conceptual level the region and colour concepts are related. The Isabelle code is presented in Listing 7.4. This

inference can be further refined by involving the data. If wines from a particular region predominantly exhibit a certain colour, based on historical data or observations, the system might infer the likely colour of a new wine from the same region.

```
context wine_dis
begin
text ⟨Inference: Region relates to Estate (through R_contains),
     Estate relates to Colour (through R_produce),
     thus Region relates to Colour⟩
lemma (in wine_dis) region_colour_relation:
    "(region, colour) ∈ R_possiblyProduce"
    using R_contains_def R_produce_def R_possiblyProduce_def relcomp_def
    by (metis insertCI relcomp.simps)
end
```

Listing 7.4: Elements of Reasoning: Inference

7.7 Conclusion

This chapter demonstrated how classical reasoning tasks, such as consistency checking, satisfiability, classification, subsumption, and inference, can be adapted and executed within DIS framework. The underlying algebraic and relational structures of DIS, namely the Boolean lattice, rooted graphs, and mapping operator, support these tasks while abstracting much of the underlying complexity from the user.

The Wine DIS was implemented in Isabelle/HOL to validate the practical feasibility of these reasoning tasks. Although Isabelle is based on higher-order logic, which is undecidable in the general case, the reasoning queries within structured and domain-specific formalisms like DIS remain tractable in practice. The implementation also posed practical challenges. The initial manual construction of the DDV was laborious, underscoring the need for template-driven automation. In addition, some reasoning tasks, particularly classification, revealed performance constraints due to the way Isabelle/HOL handles set operations and evaluation. Addressing these challenges points toward future work to improve tool support for DIS-based knowledge systems, especially in terms of automation and reasoning scalability.

Despite these limitations, the findings confirm that DIS, as formalised and reasoned about within Isabelle, provides a scalable and semantically precise foundation for structured knowledge systems. It supports the specification, maintenance, and extension of ontologies grounded in real-world datasets, while exposing a clean reasoning

interface that abstracts away the complexity of its internal logic.

Chapter 8

Conclusion and Future Work

Modern knowledge systems must align real-world data sources with formal domain representations in a scalable and maintainable way. This thesis addressed this challenge by adopting Domain Information System (DIS), a previously introduced formalism that clearly separates data-level structures from domain-level conceptual models. Through this lens, the research presented here did not introduce DIS, instead offered its formalisation and an engineering process that reveals the power of DIS as a flexible, verifiable, and scalable foundation for knowledge systems.

A core contribution of this work is the semantic formalisation of DIS through the definition of two linked theoretical layers: a data theory, which captures the structure and abstraction of organised datasets, modelled as sets of tuples in Domain Data View (DDV), and a domain knowledge theory, which represents conceptual hierarchies and relations in Domain Ontology (DOnt). Their integration via the mapping operator yields a coherent framework, in which the domain knowledge is grounded in the structure of the data itself. This alignment eliminates structural mismatches and ensures that every DIS instance is a model of the core theory, thereby inheriting semantic coherence, modularity, and formal guarantees by design.

Building on this foundation, the thesis developed a formal engineering process to guide the construction of DIS instances from structured datasets. Grounded in a set of formal design principles, the methodology defines how to construct the components of a DIS, and their integration in a way that is repeatable, and adaptable to domain-specific needs. While existing approaches often rely on post-design semantic validation, DIS allows semantic constraints to be integrated from the outset. This reduces the technical overhead for users, who benefit from automatically aligned ontologies that reflect actual data structure without manual reconciliation.

The framework was implemented in the Isabelle/HOL Interactive Theorem Prover (ITP), providing a machine-verifiable specification that enables precise reasoning about system behaviour. By implementing DIS as a structured, higher-order logic theory, this work offers a foundation for building formally grounded knowledge systems that are aligned directly with underlying datasets. Despite the undecidability of higher-order logic in general, the DIS-based reasoning tasks were shown to be tractable in practice due to the modularity and formal structure of the framework.

In parallel, the DIS engineering process was extended through a template-based automation mechanism for generating DIS instances. These templates support declarative specification of sorts, concepts, and mappings, reducing the effort required to develop formalised knowledge systems. This design allows users to generate aligned ontologies directly from their data and reduces the potential error involved in manually constructing the DIS components, laying the groundwork for wider applicability in data-intensive settings.

Finally, the reasoning capabilities of the DIS were demonstrated by expressing and verifying core inference tasks, such as consistency, classification, and subsumption, within Isabelle. These tasks were shown to be supported by the internal structure of the framework itself: the lattice-based ontology model, rooted graph semantics, and the mapping operator together enable reasoning on data and concepts without exposing users to the underlying algebraic complexity. For users, this means that a single reasoning engine can capture both domain-specific logic and data-grounded inference in a coherent, modular way.

8.1 Future Work

Several promising directions for future research arise from this work. First, the manual construction of the DDV remains a bottleneck. The template mechanism introduced in Chapter 6 can be extended into a more robust toolchain that supports fully automatic DDV generation from structured data sets. The DIS framework could benefit from more automatic support for data evolution and schema change detection.

Second, this thesis focused primarily on the structural and logical aspects of knowledge representation. This foundation can be extended to a formal definition for data, information, and knowledge, in the context of the DIS theory. We have already started to work on it, with preliminary results on knowledge classification and a formal definition of knowledge generated in a DIS as the fixed point of total knowledge. Future work could explore semantic compression and knowledge refinement, enabling DIS to scale to high-volume knowledge spaces by identifying

semantic patterns and discarding redundant data.

Third, the separation of the data layer from the contextual one in a DIS enables the support for giving different contexts to the same of data. This multi-resolution knowledge space would enable the conjecture of different kinds of knowledge from a single set of organised data, and eventually the integration of new fragments of knoweldge into a more complete view of the world.

Lastly, DIS offers strong support for modularisation techniques (LeClair *et al.*, 2020), which is one approach to handling the current complexity of data and domain conceptualisation. In (Matentzoglu *et al.*, 2015), the authors recognise the lack of reasoning methods that take into account both data and domain, as well as reasoners that make use of modularisation techniques. Traditional methods of processing and analysing data must evolve. Thus, research in the field of modularisation can be extended to the use of DIS theory.

8.2 Closing Remarks

This thesis presented the DIS as a unified formalism for integrating structured data and conceptual knowledge. Through the formal definition of its components, DDV, DOnt, and their connecting mappings, the DIS provides a solid foundation for specifying and reasoning over knowledge systems. Its implementation in Isabelle/HOL demonstrates that formally grounded ontologies can be built directly from data, and validated through machine-checked inference.

By combining a modular, algebraic and relational structure with support for automated reasoning, DIS bridges the gap between practical data-driven modelling and the rigour of formal logic. The contributions of this work, ranging from formal theory development to automated tooling and verified reasoning, highlight the potential of DIS to serve as both a conceptual and operational framework for structured knowledge engineering. The use of templates and semi-automated processes to generate DIS implementation lays the foundation for the continued convergence of formal methods and real-world data systems.

Appendix A

DIS Model Proofs

In this appendix, we provide proofs for the DIS model, described in Chapter 4.

A.1 Operators on Data Properties

In this section, we present the proofs for the properties described in Section 4.3.3. We remind the reader that the set comprehension is given as $\{E \mid x \,.\, R\}$, where E, x, and R are the *expression*, (set of) *variable*(s), and *range*, respectively. In addition, all statements of the form $A \subseteq B$ are proved by showing that the equivalent equation $A \cap B = A$ holds.

For some of the proofs below, we use the following fact: for any $s \in \mathcal{U}, T_1, T_2 \subseteq \mathcal{U}$:

$$s \notin T_2 \implies \big(\{dt \cup \{(s,v)\} \mid dt, v \,.\, dt \in \bowtie T_1 \ \wedge\ v \in s\} \cap \ \bowtie T_2\big) = \{\} \tag{A.1}$$

Equation A.1 is due to the fact that $\bowtie(T_2)$ can contain no $\{(s,v)\}$ s-values, as $s \notin T_2$. Proof for Equation A.1 is done by induction on T_2:
Step 1: $T2 = \{\}$. The proof is trivial, using the definition of $\bowtie$, $\bowtie T_2 = \{\}$, and the annihilator for $\cap$ axiom.
Step 2: **IH**: $s \notin T_2 \implies \big(\{dt \cup \{(s,v)\} \mid dt, v \,.\, dt \in \bowtie T_1 \ \wedge\ v \in s\} \cap \ \bowtie T_2\big) = \{\}$.
Show that for any $s' \in \mathcal{U}$, $s \notin (T_2 \cup \{s'\}) \implies \big(\{dt \cup \{(s,v)\} \mid dt, v \,.\, dt \in \bowtie T_1 \ \wedge\ v \in s\} \cap \ \bowtie (T_2 \cup \{s'\})\big) = \{\}$.
Case 1: $s' \in T_2$ is trivial, as it reduces to the **IH**.
Case2: $s' \notin T_2$

$$\begin{aligned}
&\{dt \cup \{(s,v)\} \mid dt, v \,.\, dt \in \bowtie T_1 \ \wedge\ v \in s\} \cap \ \bowtie (T_2 \cup \{s'\}) \\
=\!= \ &\langle \text{ Definition of } \bowtie \ (4.1) \ \rangle \\
&\{dt \cup \{(s,v)\} \mid dt, v \,.\, dt \in \bowtie T_1 \ \wedge\ v \in s\} \cap \\
&\big(\ \bowtie T_2 \ \cup \{dt' \cup \{(s',v')\} \mid dt', v' \,.\, dt' \in \bowtie T_2 \ \wedge\ v' \in s'\}\big)
\end{aligned}$$

$$
\begin{array}{ll}
= & \langle \text{ Distributivity of } \cap \text{ over } \cup \rangle \\
& \big(\{dt \cup \{(s,v)\} \mid dt, v \,.\, dt \in \bowtie T_1 \ \wedge\ v \in s\}\cap \ \bowtie T_2\big) \cup \\
& \big(\{dt \cup \{(s,v)\} \mid dt, v \,.\, dt \in \bowtie T_1 \ \wedge\ v \in s\} \cap \\
& \ \{dt' \cup \{(s',v')\} \mid dt', v' \,.\, dt' \in \bowtie T_2 \ \wedge\ v' \in s'\}\big) \\
= & \langle \textbf{ IH } \rangle \\
& \{\} \cup \\
& \big(\{dt \cup \{(s,v)\} \mid dt, v \,.\, dt \in \bowtie T_1 \ \wedge\ v \in s\} \cap \\
& \ \{dt' \cup \{(s',v')\} \mid dt', v' \,.\, dt' \in \bowtie T_2 \ \wedge\ v' \in s'\}\big) \\
= & \langle \text{ Annihilator for } \cup \rangle \\
& \{dt \cup \{(s,v)\} \mid dt, v \,.\, dt \in \bowtie T_1 \ \wedge\ v \in s\} \cap \\
& \ \{dt' \cup \{(s',v')\} \mid dt', v' \,.\, dt' \in \bowtie T_2 \ \wedge\ v' \in s'\} \\
= & \langle \ \rangle \\
\\
= & \langle \ \rangle \\
& \{\}
\end{array}
$$

Proof for Proposition (4.3.2) is done by induction on T', with assumption $T' \subseteq T$:
Step 1: $T' = \{\}$

$$
\begin{array}{ll}
& \bowtie T' \cap \ \bowtie T \\
= & \langle \text{ Assumption } T' = \text{ and Definition of } \bowtie \ (4.1) \rangle \\
& \{\} \cap \ \bowtie T \\
= & \langle \text{ Annihilator of } \cap \rangle \\
& \{\} \\
= & \langle \text{ Assumption } T' = \text{ and Definition of } \bowtie \ (4.1) \rangle \\
& \bowtie T'
\end{array}
$$

Step 2: Induction hypothesis (**IH**): $T' \subseteq T \implies \bowtie T' \cap \ \bowtie T \ = \bowtie T'$. For any $s \in \mathcal{U}$, we show that $(T' \cup \{s\}) \subseteq T \implies \bowtie (T' \cup \{s\}) \ \subseteq \bowtie T$.
Case 1: $s \in T'$ is trivial, as it reduces to the **IH**.
Case 2: $s \notin T'$ and it is immediate that $s \in T$, due to assumption $(T' \cup \{s\}) \subseteq T$.
The proof will use the additional fact:

$$
\begin{array}{rl}
& \big(\{dt \cup \{(s,v)\} \mid dt, v \,.\, dt \in \bowtie T' \ \wedge\ v \in s\} \\
\cap & \{dt \cup \{(s,v)\} \mid dt, v \,.\, dt \in \bowtie (T \backslash \{s\}) \ \wedge\ v \in s\}\big) \\
= & \{dt \cup \{(s,v)\} \mid dt, v \,.\, dt \in \bowtie T' \ \wedge\ v \in s\}
\end{array}
\tag{A.2}
$$

Equation A.2 is based on the assumptions $s \notin T'$ and $T' \subseteq T$, thus $T' \subseteq (T \backslash \{s\})$, and, due to **IH**, $\bowtie T' \ \subseteq \bowtie (T \backslash \{s\})$.

$$
\begin{array}{ll}
& \bowtie(T' \cup \{s\}) \cap \bowtie T \\
= & \quad \langle \text{ Definition of } \bowtie \ (4.1) \ \rangle \\
& (\bowtie T' \cup \{dt \cup \{(s,v)\} \mid dt, v \,.\, dt \in \bowtie T' \wedge v \in s\}) \cap \bowtie T \\
= & \quad \langle \text{ Distributivty of } \cap \text{ over } \cup \ \rangle \\
& (\bowtie T' \cap \bowtie T) \cup (\{dt \cup \{(s,v)\} \mid dt, v \,.\, dt \in \bowtie T' \wedge v \in s\} \cap \bowtie T) \\
= & \quad \langle \textbf{ IH } \rangle \\
& \bowtie T' \cup (\{dt \cup \{(s,v)\} \mid dt, v \,.\, dt \in \bowtie T' \wedge v \in s\} \cap \bowtie T) \\
= & \quad \langle \text{ Assumption } s \in T \ \rangle \\
& \bowtie T' \cup (\{dt \cup \{(s,v)\} \mid dt, v \,.\, dt \in \bowtie T' \wedge v \in s\} \cap \bowtie ((T \backslash \{s\}) \cup \{s\})) \\
= & \quad \langle \text{ Definition of } \bowtie \ (4.1) \ \rangle \\
& \bowtie T' \cup \big(\{dt \cup \{(s,v)\} \mid dt, v \,.\, dt \in \bowtie T' \wedge v \in s\} \\
& \qquad \cap (\bowtie (T \backslash \{s\}) \cup \{dt \cup \{(s,v)\} \mid dt, v \,.\, dt \in \bowtie (T \backslash \{s\}) \wedge v \in s\})\big) \\
= & \quad \langle \text{ Distributivty of } \cap \text{ over } \cup \ \rangle \\
& \bowtie T' \cup \big(\{dt \cup \{(s,v)\} \mid dt, v \,.\, dt \in \bowtie T' \wedge v \in s\} \cap \bowtie (T \backslash \{s\})\big) \\
& \qquad \cup \big(\{dt \cup \{(s,v)\} \mid dt, v \,.\, dt \in \bowtie T' \wedge v \in s\} \\
& \qquad \cap \{dt \cup \{(s,v)\} \mid dt, v \,.\, dt \in \bowtie (T \backslash \{s\}) \wedge v \in s\}\big) \\
= & \quad \langle \text{ Equations (A.1), (A.2) } \rangle \\
& \bowtie T' \cup \{\} \cup \{dt \cup \{(s,v)\} \mid dt, v \,.\, dt \in \bowtie T' \wedge v \in s\} \\
= & \quad \langle \text{ Annihilator for } \cup \text{ and Definition of } \bowtie \ (4.1) \ \rangle \\
& \bowtie (T' \cup \{s\})
\end{array}
$$

Proof for Proposition (4.3.3), for any $s \in \mathcal{U}, T \subseteq \mathcal{U}$, with assumption $s \in T$, is immediate from $s \in T \iff \{s\} \subseteq T$ and Proposition 4.3.2.

Proof for Proposition (4.3.4) is done by induction on the first argument, $T_1 \subseteq \mathcal{U}$ and any $T_2 \subseteq \mathcal{U}$.
Step 1: $T_1 = \{\}$

$$
\begin{array}{ll}
& \bowtie (T_1 \cap T_2) \\
= & \quad \langle \text{ Assumption } T_1 = \{\} \ \rangle \\
& \bowtie (\{\} \cap T_2) \\
= & \quad \langle \text{ Annihilator of } \cap \ \rangle \\
& \bowtie \{\} \\
= & \quad \langle \text{ Definition of } \bowtie \ (4.1) \ \rangle \\
& \{\} \\
= & \quad \langle \text{ Annihilator of } \cap \ \rangle \\
& \{\} \cap \bowtie T_2 \\
= & \quad \langle \text{ Definition of } \bowtie \ (4.1) \ \rangle \\
& \bowtie \{\} \cap \bowtie T_2 \\
= & \quad \langle \text{ Assumption } T_1 = \{\} \ \rangle
\end{array}
$$

$$
\begin{array}{ll}
& \bowtie T_1 \ \cap \ \bowtie T_2 \\
= & \quad \langle \text{ Definition of } \bowtie \ (4.1) \ \rangle \\
& \bowtie T_1 \ \cap \ \bowtie T_2
\end{array}
$$

Step 2: Induction hypothesis (**IH**): $T_1, T_2 \subseteq,\ \bowtie (T_1 \cap T_2) \ = \bowtie T_1 \cap \ \bowtie T_2$. We show that for any $s \in \mathcal{U},\ \bowtie \big((T_1 \cup \{s\}) \cap T_2\big) \ = \bowtie (T_1 \cup \{s\}) \cap \ \bowtie T_2$.
Case 1: $s \in T_1$ is trivial, as it reduces to the **IH**.
Case 2: $s \notin T_1 \ \wedge \ s \notin T_2$

$$
\begin{array}{ll}
& \textbf{LH}:\ \bowtie \big((T_1 \cup \{s\}) \cap T_2\big) \\
= & \quad \langle \text{ Distributivity of } \cap \text{ over } \cup \ \rangle \\
& \bowtie \big((T_1 \cap T_2) \cup (\{s\} \cup T_2)\big) \\
= & \quad \langle \text{ Assumption } s \notin T_2 \text{ and } a \notin A \iff \{a\} \cap A = \{\} \ \rangle \\
& \bowtie \big((T_1 \cap T_2) \cup \{\}\big) \\
= & \quad \langle \text{ Annihilator for } \cup \ \rangle \\
& \bowtie (T_1 \cap T_2) \\
= & \quad \langle \ \textbf{IH} \ \rangle \\
& \bowtie T_1 \cap \ \bowtie T_2
\end{array}
$$

$$
\begin{array}{ll}
& \textbf{RH}:\ \bowtie (T_1 \cup \{s\}) \cap \ \bowtie T_2 \\
= & \quad \langle \text{ Definition of } \bowtie \ (\ 4.1) \ \rangle \\
& \big(\ \bowtie T_1 \ \cup \big\{(dt \cup \{(s,v)\}) \ \mid \ dt, v \ . \ dt \in \bowtie T_1 \ \wedge \ v \in s\big\}\big) \cap \ \bowtie T_2 \\
= & \quad \langle \text{ Distributivity of } \cap \text{ over } \cup \ \rangle \\
& (\ \bowtie T_1 \cap \ \bowtie T_2 \ ) \cup \big(\big\{(dt \cup \{(s,v)\}) \ \mid \ dt, v \ . \ dt \in \bowtie T_1 \ \wedge \ v \in s\big\} \cap \ \bowtie T_2 \ \big) \\
= & \quad \langle \text{ Equation (A.1)} \ \rangle \\
& (\ \bowtie T_1 \cap \ \bowtie T_2 \ ) \cup \{\} \\
= & \quad \langle \text{ Annihilator for } \cup \ \rangle \\
& \bowtie T_1 \cap \ \bowtie T_2
\end{array}
$$

Case 3: $s \notin T_1 \ \wedge \ s \in T_2$ With these assumption, it is immediate that

$$T1 \cap (T_2 \backslash \{s\}) = T_1 \cap T_2 \tag{A.3}$$

$$\bowtie T_1 \cap \ \bowtie (T_2 \backslash \{s\}) \ = \bowtie T_1 \cap \ \bowtie T_2 \tag{A.4}$$

$$
\begin{array}{ll}
& \textbf{LH}:\ \bowtie \big((T_1 \cup \{s\}) \cap T_2\big) \\
= & \quad \langle \text{ Distributivity of } \cap \text{ over } \cup \ \rangle \\
& \bowtie \big((T_1 \cap T_2) \cup (\{s\} \cup T_2)\big) \\
= & \quad \langle \text{ Assumption } s \in T_2 \ \rangle \\
& \bowtie \big((T_1 \cap T_2) \cup \{s\}\big) \\
= & \quad \langle \text{ Definition of } \bowtie \ (4.1) \ \rangle
\end{array}
$$

$$\bowtie(T_1 \cap T_2) \ \cup \big\{(dt \cup \{(s,v)\}) \ \mid \ dt, v \ . \ dt \in \bowtie(T_1 \cap T_2) \ \wedge v \in s\big\}$$

RH: $\bowtie(T_1 \cup \{s\}) \cap \ \bowtie T_2$

$==$ $\langle$ Assumption $s \in T_2$ $\rangle$

$\bowtie(T_1 \cup \{s\}) \cap \ \bowtie\big((T_2 \backslash \{s\}) \cup \{s\}\big)$

$==$ $\langle$ Definition of $\bowtie$ (4.1) $\rangle$

$\big(\ \bowtie T_1 \ \cup \big\{(dt \cup \{(s,v)\}) \ \mid \ dt, v \ . \ dt \in \bowtie T_1 \ \wedge v \in s\big\}\big) \cap$
$\big(\ \bowtie(T_2 \backslash \{s\}) \ \cup \big\{(dt \cup \{(s,v)\}) \ \mid \ dt, v \ . \ dt \in \bowtie(T_2 \backslash \{s\}) \ \wedge v \in s\big\}\big)$

$==$ $\langle$ Distributivity of $\cap$ over $\cup$ $\rangle$

$\big(\ \bowtie T_1 \cap \ \bowtie(T_2 \backslash \{s\})\ \big) \cup$
$\big(\ \bowtie T_1 \ \cap \big\{(dt \cup \{(s,v)\}) \ \mid \ dt, v \ . \ dt \in \bowtie(T_2 \backslash \{s\}) \ \wedge v \in s\big\}\big) \cup$
$\big(\big\{(dt \cup \{(s,v)\}) \ \mid \ dt, v \ . \ dt \in \bowtie T_1 \ \wedge v \in s\big\} \cap \ \bowtie(T_2 \backslash \{s\})\ \big) \cup$
$\big(\big\{(dt \cup \{(s,v)\}) \ \mid \ dt, v \ . \ dt \in \bowtie T_1 \ \wedge v \in s\big\} \cap$
$\ \big\{(dt \cup \{(s,v)\}) \ \mid \ dt, v \ . \ dt \in \bowtie(T_2 \backslash \{s\}) \ \wedge v \in s\big\}\big)$

$==$ $\langle$ Equation (A.4) $\rangle$

$\big(\ \bowtie T_1 \cap \ \bowtie T_2\ \big) \cup$
$\big(\ \bowtie T_1 \ \cap \big\{(dt \cup \{(s,v)\}) \ \mid \ dt, v \ . \ dt \in \bowtie(T_2 \backslash \{s\}) \ \wedge v \in s\big\}\big) \cup$
$\big(\big\{(dt \cup \{(s,v)\}) \ \mid \ dt, v \ . \ dt \in \bowtie T_1 \ \wedge v \in s\big\} \cap \ \bowtie(T_2 \backslash \{s\})\ \big) \cup$
$\big(\big\{(dt \cup \{(s,v)\}) \ \mid \ dt, v \ . \ dt \in \bowtie T_1 \ \wedge v \in s\big\} \cap$
$\ \big\{(dt \cup \{(s,v)\}) \ \mid \ dt, v \ . \ dt \in \bowtie(T_2 \backslash \{s\}) \ \wedge v \in s\big\}\big)$

$==$ $\langle$ Assumption $s \notin T_1$ and Equation (A.1) $\rangle$

$\big(\ \bowtie T_1 \cap \ \bowtie T_2\ \big) \cup \{\} \cup$
$\big(\big\{(dt \cup \{(s,v)\}) \ \mid \ dt, v \ . \ dt \in \bowtie T_1 \ \wedge v \in s\big\} \cap \ \bowtie(T_2 \backslash \{s\})\ \big) \cup$
$\big(\big\{(dt \cup \{(s,v)\}) \ \mid \ dt, v \ . \ dt \in \bowtie T_1 \ \wedge v \in s\big\} \cap$
$\ \big\{(dt \cup \{(s,v)\}) \ \mid \ dt, v \ . \ dt \in \bowtie(T_2 \backslash \{s\}) \ \wedge v \in s\big\}\big)$

$==$ $\langle$ $s \notin (T_2 \backslash \{s\}$ and Equation (A.1) $\rangle$

$\big(\ \bowtie T_1 \cap \ \bowtie T_2\ \big) \cup \{\} \cup \{\} \cup$
$\big(\big\{(dt \cup \{(s,v)\}) \ \mid \ dt, v \ . \ dt \in \bowtie T_1 \ \wedge v \in s\big\} \cap$
$\ \big\{(dt \cup \{(s,v)\}) \ \mid \ dt, v \ . \ dt \in \bowtie(T_2 \backslash \{s\}) \ \wedge v \in s\big\}\big)$

$==$ $\langle$ Distributivity of $\cap$ over set comprehension $\rangle$

$\big(\ \bowtie T_1 \cap \ \bowtie T_2\big) \cup \{\} \cup \{\} \cup \big\{(dt \cup \{(s,v)\}) \ \mid \ dt, v \ . \ dt \in (\bowtie T_1 \cap \ \bowtie(T_2 \backslash \{s\})) \wedge v \in s\big\}$

$==$ $\langle$ Annihilator for $\cup$ $\rangle$

$\big(\ \bowtie T_1 \cap \ \bowtie T_2\ \big) \cup \big\{(dt \cup \{(s,v)\}) \ \mid \ dt, v \ . \ dt \in (\ \bowtie T_1 \cap \ \bowtie(T_2 \backslash \{s\})\ ) \wedge v \in s\big\}$

$==$ $\langle$ Equation (A.4) $\rangle$

$\big(\ \bowtie T_1 \cap \ \bowtie T_2\ \big) \cup \big\{(dt \cup \{(s,v)\}) \ \mid \ dt, v \ . \ dt \in (\ \bowtie T_1 \cap \ \bowtie T_2\ ) \wedge v \in s\big\}$

$==$ $\langle$ **IH** $\rangle$

$\bowtie(T_1 \cap T_2) \ \cup \big\{(dt \cup \{(s,v)\}) \ \mid \ dt, v \ . \ dt \in \bowtie(T_1 \cap T_2) \ \wedge v \in s\big\}$

Thus, **LH** = **RH** for all cases of the induction hypothesis step.

Proof for Proposition (4.3.5), for any $\kappa, \lambda \in$ and $v \in \kappa$. Case 1: $\kappa \neq \lambda$:

$$
\begin{array}{ll}
& \{(\kappa, v)\}\big\downarrow_{\lambda} \\
= & \quad \langle \text{ Definition of } \Big\downarrow \text{ (4.5) } \rangle \\
& \{(s, v) \mid s \,.\, (s, v) \in \{(\kappa, v)\} \wedge s \neq \lambda\} \\
= & \quad \langle\, x \in \{y\} \iff x = y, \text{ Variable replacing, } s = \kappa \,\rangle \\
& \{(\kappa, v) \mid \kappa \neq \lambda\} \\
= & \quad \langle \text{ Assumption } \kappa \neq \lambda \,\rangle \\
& \{(\kappa, v)\}
\end{array}
$$

Proof for Proposition (4.3.5), for any any $\kappa, \lambda \in$ and $v \in \kappa$. Case 2: $\kappa = \lambda$:

$$
\begin{array}{ll}
& \{(\kappa, v)\}\big\downarrow_{\lambda} \\
= & \quad \langle \text{ Definition of } \Big\downarrow \text{ (4.5) } \rangle \\
& \{(s, v) \mid s \,.\, (s, v) \in \{(\kappa, v)\} \wedge s \neq \lambda\} \\
= & \quad \langle\, x \in \{y\} \iff x = y, \text{ Variable replacing, } s = \kappa \,\rangle \\
& \{(\kappa, v) \mid \kappa \neq \lambda\} \\
= & \quad \langle \text{ Assumption } \kappa = \lambda, \text{ Contradiction, } P \wedge \neg P = False \,\rangle \\
& \{(\kappa, v) \mid False\} \\
= & \quad \langle\, \{x \mid False\} = \{\} \,\rangle \\
& \{\}
\end{array}
$$

Proof for Proposition (4.3.6), for any $\kappa \in \mathcal{U}$, $a \in A$, and $\forall dt \in a.\ \kappa \,\overline{\in}\, dt$:

$$
\begin{array}{ll}
& a\Big\downarrow_{\kappa} \cap a\Big\uparrow^{\kappa} \\
= & \quad \langle \text{ Definition of } \Big\downarrow \text{ (4.6) } \rangle \\
& \{dt\big\downarrow_{\kappa} \mid dt \,.\, dt \in a\} \cap a\Big\uparrow^{\kappa} \\
= & \quad \langle \text{ Definition of } \Big\uparrow \text{ (4.8), with } \kappa \in \mathcal{U},\ a \in A, \text{ and } \forall dt \in a.\ \kappa \,\overline{\in}\, dt \,\rangle \\
& \{dt\big\downarrow_{\kappa} \mid dt \,.\, dt \in a\} \cap \left\{\left(dt\big\downarrow_{\kappa} \cup \{(\kappa, v)\}\right) \mid dt, v \,.\, dt \in a \wedge v \in \kappa\right\} \\
= & \quad \langle \text{ Variable renaming, both sets } \rangle \\
& \{x \mid x, dt \,.\, x = dt\big\downarrow_{\kappa} \wedge dt \in a\} \cap \\
& \{x \mid x, dt, v \,.\, x = \left(dt\big\downarrow_{\kappa} \cup \{(\kappa, v)\}\right) \wedge dt \in a \wedge v \in \kappa\} \\
= & \quad \langle\, \{E \mid R_1 \wedge R_2\} \iff \{E \mid R_1\} \cap \{E \mid R_2\} \,\rangle \\
& \{x \mid x, dt, v \,.\, x = dt\big\downarrow_{\kappa} \wedge x = \left(dt\big\downarrow_{\kappa} \cup \{(\kappa, v)\}\right) \wedge dt \in a \wedge v \in \kappa\} \\
= & \quad \langle \text{ Contradiction, by Definition (4.5), } x = dt\big\downarrow_{\kappa} \text{ contains no } \kappa \text{ s-values } \rangle \\
& \{x \mid False\} \\
= & \quad \langle \text{ Set Comprehension } \rangle \\
& \{\}
\end{array}
$$

Proof for Proposition (4.3.7), Equation (1), for any $\kappa \in \mathcal{U}$ and any $dt_1, dt_2 \in SD^{\mathcal{M}}$:

$$
\begin{array}{ll}
& (dt_1 \cup dt_2)\big\downarrow_\kappa \\
= & \quad \langle \text{ Definition of } \big\downarrow \text{ (4.5) } \rangle \\
& \{(s,v) \mid s,v \,.\, (s,v) \in (dt_1 \cup dt_2) \wedge s \neq \kappa\} \\
= & \quad \langle \text{ Axiom, Union, } x \in X_1 \cup X_2 \iff x \in X_1 \vee x \in X_2 \rangle \\
& \{(s,v) \mid s,v \,.\, \big((s,v) \in dt_1 \vee (s,v) \in dt_2\big) \wedge s \neq \kappa\} \\
= & \quad \langle \text{ Distributivity of } \wedge \text{ over } \vee \rangle \\
& \{(s,v) \mid s,v \,.\, \big((s,v) \in dt_1 \wedge s \neq \kappa\big) \vee \big((s,v) \in dt_2 \wedge s \neq \kappa\big)\} \\
= & \quad \langle \{E \mid R_1 \vee R_2\} \iff \{E \mid R_1\} \cup \{E \mid R_2\} \rangle \\
& \{(s,v) \mid s,v \,.\, (s,v) \in dt_1 \wedge s \neq \kappa\} \cup \{(s,v) \mid s,v \,.\, (s,v) \in dt_2 \wedge s \neq \kappa\} \\
= & \quad \langle \text{ Definition of } \big\downarrow \text{ (4.5), twice } \rangle \\
& dt_1\big\downarrow_\kappa \cup dt_2\big\downarrow_\kappa
\end{array}
$$

Proof for Proposition (4.3.7), Equation (2), for any $\kappa \in \mathcal{U}$ and any $dt_1, dt_2 \in SD^{\mathcal{M}}$:

$$
\begin{array}{ll}
& (dt_1 \cap dt_2)\big\downarrow_\kappa \\
= & \quad \langle \text{ Definition of } \big\downarrow \text{ (4.5) } \rangle \\
& \{(s,v) \mid s,v \,.\, (s,v) \in (dt_1 \cap dt_2) \wedge s \neq \kappa\} \\
= & \quad \langle \text{ Axiom, Intersection, } x \in X_1 \cap X_2 \iff x \in X_1 \wedge x \in X_2 \rangle \\
& \{(s,v) \mid s,v \,.\, \big((s,v) \in dt_1 \wedge (s,v) \in dt_2\big) \wedge s \neq \kappa\} \\
= & \quad \langle \text{ Commutativity and Idempotency of } \wedge \text{ (for } s \neq \kappa) \rangle \\
& \{(s,v) \mid s,v \,.\, \big((s,v) \in dt_1 \wedge s \neq \kappa\big) \wedge \big((s,v) \in dt_2 \wedge s \neq \kappa\big)\} \\
= & \quad \langle \{E \mid R_1 \wedge R_2\} \iff \{E \mid R_1\} \cap \{E \mid R_2\} \rangle \\
& \{(s,v) \mid s,v \,.\, (s,v) \in dt_1 \wedge s \neq \kappa\} \cap \{s,v \mid (s,v) \,.\, (s,v) \in dt_2 \wedge s \neq \kappa\} \\
= & \quad \langle \text{ Definition of } \big\downarrow \text{ (4.5), twice } \rangle \\
& dt_1\big\downarrow_\kappa \cap dt_2\big\downarrow_\kappa
\end{array}
$$

Proof for Proposition (4.3.7), Equation (3), for any $\kappa \in \mathcal{U}$ and any $a, b \in A$:

$$
\begin{array}{ll}
& (a \cup b)\Big\Downarrow_\kappa \\
= & \quad \langle \text{ Definition of } \Big\Downarrow \text{ (4.6) } \rangle \\
& \{dt\big\downarrow_\kappa \mid dt \,.\, dt \in (a \cup b)\} \\
= & \quad \langle \text{ Axiom, Union, } x \in X_1 \vee x \in X_2 \iff x \in X_1 \cup X_2 \rangle \\
& \{dt\big\downarrow_\kappa \mid dt \,.\, dt \in a \vee dt \in b)\} \\
= & \quad \langle \{E \mid R_1 \vee R_2\} \iff \{E \mid R_1\} \cup \{E \mid R_2\} \rangle \\
& \{dt\big\downarrow_\kappa \mid dt \,.\, dt \in a\} \cup \{dt\big\downarrow_\kappa \mid dt \,.\, dt \in b\} \\
= & \quad \langle \text{ Definition of } \Big\Downarrow \text{ (4.6), twice } \rangle \\
& a\Big\Downarrow_\kappa \cup b\Big\Downarrow_\kappa
\end{array}
$$

Proof for Proposition (4.3.7), Equation (4), for any $\kappa \in \mathcal{U}$ and any $a, b \in A$ is done by showing that $(a \cap b)\Big\Downarrow_\kappa \cap (a\Big\Downarrow_\kappa \cap b\Big\Downarrow_\kappa) = (a \cap b)\Big\Downarrow_\kappa$

$$
\begin{array}{ll}
& (a \cap b)\Big\Downarrow_\kappa \cap (a\Big\Downarrow_\kappa \cap b\Big\Downarrow_\kappa) \\
== & \quad \langle\ A = A \cup (A \cap B)\ \rangle \\
& (a \cap b)\Big\Downarrow_\kappa \cap \big((a \cup (a \cap b))\Big\Downarrow_\kappa \cap (b \cup (a \cap b))\Big\Downarrow_\kappa\big) \\
== & \quad \langle\ \text{Associativity of } \cap\ \rangle \\
& (a \cap b)\Big\Downarrow_\kappa \cap (a \cup (a \cap b))\Big\Downarrow_\kappa \cap (b \cup (a \cap b))\Big\Downarrow_\kappa \\
== & \quad \langle\ \text{Proposition (4.3.7), Equation (3)}\ \rangle \\
& (a \cap b)\Big\Downarrow_\kappa \cap (a\Big\Downarrow_\kappa \cup (a \cap b)\Big\Downarrow_\kappa) \cap (b\Big\Downarrow_\kappa \cup (a \cap b)\Big\Downarrow_\kappa) \\
== & \quad \langle\ A \cap (A \cup B) = A, \text{ twice}\ \rangle \\
& (a \cap b)\Big\Downarrow_\kappa
\end{array}
$$

Thus, $(a \cap b)\Big\Downarrow_\kappa \subseteq (a\Big\Downarrow_\kappa \cap b\Big\Downarrow_\kappa)$.

Proof for Proposition (4.3.7), Equation (5), for any $\kappa \in \mathcal{U}$ and any $a, b \in A$:

$$
\begin{array}{ll}
& (a \cup b)\Big\Uparrow^\kappa \\
== & \quad \langle\ \text{Definition of } \Uparrow\ (4.8)\ \rangle \\
& \bigcup\{dt\big\uparrow^\kappa \mid dt\ .\ dt \in (a \cup b)\} \\
== & \quad \langle\ \text{Axiom, Union, } x \in X_1 \vee x \in X_2 \iff x \in X_1 \cup X_2\ \rangle \\
& \bigcup\{dt\big\uparrow^\kappa \mid dt\ .\ dt \in a \vee dt \in b\} \\
== & \quad \langle\ \{E \mid R_1 \vee R_2\} \iff \{E \mid R_1\} \cup \{E \mid R_2\}\ \rangle \\
& \bigcup\big(\{dt\big\uparrow^\kappa \mid dt\ .\ dt \in a\} \cup \{dt\big\uparrow^\kappa \mid dt\ .\ dt \in b\}\big) \\
== & \quad \langle\ \text{Union composition}\ \rangle \\
& \bigcup\{dt\big\uparrow^\kappa \mid dt\ .\ dt \in a\} \cup \bigcup\{dt\big\uparrow^\kappa \mid dt\ .\ dt \in b\} \\
== & \quad \langle\ \text{Definition of } \Uparrow\ (4.8), \text{ twice}\ \rangle \\
& a\Big\Uparrow^\kappa \cup b\Big\Uparrow^\kappa
\end{array}
$$

Proof for Proposition (4.3.7), Equation (6), for any $\kappa \in \mathcal{U}$ and any $a, b \in A$ is done by showing that $(a \cap b)\Big\Uparrow^\kappa \cap (a\Big\Uparrow^\kappa \cap b\Big\Uparrow^\kappa) = (a \cap b)\Big\Uparrow^\kappa$

$$
\begin{array}{ll}
& (a \cap b)\Big\Uparrow^\kappa \cap (a\Big\Uparrow^\kappa \cap b\Big\Uparrow^\kappa) \\
== & \quad \langle\ A = A \cup (A \cap B)\ \rangle \\
& (a \cap b)\Big\Uparrow^\kappa \cap ((a \cup (a \cap b))\Big\Uparrow^\kappa \cap (b \cup (a \cap b))\Big\Uparrow^\kappa) \\
== & \quad \langle\ \text{Associativity of } \cap\ \rangle \\
& (a \cap b)\Big\Uparrow^\kappa \cap ((a \cup (a \cap b))\Big\Uparrow^\kappa) \cap ((b \cup (a \cap b))\Big\Uparrow^\kappa) \\
== & \quad \langle\ \text{Proposition(4.3.7), Equation (5)}\ \rangle \\
& (a \cap b)\Big\Uparrow^\kappa \cap (a\Big\Uparrow^\kappa \cup (a \cap b)\Big\Uparrow^\kappa) \cap (b\Big\Uparrow^\kappa \cup (a \cap b)\Big\Uparrow^\kappa) \\
== & \quad \langle\ A \cap (A \cup B) = A, \text{ twice}\ \rangle \\
& (a \cap b)\Big\Uparrow^\kappa
\end{array}
$$

Thus, $(a \cap b)\Uparrow^{\kappa} \subseteq a\Uparrow^{\kappa} \cap b\Uparrow^{\kappa}$.

Proof for Proposition (4.3.8), Equation (1), for any $\kappa \in \mathcal{U}$ and any $dt \in SD^{\mathcal{M}}$:

$$
\begin{array}{ll}
& (dt\downarrow_{\kappa})\downarrow_{\kappa} \\
= & \quad \langle \text{ Definition of } \downarrow \text{ (4.5) } \rangle \\
& \{(s,v) \mid s,v \,.\, (s,v) \in dt\downarrow_{\kappa} \wedge s \neq \kappa \wedge v \in s\} \\
= & \quad \langle \text{ Definition of } \downarrow \text{ (4.5) } \rangle \\
& \{(s,v) \mid s,v \,.\, \big[(s,v) \in \{(s',v') \mid s',v' \,.\, (s',v') \in dt \wedge s' \neq \kappa \wedge v' \in s'\} \wedge \\
& \qquad\qquad s \neq \kappa \wedge v \in s\big]\} \\
= & \quad \langle\, x \in \{x \mid x \,.\, R\} \iff R \,\rangle \\
& \{(s,v) \mid s,v \,.\, (s,v) \in dt \wedge s \neq \kappa \wedge s \neq \kappa\} \\
= & \quad \langle \text{ Idempotency of } \wedge \rangle \\
& \{(s,v) \mid s,v \,.\, (s,v) \in dt \wedge s \neq \kappa\} \\
= & \quad \langle \text{ Definition of } \downarrow \text{ (4.5) } \rangle \\
& dt\downarrow_{\kappa}
\end{array}
$$

Proof for Proposition (4.3.8), Equation (2), for any $\kappa \in \mathcal{U}$ and $dt \in SD$
Case 1: $\kappa \,\overline{\notin}\, dt$

$$
\begin{array}{ll}
& (dt\uparrow^{\kappa})\Uparrow^{\kappa} \\
= & \quad \langle \text{ Definition of } \Uparrow \text{ (4.8) } \rangle \\
& \bigcup\{dt'\uparrow^{\kappa} \mid dt' \,.\, dt' \in dt\uparrow^{\kappa}\} \\
= & \quad \langle \text{ Definition of } \uparrow \text{ (4.7) } \rangle \\
& \bigcup\{dt'\uparrow^{\kappa} \mid dt' \,.\, dt' \in \{dt\}\} \\
= & \quad \langle \text{ Set comprehension } \{Ex \mid x \,.\, x \in \{y\}\} = \{Ey\} \,\rangle \\
& \bigcup\{dt\uparrow^{\kappa}\} \\
= & \quad \langle \text{ Union axiom } \bigcup\{A\} = A \,\rangle \\
& dt\uparrow^{\kappa}
\end{array}
$$

Case 2: $\kappa \,\overline{\in}\, dt$

$$
\begin{array}{ll}
& (dt\uparrow^{\kappa})\Uparrow^{\kappa} \\
= & \quad \langle \text{ Definition of } \Uparrow \text{ (4.8) } \rangle \\
& \bigcup\{dt'\uparrow^{\kappa} \mid dt' \,.\, dt' \in dt\uparrow^{\kappa}\} \\
= & \quad \langle \text{ Definition of } \uparrow \text{ (4.7) } \rangle \\
& \bigcup\left\{dt'\uparrow^{\kappa} \mid dt' \,.\, dt' \in \left\{\left(dt\downarrow_{\kappa} \cup \{(\kappa, v)\}\right) \mid v \,.\, v \in \kappa\right\}\right\} \\
= & \quad \langle \text{ Variable replacing: } x \in \{y\} \iff x = y \,\rangle \\
& \bigcup\left\{\left(dt\downarrow_{\kappa} \cup \{(\kappa, v)\}\right)\uparrow^{\kappa} \mid v \,.\, v \in \kappa\right\}
\end{array}
$$

$$
\begin{array}{ll}
= & \langle \text{ Definition of } \uparrow \text{ (4.7) } \rangle \\
& \bigcup \big\{ \{ (dt\big\downarrow_\kappa \cup \{(\kappa, v)\}) \big\downarrow_\kappa \cup \{(\kappa, v')\} \mid v' \,.\, v' \in \kappa \} \mid v \,.\, v \in \kappa \big\} \\
= & \langle \text{ Definition of } \downarrow \text{ (4.5) } \rangle \\
& \bigcup \big\{ \{ (dt\big\downarrow_\kappa) \big\downarrow_\kappa \cup \{(\kappa, v')\} \mid v' \,.\, v' \in \kappa \} \mid v \,.\, v \in \kappa \big\} \\
= & \langle \text{ Set comprehension, unbound variable } v \in \kappa \text{ and } \kappa \in \mathcal{U} \implies \kappa \neq \{\} \rangle \\
& \bigcup \big\{ \{ (dt\big\downarrow_\kappa) \big\downarrow_\kappa \cup \{(\kappa, v')\} \mid v' \,.\, v' \in \kappa \} \big\} \\
= & \langle \text{ Proposition (4.3.8), Equation (1) } \rangle \\
& \bigcup \big\{ \{ dt\big\downarrow_\kappa \cup \{(\kappa, v')\} \mid v' \,.\, v' \in \kappa \} \big\} \\
= & \langle \text{ Definition of } \uparrow \text{ (4.7) } \rangle \\
& \bigcup \{ dt\big\uparrow^\kappa \} \\
= & \langle \text{ Union axiom } \bigcup\{A\} = A \rangle \\
& dt\big\uparrow^\kappa
\end{array}
$$

Proof for Proposition (4.3.8), Equation (3), for any $\kappa \in \mathcal{U}$ and any $a \in A$:

$$
\begin{array}{ll}
& (a\Big\Downarrow_\kappa)\Big\Downarrow_\kappa \\
= & \langle \text{ Definition of } \Downarrow \text{ (4.6) } \rangle \\
& \{ dt\big\downarrow_\kappa \mid dt \,.\, dt \in a\Big\Downarrow_\kappa \} \\
= & \langle \text{ Definition of } \Downarrow \text{ (4.6) } \rangle \\
& \{ dt\big\downarrow_\kappa \mid dt \,.\, dt \in \{ dt'\big\downarrow_\kappa \mid dt' \,.\, dt' \in a \} \} \\
= & \langle \, x \in \{ x \mid x \,.\, R \} \iff R \, \rangle \\
& \{ dt\big\downarrow_\kappa \mid dt \,.\, dt = dt'\big\downarrow_\kappa \wedge dt' \in a \} \\
= & \langle \text{ Variable replacing, } dt = dt'\big\downarrow_\kappa \rangle \\
& \{ (dt'\big\downarrow_\kappa)\big\downarrow_\kappa \mid dt' \,.\, dt' \in a \} \\
= & \langle \text{ Idempotency of } \downarrow \text{ (4.3.9), (1) } \rangle \\
& \{ dt'\big\downarrow_\kappa \mid dt' \,.\, dt' \in a \} \\
= & \langle \text{ Definition of } \Downarrow \text{ (4.6) } \rangle \\
& a\Big\Downarrow_\kappa
\end{array}
$$

Proof for Proposition (4.3.8), Equation (4), for any $\kappa \in \mathcal{U}$ and any $a \in A$:

$$
\begin{array}{ll}
& (a\Big\Uparrow^\kappa)\Big\Uparrow^\kappa \\
= & \langle \text{ Definition of } \Uparrow \text{ (4.8) } \rangle \\
& \big(\bigcup \{ dt\big\uparrow^\kappa \mid dt \,.\, dt \in a \} \big)\Big\Uparrow^\kappa \\
= & \langle \text{ Proposition (4.3.7), Equation (5) } \rangle \\
& \bigcup \big(\{ dt\big\uparrow^\kappa \mid dt \,.\, dt \in a \} \big)\Big\Uparrow^\kappa \\
= & \langle \text{ Applying a function on a set } \rangle \\
& \bigcup \{ (dt\big\uparrow^\kappa)\Big\Uparrow^\kappa \mid dt \,.\, dt \in a \} \\
= & \langle \text{ Proposition (4.3.8), Equation (2) } \rangle
\end{array}
$$

$$
\begin{array}{ll}
& \bigcup\{dt\uparrow^{\kappa} \mid dt \cdot dt \in a\} \\
= & \quad \langle \text{ Definition of } \Uparrow \text{ (4.8) } \rangle \\
& a\Uparrow^{\kappa}
\end{array}
$$

Proof for Proposition (4.3.9), Equation (1), for any $\kappa, \lambda \in \mathcal{U}$ and $dt \in SD^{\mathcal{M}}$:

$$
\begin{array}{ll}
& (dt\downarrow_{\kappa})\downarrow_{\lambda} \\
= & \quad \langle \text{ Definition of } \downarrow \text{ (4.5), applied to } \lambda \rangle \\
& \{(s,v) \mid s,v \cdot (s,v) \in dt\downarrow_{\kappa} \wedge s \neq \lambda\} \\
= & \quad \langle \text{ Definition of } \downarrow \text{ (4.5), applied to } \kappa \rangle \\
& \{(s,v) \mid s,v \cdot (s,v) \in \{(s',v') \mid s',v' \cdot (s',v') \in dt \wedge s' \neq \kappa\} \wedge s \neq \lambda\} \\
= & \quad \langle\, x \in \{x \mid x \cdot R\} \iff R \,\rangle \\
& \{(s,v) \mid s,v \cdot (s,v) \in dt \wedge s \neq \kappa \wedge s \neq \lambda\} \\
= & \quad \langle \text{ Commutativity of } \wedge \rangle \\
& \{(s,v) \mid s,v \cdot (s,v) \in dt \wedge s \neq \lambda \wedge s \neq \kappa\} \\
= & \quad \langle\, x \in \{x \mid x \cdot R\} \iff R \,\rangle \\
& \{(s,v) \mid s,v \cdot (s,v) \in \{(s',v') \mid s',v' \cdot (s',v') \in dt \wedge s' \neq \lambda\} \wedge s \neq \kappa\} \\
= & \quad \langle \text{ Definition of } \downarrow \text{ (4.5), applied to } \lambda \rangle \\
& \{(s,v) \mid s,v \cdot (s,v) \in dt\downarrow_{\lambda} \wedge s \neq \kappa\} \\
= & \quad \langle \text{ Definition of } \downarrow \text{ (4.5), applied to } \kappa \rangle \\
& (dt\downarrow_{\lambda})\downarrow_{\kappa}
\end{array}
$$

Proof for Proposition (4.3.9), Equation (2), for any $\kappa, \lambda \in \mathcal{U}$ and $a \in A$:

$$
\begin{array}{ll}
& (a\Downarrow_{\kappa})\Downarrow_{\lambda} \\
= & \quad \langle \text{ Definition of } \Downarrow \text{ (4.6), applied to } \lambda \rangle \\
& \{dt\downarrow_{\lambda} \mid dt \cdot dt \in a\Downarrow_{\kappa}\} \\
= & \quad \langle \text{ Definition of } \Downarrow \text{ (4.6), applied to } \kappa \rangle \\
& \{dt\downarrow_{\lambda} \mid dt \cdot dt \in \{dt'\downarrow_{\kappa} \mid dt' \cdot dt' \in a\}\} \\
= & \quad \langle\, x \in \{x \mid x \cdot R\} \iff R, \text{ replace variable } dt \text{ with } dt'\downarrow_{\kappa} \,\rangle \\
& \{(dt'\downarrow_{\kappa})\downarrow_{\lambda} \mid dt' \cdot dt' \in a\} \\
= & \quad \langle \text{ Associativity of } \downarrow \text{ (4.3.9), (1) } \rangle \\
& \{(dt'\downarrow_{\lambda})\downarrow_{\kappa} \mid dt' \cdot dt' \in a\} \\
= & \quad \langle \text{ Variable renaming, } dt'\downarrow_{\lambda} = dt \rangle \\
& \{dt\downarrow_{\kappa} \mid dt \cdot dt = dt'\downarrow_{\lambda} \wedge dt' \in a\} \\
= & \quad \langle\, x \in \{x \mid x \cdot R\} \iff R \,\rangle \\
& \{dt\downarrow_{\kappa} \mid dt \cdot dt \in \{dt'\downarrow_{\lambda} \mid dt' \cdot dt' \in a\}\} \\
= & \quad \langle \text{ Definition of } \Downarrow \text{ (4.6), applied to } \lambda \rangle \\
& \{dt\downarrow_{\kappa} \mid dt \cdot dt \in a\Downarrow_{\lambda}\}
\end{array}
$$

$$\underset{(a\Downarrow_{\lambda})\Downarrow_{\kappa}}{=\!=} \quad \langle \text{ Definition of } \Downarrow \text{ (4.6), applied to } \kappa \rangle$$

Proof for Proposition (4.3.9), Equation (3), for $\kappa, \lambda \in \mathcal{U}, \kappa \neq \lambda$ and $a \in A$.
Case 1: $\kappa \neq \lambda$:

$$
\begin{array}{ll}
& (a\Uparrow^{\kappa})\Uparrow^{\lambda} \\
= & \quad \langle \text{ Definition of } \Uparrow \text{ (4.8), applied to } \lambda \rangle \\
& \left\{ \left(dt\big\downarrow_{\lambda} \cup \{(\lambda, v)\}\right) \mid dt, v \,.\, dt \in a\Uparrow^{\kappa} \wedge v \in \lambda \right\} \\
= & \quad \langle \text{ Definition of } \Uparrow \text{ (4.8), applied to } \kappa \rangle \\
& \left\{ \left(dt\big\downarrow_{\lambda} \cup \{(\lambda, v)\}\right) \mid dt, v \,.\, dt \in \left\{ \left(dt'\big\downarrow_{\kappa} \cup \{(\kappa, v')\}\right) \mid dt', v' \,.\, dt' \in a \wedge v' \in \kappa \right\} \wedge v \in \lambda \right\} \\
= & \quad \langle\, x \in \{x \mid x \,.\, R\} \iff R \,\rangle \\
& \left\{ \left(dt\big\downarrow_{\lambda} \cup \{(\lambda, v)\}\right) \mid dt, v, dt', v' \,.\, dt = \left(dt'\big\downarrow_{\kappa} \cup \{(\kappa, v')\}\right) \wedge dt' \in a \wedge v' \in \kappa \wedge v \in \lambda \right\} \\
= & \quad \langle \text{ Variable replacing, } dt = \left(dt'\big\downarrow_{\kappa} \cup \{(\kappa, v')\}\right) \rangle \\
& \left\{ \left(\left(dt'\big\downarrow_{\kappa} \cup \{(\kappa, v')\}\right)\big\downarrow_{\lambda} \cup \{(\lambda, v)\}\right) \mid v, dt', v' \,.\, dt' \in a \wedge v' \in \kappa \wedge v \in \lambda \right\} \\
= & \quad \langle \downarrow \text{ preserves } \cup \text{ (4.3.7), (1)} \rangle \\
& \left\{ \left[\left(dt'\big\downarrow_{\kappa}\right)\big\downarrow_{\lambda} \cup \{(\kappa, v')\}\big\downarrow_{\lambda} \cup \{(\lambda, v)\}\right] \mid dt', v, v' \,.\, dt' \in a \wedge v' \in \kappa \wedge v \in \lambda \right\} \\
= & \quad \langle \text{ Assumption } \kappa \neq \lambda, \text{ Proposition (4.3.5)} \rangle \\
& \left\{ \left[\left(dt'\big\downarrow_{\kappa}\right)\big\downarrow_{\lambda} \cup \{(\kappa, v')\} \cup \{(\lambda, v)\}\right] \mid dt', v, v' \,.\, dt' \in a \wedge v' \in \kappa \wedge v \in \lambda \right\} \\
= & \quad \langle \text{ Associativity of } \downarrow \text{ (4.3.9), 1} \rangle \\
& \left\{ \left[\left(dt'\big\downarrow_{\lambda}\right)\big\downarrow_{\kappa} \cup \{(\kappa, v')\} \cup \{(\lambda, v)\}\right] \mid dt', v, v' \,.\, dt' \in a \wedge v' \in \kappa \wedge v \in \lambda \right\} \\
= & \quad \langle \text{ Commutativity of } \cup \rangle \\
& \left\{ \left[\left(dt'\big\downarrow_{\lambda}\right)\big\downarrow_{\kappa} \cup \{(\lambda, v)\} \cup \{(\kappa, v')\}\right] \mid dt', v, v' \,.\, dt' \in a \wedge v' \in \kappa \wedge v \in \lambda \right\} \\
= & \quad \langle \text{ Assumption } \kappa \neq \lambda, \text{ Proposition (4.3.5)} \rangle \\
& \left\{ \left[\left(dt'\big\downarrow_{\lambda}\right)\big\downarrow_{\kappa} \cup \{(\lambda, v)\}\big\downarrow_{\kappa} \cup \{(\kappa, v')\}\right] \mid dt', v, v' \,.\, dt' \in a \wedge v' \in \kappa \wedge v \in \lambda \right\} \\
= & \quad \langle \downarrow \text{ preserves } \cup \text{ (4.3.7), (1)} \rangle \\
& \left\{ \left[\left(dt'\big\downarrow_{\lambda} \cup \{(\lambda, v)\}\right)\big\downarrow_{\kappa} \cup \{(\kappa, v')\}\right] \mid dt', v, v' \,.\, dt' \in a \wedge v' \in \kappa \wedge v \in \lambda \right\} \\
= & \quad \langle \text{ Variable replacing, } dt = \left(dt'\big\downarrow_{\lambda} \cup \{(\lambda, v)\}\right) \rangle \\
& \left\{ \left(dt\big\downarrow_{\kappa} \cup \{(\kappa, v')\}\right) \mid dt, dt', v, v' \,.\, dt = \left(dt'\big\downarrow_{\lambda} \cup \{(\lambda, v)\}\right) \wedge dt' \in a \wedge v' \in \kappa \wedge v \in \lambda \right\} \\
= & \quad \langle \text{ Commutativity of } \wedge \rangle \\
& \left\{ \left(dt\big\downarrow_{\kappa} \cup \{(\kappa, v')\}\right) \mid dt, dt', v, v' \,.\, dt = \left(dt'\big\downarrow_{\lambda} \cup \{(\lambda, v)\}\right) \wedge dt' \in a \wedge v \in \lambda \wedge v' \in \kappa \right\} \\
= & \quad \langle\, x \in \{x \mid x \,.\, R\} \iff R \,\rangle \\
& \left\{ \left(dt\big\downarrow_{\kappa} \cup \{(\kappa, v')\}\right) \mid dt, v' \,.\, dt \in \left\{ \left(dt'\big\downarrow_{\lambda} \cup \{(\lambda, v)\}\right) \mid dt', v \,.\, dt' \in a \wedge v \in \lambda \right\} \wedge v' \in \kappa \right\} \\
= & \quad \langle \text{ Definition of } \Uparrow \text{ (4.8), applied to } \lambda \rangle \\
& \left\{ \left(dt\big\downarrow_{\kappa} \cup \{(\kappa, v')\}\right) \mid dt, v' \,.\, dt \in a\Uparrow^{\lambda} \wedge v' \in \kappa \right\} \\
= & \quad \langle \text{ Definition of } \Uparrow \text{ (4.8), applied to } \kappa \rangle \\
& (a\Uparrow^{\lambda})\Uparrow^{\kappa}
\end{array}
$$

Proof for Proposition (4.3.9), Equation (3). Case 2, for $\kappa = \lambda$ is immediate, as both sides of the equation reduce to $a\Uparrow^{\kappa}$, due to associativity of $\Uparrow$ operator.

Proof for Proposition (4.3.10), Equation (1) , for any $\kappa \in \mathcal{U}$ and $a \in A$:

$$
\begin{array}{ll}
& \lfloor a \rfloor_\kappa \cup \lceil a \rceil^\kappa \\
= & \quad \langle \text{ Definitions for } \lfloor\rfloor \text{ and } \lceil\rceil \text{ operators (4.9), (4.10) } \rangle \\
& \{dt \mid dt \in a \wedge \neg \exists v \in \kappa.\ (\kappa, v) \in dt\} \cup \{dt \mid dt \in a \wedge \exists v \in \kappa.\ (\kappa, v) \in dt\} \\
= & \quad \langle\ \{E \mid R_1 \vee R_2\} \iff \{E \mid R_1\} \cup \{E \mid R_2\}\ \rangle \\
& \{dt \mid dt \in a \wedge \neg \exists v \in \kappa.\ (\kappa, v) \in dt \vee dt \in a \wedge \exists v \in \kappa.\ (\kappa, v) \in dt\} \\
= & \quad \langle \text{ Distributivity of } \cap \text{ over } \cup \ \rangle \\
& \{dt \mid dt \in a \wedge (\neg \exists v \in \kappa.\ (\kappa, v) \in dt \vee \exists v \in \kappa.\ (\kappa, v) \in dt)\} \\
= & \quad \langle \text{ Law of excluded middle, } P \vee \neg P \iff True \ \rangle \\
& \{dt \mid dt \in a \vee True\} \\
= & \quad \langle\ P \vee True \iff P\ \rangle \\
& \{dt \mid dt \in a\} \\
= & \quad \langle \text{ Extensionality of sets } \rangle \\
& a
\end{array}
$$

Proof for Proposition (4.3.10), Equation (2), for any $\kappa \in \mathcal{U}$ and $a \in A$:

$$
\begin{array}{ll}
& \lfloor a \rfloor_\kappa \cap \lceil a \rceil^\kappa \\
= & \quad \langle \text{ Definitions for } \lfloor\rfloor \text{ and } \lceil\rceil \text{ operators (4.9), (4.10) } \rangle \\
& \{dt \mid dt \in a \wedge \neg \exists v \in \kappa.\ (\kappa, v) \in dt\} \cap \{dt \mid dt \in a \wedge \exists v \in \kappa.\ (\kappa, v) \in dt\} \\
= & \quad \langle\ \{E \mid R_1 \wedge R_2\} \iff \{E \mid R_1\} \cap \{E \mid R_2\}\ \rangle \\
& \{dt \mid dt \in a \wedge \neg \exists v \in \kappa.\ (\kappa, v) \in dt \wedge dt \in a \wedge \exists v \in \kappa.\ (\kappa, v) \in dt\} \\
= & \quad \langle \text{ Contradiction, } P \wedge \neg P \iff False \ \rangle \\
& \{dt \mid dt \in a \wedge False\} \\
= & \quad \langle \text{ Zero of } \wedge \ \rangle \\
& \{dt \mid False\} \\
= & \quad \langle\ \{x \mid False\} = \{\}\ \rangle \\
& \{\}
\end{array}
$$

Proof for Proposition (4.3.11), we use a few helper equations. Based on Definitions (4.8) and (4.7), we have

$$a \Uparrow^\kappa = \{dt \mid dt \in a \wedge \neg(\kappa \mathrel{\overline{\in}} dt)\} \cup \left\{\left(dt\big\downarrow_\kappa \cup \{(\kappa, v)\}\right) \mid dt, v\ .\ dt \in a \wedge \kappa \mathrel{\overline{\in}} dt \wedge v \in \kappa\right\}$$

It is immediate that we can re-write the following:

$\left\{\left(dt\big\downarrow_\kappa \cup \{(\kappa, v)\}\right) \mid dt, v\ .\ dt \in a \wedge \kappa \mathrel{\overline{\in}} dt \wedge v \in \kappa\right\} = \lceil a \rceil^\kappa \cup X$, where

$X = \left\{\left(dt\big\downarrow_\kappa \cup \{(\kappa, v)\}\right) \mid dt, v\ .\ dt \in a \wedge \kappa \mathrel{\overline{\in}} dt \wedge v \in \kappa \wedge \left(dt\big\downarrow_\kappa \cup \{(\kappa, v)\}\right) \notin a\right\}$.

Thus, we have the following helper equations:

$$a\Uparrow^{\kappa} = \lfloor a \rfloor_{\kappa} \cup \lceil a \rceil^{\kappa} \cup X \tag{A.5}$$
$$X \cap \lfloor a \rfloor_{\kappa} = \{\} \tag{A.6}$$
$$X \cap \lceil a \rceil^{\kappa} = \{\} \tag{A.7}$$

Proof for Proposition (4.3.11), Equation (1), for any any $\kappa \in \mathcal{U}$ and $a \in A$:

$\lfloor a \rfloor_{\kappa} \cap a\Uparrow^{\kappa}$
$==$ $\langle$ Equation (A.5) $\rangle$
$\lfloor a \rfloor_{\kappa} \cap (\lfloor a \rfloor_{\kappa} \cup \lceil a \rceil^{\kappa} \cup X)$
$==$ $\langle$ Distributivity of $\cap$ over $\cup$ $\rangle$
$(\lfloor a \rfloor_{\kappa} \cap \lfloor a \rfloor_{\kappa}) \cup (\lfloor a \rfloor_{\kappa} \cap \lceil a \rceil^{\kappa}) \cup (\lfloor a \rfloor_{\kappa} \cap X)$
$==$ $\langle$ Idempotency of $\cap$, Proposition (4.3.10)(2), Equation (A.6) $\rangle$
$\lfloor a \rfloor_{\kappa}$

Proof for Proposition (4.3.11), Equation (2), for any any $\kappa \in \mathcal{U}$ and $a \in A$:

$\lceil a \rceil^{\kappa} \cap a\Uparrow^{\kappa}$
$==$ $\langle$ Equation (A.5) $\rangle$
$\lceil a \rceil^{\kappa} \cap (\lfloor a \rfloor_{\kappa} \cup \lceil a \rceil^{\kappa} \cup X)$
$==$ $\langle$ Distributivity of $\cap$ over $\cup$ $\rangle$
$(\lceil a \rceil^{\kappa} \cap \lfloor a \rfloor_{\kappa}) \cup (\lceil a \rceil^{\kappa} \cap \lceil a \rceil^{\kappa}) \cup (\lceil a \rceil^{\kappa} \cap X)$
$==$ $\langle$ Idempotency of $\cap$, Proposition (4.3.10)(2), Equation (A.7) $\rangle$
$\lceil a \rceil^{\kappa}$

Proof for Proposition (4.3.11), Equation (3), for any any $\kappa \in \mathcal{U}$ and $a \in A$:

$a \cap a\Uparrow^{\kappa}$
$==$ $\langle$ Proposition (4.3.10)(1) $\rangle$
$(\lfloor a \rfloor_{\kappa} \cup \lceil a \rceil^{\kappa}) \cap a\Uparrow^{\kappa}$
$==$ $\langle$ Distributivity of $\cap$ over $\cup$ $\rangle$
$(\lfloor a \rfloor_{\kappa} \cap a\Uparrow^{\kappa}) \cup (\lceil a \rceil^{\kappa} \cap a\Uparrow^{\kappa})$
$==$ $\langle$ Proposition (4.3.11), Equations (1), (2) $\rangle$
$\{\} \cup \lceil a \rceil^{\kappa}$
$==$ $\langle$ $\{\}$ neutral element for $\cup$ $\rangle$
$\lceil a \rceil^{\kappa}$

The proofs for Proposition (4.3.11), Equations (4), (5), and (6) are similar, as they are the duals of the above equations, and they have been omitted.
Proof for Proposition (4.3.12), Equation (1) for $\kappa \in \mathcal{U}$ and $a, b \in A$:

$$
\begin{array}{ll}
& \lceil a \rceil^{\kappa} \cap \lfloor b \rfloor_{\kappa} \\
== & \quad \langle \text{ Definition of } \lfloor \rfloor \ (4.9) \ \rangle \\
& \lceil a \rceil^{\kappa} \cap \{dt \mid dt \in b \wedge \neg \exists v \in \kappa.\ (\kappa, v) \in dt\} \\
== & \quad \langle \text{ Definition of } \lceil \rceil \ (4.10) \ \rangle \\
& \{dt \mid dt \in a \wedge \exists v \in \kappa.\ (\kappa, v) \in dt\} \cap \{dt \mid dt \in b \wedge \neg \exists v \in \kappa.\ (\kappa, v) \in dt\} \\
== & \quad \langle \ \{E \mid R_1 \wedge R_2\} \iff \{E \mid R_1\} \cap \{E \mid R_2\} \ \rangle \\
& \{dt \mid dt \in a \wedge \exists v \in \kappa.\ (\kappa, v) \in dt \wedge dt \in b \wedge \neg \exists v \in \kappa.\ (\kappa, v) \in dt\} \\
== & \quad \langle \text{ Contradiction } R \wedge \neg R = False \ \rangle \\
& \{dt \mid dt \in a \wedge dt \in b \wedge False\} \\
== & \quad \langle \text{ Zero of } \wedge \ \rangle \\
& \{dt \mid False\} \\
== & \quad \langle \ \{x \mid False\} = \{\} \ \rangle \\
& \{\}
\end{array}
$$

Proof for Proposition (4.3.12), Equation (2) for $\kappa \in \mathcal{U}$ and $a, b \in A$:

$$
\begin{array}{ll}
& \lceil a \rceil^{\kappa} \cap b\Big\Downarrow_{\kappa} \\
= & \quad \langle \text{ Definition of } \prod \text{ (4.10) } \rangle \\
& \{dt \mid dt \,.\, dt \in a \wedge \exists v \in \kappa.\ (\kappa, v) \in dt\} \cap b\Big\Downarrow_{\kappa} \\
= & \quad \langle \text{ Definition of } \Big\Downarrow \text{ (4.6) } \rangle \\
& \{dt \mid dt \,.\, dt \in a \wedge \exists v \in \kappa.\ (\kappa, v) \in dt\} \cap \{dt\big\downarrow_{\kappa} \mid dt \,.\, dt \in b\} \\
= & \quad \langle \text{ Variable renaming, } dt = dt'\big\downarrow_{\kappa} \rangle \\
& \{dt \mid dt \,.\, dt \in a \wedge \exists v \in \kappa.\ (\kappa, v) \in dt\} \cap \{dt \mid dt, dt' \,.\, dt = dt'\big\downarrow_{\kappa} \wedge dt' \in a\} \\
= & \quad \langle \text{ Definition of } \big\downarrow \text{ (4.5) } \rangle \\
& \{dt \mid dt \,.\, dt \in a \wedge \exists v \in \kappa.\ (\kappa, v) \in dt\} \cap \\
& \{dt \mid dt, dt' \,.\, dt = \{(s, v) \mid s, v \,.\, (s, v) \in dt' \wedge s \neq \kappa\} \wedge dt' \in a\} \\
= & \quad \langle \{E \mid R_1 \wedge R_2\} \Longleftrightarrow \{E \mid R_1\} \cap \{E \mid R_2\} \rangle \\
& \{dt \mid dt, dt' \,.\, dt \in a \wedge \exists v \in \kappa.\ (\kappa, v) \in dt \wedge \\
& \qquad dt = \{(s, v) \mid s, v \,.\, (s, v) \in dt' \wedge s \neq \kappa\} \wedge dt' \in a\} \\
= & \quad \langle \text{ Contradiction } dt = \{(s, v) \mid s, v \,.\, (s, v) \in dt' \wedge s \neq \kappa\} \text{ and} \\
& \quad \exists v \in \kappa.\ (\kappa, v) \in dt \rangle \\
& \{dt \mid dt, dt' \,.\, dt' \in a \wedge dt \in a \wedge False\} \\
= & \quad \langle \text{ Zero of } \wedge \rangle \\
& \{dt \mid False\} \\
= & \quad \langle \{x \mid False\} = \{\} \rangle \\
& \{\}
\end{array}
$$

A.2 Domain Data View Model

In this Section, we show that $\mathcal{M}_{\mathcal{A}} = \big(A, +, \star, -, \mathbf{0}, \mathbf{1}, \{\mathbf{c}_\kappa\}_{\kappa \in \mathcal{U}}\big)$ is a diagonal-free cylindric algebra. For that, we need to prove the following axioms withing $\mathcal{M}_{\mathcal{A}}$:

(C1) $\big(A, +, \star, -, \mathbf{0}, \mathbf{1}\big)$ is a Boolean algebra

(C2) $\mathbf{c}_\kappa \mathbf{0} = \mathbf{0}$

(C3) $a \leqslant \mathbf{c}_\kappa a$

(C4) $\mathbf{c}_\kappa(a \star \mathbf{c}_\kappa b) = \mathbf{c}_\kappa a \star \mathbf{c}_\kappa b$

(C5) $\mathbf{c}_\kappa \mathbf{c}_\lambda a = \mathbf{c}_\lambda \mathbf{c}_\kappa a$

For Axiom (C1), we need to show the following axioms hold in our model:

(C1.1) The binary operators $+, \star$ are associative, commutative, and distribute over each other

(C1.2) Identity axiom for $\mathbf{0}$: $a + \mathbf{0} = a$

(C1.3) Annihilator axiom for $\mathbf{0}$: $a \star \mathbf{0} = \mathbf{0}$

(C1.4) Identity axiom for $\mathbf{1}$: $a \star \mathbf{1} = a$

(C1.5) Annihilator axiom for $\mathbf{1}$: $a + \mathbf{1} = \mathbf{1}$

(C1.6) 1st Complement axiom: $a + (-a) = \mathbf{1}$

(C1.7) 2nd Complement axiom: $a \star (-a) = \mathbf{0}$

Due to the definitions of $+$ (4.11) and $\star$ (4.12), it is immediate that Axiom (C1.1) holds in this model. Both $\cup$ and $\cap$ are associative and commutative, and distribute over each other. Similarly, showing the Axioms (C1.2) and (C1.3) hold is trivial. This is done using the definitions for $\mathbf{0}$ (4.13), $+$ (4.11), $\star$ (4.12), and the facts that $\{\}$ is the neutral element for $\cup$ and annihilator for $\cap$.

Axiom (C1.4) holds, for any $a \in A$:

$$
\begin{array}{ll}
& a \star \mathbf{1} \\
= & \quad \langle \text{ Definition of } \star \text{ (4.12) } \rangle \\
& a \cap \mathbf{1} \\
= & \quad \langle \text{ Definition of } \mathbf{1} \text{ (4.14) } \rangle \\
& a \cap \bowtie \mathcal{U} \\
= & \quad \langle \ A \subseteq B \text{ iff } A \cap B = A, \text{ Equation (4.2) } \rangle \\
& a
\end{array}
$$

Axiom (C1.5) holds, for any $a \in A$:

$$
\begin{array}{ll}
& a + \mathbf{1} \\
= & \quad \langle \text{ Definition of } + \text{ (4.11) } \rangle \\
& a \cup \mathbf{1} \\
= & \quad \langle \text{ Definition of } \mathbf{1} \text{ (4.14) } \rangle \\
& a \cup \bowtie \mathcal{U} \\
= & \quad \langle \ A \subseteq B \text{ iff } A \cup B = B, \text{ Equation (4.2) } \rangle \\
& \bowtie \mathcal{U} \\
= & \quad \langle \text{ Definition of } \mathbf{1} \text{ (4.14) } \rangle \\
& \mathbf{1}
\end{array}
$$

Using the definitions of $+$ (4.11) and $-$ (4.15); and of $\star$ (4.12) and $-$ (4.15), the complement Axioms (C1.6) and (C1.7) are trivial to show they hold for $a \in A$. Thus, Axiom (C1) holds.

Axiom (C2) holds, for any $\kappa \in \mathcal{U}$:

$$
\begin{array}{ll}
& \mathbf{c}_\kappa \mathbf{0} \\
= & \quad \langle \text{ (Definition of } \mathbf{c}_\kappa \text{ 4.16) } \rangle \\
& \mathbf{0} \cup \{dt\Big\Uparrow^{\kappa} \mid dt \,.\, dt \in \mathbf{0}\} \\
= & \quad \langle \text{ Definition of } \mathbf{0} \text{ (4.13) } \rangle \\
& \mathbf{0} \cup \{dt\Big\Uparrow^{\kappa} \mid dt \,.\, dt \in \{\}\} \\
= & \quad \langle \ x \in \{\} \iff False \ \rangle \\
& \mathbf{0} \cup \{dt\Big\Uparrow^{\kappa} \mid False\} \\
= & \quad \langle \ \{x \mid False\} = \{\} \ \rangle \\
& \mathbf{0} \cup \{\} \\
= & \quad \langle \text{ Identity of } \cup) \ \rangle \\
& \mathbf{0}
\end{array}
$$

For Axiom (C3), we use the classic definition of $\leqslant$, and to show that $a \leqslant \mathbf{c}_\kappa a$ holds, we need to show that $a + \mathbf{c}_\kappa a = \mathbf{c}_\kappa a$ for $a \in A$:

$$
\begin{array}{ll}
& a + \mathbf{c}_\kappa a \\
\iff & \quad \langle \text{ Definition of } + \text{ (4.11) } \rangle \\
& a \cup \mathbf{c}_\kappa a \\
\iff & \quad \langle \text{ Definition of } \mathbf{c}_\kappa \text{ (4.16) } \rangle \\
& a \cup a \cup \{dt\Big\Uparrow^{\kappa} \mid dt \,.\, dt \in a\} \\
\iff & \quad \langle \text{ Idempotency of } \cup \ \rangle \\
& a \cup \{dt\Big\Uparrow^{\kappa} \mid dt \,.\, dt \in a\}
\end{array}
$$

$$
\begin{array}{ll}
\Longleftrightarrow & \langle \text{ Definition of } \mathbf{c}_\kappa \text{ (4.16) } \rangle \\
\quad \mathbf{c}_\kappa a &
\end{array}
$$

Axiom (C4) holds for any $a, b \in A$ and $\kappa \in \mathcal{U}$:

$$
\begin{array}{ll}
\quad \mathbf{c}_\kappa(a \star \mathbf{c}_\kappa b) & \\
= & \langle \text{ Definition of } \mathbf{c}_\kappa \text{ (4.16) } \rangle \\
\quad (a \star \mathbf{c}_\kappa b) \cup (a \star \mathbf{c}_\kappa b)\Uparrow^\kappa & \\
= & \langle \text{ Definition of } \star \text{ (4.12) } \rangle \\
\quad (a \cap \mathbf{c}_\kappa b) \cup (a \cap \mathbf{c}_\kappa b)\Uparrow^\kappa & \\
= & \langle \text{ Distributivity of } \Uparrow \text{ over } \cap \text{ (4.3.7), (5) } \rangle \\
\quad (a \cap \mathbf{c}_\kappa b) \cup (a\Uparrow^\kappa \cap (\mathbf{c}_\kappa b)\Uparrow^\kappa) & \\
= & \langle \text{ Definition of } \mathbf{c}_\kappa \text{ (4.16) } \rangle \\
\quad (a \cap (b \cup b\Uparrow^\kappa)) \cup (a\Uparrow^\kappa \cap (b \cup b\Uparrow^\kappa)\Uparrow^\kappa) & \\
= & \langle \text{ Distributivity of } \Uparrow \text{ over } \cap \text{ (4.3.7), (5) } \rangle \\
\quad (a \cap (b \cup b\Uparrow^\kappa)) \cup (a\Uparrow^\kappa \cap (b\Uparrow^\kappa \cup (b\Uparrow^\kappa)\Uparrow^\kappa)) & \\
= & \langle \text{ Idempotency of } \Uparrow \text{ (4.3.8), (4) } \rangle \\
\quad (a \cap (b \cup b\Uparrow^\kappa)) \cup (a\Uparrow^\kappa \cap (b\Uparrow^\kappa \cup b\Uparrow^\kappa)) & \\
= & \langle \text{ Idempotency of } \cup \rangle \\
\quad (a \cap (b \cup b\Uparrow^\kappa)) \cup (a\Uparrow^\kappa \cap b\Uparrow^\kappa) & \\
= & \langle\ b\Uparrow^\kappa = b \cup b\Uparrow^\kappa \text{, from Axiom (C3) } \rangle \\
\quad (a \cap (b \cup b\Uparrow^\kappa)) \cup (a\Uparrow^\kappa \cap (b \cup b\Uparrow^\kappa)) & \\
= & \langle \text{ Distributivity of } \cap \text{ over } \cup \rangle \\
\quad (a \cup a\Uparrow^\kappa) \cap (b \cup b\Uparrow^\kappa) & \\
= & \langle \text{ Definition of } \mathbf{c}_\kappa \text{ (4.16) } \rangle \\
\quad \mathbf{c}_\kappa a \cap \mathbf{c}_\kappa b & \\
= & \langle \text{ Definition of } \star \text{ (4.12) } \rangle \\
\quad \mathbf{c}_\kappa a \star \mathbf{c}_\kappa b &
\end{array}
$$

Axiom (C5) holds, for any $\kappa, \lambda \in \mathcal{U}$ and $a \in A$

$$
\begin{array}{ll}
\quad \mathbf{c}_\kappa \mathbf{c}_\lambda a & \\
= & \langle \text{ Definition of } \mathbf{c}_\kappa \text{ (4.16), applied to } \kappa \rangle \\
\quad \mathbf{c}_\lambda a \cup (\mathbf{c}_\lambda a)\Uparrow^\kappa & \\
= & \langle \text{ Definition of } \mathbf{c}_\kappa \text{ (4.16), applied to } \lambda \rangle \\
\quad (a \cup a\Uparrow^\lambda) \cup (a \cup a\Uparrow^\lambda)\Uparrow^\kappa & \\
= & \langle \text{ Distributivity of } \Uparrow \text{ over } \cup \text{ (4.3.7),(5) } \rangle \\
\quad a \cup a\Uparrow^\lambda \cup a\Uparrow^\kappa \cup (a\Uparrow^\lambda)\Uparrow^\kappa & \\
= & \langle \text{ Associativity of } \Uparrow \text{ (3), Commutativity of } \cup \rangle
\end{array}
$$

$$
\begin{array}{ll}
& a \cup a\Uparrow^{\kappa} \cup a\Uparrow^{\lambda} \cup (a\Uparrow^{\kappa})\Uparrow^{\lambda} \\
== & \quad \langle \text{ Distributivity of } \Uparrow \text{ over } \cup \text{ (4.3.7),(5) } \rangle \\
& (a \cup a\Uparrow^{\kappa}) \cup (a \cup a\Uparrow^{\kappa})\Uparrow^{\lambda} \\
== & \quad \langle \text{ Definition of } \mathbf{c}_{\kappa} \text{ (4.16), applied to } \kappa \rangle \\
& \mathbf{c}_{\kappa} a \cup (\mathbf{c}_{\kappa} a)\Uparrow^{\kappa} \\
== & \quad \langle \text{ Definition of } \mathbf{c}_{\kappa} \text{ (4.16), applied to } \lambda \rangle \\
& \mathbf{c}_{\lambda}\mathbf{c}_{\kappa} a
\end{array}
$$

Thus, the structure $\mathcal{M}_{\mathcal{A}} = \big(A, +, \star, -, \mathbf{0}, \mathbf{1}, \{\mathbf{c}_{\kappa}\}_{\kappa \in \mathcal{U}}\big)$ is a model for the DDV.

A.3 Domain Ontology Model

We remind the reader that the model for the DOnt component of a DIS consists of three mathematical structures: $\mathcal{M}_{\mathcal{C}} = \big(C_{At_{\mathcal{C}}}, \oplus, \mathbf{e}_{_C}\big)$, $\mathcal{M}_{\mathcal{L}} = \big(L_{At_{\mathcal{L}}}, \oplus, \otimes, \ominus, \mathbf{e}_{_C}, \top_{\mathcal{L}}\big)$, and $\mathcal{M}_{\mathcal{G}} = \{G_{t_i}\}_{t_i \in L, i \in I}$, with the model for DOnt given by $\mathcal{M}_{\mathcal{O}} = \big(\mathcal{M}_{\mathcal{C}}, \mathcal{M}_{\mathcal{L}}, \mathcal{M}_{\mathcal{G}}\big)$. In this section, we show that $\mathcal{M}_{\mathcal{C}}$ is a commutative idempotent monoid, and $\mathcal{M}_{\mathcal{L}}$ is a Boolean algebra.

To show that $\mathcal{M}_{\mathcal{C}}$ is a commutative idempotent monoid, we need to show that $\oplus$ is commutative and idempotent. This is immediate, from the definition of $\oplus$ (4.17), with $\cup$ a commutative, idempotent operator. In addition, the $\mathbf{e}_{_C}$ is the neutral element for $\oplus$, due to the definition of $\mathbf{e}_{_C}$ (4.18) and the fact that $\{\}$ is the neutral element for $\cup$. Thus, $\mathcal{M}_{\mathcal{C}}$ is a commutative idempotent monoid.

Similar to the proofs for the DDV model, showing that $\mathcal{M}_{\mathcal{L}}$ is a Boolean algebra is immediate. In addition, by construction, the graphs in $\mathcal{M}_{\mathcal{G}}$ are rooted in concepts of the Boolean algebra, thus they are rooted graphs. Therefore, the structure $\mathcal{M}_{\mathcal{O}}$ is a model for the DOnt component of the DIS theory.

A.4 Mapping operator

In this section, we show the properties of the type operator τ hold in the chosen model. For that, we show that the type operator preserves the zero and one between the two Boolean algebras of the DDV and DOnt (Axioms ((A15)), ((A16))), as well as the plus operators (Axiom ((A17))). In addition, we provide the proofs for the type operator results in Proposition (4.5.2).

Axiom (A15) holds for any $a \in A$:

$$
\begin{array}{ll}
& \tau(\mathbf{0}) \\
= & \quad \langle \text{ Definition of } \tau \text{ (4.23) } \rangle \\
& \{\eta(sv.sort) \mid sv, dt \,.\, sv \in dt \wedge dt \in \mathbf{0}\} \\
= & \quad \langle \text{ Definition of } \mathbf{0} \text{ (4.13) } \rangle \\
& \{\eta(sv.sort) \mid sv, dt \,.\, sv \in dt \wedge dt \in \{\}\} \\
= & \quad \langle\ x \in \{\} \iff False\ \rangle \\
& \{\eta(sv.sort) \mid False\} \\
= & \quad \langle\ \{x \mid False\} = \{\}\ \rangle \\
& \{\} \\
= & \quad \langle \text{ Definition of } \mathbf{e}_C \text{ (4.18) } \rangle \\
& \mathbf{e}_C
\end{array}
$$

Axiom (A16) holds for any $a \in A$:

$$
\begin{array}{ll}
& \tau(\mathbf{1}) \\
= & \quad \langle \text{ Definition of } \tau \text{ (4.23) } \rangle \\
& \{\eta(sv.sort) \mid sv, dt \,.\, sv \in dt \wedge dt \in \mathbf{1}\} \\
= & \quad \langle \text{ Definition of } \mathbf{1} \text{ (4.14) } \rangle \\
& \{\eta(sv.sort) \mid sv, dt \,.\, sv \in dt \wedge dt \in \bowtie \mathcal{U}\ \} \\
= & \quad \langle \text{ Here I am lost completely on how to approach it } \rangle \\
& \{\eta(sv.sort) \mid sv \,.\, sv.sort \in \mathcal{U}\} \\
= & \quad \langle \text{ By construction of } A \rangle \\
& \top_{\mathcal{L}}
\end{array}
$$

Axiom (A17) holds for any $a, b \in A$:

$$
\begin{array}{ll}
& \tau(a + b) \\
= & \quad \langle \text{ Definition of } \tau \text{ (4.23) } \rangle \\
& \{\eta(sv.sort) \mid sv, dt \,.\, sv \in dt \wedge dt \in (a + b)\} \\
= & \quad \langle \text{ Definition of } + \text{ (4.11) } \rangle \\
& \{\eta(sv.sort) \mid sv, dt \,.\, sv \in dt \wedge dt \in (a \cup b)\} \\
= & \quad \langle\ \{E \mid R_1 \vee R_2\} \iff \{E \mid R_1\} \cup \{E \mid R_2\}\ \rangle \\
& \{\eta(sv.sort) \mid sv, dt \,.\, sv \in dt \wedge (dt \in a \vee dt \in b)\} \\
= & \quad \langle \text{ Distributivity of } \wedge \text{ over } \vee \rangle \\
& \{\eta(sv.sort) \mid sv, dt \,.\, (sv \in dt \wedge dt \in a) \vee ((s, v) \in dt \wedge dt \in b)\} \\
= & \quad \langle\ \{E \mid R_1 \vee R_2\} \iff \{E \mid R_1\} \cup \{E \mid R_2\}\ \rangle \\
& \{\eta(sv.sort) \mid sv, dt \,.\, sv \in dt \wedge dt \in a\} \cup \{\eta(sv.sort) \mid sv, dt \,.\, sv \in dt \wedge dt \in b\} \\
= & \quad \langle \text{ Definition of } \tau \text{ (4.23) } \rangle \\
& \tau(a) \cup \tau(b) \\
= & \quad \langle \text{ Definition of } \oplus \text{ (4.17) } \rangle
\end{array}
$$

$\tau(a) \oplus \tau(b)$

Proof for Proposition (4.5.1), Equation (1), for $\kappa \in \mathcal{U}$ and $a \in A$:

$$
\begin{array}{ll}
& \eta(\kappa) \in \tau(a\Uparrow^{\kappa}) \\
\Longleftrightarrow & \quad \langle \text{ Definition of } \Uparrow \ (4.8) \ \rangle \\
& \eta(\kappa) \in \tau(\{ (dt\downarrow_{\kappa} \cup \{(\kappa, v)\}) \mid dt, v \ . \ dt \in a \wedge v \in \kappa \}) \\
\Longleftrightarrow & \quad \langle \text{ Set comprehension } \rangle \\
& \eta(\kappa) \in \tau\Big(\bigcup_{dt \in a} \bigcup_{v \in \kappa} (dt\downarrow_{\kappa} \cup \{(\kappa, v)\}) \Big) \\
\Longleftrightarrow & \quad \langle \text{ Distrbutivity of } \tau \text{ over } \cup \ \rangle \\
& \eta(\kappa) \in \bigcup_{dt \in a} \bigcup_{v \in \kappa} \tau\Big((dt\downarrow_{\kappa} \cup \{(\kappa, v)\}) \Big) \\
\Longleftrightarrow & \quad \langle \text{ Distrbutivity of } \tau \text{ over } \cup \ \rangle \\
& \eta(\kappa) \in \bigcup_{dt \in a} \bigcup_{v \in \kappa} [\tau(dt\downarrow_{\kappa}) \cup \tau(\{(\kappa, v)\})] \\
\Longleftrightarrow & \quad \langle \text{ Definition of } \tau \ (4.23) \ \rangle \\
& \eta(\kappa) \in \bigcup_{dt \in a} \bigcup_{v \in \kappa} [\tau(dt\downarrow_{\kappa}) \cup \{\eta(\kappa)\}] \\
\Longleftrightarrow & \quad \langle \text{ Set membership } \rangle \\
& True
\end{array}
$$

Proof for Proposition (4.5.1), Equation (2), for $\kappa \in \mathcal{U}$ and $a \in A$:

$$
\begin{array}{ll}
& \tau(a\Big\Downarrow_{\kappa}) \\
= & \quad \langle \text{ Definition of } \tau \text{ (4.23) } \rangle \\
& \{\eta(sv.sort) \mid sv, dt \,.\, sv \in dt \wedge dt \in a\Big\Downarrow_{\kappa}\} \\
= & \quad \langle \text{ Definition of } \Big\Downarrow \text{ (4.6) } \rangle \\
& \{\eta(sv.sort) \mid sv, dt \,.\, sv \in dt\Big\downarrow_{\kappa} \wedge dt \in a\} \\
= & \quad \langle \text{ Definition of } \downarrow \text{(4.5) } \rangle \\
& \{\eta(sv.sort) \mid sv, dt \,.\, sv \in dt\Big\downarrow_{\kappa} \wedge dt \in a \wedge sv.sort \neq \kappa\}
\end{array}
$$

Thus, $\eta(\kappa)$ cannot be a part of $\tau(a\Big\Downarrow_{\kappa})$.

For Proposition (4.5.2), we make the following notations, for any $a, b \in A$

$$a_{a\backslash b} = \{dt \mid dt \in a \wedge dt \notin b\} \tag{A.8}$$
$$a_{ab} = \{dt \mid dt \in a \wedge dt \in b\} \tag{A.9}$$

It is immediate that

$$a = a_{a\backslash b} \cup a_{ab} \tag{A.10}$$
$$a_{a\backslash b} \cap a_{ab} = \{\} \tag{A.11}$$
$$a_{a\backslash b} \cap b_{b\backslash a} = \{\} \tag{A.12}$$
$$a_{ab} = b_{ba} \tag{A.13}$$

We show that, for any $a, b \in A$, the following equation holds:

$$a \star b = a_{ab} \tag{A.14}$$

$a \star b$
$==$ $\langle$ Definition of $\otimes$ (4.20) $\rangle$
$a \cap b$
$==$ $\langle$ Notation (A.10), applied twice $\rangle$
$(a_{a\backslash b} \cup a_{ab}) \cap (b_{b\backslash a} \cup b_{ba})$
$==$ $\langle$ Distributivity of $\cap$, $\cup$ over each other $\rangle$
$(a_{a\backslash b} \cap b_{b\backslash a}) \cup (a_{a\backslash b} \cap b_{ba}) \cup (a_{ab} \cap b_{b\backslash a}) \cup (a_{ab} \cap b_{ba})$
$==$ $\langle$ Equation (A.13), three times $\rangle$
$(a_{a\backslash b} \cap b_{b\backslash a}) \cup (a_{a\backslash b} \cap a_{ab}) \cup (b_{ba} \cap b_{b\backslash a}) \cup (a_{ab} \cap a_{ab})$
$==$ $\langle$ Equations (A.12), (A.11) $\rangle$
$\{\} \cup \{\} \cup \{\} \cup (a_{ab} \cap a_{ab})$
$==$ $\langle$ Zero of $\cup$ $\rangle$
$a_{ab} \cap a_{ab}$
$==$ $\langle$ Idempotency of $\cap$ $\rangle$
a_{ab}

We show the following equation holds, for any $a, b \in A$:

$$\tau(a) \otimes \tau(b) = \big(\tau(a_{a\backslash b}) \cap \tau(b_{b\backslash a})\big) \cup \big(\tau(a_{a\backslash b}) \cap \tau(a_{ab})\big) \cup \big(\tau(a_{ab}) \cap \tau(b_{b\backslash a})\big) \cup \tau(a_{ab}) \tag{A.15}$$

$\tau(a) \otimes \tau(b)$
$==$ $\langle$ Equation (A.10) $\rangle$
$\tau(a_{a\backslash b} \cup a_{ab}) \otimes \tau(b_{b\backslash a} \cup b_{ba})$
$==$ $\langle$ Definition of $+$ (4.11) $\rangle$

$$
\begin{array}{ll}
& \tau(a_{a\backslash b} + a_{ab}) \otimes \tau(b_{b\backslash a} + b_{ba}) \\
= & \quad \langle\ \tau \text{ preserves } + \ ((A17))\ \rangle \\
& \big(\tau(a_{a\backslash b}) \oplus \tau(a_{ab})\big) \otimes \big(\tau(b_{b\backslash a}) \oplus \tau(b_{ba})\big) \\
= & \quad \langle\ \oplus, \otimes \text{ distribute over each other}\ \rangle \\
& \big(\tau(a_{a\backslash b}) \otimes \tau(b_{b\backslash a})\big) \oplus \big(\tau(a_{a\backslash b}) \otimes \tau(b_{ba})\big) \oplus \big(\tau(a_{ab}) \otimes \tau(b_{b\backslash a})\big) \oplus \big(\tau(a_{ab}) \otimes \tau(b_{ba})\big) \\
= & \quad \langle\ \text{Equation } (A.13)\ \rangle \\
& \big(\tau(a_{a\backslash b}) \otimes \tau(b_{b\backslash a})\big) \oplus \big(\tau(a_{a\backslash b}) \otimes \tau(b_{ba})\big) \oplus \big(\tau(a_{ab}) \otimes \tau(b_{b\backslash a})\big) \oplus \big(\tau(a_{ab}) \otimes \tau(b_{ba})\big) \\
= & \quad \langle\ \text{Definitions of } \oplus\ (4.17),\ \cap\ (4.20)\ \rangle \\
& \big(\tau(a_{a\backslash b}) \cap \tau(b_{b\backslash a})\big) \cup \big(\tau(a_{a\backslash b}) \cap \tau(b_{ba})\big) \cup \big(\tau(a_{ab}) \cap \tau(b_{b\backslash a})\big) \cup \big(\tau(a_{ab}) \cap \tau(b_{ba})\big) \\
= & \quad \langle\ \text{Equation } (A.13)\ \rangle \\
& \big(\tau(a_{a\backslash b}) \cap \tau(b_{b\backslash a})\big) \cup \big(\tau(a_{a\backslash b}) \cap \tau(b_{ba})\big) \cup \big(\tau(a_{ab}) \cap \tau(b_{b\backslash a})\big) \cup \big(\tau(a_{ab}) \cap \tau(b_{ba})\big) \\
= & \quad \langle\ \text{Idempotency of } \cap\ \rangle \\
& \big(\tau(a_{a\backslash b}) \cap \tau(b_{b\backslash a})\big) \cup \big(\tau(a_{a\backslash b}) \cap \tau(a_{ab})\big) \cup \big(\tau(a_{ab}) \cap \tau(b_{b\backslash a})\big) \cup \tau(a_{ab})
\end{array}
$$

Proof for Proposition (4.5.2), Equation (2), for any $a, b \in A$:

$$
\begin{array}{ll}
& \tau(a) \otimes \tau(b) = \mathbf{e}_C \\
\Longleftrightarrow & \quad \langle\ \text{Equation } (A.15)\ \rangle \\
& \big(\tau(a_{a\backslash b}) \cap \tau(b_{b\backslash a})\big) \cup \big(\tau(a_{a\backslash b}) \cap \tau(a_{ab})\big) \cup \big(\tau(a_{ab}) \cap \tau(b_{b\backslash a})\big) \cup \tau(a_{ab}) = \mathbf{e}_C \\
\Longleftrightarrow & \quad \langle\ \text{Definition of } \mathbf{e}_C\ (4.18)\ \rangle \\
& \big(\tau(a_{a\backslash b}) \cap \tau(b_{b\backslash a})\big) \cup \big(\tau(a_{a\backslash b}) \cap \tau(a_{ab})\big) \cup \big(\tau(a_{ab}) \cap \tau(b_{b\backslash a})\big) \cup \tau(a_{ab}) = \{\} \\
\Longrightarrow & \quad \langle\ S \cup T = \{\} \implies S = \{\}, \text{ with } S = \tau(a_{ab})\ \rangle \\
& \tau(a_{ab}) = \{\} \\
\Longleftrightarrow & \quad \langle\ \text{Equation } (A.14)\ \rangle \\
& \tau(a \otimes b) = \{\} \\
\Longleftrightarrow & \quad \langle\ \text{Definition of } \mathbf{e}_C\ (4.18)\ \rangle \\
& \tau(a \otimes b) = \mathbf{e}_C
\end{array}
$$

Proof for Proposition (4.5.2), Equation (3), for any $a, b \in A$, using $S \subseteq T \iff S \cap T = S$, for $S = \tau(a \star b)$ and $T = \tau(a) \otimes \tau(b)$:

$$
\begin{array}{ll}
& \tau(a \star b) \cap \big(\tau(a) \otimes \tau(b)\big) \\
== & \quad \langle \text{ Equation (A.14) } \rangle \\
& \tau(a_{ab}) \cap \big(\tau(a) \otimes \tau(b)\big) \\
== & \quad \langle \text{ Equation (A.15) } \rangle \\
& \tau(a_{ab}) \cap \big[\big(\tau(a_{a\backslash b}) \cap \tau(b_{b\backslash a})\big) \cup \big(\tau(a_{a\backslash b}) \cap \tau(a_{ab})\big) \cup \big(\tau(a_{ab}) \cap \tau(b_{b\backslash a})\big) \cup \tau(a_{ab})\big] \\
== & \quad \langle \ S \cap (S \cup T) = S, \text{ with } S = \tau(a_{ab}),\ T = \big(\tau(a_{a\backslash b}) \cap \tau(b_{b\backslash a})\big) \cup \big(\tau(a_{a\backslash b}) \cap \\
& \qquad \tau(a_{ab})\big) \cup \big(\tau(a_{ab}) \cap \tau(b_{b\backslash a})\big) \ \rangle \\
& \tau(a_{ab}) \\
== & \quad \langle \text{ Equation (A.14) } \rangle \\
& \tau(a \star b)
\end{array}
$$

Proof for Proposition (4.5.2), Equation (4) is done by induction on $T \subseteq \mathcal{U}$, for $a \in A$.
Step 1: $T = \{\}$

$$
\begin{array}{ll}
& a = \bowtie T \\
\iff & \quad \langle \text{ Assumption } T = \{\} \ \rangle \\
& a = \bowtie(\{\}) \\
\iff & \quad \langle \text{ Definition of } \bowtie \ (4.1) \ \rangle \\
& a = \{\} \\
\implies & \quad \langle \text{ Application of operator } \tau \ \rangle \\
& \tau(a) = \tau(\{\}) \\
\iff & \quad \langle \text{ Equation ((A15)) } \rangle \\
& \tau(a) = \{\} \\
\iff & \quad \langle \text{ Set comprehension } \rangle \\
& \tau(a) = \{\eta(s) \mid s \,.\, s \in \{\}\} \\
\iff & \quad \langle \text{ Assumption } T = \{\} \ \rangle \\
& \tau(a) = \{\eta(s) \mid s \,.\, s \in T\}
\end{array}
$$

Step 2: $T = \{s\}$, with $s \in \mathcal{U}$,

$$
\begin{array}{ll}
& a = \bowtie T \\
\iff & \quad \langle \text{ Assumption } T = \{s\} \ \rangle \\
& a = \bowtie\{s\} \\
\iff & \quad \langle \text{ Definition of } \bowtie \ (4.1) \ \rangle \\
& a = \big\{\{(s, v)\} \mid v \,.\, v \in s\big\} \\
\implies & \quad \langle \text{ Application of operator } \tau \ \rangle \\
& \tau(a) = \tau(\big\{\{(s, v)\} \mid v \,.\, v \in s\big\})
\end{array}
$$

$$
\begin{array}{ll}
\iff & \langle \text{ Definition of } \tau \text{ (4.23) } \rangle \\
\quad \tau(a) = \{\eta(s) \mid s \,.\, s \in T\}
\end{array}
$$

Step 3: Induction hypothesis $a = \bowtie T \implies \tau(a) = \{\eta(s) \mid s \in T\}, T \subseteq \mathcal{U}, a \in A$. We show that for any $s' \in \mathcal{U}, s' \notin T, a = \bowtie (T \cup \{s'\}) \implies \tau(a) = \{\eta(s) \mid s \in (T \cup \{s'\})\}$.

$$
\begin{array}{ll}
\quad a = \bowtie (T \cup \{s'\}) & \\
\iff & \langle \text{ Definition of } \bowtie \text{ (4.1) } \rangle \\
\quad a = \bowtie T \cup \{dt \cup \{(s', v)\} \mid dt, v \,.\, dt \in \bowtie T \wedge v \in s'\} & \\
\implies & \langle \text{ Application of operator } \tau \rangle \\
\quad \tau(a) = \tau\Big(\bowtie T \cup \{dt \cup \{(s', v)\} \mid dt, v \,.\, dt \in \bowtie T \wedge v \in s'\}\Big) & \\
\iff & \langle \text{ Definition of } \oplus \text{ (4.17), } \tau \text{ operator preserves } \oplus \text{ ((A17)) } \rangle \\
\quad \tau(a) = \tau(\bowtie T) \cup \tau(\{dt \cup \{(s', v)\} \mid dt, v \,.\, dt \in \bowtie T \wedge v \in s'\}) & \\
\iff & \langle \text{ Inductive hypothesis } \rangle \\
\quad \tau(a) = \{\eta(s) \mid s \,.\, s \in T\} \cup \tau(\{dt \cup \{(s', v)\} \mid dt, v \,.\, dt \in \bowtie T \wedge v \in s'\}) & \\
\iff & \langle \text{ Definition of } \tau \text{ (4.23) } \rangle \\
\quad \tau(a) = \{\eta(s) \mid s \,.\, s \in T\} \cup & \\
\qquad \{\eta(s) \mid s, v \,.\, (s, v) \in (dt \cup \{(s', v)\}) \wedge v \in s \wedge dt \in \bowtie T \wedge v \in s'\} & \\
\iff & \langle \; x \in X_1 \cup X_2 \iff x \in X_1 \vee x \in X_2 \; \rangle \\
\quad \tau(a) = \{\eta(s) \mid s \,.\, s \in T\} \cup & \\
\qquad \{\eta(s) \mid s, v \,.\, ((s, v) \in dt \vee (s, v) \in \{(s', v)\}) \wedge v \in s \wedge dt \in \bowtie T \wedge v \in s'\} & \\
\iff & \langle \text{ Distributivity of } \wedge, \vee \text{ over each other } \rangle \\
\quad \tau(a) = \{\eta(s) \mid s \,.\, s \in T\} \cup & \\
\qquad \{\eta(s) \mid s \,.\, ((s, v) \in dt \wedge v \in s \wedge dt \in \bowtie T \wedge v \in s') \vee & \\
\qquad\qquad ((s, v) \in \{(s', v)\} \wedge v \in s \wedge dt \in \bowtie T \wedge v \in s')\} & \\
\implies & \langle \; (s, v) \in dt \wedge dt \in \bowtie T \implies s \in T \; \rangle \\
\quad \tau(a) = \{\eta(s) \mid s \,.\, s \in T\} \cup & \\
\qquad \{\eta(s) \mid s \,.\, (s \in T \vee ((s, v) \in \{(s', v)\} \wedge v \in s)\} & \\
\implies & \langle \; (s, v) \in \{(s', v)\} \implies s \in \{s'\} \; \rangle \\
\quad \tau(a) = \{\eta(s) \mid s \,.\, s \in T\} \cup \{\eta(s) \mid s \,.\, s \in T \vee s \in \{s'\}\} & \\
\iff & \langle \; \{E \mid R_1 \vee R_2\} \iff \{E \mid R_1\} \cup \{E \mid R_2\} \; \rangle \\
\quad \tau(a) = \{\eta(s) \mid s \,.\, s \in T \vee s \in T \vee s \in \{s'\}\} & \\
\iff & \langle \; x \in X_1 \cup X_2 \iff x \in X_1 \vee x \in X_2 \; \rangle \\
\quad \tau(a) = \{\eta(s) \mid s \,.\, s \in (T \cup T \cup s \in \{s'\}\} & \\
\iff & \langle \text{ Idempotency of } \cup \; \rangle \\
\quad \tau(a) = \{\eta(s) \mid s \,.\, s \in (T \cup s \in \{s'\})\} &
\end{array}
$$

Proof for Proof for Proposition (4.5.2), Equation (5), for any $a, b \in A$, and $T_a, T_b \subseteq \mathcal{U}$, with $a = \bowtie T_a \wedge b = \bowtie T_b$.

$$
\begin{array}{ll}
& \tau(a \star b) \\
= & \quad \langle \text{ Definition of } \star \text{ (4.12) } \rangle \\
& \tau(a \cap b) \\
= & \quad \langle \text{ Assumptions } a = \bowtie T_a \ \wedge \ b = \bowtie T_b \ \rangle \\
& \tau(\ \bowtie T_a \cap \ \bowtie T_b\ ) \\
= & \quad \langle \text{ Proposition (4.3.4) } \rangle \\
& \tau(\ \bowtie (T_a \cap T_b)\ ) \\
= & \quad \langle \text{ Proposition (4.5.2), (4) } \rangle \\
& \{\eta(s) \ \mid \ s \ . \ s \in T_a \cap T_b\} \\
= & \quad \langle \text{ Set comprehension } \rangle \\
& \{\eta(s) \ \mid \ s \ . \ s \in T_a\} \cap \{\eta(s) \ \mid \ s \ . \ s \in T_b\} \\
= & \quad \langle \text{ Assumptions, and Proposition (4.5.2), (4) } \rangle \\
& \tau(a) \cap \tau(b) \\
= & \quad \langle \text{ Definition of } \otimes \text{ (4.20) } \rangle \\
& \tau(a) \otimes \tau(b)
\end{array}
$$

Appendix B

DIS Specification in Isabelle/HOL

This Appendix has been automatically generated by Isabelle/HOL, by applying the document generator tool on the specifications written in Isabelle.

B.1 Set Comprehension Results

Domain Information System proofs use well-known set comprehension laws, that are not found in the core of Isabelle/HOL and are proved in this section.

```
theory SetComprehension
  imports
    Main
begin

lemma setcomp_union: "{x. x ∈ A ∪ B} = {x. x ∈ A} ∪ {x. x ∈ B}" by auto
lemma setcomp_int: "{x. x ∈ A ∩ B} = {x. x ∈ A} ∩ {x. x ∈ B}" by auto
lemma setcomp_and: "{x| x. x ∈ {y| y. y ∈ A ∧ P y} ∧ Q x} =
       {x| x. x ∈ A ∧ P x ∧ Q x}"
  by auto
lemma setcomp_congr0:
  "{E x | x. (∃y z. x = F y z) ∧ Q x} = {E x | x y z. x = F y z ∧ Q x}" by
simp
lemma setcomp_congr1:
  assumes "⋀x y z. E x y z = F x y z"
shows "{E x y z | x y z. P x y z} = {F x y z| x y z. P x y z}"
  using assms by auto
lemma setcomp_distr0:
      "{x| x. (x ∈ X ∧ P x) ∨ (x ∈ X ∧ Q x)} =
       {x| x. x ∈ X ∧ (P x ∨  Q x)}"
  by auto
```

```
lemma setcomp_distr1:
      "{x| x. (x ∈ X ∧ P x) ∨ (x ∈ X ∧ ¬(P x))} = {x| x. x ∈ X}"
  by auto
lemma setcomp_distr2:
      "{x| x. (x ∈ X ∧ P x) ∧ (x ∈ X ∧ ¬(P x))} = {}"
  by auto

lemma set_image_union:
  shows "{f x | x. x ∈ A ∪ B} = {f x| x. x ∈ A} ∪ {f x| x. x ∈ B}"
proof -
  have "{f x| x. x ∈ A} = f`A"
    by (simp add: Setcompr_eq_image)
  then show ?thesis
    using Set.image_Un by auto
qed

lemma m: "X≠{} ⟹ Y≠{} ⟹ {{x| x. x ∈ X} | v. v ∈ Y} = {{x| x. x ∈
X}}"
  by blast

lemma "⋃{A} = A"
  by simp

lemma "{E x| x. x ∈ {y}} = {E y}"
  by simp


end
```

B.2 Inductive Finite Sets

Throughout the Universe and DDV theories, we use the fact that sorts and, ultimately, both sdatas and sdatums are finite sets. In this section, we give the theory of inductively defined finite sets, to support Universe and DDV proofs in using the inductive rule.

```
theory FinIndSet
  imports
    Main

begin

inductive_set Fin where
```

```
  emptyI: "{} ∈ Fin" |
  insertI: "A ∈ Fin ⟹ insert a A ∈ Fin"

declare Fin.intros [intro]

lemma Fin_is_finite: "x ∈ Fin ⟷ finite x"
proof
  show "x ∈ Fin ⟹ finite x"
    using Fin.inducts by auto
  then show "finite x ⟹ x ∈ Fin"
    by (metis Fin.simps Finp_Fin_eq finite_ne_induct)
qed


end
```

B.3 Diagonal-free Cylindric algebra

```
theory DFCylindricAlgebra
  imports
    Main
    "HOL.Boolean_Algebras"

begin
```

A diagonal-free algebra is a Boolean algebra with a cylindrification operator

```
locale dfcyl_algebra =
   balg: abstract_boolean_algebra "mult" "plus" "compl" "zero" "one"
   for  mult :: "'a ⇒ 'a ⇒ 'a"  (infixr "*" 70)
    and plus :: "'a ⇒ 'a ⇒ 'a"  (infixr "+" 65)
    and compl :: "'a ⇒ 'a"  ("~_" [81] 80)
    and zero :: "'a"  ("0")
    and one  :: "'a"  ("1")  +
  fixes cyl :: "'b ⇒ 'a ⇒ 'a" ("c_ _" [91] 90)
    and carrier :: "'a set"
  assumes
        ck_norm: "(c_k 0) = 0"
    and ck_mod: "x ∈ carrier ⟹ y ∈ carrier ⟹ (c_k (x * (c_k y))) = (c_k
x) * (c_k y)"
    and ck_comm: "x ∈ carrier ⟹ (c_k (c_l x)) = (c_l (c_k x))"
begin
lemma (in dfcyl_algebra) ck_zero: "x + (c_k 0) = x"
```

```
  by (simp add: ck_norm)
end

end
```

B.4 Domain Data View Types

The Domain Data View is built using a Cylindric Algebra whose carrier set elements are records of data. Usually, a record is represented as a tuple of values, each value corresponding to an attribute in the dataset. Tuples are ordered structures, and the cylindric algebra is based on a Boolean algebra. Thus, to achieve the commutativity of the Boolean algebra operators, the records must be unordered. In our interpretation, a record is a set of *sdatum*.

```
theory DDVTypes
  imports
    Main

begin
```

B.4.1 Polimorphic Type: Definitions

In Isabelle, sets are monomorphic, therefore we create a datatype *dtype* to simulate polimorphic types. This is not a perfect solution, because datatypes are not extendable.

```
datatype dtype = Nat nat | Int int | Bool bool | Str string
```

To help construction of the generic *dtype* from concrete types, we provide translation functions.

```
definition nat2dtype :: "nat set ⇒ dtype set"
  where "nat2dtype s = {Nat n | n. n ∈ s}"

definition int2dtype :: "int set ⇒ dtype set"
  where "int2dtype s = {dtype.Int i | i. i ∈ s}"

definition bool2dtype :: "bool set ⇒ dtype set"
  where "bool2dtype s = {Bool b | b. b ∈ s}"

definition str2dtype :: "string set ⇒ dtype set"
  where "str2dtype s = {Str x | x. x ∈ s}"

type_synonym sort = "dtype set"
```

B.4.2 Sorted-values, -datums, and -datas: Definitions

The *svalue* record is a pair (sort-name, sort-value). It is the foundation of transforming a data record tuple (ordered structure) into a set of pairs (unordered structure). In doing so, the usual operators on sets (union, intersection) ensure the commutativity of operators required by the Boolean algebra.

```
type_synonym svalue = "string × dtype"

definition make_svalue :: "string ⇒ dtype ⇒ svalue" ("⟨_, _⟩")  where
  "⟨s, v⟩ = (s, v)"

definition sname :: "svalue ⇒ string" where "sname x = fst x"
definition sval :: "svalue ⇒ dtype" where "sval x = snd x"
```

The following type synonyms are syntactic sugar, to simplify notation and understanding of the Domain Data View (DDV) structures. A set of *svalue*s is abbreviated as an *sdatum*, and a set of *sdatum*s is abbreviated as an *sdata*. The carrier set of the cylidric algebra consists of *sdata*s.

```
type_synonym sdatum = "svalue set"
type_synonym sdata = "sdatum set"

definition is_sdwf :: "sdatum ⇒ bool"
  where "is_sdwf dt ≡ True"
```

B.4.3 Sorted-values: Properties

```
lemma svalue_eq [iff]: "x = y ⟷ (sname x = sname y ∧ sval x = sval y)"
  by (metis prod_eqI sname_def sval_def)

lemma svalue_neq_fst: "sname x ≠ sname y ⟹ x ≠ y"
  by auto
lemma svalue_neq_snd: "sval x ≠ sval y ⟹ x ≠ y"
  by auto


end
```

B.5 Domain Data View Universe

The foundational element of the Domain Data View is a finite set of sorts, called the universe. The sorts are given by the attributes of the dataset and each sort consists

of a finite set of values.

```
theory DDVUniverse
  imports DDVTypes
          FinIndSet
          "HOL-Library.FuncSet"
begin

locale universe =
  fixes 𝒰 :: "sort set"
    and sort2name :: "sort ⇒ string"
    and name2sort :: "string ⇒ sort"

assumes U_notempty: "𝒰 ≠ {}"
      and U_fin: "finite 𝒰"
      and U_sorts_fin: "s ∈ 𝒰 ⟹ finite s"
      and U_sorts_notempty: "s ∈ 𝒰 ⟹ s ≠ {}"
      and sort2name_inj: "⋀x y. x ∈ 𝒰 ⟹ y ∈ 𝒰 ⟹ sort2name x =
sort2name y ⟹ x = y"
      and sort2name_fun: "⋀x y. x ∈ 𝒰 ⟹ y ∈ 𝒰 ⟹ x ≠ y ⟹ sort2name
x ≠ sort2name y"
      and sort2name2sort: "sort2name (name2sort n) = n"
      and name2sort2name: "⋀s. s ∈ 𝒰 ⟹ name2sort (sort2name s) = s"
begin
```

Syntactic sugar for `svalue` construction

```
definition svalue_constr :: "sort ⇒ dtype ⇒ svalue" ("⟪_, _⟫") where
  "⟪s, v⟫ = ⟨sort2name s, v⟩"

definition sort_eq :: "svalue ⇒ sort ⇒ bool" (infix "=S" 55) where
  "sv =S k ≡ sname sv = sort2name k"
definition sort_neq :: "svalue ⇒ sort ⇒ bool" (infix "≠S" 55) where
  "sv ≠S k ≡ sname sv ≠ sort2name k"

lemma sort_svalue: "sname ⟪s, v⟫ = sort2name s"
  by (simp add: make_svalue_def svalue_constr_def sname_def)
```

B.5.1 Universe operators: Definitions

Extension operator `ext` is used to construct `SV`, the set of all possible svalues in the Universe.

```
definition ext :: "sort ⇒ svalue set" where
  "ext s = {⟪s, v⟫ | v. v ∈ s}"
```

```
definition SV :: "svalue set" where
  "SV = ⋃{ext s | s. s ∈ 𝒰}"
```

SD, the set of all possible sdatas in the Universe, is defined using the ⋈ operator, which in turn is defined through a special Carthesian product operator, *pcart*.

```
definition pcart :: "svalue set set ⇒ svalue set set"  where
  "pcart A = {X. ∃f. f ∈ X → A ∧ inj_on f X ∧ (∀x∈X. x ∈ f x)}"

definition bow :: "sort set ⇒ svalue set set" ("⋈") where
  "⋈U = pcart (ext ‘ U)"

definition SD :: "sdatum set" where
  "SD = ⋈𝒰"

end
```

B.5.2 Universe operators: Properties

Properties of Carthesian product operator

```
context universe
begin
lemma (in universe) pcartI:
  assumes "f ∈ X → A" "inj_on f X" "⋀x. x ∈ X ⟹ x ∈ f x"
  shows   "X ∈ pcart A"
  using assms by (auto simp: pcart_def)

lemma (in universe) pcartE:
  assumes "X ∈ pcart A"
  obtains f where "f ∈ X → A" "inj_on f X" "⋀x. x ∈ X ⟹ x ∈ f x"
  using assms by (auto simp: pcart_def)

lemma (in universe) pcart_mono:
  assumes "A ⊆ B"
  shows   "pcart A ⊆ pcart B"
  using assms unfolding pcart_def by fast

lemma (in universe) pcart_zero: "pcart {} = {{}}"
  using pcart_def by blast

lemma (in universe) pcart_insert:
  assumes "a ∉ A"
```

```
  shows "pcart (insert a A) = pcart A ∪ {insert x y |x y. x ∈ a ∧ y ∈
pcart A}"
proof (intro equalityI subsetI)
  fix X assume "X ∈ pcart (insert a A)"
  then obtain f where f: "f ∈ X → insert a A" "inj_on f X" "⋀x. x ∈ X
⟹ x ∈ f x"
    using pcartE by blast
  show "X ∈ pcart A ∪ {insert x y |x y. x ∈ a ∧ y ∈ pcart A}"
  proof (cases "a ∈ f ‘ X")
    case False
    have "X ∈ pcart A"
    proof (rule pcartI)
      show "f ∈ X → A"
        using f False by (auto simp: Pi_def)
    qed (use f in auto)
    thus ?thesis
      by blast
  next
    case True
    obtain xa where xa: "xa ∈ X" "a = f xa"
      using True by blast
    have fa: "f ∈ X-{xa} → A" "inj_on f X" "⋀x. x ∈ X-{xa} ⟹ x ∈ f x"
    proof -
      show "⋀x. x ∈ X-{xa} ⟹ x ∈ f x"
        using xa f(3) by auto
      show "inj_on f X"
        using f(2) by auto
      then have f01:  "⋀x. x ∈ X-{xa} ⟹ f x ≠ a"
        using f(2) xa by (auto simp: inj_on_contraD)
      then have f02:  "⋀x. x ∈ X ⟹ f x ∈ insert a A"
        using f(1) by auto
      then have "⋀x. x ∈ X-{xa} ⟹ f x ∈ A"
        using f01 f02 by auto
      then show "f ∈ X-{xa} → A"
        using xa f(1) by auto
    qed
    have X_xa: "X - {xa} ∈ pcart A"
      using pcartI xa fa by (meson inj_on_diff)
    have "{xa} ∈ {insert x y |x y. x ∈ a ∧ y ∈ pcart A}"
      using f pcart_def xa
    proof -
      have "{} ∈ {P. ∃f. f ∈ P → A ∧ inj_on f P ∧ (∀p. p ∈ P ⟶ p ∈ f
p)}"
```

```
        by simp
      then have "∃p P. {xa} = insert p P ∧ p ∈ a ∧ P ∈ pcart A"
        by (metis (lifting) f(3) pcart_def xa(1) xa(2))
      then show ?thesis
        by fastforce
    qed
    then show ?thesis
      using X_xa
      by (smt (verit, del_insts) UnI2 f insert_Diff mem_Collect_eq xa)
  qed
next
  fix X assume "X ∈ pcart A ∪ {insert x y |x y. x ∈ a ∧ y ∈ pcart A}"
  thus "X ∈ pcart (insert a A)"
  proof
    assume "X ∈ pcart A"
    thus "X ∈ pcart (insert a A)"
      using pcart_mono by blast
  next
    assume "X ∈ {insert x y |x y. x ∈ a ∧ y ∈ pcart A}"
    then obtain x X' where *: "X = insert x X'" "x ∈ a" "X' ∈ pcart A"
      by blast
    then obtain f' where f': "f' ∈ X' → A" "inj_on f' X'" "⋀x'. x' ∈
X' ⟹ x' ∈ f' x'"
      using pcartE by blast
    show "X ∈ pcart (insert a A)"
    proof (cases "x ∈ X'")
      case True
      then show ?thesis using f' *
        by (metis insert_absorb pcart_mono subsetD subset_insertI)
    next
      case False
      obtain f where f: "f ≡ (λw. if w = x then a else f' w)"
        by auto
      show ?thesis
      proof (rule pcartI)
        show "f ∈ X → insert a A"
          using f f' assms * Pi_I Pi_mem
          by (smt (verit) insert_iff)
        show "inj_on f X"
          using f f' assms *
          by (smt (verit) Pi_iff inj_on_def insert_iff)
       show "⋀x. x ∈ X ⟹ x ∈ f x"
          using f f' assms *
```

```
          by (metis insertE)
      qed
    qed
  qed
qed
end
```

Properties of sort extension operator

```
context universe
begin

lemma (in universe) extI:
  assumes "v ∈ s"
  shows   "⟪s, v⟫ ∈ ext s"
  using assms by (auto simp: ext_def)

lemma (in universe) extE:
  assumes "⟪s, v⟫ ∈ ext s"
  obtains v where "v ∈ s"
  using assms by (auto simp: ext_def)

lemma (in universe) ext_zero:
  "ext {} = {}"
  using ext_def by auto

lemma (in universe) ext_nzero:
  assumes "s ≠ {}"
  shows "ext s ≠ {}"
proof -
  obtain v where v: "v = (SOME x. x ∈ s)"
    using assms by auto
  then have "⟪s, v⟫ ∈ ext s"
    using v ext_def assms some_in_eq by auto
  thus ?thesis  by auto
qed

lemma (in universe) svconstr_neq_v:
  assumes "v1 ≠ v2 "
  shows "⟪s, v1⟫ ≠ ⟪s, v2⟫"
  using svalue_constr_def make_svalue_def assms svalue_neq_snd
  by (metis old.prod.inject)
```

```
lemma (in universe) ext_v_nin_s:
  assumes "v ∉ s"
  shows "⟪s, v⟫ ∉ ext s"
  by (smt (verit, ccfv_threshold) assms ext_def mem_Collect_eq
svconstr_neq_v)

lemma (in universe) ext_neq_s:
  assumes "s1 ∈ 𝒰" "s2 ∈ 𝒰" "s1 ≠ s2"
  shows "ext s1 ≠ ext s2"
proof (cases "s1≠{} ∧ s2≠{}")
  case False
  then show ?thesis using ext_zero ext_nzero False
    by (metis assms(3))
next
  case True
  obtain v1 where v1: "v1 = (SOME x. x ∈ s1)" using True by auto
  obtain v2 where v2: "v2 = (SOME x. x ∈ s2)" using True by auto
  have ev1: "⟪s1, v1⟫ ∈ ext s1"
    by (simp add: True extI some_in_eq v1)
  have ev2: "⟪s2, v2⟫ ∈ ext s2"
    by (simp add: True extI some_in_eq v2)
  have "⟪s1, v1⟫ ≠ ⟪s2, v2⟫"
    using assms sort2name_fun svalue_neq_fst
    by (metis sort_svalue)
  then show ?thesis
    using True ev1 ev2
    by (smt (verit) assms ext_def mem_Collect_eq name2sort2name sort_svalue)
qed

lemma (in universe) ext_eq:
  assumes "s1 ∈ 𝒰" "s2 ∈ 𝒰" "ext s1 = ext s2"
  shows "s1 = s2"
  using assms ext_neq_s by auto

lemma (in universe) ext_inserta0:
  assumes "a ∈ 𝒰" "A ⊆ 𝒰" "ext a ∈ ext ‘ A"
  shows "a ∈ A"
proof -
  obtain x where x: "x ∈ A" "ext a = ext x"
    using assms by auto
  then have "a = x"
    using ext_eq x(2) assms(1) assms(2) by auto
  thus ?thesis using x(1) by auto
```

```
qed

lemma (in universe) ext_inserta:
  assumes "a ∈ 𝒰" "A ⊆ 𝒰" "a ∉ A"
  shows "(ext a) ∉ (ext ‘ A)"
  using ext_inserta0 assms by auto
end
```

Properties of bow operator

```
context universe
begin

lemma (in universe) bow_mono:
  assumes "A ⊆ B"
  shows   "⋈ A ⊆ ⋈ B"
  using assms pcart_mono bow_def
  by (simp add: image_mono)
```

Definition 4.1 (induction base)

```
lemma (in universe) bow_zero: "⋈{} = {{}}"
  using bow_def pcart_zero by auto
```

Definition 4.1 (induction hypothesis)

```
lemma bow_insert:
  assumes "a ∈ 𝒰" "A ⊆ 𝒰" "a ∉ A"
  shows "⋈ (insert a A) = ⋈ A ∪ {insert x y |x y. x ∈ ext a ∧ y ∈ ⋈ A}"
proof -
  have "⋈ (insert a A) = pcart (insert (ext a) (ext ‘ A))"
    using Set.image_insert bow_def by auto
  then have "pcart (insert (ext a) (ext ‘ A)) = pcart (ext ‘ A) ∪
                {insert x y |x y. x ∈ ext a ∧ y ∈ pcart (ext ‘ A)}"
    using pcart_insert ext_inserta assms by auto
  thus ?thesis using bow_def by auto
qed

lemma (in universe) bow_sg:
  assumes "s ∈ 𝒰"
  shows "⋈{s} = {{}} ∪ {{x}| x. x ∈ ext s}"
    using bow_insert bow_zero assms by auto

lemma (in universe) bow_zero_mono:
  assumes "T ⊆ 𝒰"
```

```
  shows "{} ∈ ⋈T"
  using bow_insert bow_zero bow_mono assms
  by force
end

context universe
begin
```

Proposition 4.3.2

```
lemma (in universe) sdprod_s_in_T:
  assumes "T ⊆ 𝒰" "s ∈ T"
  shows "⋈{s} ∩ ⋈T = ⋈{s}"
proof-
  from assms obtain T' where T': "T' = T - {s}" by auto
  then show ?thesis
  proof (cases "T'={}")
    case True
    then show ?thesis using True assms
      by (metis Int_absorb T' insert_Diff)
  next
    case False
    then show ?thesis
      using assms(2) bot.extremum bow_mono inf.orderE insert_subsetI by
blast
  qed
qed


end

end
```

B.6 Domain Data View Boolean Algebra

The DomainDataView theory is built on top of a cylindric algebra, which, in turn, is built on top of a Boolean Algebra. In this section, we define the DDV Boolean Algebra theory *DDV_BA* and describe its properties.

```
theory DDV_BA
  imports
    DDVUniverse
    "HOL.Boolean_Algebras"
```

```
    "HOL-Algebra.Group"
begin
locale ddv_ba = universe +
  assumes a_in_SD: "⋀a::sdata. a ⊆ SD"
begin
```

B.6.1 Boolean Algebra Operators: Definitions

DDV Boolean algebra operators: plus, star, zero, one, and complements

```
definition plus :: "sdata ⇒ sdata ⇒ sdata" (infixl "+"  80) where
  "a + b = a ∪ b"
definition star :: "sdata ⇒ sdata ⇒ sdata" (infixl "*" 70) where
  "a * b = a ∩ b"
definition zero :: "sdata" ("0") where
  "0 = {}"

definition one :: "sdata" ("1") where
  "1 = SD"
definition compl :: "sdata ⇒ sdata" ("-") where
  "-a = 1 - a"

definition carrier_set :: "sdata set" ("A_A") where
    "A_A ≡ Pow SD"

lemma (in ddv_ba) dt_in_SD: "dt ∈ SD ⟹ {dt} ∈ A_A"
  using carrier_set_def by auto
end
```

B.6.2 Boolean Algebra Operators: Properties

Properties of closure on carrier set

```
context ddv_ba
begin
lemma (in ddv_ba)  one_closed: "1 ∈ A_A"
  by (simp add: carrier_set_def local.one_def)

lemma (in ddv_ba) a_in_one:
  assumes "a ∈ A_A"
    shows "a ⊆ 1"
  using assms one_def carrier_set_def by auto

lemma (in ddv_ba) a_in_one0: "a ⊆ 1"
```

```
  using a_in_SD a_in_one one_def by auto

lemma (in ddv_ba) compl_closed:
  assumes "a ∈ A_A"
  shows "-a ∈ A_A"
  using assms carrier_set_def compl_def one_def by auto

lemma (in ddv_ba) plus_closed:
  assumes "a ∈ A_A" "b ∈ A_A"
  shows "a + b ∈ A_A"
  using assms a_in_one carrier_set_def one_def plus_def by auto

lemma (in ddv_ba) zero_closed: "0 ∈ A_A"
  using zero_def carrier_set_def by auto

lemma (in ddv_ba) star_closed:
  assumes "a ∈ A_A" "b ∈ A_A"
  shows "a * b ∈ A_A"
  using assms a_in_one carrier_set_def one_def star_def by auto
end
```

Properties of plus operator

```
context ddv_ba
begin
lemma (in ddv_ba) plus_comm: "a + b = b + a"
  by (simp add: Un_commute plus_def)

lemma (in ddv_ba) plus_assoc: "a + (b + c) = (a + b) + c"
  by (simp add: Un_assoc plus_def)

lemma (in ddv_ba) plus_zero_r: "a + 0 = a"
  by (simp add: plus_def zero_def)

lemma (in ddv_ba) plus_zero_l: "0 + a = a"
  by (simp add: plus_def zero_def)

lemma (in ddv_ba) plus_one_r:
  assumes "a ∈ A_A"
  shows "a + 1 = 1"
  using assms one_def carrier_set_def plus_def by auto
end
```

Properties of star operator

```
context ddv_ba
begin
lemma (in ddv_ba) star_comm: "a * b = b * a"
  by (simp add: Int_commute star_def)

lemma (in ddv_ba) star_assoc: "a * (b * c) = (a * b) * c"
  by (simp add: Int_assoc star_def)

lemma (in ddv_ba) star_one_r:
  assumes "a ∈ A_A"
  shows "a * 1 = a"
  using assms one_def star_def carrier_set_def by auto

lemma (in ddv_ba) star_one_l:
  assumes "a ∈ A_A"
  shows "1 * a = a"
  using assms star_one_r star_comm by auto

lemma (in ddv_ba) star_zero_r: "a * 0 = 0"
  by (simp add: star_def zero_def)

end
```

Properties of distributivity of plus/star operators over each other

```
context ddv_ba
begin

lemma (in ddv_ba) plus_distr_star: "c * (a + b) = (c * a) + (c * b)"
  by (simp add: Int_Un_distrib plus_def star_def)
lemma (in ddv_ba) plus_distr_star2: "(a + b) * c = (a * c) + (b * c)"
  by (simp add: Int_Un_distrib2 plus_def star_def)

lemma (in ddv_ba) star_distr_plus: "c + (a * b)  = (c + a) * (c + b)"
  by (simp add: Un_Int_distrib plus_def star_def)
lemma (in ddv_ba) star_distr_plus2: "(a * b) + c = (a + c) * (b + c)"
  by (simp add: Un_Int_distrib2 plus_def star_def)

end
```

Properties of complement operator

```
context ddv_ba
begin
lemma (in ddv_ba) complement_star: "a * -a = 0"
  using star_def compl_def zero_def by simp

lemma (in ddv_ba) complement_plus:
  assumes "a ∈ A_A"
  shows "a + -a = 1"
  using assms plus_def compl_def one_def a_in_one plus_one_r by auto

lemma (in ddv_ba) complement_plus0: "a + -a = 1"
  using complement_plus a_in_SD carrier_set_def one_def by auto

end
```

B.6.3 DDV Boolean Algebra: Properties

Two monoids within the DDV

```
context ddv_ba
begin
interpretation plus_monoid: comm_monoid "⦇carrier = A_A, mult = (+), one =
0⦈"
  apply unfold_locales
  — Carrier set is closed over Plus
  apply (simp add: plus_closed)
  — Plus is associative
  apply (simp add: plus_assoc)
  — Zero is in the carrier set
  apply (simp add: zero_closed)
  — Zero is neutral element for plus
  apply (simp add: plus_zero_l)
  apply (simp add: plus_zero_r)
  — Plus is commutative
  apply (simp add: plus_comm)
  done

interpretation star_monoid: comm_monoid "⦇carrier = A_A, mult = (*), one =
1⦈"
  apply unfold_locales
  — Carrier set is closed over Star
  apply (simp add: star_closed)
```

```
  — Star is associative
  apply (simp add: star_assoc)
  — One is in the carrier set
  apply (simp add: one_closed)
  — One is neutral elements for Star
  apply (simp add: star_one_l)
  apply (simp add: star_one_r)
  — Star is commutative
  apply (simp add: star_comm)
  done
end
```

The DDV as a model for a Boolean algebra

```
context ddv_ba
begin
```

The *Isabelle/HOL* Boolean algebra is using a few extra operators (substract and ordering). In this section, we define them using the classic set definitions, and we show some of their properties, to be used by Boolean algebra interpretation proof)

```
definition minus :: "sdata ⇒ sdata ⇒ sdata" (infixl "\<^sub>A" 80) where
  "a \<^sub>A b = a - b"
definition lesseq :: "sdata ⇒ sdata ⇒ bool" (infixl "⩽\<^sub>A" 85) where
  "a ⩽\<^sub>A b = (a ⊆ b)"
definition less :: "sdata ⇒ sdata ⇒ bool" (infixl "<\<^sub>A" 85) where
  "a <\<^sub>A b = (a ⊂ b)"

lemma (in ddv_ba) leI: "a <\<^sub>A b = (a ⩽\<^sub>A b ∧ ¬ b ⩽\<^sub>A a)"
  using less_def lesseq_def
  by (simp add: dual_order.strict_iff_not)
lemma (in ddv_ba) le_refl: "a ⩽\<^sub>A a"
  using lesseq_def by simp
lemma (in ddv_ba) le_trans:
  assumes "a ⩽\<^sub>A b" "b ⩽\<^sub>A c"
  shows "a ⩽\<^sub>A c"
  using assms lesseq_def by simp
lemma (in ddv_ba) le_eq:
  assumes "a ⩽\<^sub>A b" "b ⩽\<^sub>A a"
  shows "a = b"
  using assms lesseq_def by simp

lemma (in ddv_ba) le_star: "(a * b) ⩽\<^sub>A a"
  using lesseq_def star_def by simp
```

```
lemma (in ddv_ba) le_star_trans:
  assumes "a ⩽A b" "a ⩽A c"
  shows "a ⩽A (b * c)"
  using assms lesseq_def star_def by auto

lemma (in ddv_ba) complement_minus:
  assumes "a ∈ AA" "b ∈ AA"
  shows "a \A b = a * -b"
  using assms minus_def star_def compl_def a_in_one by force

lemma (in ddv_ba) complement_minus0:
  shows "a \A b = a * -b"
  using complement_minus a_in_SD carrier_set_def by auto

interpretation boolean_algebra "(\A)" "(-)" "(*)" "(⩽A)" "(<A)" "(+)" "0"
"1"
  apply unfold_locales
                  apply (simp add: leI)
                 apply (simp add: lesseq_def)
                apply (simp add: lesseq_def)
               apply (simp add: lesseq_def)
              apply (simp add: lesseq_def star_def)
             apply (simp add: lesseq_def star_def)
            apply (simp add: lesseq_def star_def)
           apply (simp add: lesseq_def plus_def)
          apply (simp add: lesseq_def plus_def)
         apply (simp add: lesseq_def plus_def)
        apply (simp add: lesseq_def zero_def)
       apply (simp add: lesseq_def a_in_one0)
      apply (simp add: star_distr_plus)
     apply (simp add: complement_star)
    apply (simp add: complement_plus0)
  apply (simp add: complement_minus0)
  done

interpretation abstract_boolean_algebra "(*)" "(+)" "(-)" "0" "1"
  by (simp add: local.boolean_algebra.abstract_boolean_algebra_axioms)
end

end
```

B.7 Domain Data View Base

The DDV base theory is built on top of the DDV Boolean Algebra theory and it describes the data operators and their properies.

```
theory DDV_Base
  imports
    DDV_BA
    SetComprehension

begin
```

The DDV base is a DDV Boolean algebra with finite and well formed sdatums and sdatas

```
locale ddv_base = ddv_ba +
  assumes a_fin: "a ∈ A_A ⟹ finite a"
      and dt_fin: "⋀dt. dt ∈ SD ⟹ finite dt"
      and dt_one_k: "⋀dt. dt ∈ SD ⟹ ⟪s, v⟫ ∈ dt ⟹ ¬(∃ v'. v'≠v ∧ ⟪s,
v'⟫ ∈ dt)"
      and dt_v_in_k: "⋀dt. dt ∈ SD ⟹ ⟪s, v⟫ ∈ dt ⟹ v ∈ s"
      and dt_not_empty: "⋀a dt. a ∈ A_A ⟹ dt ∈ a ⟹ dt ≠ {}"
begin
```

B.7.1 Data Operators: Definitions

Definition of reduce/extend data operators

Definition 4.4

```
definition reduce_sd :: "sdatum ⇒ sort ⇒ sdatum" ("_↓_")  where
  "(dt↓_k) = {sv| sv. sv ∈ dt ∧ sv ≠_S k}"
```

Definition 4.6

```
definition reduce :: "sdata ⇒ sort ⇒ sdata" ("_⇓_") where
  "(a⇓_k) = {dt↓_k| dt. dt ∈ a}"

definition k_in_dt :: "sort ⇒ sdatum ⇒ bool" (infixl "∈" 120) where
  "k ∈ dt = (∃ v∈ k. ⟪k, v⟫ ∈ dt)"
definition k_nin_dt :: "sort ⇒ sdatum ⇒ bool" (infixl "∉" 120) where
  "k ∉ dt = (¬(k ∈ dt))"
definition k_in_a :: "sort ⇒ sdata ⇒ bool" (infixl "∈_a" 101)
  where "k ∈_a a = (∃ dt ∈ a. k ∈ dt)"
```

Definition 4.7

```
definition extend_sd :: "sdatum ⇒ sort ⇒ sdata" ("_↑-") where
```

```
  "(dt↑k) = (if k ∉ dt then {dt} else {(dt↓k) ∪ {⟪k, v⟫| v. v ∈ k})"
```

Definition 4.8

```
definition extend :: "sdata ⇒ sort ⇒ sdata" ("_⇑-") where
  "(a⇑k) = ⋃{dt↑k | dt. dt ∈ a}"
```

Definition of floored/raised data operators

Definition 4.11

```
definition floored :: "sdata ⇒ sort ⇒ sdata" ("⌊_⌋_")  where
  "(⌊a⌋k) = {dt. dt ∈ a ∧ k ∈ dt}"
```

Definition 4.12

```
definition raised :: "sdata ⇒ sort ⇒ sdata" ("⌈_⌉-")  where
  "(⌈a⌉k) = {dt. dt ∈ a ∧ k ∉ dt}"

end
```

B.7.2 Data Operators: Properties

Properties of data operators: helpers

```
context ddv_base
begin

lemma (in ddv_base) dt_in_SD:
  shows "dt ∈ SD"
  using a_in_SD by auto

lemma (in ddv_base) dt_in_extenda:
  assumes "dt ∈ a"
  shows "(dt↑k) ⊆ (a⇑k)"
    using extend_def assms
    by auto

lemma (in ddv_base) kv_nin_reduce:
  assumes "dt ∈ SD" "k ∈ 𝒰" "v ∈ k"
  shows "⟪k, v⟫ ∉ (dt↓k)"
  using assms dt_in_SD dt_v_in_k by auto

lemma (in ddv_base) k_nin_reduce:
  assumes "dt ∈ SD" "k ∈ 𝒰"
  shows "k ∉ (dt↓k)"
```

```
  using assms kv_nin_reduce k_nin_dt_def k_in_dt_def by auto

lemma (in ddv_base) kv_nin_dt:
  assumes "a ∈ A_A" "dt ∈ a" "k ∉ dt"
  shows "¬(∃ v. ⟪k, v⟫ ∈ dt)"
  using assms dt_one_k k_nin_dt_def k_in_dt_def dt_v_in_k dt_in_SD by metis

lemma (in ddv_base) dt_has_kv:
  assumes "a ∈ A_A" "dt ∈ a"
  shows "∃ k v. ⟪k, v⟫ ∈ dt"
  using assms a_fin dt_not_empty
  using a_in_one0 one_closed by blast

lemma (in ddv_base) dt_and_reduce:
  assumes "a ∈ A_A" "dt ∈ a"
  shows "dt = (dt↓k) ∪ {⟪k, v⟫}"
  using assms dt_has_kv
  using dt_in_SD local.one_def one_closed by auto

end
```

Properties of extend data operator w.r.t zero and one

```
context ddv_base
begin

lemma (in ddv_base) zero_ext: "((0)⇑^k) = 0"
  using zero_def extend_def
  by simp

lemma (in ddv_base) one_ext: "((1)⇑^k) = 1"
  using one_def extend_def dt_in_SD dt_not_empty one_closed by fastforce

end
```

Properties of data operators $\bowtie$, $\downarrow$, $\uparrow$, $\Downarrow$, $\Uparrow$

Proposition 4.3.5

```
context ddv_base
begin
lemma (in ddv_base) reduce_sd_kappa_on_kappa:
  assumes "k ∈ 𝒰"
  shows "({⟪k, v⟫}↓k) = {}"
```

```
proof -
  have "({⟪k, v⟫}↓k) = {sv| sv. sv = ⟪k, v⟫ ∧ sv≠Sk}"
    using reduce_sd_def assms by auto
  then have "({⟪k, v⟫}↓k) = {sv| sv. False}"
    using a_in_one0 dt_not_empty one_closed by blast
  then show ?thesis by auto
qed

lemma (in ddv_base) reduce_sd_lambda_on_kappa:
  assumes "k ∈ 𝒰" "l ∈ 𝒰" "k ≠ l"
  shows "({⟪k, v⟫}↓l) = {⟪k, v⟫}"
  using dt_in_SD dt_not_empty local.one_def one_closed by auto
end
```

Proposition 4.3.6

```
context ddv_base
begin
lemma (in ddv_base) disjoint_reduced_extended:
  assumes "a ∈ A_A" "k ∈a a"
  shows   "(a⇓k) ∩ (a⇑k) = {}"
proof -
  have f: "a ∈ Fin"
    using assms a_fin Fin_is_finite by auto
  then show ?thesis
  proof (induct a)
    case emptyI
    then show ?case
      using zero_def zero_ext by auto
  next
    case (insertI a dt)
    then show ?case
      by (meson all_not_in_conv dt_in_SD dt_v_in_k insertI1)
  qed
qed
end
```

Proposition 4.3.7

```
context ddv_base
begin
```

Proposition 4.3.7, Eq.1

```
lemma (in ddv_base) reducesd_distr_union: "((dt₁ ∪ dt₂)↓ₖ) = (dt₁↓ₖ) ∪
(dt₂↓ₖ)"
  using reduce_sd_def setcomp_union by auto
```

Proposition 4.3.7, Eq.2

```
lemma (in ddv_base) reducesd_distr_int: "((dt₁ ∩ dt₂)↓ₖ) = (dt₁↓ₖ) ∩
(dt₂↓ₖ)"
proof -
  have f1: "((dt₁ ∩ dt₂)↓ₖ) = {sv| sv. sv ∈ dt₁ ∩ dt₂ ∧ sv ≠ₛ k}"
    using reduce_sd_def by auto
  have f2: "{sv| sv. sv ∈ dt₁ ∩ dt₂ ∧ sv ≠ₛ k} =
            {sv| sv. (sv ∈ dt₁ ∧ sv ≠ₛ k) ∧ (sv ∈ dt₂ ∧ sv ≠ₛ k)}"
    by auto
  have "{sv| sv. (sv ∈ dt₁ ∧ sv ≠ₛ k) ∧ (sv ∈ dt₂ ∧ sv ≠ₛ k)} =
            {sv| sv. sv ∈ dt₁ ∧ sv ≠ₛ k} ∩ {sv| sv. sv ∈ dt₂ ∧ sv ≠ₛ k}"
    by blast
  then show ?thesis
    using f1 f2 reduce_sd_def by auto
qed
```

Proposition 4.3.7, Eq.3

```
lemma (in ddv_base) reduce_distr_union: "((a ∪ b)⇓ₖ) = (a⇓ₖ) ∪ (b⇓ₖ)"
  using reduce_def reduce_sd_def setcomp_union by auto
```

Proposition 4.3.7, Eq.4

```
lemma (in ddv_base) reduce_distr_int: "((a ∩ b)⇓ₖ) ⊆ (a⇓ₖ) ∩ (b⇓ₖ)"
proof -
  have f1: "((b ∪ (a ∩ b))⇓ₖ) = ((b⇓ₖ) ∪ ((a ∩ b)⇓ₖ))"
    by (metis reduce_distr_union)
  have "((a ∪ (a ∩ b))⇓ₖ) = ((a⇓ₖ) ∪ ((a ∩ b)⇓ₖ))"
    by (metis reduce_distr_union)
  then show ?thesis
    using f1 by (simp add: sup.absorb_iff1 sup_inf_distrib2)
qed
```

Proposition 4.3.7, Eq.5

```
lemma (in ddv_base) extend_distr_union: "((a ∪ b)⇑ᵏ) = (a⇑ᵏ) ∪ (b⇑ᵏ)"
  using extend_def setcomp_union by auto
```

Proposition 4.3.7, Eq.6

```
lemma (in ddv_base) extend_distr_int0: "((a ∩ b)⇑ᵏ) ⊆ (a⇑ᵏ) ∩ (b⇑ᵏ)"
proof -
  have f1: "((b ∪ (a ∩ b))⇑ᵏ) = ((b⇑ᵏ) ∪ ((a ∩ b)⇑ᵏ))"
```

```
    by (metis extend_distr_union)
  have "((a ∪ (a ∩ b))⇑k) = ((a⇑k) ∪ ((a ∩ b)⇑k))"
    by (metis extend_distr_union)
  then show ?thesis
    using f1 by (simp add: sup.absorb_iff1 sup_inf_distrib2)
qed
end
```

Proposition 4.3.8

```
context ddv_base
begin
```

Proposition 4.4.6, Eq.1

```
lemma (in ddv_base) reducesd_idemp: "((dt↓k)↓k) = (dt↓k)"
proof -
  have "((dt↓k)↓k) = {sv| sv. sv ∈ (dt↓k) ∧ sv ♯S k}"
    using reduce_sd_def by auto
  have "{sv| sv. sv ∈ (dt↓k) ∧ sv ♯S k} =
        {sv| sv. sv ∈ {sv'| sv'. sv' ∈ dt ∧ sv' ♯S k} ∧ sv ♯S k}"
    using reduce_sd_def by auto
  have "{sv| sv. sv ∈ {sv'| sv'. sv' ∈ dt ∧ sv' ♯S k} ∧ sv ♯S k} =
        {sv| sv. sv ∈ dt ∧ sv ♯S k}"
    by blast
  then show ?thesis using reduce_sd_def by auto
qed
```

Proposition 4.3.8, Eq.2

```
lemma (in ddv_base) extsd_idemp: "((dt↑k)⇑k) = (dt↑k)"
  using dt_in_SD dt_not_empty local.one_def one_closed by auto
```

Proposition 4.3.8, Eq.3

```
lemma (in ddv_base) extend_idemp: "((a⇑k)⇑k) = (a⇑k)"
  using dt_in_SD dt_not_empty local.one_def one_closed by auto
```

Proposition 4.3.8, Eq.4

```
lemma (in ddv_base) reduce_idemp: "((a⇓k)⇓k) = (a⇓k)"
proof -
  have f1: "((a⇓k)⇓k) = {dt↓k| dt. dt ∈ (a⇓k)}"
    using reduce_def by auto
  have f2: "{dt↓k| dt. dt ∈ (a⇓k)} =
        {dt↓k| dt. dt ∈ {dt'↓k| dt'. dt' ∈ a}}"
    using reduce_def by auto
  have f3: "{dt↓k| dt. dt ∈ {dt'↓k| dt'. dt' ∈ a}} =
```

```
        {(dt↓k)↓k| dt. dt ∈ a}" by auto
  have f4: "{(dt↓k)↓k| dt. dt ∈ a} = {dt↓k| dt. dt ∈ a}"
    using reducesd_idemp by auto
  then show ?thesis using reduce_def f1 f2 f3 by auto
qed

end
```

Proposition 4.3.9

```
context ddv_base
begin
```

Proposition 4.3.9, Eq.1 - helper

```
lemma (in ddv_base) reduce_sd_assoc0: "((dt↓k)↓l) = {sv| sv. sv ∈ dt ∧
sv≠Sk ∧ sv ≠Sl}"
proof -
  have f1: "((dt↓k)↓l) = {sv| sv. sv ∈ (dt↓k) ∧ sv ≠S l}"
    using reduce_sd_def by auto
  have f2: "{sv| sv. sv ∈ (dt↓k) ∧ sv ≠S l} =
        {sv| sv. sv ∈ {sv'| sv'. sv' ∈ dt ∧ sv'≠Sk} ∧ sv≠Sl}"
    using reduce_sd_def by auto
  have f3: "{sv| sv. sv ∈ {sv'| sv'. sv' ∈ dt ∧ sv'≠Sk} ∧ sv≠Sl} =
        {sv| sv. sv ∈ dt ∧ sv≠Sk ∧ sv ≠Sl}"
    using setcomp_and by blast
  then show ?thesis using f1 f2  by auto
qed
```

Proposition 4.3.9, Eq.1

```
lemma (in ddv_base) reducesd_assoc: "((dt↓k)↓l) = ((dt↓l)↓k)"
  using reduce_sd_assoc0 by auto
```

Proposition 4.3.9, Eq.2 - helper

```
lemma (in ddv_base) reduce_assoc0: "((a⇓k)⇓l) = {(dt↓k)↓l| dt. dt ∈ a}"
proof -
  have f1: "((a⇓k)⇓l) = {dt↓l| dt. dt ∈ (a⇓k)}"
    using reduce_def by auto
  have f2: "{dt↓l| dt. dt ∈ (a⇓k)} = {dt↓l| dt. dt ∈ {dt'↓k| dt'. dt'∈a}}"
    using reduce_def by auto
  have f3: "{dt↓l| dt. dt ∈ {dt'↓k| dt'. dt'∈a}} =
          {(dt↓k)↓l| dt. dt ∈ a}"
    by auto
  then show ?thesis using f1 f2 by auto
qed
```

Proposition 4.3.9, Eq.2

```
lemma (in ddv_base) reduce_assoc: "((a⇓k)⇓l) = ((a⇓l)⇓k)"
  using reduce_def reducesd_assoc reduce_assoc0 by auto
```

Proposition 4.3.9, Eq.3 - helper

```
lemma (in ddv_base) extend_assoc_sg: "(({dt}⇑k)⇑l) = (({dt}⇑l)⇑k)"
proof -
  consider (kl_in_dt) "k ∈ dt ∧ l ∈ dt" |
           (kl_nin_dt) "k ∉ dt ∧ l ∉ dt" |
           (k_in_dt) "k ∈ dt ∧ l ∉ dt" |
           (l_in_dt) "k ∉ dt ∧ l ∈ dt"
    using k_nin_dt_def by auto
  then show ?thesis
  proof cases
    case kl_in_dt
    then show ?thesis
      using dt_in_SD dt_v_in_k by blast
  next
    case kl_nin_dt
    then show ?thesis
      using dt_in_SD dt_v_in_k by blast
  next
    case k_in_dt
    then show ?thesis
      using dt_in_SD dt_v_in_k by blast
  next
    case l_in_dt
    then show ?thesis
      using dt_in_SD dt_v_in_k by blast
  qed
qed
```

Proposition 4.3.9, Eq.3

```
lemma (in ddv_base) extend_assoc:
  assumes "a ∈ A_A"
  shows "((a⇑k)⇑l) = ((a⇑l)⇑k)"
proof -
  have "a ∈ Fin"
    using assms a_in_SD a_fin Fin_is_finite by auto
  then show ?thesis
  proof (induct a)
    case emptyI
    then show ?case
```

```
      using zero_def zero_ext by auto
  next
    case (insertI a dt)
    show ?case
    proof (cases "k=l")
      case True
      then show ?thesis by simp
    next
      case False
      then show ?thesis
        using extend_assoc_sg extend_distr_union
        by (metis insertI.hyps(2) insert_is_Un)
    qed
  qed
qed

end
```

Proposition 4.3.10

```
context ddv_base
begin
```

Proposition 4.3.10, Eq.1

```
lemma (in ddv_base) floor_raise_union: "(⌊a⌋_k) ∪ (⌈a⌉^k) = a"
proof -
  have f1: "(⌊a⌋_k) ∪ (⌈a⌉^k) = {dt. dt ∈ a ∧ k ∈ dt } ∪
                                 {dt. dt ∈ a ∧ k ∉ dt }"
    using floored_def raised_def by auto
  have f2: "{dt. dt ∈ a ∧ k ∈ dt } ∪
            {dt. dt ∈ a ∧ k ∉ dt } =
            {dt. (dt ∈ a ∧ k ∈ dt) ∨ (dt ∈ a ∧ k ∉ dt) }"
    by (simp add: Collect_disj_eq)
  have "{dt. (dt ∈ a ∧ k ∈ dt) ∨
             (dt ∈ a ∧ k ∉ dt) } =
        {dt. dt ∈ a}"
    using k_nin_dt_def by auto
  then show ?thesis using f1 f2 by auto
qed
```

Proposition 4.3.10, Eq.2

```
lemma (in ddv_base) floor_raise_disjoint: "(⌊a⌋_k) ∩ (⌈a⌉^k) = {}"
proof -
```

```
  have f1: "(⌊a⌋k) ∩ (⌈a⌉^k) = {dt. dt ∈ a ∧ k ∈ dt } ∩
                          {dt. dt ∈ a ∧ k ∉ dt }"
    using floored_def raised_def by auto
  also have f2: "{dt. dt ∈ a ∧ k ∈ dt } ∩
            {dt. dt ∈ a ∧ k ∉ dt } =
            {dt. (dt ∈ a ∧ k ∈ dt) ∧ (dt ∈ a ∧ k ∉ dt) }"
    using Collect_conj_eq by auto
  finally have "{dt. (dt ∈ a ∧ k ∈ dt) ∧ (dt ∈ a ∧ k ∉ dt) } = {}"
    using k_nin_dt_def by auto
  thus ?thesis using f1 f2 by auto
qed

end

lemma (in ddv_base) absorb_extend: "((a ∩ b)⇑^k) ⊆ (a⇑^k)"
proof -
  have "(a = (a ∪ (a ∩ b)))"
    by blast
  then show ?thesis
    by (metis (no_types) extend_distr_union sup.absorb_iff1)
qed

end
```

B.8 Domain Data View

The Domain Data View is extending the base with the original dataset `D`, that may be used for reasoning.

```
theory DomainDtVw
  imports
    DFCylindricAlgebra
    DDV_Base
begin

type_synonym dataconcept = "sdata set"

locale domain_data_view = ddv_base +
  fixes D :: "sdatum set"
begin
```

B.8.1 Cylindrification Operator: Definition

```
definition cyl :: "sort ⇒ sdata ⇒ sdata" ("c_ _" 120) where
  "(c_k a) ≡ ⋃{dt↑^k | dt. dt ∈ a}"

end
```

B.8.2 Cylindrification Operator: Properties

Axiom A.11

Axiom A.11 of DIS: $c_k\ 0 = 0$

```
context domain_data_view
begin
lemma (in domain_data_view)  cyl_zero: "(c_k 0) = 0"
  using cyl_def zero_def zero_ext
  by simp
end
```

Proposition 4.4.11

Cylindrification distributes over plus: $\forall a\ b \in A_A.\ c_k\ (a + b) = c_k\ a + c_k\ b$

```
context domain_data_view
begin
lemma (in domain_data_view) cyl_plus_sg:
  assumes "a ∈ A_A" and "dt ∈ SD" and "dt ∉ a"
  shows "(c_k ({dt} ∪ a)) = (c_k {dt}) ∪ (c_k a)"
proof -
  have "a ∈ Fin"
    using assms a_in_SD a_fin Fin_is_finite by auto
  then show ?thesis
  proof (induct a)
    case emptyI
    then show ?case
      using cyl_zero zero_def by auto
  next
    case (insertI a dt')
    then show ?case
      using dt_in_SD dt_v_in_k by blast
  qed
qed

lemma (in domain_data_view) cyl_plus:
```

```
  assumes "a ∈ A_A" "b ∈ A_A"
  shows "(c_k (a + b)) = (c_k a) + (c_k b)"
proof -
  have f: "a ∈ Fin"
    using <a ∈ A_A> a_in_SD a_fin Fin_is_finite by auto
  then show ?thesis
  proof (induct a arbitrary: b)
    case emptyI
    then show ?case
      using cyl_zero zero_def plus_def by auto
  next
    case (insertI a dt)
    then show ?case
    proof (cases "dt ∉ a")
      case True
      then show ?thesis
        by (meson Set.set_insert a_in_SD dt_v_in_k insert_subset
subset_insertI)
    next
      case False
      then show ?thesis
        by (simp add: insertI.hyps(2) insertI.prems insert_absorb)
    qed
  qed
qed
end
```

Axiom A.12

Axiom A.12 of DIS: $\forall$ `a ∈ A_A. a < c_k a`

```
context domain_data_view
begin
lemma (in domain_data_view) cyl_le_sg:
  assumes "k ∈ 𝒰" "a ∈ A_A" "a = {dt}"
  shows "a + (c_k a) = (c_k a)"
proof -
  have f: "(c_k a) = (dt↑^k)"
    using cyl_def assms by auto
  then show ?thesis
  proof (cases "k ∈ dt")
    case True
      have "(dt↑^k) = {(dt↓_k) ∪ {⟪k, v⟫| v. v ∈ k}"
        using extend_sd_def True k_nin_dt_def by auto
```

```
      obtain v::dtype where v: "(⟪k, v⟫ ∈ dt)"
        using True k_in_dt_def by auto
      have "dt = (dt↓k) ∪ {⟪k, v⟫}"
        using assms dt_and_reduce v dt_in_SD dt_in_SD by blast
      have "{dt} ∪ (dt↑k) = (dt↑k)"
        using dt_in_SD dt_not_empty local.one_def one_closed by auto
      then show ?thesis
        using assms f plus_def by auto
  next
    case False
    have "(c_k a) = a"
    proof -
      have "(dt↑k) = {dt}"
        using extend_sd_def False kv_nin_reduce k_nin_dt_def by auto
      then show ?thesis using assms f by auto
    qed
    then show ?thesis
      using plus_def assms by auto
  qed
qed

lemma (in domain_data_view) cyl_le:
  assumes "a ∈ A_A" "k ∈ 𝒰"
  shows "a + (c_k a) = (c_k a)"
proof -
  have "a ∈ Fin"
    using assms a_fin a_in_SD Fin_is_finite by auto
  then show ?thesis
  proof (induct a)
    case emptyI
    then show ?case
      using plus_zero_l zero_def by auto
  next
    case (insertI a dt)
    then show ?case
    proof (cases "dt ∈ a")
      case True
      then show ?thesis
        by (simp add: insertI.hyps(2) insert_absorb)
    next
      case False
      have "(c_k (a ∪ {dt})) = (c_k a) ∪ (c_k {dt})"
        using cyl_plus plus_def dt_in_SD
```

```
        by (metis Diff_iff dt_v_in_k insertI1)
      then have f: "(c_k (insert dt a)) = (c_k {dt}) ∪ (c_k a)"
        by (simp add: Un_commute)
      have f3: "{dt} ∪ (c_k {dt}) = (c_k {dt})"
        using assms cyl_le_sg cyl_plus_sg assms
        using ddv_ba.dt_in_SD ddv_ba_axioms dt_in_SD plus_def by auto
      then show ?thesis
        using f insertI.hyps(2) plus_def by auto
    qed
  qed
qed
end
```

Proposition 4.4.12

Idempotency of cylindrification operator: $\forall$ a $\in$ *SD*. $\forall$ *k* $\in \mathcal{U}$. $c_k(c_k\ a) = c_k\ a$

```
context domain_data_view
begin

lemma (in domain_data_view) cyl_assoc_sg:
  assumes "dt ∈ SD"
  shows "(c_k(c_k {dt})) = (c_k {dt})"
proof -
  have f: "(c_k {dt}) = (dt↑^k)"
    using cyl_def  by auto
  then show ?thesis
  proof (cases "k ∈ dt")
    case False
    have "(dt↑^k) = {dt}"
      using extend_sd_def False kv_nin_reduce k_nin_dt_def by auto
    then show ?thesis using f by auto
  next
    case True

    have f1: "(c_k (c_k {dt})) = (c_k (dt↑^k))"
      using cyl_def by auto
    have f2: "(c_k(dt↑^k)) = ⋃{x↑^k | x. x ∈ (dt↑^k)}"
      using cyl_def by auto
    have f3: "⋃{x↑^k | x. x ∈ (dt↑^k)} = ⋃{x↑^k | x. x ∈ {(dt↓_k) ∪ {⟪k, v⟫}|
v. v ∈ k}}"
      using extend_sd_def True k_nin_dt_def by auto
    have f4: "v ∈ k ⟹ k ∈ ((dt↓_k) ∪ {⟪k, v⟫})"
      using k_in_dt_def by auto
```

```
    have f5: "⋃{x↑ᵏ | x. x ∈ {(dt↓ₖ) ∪ {⟪k, v⟫}| v. v ∈ k}} =
              ⋃{((dt↓ₖ) ∪ {⟪k, v⟫})↑ᵏ | v. v ∈ k}"
      by auto
    have f6: "{((dt↓ₖ) ∪ {⟪k, v⟫})↑ᵏ | v. v ∈ k} =
              {{(((dt↓ₖ) ∪ {⟪k, v⟫})↓ₖ) ∪ {⟪k, v'⟫}| v'. v' ∈ k }| v. v ∈
k}"
      using extend_sd_def f4 k_in_dt_def k_nin_dt_def extend_def
      by (meson Set.set_insert dt_in_SD dt_v_in_k insertI1)
    have f7: "(((dt↓ₖ) ∪ {⟪k, v⟫})↓ₖ) = (dt↓ₖ)"
      using kv_nin_reduce reducesd_distr_union reducesd_idemp
            reduce_sd_lambda_on_kappa assms
            dt_and_reduce local.one_def one_closed by presburger
    have f8: "{{(((dt↓ₖ) ∪ {⟪k, v⟫})↓ₖ) ∪ {⟪k, v'⟫}| v'. v' ∈ k }| v. v ∈
k} =
              {{(dt↓ₖ) ∪ {⟪k, v'⟫}| v'. v' ∈ k }}"
      by (meson Set.set_insert dt_in_SD dt_v_in_k insertI1)
    have "{(dt↓ₖ) ∪ {⟪k, v'⟫}| v'. v' ∈ k } = (dt↑ᵏ)"
      using extend_sd_def True k_nin_dt_def by auto
    then have "⋃{{(dt↓ₖ) ∪ {⟪k, v'⟫}| v'. v' ∈ k }} = (cₖ {dt})"
      using cyl_def by auto
    then show ?thesis
      using f1 f2 f3 f5 f6 f8 by presburger
  qed
qed

lemma (in domain_data_view) cyl_assoc:
  assumes "a ∈ A_A" "k ∈ 𝒰"
  shows  "(cₖ(cₖ a)) = (cₖ a)"
proof -
  have f: "a ∈ Fin"
    using assms a_fin a_in_SD Fin_is_finite by auto
  then show ?thesis
  proof (induct a)
    case emptyI
    then show ?case using cyl_zero zero_def by auto
  next
    case (insertI a dt)
    then show ?case
    proof (cases "dt ∈ a")
      case True
      then show ?thesis
        by (simp add: insertI.hyps(2) insert_absorb)
    next
```

```
      case False
      have "(c_k (a ∪ {dt})) = (c_k a) ∪ (c_k {dt})"
        using cyl_plus plus_def dt_in_SD
        by (metis PowI a_in_SD carrier_set_def)
      then have f2: "(c_k (insert dt a)) = (c_k {dt}) ∪ (c_k a)"
        by auto
      have f3: "(c_k(c_k (insert dt a))) = (c_k((c_k {dt}) ∪ (c_k a)))"
        using f2 cyl_plus plus_def
        by simp
      have f4: "(c_k((c_k {dt}) ∪ (c_k a))) = (c_k(c_k {dt})) ∪ (c_k(c_k a))"
        using cyl_plus plus_def
        by (meson Set.set_insert a_in_SD dt_v_in_k insertI1 insert_subset)
      then show ?thesis
        by (simp add: dt_in_SD cyl_assoc_sg f2 f4 insertI.hyps(2))
    qed
  qed
qed
end
```

Axiom A.13

Axiom A.13 of DIS: $\forall a\ b \in A_A.\ \forall k \in \mathcal{U}.\ c_k\ (a * c_k\ b) = c_k\ a * c_k\ b$

```
context domain_data_view
begin

lemma (in domain_data_view)  cyl_mod:
  assumes "a ∈ A_A" "b ∈ A_A"
  shows "(c_k (a * (c_k b))) = (c_k a) * (c_k b)"
proof -
  have f: "a ∈ Fin"
    using assms a_fin a_in_SD Fin_is_finite by auto
  then show ?thesis
  proof (induct a arbitrary: b)
    case emptyI
    then show ?case
      using cyl_zero star_def zero_def by auto
  next
    case (insertI a dt)
    then show ?case
      by (meson Set.set_insert a_in_SD dt_v_in_k insert_subset
subset_insertI)
  qed
qed
```

```
end
```

Axiom A.13

Axiom A.14 of DIS: $\forall$ a $\in$ A_A. $\forall$ k l $\in$ $\mathcal{U}$. c_k (c_l a) = c_l (c_k a)

```
context domain_data_view
begin
lemma (in domain_data_view)  cyl_comm:
  assumes "a ∈ A_A"
  shows "(c_k (c_l a)) = (c_l (c_k a))"
proof -
  have f: "a ∈ Fin"
    using assms a_fin a_in_SD Fin_is_finite by auto
  then show ?thesis
  proof (induct a)
    case emptyI
    then show ?case
      using cyl_zero zero_def by auto
  next
    case (insertI a dt)
    then show ?case
      using dt_in_SD dt_not_empty local.one_def one_closed by auto
  qed
qed
end
```

B.8.3 Domain Data View: Properties

DDV as model for a diagonal-free cylindric algebra

```
context domain_data_view
begin
interpretation dfcyl_algebra "(*)" "(+)" "(-)" "0" "1" "cyl" "A_A"
 apply unfold_locales

  apply (simp add: star_assoc)

  apply (simp add: star_comm)

  apply (simp add: plus_assoc)

  apply (simp add: plus_comm)
```

```
  apply (simp add: plus_distr_star)

  apply (simp add: star_distr_plus)

  apply (simp add: a_in_one0 star_def subset_antisym)

  apply (simp add: plus_zero_r)

  apply (simp add: complement_star)

  apply (simp add: complement_plus0)

  apply (simp add: cyl_zero)

  apply (simp add: cyl_mod)

  apply (simp add: cyl_comm)
  done
end

context domain_data_view
begin
end

end
```

The DIS is a novel data-centered framework for knowledge representation. It consists of a domain knowledge representation, called the domain ontology $\mathcal{O}$, a domain data representation, called the domain data view $\mathcal{A}$, and an operator τ that maps `A` to $\mathcal{O}$. To define the domain ontology, we use a monoid of concepts $\mathcal{C}$, a Boolean lattice of concepts $\mathcal{L}$, and a family of rooted graphs $\mathcal{G}$.

In a DIS, the concepts in the domain ontology originate from two sources: (i) the data, and (ii) the domain of application. The concepts in the domain ontology can be composed to form new concepts. We understand the composition as the Cartesian construction of concepts. Similar to how the information in a dataset record is not dependant on the order of attributes in a dataset, we require the composition operator to be commutative (up to an isomorphism), associative, and idempotent. The commutativity and associativy properties ensure that the order in which concepts are composed is not relevant. The idempotency property ensures that by composing a concept with itself the system does not create a new concept, or new knowledge from the existing knowledge in the system.

In current approaches to connect an ontology to existing data, mapping the two components to each other is a time consuming task, as it needs to bridge the conceptual gap between the two layers. To close this conceptual gap between the domain representation and the data view, in DIS, the core component of the domain ontology $\mathcal{O}$ is built using the Cartesian *partOf* relation. This is similar to the relation between elements of the data view. A concept with no sub-parts is called an *atomic* concept or an *atom*.

B.9 Concept

```
theory Concept
  imports
    Main
begin
```

The concept datatype is a set of atoms. The atoms are defined as a datatype on the concrete DIS implementation.

```
type_synonym 'a concept = "'a set"
```

B.9.1 Concept Operators: Definitions

With concepts as sets, the empty concept e_C is simply a notation for the empty set, the composition operator $\oplus_C$ a renaming of the set union operator, and the partial order $\sqsubseteq_C$ (or *partOf* a renaming of the set inclusion relation.

Definition 4.19

```
definition oplus :: "'a concept ⇒ 'a concept ⇒ 'a concept" (infixl "⊕C" 75)
  where "c1 ⊕C c2 = (c1 ∪ c2)"
```

Definition 4.20

```
definition zero :: "'a concept" ("eC")
  where "zero = {}"
```

Definition 4.21

```
definition partOf :: "'a concept ⇒ 'a concept ⇒ bool" (infixl "⊑C" 66)
  where "c1 ⊑C c2 = (c1 ⊆ c2)"
```

Within a given set *A* of concepts, atomicity is defined using the *partOf* relationship

```
definition isAtom :: "'a concept ⇒ 'a concept set ⇒ bool"
  where "isAtom a A ≡ ((a ∈ A) ∧ ¬(∃x. x ∈ A ∧ x ≠ a ∧ x ⊑C a))"

definition atomSet :: "'a concept set ⇒ bool"
```

```
  where "atomSet A = (∀ x ∈ A. isAtom x A)"
```

We give the definition of the closure of the $\oplus_C$ operator over a given set of concepts `A`, as an inductive set.

```
inductive_set
  oplus_closure :: "'a concept set ⇒ 'a concept set"
  for A :: "'a concept set"
  where
      a_in_A:  "a ∈ A ⟹ a ∈ oplus_closure A"
    | ab_in_closure: "a ∈ oplus_closure A ⟹ b ∈ oplus_closure A ⟹ a ⊕C
b ∈ oplus_closure A"
```

B.9.2 Concept Operators: Properties

Properties of composition operator

```
context
begin
```

We can prove some basic properties of the composition $\oplus_C$ operator, such as commutativity, associativity, and idempotency.

```
lemma oplus_comm: "c1 ⊕C c2 = c2 ⊕C c1"
  by (simp add: oplus_def sup_commute)

lemma oplus_assoc: "(c1 ⊕C c2) ⊕C c3 = c1 ⊕C (c2 ⊕C c3)"
  by (simp add: oplus_def sup_assoc)

lemma oplus_idempotent: "c ⊕C c = c"
  by (simp add: oplus_def)
end
```

Properties of empty concept

```
context
begin
```

We can prove that the empty concept e_C is a neutral element for the composition $\oplus_C$ operator, both to the right and the left.

```
lemma zero_r0: "⋀x. x ⊕C e_C = x"
  by (simp add: zero_def oplus_def)

lemma zero_l0: "⋀x. e_C ⊕C x = x"
  by (simp add: zero_def oplus_def)
end
```

end

B.10 Concept Monoid

To achieve the transformation from data to information and further to knowledge, we need to give meaning to the data, specifically to its sorts (or attributes). Thus we take the sorts of the dataset and consider them in a specific context. We call this context the domain of application for the data, and within it we capture the relevant concepts.

The ConceptMonoid theory describes the DIS monoid of concepts.

theory *ConceptMonoid*
  imports
    Concept
    "HOL-Algebra.Group"
begin

A set of concepts `C`, with $\oplus_C$ and e_C are syntactically equivalent to the *monoid* structure defined in HOL-Algebra.Group. The term "`C C`" is an abbreviation for the *monoid* structure.

abbreviation *plus_monoid :: "'a concept set ⇒ 'a concept monoid" ("C")*
  where *"plus_monoid C ≡ (|carrier = C, mult = (⊕C), one = eC|)"*

A set of atomic concepts (or atoms), together with the composition operator and the empty concept, form a monoid of concepts, described below with the *concept_monoid* locale. The set of atoms correspond to attributes of dataset(s) or to notions found in the domain of application, therefore it is assumed to be finite and non-empty. The carrier set `C` of the monoid is built as the $\oplus_C$ closure over the set of atoms `At`, which is the parameter of the *concept_monoid* locale. In addition, the empty concept e_C is assumed to be part of the carrier set.

locale *concept_monoid =*
  fixes *At :: "'a concept set"*
  assumes *atoms_not_empty: "At ≠ {}"*
  assumes *atoms_finite: "finite At"*

begin

B.10.1 Concept Monoid Operators: Definitions

The carrier set of the monoid of concepts is the closure of oplus operator over the set of atoms.

definition *C :: "'a concept set"*
  where *"C ≡ oplus_closure At ∪ {e$_C$}"*

end

B.10.2 Concept Monoid Operators: Properties

Properties of the composition operator

context *concept_monoid*
begin

We prove two basic properties of the monoid of concepts, that e_C is its neutral element, both to the left and right.

lemma *(in concept_monoid) zero_l: "⋀x. x ∈ C ⟹ e$_C$ ⊕$_C$ x = x"*
  by *(simp add: zero_def oplus_def)*

lemma *(in concept_monoid) zero_r: "⋀x. x ∈ C ⟹ x ⊕$_C$ e$_C$ = x"*
  by *(simp add: zero_r0)*

Within the *concept_monoid*, $\oplus_C$ is closed over the carrier set *C*.

lemma *(in concept_monoid) oplus_closed: "⋀x y. x ∈ C ⟹ y ∈ C ⟹ x ⊕$_C$ y ∈ C"*
  by *(metis C_def UnE UnI1 oplus_closure.intros(2) oplus_comm singletonD zero_r)*
end

B.10.3 Concept Monoid: Properties

Concept Monoid as a commutative monoid

context *concept_monoid*
begin

We show that the structure *"C C"* is a model of the *comm_monoid* structure, as described in HOL-Algebra.Group. Thus, all the theorems proved in the HOL commutative monoid theory are satisfied for the monoid of concepts of the DIS.

interpretation *comm_monoid "C C"*
  apply *unfold_locales*
  — Composition is closed over the carrier set

```
  apply (simp add: oplus_closed)
  — Composition is associative
  apply (simp add: oplus_assoc)
  — Empty concept is in the carrier set
  apply (simp add: C_def)
  — Empty concept is neutral over the composition
  apply (simp add: zero_l)
  apply (simp add: zero_r)
  — Composition is commutative
  apply (simp add: oplus_comm)
  done
end

end
```

B.11 Concept Lattice

The ConceptLattice theory describes the DIS Boolean lattice of concepts. We begin the construction of a new DIS from an existing dataset, which guides the design of the core component of the domain ontology (its Boolean lattice of concepts). The attributes in the dataset schema correspond one-to-one to the atoms of the Boolean lattice. We take a subset of these atoms and use the *partOf* relation to generate a free Boolean lattice over it. By the construction of the lattice, the task of mapping the data to the ontology becomes trivial, as the attributes of the data view schema have a one-to-one correspondence to the atoms of the Boolean lattice.

```
theory ConceptLattice
  imports
    Concept
    "HOL.Boolean_Algebras"
    "HOL-Algebra.Group"

begin
```

A lattice can be represented as a relational structure (a carrier set and a partial order with certain properties) or as an algebraic structure (a carrier set, two binary operators, meet and join, one unary operator, complement, and two constants, the neutral elements for the two binary operators, respectively). The two representations are equivalent, and in our work we use the terms *Boolean lattice* and *Boolean algebra* interchangeably.

The other two elements of the Boolean lattice, its unary operator (complement) and the $\otimes_C$ neutral element (top), can only be defined w.r.t. the carrier set of the lattice.

Thus, we need to separate the declaration of the *concept_lattice* into two locales. within the *concept_lattice0* we define the lattice operators and its carrier set, and within the *concept_lattice* we show it is a Boolean lattice (algebra).

B.11.1 Concept Lattice: the base

From a given set of atoms we can form a lattice of concepts. In it, the $\oplus_C$ and $\otimes_C$ correspond to the join and meet of a lattice, respectively. The empty concept e_C is its bottom, and its top will be defined as the composition of all the atoms. Similar to the monoid structure, the carrier set *L* of the lattice is built as the $\oplus_C$ closure over the atomic set of concepts *At*, given as the parameter of the *concept_lattice0* locale. In addition, the empty concept e_C is assumed to be part of the carrier set.

```
locale concept_lattice0 =
  — concept_lattice0 parameters, the set of lattice atoms
  fixes At :: "'a concept set"
  assumes atoms_not_empty: "At ≠ {}"
begin
```

Concept Lattice Operators: Definitions

```
definition L :: "'a concept set"
  where "L ≡ oplus_closure At ∪ {e_C}"
```

With concepts as sets, the composition operator dual $\otimes_C$ is the intersection of sets.

Definition 4.22

```
definition otimes :: "'a concept ⇒ 'a concept ⇒ 'a concept" (infixl "⊗_C"
65)
  where "c1 ⊗_C c2 = (c1 ∩ c2)"
```

Definition 4.23

```
definition topL :: "'a concept" ("⊤_L")
  where "⊤_L = ⋃{c. c∈ At}"
```

Definition 4.24

```
definition complL :: "'a concept ⇒ 'a concept" ("⊖_L _" [81] 80)
  where "⊖_L c = ⋃{u. u ∈ At ⟶ ¬(u ⊑_C c)}"

end
```

Concept Lattice Operators: Properties

```
context concept_lattice0
```

begin

Basic properties of $\otimes_C$: commutative, associative

lemma (**in** *concept_lattice0*) *otimes_comm*: "*c1* $\otimes_C$ *c2* = *c2* $\otimes_C$ *c1*"
  by (*simp add*: *otimes_def inf_commute*)

lemma (**in** *concept_lattice0*) *otimes_assoc*: "(*c1* $\otimes_C$ *c2*) $\otimes_C$ *c3* = *c1* $\otimes_C$ (*c2* $\otimes_C$ *c3*)"
  by (*simp add*: *otimes_def inf_assoc*)

The two binary operators distribute over each other

lemma (**in** *concept_lattice0*) *oplus_distr_otimes*: "*c1* $\oplus_C$ (*c2* $\otimes_C$ *c3*) = (*c1* $\oplus_C$ *c2*) $\otimes_C$ (*c1* $\oplus_C$ *c3*)"
  by (*simp add*: *oplus_def otimes_def sup_inf_distrib1*)

lemma (**in** *concept_lattice0*) *otimes_distr_oplus*: "*c1* $\otimes_C$ (*c2* $\oplus_C$ *c3*) = (*c1* $\otimes_C$ *c2*) $\oplus_C$ (*c1* $\otimes_C$ *c3*)"
  by (*simp add*: *otimes_def oplus_def inf_sup_distrib1*)
end

B.11.2 Concept Lattice as a Boolean algebra

locale *concept_lattice* = *concept_lattice0* +
  assumes *top_r*: "$\bigwedge$*c*. *c* $\otimes_C$ $\top_L$ = *c*"
    and *top_inL*: "$\top_L$ $\in$ *L*"
    and *otimes_closed_L*: "$\bigwedge$*x y*. *x* $\in$ *L* $\Longrightarrow$ *y* $\in$ *L* $\Longrightarrow$ *x* $\otimes_C$ *y* $\in$ *L*"
    and *otimes_cancel_r*: "$\bigwedge$*x*. *x* $\otimes_C$ $\ominus_L$ *x* = e_C"
    and *oplus_cancel_r*: "$\bigwedge$*x*. *x* $\oplus_C$ $\ominus_L$ *x* = $\top_L$"
begin
end

Concept Lattice Operators: Properties

context *concept_lattice*
begin

The basic properties of e_C within the lattice of concepts: neutral to the left and right.

lemma (**in** *concept_lattice*) *zero_inL*: "e_C $\in$ *L*"
  by (*simp add*: *L_def*)

lemma (**in** *concept_lattice*) *zero_l*: "$[\![$*c* $\in$ *L*$]\!]$ $\Longrightarrow$ e_C $\oplus_C$ *c* = *c*"
  by (*simp add*: *zero_def oplus_def*)

Within the *concept_lattice*, $\oplus_C$ is closed over the carrier set *L*.

```
lemma (in concept_lattice) oplus_closed_L: "⋀x y. x ∈ L ⟹ y ∈ L ⟹ x ⊕C y ∈ L"
  by (metis L_def UnE UnI1 concept_lattice.zero_l concept_lattice_axioms oplus_closure.intros(2) singleton_iff zero_r0)
```

Proprties of the *partOf* relationship in the context of *concept_lattice*.

```
lemma (in concept_lattice) partOf_trans:
    "⋀x y z. x ∈ L ⟹ y ∈ L ⟹ z ∈ L ⟹ x ⊑C y ⟹ y ⊑C z ⟹ x ⊑C z"
  by (simp add: partOf_def)

lemma (in concept_lattice) partOf_oplus: "⋀x y. x ∈ L ⟹ y ∈ L ⟹ x ⊑C (x ⊕C y)"
  unfolding partOf_def oplus_def
  by auto

lemma (in concept_lattice) partOf_times: "⋀x y. x ∈ L ⟹ y ∈ L ⟹ (x ⊗C y) ⊑C x"
  unfolding partOf_def otimes_def
  by auto
```

The basic property of $\top_L$ within the lattice of concepts: neutral to the left.

```
lemma (in concept_lattice) top_l: "⟦c ∈ L⟧ ⟹ ⊤L ⊗C c = c"
  by (simp add: top_r otimes_comm)

end
```

Concept Lattice as a dual abelian mondoid

```
context concept_lattice
begin
```

The lattice is a dual abelian monoid: one over composition and empty concept, and one over their dual operators, $\otimes_C$ and $\top_L$.

```
interpretation plus_monoidL: comm_monoid "(|carrier = L, mult = (⊕C), one = eC|)"
  apply unfold_locales
  — Composition is closed over the carrier set
  apply (simp add: oplus_closed_L)
  — Composition is associative
  apply (simp add: oplus_assoc)
  — Empty concept is in the carrier set
  apply (simp add: zero_inL)
```

— Empty concept is neutral over the composition
apply *(simp add: zero_l)*
apply *(simp add: zero_r0)*
— Composition is commutative
apply *(simp add: oplus_comm)*
done

interpretation *mult_monoidL: comm_monoid "(|carrier = L, mult = (⊗C), one = ⊤L|)"*
apply *unfold_locales*
— Composition is closed over the carrier set
apply *(simp add: otimes_closed_L)*
— Composition is associative
apply *(simp add: otimes_assoc)*
— Empty concept is in the carrier set
apply *(simp add: top_inL)*
— Empty concept is neutral over the composition
apply *(simp add: top_l)*
apply *(simp add: top_r)*
— Composition is commutative
apply *(simp add: otimes_comm)*
done

end

Concept Lattice as a Boolean algebra

context *concept_lattice*
begin

In addition, *L* together with its two binary operators, the unary operator, and the two neutral elements is a Boolean algebra, or it is a model for the *abstract_boolean_algebra*, as described in *HOL.Boolean_Algebras*.

interpretation *abstract_boolean_algebra "(⊗C)" "(⊕C)" complL eC "⊤L"*
apply *unfold_locales*
— $\otimes_C$ and $\oplus_C$ properties: associative and commutative
apply *(simp add: otimes_assoc)*
apply *(simp add: otimes_comm)*
apply *(simp add: oplus_assoc)*
apply *(simp add: oplus_comm)*
— $\otimes_C$ and $\oplus_C$ distribute over each other
apply *(simp add: otimes_distr_oplus)*
apply *(simp add: oplus_distr_otimes)*

```
  — Top and bottom (zero) are neutral elements
  apply (simp add: top_r)
  apply (simp add: zero_r0)
  — Complement cancels over ⊗C and ⊕C
  apply (simp add: otimes_cancel_r)
  apply (simp add: oplus_cancel_r)
  done

end

end
```

B.12 Concept Rooted Graph

Recall that in a DIS, we start by considering the sorts of the dataset to be a subset of the concepts of the domain.We then ask domain experts to capture other concepts that are deemed significant to the domain of application and that are not part of the Boolean lattice. These concepts are captured from the domain of knowledge or from an existing ontology as rooted graphs. A rooted graph is an acyclick directed graph, with a distinguished element, the root. The root is a unique concept (vertice) of the graph which is reachable from any other vertice.

Theory ConceptRootGraph describes such a rooted graph, using the directed graph theory Digraph authored by Noschinski and Neumann.

```
theory ConceptRootGraph
  imports
    Concept
    Digraph
    "HOL.Transitive_Closure"
begin
```

B.12.1 Concept Rooted Graph: Definitions

In DIS, we need to declare a family of rooted graphs. As such, the rooted graphs are defined with the use of an Isabelle/HOL `record`. A record is a complex datatype, defined as an extensible n-tuple. The record's fields (components) have names by which they can be accessed. Records and their fields make expressions easier to read and reduce the risk of confusing one field for another. As a complex datatype, one can declare a set of records (unlike a "set of locales").

The record `rgraph` has three components: the vertices `V`, a set of concepts, the edges `R`, a set of pairs of concepts, and the root `t`, a concept. All three components are

indexable, and marked as such with the ı symbol. This will allow one to declare two graphs, *G* and *H*, and use the terms V_G, V_H to refer to the vertices components on the graph *G*, respectively *H*. With this understanding, we define two operators, tail and head, to mimic the *pre_digraph* record components. The operator tail gives access to the tail of a given edge (or the first element of the edge pair), while the operator head gives access to the head of it (or the 2nd element of the edge pair).

```
type_synonym 'a edge = "'a concept × 'a concept"

record 'a rgraph =
  vertices :: "'a concept set" ("V⇩ı")
  edges :: "'a edge set" ("R⇩ı")
  root :: "'a concept" ("t⇩ı")

definition tail :: "'a edge ⇒ 'a concept"
  where "tail e = fst e"

definition head :: "'a edge ⇒ 'a concept"
  where "head e = snd e"
```

— Pre-defined relation (edge) properties: transitivity, reflexitivity, and reflexive-transitivity. More properties can be defined in extensions of the ConceptRootGraph.

```
definition is_trans :: "'a edge set ⇒ bool"
  where "is_trans E ≡ trancl E = E"

definition is_refl :: "'a edge set ⇒ bool"
  where "is_refl E ≡ reflcl E = E"

definition is_refltr :: "'a edge set ⇒ bool"
  where "is_refltr E ≡ rtrancl E = E"
```

The *vertices* and *edges* component of the record *rgraph* along with the two operators *tail* and *head* are syntactically equivalent to the *pre_digraph* record. The term "$\mathcal{G}_r$ *G*" is an abbreviation for the *pre_digraph* record.

```
abbreviation digraph :: "'a rgraph ⇒ ('a concept,'a edge) pre_digraph"
("𝒢⇩r")
  where "digraph G  ≡ (|verts = V⇩G, arcs = R⇩G, tail = tail, head = head |)"
```

B.12.2 Concept Rooted Graph

A rooted graph is an acyclic directed graph that has a unique element (the root), reachable from every vertice in the graph. It is defined using the pre-existing *loopfree_digraph* locale. It has only one parameter, the *rgraph* G_r structure.

```
locale concept_root_graph =
 dag: loopfree_digraph where G = "𝒢_r G_r"
for G_r :: "'a rgraph"  (structure) +
  assumes
        G_def: "G = 𝒢_r G_r"
    and lf_graph: "loopfree_digraph G"
    and R_def: "R ⊆ (C × C)"
    and root_in_C: "t ∈ C"
    and root_reachable: "⋀c. c ∈ C ⟶ c →*_G t"
    and root_unique: "⋀c. c →*_G t' ⟶ t = t'"
begin
print_locale! concept_root_graph
end
```

Concept Rooted Graphs: Properties

The concept rooted graph is acyclic

```
lemma (in concept_root_graph) loop_free: "(c1, c2) ∈ R ⟹ c1 ≠ c2"
  using G_def by auto
```

B.12.3 Family of Concept Rooted Graphs

The family of rooted graphs is simply a set of *rgraph* records, where each such a record is a *concept_rooted_graph* structure.

```
locale family_root_graphs =
  fixes 𝒢 :: "'a rgraph set"
  assumes 𝒢_def: "⋀ g. g ∈ 𝒢 ⟶ concept_root_graph g"

end
```

B.13 Domain Ontology

Recall that the Domain Ontology component of a DIS is formed of a monoid of concepts, a Boolean lattice (in which the atoms correspond to the attributes of the dataset under consideration), and a family of graphs all rooted in the Boolean lattice. The *DomainOnt* theory encompasses this mathematical structure.

```
theory DomainOnt
  imports
    ConceptMonoid
    ConceptLattice
    ConceptRootGraph
```

begin

The *domain_ontology* instantiates three parameters: (i) the concepts monoid given by its set of atoms A_C, (ii) the concepts Boolean lattice *concept_lattice*, given by its sets of atoms A_L, and (iii) the family (set) of rooted graphs. We impose the DIS Domain Ontology axioms: the set of lattice atoms is a subset of the monoid atoms, the roots of each graph are in the lattice, and the vertices of each graph are a subset of the monoid carrier set.

```
locale domain_ontology =
  cmonoid: concept_monoid where At = A_C +
  clattice: concept_lattice where At = A_L +
  cgraphs: family_root_graphs where G = G

 for   A_C :: "'a concept set"
   and A_L :: "'a concept set"
   and G ::  "'a rgraph set" +
assumes
      atomsL_in_C: "A_L ⊆ A_C"
  and graph_roots_in_L: "⋀ g. g ∈ G ⟶ t_g ∈ clattice.L"
  and graph_carriers_in_C: "⋀ g. g ∈ G ⟶ V_g ⊆ cmonoid.C"
begin
print_locale! domain_ontology
end

end
```

B.14 Domain Information System

In this section, the Domain Information System (DIS) is put together using the DomainOnt and DomainDtVw components

```
theory DomainInfSys
  imports
    DomainDtVw
    DomainOnt
begin

locale dis =
  ddv: domain_data_view where U = U and sort2name = sort2name and
name2sort = name2sort and D = D +
  dont: domain_ontology where A_C = A_C and A_L = A_L and G = G
 for   A_C :: "'a concept set"
```

```
  and A_L :: "'a concept set"
  and 𝒢 ::  "'a rgraph set"
  and 𝒰 :: "sort set"
  and sort2name :: "sort ⇒ string"
  and name2sort :: "string ⇒ sort"
  and D :: "sdatum set" +
 fixes sort2atom :: "sort ⇒ 'a" ("η_" [55])
 assumes sort2atom_inj: "η x = η y ⟹ x = y"
begin

end
```

B.14.1 Mapping Operator: Definitions

```
context dis
begin

definition (in dis) sval_sort :: "svalue ⇒ sort" ("τ_v_" [80]) where
  "τ_v sv ≡ name2sort (sname sv)"
```

Definition 4.25

```
definition (in dis) type :: "sdata ⇒ 'a concept" ("τ_" [80]) where
  "τ a = {η (τ_v sv) | sv . sv ∈ ⋃{dt. dt ∈ a}}"


definition (in dis) dtc_type :: "dataconcept ⇒ 'a concept" ("τ_dc_" [80])
where
  "τ_dc c = ⋃{τ a| a. a ∈ c}"
```

Syntactic sugar: notation for the top of the Boolean lattice

```
definition (in dis) top_lattice :: "'a concept"
  where "top_lattice = dont.clattice.topL"
notation top_lattice ("⊤_L")

end
```

B.14.2 Mapping Operator: Preserving Properties

```
context dis
begin
```

Axiom 4.26

Axiom 4.26: Preserving the composition operator

```
lemma (in dis) type_plus: "τ (a+b) = τ a ⊕C τ b"
proof -
  have "τ (a+b) = {η (τv sv) | sv . sv ∈ ⋃{dt. dt ∈ (a+b)}}"
    by (simp add: type_def)
  moreover have "{η (τv sv) | sv . sv ∈ ⋃{dt. dt ∈ (a+b)}} =
                 {η (τv sv) | sv . sv ∈ ⋃{dt. dt ∈ (a∪b)}}"
    by (simp add: ddv.plus_def)
  moreover have "{η (τv sv) | sv . sv ∈ ⋃{dt. dt ∈ (a∪b)}} =
                 {η (τv sv) | sv . sv ∈ ⋃{dt. dt ∈ a} ∪ ⋃{dt. dt ∈ b}}"
    by auto
  moreover have "{η (τv sv) | sv . sv ∈ ⋃{dt. dt ∈ a} ∪ ⋃{dt. dt ∈ b}} =

                 {η (τv sv) | sv . sv ∈ ⋃{dt. dt ∈ a}} ∪
                 {η (τv sv) | sv . sv ∈ ⋃{dt. dt ∈ b}}"
    using set_image_union by (metis Union_Un_distrib)
  moreover have "{η (τv sv) | sv . sv ∈ ⋃{dt. dt ∈ a}} ∪
                 {η (τv sv) | sv . sv ∈ ⋃{dt. dt ∈ b}} = (τ a) ∪ (τ b)"
    by (simp add: type_def)
  moreover have "(τ a) ∪ (τ b) = τ a ⊕C τ b"
    by (simp add: oplus_def)
  ultimately show ?thesis
    by auto
qed
```

Axiom 4.27

Axiom 4.27: Preserving the zero

```
lemma (in dis) type_zero: "τ 0 = eC"
  by (simp add: type_def ddv.zero_def Concept.zero_def)
```

Axiom 4.28

Axiom 4.28: Preserving the one

```
lemma (in dis) type_one: "τ 1 = ⊤L"
proof -
  have f1: "τ 1 = {η (τv sv) |sv. sv ∈ ⋃{dt. dt ∈ 1}}"
    by (simp add: type_def)
  have f2: "{η (τv sv) |sv. sv ∈ ⋃{dt. dt ∈ 1}} =
            {η (τv sv) |sv. sv ∈ ⋃{dt. dt ∈ ⋈𝒰}}"
    by (simp add: ddv.one_def ddv.SD_def)
  then show ?thesis
    using dont.clattice.topL_def top_lattice_def ddv.one_def type_def
```

```
    by (meson Set.set_insert ddv.dt_in_SD ddv.dt_v_in_k insertI1)
qed

end
```

B.14.3 Mapping Operator: Properties

Properties of mapping operator: helpers

```
context dis
begin

lemma (in dis) type_mono:
  assumes "a ⊆ b"
  shows "τ a ⊆ τ b"
proof -
  have f1: "a ∪ b = b"
    using assms by auto
  have f2: "τ (a ∪ b) = τ a ∪ τ b"
    using type_plus ddv.plus_def oplus_def
    by metis
  then show ?thesis
    using assms f1 by auto
qed

lemma (in dis) k_in_a_in_type:
  assumes "k ∈a a" "a ∈ A_A"
    shows "η k ∈ τ a"
proof -
  have "a ∈ Fin"
    using assms ddv.a_in_SD ddv.a_fin Fin_is_finite by auto
  then show ?thesis
  proof -
    obtain dt::sdatum where exDt: "dt ∈ a ∧ k ∈ dt"
      using assms ddv.k_in_a_def ddv.k_in_dt_def
      by auto
    obtain v::dtype where exVDt: "⟪k, v⟫ ∈ dt"
      using exDt ddv.k_in_dt_def by auto
    obtain sv::svalue where exSv: "sv = ⟪k, v⟫"
      by auto
    have f1: "η k = η (τ_v sv)"
      using ddv.SD_def ddv.bow_zero_mono ddv.dt_not_empty ddv.one_closed
ddv.one_def by auto
```

```
    have f2: "η (τv sv) ∈ {η (τv sv)| sv. sv ∈ ⋃{dt| dt. dt ∈ a}}"
      using exDt exSv exVDt by blast
    then show ?thesis
      using f1 f2 type_def by auto
  qed
qed
end
```

Proposition 4.6.1

```
context dis
begin
```

Proposition 4.6.1 Equation 1

```
lemma (in dis) k_in_akextension:
  assumes "a ∈ A_A" "k ∈ 𝒰" "∃dt∈a. k ∈ dt"
  shows "{(η k)} ⊑C τ (a⇑k)"
proof -
    obtain dt::sdatum where exDt: "dt ∈ a ∧ k ∈ dt"
      using assms  ddv.k_in_dt_def
      by auto
    obtain v::dtype where exVDt: "⟪k, v⟫ ∈ dt"
      using exDt ddv.k_in_dt_def by auto
    have f0: "dt ∈ a"
      using exDt by auto
  have f1: "(dt↑k) ⊆ (a⇑k)"
    using f0 ddv.dt_in_extenda by auto
  have f2: "τ (dt↑k) = {η (τv sv) | sv . sv ∈ ⋃{dt'. dt' ∈ (dt↑k)}}"
    using type_def by auto
  have f3: "⟪k, v⟫ ∈ ⋃{dt'. dt' ∈ (dt↑k)}"
    using exVDt ddv.a_in_SD ddv.dt_not_empty ddv.one_closed ddv.one_def by
blast
  have f4: "(η k) = η (τv ⟪k, v⟫)"
    using assms ddv.dt_in_SD ddv.dt_v_in_k by blast
  have f5: "(η k) ∈ τ (dt↑k)"
    using f2 f3 f4 by blast
  have f6: "τ (dt↑k) ⊆ τ (a⇑k)"
    using f1 type_def type_mono by auto
  then show ?thesis
    using f1 f5 f6 partOf_def by fastforce
qed
```

Proposition 4.6.1 Equation 2

```
lemma (in dis) k_notin_kreduction:
```

```
  assumes "k ∈ 𝒰" "a ∈ A_A"
  shows "(η k) ∉ τ (a⇓_k)"
proof -
  have f: "τ (a⇓_k) = {η (τ_v sv)| sv. sv ∈ ⋃{dt↓_k| dt. dt ∈ a}}"
    using type_def ddv.reduce_def by auto
  have "¬(∃ v. ⟪k, v⟫ ∈ ⋃{dt↓_k| dt. dt ∈ a})"
    using assms ddv.kv_nin_reduce ddv.k_in_dt_def ddv.k_nin_dt_def
          ddv.dt_in_SD ddv.dt_not_empty ddv.one_closed ddv.one_def
    by metis
  then have "k ∉ {(τ_v sv)| sv. sv ∈ ⋃{dt↓_k| dt. dt ∈ a}}"
    using sval_sort_def ddv.sort2name2sort ddv.sort_neq_def
          ddv.reduce_sd_def by force
  then have "(η k) ∉ {η (τ_v sv)| sv. sv ∈ ⋃{dt↓_k| dt. dt ∈ a}}"
    using assms sort2atom_inj by blast
  then show ?thesis
    using f by auto
qed
end
```

Proposition 4.6.2

```
context dis
begin
```

Proposition 4.6.2 Equation 1

```
lemma (in dis) type_prod_zero:
  assumes "a ∈ A_A" "b ∈ A_A" "τ a ⊗_C τ b =  e_C"
  shows "τ (a * b) = e_C"
proof -
  have "a * b = 0"
  proof (rule ccontr)
    assume ab_zero: "a * b ≠ 0"
    obtain dt::sdatum where exDt: "dt = (SOME dt. dt ∈ a * b)"
      using ab_zero mult_def ddv.zero_def ddv.dt_not_empty by auto
    have f1: "dt ≠ {}"
      using exDt ddv.dt_not_empty ddv.star_closed ddv.a_in_SD
ddv.carrier_set_def by auto
    obtain sv::svalue where exSv: "sv = (SOME sv. sv ∈ dt)"
      using f1 by auto
    obtain k::sort where letK: "k = τ_v sv"
      using exSv by auto
    have f2: "k ∈_a a ∧ k ∈_a b"
      using assms exSv ddv.k_in_a_def ddv.k_in_dt_def ddv.dt_in_SD
```

```
ddv.dt_not_empty
      by (metis ddv.one_closed ddv.one_def)
    have f3: "η k ∈ τ a ∧ η k ∈ τ b"
      using f2 k_in_a_in_type assms by auto
    have f4: "η k ∈ τ a ⊗C τ b"
      using f3 dont.clattice.otimes_def by fastforce
    have f5: "τ a ⊗C τ b ≠ eC"
      using f4 zero_def by auto
    with <τ a ⊗C τ b = eC>  show False by auto
  qed
  then show ?thesis
    by (simp add: assms type_zero)
qed
```

Proposition 4.6.2 Equation 2

```
lemma (in dis) type_prod_weak:
  assumes "a ∈ AA" "b ∈ AA"
    shows "τ (a * b) ⊑C (τ a ⊗C τ b)"
  by (simp add: ddv.star_def dont.clattice.otimes_def partOf_def type_mono)
```

Proposition 4.6.2 Equation 3

```
lemma (in dis) type_on_bow:
  assumes "a ∈ AA" "T ⊆ 𝒰" "a = ⋈T"
  shows "τ a = {η s | s. s ∈ T}"
  using assms ddv.bow_zero_mono ddv.dt_not_empty by blast
```

Proposition 4.6.2 Equation 4

```
lemma (in dis) type_preserve_prod:
  assumes "a ∈ AA" "b ∈ AA" "Ta ⊆ 𝒰" "Tb ⊆ 𝒰" "a = ⋈Ta" "b = ⋈Tb"
    shows " τ(a * b) = τ a ⊗C τ b"
  using assms ddv.bow_zero_mono ddv.dt_not_empty by blast

end

end
```

B.15 Wine Universe

A model of the Universe theory,the Wine Universe is implemented as an instantiation of the *universe* locale. Foundational elements, such as the sorts and their mappings are defined in this section.

```
theory WineUniv
```

imports *DDVUniverse*

begin

The sorts of the universe, given as finite sets. They can be imported using the templates.

definition S_W :: *sort* **where**
"S_W = {Str ''Merlot'', Str ''Chardonay'', Str ''Vidal Blanc'', Str ''Magliocco''}"

definition S_C :: *sort* **where**
"S_C = {Str ''Red'', Str ''White'', Str ''Rose''}"

definition S_S :: *sort* **where**
"S_S = {Str ''Dry'', Str ''Sweet''}"

definition S_B :: *sort* **where**
"S_B = {Str ''Full'', Str ''Medium'', Str ''Fruity''}"

definition $\mathcal{U}$ :: "sort set" **where**
"$\mathcal{U}$ = {S_W, S_C, S_S, S_B}"

The mappings (helper operators), bijections between sorts and their names.

definition *wine_sort_to_name :: "sort ⇒ string" ("s2n _" [99] 100)* **where**
"(s2n s) = (if s = S_W then ''wine'' else
(if s = S_C then ''col'' else
(if s = S_S then ''sugar'' else
(if s = S_B then ''body'' else undefined))))"

definition *wine_name_to_sort :: "string ⇒ sort" ("n2s _" [99] 100)* **where**
"(n2s n) = (if n = ''wine'' then S_W else
(if n = ''col'' then S_C else
(if n = ''sugar'' then S_S else
(if n = ''body'' then S_B else undefined))))"

The instantiation of the *universe* locale with the given universe and its mappings

locale *wine_univ =*
wine_univ: universe **where** $\mathcal{U}$ = $\mathcal{U}$ **and** *sort2name = sort2name* **and** *name2sort = name2sort*
for $\mathcal{U}$ :: "sort set"
and *sort2name :: "sort ⇒ string"*
and *name2sort :: "string ⇒ sort" +*
assumes *univ_def: "$\mathcal{U}$ = WineUniv.$\mathcal{U}$"*

```
    and s2n_def: "sort2name s = s2n s"
    and n2s_def: "name2sort n = n2s n"
begin
end

context wine_univ
begin
```

Optional, a list of svalues that will be used later in reasoning. Syntactic sugar, for ease of use only.

```
  definition merlot :: svalue where "merlot = ⟪S_W, Str ''Merlot''⟫"
  definition chard :: svalue where "chard = ⟪S_W, Str ''Chardonay''⟫"
  definition vidal :: svalue where "vidal = ⟪S_W, Str ''Vidal Blanc''⟫"
  definition magliocco :: svalue where "magliocco = ⟪S_W, Str
''Magliocco''⟫"

  definition red :: svalue where "red = ⟪S_C, Str ''Red''⟫"
  definition white :: svalue where "white = ⟪S_C, Str ''White''⟫"
  definition rose :: svalue where "rose = ⟪S_C, Str ''Rose''⟫"

  definition dry :: svalue where "dry = ⟪S_S, Str ''Dry''⟫"
  definition sweet :: svalue where "sweet = ⟪S_S, Str ''Sweet''⟫"

  definition medium :: svalue where "medium = ⟪S_B, Str ''Medium''⟫"
  definition full :: svalue where "full = ⟪S_B, Str ''Full''⟫"
  definition fruity :: svalue where "fruity = ⟪S_B, Str ''Fruity''⟫"
end

end
```

B.16 Wine Domain Data View

A model of the Domain Data View theory,the WineDDV is implemented as an instantiation of the *domain_data_view* locale.

```
theory WineDDV
  imports
    WineUniv
    DomainDtVw

begin
```

The dataset that guides the construction of the entire DIS is part of the declarations, given as the definition of the generator set.

```
context wine_univ
begin

definition wine_generator :: "sdatum set" where
  "wine_generator = {{merlot, red, dry, full},
                     {chard, white, dry, medium},
                     {vidal, rose, sweet, fruity},
                     {magliocco, red, dry, full}}"
end

locale wine_ddv = wine_univ +
  wine_ddv: domain_data_view where  D = D
  for D :: "sdatum set"  +
 assumes D_def: "D = wine_generator"
begin
end

context wine_ddv
begin

lemma (in wine_ddv) "{merlot, red, dry, full} ∈ D"
  using D_def wine_generator_def by simp

lemma (in wine_ddv) "{merlot, red, dry, full} ∈ Set.filter (λx. merlot ∈ x)
D"
  using D_def wine_generator_def by simp

end

end
```

B.17 Wine Domain Ontology

In the Wine Domain Ontology section, we implement the Domain Ontology theory. We start from the dataset defined in the Domain Data View section, with the attributes *grape*, *sugar*, *body*, and *colour*. In addition, other (atomic) concepts from the domain of application, such as *producer* and *region* are captured by the domain experts. The process of capturing the dataset attributes can be automated. The other concepts in the domain of application are to be defined manually by the ontology engineer(s), with assistance from the domain experts.

```
theory WineDOnt
```

imports *DomainOnt*
begin

— The abstract datatype for the Wine Ontology is $wine_{at}$, a collection of all atoms of the ontology.

datatype *wine* = $grape_{at}$ | $sugar_{at}$ | $body_{at}$ | $colour_{at}$ | $producer_{at}$ | $region_{at}$ | $estate_{at}$

— The atoms of the *concept_lattice*:

definition *grape :: "wine concept"*
where *"grape = {*$grape_{at}$*}"*

definition *sugar :: "wine concept"*
where *"sugar = {*$sugar_{at}$*}"*

definition *body :: "wine concept"*
where *"body = {*$body_{at}$*}"*

definition *colour :: "wine concept"*
where *"colour = {*$colour_{at}$*}"*

definition *wine :: "wine concept"*
where *"wine = grape* $\oplus_C$ *sugar* $\oplus_C$ *body* $\oplus_C$ *colour"*

— The other concepts in the domain of application, atoms of the *concept_monoid*

definition *producer :: "wine concept"*
where *"producer = {*$producer_{at}$*}"*

definition *region :: "wine concept"*
where *"region = {*$region_{at}$*}"*

definition *estate :: "wine concept"*
where *"estate = {*$estate_{at}$*}"*

definition A_L *:: "wine concept set"* **where**
"A_L *= {grape, sugar, body, colour}"*

definition *non_lattice_atoms :: "wine concept set"*
where *"non_lattice_atoms = {producer, region, estate}"*

```
definition A_C :: "wine concept set"
  where "A_C = A_L ∪ non_lattice_atoms"
```

The Wine Ontology *wine_dont* instantiates the *domain_ontology* structure, taking two parameters: A_C and A_L, the sets of atoms over which the *concept_monoid*, respectively *concept_lattice* are built. In addition, it instantiates the *concept_rooted_graph* structure as a graph rooted at *grape*.

```
definition V_r :: "wine concept set"
  where "V_r = {producer, region, estate}"

definition R_r :: "wine edge set"
  where "R_r = {(region, producer), (producer, estate), (estate, grape)}"

definition G_r :: "wine rgraph"
  where "G_r = (|vertices = V_r, edges = R_r, root = grape |)"

definition  𝒢 ::  "wine rgraph set"
  where "𝒢 = {G_r}"

locale wine_dont =
  dont: domain_ontology where A_C = A_C and A_L = A_L and 𝒢 = 𝒢
  for  A_C :: "wine concept set"
   and A_L :: "wine concept set"
   and 𝒢 ::  "wine rgraph set" +
assumes
      atom_l_def: "A_L = WineDOnt.A_L"
  and atom_c_def: "A_C = WineDOnt.A_C"
  and graphs_def: "𝒢 = WineDOnt.𝒢"
  and r_trans: "is_trans R_r"
begin
print_locale! wine_dont
end
```

Simple results: to show an element is not part of the set of concepts A_L, we show it is distinct from all the elements in that set

```
context wine_dont
begin
lemma (in wine_dont) gc: "grape ∈ A_C"
  using A_L_def dont.atomsL_in_C atom_l_def by auto
lemma (in wine_dont) producer_in_ac: "producer ∈ A_C"
  by (simp add: V_r_def A_C_def atom_c_def non_lattice_atoms_def)
lemma (in wine_dont) producer_neq_name: "producer ≠ grape"
  by (simp add: producer_def grape_def)
```

```
lemma (in wine_dont) producer_neq_sugar: "producer ≠ sugar"
  by (simp add: producer_def sugar_def)
lemma (in wine_dont) el: "producer ∉ A_L"
  by (simp add: A_L_def atom_l_def producer_def grape_def sugar_def body_def
colour_def)
end

context wine_dont
begin
```

Composition of *grape* and *sugar* is not an element of the set of atoms.

```
lemma (in wine_dont) l0: "sugar_at ∈ (grape ⊕_C sugar)"
  by (simp add: oplus_def sugar_def)

lemma (in wine_dont) l1: "(grape ⊕_C sugar) ∉ A_L"
proof -
  have f1: "grape ≠ (grape ⊕_C sugar)"
    by (simp add: oplus_def sugar_def grape_def)
  have f2: "sugar ≠ (grape ⊕_C sugar)"
    by (simp add: oplus_def sugar_def grape_def)
  have f3: "body ≠ (grape ⊕_C sugar)"
    using l0 body_def by blast
  then have "colour ≠ (grape ⊕_C sugar)"
    using l0 colour_def by blast
  then show ?thesis
    using A_L_def atom_l_def grape_def oplus_def sugar_def body_def
colour_def
          f1 f2 f3 by blast
qed

lemma (in wine_dont) l2: "(grape ⊕_C sugar) ∉ A_C"
proof -
  have f1: "sugar_at ∉ producer"
    by (simp add: producer_def)
  have f2: "sugar_at ∉ region"
    by (simp add: region_def)
  have f: "sugar_at ∉ estate"
    by (simp add: estate_def)
  then show ?thesis
    using V_r_def l0 f1 f2 l1 A_C_def atom_l_def atom_c_def
non_lattice_atoms_def insertE by auto
qed
```

Composition of *grape* and *sugar* is an element of the monoid carrier set.

```
lemma (in wine_dont) l3: "(grape ⊕C sugar) ∈ dont.cmonoid.C"
  using dont.graph_carriers_in_C G_def graphs_def by blast

lemma (in wine_dont) l4: "(grape ⊕C sugar) ∈ dont.clattice.L"
  by (simp add: dont.graph_roots_in_L  G_def graphs_def)
```

There are elements in the monoid carrier set that are not elements of the atoms set.

```
lemma (in wine_dont) l5: "∃ a. a ∈ dont.cmonoid.C ∧ a ∉ A_C"
  using l2 l3 by auto
```

More results: concept *grape* and *sugar* are not *partOf* each other

```
lemma (in wine_dont) gr_notpart_sugar: "¬ (grape ⊑C sugar)"
  by (simp add: grape_def partOf_def sugar_def)
lemma (in wine_dont) gr_notpart_body: "¬ (grape ⊑C body)"
  by (simp add: grape_def partOf_def body_def)
lemma (in wine_dont) gr_notpart_colour: "¬ (grape ⊑C colour)"
  by (simp add: grape_def partOf_def colour_def)
lemma (in wine_dont) sugar_notpart_name: "¬ (sugar ⊑C grape)"
  by (simp add: grape_def partOf_def sugar_def)
lemma (in wine_dont) body_notpart_name: "¬ (body ⊑C grape)"
  by (simp add: grape_def partOf_def body_def)
lemma (in wine_dont) colour_notpart_name: "¬ (colour ⊑C grape)"
  by (simp add: grape_def partOf_def colour_def)
```

Concept *grape* is an atom of the lattice set of atoms A_L.

```
lemma (in wine_dont) "isAtom grape A_L"
  using grape_def isAtom_def  A_L_def
        sugar_notpart_name body_notpart_name colour_notpart_name
  using atom_l_def empty_iff insert_iff by blast
```

However, the composite concept *grape* $\oplus_C$ *sugar* is not an atom of the lattice atoms; it is not even a member of that set, as demonstrated in lemma b2.

```
lemma (in wine_dont) "¬ (isAtom (grape ⊕C sugar) A_L)"
  using l1 l2 isAtom_def by auto
end

end
```

B.18 Wine Domain Information System

The Wine DIS is where everything comes together, linking the Wine DDV to the Wine DOnt.

```
theory WineDIS
  imports
    DomainInfSys
    WineDDV
    WineDOnt
    "HOL-Eisbach.Eisbach"

begin

locale wine_dis = wine_ddv + wine_dont +
  fixes sort2atom :: "sort ⇒ wine"
  assumes s2a_def: "sort2atom s = (if s = S_W then grape_at else
                                   (if s = S_C then colour_at else
                                    (if s = S_B then body_at else
                                     (if s = S_S then sugar_at else
undefined))))"
    and s2a_inj: "sort2atom x = sort2atom y ⟹ x = y"
begin



interpretation dis A_C A_L 𝒢 𝒰 sort2name name2sort D sort2atom
  apply unfold_locales
  apply (simp add: s2a_inj)
  done
```

The wine mapping operator is defined by overwriting the DIS mapping operator.

```
definition wine_type :: "sdata ⇒ wine set" ("τ_w _" [99] 100) where
  "τ_w a = τ a"
```

Definitions for .

```
definition (in wine_dis) filter0 :: "svalue ⇒ sdatum set ⇒ sdata" where
  "filter0 sv X = Set.filter (λx. sv ∈ x) X"

definition (in wine_dis) reds :: "sdata" where "reds = filter0 red D"
definition (in wine_dis) whites :: sdata where "whites = filter0 white D"

definition (in wine_dis) red_grapes :: "sdatum" where
  "red_grapes = Set.filter(λw. sname w = sort2name S_W) (⋃{x. x ∈ reds})"
definition (in wine_dis) white_grapes :: "sdatum" where
  "white_grapes = Set.filter(λw. sname w = sort2name S_W) (⋃{x. x ∈
whites})"
```

```
definition (in wine_dis)  whiteWine :: "sdatum set ⇒ dataconcept" where
  "whiteWine X = {{dt} | dt. dt ∈ X ∧ τ_w {dt} = wine}"

end
```

B.18.1 Elements of Reasoning: Consistency Checking

```
context wine_dis
begin
```

Attribute consistency: same wine cannot be different colors in the generator set

```
lemma (in wine_dis) reds_whites_int: "reds ∩ whites = {}"
  using D_def wine_generator_def reds_def whites_def WineDDV.wine_ddv.axioms
  using local.univ_def wine_ddv.dt_in_SD wine_ddv.dt_v_in_k by blast

end
```

B.18.2 Elements of Reasoning: Classification and Subsumption

```
context wine_dis
begin
```

Classification: The merlot wine is classifed as a red wine.

```
lemma (in wine_dis) merlot_in_reds: "{merlot, red, dry, full} ∈ reds"
  using D_def wine_generator_def reds_def
  using wine_ddv.dt_in_SD wine_ddv.dt_v_in_k by fastforce
```

Classification: The merlot is classifed as a red grape.

```
lemma (in wine_dis) merlot_red: "merlot ∈ red_grapes"
  using merlot_def red_grapes_def D_def reds_def wine_generator_def
wine_univ.sort_svalue
  using local.univ_def wine_ddv.dt_in_SD wine_ddv.dt_v_in_k by blast
```

Classification: The magliocco is classifed as a red grape.

```
lemma (in wine_dis) magliocco_red: "magliocco ∈ red_grapes"
  using magliocco_def red_grapes_def D_def reds_def wine_generator_def
wine_univ.sort_svalue
  using local.univ_def wine_ddv.dt_in_SD wine_ddv.dt_v_in_k by blast
```

Other classification and subsumption results

```
lemma (in wine_dis) rg_def: "red_grapes = {merlot, magliocco}"
proof
```

```
  show "red_grapes ⊆ {merlot, magliocco}"
    using wine_ddv.a_in_one wine_ddv.dt_not_empty wine_ddv.one_closed
          local.univ_def wine_ddv.dt_in_SD wine_ddv.dt_v_in_k by blast
next
  show "{merlot, magliocco} ⊆ red_grapes"
    using merlot_red magliocco_red by auto
qed

lemma (in wine_dis) wg_def: "white_grapes = {chard}"
  using ddv_base.dt_in_SD wine_ddv.ddv_base_axioms wine_ddv.dt_not_empty
        wine_ddv.one_closed wine_ddv.one_def by blast

lemma (in wine_dis) chard_not_merlot: "chard ≠ merlot"
  using chard_def merlot_def svalue_eq
  using local.univ_def wine_ddv.dt_in_SD wine_ddv.dt_v_in_k by blast

lemma (in wine_dis) chard_not_magliocco: "chard ≠ magliocco"
  using chard_def magliocco_def svalue_eq
  using local.univ_def wine_ddv.dt_in_SD wine_ddv.dt_v_in_k by blast

lemma (in wine_dis) chard_nred: "chard ∉ red_grapes"
  using chard_not_magliocco chard_not_merlot rg_def by auto

lemma (in wine_dis) chard_red: "chard ∈ white_grapes"
  using chard_not_magliocco chard_not_merlot wg_def by auto

lemma (in wine_dis) "red_grapes ∩ white_grapes = {}"
  using rg_def wg_def chard_not_merlot chard_not_magliocco by auto

end
```

B.18.3 Elements of Reasoning: Concept Satisfiability

```
context wine_dis
begin
```

Datascape concept satisfiability: The red wines ***reds*** contains at lest one s-data instance.

```
lemma (in wine_dis) reds_satisfiable: "reds ≠ {}"
  using merlot_in_reds by auto

end
```

B.18.4 Elements of Reasoning: Inference

context *wine_dis*
begin

Inference: Region relates to Producer, Producer relates to Estate, Estate relate to Grape (through (R_r)), (R_r) is transitive, thus Region relates to Grape.

lemma (**in** *wine_dis*) *region_grape_relation:* "(*region, grape*) ∈ R_r"
  using R_r*_def r_trans is_trans_def*
  by (*metis insertCI r_r_into_trancl*)

end

end

Appendix C

DIS Templates for Isabelle/HOL

C.1 BNF Production Rules: Meta

As described in Section 6.2, the templates follow a similar format. Let the theory be named `#TheoryName#` and the template component be `COMP`, where `comp` can take, in turn, any of the following values: `Univ`, `DDV`, `DOnt`, and `DIS`. Note that `comp` denotes the component name, all lower case. The generic BNF production rules for the component follow this format:

⟨*template_comp*⟩ ::= ⟨*preamble_comp*⟩ ⟨*theory_comp*⟩
⟨*preamble_comp*⟩ ::=

```
(* Title:      #TheoryName#COMP.thy
   Author(s): #authorList#
   Date:      #date#
*)
section <<#TheoryName#COMP Theory>>
text <<#TheoryNameDDVDescription#>>
```

⟨*theory_comp*⟩ ::=

```
theory #TheoryName#COMP
```

⟨*theory_imports_comp*⟩

```
begin
```

⟨*theory_content_comp*⟩

```
end
```

⟨*theory_content_comp*⟩ ::= ⟨*declarations_comp*⟩
⟨*instance_comp*⟩
⟨*context_comp*⟩

⟨*context_comp*⟩ ::=

```
context #TheoryName#_comp
```

```
begin
end
```

The production rules for the ⟨*theory_imports_comp*⟩, ⟨*declarations_comp*⟩, and ⟨*instance_comp*⟩ nonterminals are specific to each DIS component. Thus, they are detailed below in their respective sections.

C.2 Universe Template: BNF Production Rules

```
⟨template_univ⟩ ::= ⟨preamble_univ⟩  ⟨theory_univ⟩
⟨preamble_univ⟩ ::=
(* Title:      #TheoryName#Univ.thy
    Author(s): #authorList#
    Date:      #date#
*)
section <<#TheoryName#Univ Theory>>
text <<#TheoryNameUnivDescription#>>
⟨theory_univ⟩       ::=
theory #TheoryName#Univ
⟨theory_imports_univ⟩
begin
   ⟨theory_content_univ⟩
end
⟨theory_imports_univ⟩ ::= imports DDVUniverse
⟨theory_content_univ⟩ ::= ⟨declarations_univ⟩
                          ⟨locale_univ⟩
                          ⟨context_univ⟩
⟨declarations_univ⟩ ::= ⟨sorts_list⟩ ⟨universe_set⟩ ⟨universe_mappings⟩
⟨sorts_list⟩         ::=
!foreach #sort#!
definition #sort# :: sort where
  "#sort# = #sort_type# {!listgen #value#!}
!eforeach!
⟨universe_set⟩       ::=
definition 𝒰 :: "sort set" where
  "𝒰 = {!listgen #sort#!}
⟨universe_mappings⟩ ::=
definition #thname#_sort2name :: "sort ⇒ string" ("s2n _") where
 "(s2n s) = !foreach #sort#!
```

```
              (if s=#sort# then ''#sort#'' else
                !if 'position eq last'! '''' !eif!
             !eforeach!
             !foreach sort#! ) !eforeach!"
definition #thname#_name2sort :: "string ⇒ sort" ("n2s _") where
 "(n2s n) = !foreach #sort#!
              (if n=''#sort#'' then #sort# else
                !if 'position eq last'! {} !eif!
             !eforeach!
             !foreach sort#! ) !eforeach!"
⟨locale_univ⟩   ::=
locale #TheoryName#_univ =
 #TheoryName#_univ: universe where 𝒰 = 𝒰 and
           sort2name = sort2name and name2sort = name2sort
  for 𝒰 :: "sort set"
   and sort2name :: "sort ⇒ string"
   and name2sort :: "string ⇒ sort" +
  assumes univ_def: "𝒰 = #TheoryName#Univ.𝒰"
   and s2n_def: "sort2name s = (s2n s)"
   and n2s_def: "name2sort n = (n2s n)"
begin
end
⟨context_univ⟩    ::=
context #TheoryName#_univ
begin
end
```

C.3 Domain Data View Template: BNF Production Rules

```
⟨template_ddv⟩    ::= ⟨preamble_ddv⟩ ⟨theory_ddv⟩
⟨preamble_ddv⟩   ::=
(* Title:      #TheoryName#DDV.thy
  Author(s): #authorList#
  Date:      #date#
*)
section <<#TheoryName#DDV Theory>>
```

```
text <<#TheoryNameDDVDescription#>>
⟨theory_ddv⟩       ::=
theory #TheoryName#DDV
⟨theory_imports_ddv⟩
begin
  ⟨theory_content_ddv⟩
end
⟨theory_imports_ddv⟩ ::= imports
                           #TheoryName#Univ
                           DomainDtView
⟨theory_content_ddv⟩ ::= ⟨declarations_ddv⟩ ⟨locale_ddv⟩
⟨declarations_ddv⟩ :: =
context #TheoryName#_univ
begin
definition #TheoryName#_generator :: "sdata set"
   where "#TheoryName#_generator = {!listgen #sdata#!}"
end
⟨locale_ddv⟩    ::=
locale #TheoryName#_ddv = #TheoryName#_univ +
   #TheoryName#ddv: ddv where D = D
   for D :: "sdata set" +
assumes "D = #TheoryName#_generator"
begin
end
⟨context_ddv⟩     ::=
context #TheoryName#_ddv
begin
end
```

C.4 Domain Ontology Template: BNF Production Rules

```
⟨template_dont⟩ ::= ⟨preamble_dont⟩ ⟨theory_dont⟩
⟨preamble_dont⟩ ::= (* Title:    #TheoryName#DOnt.thy
                       Author(s): #authorList#
                       Date:      #date#
                    *)
```

```
                    section <<#TheoryName#DOnt Theory>>
                    text <<#TheoryNameDOntDescription#>>
⟨theory_dont⟩     ::= theory #TheoryName#DOnt
                      ⟨theory_imports_dont⟩
                      begin
                        ⟨theory_content_dont⟩
                      end
⟨theory_imports_dont⟩ ::= imports DomainOnt
⟨theory_content_dont⟩ ::= ⟨declarations_dont⟩ ⟨locale_dont⟩
⟨declarations_dont⟩ ::= ⟨atomic_concepts⟩
                         [⟨named_concepts⟩]
                         ⟨rooted_graphs⟩
⟨atomic_concepts⟩ ::=
datatype #thname# = !listgen #atom# sep=‖ lstitem=#atom#_at!
!foreach #atom#!
definition #atom# :: "#thname# concept"
   where "#atom# = {#atom#_at}"
!eforeach!
definition A_L :: "#thname# concept set"
   where "A_L = {!listgen #lattice_atom#!}"
⟨named_concepts⟩ ::=
!foreach #named_concept#!
definition #named_concept# :: "#thname# concept"
   where #named_concept# = !listgen #concept# sep=⊕!
!eforeach!
⟨rooted_graphs⟩ ::=
!foreach #rooted_graph#!
definition #V# :: "#thname# concept set"
   where "#V# = {!listgen (#vertice#)!}"
definition #R# :: "#thname# edge set"
   where "#R# = {!listgen #edge# lstitem=(#tail#, #head#)!}"
definition #G# :: "#thname# rgraph"
   where "#G# = (|vertices = V, edges = R, root = #root#|)"
!eforeach!
⟨locale_dont⟩ ::=
locale #TheoryName#_dont =
    dont: domain_ontology where A_C = A_C and A_L = A_L and G = G
for    A_C :: "#thname# concept set"
    and A_L :: "#thname# concept set"
```

```
    and 𝒢 :: "#thname# rgraph set" +
assumes atom_l_def: "A_L = #TheoryName#DOnt.A_L"
    and atom_c_def: "A_C = A_L ∪ {!listgen #other_concept#!}"
    and graphs_def: "𝒢 = {!listgen #rg#}"
begin
end
⟨context_dont⟩    ::=
context #TheoryName#_dont
begin
end
```

C.5 Domain Information System Template: BNF Production Rules

```
⟨template_dis⟩ ::= ⟨preamble_dis⟩ ⟨theory_dis⟩
⟨preamble_dis⟩ ::= (* Title:   #TheoryName#DIS.thy
                      Author(s): #authorList#
                      Date:      #date#
                   *)
                   section <<#TheoryName#DIS Theory>>
                   text <<#TheoryDISDescription#>>
⟨theory_dis⟩ ::=
theory #TheoryName#DIS
⟨theory_imports_dis⟩
begin
    ⟨theory_core_dis⟩
end
⟨theory_imports_dis⟩ ::= imports
                         DomainInfSys
                         #TheoryName#DDV
                         #TheoryName#DOnt
⟨theory_content_dis⟩  ::= ⟨declarations_dis⟩ ⟨locale_dis⟩ ⟨context_dis⟩
⟨declarations_dis⟩ ::=
⟨locale_dis⟩ ::= ⟨locale_core_dis⟩
                 [⟨datascape_concept_definition⟩]+
⟨locale_core_dis⟩ ::=
locale #TheoryName#_dis = #TheoryName#_ddv + #TheoryName#_dont +
```

```
fixes sort2atom :: "sort ⇒ #thname# concept"
assumes s2a_def: "sort2atom s = !foreach #sort#!
              (if s=#sort# then ''#atom#'' else
                !if 'position eq last'! '''' !eif!
             !eforeach!
             !foreach sort#! ) !eforeach!"
begin
interpretation dis A_C A_L G U sort2name name2sort D sort2atom
    apply unfold_locales
    done
definition #thname#_type :: "sdata ⇒ #thname# set"
       ("τ_#thnamei# _" [99] 100) where
    "τ_#thnamei# a = τ a"
end
⟨datascape_concept_definition⟩ ::=
definition #name# :: "sdata set ⇒ dataconcept" where
    "#name# X = {a | a. a ∈ X ∧ ⟨condition⟩
                                ∧ τ_#thnamei# a = ⟨concept⟩}"
⟨concept⟩ ::= #atom# | #named_concept# | ⟨concept⟩⊕⟨concept⟩
⟨context_dis⟩    ::=
context #TheoryName#_dis
begin
end
```

Appendix D

Additional Material on Mathematical Background

D.1 Domain Information System (DIS)

The following is a list of all axioms that hold in a DIS structure $\mathcal{I} = (\mathcal{O}, \mathcal{A}, \tau)$. Let $a, b \in A, k, k_1, k_2 \in C, \kappa, \lambda \in \mathcal{U}, i \in \mathbb{N}, G_{t_i} \in \mathcal{G}$, with $G_{t_i} = \big(C_i, R_i, t_i\big)$

(A1) $\big(C, \oplus, \mathbf{e}_C\big)$ is a commutative idempotent monoid

(A2) $\big(L, \oplus, \otimes, \ominus, \mathbf{e}_C, \top_{\mathcal{L}}\big)$ is a Boolean algebra

(A3) $L \subseteq C$

(A4) $C_i \subseteq C$

(A5) $R_i \subseteq C_i \times C_i$

(A6) $(k_1, k_2) \in R_i \implies k_1 \neq k_2$

(A7) $t_i \in L \wedge \forall k \in C_i.\ k = t_i \vee (k, t_i) \in {R_i}^+$

(A8) $(k_1, k_2) \in R_i \implies k_2 = t \vee (k_2, t_i) \in R_i^+$

(A9) $k \in C \implies k \in L \vee \exists G_{t_i} = \big(C_i, R_i, t_i\big) \in \mathcal{G}.\ k \in C_i$

(A10) $\big(A, +, \star, -, 0_A, 1_A\big)$ is a Boolean algebra

(A11) $\mathbf{c}_\kappa(0) = 0$

(A12) $a \leqslant \mathbf{c}_\kappa(a)$

(A13) $\mathbf{c}_\kappa(a \star \mathbf{c}_\kappa(b)) = \mathbf{c}_\kappa(a) \star \mathbf{c}_\kappa(b)$

(A14) $\mathbf{c}_\kappa(\mathbf{c}_\lambda(a)) = \mathbf{c}_\lambda(\mathbf{c}_\kappa(a))$

(A15) $\tau(0_A) = \mathbf{e}_C$

(A16) $\tau(1_A) = \top_{\mathcal{L}}$

(A17) $\tau(a+b) = \tau(a) \oplus \tau(b)$

Axiom (A1) describes the monoid of concepts $\mathcal{C}$. Axioms (A2)−(A3) describe the Boolean lattice $\mathcal{L}$. Axioms (A4)−(A9) describe the family of rooted graphs $\mathcal{G}$, as well as the boundary set on C. Axioms (A10)−(A14) describe the cylindric algebra $\mathcal{A}$. Axioms (A15)−(A17) describe the operator τ. With every monoid there is a natural order. In the monoid $\mathcal{C}$ (of the DOnt component), this is expressed as the partial order $\sqsubseteq_C$, defined as $k_1 \sqsubseteq_C k_2 \overset{\text{def}}{\iff} k_1 \oplus k_2 = k_2$. In the cylindric algebra $\mathcal{A}$ (of the DDV component), the natural order denoted by $\leqslant$ is defined as $a \leqslant b \overset{\text{def}}{\iff} a+b = b$.

Note that as discussed in Chapter 3, Section 3.2.4, the Boolean lattice $\mathcal{L}$ of the DIS $\mathcal{I}$ is isomorphic to a Boolean algebra, and we give the axioms in algebraic terms.

D.2 Cylindric Algebra

To better understand cylindric algebras, we give its geometrical interpretation as a k-dimensional cylindric algebra that represents the universe of all k-dimensional objects (a set of k-dimensional points) on which the Boolean operators (intersection, union, negation) create new solids (Tarski *et al.*, 1971). The operation of cylindrification on the i-th dimension can be understood as a projection of the solid on the k-1 space that is being extended to the whole "cylinder" along the removed dimension. Figure D.1 shows the representation of a two-dimensional cylindric algebra. In Figure D.2 we give the geometrical interpretation of axiom $(C4)$ in a two-dimensional cylindric algebra.

In (Imielinski and Lipski, 1984), the authors show there exists a natural embedding of the RA described below and the diagonal-free cylindric set algebra. Since its introduction in (Codd, 1970), Codd's relational model of data has been accepted as a clear and succinct model for relational databases.

Let $\mathcal{U}$ be a fixed set of attributes, $A_1, A_2, \cdots, A_n$. We call a set of attributes $J \subseteq \mathcal{U}$ a *type*. With each attribute $A_i \in \mathcal{U}$ there is associated a non-empty *attribute domain*, $D(A_i)$. A *relation of type* J is a set of tuples $R \subseteq \Pi_{A_i \in J} D(A_i)$; an element $t \in R$ is called a tuple (of type J). For a tuple t in R, we write $\tau(R) = \tau(t) = J$, and we call

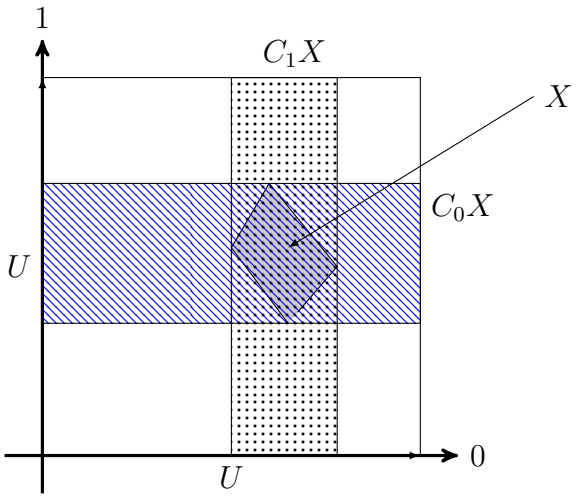

Figure D.1: Cylindric algebra: geometrical representation Tarski *et al.* (1971)

the τ operator the type of R (or t, respectively). We say that a tuple t of type J is a mapping, associating a value $t(A_i) \in D(A_i)$ with each attribute $A_i \in J$. A restriction of the mapping to $K \subseteq J$ is written as $t[K]$.

In database theory, a relation of type J is generally assumed to be finite and it is represented as a table with columns representing each attribute in J, and rows corresponding to tuples. The following basic relational operators are defined:

- *Projection* ("vertical" decomposition): $\pi_K(R) = \{t[K] \mid t \in R\}$, where $K \subseteq \tau(R)$
- *Selection* ("horizontal" decomposition): $\sigma_E(R) = \{t \in R \mid E(t)\}$, where E is the selection condition, usually defined as a logical formula where the atomic conditions are of the form $(A_i = a), a \in D(A_i)$ or $(A_i = A_j), A_i, A_j \in \mathcal{U}$.
- *Union* (the usual set theory union): $R \cup Q$
- *Join* (natural join): $R \bowtie Q = \{t \mid \tau(t) = J \cup K \wedge t[J] \in R \wedge t[K] \in Q\}$

The following results are borrowed from (Imielinski and Lipski, 1984).
Let R be any relation, let $J = \tau(R)$, and let us define $\mathbb{1} = \Pi_{A_i \in J} D(A_i)$ (the universe of tuples). The mapping

$$h(R) = \{t \in \mathbb{1} \mid t[J] \in R\}$$

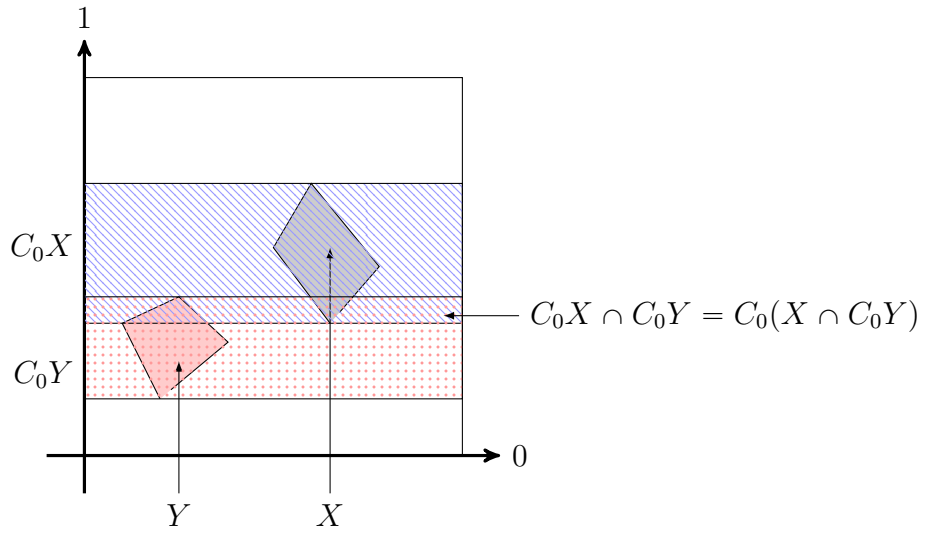

Figure D.2: Cylindric algebra: geometrical representation of axiom (C4). Tarski *et al.* (1971)

is obtained by extending every tuple in R to the entire set of attributes in $\mathcal{U}$ (in all possible ways). It is easy to observe that $h(R)$ can be considered an element of a diagonal-free cylindric set algebra of subsets of $\mathbb{1}$, with the cylindrification operator corresponding to attributes in $\mathcal{U}$.

On a cylindric algebra, we consider generalised cylindrification on a subset of multiple dimensions. Let $J = \{k_1, k_2, \cdots k_n\}$, such that $\forall i \in \mathbb{N}, k_i \in J.\ i \leqslant n \wedge k_i < \alpha$.

$$\mathbf{c}_{(J)}x \stackrel{\text{def}}{=} c_{k_1}c_{k_2}\cdots c_{k_n}x$$

The mapping h defines a natural embedding of the RA into the diagonal-free cylindric set algebra of subsets of $\mathbb{1}$.

Theorem D.2.1 (Imielinski and Lipski (1984))**.**

(i) $h(\pi_K(R)) = \mathbf{c}_{(\mathcal{U}-K)}h(R)$

(ii) $h(\sigma_E(R)) = \sigma_E(h(R)) = h(R) \cap \sigma_E(\mathbb{1})$

(iii) $h(R \cup Q) = h(R) \cup h(Q)(\tau(R) = \tau(Q))$

(iv) $h(R \bowtie Q) = h(R) \cap h(Q)$ □

The proof of the above results can be found in (Imielinski and Lipski, 1984).
Part (a) of the theorem shows that we perform a projection not by shrinking the relation (through removal of one or more attributes), but by expanding it with every possible value of the removed attributes. In (Goczyła *et al.*, 2009), the authors observe this method of projection bridges the gap between relational database and ontologies. In traditional relational models, the absence of data is generally treated as negative information, while in knowledge systems it is treated as absence of knowledge. For example, in a database, two address rows (tuples) related to a person, Jane, can be interpreted as 'Jane has exactly 2 homes'. In an ontology, the same information is be interpreted as 'Jane has at least 2 homes'. Thus, ontologies make no assumption about facts that have not been explicitly described; this open-world behaviour is easily captured by the cylindrification operator, while the projection would have inaccurately restrict the information. The removed attributes describe the part of knowledge we do not explicitly possess, we may assume any value for those attributes.

Appendix E

Isabelle Overview

In this appendix, we provide more details on the ITP Isabelle. The appendix is structured as follows: in Section E.1, we describe the basic elements of Isabelle's syntax, and in Section E.2 we present two core components of Isabelle, its theories and locales. In Section E.3 we detail the concrete syntax of Isabelle and describe elements the user should be aware of. Finally, in Section E.4, we describe the Isabelle proof engine, with its main elements, and the recommended usage. Throughout this appendix, the generic Isabelle commands are described in standard BNF, briefly described in Section 6.1. We remind the reader that throughout the text, we use `black truetype` font for Isabelle keywords. Within the listings, we use colour to highlight the keywords.

E.1 Types, Terms, Formulae, and Variables

In our work, we focus on Isabelle/HOL, a widely used instantiation of Isabelle, tailored for reasoning in HOL. HOL is a typed logic, based on typed λ-calculus, and similar to functional programming languages like ML or Haskell (Gerwin Klein, 2024; Nipkow, 2025). Its **types** are:

- *base types* such as `bool`, `nat` ($\mathbb{N}$), `int` ($\mathbb{Z}$)
- *type constructors*, written postfix, such as list or set. E.g., `int set` describes the type of sets whose elements are integers
- *function types*, denoted by $\Rightarrow$. In Isabelle/HOL, functions are total. The notation $[\tau_1, \ldots, \tau_n] \Rightarrow \tau$ is understood as $\tau_1 \Rightarrow \ldots \Rightarrow \tau_n \Rightarrow \tau$
- *type variables*, denoted by ‘a, ‘b etc.

In Isabelle, terms and formulae must be well-typed. However, when possible Isabelle uses type inference and automatically computes the type of each variable in a term, giving an error when the type cannot be automatically inferred.

Terms are formed by applying functions to arguments. If `f` is a function of type $\tau_1 \Rightarrow \tau_2$ and `t` is a term of type τ_1 then `f t` is a term of type τ_2. We write `t` :: τ and mean that the term `t` is of type τ. In addition to the standard postfix, through syntax annotations, Isabelle allows the declaration of infix and midfix notations, discussed in Section E.3. Terms may also contain λ-abstractions, such as λ`x. x + 1`. Nested λ-abstractions can be abbreviated, e.g., the term λ`x y. t` can be used instead of λ`x`. λ`y. t`.

In Isabelle/HOL, **formulae** are treated as propositions, i.e., formulae are terms of type `bool`. The type `bool` contains the constants `True` and `False`, and it uses the logical connectives $\neg$, $\wedge$, $\vee$, $\forall, \exists$, and $\longrightarrow$. The binary connectives associate to the right, i.e., `A` $\longrightarrow$ `B` $\longrightarrow$ `C` is equivalent to `A` $\longrightarrow$ (`B` $\longrightarrow$ `C`). Equality is available in the form of the infix function = of type 'a $\Rightarrow$ 'a $\Rightarrow$ `bool`. The equality connective works for formulae as well, where it is interpreted as *iff*. Quantifier fomulae are written $\forall$`x. P` and $\exists$`x. P`. Similar to λ-abstractions, quantifiers may be nested, i.e., $\forall$`x y z. P` may be written instead of $\forall$`x`. $\forall$`y`. $\forall$`z. P`. For considerations of consistency, clarity, and parsing efficiency, the HOL connectives may appear after the meta-logic connectives, and not the other way around. Therefore, HOL implication $\longrightarrow$ binds more tightly than the meta-logic implication $\Longrightarrow$. E.g., the term $\forall$`x. P` $\longrightarrow$ `Q` is equivalent to $\forall$`x`. (`P` $\longrightarrow$ `Q`), as the quantifier $\forall$`x` binds to `P` $\longrightarrow$ `Q`. In contrast, the term $\forall$`x. P` $\Longrightarrow$ `Q` is equivalent to ($\forall$`x. P`) $\Longrightarrow$ `Q`, as the quantifier binds to `P`.

Sometimes type inference fails and it is necessary to attach an explicit type annotation to a variable or term. The explicit type annotation is weakly binding. E.g., the term `x + y` :: `nat` is interpreted as (`x + y`) :: `nat`. To avoid that, the type annotation must be enclosed in parentheses. E.g., the term `x` + (`y` :: `nat`) is well formed and expresses the intention to annotate the type of `y`.

In Isabelle, there are three types of **variables**: free, bound, and *schematic*. The free and bound variables have the same meaning as in FOL. To avoid name-clashes with free variables, bound variables are automatically renamed by the system. The *schematic* variables have a ? as their first character. They are logically free variables and, during the proof process, they may be arbitrarily intantiated by other terms. In contrast, free and bound variables remain fixed. In Prolog, the *schematic* variables are called *logical* variables.

In Table 5.1 we presented general syntactic rules for Isabelle. A good usage rule is

to use parantheses and proper identifiers to ensure that what is written is what was intended. Note that space symbols matter in the Isabelle editor. E.g., the term λx.P is interpreted as λ(x.P). This is most likely not at all what the user intended, which is to express λx. P (note the space after the .).

E.2 Theories and Locales

In Isabelle, *theories* and *locales* serve as fundamental constructs in the structuring and organisation of formal developments. Theories manage global definitions and proofs, while locales offer adaptable, reusable contexts for assumptions and parameters specific to proofs. A *theory* represents a collection of formal definitions, types, functions, and proofs (i.e., `lemmas`). Theories are defined using the `theory` keyword and serve as modules that can be built on or imported by other theories. Thus, a theory is similar to a specification in a specification language. The general structure of a theory `T` is provided in Listing E.1. The `T1 ... Tn` are parent theories for `T`. All definitions and proofs of the parent theories are inherited in theory `T`. Names can be qualified to avoid clashes, e.g., `T1.f` or `T.f`. The theory `T` must be saved into a file theory named exactly "T.thy".

```
theory T
imports T1 ... Tn
begin
  definitions, theorems and proofs
end
```

Listing E.1: Theory: General Structure

Locales are designed as a system of modules that allow the user to represent the complex dependencies between mathematical structures in abstract algebra. Locales correspond to parametrised theories (Definition 3.4.4) (Ballarin, 2010). They offer a method to further enhance the structure of a theory, by defining a context in which assumptions and parameters are locally valid. The *context* includes assumptions, constants, and parameters, and represents a reusable module that can be instantiated at a later time, as needed. The context is a formula schema, as described in Listing E.2, Eq.(1). The variables $\mathtt{x}_1, \ldots, \mathtt{x}_n$ are called *parameters*, and the premises $\mathtt{A}_1, \ldots, \mathtt{A}_m$ are called *assumptions*. Within a given context, a formula `C` that is a conclusion (i.e., a specific assertion or statement derived from the context) is called a *theorem*, as shown in Listing E.2, Eq.(2).

(1) $\bigwedge \mathtt{x}_1 \ldots \mathtt{x}_n. [\![\mathtt{A}_1; \mathtt{A}_2; \ldots; \mathtt{A}_m]\!] \Longrightarrow \ldots$
(2) $\bigwedge \mathtt{x}_1 \ldots \mathtt{x}_n. [\![\mathtt{A}_1; \mathtt{A}_2; \ldots; \mathtt{A}_m]\!] \Longrightarrow \mathtt{C}$

Listing E.2: Locales: Context

Note that the $\bigwedge x_1 \ldots x_n.\ [\![A_1; A_2; \ldots; A_m]\!] \Longrightarrow C$ formula is interpreted in Isabelle as $\bigwedge x_1 \ldots x_n.\ A_1 \Longrightarrow (A_2 \Longrightarrow \ldots (A_m \Longrightarrow C) \ldots)$, and it reads as "For all $x_1 \ldots x_n$, from A_1 and A_2 and ... and A_m we have C".

A locale is defined using the keyword `locale`, and includes a sequence of parameters (denoted by the keyword `fixes`) and assumptions (denoted by the keyword `assumes`). Parameters are declarations of operators (zero-nary or constants, unary, binary etc.) or relations. Assumptions are axioms that describe specific properties of the given locale parameters. In Listing E.3, we give the specification for a semigroup. The parameter of locale `semigroup` is `mult`, a binary operator with infix syntax $*$. The parameter syntax is available in the subsequent assumption, which defines the operator as associative. The unbound names `a, b, c` are recognised as free variables, and they are implicitly universally qualified (i.e., their type is any type `'a`) in the locale assumptions. Thus the formula of the `assoc` assumption is understood as "$\bigwedge a\ b\ c.\ a * b * c = a * (b * c)$".

```
locale semigroup =
  fixes mult :: "'a ⇒ 'a ⇒ 'a" (infixl "*" 70)
  assumes assoc: "a * b * c = a * (b * c)"
```

Listing E.3: Locales: Semigroup Specification

The locale contexts are recorded into Isabelle's kernel. In addition, the locale framework allows the user to declare and combine contexts and reuse the theorems proved in these contexts. Isabelle provides several commands to inspect locales. The command `print_locales` prints the names of all locales in the current theory. The command `print_locale loc` lists the parameters and assumptions of locale `loc`. The command `print_locale! loc` lists the theorems that have been stored in the locale `loc`.

E.g., when the user inspects the locale `semigroup` with the command `print_locale!semigroup`, Isabelle provides the output in Listing E.4. The output is slightly different from the locale declaration, and it is what Isabelle has recorded in its kernel for the `semigroup` locale.

```
locale semigroup
  fixes mult :: "'a ⇒ 'a ⇒ 'a" (infixl ⟨*⟩ 70)
  assumes "semigroup (*)"
  notes "assoc" = (⟨?a * ?b * ?c = ?a * (?b * ?c)⟩)
```

Listing E.4: Semigroup Locale: Kernel Record

The parameter declaration remains the same. Isabelle introduces a new assumption into the theory, the *locale predicate* "`semigroup (*)`". Then, Isabelle transforms

each assumption of the locale specification into a new conclusion, denoted by the use of keyword `notes`. For each such conclusion, Isabelle introduces a corresponding *foundational theorem* in the theory. A foundational theorem is composed of a context (i.e., the list of parameters and assumptions) and a conclusion. A foundational theorem can be invoked by its fully qualified name ⟨`locale`⟩.⟨`assumption`⟩. E.g., in the `semigroup` locale, the original assumption (`assoc`) has been turned into a conclusion and transformed into a foundational theorem. This theorem can be accessed using its fully qualified name `semigroup.assoc`.

The specification of a locale (i.e., its parameters and assumptions) is fixed. The list of the locale conclusions can be extended by using Isabelle commands that take a *target* argument, such as the `definition` and `lemma` commands. When using these commands, the target argument is indicated with the keyword `in`, as shown in Listing E.5.

```
definition (in semigroup)
  le :: "'a ⇒ 'a ⇒ bool" (infixl "⊑" 50)
  where "(x ⊑ y) = (x * y = y)"
```

Listing E.5: Extending the Semigroup Locale

This declarations extends the `semigroup` locale, by adding a new *foundational constant* named `semigroup.le`. Isabelle next adds the conclusion `semigroup.le_def` to the locale. This new foundational constant can be accessed using its abbreviated name `le`, which is printed and parsed as ⊑.

Within locales, theorems are proved in the context of the given set of assumptions. The theorems can then be used in other contexts where the assumptions themselves are theorems. This form of theorem reuse is called *interpretation* and it is made available through the command `interpretation`, as described in Listing E.6. The parameters "`param`$_1$" ... "`param`$_n$" must be given in the order of declaration, i.e., the order the `print_locale` command outputs them. Note that Isabelle abuses the term `interpretation`, which is meant as *model*. Thus, once an `interpretation` command is issued, Isabelle requires proof (i.e., ⟨`proof`⟩) that the set of parameters form a model for the interpreted locale.

```
interpretation ⟨label⟩ : ⟨loc⟩ "param1" ... "paramn" ⟨proof⟩
```

Listing E.6: Locale Intepretation

In Listing E.7, we provide an example of an interpretation for the `semigroup` locale. The parameter of the `semigroup` locale has been replaced by the binary operator (+) that operates on naturals.

```
interpretation nats: semigroup "(+) :: nat ⇒ nat ⇒ nat"
    by unfold_locales auto
```

Listing E.7: Locale Intepretation: Naturals Semigroup

After the `interpretation` command is issued, instances of all conclusions of the locale are available in the theory and are accessible by their name, prefixed by the `interpretation` label. E.g., within the `nats` interpretation of the `semigroup` locale, the theorem of associativity for naturals is named `nats.assoc` and is described by the theorem in Listing E.8. The schematic variables ?`x`, ?`y`, ?`z` are of type `nat` (i.e., they are all naturals).

```
?x * (?y * ?z) = ?x * ?y * ?z
```

Listing E.8: Naturals Semigroup Intepretation: Associativity

Note that in Isabelle, another method to modularise theories is the use of *classes*. Classes correspond to interfaces in object-oriented languages and can be used to define abstract structures, similar to the use of locales. Similarly to locales, classes can be instantiated to concrete structures, s.t. the theorems of the abstract class are inherited by the concrete structure. The main limitation is that in classes, type inference is enforced automatically, meaning the type system ensures that all operations within a class apply to a single type. This can lead to restrictions where only one type variable is allowed per instance to maintain consistent typing. On the other hand, in locales, there is no automatic type inference. As a result, variables in locales can involve multiple types without the restrictions imposed by type classes, allowing greater flexibility in defining assumptions and structures.

E.3 Concrete Syntax

The concrete syntax of the Isabelle framework uses *mixfix annotations*, in which operators can have multiple notations, and can be either infix or postfix. The mixfix operators use precedence and associativity to parse a term. E.g., consider the declaration of a binary operator in Listing E.9 Eq.(1). The term `plus a b` is well formed, however the term `plus a plus b c` throws a parser error. To parse it, the term must be written as `plus a (plus b c)`. The postfix notation is not natural, and Isabelle allows the user to annotate the declaration, as shown in Listing E.9 Eq.(2).

```
(1) plus :: "'a ⇒ 'a ⇒ 'a"
(2) plus :: "'a ⇒ 'a ⇒ 'a" (infixl "+" 50)
```

Listing E.9: Elements of Isabelle Syntax: Operator Annotation

Precedence	Associativity	Result
$+ < *$		$a + (b * c)$
$+ > *$		$(a + b) * c$
$+ = *$	Both Right	$a + (b * c)$
$+ = *$	Both Left	$(a + b) * c$
`Any`	Other Combination	Parse Error!

Table E.1: Parsing the term $a + b * c$ in Isabelle

The infix can associate to the right (keyword `infixr`), to the left (keyword `infixl`), or it may have no orientation (keyword `infix`). In Listing E.9 Eq.(2), the symbol + represents the literal token of the operator plus, and the number (50) determines the precedence of the operator. An operator with a higher precedence binds tighter than an operator with a lower precedence. In case of equal precedence, the parser uses associativity of the operator to parse a term and decide if it is well-formed or not. In Table E.1, we show the result of the parsing of the term $a + b * c$, according to the precedence and associativity of both operators.

Mixfix annotations are syntactic decorations, i.e., they replace a postfix application of a certain function with its concrete syntax notation. E.g., the term `plus a b` is replaced with the more readable term $a + b$. In contrast, an *abbreviation* introduces a constant that stands in for a more complex term, effectively functioning as a syntactic macro to simplify expressions. An abbreviation is used as a rewrite rule after parsing and before printing. E.g., assume we want to represent the relation of similarity. We introduce a relation `Sim`, defined as a set of pairs. We can then introduce an abbreviation for the relational notation for the membership in `Sim`, replacing the notation $(a, b) \in Sim$ with a more natural $a \sim b$, as shown in Listing E.10. After parsing, Isabelle replaces all occurrences of $a \sim b$ by $(a, b) \in Sim$. Before printing, it does the opposite, and replaces all occurrences of $(a, b) \in Sim$ by $a \sim b$. The name of the abbreviation (in our case `sim`) is not critical; it simply needs to be a unique identifier.

```
const Sim :: "('a, 'a) set"
abbreviation sim :: "'a ⇒ 'a ⇒ bool" (infix "∼" 50)
```

```
  where "a ∼ b ≡ (a, b) ∈ Sim"
```

Listing E.10: Elements of Isabelle Syntax: Abbreviations

E.4 Proofs in Isabelle

Proof commands perform transitions of the prover (i.e., the Isar machine configurations) and only certain proof commands are allowed in each proof mode. There are three such modes: **prove**, **state**, and **chain**. In the **prove** mode (annotated in blue in Listing E.11), a new goal is now stated and its proof must follow. In the **state** mode (annotated in red in Listing E.11), a proof block is now open and the context may be extended with intermediate results, additional assumptions etc. In this mode, a **from** statement, a goal statement, or a list of assumptions may follow. In the **chain** mode (annotated in green in Listing E.11), a **from** statement just occurred, and a goal statement must follow. The proof mode can be interactively checked during proof writing, and it is a guide for the proof writer to understand what kind of commands may be issued next. An example of each mode is illustrated in Listing E.11.

```
lemma "⟦A; B⟧ ⟹ A ∧ B" [prove]
proof (rule conjI) [state]
  assume a: "A" [state]
  from a [chain] show "A" [prove] by assumption [state]
next [state]
  assume b: "B" [state]
  from b [chain] show "B" [prove] by assumption [state]
qed [state]
```

Listing E.11: Isabelle Proof Modes

There are two common kinds of proofs In Isabelle, *procedural* and *structured*. The *procedural* proofs, also called apply-style proofs, are formed by sequential application of proof methods until all subgoals are solved. Subgoal management is mainly handled implicitly, and the tactics are applied to the entire proof state at once. In contrast, the *structured* proofs, also called Isar proofs, are organised into modular, explicit blocks, using the Isar proof language. Isar proofs explicitly manage subgoals, allowing for fine-grained control over the proof state. Due to their readability and modularity, Isar proofs are easier to understand and maintain. In this section, we detail both kinds.

The general schema of a procedural proof is

```
lemma [name:] "⟨goal⟩"
```

```
apply ⟨method⟩
...
apply ⟨method⟩
done
```

The `apply` methods solve subgoals in the order provided by the proof state i.e., each method must solve the current subgoal. Isabelle uses both backward and forward reasoning. A backward proof decomposes the conclusion of the goal (i.e., the formula to the right of the $\Longrightarrow$), and a forward proof decomposes the assumptions of the goal (i.e., the formula to the left of the $\Longrightarrow$). E.g., given the Isabelle goal $[\![\mathtt{P};\ \mathtt{Q}]\!] \Longrightarrow \mathtt{P} \wedge \mathtt{Q}$, a forward proof states that "If `P` holds and `Q` holds, then $\mathtt{P} \wedge \mathtt{Q}$ holds", while a backward proof states that "To show $\mathtt{P} \wedge \mathtt{Q}$, show that `P` holds and `Q` holds". A list of commonly used Isabelle method proofs is described in Table 5.2. For more details, we refer the reader to the Isabelle manuals and tutorials (Ballarin, 2008; Nipkow *et al.*, 2021; Wenzel *et al.*, 2021).

Isabelle uses *natural deduction* rules, which captures human reasoning patterns. Each logical connective has two kind of rules, *introduction* rules that infer the connective and *elimination* rules that allow the consequences of the connective to be deduced. A list of some common natural deduction rules for both the introduction and elimination is described in Table 5.3. Note that in Isabelle rules can be *safe* or *unsafe*. A *safe* rule can be applied using backward reasoning with no loss of information, preserving the goal provability. Upon the application of an *unsafe* rule, there is loss of information, which may transform the goal into an unprovable goal. E.g., the **disjI** rule is unsafe, because it reduces $\mathtt{P} \vee \mathtt{Q}$ to `P`, which might turn to be an unprovable goal. This distinction between safe and unsafe rules affects the proof search: if a proof attempt fails, the prover will backtrack to the nearest application of an *unsafe* rule, and make a different choice. A good practice is to always apply safe rules before unsafe ones.

In Isabelle, some basic rules methods are *rule* (introduction), *erule* (elimination), *drule* (destruction or direct), *frule* (forward). Assume we have the following rule, named `R`:

$$[\![\mathtt{P}_1; \ldots; \mathtt{P}_n]\!] \Longrightarrow \mathtt{Q}$$

and a current subgoal, named `G`:

$$[\![\mathtt{A}_1; \ldots; \mathtt{A}_m]\!] \Longrightarrow \mathtt{C}$$

When method (`rule R`) is applied to current subgoal `G`, it unifies `Q` with `C`, and it replaces `G` with `n` new subgoals $\mathtt{P}_1 \ldots \mathtt{P}_n$. This method is appropriate for introduction

rules, and it is backward reasoning.

The application of method (`erule R`) to the current subgoal `G` unifies `Q` with `C` and the first assumption P_1 with some subgoal assumption A_i. The current subgoal is then replaced by $n-1$ new subgoals $P_2 \ldots P_n$, and the matching assumption is removed. This method is appropriate for elimination rules. A similar result is obtained by applying (`rule R`, `assumption`), with the difference that the assumption is not deleted.

By applying method (`drule R`) to the current subgoal, it unifies the first assumption P_1 with some assumption A_i, and deletes that assumption. The current subgoal is replaced by $n-1$ new subgoals $P_2 \ldots P_n$, and an nth subgoal is added, similar with the original subgoal, but with an additional assumption, an instance of `Q`. This method is forward reasoning, and it is appropriate for destruction rules. Method (`frule R`) is similar, but the matching assumption is not deleted.

For an example of backward proof and how Isabelle methods work, let us consider the lemma in Listing E.12.

```
lemma some_thm: "⟦A; B⟧ ⟹ A ∧ (B ∧ A)"    (Step 0)
  apply (rule conjI)                      (Step 1)
  apply assumption                        (Step 2)
  apply (rule conjI)                      (Step 3)
  apply assumption                        (Step 4)
  apply assumption                        (Step 5)
  done                                    (Step 6)
```

Listing E.12: Backward Proof: Example

Isabelle translates the goal of the theorem into the following output:

```
goal (1 subgoal):
1. A ⟹ B ⟹ A ∧ B ∧ A
```

Listing E.13: Backward Proof: Output (Step 0)

In Listing E.12 (Step 1), the prover applies the conjunction introduction rule `conjI`. This unifies the conclusion of the first (and only) goal in Listing E.13 to the conclusion of the rule, and replaces the conclusion by the subgoals of the rule. Thus, the conclusion `P ∧ Q` is replaced by its subgoals, `P` and `Q`, where `P` stands for `A`, and `Q` for `B ∧ A`. Thus, at (Step 1), the output shows two subgoals, as follows:

```
goal (2 subgoals):
1. A ⟹ B ⟹ A
```

```
2. A ⟹ B ⟹ B ∧ A
```

Listing E.14: Backward Proof: Output (Step 1)

In Listing E.12 (Step 2), the prover applies the assumption, which solves the first subgoal. `A` is immediately inferred from the assumption ⟦`A`; `B`⟧ and the subgoal `2.` becomes the only subgoal. In a similar way, at (Step 3), the prover applies the conjunction introduction rule and creates two new subgoals, as follows:

```
goal (2 subgoals):
1. A ⟹ B ⟹ B
2. A ⟹ B ⟹ A
```

Listing E.15: Backward Proof: Output (Step 3)

At (Step 4) and (Step 5), each application of the assumption automatically solves each subgoal. The keyword **done** associates the proved lemma with its given name, and Isabelle output is shown in Listing E.16. The output details that a new theorem, called `some_thm`, has been added to the Isabelle kernel. In the new theorem, the free variables `A` and `B` are replaced by the schematic variables `?A` and `?B`, respectively. Unlike free variables, schematic variables can be instantiated, i.e., they can be replaced by specific terms or expressions during the proof process. Any time from now on, when the prover knows two facts, `P` and `Q`, it can use this theorem to infer that `P ∧ (Q ∧ P)`.

```
theorem some_thm: ?A ⟹ ?B ⟹ ?A ∧ ?B ∧ ?A
```

Listing E.16: Backward Proof: Output (Step 6)

```
lemma "A ∨ B ⟹ B ∨ A"                goal (1 subgoal):
                                     1. A ∨ B ⟹ B ∨ A
    apply (erule disjE)              goal (2 subgoals):
                                     1. A ⟹ B ∨ A
                                     2. B ⟹ B ∨ A
    apply (rule disjI2)              goal (2 subgoals):
                                     1. A ⟹ A
                                     2. B ⟹ B ∨ A
    apply assumption                 goal (1 subgoal):
                                     1. B ⟹ B ∨ A
    apply (rule disI1)               goal (1 subgoal):
                                     1. B ⟹ B
    apply assumption                 goal:
                                     No subgoals!
done                                 theorem: "?A ∨ ?B ⟹ ?B ∨ ?A"
```

Listing E.17: Mixed Proof: Example

In Listing E.17, we show an example of mixed reasoning in Isabelle. On the left side, we show the Isabelle proof, and on the right side, we present the prover output. The application of the `erule` method is an example of forward reasoning, i.e., the assumption of the current goal `A ∨ B` is unified with the first assumption of the `disjE` rule `P ∨ Q`, and the rest of the assumptions of the rule become new subgoals. In the given example, `P` stands for `A`, `Q` for `B`, and `R` for `B ∨ A`. Upon application of the method (`erule disjE`), the original goal is replaced by two subgoals, one matching `P ⟹ R` and the other `Q ⟹ R`. From this point on, the proof uses backward reasoning. Note that the `apply` method always affects the first subgoal. Thus, the command `apply (rule disjI2)` applies to the first subgoal, i.e., `A ⟹ B ∨ A`, and transforms it into `A ⟹ A`. The next `apply assumption` command solves this goal automatically, therefore the only remaining goal is now `B ⟹ B ∨ A`, which is solved in a similar way.

In large-scale applications that involve complex mathematical proofs with potentially hundreds of steps, the sequential nature of `apply` scripts makes them hard to maintain, due to their lack of structure. To manage proofs in a more organised, maintainable and elegant way, we use *Isar*, which leverages Isabelle's full range of provers in a structured format (Ballarin, 2008; Nipkow, 2011; Wenzel *et al.*, 2021). E.g., a typical Isar proof for a generic goal `A₀ ⟹ C` is shown in Listing E.18. Provided that each subseqent proof step succeeds, the proof is successful.

```
proof
  assume "A₀"
  have "F₁" by ...
  ⋮
  have "Fₙ" by ...
  show "C" by ...
qed
```

Listing E.18: Generic Isar Proof

The `have` statements are intermediate steps to the `show` statement that proves the actual goal. In Listing E.19 we present the general syntax of Isar proofs, using BNF. Recall that a symbol enclosed between ⟨⟩ denotes a non-terminal symbol. An Isar proof can be *atomic* (described as the `by`... line) or it can contain a *proof block* (described as the `proof`...`qed` line).

⟨*proof*⟩ ::= `proof` [⟨*method*⟩] ⟨*statement*⟩* `qed` [⟨*method*⟩]
| `by` ⟨*method*⟩ [⟨*method*⟩]

⟨*method*⟩ ::= `(simp ...) | (blast ...) | (rule ...) | ...`

```
⟨statement⟩ ::= fix ⟨variable⟩+                    (⋀)
              | assume ⟨proposition⟩                (⟹)
              | [from ⟨name⟩+]
              | (have | show) ⟨propostion⟩ ⟨proof⟩
              | next                (separates subgoals)
```

Listing E.19: Isar Proof Syntax

An *atomic proof* is self-contained and completes in a single step without breaking down into sub-proofs. It directly establishes the validity of the statement by concluding the proof immediately, through the use of the command `by`. In contrast, a *block proof* is a sequence of logically connected proof steps. A proof block may introduce local variables, assumptions, and intermediate goals (i.e., conclusions). Local variables and assumptions are introduced with the keywords `fix` and `assume`, respectively. Intermediate goals are introduced with two commands: the `have` command that introduces intermediate results or lemmas, and the `show` command that indicates the final conclusion of the current block. Additional examples of Isar proofs are discussed in the remainder of this section.

When the `proof` command takes an argument of the form (`rule some−rule`), the prover applies the rule it states. The method `rule` can take a list of rules and applies the first matching rule in the list. Note that elimination rules are tried first. To simplify the proofs, the command `proof` automatically attempts to select a rule, based on the current goal and the Isabelle predefined list of introduction and elimination rules, which we introduced in Table 5.3. Thus, the `proof` command without arguments is an abbreviation for the command `proof (rule elim−rules intro−rules)`.

In Listing E.20, we show a simple Isar proof. In it, the application of the command `proof (rule conjI)` replaces the conclusion of the current goal (i.e., `A ∧ B`) with the two subgoals of the `conjI` rule (i.e., `A` and `B`). Each subgoal is individually and immediately proved by assumption. It is recommended that the Isar proofs are built in such a way that the proof of each proposition builds on the previous proposition. The previous proposition is referred to by the abbreviation `this`. When the proof is immediate (i.e., using the command `by assumption`), the command can be replaced by ".". In Listing E.21, we show the use of these two abbreviations, as a simplified version of Listing E.20. For a list of the abbreviations most commonly used in Isabelle proofs, we refer the reader to Table 5.4.

```
lemma "⟦A; B⟧ ⟹ A ∧ B"
proof (rule conjI)
  assume a: "A"
  from a show "A" by assumption
next
  assume b: "B"
  from b show "B" by assumption
qed
```

Listing E.20: Isar Example

```
lemma ⟦A; B⟧ ⟹ A ∧ B
proof
  assume "A"
  from this show "A" .
next
  assume "B"
  from this show "B" .
qed
```

Listing E.21: Isar Example

```
lemma
  assumes ab: "A ∨ B"
  shows "B ∨ A"
proof -
    from ab show ?thesis
    proof
        assume A thus ?thesis ..
    next
        assume B thus ?thesis ..
    qed
qed
```

Listing E.22: Rule Suppress

```
lemma
  assumes ab: "A ∨ B"
  shows "B ∨ A"
using ab
proof
    assume A thus ?thesis ..
next
    assume B thus ?thesis ..
qed
```

Listing E.23: Using Command

In some cases, it is necessary to suppress the implicit application of rules in the `proof` command. Given the goal `show` "`A ∨ B`", an implicit `proof` command applies the first $\vee$-introduction rule and requires to prove `A`, which may not be provable. To avoid this, the command `proof−` allows the suppression of any predefined Isabelle/Isar rules, as shown in Listing E.22.

The code can be even more simplified, with the `using` command, as shown in Listing E.23. The `using` command can be placed ahead of the `proof` command, allowing additional facts to be included alongside those already referenced in the proof context. In Listing E.23, the `using` command feeds the fact `ab` directly into the proof and triggers the application of the elimination rule. When using an `assumes-shows` block to state a proposition in Isar, the system implicitly introduces the keyword `assms`, which represents the list of assumptions introduced in that block. The assumptions can be referred to individually by `assms(1)`, `assms(2)`, etc., without the need to name each individually. E.g., in Listing E.23, the command `using` can take the argument `assms(1)`, with the same effect as the name argument `ab`.

Just as with unstructured proofs, in Isar proofs the method `rule` can apply forward and backward reasoning, using elimination and introduction rules, respectively. Forward reasoning is presented in Listing E.24. In it, the `proof` command picks an elimination rule, such that the first assumption of the rule unifies with the fact `ab`. Backward reasoning is presented in Listing E.25. In it, the `proof` command picks an introduction rule, and the conclusion of the rule must unify with the `have` goal (`A ∧ B`).

```
assume ab: "A ∧ B"
from ab have "..." proof
```

Listing E.24: Forward Reasoning

```
have "A ∧ B" proof
```

Listing E.25: Backward Reasoning

E.5 Commonly used proof patterns

So far, we have demonstrated several proof patterns using examples. In this section, we outline the most commonly used logic proof patterns, such as *case analysis*, *logical equivalence*, *contradiction*, *quantifiers*, and *set operations*.

For proving a statement `R`, *case analysis* patterns come in two forms: (i) beginning with a formula `P`, the reasoning considers `P` and ¬`P` and (ii) given a disjunctive fact `P ∨ Q`, the reasoning considers each disjunct `P`, `Q`, separately. The first pattern is described in Listing E.26 and the second in Listing E.27.

```
show "R"
proof cases
  assume "P"
  ⋮
  show "R" ⟨proof⟩
next
  assume "¬P"
  ⋮
  show "R" ⟨proof⟩
qed
```

Listing E.26: Case Analysis (i)

```
have "P ∨ Q ⟨proof⟩
then show "R"
proof
  assume "P"
  ⋮
  show "R" ⟨proof⟩
next
  assume "Q"
  ⋮
  show "R" ⟨proof⟩
qed
```

Listing E.27: Case Analysis (ii)

A *logical equivalence* pattern proof consists of two steps: first we assume the left side and we show that it implies the right side, then we swap the assumption and conclusion. The pattern is presented in Listing E.28

```
show "P ⟷ Q"
proof
  assume "P"
  ⋮
  show "Q" ⟨proof⟩
next
  assume "Q"
  ⋮
  show "P" ⟨proof⟩
qed
```

Listing E.28: Logical Equivalence

Contradiction patterns come in two forms, described in Listings E.29 and E.30. If we aim to prove ¬P, we assume P and derive a contradiction (i.e., `show False`). Alternatively, if we aim to prove P, we apply the `ccontr` rule, we assume ¬P and derive a contradiction.

```
show "¬P"
proof
  assume "P"
  ⋮
  show "False" ⟨proof⟩
qed
```

Listing E.29: Contradiction (¬P)

```
show "P"
proof (rule ccontr )
  assume "¬ P"
  ⋮
  show "False" ⟨proof⟩
qed
```

Listing E.30: Contradiction (`ccontr`)

In Listings E.31 and E.32, we detail the *quantified formula* patterns for the universal and existential quantifiers, respectively. In the universal quantifier pattern, the `fix x` statement introduces a locally fixed variable into the proof. This statement is equivalent to the well-known "arbitrary, but fixed value" used in mathematical proofs. Note that this new variable is different from the bound variable `x` in the quantifier formula. Internally, Isabelle renames it to ensure there is no confusion.

```
show "∀x. P x"
proof
  fix x
  ⋮
  show "P x" ⟨proof⟩
qed
```

Listing E.31: Universal Quantifier

```
show "∃x. P x"
proof
  ⋮
  show "P witness" ⟨proof⟩
qed
```

Listing E.32: Existential Quantifier

In the existential quantifier pattern, the `witness` is an arbitrary term for which we can prove that it satisfies `P`. If we need to reason forward from the ∃`x`. `P x` formula, we use the `obtain` command to introduce a witness for which `P` is satisfied. The wittness variable is usually also denoted by a local variable `x`. Note that, again, this new variable `x` is different from the bound variable of the existential quantifier and can have any other name.

Finally, in Listings E.33 and E.34, we detail two more patterns for **set operations**, for inclusion and equality, respectively.

```
show "A ⊆ B
proof
  fix x
  assume "x ∈ A"
  ⋮
  show "x ∈ B" ⟨proof⟩
qed
```

Listing E.33: Set Inclusion

```
show "A = B"
proof
  show "A ⊆ B" ⟨proof⟩
next
  show "B ⊆ A" ⟨proof⟩
qed
```

Listing E.34: Set Equality

For more details, we invite the reader to check the extensive Isabelle documentation and tutorials (Nipkow, 2025; Wenzel *et al.*, 2021; Nipkow *et al.*, 2021).

Bibliography

Amoroso, N. (2010). *The exposed city: mapping the urban invisibles*. Routledge.

Angele, J., Kifer, M., and Lausen, G. (2009). Ontologies in f-logic. In *Handbook on ontologies*, pages 45–70. Springer.

Antoniou, G., Batsakis, S., Mutharaju, R., Pan, J. Z., Qi, G., Tachmazidis, I., Urbani, J., and Zhou, Z. (2018). A survey of large-scale reasoning on the web of data. *The Knowledge Engineering Review*, **33**.

AstroML (2021). AstroML: Machine learning and data mining for astronomy. http://www.astroml.org. Accessed: April. 10, 2025.

Baader, F., Calvanese, D., McGuinness, D., Patel-Schneider, P., Nardi, D., *et al.* (2003). *The description logic handbook: Theory, implementation and applications*. Cambridge university press.

Baader, F., Koopmann, P., Michel, F., Turhan, A.-Y., and Zarrieß, B. (2022). Efficient TBox Reasoning with Value Restrictions using the wer Reasoner. *Theory and Practice of Logic Programming*, **22**(2), 162–192.

Ballarin, C. (2008). Introduction to the isabelle proof assistant. https://www21.in.tum.de/~ballarin/belgrade08-tut/. Accessed: April. 10, 2025.

Ballarin, C. (2010). Tutorial to locales and locale interpretation. In *Contribuciones científicas en honor de Mirian Andrés Gómez*, pages 123–140. Universidad de La Rioja.

Barendregt, H. and Wiedijk, F. (2005). The challenge of computer mathematics. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, **363**(1835), 2351–2375.

Baumann, P., Mazzetti, P., Ungar, J., Barbera, R., Barboni, D., Beccati, A., Bigagli, L., Boldrini, E., Bruno, R., Calanducci, A., *et al.* (2016). Big data analytics for earth sciences: the earthserver approach. *International Journal of Digital Earth*, **9**(1), 3–29.

Ben-Ari, M. (2012). *Mathematical logic for computer science*. Springer Science & Business Media.

Benzmüller, C., Cheikhrouhou, L., Fehrer, D., Fiedler, A., Huang, X., Kerber, M., Kohlhase, M., Konrad, K., Meier, A., Melis, E., *et al.* (1997). Ω: Towards a mathematical assistant. In *International Conference on Automated Deduction*, pages 252–255. Springer.

Bhatia, J., Evans, M. C., and Breaux, T. D. (2019). Identifying incompleteness in privacy policy goals using semantic frames. *Requirements Engineering*, **24**(3), 291–313.

Birkhoff, G. and MacLane, S. (1941). *A Survey of Modern Algebra*. Macmillan, New York.

Borgida, A. and Patel-Schneider, P. F. (1993). A semantics and complete algorithm for subsumption in the CLASSIC description logic. *Journal of Artificial Intelligence Research*, **1**, 277–308.

Borgo, S. and Hitzler, P. (2018). Some Open Issues After Twenty Years of Formal Ontology. In *FOIS*, pages 1–9.

Brachman, R. and Levesque, H. (2004). *Knowledge Representation and Reasoning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Brucker, A. D., Ait-Sadoune, I., Crisafulli, P., and Wolff, B. (2018). Using the isabelle ontology framework. In *International Conference on Intelligent Computer Mathematics*, pages 23–38. Springer.

Burton-Jones, A., Storey, V. C., Sugumaran, V., and Ahluwalia, P. (2005). A semiotic metrics suite for assessing the quality of ontologies. *Data & Knowledge Engineering*, **55**(1), 84–102.

Calvanese, D. and Franconi, E. (2018). First-order ontology mediated database querying via query reformulation. In *A Comprehensive Guide Through the Italian Database Research Over the Last 25 Years*, pages 169–185. Springer.

Chantrapornchai, C. and Choksuchat, C. (2016). Ontology construction and application in practice case study of health tourism in Thailand. *SpringerPlus*, **5**(1), 2106.

Ciesielski, K. (1997). *Set theory for the working mathematician*. Cambridge University Press.

Codd, E. (1970). A relational model of data for large shared databanks. *ACM*, **13**, 377–387.

Davey, B. and Priestly, H. (1990). *Introduction to Lattices and Order*. Cambridge University Press.

De Giacomo, G., Lembo, D., Lenzerini, M., Poggi, A., and Rosati, R. (2018). *Using Ontologies for Semantic Data Integration*, pages 187–202. Springer International Publishing, Cham.

de Haan, E., Padigar, M., El Kihal, S., Kübler, R., and Wieringa, J. E. (2024). Unstructured data research in business: Toward a structured approach. *Journal of Business Research*, **177**, 114655.

De Jong, T. and Ferguson-Hessler, M. G. (1996). Types and qualities of knowledge. *Educational psychologist*, **31**(2), 105–113.

De Nicola, A. and Missikoff, M. (2016). A lightweight methodology for rapid ontology engineering. *Communications of the ACM*, **59**(3), 79–86.

De Nicola, A., Missikoff, M., and Navigli, R. (2009). A software engineering approach to ontology building. *Information systems*, **34**(2), 258–275.

Di Pinto, F., De Giacomo, G., Lembo, D., Lenzerini, M., and Rosati, R. (2019). Acquiring ontology axioms through mappings to data sources. *Future Internet*, **11**(12), 260.

Donini, F. M., Lenzerini, M., Nardi, D., and Schaerf, A. (1996). Reasoning in description logics. *Principles of knowledge representation*, **1**, 191–236.

Ehrig, H. and Mahr, B. (1985). *Fundamentals of Algebraic Specifications*. Springer.

Ehrig, H., Kreowski, H.-J., Mahr, B., and Padawitz, P. (1982). Algebraic implementation of abstract data types. *Theoretical Computer Science*, **20**(3), 209–263.

Ekaputra, F., Sabou, M., Serral Asensio, E., Kiesling, E., and Biffl, S. (2017). Ontology-based data integration in multi-disciplinary engineering environments: A review. *Open Journal of Information Systems*, **4**(1), 1–26.

Fernández-López, M., Gómez-Pérez, A., and Juristo, N. (1997). Methontology: from ontological art towards ontological engineering. In *Proc. Symposium on Ontological Engineering of AAAI*.

Ganter, B., Rudolph, S., and Stumme, G. (2019). Explaining data with formal concept analysis. In *Reasoning web. Explainable artificial intelligence*, pages 153–195. Springer.

Genesereth, M. R., Fikes, R. E., *et al.* (1992). *Knowledge interchange format-version 3.0: reference manual*. Citeseer.

Gerwin Klein, June Andronick, T. M. (2024). Type classes & locales. http://www.cse.unsw.edu.au/~cs4161/11s2/week12A_4p.pdf.

Goczyła, K., Waloszek, A., and Waloszek, W. (2009). Algebra of ontology modules for semantic agents. In *International Conference on Computational Collective Intelligence*, pages 492–503. Springer.

Gómez-Pérez, A., Fernández-López, M., and Corcho, O. (2006). *Ontological Engineering: with examples from the areas of Knowledge Management, e-Commerce and the Semantic Web*. Springer Science & Business Media.

Gordon, M. J., Milner, A. J., and Wadsworth, C. P. (1979). *Edinburgh LCF: a mechanised logic of computation*. Springer.

Grimm, S. (2009). Knowledge representation and ontologies. In *Scientific data mining and knowledge discovery: principles and foundations*, pages 111–137. Springer.

Gross, J. L., Yellen, J., and Anderson, M. (2018). *Graph theory and its applications*. Chapman and Hall/CRC.

Gruber, T. R. (1995). Toward principles for the design of ontologies used for knowledge sharing? *International journal of human-computer studies*, **43**(5-6), 907–928.

Grüninger, M., Chow, A., and Wong, J. (2023). Semiautomatic design of ontologies. In *IFIP Working Conference on The Practice of Enterprise Modeling*, pages 143–158. Springer.

Guarino, N., Oberle, D., and Staab, S. (2009). What is an ontology? In *Handbook on ontologies*, pages 1–17. Springer.

Harper, R. (2016). *Practical foundations for programming languages*. Cambridge University Press.

Harrison, J. (2009). HOL light: An overview. In *International Conference on Theorem Proving in Higher Order Logics*, pages 60–66. Springer.

Harrison, J., Urban, J., and Wiedijk, F. (2014). History of Interactive Theorem Proving. In *Computational Logic*, volume 9, pages 135–214.

Hatcher, W. and Hebert, M. (1993). *Model Theory*.

Hirsch, R. (2007). Relation algebra reducts of cylindric algebras and complete representations. *The Journal of Symbolic Logic*, **72**(2), 673–703.

Hodges, W., Wilfrid, H., *et al.* (1993). *Model theory*. Cambridge university press.

HOL (2012). The HOL system - university of cambridge. http://www.cl.cam.ac.uk/research/hvg/HOL/. Accessed: April. 10, 2025.

Hong, J. L. (2016). Automated data extraction with multiple ontologies. *International Journal of Grid and Distributed Computing*, **9**(6), 381–392.

Horrocks, I. (1998). Using an expressive description logic: FaCT or fiction? *KR*, **98**, 636–645.

IMDb (2020). IMDb Datasets. http://www.imdb.com/interfaces/. Accessed: April. 10, 2025.

Imielinski, T. and Lipski, W. (1984). The relational model of data and cylindric algebras. *Journal of Computer and System Sciences*, **28**, 80–102.

Isabelle (2025). Isabelle: A generic interactive proof assistant. http://isabelle.in.tum.de/index.html. Accessed: April. 10, 2025.

Jackermeier, M., Chen, J., and Horrocks, I. (2023). Box2EL: Concept and role box embeddings for the description logic el++. *arXiv preprint arXiv:2301.11118*.

Jaskolka, J., MacCaull, W., and Khedri, R. (2015). Towards an ontology design architecture. In *2015 International Conference on Computational Science and Computational Intelligence (CSCI)*, pages 132–135. IEEE.

Kant, I. (1908). Critique of pure reason. 1781. *Modern Classical Philosophers, Cambridge, MA: Houghton Mifflin*, pages 370–456.

Kendall, E. F. and McGuinness, D. L. (2019). Ontology engineering. *Synthesis Lectures on The Semantic Web: Theory and Technology*, **9**(1), i–102.

Kohlas, J. and Schmid, J. (2014). An algebraic theory of information: An introduction and survey. *Information*, **5**(2), 219–254.

Kotis, K. I., Vouros, G. A., and Spiliotopoulos, D. (2020). Ontology engineering methodologies for the evolution of living and reused ontologies: status, trends, findings and recommendations. *The Knowledge Engineering Review*, **35**.

Krötzsch, M., Simancik, F., and Horrocks, I. (2012). A description logic primer. *arXiv preprint arXiv:1201.4089*.

Kunčar, O. and Popescu, A. (2015). A consistent foundation for Isabelle/HOL. In *Interactive Theorem Proving: 6th International Conference, ITP 2015, Nanjing, China, August 24-27, 2015, Proceedings 6*, pages 234–252. Springer.

Le Clair, A., Marinache, A., El Ghalayini, H., MacCaull, W., and Khedri, R. (2022). A review on ontology modularization techniques: a multi dimensional perspective. *IEEE Transactions on Knowledge and Data Engineering*.

Leadbetter, A., Smyth, D., Fuller, R., O'Grady, E., and Shepherd, A. (2016). Where big data meets linked data: Applying standard data models to environmental data streams. In *Big Data (Big Data), 2016 IEEE International Conference on*, pages 2929–2937. IEEE.

LeClair, A., Khedri, R., and Marinache, A. (2019). Toward Measuring Knowledge Loss due to Ontology Modularization. In *Proceedings of the 11th International Joint Conference on Knowledge Discovery, Knowledge Engineering and Knowledge Management - Volume 2: KEOD*, pages 172–184. INSTICC, SciTePress.

LeClair, A., Khedri, R., and Marinache, A. (2020). Formalizing graphical modularization approaches for ontologies and the knowledge loss. In *International Joint Conference on Knowledge Discovery, Knowledge Engineering, and Knowledge Management*, pages 388–412. Springer.

Lenzerini, M., Milano, D., and Poggi, A. (2004). Ontology representation & reasoning. *Universit di Roma La Sapienza, Roma, Italy, Tech. Rep. NoE InterOp (IST-508011)*.

Loveland, D. W. (2016). *Automated theorem proving: A logical basis*. Elsevier.

Lukácsy, G. and Szeredi, P. (2009). Efficient description logic reasoning in Prolog: the DLog system. *Theory and Practice of Logic Programming*, **9**(3), 343–414.

MacGregor, R. M. (1994). A description classifier for the predicate calculus. In *AAAI*, volume 94, pages 213–220.

Madni, A. M., Lin, W., and Madni, C. C. (2001). IDEON: An extensible ontology for designing, integrating, and managing collaborative distributed enterprises. *Systems Engineering*, **4**(1), 35–48.

Marinache, A. (2016). *On the Structural Link Between Ontologies and Organised Data Sets*. Master's thesis, McMaster University, Hamilton, ON, Canada.

Marinache, A., Khedri, R., and MacCaull, W. (2019). A Data-Centered Framework for Domain Knowledge Representation. Technical report, McMaster University.

Marinache, A., Khedri, R., LeClair, A., and MacCaull, W. (2021). DIS: A data-centred knowledge representation formalism. In *2021 Reconciling Data Analytics, Automation, Privacy, and Security: A Big Data Challenge (RDAAPS)*, pages 1–8. IEEE.

Marinache, A., Khedri, R., and MacCaull, W. (2025a). Bridging data and knowledge: A roadmap from Domain Information Systems (DIS) Theory to Practical Reasoning. To be Submitted for publication.

Marinache, A., Khedri, R., and MacCaull, W. (2025b). Domain Information System (DIS): From theory to semantics. Technical Report CAS-25-02-RK, McMaster University.

Marinache, A., Khedri, R., and MacCaull, W. (2025c). Domain Information System (DIS): Specification and automation. Technical Report CAS-25-01-RK, McMaster University.

Marker, D. (2000). *Model Theory: An Introduction*. Springer.

Matentzoglu, N., Leo, J., Hudhra, V., Sattler, U., and Parsia, B. (2015). A survey of current, stand-alone OWL reasoners. In *ORE*, pages 68–79.

Mizoguchi, R. (2003). Part 1: Introduction to ontological engineering. *New generation computing*, **21**(4), 365–384.

Mizoguchi, R. (2019). Knowledge engineering. In *Ontology Makes Sense*, pages 69–81.

Motik, B. (2009). Resolution-based reasoning for ontologies. In *Handbook on Ontologies*, pages 529–550. Springer.

Motik, B., Shearer, R., and Horrocks, I. (2009). Hypertableau reasoning for description logics. *Journal of Artificial Intelligence Research*, **36**, 165–228.

Najafabadi, M. M., Villanustre, F., Khoshgoftaar, T. M., Seliya, N., Wald, R., and Muharemagic, E. (2015). Deep learning applications and challenges in big data analytics. *Journal of Big Data*, **2**(1), 1.

Nativi, S., Mazzetti, P., Santoro, M., Papeschi, F., Craglia, M., and Ochiai, O. (2015). Big data challenges in building the global earth observation system of systems. *Environmental Modelling & Software*, **68**, 1–26.

Nawaz, M. S., Malik, M., Li, Y., Sun, M., and Lali, M. (2019). A survey on theorem provers in formal methods. *arXiv preprint arXiv:1912.03028*.

Nipkow, T. (2011). A tutorial introduction to structured isar proofs.

Nipkow, T. (2025). Programming and proving in Isabelle/HOL. https://isabelle.in.tum.de/doc/prog-prove.pdf. Accessed: April. 28, 2025.

Nipkow, T. and Roßkopf, S. (2021). Isabelle's metalogic: Formalization and proof checker. In *CADE*, pages 93–110.

Nipkow, T., Wenzel, M., and Paulson, L. C. (2021). *Isabelle/HOL: a proof assistant for higher-order logic*. Springer. Accessed: April. 10, 2025.

NuPRL (2014). Proof/program refinement logic. https://nuprl-web.cs.cornell.edu/. Accessed: April. 10, 2025.

OMDbAPI (2019). The Open Movie Database. http://www.omdbapi.com/. Accessed: April. 10, 2025.

Patel, A. and Jain, S. (2018). Formalisms of representing knowledge. *Procedia Computer Science*, **125**, 542–549.

PhoX (2024). The phox proof assistant. https://raffalli.eu/phox/index.html. Accessed: April. 10, 2025.

PVS (2023). PVS specification and verification system. http://pvs.csl.sri.com. Accessed: April. 10, 2025.

Quantz, J. and Kindermann, C. (1990). *Implementation of the BACK-system version 4*. Technische Universitaet Berlin.

Ringer, T., Palmskog, K., Sergey, I., Gligoric, M., Tatlock, Z., *et al.* (2019). QED at large: A survey of engineering of formally verified software. *Foundations and Trends® in Programming Languages*, **5**(2-3), 102–281.

ROCQ (2025). The rocq prover. https://rocq-prover.org/. Accessed: April. 10, 2025.

Sankappanavar, H. P. and Burris, S. (1981). A course in universal algebra. *Graduate Texts Math*, **78**, 56.

Sattar, A., Surin, E. S. M., Ahmad, M. N., Ahmad, M., and Mahmood, A. K. (2020). Comparative analysis of methodologies for domain ontology development: A systematic review. *International Journal of Advanced Computer Science and Applications*, **11**(5).

Sayed Ahmed, T. (2022). Notions of representability for cylindric algebras: some algebras are more representable than others. *Periodica Mathematica Hungarica*, pages 1–35.

Schneider, T. and Šimkus, M. (2020). Ontologies and data management: a brief survey. *KI-Künstliche Intelligenz*, **34**(3), 329–353.

Scott, D. S. (1993). A type-theoretical alternative to ISWIM, CUCH, OWHY. *Theoretical Computer Science*, **121**(1-2), 411–440.

Smith, B. (2006). Against idiosyncrasy in ontology development. *Frontiers in Artificial Intelligence and Applications*, **150**, 15.

Song, W., Spencer, B., and Du, W. (2011). Hybrid reasoning for ontology classification. In *Canadian Conference on Artificial Intelligence*, pages 372–376. Springer.

Sowa, J. F. (2014). *Principles of semantic networks: Explorations in the representation of knowledge*. Morgan Kaufmann.

Stojanovic, L. (2004). *Methods and tools for ontology evolution*. PhD thesis, University of Karlsruhe.

Suárez-Figueroa, M. C., Gómez-Pérez, A., and Fernández-López, M. (2012). The NeOn methodology for ontology engineering. In *Ontology engineering in a networked world*, pages 9–34. Springer.

Sure, Y., Staab, S., and Studer, R. (2004). On-to-knowledge methodology (OTKM). In *Handbook on ontologies*, pages 117–132. Springer.

Tahrat, S., Braun, G., Artale, A., Gario, M., and Ozaki, A. (2020). Automated reasoning in temporal DL-Lite. *arXiv preprint arXiv:2008.07463*.

Tarski, A., Henkin, L., and Monk, J. (1971). *Cylindric Algebras*. North-Holland.

Tudorache, T. (2020). Ontology engineering: Current state, challenges, and future directions. *Semantic Web*, **11**(1), 125–138.

Van Harmelen, F., Lifschitz, V., and Porter, B. (2008). *Handbook of knowledge representation*. Elsevier.

Vassiliadis, P., Zarras, A. V., and Skoulis, I. (2015). How is life for a table in an evolving relational schema? Birth, death and everything in between. In *International Conference on Conceptual Modeling*, pages 453–466. Springer.

W3C (2025). World wide web consortium (W3C). https://www.w3.org/. Accessed: April. 10, 2025.

Wang, Y., Chen, Y., Alomair, D., and Khedri, R. (2022). DISEL: A Language for Specifying DIS-Based Ontologies. In *Knowledge Science, Engineering and Management (KSEM)*, volume 13369 of *Lecture Notes in Artificial Intelligence*, pages 155–171. Springer.

Wenzel, M. *et al.* (2021). The isabelle/isar reference manual. http://isabelle.in.tum.de/dist/Isabelle2021-1/doc/isar-ref.pdf. Accessed: April. 10, 2025.

Wenzel, M. and Wolff, B. (2007). Building formal method tools in the isabelle/isar framework. In *International Conference on Theorem Proving in Higher Order Logics*, pages 352–367. Springer.

Westhofen, L., Neurohr, C., Butz, M., Scholtes, M., and Schuldes, M. (2022). Using ontologies for the formalization and recognition of criticality for automated driving. *IEEE Open Journal of Intelligent Transportation Systems*, **3**, 519–538.

Xiao, G., Calvanese, D., Kontchakov, R., Lembo, D., Poggi, A., Rosati, R., and Zakharyaschev, M. (2018). Ontology-based data access: a survey. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence*, pages 5511–5519. AAAI Press.

Zablith, F., Antoniou, G., d'Aquin, M., Flouris, G., Kondylakis, H., Motta, E., Plexousakis, D., and Sabou, M. (2015). Ontology evolution: a process-centric survey. *The knowledge engineering review*, **30**(1), 45–75.

Zhang, Y. and Zhao, Y. (2015). Astronomy in the big data era. *Data Science Journal*, **14**, 11–11.

Zombori, Z. (2008). Efficient two-phase data reasoning for description logics. In *IFIP International Conference on Artificial Intelligence in Theory and Practice*, pages 393–402. Springer.

Glossary

A

A-Box Assertional Box. 2, 17, 23, 40, 44, 75, 108

ATP Automated Theorem Prover. 26

B

BDD Binary Decision Diagram. 27

BNF Backus–Naur form. viii, 91, 92, 101, 216–219, 221, 228, 239

C

CA Cylindric Algebra. 15, 29

CG Conceptual Graphs. 14

CQ Competency Questions. 52, 73

CWA closed-world assumption. 13, 15

D

D-Box Data Box. 44

DDV Domain Data View. 6, 10, 30, 40, 41, 43, 47, 49, 51–54, 60–66, 68–75, 83, 85, 87, 90, 91, 95, 97–99, 102, 104, 108, 110, 111, 113, 114, 116–118, 140, 224

DIS Domain Information System. iv, vii, xi, 3–11, 13, 25, 26, 30–32, 40–45, 47–55, 60, 64, 67–75, 77, 83–87, 89–95, 98–100, 102–108, 110–114, 116–119, 140, 217, 223, 224

DL Description Logic. iv, 2, 3, 7, 8, 14, 16–18, 21, 23–25, 28–30, 40, 72, 75, 106, 108, 110, 112

DOnt Domain Ontology. 6, 10, 11, 30, 40–43, 47, 49, 51, 52, 65–67, 69–72, 74, 75, 83, 85, 87, 89–91, 98–100, 102, 104, 108, 116, 118, 140, 224

F

F-Logic Frame Logic. 16, 28, 29

FCA Formal Concept Analysis. 13, 14

FOL First Order Logic. 7–9, 14, 16, 17, 23, 24, 29, 77–80, 229

H

HOL Higher Order Logic. 9, 26, 30, 77, 228

I

IA Information Algebra. 15

Isabelle/HOL generic Higher-Order Logic proof assistant Isabelle. 5, 7–11, 32, 47, 83, 85, 90, 101, 106, 110, 114, 117, 118, 149, 228

ISO 15926 ISO Standard 15926 - Integration of life-cycle data for process plants including oil and gas production facilities. 20

ITP Interactive Theorem Prover. 26–28, 30, 77, 117, 228

K

KB Knowledge Base. 18, 23–25, 108

KIF Knowledge Interchange Format. 16, 17

KRR Knowledge Representation and Reasoning. 1, 7, 11

L

LCF Logic of Computable Functions. 26–28, 30

M

ML Meta Language. 26–28, 93

O

OBDA Ontology-based Data Access. 75

OCaml Objective Caml. 27, 28

OMDb Open Movie Database. 51

OO Object-Oriented. 20

OTKM On-To-Knowledge Methodology. 21

OWL Web Ontology Language. 16, 21, 25

P

PIDE Prover IDE. 78

PVS Prototype Verification System. 28

R

R-Box Rule Box. 2

RA Relational Algebra. 15, 29, 224, 226

RDF Resource Description Framework. 16

RDFS Resource Description Framework Schema. 16

S

SAT Boolean Satisfiability. 27

SEADOO SEmi-Automatic Design Of Ontologies. 22, 104

SMT Satisfiability Modulo Theory. 27

SPARQL SPARQL Protocol and RDF Query Language. 16

T

T-Box Terminological Box. 2, 17, 23, 25, 40, 44, 75, 108

U

UPON Lite Unified Process for ONtology building. 22

W

W3C World Wide Web Consortium. 16

Z

ZF Zermelo-Fraenkel set theory. 77