Structure Transfer: an Inference-Based Calculus for the Transformation of Representations

Daniel Raggi daniel.raggi@cl.cam.ac.uk
Gem Stapleton ges55@cam.ac.uk
Mateja Jamnik mateja.jamnik@cl.cam.ac.uk
University of Cambridge, Cambridge, UK

Aaron Stockdill a.a.stockdill@sussex.ac.uk
Grecia Garcia Garcia g.garcia-garcia@sussex.ac.uk
Peter C-H. Cheng p.c.h.cheng@sussex.ac.uk
University of Sussex, Brighton, UK

Abstract

Representation choice is of fundamental importance to our ability to communicate and reason effectively. A major unsolved problem, addressed in this paper, is how to devise *representational-system (RS) agnostic* techniques that drive representation transformation and choice. We present a novel calculus, called *structure transfer*, that enables representation transformation across diverse RSs. Specifically, given a *source* representation drawn from a source RS, the rules of structure transfer allow us to generate a *target* representation for a target RS. The generality of structure transfer comes in part from its ability to ensure that the source representation and the generated target representation satisfy *any* specified relation (such as semantic equivalence). This is done by exploiting *schemas*, which encode knowledge about RSs. Specifically, schemas can express *preservation of information* across relations between any pair of RSs, and this knowledge is used by structure transfer to derive a structure for the target representation which ensures that the desired relation holds. We formalise this using Representational Systems Theory (Raggi, Stapleton, Stockdill, Jamnik, Garcia, & Cheng, 2023), building on the key concept of a *construction space*. The abstract nature of construction spaces grants them the generality to model RSs of diverse kinds, including formal languages, geometric figures and diagrams, as well as informal notations. Consequently, structure transfer is a system-agnostic calculus that can be used to identify alternative representations in a wide range of practical settings.

1. Introduction

Symbols mediate how we store, use, and transmit knowledge. We invent, carve, utter and manipulate symbols. We let our symbols make predictions and solve problems for us, and we create machines which we command through symbols we made just for them. Symbols are organised into *systems*, comprising a set of symbols along with their natural and prescribed relations to each other (Palmer, 1978; Barwise & Etchemendy, 2019; Shimojima, 1999). Symbolic systems change, and new notations are often conceived. New programming languages are created, each time with a claim of supremacy on *some* aspect. Educators and communicators look for new and more effective ways of presenting information. Given their utility and complexity, we place enormous value on the development of symbolic systems, and in the knowledge of how to use them effectively and what to use them for. But with respect to both logic and cognition, there is no one system of symbols that works best for *every* task, and often a change of representation can determine whether we understand our task, and ultimately whether we can perform it effectively (Polya, 1957; Newell, Simon, et al., 1972; Cheng, Lowe, & Scaife, 2001). Thus, given any complex task for which the use of symbols is required, we face a large variety of symbolic systems – each with its own advantages – and it is up to us to decide which one to use, how to represent our task in it, and how to transfer potentially useful knowledge across systems. To be able to do all of this effectively we need to understand the connections between systems.

The major contribution of this paper is a calculus for transforming any given representation from a *source* system into another representation in a *target* system. This is done using only the structure of the given representation and knowledge about invariants – that is, homomorphisms – between the source and target systems. We formalise our approach using Representational Systems Theory (RST) (Raggi et al., 2023) and, within this context, a representation is taken to be a syntactic entity whose structure can be encoded by RST. The specific research questions we address are:

RQ1: Is there a general calculus for transforming a given representation into a new representation in such a way that any specified relation is guaranteed to hold between them? We show this is possible using the theory of *schemas* developed in this paper.

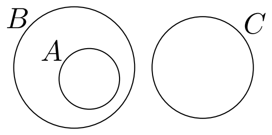
RQ2: Is it possible to do this rigorously and under limited or uncertain knowledge? Our approach explicitly includes derivation of new facts, extending the knowledge base. Moreover, even when insufficient facts are available, partial transformations are supported. The inference framework allows fuzzy or multi-valued logics for reasoning under uncertainty – and still produce desirable transformations.

As a result of the calculus presented in this paper we can transform between the representations of any two representational systems across any relation, given some knowledge of the invariants across systems, that is, knowledge about *information being preserved through relations*. This has wide ranging applications, demonstrating the potential significance and impact of our new theory for scientific development, communication, and education. Such applications include automatically generating diagrams and figures from formal languages, or improving human-computer interaction in scientific software, theorem provers or computer algebra systems, and enabling creative problem-solving in machines – which often requires the solver to consider different representations of the problem. We will also address the connections and applications of our calculus to some problems of related areas, from formal methods to analogy, demonstrating the generality of our approach. While our methods are general, for concreteness we will focus primarily on one illustrative application: that of *depiction and observation*, where a given statement is depicted graphically, and then consequences of the original statement are observed from the depiction. This is of particular interest to us because it demonstrates the notion of the *observational advantages* (Stapleton, Jamnik, & Shimojima, 2017) of some representations over others.

1.1 Example: the depict-and-observe process

Consider the statement: *A is contained in B, and B is disjoint from C*, otherwise written formally as $A \subseteq B \wedge B \cap C = \emptyset$. There are infinitely-many conclusions that we could derive from that statement. Of course, the reader may be guessing where we are going: $A \cap C = \emptyset$. This is a simple and obvious semantic consequence which is not syntactically explicit.

We could try to work out how the premise, $A \subseteq B \wedge B \cap C = \emptyset$, can be manipulated using inference rules, to conclude $A \cap C = \emptyset$. However, it is probably fair to assume that most readers will *interpret* the statement and imagine a depiction of the premises, similar to the figure shown here, from which the conclusion, $A \cap C = \emptyset$, can be observed without any additional syntactic manipulations.



Those with a formal mathematical education will easily produce some form of sentential argument wherein we introduce and eliminate symbols such as $\in$ and $\wedge$ (or the words 'in' and 'and'.). The sentential style of formal reasoning has been investigated exhaustively to the point where many methods and implementations exist for producing such arguments. However, to investigate and produce rigorous methods that formalise the process of depicting a picture and observing a conclusion from it, we need different tools. Specifically, we need to be able to encode diagrams over the same foundations in which we encode typical formal languages, and we need to be able to specify rigorously the relations between diagrammatic systems and sentential/linguistic systems. This, we contend, is provided by RST.

In this paper we will use the depict-and-observe process as a working example. This will demonstrate how RST can be used to understand, formally, the whole process from depicting the premises to observing the conclusion, using transformations between encodings of Set Algebra and Euler diagrams. Furthermore, we will illustrate the generality of our methods with some examples of a different nature.

1.2 How are different systems connected?

The connections between symbolic systems are sometimes built-in by design (e.g., a high-level programming language and an assembly language), but sometimes they need to be discovered (e.g., the algebraic vs geometric representations of complex numbers). Some connections may underlie cognitive processes without being explicitly encoded into our external symbolic systems, though these are often manifested in our complex and consistent uses of metaphor and analogy (Lakoff & Johnson, 2008), even in mathematics (Lakoff & Núñez, 2000). Understanding these relations is fruitful, as it enables the transference of knowledge and expertise from one system to another. For example, in linear algebra, the discovery that linear maps can be represented as matrices (and, more importantly, that operations and properties of linear maps translate to simple operations and properties of matrices), allows us to apply computational methods to solve linear algebra problems. But many of the most commonly used and fruitful transformations exist outside of the purview of formal knowledge. For example, the use of diagrams is ubiquitous in reasoning and learning (Larkin & Simon, 1987; Barwise & Etchemendy, 1990; Ainsworth, 1999), and the benefits of diagrams are well-known (Cheng et al., 2001; Cheng, 2002, 2011) and have been studied in terms of *free rides* and *observational advantages* (Stapleton et al., 2017; Blake, Stapleton, Rodgers, & Touloumis, 2021), yet there is no general theory that satisfactorily reveals how to systematically and effectively transfer information between *any* pair of systems regardless of whether they are sentential or diagrammatic – and whether they are considered formal or *informal*.

The concepts and methods presented in this paper, built on the foundations of Representational Systems Theory (Raggi et al., 2023), exploit the capacity of RST to capture structure from diverse representational systems and extend it with the means for transforming representations. As we will see, the methods presented here are very general, not only because of the expressive generality inherited from RST, but because the transformation calculus is based on a simple principle: the *transfer schema*, a formalism for capturing *invariants across systems through arbitrary relations*, where transfer schema *applications* allow us to derive the structure of the transformed representation. Transfer schemas can capture analogies between concepts in different representational systems, but more specifically they express concisely that *some* information is preserved across relations.

In principle, any relation between the objects of two systems can be used for *encoding* some objects of one system into the other. But the encoding is only fruitful if it preserves information *desirably*. What this means is a question of pragmatics. If we are interested in the composition and application of linear maps and we have computational power at our disposal, then the canonical transformation that relates each linear map to a matrix is *desirable* because composition and application of linear maps is preserved – as matrix multiplication – across the transformation from linear maps into a field of matrices. Moreover, this structure is preserved *desirably* because matrix multiplication is a simple operation given the availability of computational power.

Invariants between algebraic structures are generally captured by the notion of *homomorphism*. These capture preservation of algebraic information (e.g., group operations, vector space structure, and so forth.). A homomorphism from a set A to a set B is a function, $f\colon A \to B$, for which an operation μ_A of A is *preserved* as some operation μ_B of B. This is captured by the equation $f(\mu_A(a_1,\ldots,a_n)) = \mu_B(f(a_1),\ldots,f(a_n))$, and illustrated here (right) by a 'commutative' diagram.

$$\begin{tikzcd}
A \arrow[r, "f"] & B \\
A \times \cdots \times A \arrow[u, "\mu_A"] \arrow[r, "{(f,\ldots,f)}"] & B \times \cdots \times B \arrow[u, "\mu_B"']
\end{tikzcd}$$

Of course, the map f and the operations μ_A and μ_B need not be functions and the diagram will still capture the core principle that information is preserved *through the arrows.* Thus we can replace f with arbitrary relations, $R_1,\ldots,R_n, R$, with arbitrary domains and codomains, as illustrated here (right). This version readily captures the well-known invariant between linear maps and matrices.

$$\begin{tikzcd}
A \arrow[r, "R"] & B \\
A_1 \times \cdots \times A_n \arrow[u, "\mu_A"] \arrow[r, "{(R_1,\ldots,R_n)}"] & B_1 \times \cdots \times B_n \arrow[u, "\mu_B"']
\end{tikzcd}$$

Consider the function $Mat\colon T \to M$ that encodes every linear map as a matrix, where T is a set of linear transformations and M is a set of matrices. Moreover, consider the function $Enc\colon V \to M$ that encodes any vector (x,y) as a matrix $\left[\begin{smallmatrix} x \\ y \end{smallmatrix}\right]$. Then, the well-known invariant across Mat and Enc is illustrated by the commutative diagram here (right).

$$\begin{tikzcd}
V \arrow[r, "Enc"] & M \\
T \times V \arrow[u, "\texttt{apply}"] \arrow[r, "{(Mat, Enc)}"] & M \times M \arrow[u, "\texttt{multiply}"']
\end{tikzcd}$$

As we will see, the concept of *transfer schema* captures invariants across systems at this level of generality[1], and it can be used for re-representation.

1. From a practical perspective, transfer schemas are even more general than this. For instance, the arities of source and target (e.g. $T \times V$ and $M \times M$ in the linear map/matrix example) are not required to be the same (2 in this case).

The calculus that we present in this paper, called *structure transfer*, uses transfer schemas to produce a structure in the target system whose existence guarantees that a desired relation holds between the source and target representations. Our approach has the following properties, which address research questions RQ1 and RQ2 stated above:

1. **Representational generality**: structure transfer is applicable to any system that can be captured by RST, thus inheriting its scope in terms of the variety of representational systems that we can apply it to. [RQ1]
2. **Relational generality**: any set of relations between the representations across a pair of systems can be used to perform a transformation, as transfer schemas can capture invariants across systems through arbitrary relations. [RQ1]
3. **Validity**: given a trusted knowledge-base, a transformation ensures that the desired relation holds between the source and target representations. [RQ1]
4. **Partiality and extendability**: schemas are used as units of knowledge. New facts are inferred in the process of structure transfer, which means that the map (knowledge base) only needs to be partially specified a-priori. Existing knowledge will be used in an attempt to produce the transformation, and even when unsuccessful in producing a verified transformation, it may produce a partial, or unverified-yet-correct, transformation. [RQ2]
5. **Logic-agnosticism**: we assume very little about the logic in which the schemas are encoded[2], which opens the door for the use of fuzzy or multi-valued logics to encode uncertain knowledge and still produce transformations under uncertainty. [RQ2]

Others have built systems for exploiting heterogeneity in ontologies and mathematical knowledge management (Kutz, Mossakowski, & Lücke, 2010; Rabe & Kohlhase, 2013; Mossakowski, Maeder, & Lüttich, 2007), and some have implemented tools for knowledge transference within theorem proving environments (Huffman & Kunčar, 2013; Raggi, Bundy, Grov, & Pease, 2016). Some heterogeneous systems have been built with a hardcoded transformation between two kinds of representations (Barwise & Etchemendy, 1992). Our use of RST makes our approach more general, as it is not committed to any *kind* of representation. A typical formal definition of *language* is as a set of *words*, defined as strings over a set of symbols, and a typical approach to defining the *grammar* of a language involves rules for composing and decomposing words, yielding some structure (usually a rooted tree). RST weakens these assumptions by abstracting the explicit nature of the *words*, which instead we call *tokens*, and focusing on structure and classification – that is, how tokens relate to one-another and what their *types* are. This treatment of representational systems opens the door for modelling representations typically consider informal, such as geometric figures, plots, and other kinds of diagrams. But more importantly, by taking these representations seriously and within the same meta-theory as more typical systems, it allows us to reason rigorously about the relations between representations within a system and the relations between representations across systems.

In this paper we will focus on *transformation* and the necessary theory for realising them, which includes concepts for effecting *inference* in RST. This is part of a wider project,

2. In fact, any logic that can be encoded within the RST framework may be used. This means that the logic used to derive a transformation can itself be diagrammatic, or atypical in other ways.

rep2rep, which has a main goal to understand and produce tools that enable *effective choice of representations*. Raggi et al. (2023) already introduced the foundations of Representational Systems Theory. Some of our previous work dealt with the problem of representation selection more informally (Raggi, Stapleton, Stockdill, Jamnik, Garcia, & Cheng, 2020) and some dealt with the cognitive aspect (Cheng, Garcia, Raggi, Stockdill, & Jamnik, 2021) of representation processing.

Overview of this paper First we will present preliminary knowledge and notations (Section 2) and an overview of RST (Section 3). The main contributions of this paper come in the sections that follow. In Section 4 we introduce the concept of a *schema*, for expressing invariants within and across representational systems, leading to the crucial concept of a *transfer schema application*. In Section 5 we present *structure transfer* with an algorithmic approach. In Section 6 we compare structure transfer to well-known methods, such as term rewriting, and we present some applications, including the use and discovery of analogies. We conclude in Section 7.

2. Preliminary Concepts and Conventions

Here we present a brief overview of standard concepts needed throughout the paper.

Functions Given a function, $f\colon X \to Y$, and a set $Z \subseteq X$, the **restriction** of f to Z is denoted by $f|_Z\colon Z \to Y$. The **image** of f with its domain restricted to Z is denoted $f[Z]$.

Graph Notation We extensively use directed, bipartite graphs, where the vertices and arrows (i.e., directed edges) are labelled. A **directed labelled bipartite graph**, or simply **graph**, is a tuple, $g = (V_1, V_2, A, iv, l_A, l_1, l_2)$, where: V_1 and V_2 are disjoint sets of **vertices**; A is a set of **arrows** that is disjoint from $V_1 \cup V_2$; $iv\colon A \to (V_1 \times V_2) \cup (V_2 \times V_1)$ is an **incident vertices function**; and $l_A\colon A \to L_A$, $l_1\colon V_1 \to L_1$, and $l_2\colon V_2 \to L_2$ are **labelling functions**, where L_A, L_1 and L_2 are sets of labels. We write $V_1(g)$, $V_2(g)$, and $A(g)$ for the sets of vertices and arrows in g and set $V = V(g) = V_1 \cup V_2$. The **neighbourhood** of a vertex, v, in g is the smallest subgraph of g containing v and all the arrows incident with v. We also use the standard concept of **subgraph**.

Graph Operations To perform operations on graphs, we need the notion of *compatibility*: graphs $g = (V_1, V_2, A, iv, l_A, l_1, l_2)$ and $g' = (V_1', V_2', A', iv', l_A', l_1', l_2')$ are **compatible** provided their argument-wise union, $(V_1 \cup V_1', V_2 \cup V_2', A \cup A', iv \cup iv', l_A \cup l_A', l_1 \cup l_1', l_2 \cup l_2')$, is a graph. Whenever g and g' are compatible, their **union**, $g \cup g' = (V_1 \cup V_1', ..., l_2 \cup l_2')$ and **intersection**, $g \cap g' = (V_1 \cap V_1', ..., l_2 \cap l_2')$ are graphs.

Graph Morphisms To define various morphisms from $g = (V_1, V_2, A, iv, l_A, l_1, l_2)$ to $g' = (V_1', V_2', A', iv', l_A', l_1', l_2')$, we exploit functions of the form $f\colon V_1 \cup V_2 \cup A \to V_1' \cup V_2' \cup A'$, where $f[V_1] \subseteq V_1'$, $f[V_2] \subseteq V_2'$ and $f[A] \subseteq A'$. We write $f\colon g \to g'$ to mean such a function. A **homomorphism** is a function, $f\colon g \to g'$, such that for any $a \in A$ if $iv(a) = (v_i, v_j)$, then $iv'(f(a)) = (f(v_i), f(v_j))$. The image, $f[g]$, of f is taken to be the subgraph of g' mapped to by g under f. A homomorphism, $f\colon g \to g'$, is **label-preserving** provided for any $x \in V_1 \cup V_2 \cup A$ it is the case that $l(x) = l'(f(x))$. We say f is **label-preserving up to** a set $W \subseteq V_1 \cup V_2 \cup A$ provided it preserves labels for $(V_1 \cup V_2 \cup A) \setminus W$, but not necessarily for W. A bijective homomorphism is an **isomorphism**. Graphs g and g' are

isomorphic whenever there exists an isomorphism $f\colon g \to g'$. A homomorphism, $f\colon g \to g'$ is a **monomorphism** provided f is an isomorphism from g to its image $f[g]$. Lastly, we need homomorphisms that map from, say, $g_1 \cup g_2$ to $h_1 \cup h_2$ that guarantees g_1 maps to h_1 and g_2 to h_2: more generally, given tuples, $(g_1, \ldots, g_n)$ and $(h_1, \ldots, h_n)$, we denote by $f\colon (g_1, \ldots, g_n) \to (h_1, \ldots, h_n)$, any homomorphism $f\colon g_1 \cup \ldots \cup g_n \to h_1 \cup \ldots \cup h_n$ such that the image of every g_i under f is contained in h_i, that is, $f[g_i] \subseteq h_i$.

3. Representational System Theory

Representational System Theory was developed to understand the structure of representations and to facilitate transformations between them (Raggi et al., 2023). Most definitions in this section are from Raggi et al. (2023); those without citations are new. In RST, a *representational system* comprises three spaces: grammatical, entailment and identification spaces. A *grammatical space* encodes the relationship between *tokens*, capturing which tokens build other tokens; for example, $a + 1$ is built from a, $+$ and 1. An *entailment space* encodes inferential relations between tokens, given some notion of entailment; for example, $a + 0$ can be rewritten to a given standard arithmetic entailment. An *identification space* encodes properties of the tokens; for example, 3 uses fewer pixels than $1 + 2$. To formalise these spaces, RST uses one fundamental abstraction: a *construction space.*

3.1 From Construction Spaces to Multi-Space Systems

We present an overview of construction spaces and associated concepts that are necessary for the original contributions of Sections 4 and 5. Building on those concepts, we introduce the novel definition of a *multi-space system.* Firstly, a construction space encodes information about how *tokens* (i.e., representations) relate to one another. To define construction spaces we need the foundational concepts of a *type system* and a *constructor specification.*

Definition 3.1. A **type system** is a pair, $T = (Ty, \leq)$, where Ty is a set whose elements are called **types**, and $\leq$ is a partial order over Ty. (Raggi et al., 2023)

Types encode classes of tokens which are, at a certain level of abstraction, indistinguishable. The partial order, $\leq$, captures hierarchies of abstraction relative to what is considered *relevant* within a representational system. For example, we can define the type `two` as an abstraction of instances of the numeral 2, including two occurrences of it in $2 + 2$. Whilst types `two` and `three` distinguish 2 and 3, a *supertype*, `numeral`, may capture both, along with the rest of the numerals. To encode this, we may define `two` $\leq$ `numeral` and `three` $\leq$ `numeral`.

Definition 3.2. A **constructor specification** over a type system, $T = (Ty, \leq)$, is a pair, $C = (Co, sig)$, where

1. Co is a set, disjoint from Ty, whose elements are called **constructors**,
2. sig is a function over Co such that, for any $c \in Co$, the **signature** of c, $sig(c) = ([\tau_1, \ldots, \tau_n], \tau)$, where $[\tau_1, \ldots, \tau_n]$ is a sequence of types, τ is a type, and $n > 0$.

Given $sig(c) = ([\tau_1, \ldots, \tau_n], \tau)$, we call $[\tau_1, \ldots, \tau_n]$ the **input types** of c, and τ the **output type** of c. (Raggi et al., 2023)

Constructors encode basic relations between representations, such as the fact that a, $+$ and 1 produce $a+1$ when configured in a certain way. But a constructor may also encode the fact that P and $\neg P \vee Q$ can be used to produce Q through an inference rule.

Example 3.1 (Set Algebra)**.** The type system, $(\mathit{Ty}, \leq)$, of Set Algebra is such that Ty includes types `var`, `const`, `setExp`, `unaryOp`, `binOp`, `binRel`, `formula`. This means that the tokens of Set Algebra will include set expressions (including variables and some constants), operator and relation symbols, and formulas. This is relatively typical in the specification (often called the signature) of a formal language. However, in Set Algebra we will also include many types in Ty, which would typically be considered as terms. For example, for each A, $A \cup B$ and $\emptyset \subseteq A$, we will introduce a type: `A`, `A_union_B` or `empty_subset_A`. This is a fairly atypical use of types in type theory, but it respects the token/type dichotomy of semiotics (Wetzel, 2018) and Information Flow theory (Barwise & Seligman, 1997), while still allowing us to use types for their more typical type-theoretic purposes using their subtype order, $\leq$. For example, $\texttt{A} \leq \texttt{var} \leq \texttt{setExp}$ means that any instance of type `A` will also fall under types `var` and `setExp`. Similarly, we have $\texttt{union} \leq \texttt{binOp}$, $\texttt{empty\_subset\_A} \leq \texttt{formula}$, and so forth.

The constructors of Set Algebra include `infixRel`, which infixes a binary relation to produce a formula, with signature $\mathit{sig}(\texttt{infixRel}) = ([\texttt{setExp}, \texttt{binRel}, \texttt{setExp}], \texttt{formula})$. Given the subtype order, `infixRel` will take as input things of more *specialised* types, such as $[\texttt{A}, \texttt{subset}, \texttt{empty\_union\_B}]$, and may yield output `A_subset_empty_union_B`, where $\texttt{A\_subset\_empty\_union\_B} \leq \texttt{formula}$.

Of course, we have not yet formalised what it means for a constructor to 'take inputs' or 'yield outputs'. In order to capture this relation between constructors and their inputs and outputs, we use the fundamental concept of a *structure graph*[3], whose vertices are labelled with constructors and types. The conditions in the definition of structure graph ensure that for every vertex labelled by a constructor, c, its input/output arrows are connected to tokens whose types are subtypes of those prescribed by the signature of c. Structure graphs generalise the notion of a syntax tree.

Definition 3.3. Let $C = (\mathit{Co}, \mathit{sig})$ be a constructor specification over a type system $(\mathit{Ty}, \leq)$. A **structure graph** for C is a graph, $(\mathit{To}, \mathit{Cr}, A, \mathit{iv}, \mathit{index}, \mathit{type}, \mathit{co})$, where

1. $\mathit{iv}\colon A \to (\mathit{To} \times \mathit{Cr}) \cup (\mathit{Cr} \times \mathit{To})$ is such that, for every $u \in \mathit{Cr}$:
   (a) u has exactly one outgoing arrow[4]: $|\mathit{out}_A(u)| = 1$,
   (b) u has at least one incoming arrow: $|\mathit{in}_A(u)| > 0$,
2. $\mathit{index}\colon A \to \mathbb{N}$ is a partial function,
3. $\mathit{type}\colon \mathit{To} \to \mathit{Ty}$ labels the elements of To, called **tokens**, with types, and
4. $\mathit{co}\colon \mathit{Cr} \to \mathit{Co}$ labels the elements of Cr, with constructors

3. Raggi et al. (2023) defines structure graphs from the concept of a *configuration*, which is a structure graph with exactly one vertex labelled by a constructor; we do not need this level of conceptual refinement. The definitions are trivially equivalent.
4. The sets of incoming and outgoing arrows of v in g are, respectively, $\mathit{in}_A(v) = \{a \in A\colon \exists v' \in V\ \mathit{iv}(a) = (v', v)\}$ and $\mathit{out}_A(v) = \{a \in A\colon \exists v' \in V\ \mathit{iv}(a) = (v, v')\}$.

such that for every $a \in A$, and $t \in To$, $u \in Cr$, with $sig(co(u)) = ([\tau_1, \ldots, \tau_n], \tau)$:

- $index|_{in_A(u)}$ is a bijection to $\{1, \ldots, n\}$ and $index|_{out_A(u)}$ is undefined,
- if $iv(a) = (t, u)$ then $type(t) \leq \tau_{index(a)}$,
- if $iv(a) = (u, t)$ then $type(t) \leq \tau$. (Adapted from Raggi et al. (2023).)

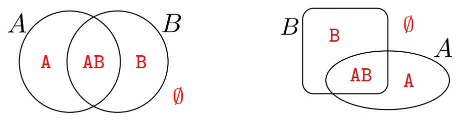
In summary, a structure graph contains tokens which are assigned types, and they connect through arrows to vertices labelled by constructors. Moreover, for any vertex, u, labelled by a constructor, c, its inputs and outputs respect the signature of c so that any token is assigned a type which is a subtype of the corresponding one in the signature of c.

It is important to note that in a structure graph each token, t, its assigned type, $type(t)$, is taken to be its minimum type. In general, we say that t is an instance of all supertypes of $type(t)$.

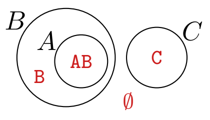
Example 3.2 (Euler Diagrams)**.** This example is based on the design of Euler diagrams described by Stapleton, Rodgers, Howse, and Zhang (2010). The tokens that represent Euler diagrams are comprised of a set of *simple closed curves* (e.g., circles) superimposed on a canvas. Each of which is assigned a distinct label. For this example we require that every curve gives rise to a pair of complementary non-empty regions: its interior and its exterior. This means that each diagram is partitioned into a set of minimal regions, called *zones*, and every zone can be characterised by a set of curves. Specifically, if L is the set of labels of a diagram, then a zone z is characterised by a set $Z \subseteq L$, such that z is interior to all the curves labelled by Z and exterior to the rest. For simplicity, zones will be written as words, for example, we write `AB` or `BA` to denote a zone which is interior to curves labelled by `A` and `B` and exterior to any other curves in the diagram. Thus, given a set of labels, L, a diagram can be described by a *set of zones* which, as we established, we write as a set of words from alphabet L.

This characterisation of diagrams is the basis for our type system, where we have one type `label`, and any element in L is a minimal type, subtype of `label`, and any set of words for L will be a minimal type and a subtype of type `diagram`. We illustrate that here (note that the tags for zones, in red, are there only for elucidation: they are not part of the diagrams):

Two tokens of type $\{\texttt{AB}, \texttt{A}, \texttt{B}, \emptyset\}$*:*

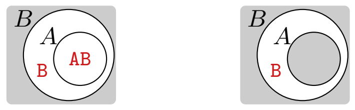


A token of type $\{\texttt{AB}, \texttt{B}, \texttt{C}, \emptyset\}$*:*

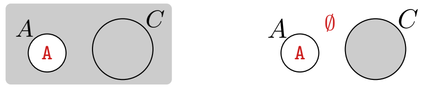


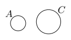
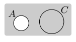
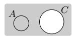
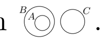
Moreover, for convenience, we see the regions of a diagram as tokens themselves, also assigned sets of zones as types. Thus, we have type `region`. Regions are abstractions for a given diagram, but here we will illustrate them by shading their complement:

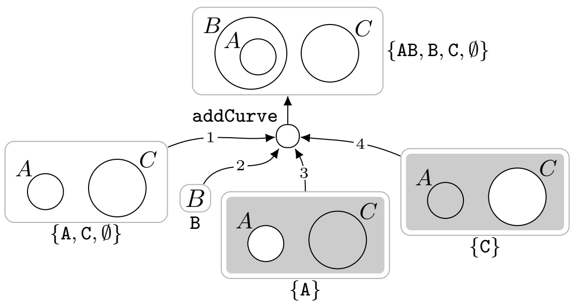
In the context of diagram of type $\{\texttt{AB}, \texttt{B}, \emptyset\}$*, we illustrate regions* $\{\texttt{AB}, \texttt{B}\}$ *(left) and* $\{\texttt{B}\}$ *(right):*

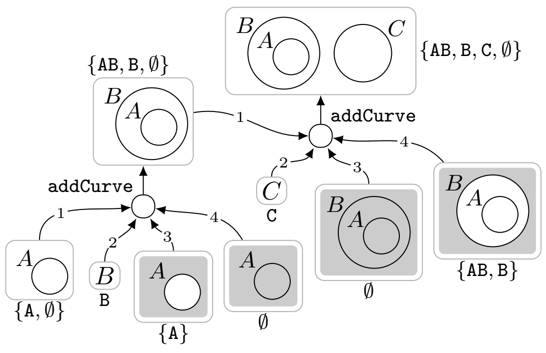


In the context of a diagram of type $\{\texttt{A}, \texttt{C}, \emptyset\}$*, we illustrate regions* $\{\texttt{A}\}$ *(left) and* $\{\texttt{A}, \emptyset\}$ *(right):*



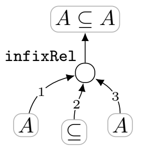
Given our type system, we can introduce constructor `addCurve`, with signature ([`diagram`, `label`, `region`, `region`], `diagram`), which takes four inputs and outputs a diagram. Intuitively, `addCurve` contains instructions for how to add a curve with a new label to a given diagram. The first and second inputs are the diagram and label for the new curve. The third and fourth inputs are the regions that should be, respectively, interior and exterior to the new curve being added. This information is sufficient to determine the type of the output diagram (Stapleton et al., 2010)[5]. Below we show two examples of structure graphs that use the constructor `addCurve`. On the left, we take a diagram  and draw a new curve labelled by B with the provision that the region  must be interior to B, and  must be exterior to B. This results in a construction of diagram . On the right, the same diagram is drawn in a different order, in addition to displaying an extra step.

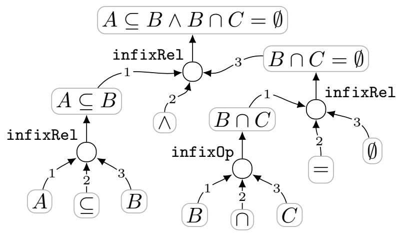


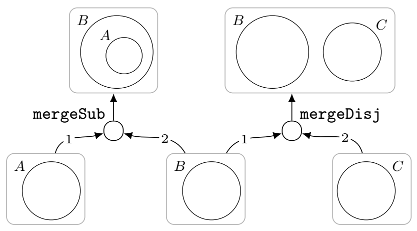


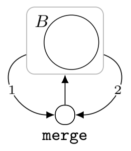
Structure graphs for Euler diagrams may also include other constructors, such as `merge`, `mergeSub` and `mergeDisj` which represent some more complex operations of Euler diagrams: merging two diagrams that may have overlapping labels, merging with the constraint that some region must be contained in another one, or merging with the constraint that two regions must be disjoint.

Below are some examples of structure graphs for Set Algebra (left) and Euler Diagrams (right) that illustrate possible uses of multiple tokens of the same type vs multiple uses of the same token. In the left-most graph there are two distinct tokens of type `A` as inputs to a constructor vertex. In the second graph there are two tokens of type `B`. In the third graph one token is used as input for two different constructor vertices. In the fourth graph one token is used twice as input, and also happens to be the output; making a cycle.







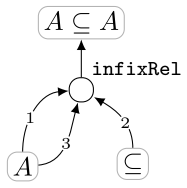


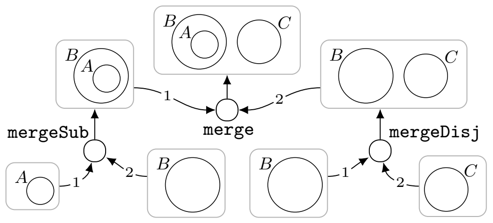
5. The regions that are neither in the interior or exterior are the ones which are cut through by the new curve.

By definition, structure graphs may be infinite and have arbitrary complexity. In particular, Definition 3.4, of a construction space, requires a potentially infinite structure graph called its *realm*, that essentially represents the universe of all representations encompassed by the construction space.

Definition 3.4. A **construction space** is a triple, $\mathcal{C} = (T, C, G)$, where T is a type system, C is a constructor specification over T, and G is a structure graph for C (Raggi et al., 2023). We call G the **realm** of $\mathcal{C}$. A structure graph, g, is called a **construction graph for** $\mathcal{C}$ if it is a subgraph of G. In such case we say that g **belongs** to $\mathcal{C}$.

The realm of $\mathcal{C}$ determines whether any specific structure graph (such as the ones displayed above) actually *belong* to $\mathcal{C}$. Some graphs which satisfy the rules determined by the constructor specification (e.g., the input and output types are correct) may not live in the realm. For example, given reasonable type assignments, the following graphs *are* structure graphs for the *constructor specifications* of Set Algebra and Euler Diagrams, but they *do not* live in their respective realms: one (left) does not distinguish between two distinct token occurrences, and the other (right) distinguishes a pair of tokens which actually only appears once in the token which is being constructed[6].





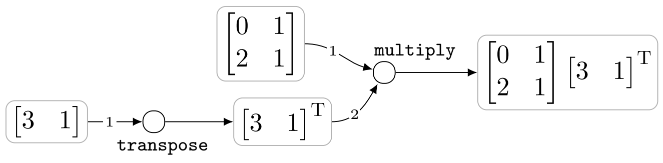
It is easy to see that we can specify construction spaces for Set Algebra and Euler Diagrams, and we have demonstrated how the type systems and constructor specifications can be defined. The examples of construction spaces that we have shown for these systems are what we earlier called *grammatical spaces*. In this paper, we do not need to preoccupy ourselves with the three spaces that form a representational system: the methods and results presented here concern construction spaces, with no regard to whether they are the grammatical, entailment, or identification spaces. In order to demonstrate what can be modelled with construction spaces, we give an example below and further examples – on geometric constructions and proofs – can be found in Appendix A.

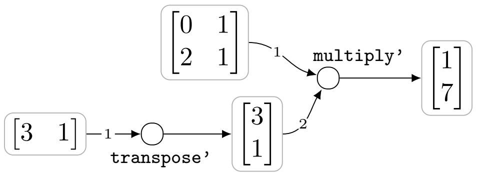
Example 3.3 (Matrix Algebra)**.** The tokens of one construction space are matrix expressions where the constructors `transpose` and `multiply` take one and two inputs, respectively, and output the resulting expression[7]; see the structure graph on the left. We

6. Such distinctions are modelling choices; the construction $A \subseteq A$ may belong to a construction space for Set Algebra if we do not care to distinguish between different instances of the same type. If there is only one token of type A, can we model the property *is written to the left of*? No!

7. See here that the token T is not an input to the `transpose` vertex. In relation to Raggi et al. (2023), the illustrated construction space would *not* typically be considered as being in the realm of a grammatical space for a 'matrix algebra' representational system: any such grammatical space would, intuitively, indicate that the token $[3\ \ 1]^{\mathrm{T}}$ was built from $[3\ \ 1]$ and T. Thus, this example illustrates the generality of the theory presented in this paper: the structure graph of any construction space is a graph encoding of any (and all) relations of interest, between the tokens, as captured by the constructors.

can also define another construction space where, instead of 'putting tokens together' to form composite matrix expressions, the constructors represent calculations in question. For example, below right is a graph[8], isomorphic to that on the left, but the 'output' tokens are the result of evaluating the 'input' expressions.





Recall that our major aim is to transform one token into another, such that some *specified relation* holds between them. We need a way to capture *relations across* spaces – not only within a space – defined using *properties* of tokens. A *multi-space system*augments a tuple of construction spaces, $\mathcal{C}_1, \ldots, \mathcal{C}_n$, with another construction space, $\mathcal{G}$, which captures properties of, and relations between, tokens. Here we use the concept of a *meta-space*, which generalises the concept of an identification space from Raggi et al. (2023).

We now define a *multi-space system* which will express relations both *across the tokens* of n construction spaces and *within* each of the individual construction spaces using a *meta-space*, $\mathcal{G}$.

Definition 3.5. A tuple of construction spaces, $\mathcal{M} = (\mathcal{C}_1, \ldots, \mathcal{C}_n, \mathcal{G})$, is a **a multi-space system** provided

1. for each $\mathcal{C}_i$, where $i \leq n$, the type system, $T_{\mathcal{G}} = (Ty_{\mathcal{G}}, \leq_{\mathcal{G}})$, of $\mathcal{G}$ extends the type system, $T_{\mathcal{C}_i} = (Ty_{\mathcal{C}_i}, \leq_{\mathcal{C}_i})$, of $\mathcal{C}_i$, that is: $\leq_{\mathcal{C}_i} \subseteq \leq_{\mathcal{G}}$,
2. $\mathcal{C}_1 \cup \cdots \cup \mathcal{C}_n \cup \mathcal{G}$ is a construction space.

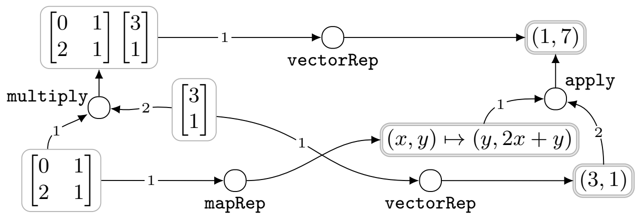
The spaces $\mathcal{G}$ and, resp., $\mathcal{S} = \mathcal{C}_1 \cup \cdots \cup \mathcal{C}_n \cup \mathcal{G}$ are called the **meta-space** and, resp., the **superspace** of $\mathcal{M}$. A tuple of structure graphs, $(g_1, \ldots, g_n, g)$, where each g_i belongs to $\mathcal{C}_i$ and g belongs to $\mathcal{G}$, is called a **construction graph**[9] that **belongs** to $\mathcal{M}$.

The next example shows that meta-spaces can be used for the purpose of encoding the semantics of a construction space. Appendix A includes a further example, focusing on low-level properties of tokens drawn from set algebra.

Example 3.4 (Semantics for Matrix Algebra)**.** Take a construction space, $\mathcal{C}$, for matrix algebra as in example 3.3, and meta-space, $\mathcal{G}$, for $\mathcal{C}$ that contains, for example, a token for each linear function. Then $\mathcal{G}$ represents some relations using constructors `mapRep` and `vectorRep`, take matrix expressions as input and output corresponding vectors. Thus, we can capture the typical linear-algebra semantics of matrices. If the realms of $\mathcal{C}$ and $\mathcal{G}$ model matrix algebra and its linear algebra semantics correctly, we expect to see that matrix multiplication and linear map application behave similarly. That is, we expect that *whenever* the inputs of `multiply` and `apply` are related by `mapRep` and `vectorRep` then the

8. This structure graph *could* be considered as part of the realm of a matrix algebra entailment space.
9. Strictly, of course, $(g_1, \ldots, g_n, g)$ is not a graph. We call it a graph for simplicity, noting that the union of the graphs, namely $g_1 \cup \cdots \cup g_n \cup g$, is a construction graph for the superspace $\mathcal{S}$.

outputs *must* be related by the `vectorRep` relation. Thus, we will find structure graphs, such as the following, in the realm of $\mathcal{C} \cup \mathcal{G}$.

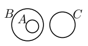


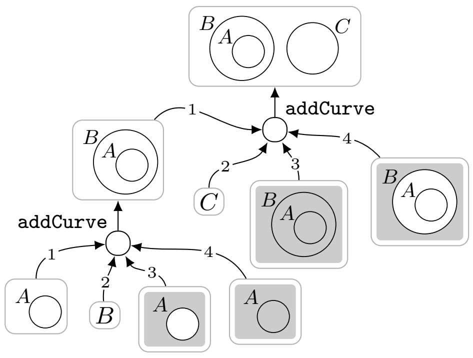
$\begin{bmatrix}0 & 1\\2 & 1\end{bmatrix}\begin{bmatrix}3\\1\end{bmatrix}$
vectorRep
$(1,7)$
multiply
apply
$\begin{bmatrix}3\\1\end{bmatrix}$
$(x,y) \mapsto (y, 2x+y)$
$\begin{bmatrix}0 & 1\\2 & 1\end{bmatrix}$
mapRep
vectorRep
$(3,1)$
1
2

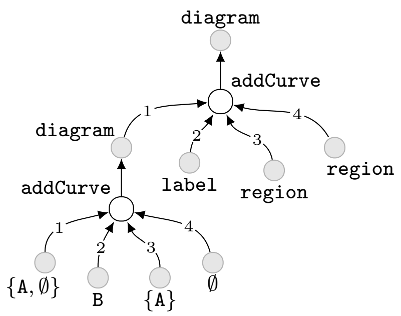
Now, every construction space, $\mathcal{C}$, is a special case of a multi-space system: we have equivalence with the multi-space system $\mathcal{M} = (\mathcal{G})$, where $\mathcal{G} = \mathcal{C}$. Thus, we will allow ourselves to omit brackets, writing 'let $\mathcal{C}$ be a multi-space system' to mean 'let $\mathcal{M} = (\mathcal{C})$ be a multi-space system'. Similarly, we will write 'Let $\mathcal{C}$ be a construction space' then use it as if it were a multi-space system. In general, multi-space systems are crucial to our approach to structure transfer, where we seek to identify a token (or tokens) in some construction spaces that are in defined relationships with tokens in other construction spaces.

3.2 Pattern Graphs

Given a construction space, $\mathcal{C} = (T, C, G)$, a *pattern graph* for it is a structure graph that satisfies the constraints laid down by the type system, T, and constructor specification, C, but which need not be part of the realm, G. In other words, the vertices of a pattern graph for $\mathcal{C} = (T, C, G)$ are labelled by either types of T or by constructors of C, and for any vertex labelled by a constructor, c, its inputs and outputs must be in agreement with the signature, $sig(c)$. A *pattern space*, $\mathcal{P}$, for $\mathcal{C}$, is a space where pattern graphs for $\mathcal{C}$ are taken from, with the condition that the realm, G, of $\mathcal{C}$ instantiates the realm, Γ, of $\mathcal{P}$.

Example 3.5. Given the construction space of Euler Diagrams, the construction graph on the left is an *instantiation* of the *pattern graph* on the right. The construction graph on the left lives in the realm of Euler Diagrams and it *instantiates* the pattern graph on the right. Many other construction graphs in the realm of Euler Diagrams also instantiate this given pattern graph. For instance, the vertex with type `label` could be instantiated with, say, D, and the vertices with type `region` could be instantiated with other regions. Some instantiations would result in a construction of 'equivalent' Euler diagrams (to ) but some may be genuinely different. Hence, this pattern graph characterises many construction graphs in the realm of Euler Diagrams.





As shown in the previous example, our convention moving forward will be to draw tokens within vertices, while types will be written outside – labelling the vertices. Thus, when we draw pattern graphs, where the specific tokens are irrelevant, we will draw empty vertices with their types.

Definition 3.6. Let $\mathcal{C} = (T, C, G)$ and $\mathcal{P} = (T, C, \Gamma)$ be construction spaces and let g and κ be construction graphs for $\mathcal{C}$ and $\mathcal{P}$, respectively. We say that g is a **specialisation** of κ provided there exists an isomorphism, $s\colon \kappa \to g$, that preserves constructor labels, and for each token, t, of κ, the type of $s(t)$ in $\mathcal{C}$ is a subtype of the type of t in $\mathcal{P}$ (adapted from (Raggi et al., 2023)).

If g is a specialisation of κ then κ is a **generalisation** of g, and the functions s and s^{-1} are called **specialisation** and **generalisation functions**, respectively. We also say that g is an **instantiation of $\boldsymbol{\kappa}$ in $\boldsymbol{\mathcal{C}}$** and, thus, that κ is **instantiatable** in $\mathcal{C}$.

Definition 3.7. A **pattern space** for construction space $\mathcal{C} = (T, C, G)$ is a construction space, $\mathcal{P} = (T, C, \Gamma)$, where G instantiates some subgraph of Γ (adapted from (Raggi et al., 2023)). A **pattern graph** for $\mathcal{C}$ is any structure graph that belongs to the realm of some pattern space for $\mathcal{C}$.

Importantly, we can easily determine whether a graph, κ, is a pattern graph for $\mathcal{C}$: if κ uses only constructors and types in $\mathcal{C}$ and ensures that configurations of the constructors agree with the constructor's signature as defined in $\mathcal{C}$, then κ is a pattern graph for $\mathcal{C}$. We note that Definition 3.6, whilst using our convention of denoting construction space $\mathcal{C}$'s pattern space by $\mathcal{P}$, is not restricted to a construction space, $\mathcal{C}$, and its pattern space, $\mathcal{P}$. Indeed, we often wish to talk of specialisations, $s\colon \kappa \to \kappa'$, between pattern graphs in a pattern space $\mathcal{P}$. This motivates our use of dual terminology: when working across spaces, we typically talk of instantiations whereas when working within a space we tend to talk of specialisations. The graphs in a pattern space, $\mathcal{P}$, for $\mathcal{C}$ must satisfy the rules given by the type system and constructor specification for $\mathcal{C}$, yet they need not live in the realm of $\mathcal{C}$. We require that the realm, G, of $\mathcal{C}$ is an *instantiation* of some part of the realm, Γ, of $\mathcal{P}$. Thus, a pattern space, $\mathcal{P}$, provides a supply of pattern graphs which can be used to describe construction graphs that belong to $\mathcal{C}$. Given a construction space, $\mathcal{C}$, we will denote its construction graphs using the Latin alphabet, while its pattern graphs will be denoted using the Greek alphabet. We depict vertices in pattern spaces with a shaded background, and their labels will represent their assigned type, for example, a token, t, with type τ will be drawn as: $\textcircled{t}\tau$, or simply $\bigcirc\tau$ when there is no reason to name the token. We now generalise the concept of a pattern space to a *pattern system* for a multi-space system.

Definition 3.8. Let $\mathcal{M} = (\mathcal{C}_1, \ldots, \mathcal{C}_n, \mathcal{G})$ be a multi-space system with superspace $\mathcal{S}$. A **pattern system** for $\mathcal{M}$ is a multi-space system, $(\mathcal{P}_1, \ldots, \mathcal{P}_n, \mathcal{P})$, such that

1. each $\mathcal{P}_i$ is a pattern space for $\mathcal{C}_i$,
2. $\mathcal{P}$ is a pattern space for $\mathcal{G}$, and
3. $\mathcal{P}_1 \cup \cdots \cup \mathcal{P}_n \cup \mathcal{P}$ is a pattern space for $\mathcal{S}$.

A tuple of pattern graphs, $(\kappa_1, \ldots, \kappa_n, \kappa)$, where each κ_i is a pattern graph for $\mathcal{C}_i$ and κ is a pattern graph for $\mathcal{G}$ is a **pattern graph** for $\mathcal{M}$.

In what follows, whenever we exploit pattern graphs for any $\mathcal{C}_i$ or $\mathcal{G}$, we assume that they are drawn from a fixed pattern system for $\mathcal{M}$; we do not make the pattern system explicit. This simply ensures the compatibility of any pair of pattern graph and does not lose generality. In particular, if we have a pattern graph, κ_i, for each $\mathcal{C}_i$ and a pattern graph, κ, for $\mathcal{G}$ then it is taken that $\kappa_1 \cup \cdots \cup \kappa_n \cup \kappa$ is a pattern graph for $\mathcal{S}$.

3.3 Special Kinds of Morphisms

A very important kind of specialisation, $s\colon \kappa \to \kappa'$, is where κ' can be instantiated in $\mathcal{C}$. We will often refer to these as *instantiatable* specialisations.

Definition 3.9. Let $\mathcal{C}$ be a construction system with pattern graphs κ and κ'. We say that κ' is an **instantiatable specialisation** of κ if there exists a specialisation function, $s\colon \kappa \to \kappa'$ and κ' can be instantiated. We call s an **instantiatable specialisation function**.

Moving forward, we simplify statements such as 'let κ' be a specialisation of κ with specialisation function $s\colon \kappa \to \kappa'$' to simply 'let $s\colon \kappa \to \kappa'$ be a specialisation function', introducing the function and codomain together. When we say *instantiatable* specialisation function, $s\colon \kappa \to \kappa'$, we mean κ' can be instantiated in a specified construction space, $\mathcal{C}$.

Now we define a *reification* of a pattern graph, κ: to reify κ means to enlarge it, by adding vertices and arrows, and to specialise its types. Enlarging κ adds *structural context*, and replacing types with subtypes makes it *more concrete* because subtypes are 'closer' to tokens in $\mathcal{C}$.

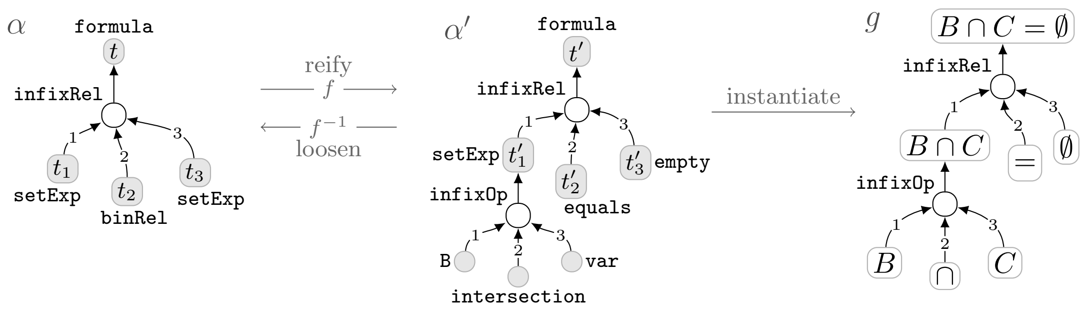
Definition 3.10. Let κ and κ' be graphs for construction space $\mathcal{P}$. Then κ' is a **reification** of κ provided there exists a monomorphism, $f\colon \kappa \to \kappa'$, called a **reification function**, where $f[\kappa]$ specialises κ.

Reifications will be used abundantly in this paper. As well as reifications, which specialise types and enlarge a pattern graph, we also require the notion of a *loosening*. A loosening allows us to shrink a pattern graph whilst generalising its types (i.e., replacing types with supertypes). Essentially, a loosening *removes structural context* and makes it *less concrete* because supertypes are 'further' from tokens in $\mathcal{C}$.

Definition 3.11. Let κ and κ' be graphs for construction space $\mathcal{P}$. Then κ is a **loosening** of κ' provided there exists a partial surjective function, $\ell\colon \kappa' \to \kappa$, called a **loosening map**, whose inverse is a reification function. Given a set, W, of tokens, in κ', we say that ℓ is a loosening **up to W** provided ℓ generalises the types of all tokens in its pre-image that are not in W.

Similar language conventions are adopted for *instantiatable* reifications and loosenings, which require the instantiatability of the codomain.

Example 3.6. In the patterns, α and α' depicted below, for Set Algebra, the function, $f : \alpha \to \alpha'$, where $f(t) = t'$, $f(t_1) = t'_1$, $f(t_2) = t'_2$ and $f(t_3) = t'_3$, is a reification, and it is also an instantiatable reification, as there exists an instantiation, g, of α'. The partial function $f^{-1} : \alpha' \to \alpha$ is a loosening.



The next two lemmas follow trivially from the definitions above.

Lemma 3.1. *For any instantiatable reification function, $f\colon \kappa \to \kappa'$, κ is instantiatable.*

Lemma 3.2. *For any loosening map, $\ell\colon \kappa \to \kappa'$, if κ is instantiatable then so is κ'.*

We now generalise the concepts of specialisation and reification to multi-space systems.

Definition 3.12. Let $\mathcal{M} = (\mathcal{C}_1, \ldots \mathcal{C}_n, \mathcal{G})$ be a multi-space system. Let $(\kappa_1, \ldots, \kappa_n, \kappa)$ and $(\kappa_1', \ldots, \kappa_n', \kappa')$ be pattern graphs for $\mathcal{M}$. A homomorphism, $f\colon (\kappa_1, \ldots, \kappa_n, \kappa) \to (\kappa_1', \ldots, \kappa_n', \kappa')$, is a **specialisation function** (resp., a **reification function**) in $\mathcal{M}$ provided $f|_\kappa\colon \kappa \to \kappa'$ and all $f|_{\kappa_i}\colon \kappa_i \to \kappa_i'$ for $i \leq n$, are specialisations (resp. reifications).
We say that $(\kappa_1', \ldots, \kappa_n', \kappa')$ **specialises** (resp. **reifies**) $(\kappa_1, \ldots, \kappa_n, \kappa)$ in $\mathcal{M}$.

To conclude this section, we extend the notion of instantiation to multi-space systems.

Definition 3.13. Let $\mathcal{M} = (\mathcal{C}_1, \ldots \mathcal{C}_n, \mathcal{G})$ be a multi-space system with pattern graph $(\kappa_1, \ldots, \kappa_n, \kappa)$. Then $(\kappa_1, \ldots, \kappa_n, \kappa)$ is **instantiatable** in $\mathcal{M}$ provided there exists a construction graph, $(g_1, \ldots, g_n, g)$, that belongs to $\mathcal{M}$ and specialises $(\kappa_1, \ldots, \kappa_n, \kappa)$. An **instantiatable specialisation function** is a specialisation function, $s\colon (\kappa_1, \ldots, \kappa_n, \kappa) \to (\kappa_1', \ldots, \kappa_n', \kappa')$, such that $(\kappa_1', \ldots, \kappa_n', \kappa')$ can be instantiated in $\mathcal{M}$. We say that $(\kappa_1', \ldots, \kappa_n', \kappa')$ is an instantiatable specialisation of $(\kappa_1, \ldots, \kappa_n, \kappa)$ in $\mathcal{M}$.

4. Sequents and Schemas for Multi-Space Systems

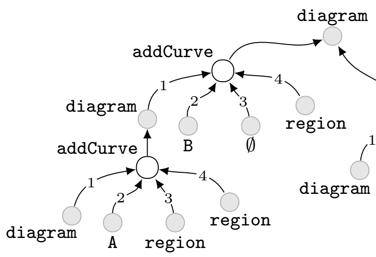
Here we will introduce some notions that amount to a meta-logic for deriving knowledge about construction spaces. As we will see, this meta-logic can be used for solving various problems. These problems can be grouped in the following three classes:

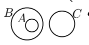
1. can we *instantiate* a pattern graph in a construction space? (Section 4.1),
2. can we *infer* that a particular relation holds for instantiations of a pattern graph in a multi-space system? (Section 4.2), and
3. can we *transform* a construction graph in a multi-space system, $\mathcal{M}$, so that it is ensured that some specified relation holds between the tokens of the resulting graph and the rest of tokens in $\mathcal{M}$? (Section 4.3).

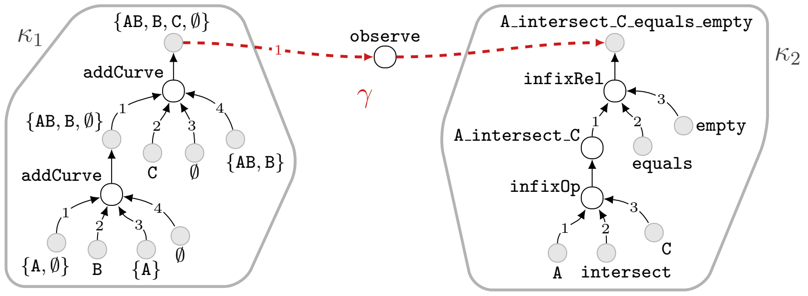
As we will see in Sections 4.1 and 4.2, instantiation and inference are two uses of the same process. However, in Section 4.3 we will see that transformations exploit the same process but with a twist: a target pattern graph is reified, resulting in a new structure which, if instantiated, gives rise to a desired re-representation. This is the idea behind the notion of *structure transfer*.

First we will look at one example of each of the categories above: instantiation, inference, and transformation.

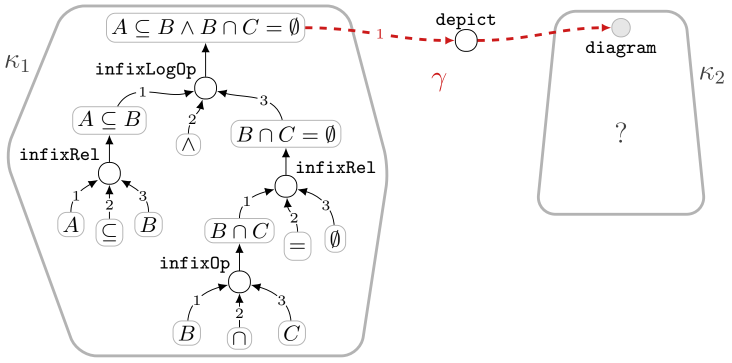
Instantiation. Can the pattern graph γ, drawn below, be instantiated in the construction space for EULER DIAGRAMS?



Inference. Can $A \cap C = \emptyset$ be observed from an Euler diagram with type $\{\mathtt{AB}, \mathtt{B}, \mathtt{C}, \emptyset\}$, given the patterns κ_1 (for EULER DIAGRAMS) and κ_2 (for SET ALGEBRA)? In other words, can $A \cap C = \emptyset$ can be observed from the diagram  ? Note that the *goal* of determining observability is captured by a pattern, γ (drawn with red dashed arrows), for the meta-space of a multi-space containing EULER DIAGRAMS and SET ALGEBRA.



Transformation. Can we find a pattern graph, κ_2, which ensures that any instantiation of it contains a token that *depicts* $A \subseteq B \wedge B \cap C = \emptyset$? Similarly to the inference problem, we want to determine whether some pattern graph, γ (drawn with red dashed arrows), can be instantiated for κ_1 and κ_2, but in this case only κ_1 is given, and we wish to find some κ_2 that ensures the instantiatability of γ.



The problem of transformation, above, is generally motivated by the wish to produce a transformation of one token, t_1, into another token, t_2, using relations defined in an multi-space system. Later in Section 4.3 we will see how we might obtain enough information about such a diagram, t_2, to actually build it. The process we suggest produces a new pattern graph, κ_2 by reifying it iteratively in such a way that it is ensured that any instantiation of κ_2 results in a t_2 that is in the desired relationship with t_1. The way that κ_2 is generated is by using *schemas*, which can be understood as units of knowledge that encode invariants within and across different spaces. Informally, some schemas can be understood as capturing analogies across spaces. For example, as we will see, the analogy between the use of the symbol $\subseteq$, in Set Algebra and the actual *containment* of a region inside a curve in Euler Diagrams can be readily captured by a schema. The formalisation of this process, called *structure transfer*, by which schemas are used to produce a target graph from a given graph and some constraints, is not restricted to pairs of tokens, but to any n-tuples of structure graphs in a multi-space. Nevertheless, our motivating examples are for $n = 2$.

We start with the definition of a *sequent*, drawing analogy from sequent calculus: given a collection of pattern graphs representing *assumptions* and another representing a *goal*, drawn from a multi-space system, we can form a *sequent*. Then we define the concept of a *schema*, which is a rule for discerning whether a goal can be instantiated, given an instantiation of the assumptions. The operation of *applying* schemas to sequents will capture how a sequent needs to be manipulated in order to derive whether, given an instantiation of the assumptions, there exists a suitable instantiation of the goal. In addition, schemas are foundational for defining *instantiation*, *inference* and *transfer* schemas.

Definition 4.1. A **sequent** for multi-space system $\mathcal{M}$ is a pattern graph, $(\kappa_1, \ldots, \kappa_n, \alpha)$, for $\mathcal{M}$ and a pattern graph, γ, for the meta-space, $\mathcal{G}$, denoted $\langle \kappa_1, \ldots, \kappa_n, \alpha \Vdash \gamma \rangle$. We call $(\kappa_1, \ldots, \kappa_n)$ the **context**, α the **antecedent**, and γ the **consequent** of the sequent. If the context and antecedent are empty graphs we will write $\langle \ \Vdash \gamma \rangle$. Similarly, if the consequent is an empty graph we will write $\langle \kappa_1, \ldots, \kappa_n, \alpha \Vdash \ \rangle$.

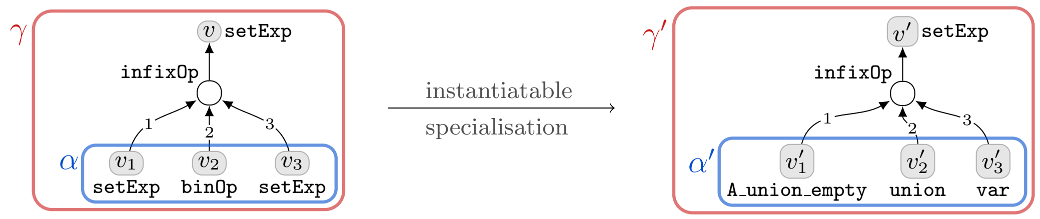
Sequents can be used to capture problems. In the context where $\langle \kappa_1, \ldots, \kappa_n, \alpha \Vdash \gamma \rangle$ represents a problem we will call $(\kappa_1, \ldots, \kappa_n, \alpha)$ the **assumptions** and γ the **goals** of the problem. The instantiation problem above is captured by a sequent, $\langle \ \Vdash \gamma \rangle$, because the goal is to instantiate γ. The inference and transformation problems are captured by a sequent $\langle \kappa_1, \kappa_2, \alpha \Vdash \gamma \rangle$ where α is empty. To *solve* instantiation and inference problems means, roughly, to determine that the instantiatability of the assumptions implies the instantiatability of the goal. As we will define next, any sequent that satisfies this property is called a *schema*. Thus, solving the instantiation or inference problem $\langle \kappa_1, \kappa_2, \alpha \Vdash \gamma \rangle$ means proving that it is a schema. The transformation case is different: solving a transformation problem means finding the graph κ_2 for which $\langle \kappa_1, \kappa_2, \alpha \Vdash \gamma \rangle$ is a schema.

Throughout this paper, we will draw an analogy between the theory of schemas and model theory (more specifically, satisfiability of formulas in the sequent calculus). The property of *being a schema for a construction space*, for sequents, is analogous to the property of *being satisfiable via an interpretation*, for formulas. This is reflected by the language and notation, where we use words like *antecedent* and *consequent*, and the symbol $\Vdash$ in analogy with the turnstile, $\vdash$, of the sequent calculus. The foundation of the analogy is this: to be instantiatable in a construction space is analogous to being satisfiable via some interpretation.

Definition 4.2. Let $\langle \kappa_1, \ldots, \kappa_n, \alpha \Vdash \gamma \rangle$ be a sequent for multi-space system $\mathcal{M}$. Then $\langle \kappa_1, \ldots, \kappa_n, \alpha \Vdash \gamma \rangle$ is a **schema** for $\mathcal{M}$ provided for any instantiatable specialisation function, $s\colon (\kappa_1, \ldots, \kappa_n, \alpha) \to (\kappa'_1, \ldots, \kappa'_n, \alpha')$, there exists an instantiatable specialisation, $s'\colon (\kappa_1, \ldots, \kappa_n, \alpha \cup \gamma) \to (\kappa'_1, \ldots, \kappa'_n, \alpha' \cup \gamma')$ that extends s.

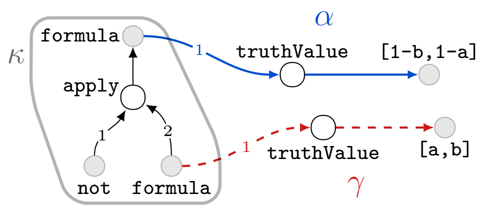
Next we will show three examples of schemas in increasing levels of complexity – first for a case where $\mathcal{M} = \mathcal{C}$, then for a case where $\mathcal{M} = (\mathcal{C}, \mathcal{G})$, and lastly for a case where $\mathcal{M} = (\mathcal{C}, \mathcal{D}, \mathcal{G})$.

Example 4.1. Let $\mathcal{C}$ be the construction space for Set Algebra. The sequent $\langle \alpha \Vdash \gamma \rangle$, visualised below left, is a schema for $\mathcal{C}$. The pattern graph α contains exactly the inputs for an `infixOp` constructor. The pattern graph, α' is an example of an instantiatable specialisation of α. The fact that $\langle \alpha \Vdash \gamma \rangle$ is a schema guarantees that some γ' must exist for which $\alpha' \cup \gamma'$ is an instantiatable specialsation of $\alpha \cup \gamma$.



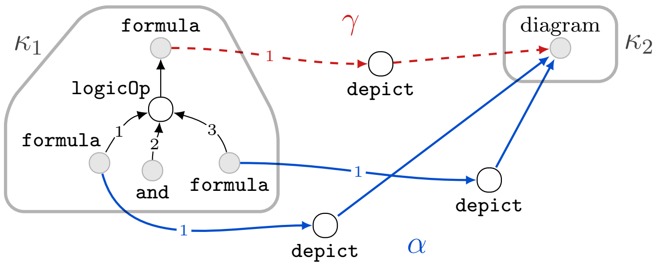
In other words, $\langle \alpha \Vdash \gamma \rangle$ is a schema because whichever way we restrict the types of v_1, v_2, and v_3, (e.g., to `A_union_empty`, `union`, `var`), if we can instantiate α' then we can instantiate γ' respecting the types chosen for the specialisation, α', of α (e.g, v' can be instantiated to a token of the form $A \cup \emptyset \cup B$).

Example 4.2. Consider a multi-space system $(\mathcal{C}, \mathcal{G})$ where $\mathcal{C}$ encodes some basic logic and $\mathcal{G}$ encodes some relations *fuzzily* – every number between 0 and 1 is a truth value token in $\mathcal{G}$ and, for all $0 \leq \mathtt{a} \leq \mathtt{b} \leq 1$, the interval `[a,b]` is a type, where the subtype order, $\preccurlyeq$, satisfies the following: `[a,b]` $\preccurlyeq$ `[c,d]` iff $\mathtt{c} \leq \mathtt{a}$ and $\mathtt{b} \leq \mathtt{d}$. For any number $0 \leq x \leq 1$, we have that $\mathit{type}(x) =$ `[x,x]`. Thus, certainty of truth is represented by token 1 of type `[1,1]` and certainty of falsity is represented by token 0 of type `[0,0]`. For all $0 \leq \mathtt{a} \leq \mathtt{b} \leq 1$, the sequent $\langle \kappa, \alpha \Vdash \gamma \rangle$, illustrated below, is a schema[10]. It captures a generalisation of the fact that the truth value of a formula is one minus the truth value of its negation.



10. From here on, our drawing convention for any sequent, $\langle \kappa_1, \ldots, \kappa_n, \alpha \Vdash \gamma \rangle$, is that the context graphs, $\kappa_1, \ldots, \kappa_n$, will drawn with black arrows, the antecedent, α, will be drawn with thick blue arrows, and the consequent, γ, will be drawn with dashed red arrows.

Example 4.3. Let $\mathcal{M} = (\mathcal{C}, \mathcal{D}, \mathcal{G})$ be a multi-space where $\mathcal{C}$ encodes Set Algebra, $\mathcal{D}$ encodes Euler Diagrams and $\mathcal{G}$ encodes relations across and within the spaces. The sequent $\langle \kappa_1, \kappa_2, \alpha \Vdash \gamma \rangle$ for $\mathcal{M}$, depicted below, is as schema. It captures the intuition that a diagram depicts a conjunction if it depicts both conjuncts.

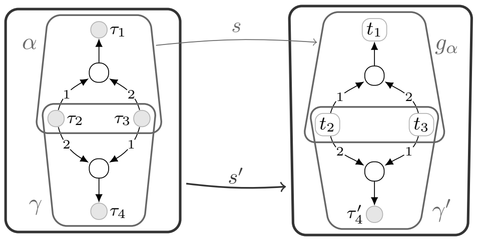


In terms of our three overarching problems (on instantiation, inference and transformation), we need to answer the following question: if we can instantiate $(\kappa_1, \ldots, \kappa_n, \alpha)$, can we also instantiate $(\kappa_1, \ldots, \kappa_n, \alpha \cup \gamma)$? Schemas are our route to an answer, so it is useful to have some results that establish when sequents are also schemas. Lemma 4.1 shows that any sequent where $\gamma \subseteq \alpha$ is a schema. Secondly, lemma 4.2 asserts that we can shrink the consequent of a known schema to a subraph, $\gamma' \subseteq \gamma$, and also add some subgraph of the antecedent, $\alpha' \subseteq \alpha$, to the goal, and the resulting sequent is also schema. Since both lemmas hold trivially, we omit their proofs.

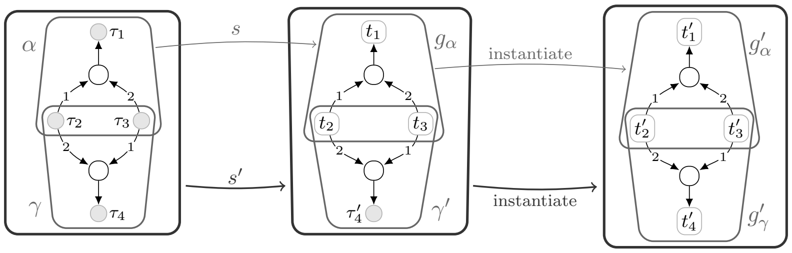
Lemma 4.1. *Any sequent $\langle \kappa_1, \ldots, \kappa_n, \alpha \Vdash \gamma \rangle$ for multi-space system $\mathcal{M}$ where $\gamma \subseteq \alpha$ is a schema for $\mathcal{M}$.*

Lemma 4.2. *Let $\langle \kappa_1, \ldots, \kappa_n, \alpha \Vdash \gamma \rangle$ be a schema for multi-space system $\mathcal{M}$. Let α' and γ' be pattern graphs for $\mathcal{M}$'s meta-space such that $\alpha' \subseteq \alpha$ and $\gamma' \subseteq \gamma$. Then $\langle \kappa_1, \ldots, \kappa_n, \alpha \Vdash \gamma' \cup \alpha' \rangle$ is a schema for $\mathcal{M}$.*

A crucial property of schemas is that they allow us to infer that an instantiation of $(\kappa_1, \ldots, \kappa_n, \alpha \cup \gamma)$ exists given the instantiatability of $(\kappa_1, \ldots, \kappa_n, \alpha)$. That is, if an instantiation, $(g_1, \ldots, g_n, g_\alpha)$, of $(\kappa_1, \ldots, \kappa_n, \alpha)$ exists, then an instantiation $(g'_1, \ldots, g'_n, g'_\alpha \cup g'_\gamma)$ of $(\kappa_1, \ldots, \kappa_n, \alpha \cup \gamma)$ must exist. It is important to note that such $(g'_1, \ldots, g'_n, g'_\alpha)$ need not have the same tokens as $(g_1, \ldots, g_n, g_\alpha)$, but the types, however minimal, must be the same. This is because any instantiation is itself an instantiatable specialisation, so if we take $s\colon (\kappa_1, \ldots, \kappa_n, \alpha) \to (g_1, \ldots, g_n, g_\alpha)$ as our instantiatable specialisation function, we can extend this to some instantiatable specialisation function, $s'\colon (\kappa_1, \ldots, \kappa_n, \alpha) \to (g_1, \ldots, g_n, g_\alpha \cup \gamma')$, noting that $(g_1, \ldots, g_n, g_\alpha \cup \gamma')$ may not be an instantiation itself, but that it is instantiatable. In the illustration below, where $(g_1, \ldots, g_n)$ is empty, notice that while s instantiates α, s' specialises the type τ_4 to some type τ'_4:



In turn, this guarantees that there exists an instantiation, g'_α, of g_α whose types are subtypes of the corresponding ones in g_α, that can be extended to an instantiation, $g'_\alpha \cup g'_\gamma$, of $g_\alpha \cup \gamma'$ and, therefore, of $\alpha \cup \gamma$:



Moving back to our general context, we are seeking to establish whether a sequent, $\langle \kappa_1, \ldots, \kappa_n, \alpha \Vdash \gamma \rangle$, is a schema. Our approach is to see whether we can grow α and reduce γ in such a way that α envelops γ, enabling us to invoke Lemma 4.1. Such manipulations, which grow the antecedent or reduce the consequent, are covered by what we call *applications* of a known schema to a sequent (Definition 4.2). If we can apply schemas to a sequent until we reach a schema, then we say that the sequent is *valid* (Definition 4.7).

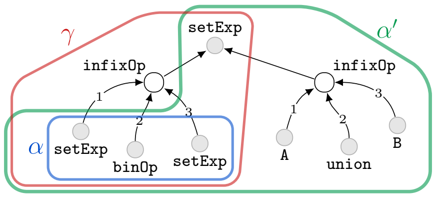
In order for the application of schemas to be as intended we need the notion of *monotonicity*. The property of monotonicity is analogous to the property of monotonicity in logic where, given a valid inference, additional facts cannot invalidate it (i.e., the property that if $A \vdash B$ then $A, C \vdash B$ for any C). Monotonicity allows us to use schemas to *grow* instantiations. That is, we will know that it is possible to *grow* an instantiation of $\kappa_1 \cup \cdots \cup \kappa_n \cup \alpha$ to an instantiation of $\kappa_1 \cup \cdots \cup \kappa_n \cup \alpha \cup \gamma$.

Definition 4.3. Let $\langle \kappa_1, \ldots, \kappa_n, \alpha \Vdash \gamma \rangle$ be a schema and $(\kappa'_1, \ldots, \kappa'_n, \alpha')$ be a pattern graph for multi-space system $\mathcal{M}$. Then $\langle \kappa_1, \ldots, \kappa_n, \alpha \Vdash \gamma \rangle$ is **monotonic** for $(\kappa'_1, \ldots, \kappa'_n, \alpha')$ in $\mathcal{M}$ provided $\langle \kappa_1 \cup \kappa'_1, \ldots, \kappa_n \cup \kappa'_n, \alpha \cup \alpha' \Vdash \gamma \rangle$ is a schema for $\mathcal{M}$. Such a pattern graph, $(\kappa'_1, \ldots, \kappa'_n, \alpha')$, is called a **monotonic extender** for $\langle \kappa_1, \ldots, \kappa_n, \alpha \Vdash \gamma \rangle$.

We say that $\langle \kappa_1, \ldots, \kappa_n, \alpha \Vdash \gamma \rangle$ is **monotonic** for $\mathcal{M}$ provided it is monotonic for every pattern graph, $(\kappa'_1, \ldots, \kappa'_n, \alpha')$, for $\mathcal{M}$.

Many schemas are not monotonic for all pattern graphs but may still have non-trivial monotonic extenders.

Example 4.4. The schema $\langle \alpha \Vdash \gamma \rangle$ for Set Algebra from Example 4.1 is not monotonic for all pattern graphs. The pattern α', illustrated below, contains α and is instantiatable. Yet, if the vertex labelled with type `binOp` is specialised with, say, type `intersection`, then γ cannot be instantiated, as the set expression at the top would need to be instantiated to $A \cup B$ and also to some expression of the form $x \cap y$. Thus, α' is a not a monotonic extender for $\langle \alpha \Vdash \gamma \rangle$.



At the heart of our approach to defining applications of schemas is the notion of *weakening*. This operation takes a sequent, $\langle \kappa_1, \ldots, \kappa_n, \alpha \Vdash \gamma \rangle$, and allows us to specialise and augment the assumptions via a reification. In addition, a weakening allows us to generalise the goal and remove parts of it, via a loosening map. As we will see in theorem 4.1, a weakening applied to a schema yields also a schema if a monotonicity condition is satisfied given the enlargement of the assumptions and goal, captured by the definition of a *refinement*. This will be crucial for the notion of schema application, as it will allow us to modify a schema to adapt it to the context in which it will be used.

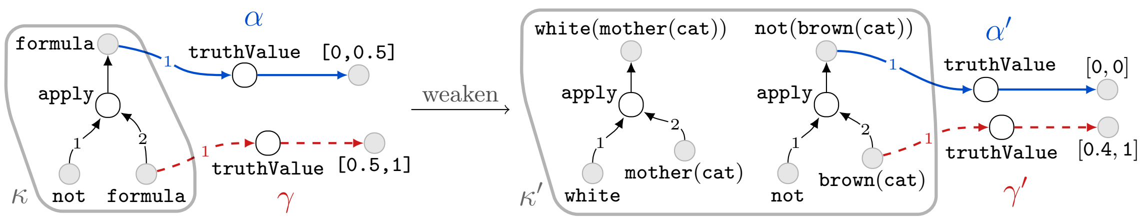
Definition 4.4. Let $\langle \kappa_1, \ldots, \kappa_n, \alpha \Vdash \gamma \rangle$ and $\langle \kappa'_1, \ldots, \kappa'_n, \alpha' \Vdash \gamma' \rangle$ be sequents for multi-space system $\mathcal{M}$. Then $\langle \kappa'_1, \ldots, \kappa'_n, \alpha' \Vdash \gamma' \rangle$ is a **weakening** of $\langle \kappa_1, \ldots, \kappa_n, \alpha \Vdash \gamma \rangle$ provided there exists a partial function, $r\colon (\kappa_1, \ldots, \kappa_n, \alpha \cup \gamma) \to (\kappa'_1, \ldots, \kappa'_n, \alpha' \cup \gamma')$, such that

1. $r|_{(\kappa_1, \ldots, \kappa_n, \alpha)}$ is a reification, and
2. $r|_\gamma$ is a loosening map, up to the tokens in $(\kappa_1 \cup \cdots \cup \kappa_n \cup \alpha) \cap \gamma$.

We call r a **weakening map**.

To understand weakenings better, let us build more on the analogy to the sequent calculus. Refining a schema is analogous to weakening it by specialising its antecedent and generalising its consequent. For example, in logic if we have a statement, $P(x) \vdash Q(x, 3)$, we can weaken it to, $P(2), R \vdash Q(2, z)$, by specialising x to 2, generalising z to 3 and adding another assumption, R. If we know that $P(x) \vdash Q(x, 3)$ is satisfiable then $P(2), R \vdash Q(2, z)$ must also be satisfiable. Now, given that schemas need not be generally monotonic, a weakening may make a schema invalid. However, Definition 4.5, of refinement, captures the conditions under which schemas can be weakened to ensure the result is also a schema.

Example 4.5. Consider schema $\langle \kappa, \alpha \Vdash \gamma \rangle$ from Example 4.2, for values $a = 0$ and $b = 0.5$, shown below left. This schema asserts that if the truth value of *not P* is a value between 0 and 0.5, then the truth value of `P` is a value between 0.5 and 1. The sequent, $\langle \kappa', \alpha' \Vdash \gamma' \rangle$, weakens it by augmenting and specialising the context and antecedent, and by generalising the consequent.



The weaker version, $\langle \kappa', \alpha' \Vdash \gamma' \rangle$ enlarges κ to κ', and asserts that if the truth value of `not(brown(cat))` is exactly 0 then the truth value of `brown(cat)` is *at least* 0.4. Of course it is! the value of `brown(cat)` is actually 1, but we have now weakened the schema.

As this example illustrates, weakening will be useful for contextualising a schema to a problem. Perhaps we need to show that the truth value in the interval $[0.4, 1]$, and we actually have a schema that guarantees a truth value in the interval $[0.5, 1]$. If we can weaken such a schema to fit the goals of the problem, then we will be able to apply it. Of course, this is only useful if weakening actually preserves the property of being a schema, which is not generally true for non-monotonic schemas. In order to show that it is true for some specific kinds of weakenings, we first prove, in the next lemma, that it is true for a restriction of a weakening to its 'image'.

Lemma 4.3. *Let $\langle \kappa_1, \ldots, \kappa_n, \alpha \Vdash \gamma \rangle$ be a sequent for multi-space system $\mathcal{M}$ and let $\langle \kappa'_1, \ldots, \kappa'_n, \alpha' \Vdash \gamma' \rangle$ be a weakening of $\langle \kappa_1, \ldots, \kappa_n, \alpha \Vdash \gamma \rangle$ with map r. If $\langle \kappa_1, \ldots, \kappa_n, \alpha \Vdash \gamma \rangle$ is a schema for $\mathcal{M}$, then so is $\langle r[\kappa_1], \ldots, r[\kappa_n], r[\alpha] \Vdash r[\gamma] \rangle$.*

Later, Theorem 4.1 will build up on Lemma 4.3 by using the notion of a *refinement*, which both restricts and builds . Theorem 4.1 is important since it allows us to modify schemas to obtain new schemas. The sufficient conditions are given in Definition 4.5. Proofs of both Lemma 4.3 and Theorem 4.1 can be found in Appendix B.

A *refinement* of a schema not only weakens it, but also can augment the consequent γ to γ' as long as γ' does not add anything beyond the assumptions. Importantly, the extra conditions ensure that the refinement of a schema is necessarily a schema.

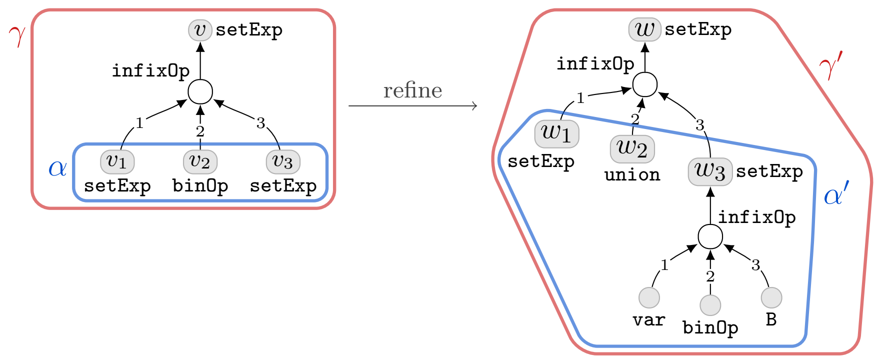
Definition 4.5. Let $\langle \kappa_1, \ldots, \kappa_n, \alpha \Vdash \gamma \rangle$ be a schema and $\langle \kappa'_1, \ldots, \kappa'_n, \alpha' \Vdash \gamma' \rangle$ a sequent for multi-space system $\mathcal{M}$. Then $\langle \kappa'_1, \ldots, \kappa'_n, \alpha' \Vdash \gamma' \rangle$ is a **refinement** of $\langle \kappa_1, \ldots, \kappa_n, \alpha \Vdash \gamma \rangle$ provided there exists a weakening map, $r\colon (\kappa_1, \ldots, \kappa_n, \alpha \cup \gamma) \to (\kappa'_1, \ldots, \kappa'_n, \alpha' \cup r[\gamma])$, such that

1. $\langle r[\kappa_1], \ldots, r[\kappa_n], r[\alpha] \Vdash r[\gamma] \rangle$ is monotonic for $(\kappa'_1, \ldots, \kappa'_n, \alpha')$ in $\mathcal{M}$, and
2. $\gamma' \subseteq \alpha' \cup r[\gamma]$.

We call r a **refinement map**.

The weakening of Example 4.2 is a refinement provided the multi-space system is built so that monotonicity holds. Moreover, in this example, γ' equals $r[\gamma]$, so the condition $\gamma' \subseteq \alpha' \cup r[\gamma]$ is met trivially. In the following example we see γ being augmented to some γ' that is larger than $r[\gamma]$.

Example 4.6. Consider the schema $\langle \alpha \Vdash \gamma \rangle$ from Example 4.1. The sequent $\langle \alpha' \Vdash \gamma' \rangle$, visualised below, is a refinement of $\langle \alpha \Vdash \gamma \rangle$, with refinement map $r\colon (\alpha, \gamma) \to (\alpha', \gamma')$, where $r(v) = w$, $r(v_1) = w_1$, $r(v_2) = w_2$, and $r(v_3) = w_3$.



Monotonicity is true by design of the space of Set Algebra: no instantiatiatable specialisation of α' *interferes* with our ability to also instantiatably specialise γ'. Moreover, the condition $\gamma' \subseteq \alpha' \cup r[\gamma]$ is clearly satisfied because $\gamma' = \alpha' \cup r[\gamma]$.

Theorem 4.1. *Let $\langle \kappa_1, \ldots, \kappa_n, \alpha \Vdash \gamma \rangle$ be a schema for multi-space system $\mathcal{M}$ with refinement $\langle \kappa'_1, \ldots, \kappa'_n, \alpha' \Vdash \gamma' \rangle$. Then $\langle \kappa'_1, \ldots, \kappa'_n, \alpha' \Vdash \gamma' \rangle$ is a schema for $\mathcal{M}$.*

Now we are in a position to define the notion of *application* of a schema to a sequent. Applications are done in either *forward* or *backward* manner. Both cases rely on finding a refinement of the schema where the context of the schema is identical to the context of the sequent. For a forward application we try to refine the schema so that the context and antecedent are identical to the sequent's assumptions, and this will induce a consequent that can be added to the assumptions. For a backward application we try to refine the schema so that the consequent is identical to the goal, and this yields an assumption that will replace the goal of the sequent.

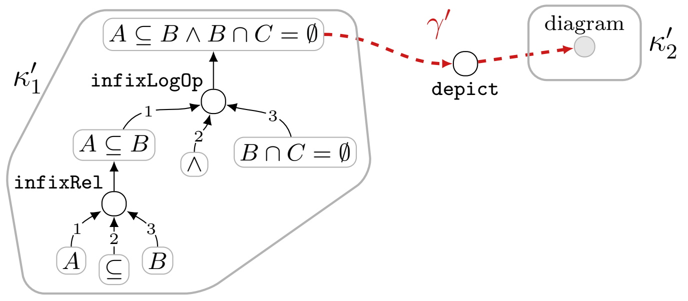
Definition 4.6. Let $\langle \kappa_1, \ldots, \kappa_n, \alpha \Vdash \gamma \rangle$ be a schema and $\langle \kappa'_1, \ldots, \kappa'_n, \alpha' \Vdash \gamma' \rangle$ be a sequent for multi-space system $\mathcal{M}$. An **application** of $\langle \kappa_1, \ldots, \kappa_n, \alpha \Vdash \gamma \rangle$ to $\langle \kappa'_1, \ldots, \kappa'_n, \alpha' \Vdash \gamma' \rangle$ is either

1. a sequent of the form $\langle \kappa'_1, \ldots, \kappa'_n, \alpha' \cup \delta \Vdash \gamma' \rangle$ where $\langle \kappa'_1, \ldots, \kappa'_n, \alpha' \Vdash \delta \rangle$ is a refinement of $\langle \kappa_1, \ldots, \kappa_n, \alpha \Vdash \gamma \rangle$, or
2. a sequent of the form $\langle \kappa'_1, \ldots, \kappa'_n, \alpha' \Vdash \delta \rangle$ where $\langle \kappa'_1, \ldots, \kappa'_n, \alpha' \cup \delta \Vdash \gamma' \rangle$ is a refinement of $\langle \kappa_1, \ldots, \kappa_n, \alpha \Vdash \gamma \rangle$.

Then $\langle \kappa'_1, \ldots, \kappa'_n, \alpha' \cup \delta \Vdash \gamma' \rangle$ and $\langle \kappa'_1, \ldots, \kappa'_n, \alpha' \Vdash \delta \rangle$ are, respectively, **δ-forward** and **δ-backward** applications of $\langle \kappa_1, \ldots, \kappa_n, \alpha \Vdash \gamma \rangle$ to $\langle \kappa'_1, \ldots, \kappa'_n, \alpha' \Vdash \gamma' \rangle$.

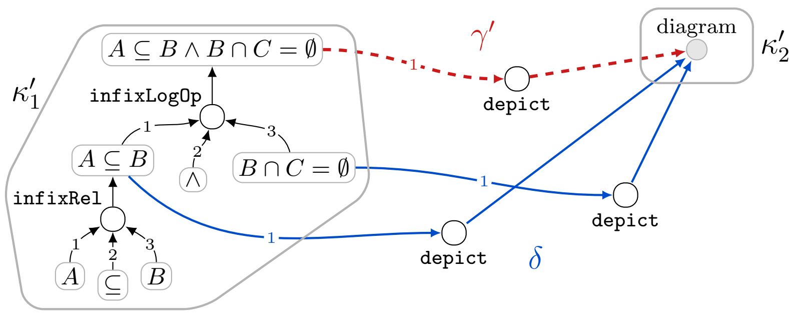
Example 4.7. Let $\mathcal{M} = (\mathcal{C}, \mathcal{D}, \mathcal{G})$ be a multi-space where $\mathcal{C}$ encodes Set Algebra, $\mathcal{D}$ encodes Euler Diagrams and $\mathcal{G}$ encodes relations across and within the spaces. Consider the sequent $\langle \kappa'_1, \kappa'_2, \alpha' \Vdash \gamma' \rangle$ where α' is the empty graph, as illustrated below.

$\langle \kappa'_1, \kappa'_2, \alpha' \Vdash \gamma' \rangle$:



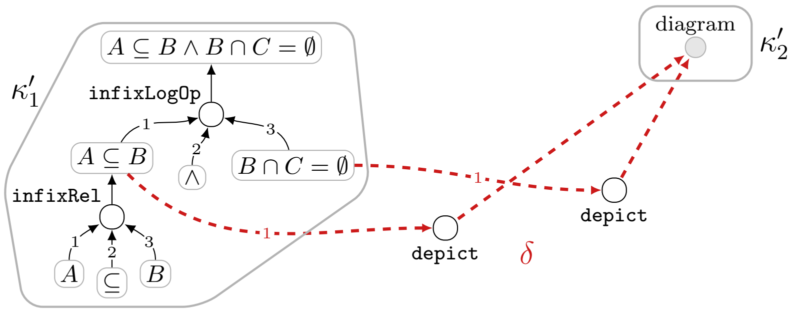
The schema $\langle \kappa_1, \kappa_2, \alpha \Vdash \gamma \rangle$ from Example 4.3 tells us that we can show that a diagram depicts a conjunction if it depicts both of the conjuncts. We will apply it backward to $\langle \kappa'_1, \kappa'_2, \alpha' \Vdash \gamma' \rangle$. For that purpose we need to refine it to suit our sequent. In other words, we need to find a δ such that $\langle \kappa_1', \kappa_2', \alpha' \cup \delta \Vdash \gamma' \rangle$ is a refinement of $\langle \kappa_1, \kappa_2, \alpha \Vdash \gamma \rangle$. Such a refinement is illustrated below:

$\langle \kappa_1', \kappa_2', \alpha' \cup \delta \Vdash \gamma' \rangle$:



Then, by Definition 4.6, the sequent $\langle \kappa_1', \kappa_2', \alpha' \Vdash \delta \rangle$, illustrated below, is a δ-backward application of $\langle \kappa_1, \kappa_2, \alpha \Vdash \gamma \rangle$ to $\langle \kappa_1', \kappa_2', \alpha' \Vdash \gamma' \rangle$:

$\langle \kappa_1', \kappa_2', \alpha' \Vdash \delta \rangle$:



Intuitively, we were able to replace the goal γ' with two subgoals, captured by δ, because the schema could be refined for the task.

The next theorem states that if we apply a schema to a sequent, $\langle \kappa_1', \ldots, \kappa_n', \alpha' \Vdash \gamma' \rangle$, and we obtain a schema, then $\langle \kappa_1', \ldots, \kappa_n', \alpha' \Vdash \gamma' \rangle$ must be a schema itself. Its proof can be found in Appendix B.

Theorem 4.2. *Let $\langle \kappa_1, \ldots, \kappa_n, \alpha \Vdash \gamma \rangle$ be a schema and $\langle \kappa_1', \ldots, \kappa_n', \alpha' \Vdash \gamma' \rangle$ be a sequent for multi-space system $\mathcal{M}$. If $\langle \kappa_1', \ldots, \kappa_n', \alpha'' \Vdash \gamma'' \rangle$ is an application of $\langle \kappa_1, \ldots, \kappa_n, \alpha \Vdash \gamma \rangle$ to $\langle \kappa_1', \ldots, \kappa_n', \alpha' \Vdash \gamma' \rangle$ and a schema for $\mathcal{M}$ then $\langle \kappa_1', \ldots, \kappa_n', \alpha' \Vdash \gamma' \rangle$ is also a schema for $\mathcal{M}$.*

Ultimately, the purpose of schema applications is to derive whether the goal pattern, γ, drawn from the meta-space, in a sequent can be instantiated, along with the assumptions, $(\kappa_1, \ldots, \kappa_n, \alpha)$. The sequents that result from the iterative application of schemas will be called *valid*. As we will show in Theorem 4.3, whose proof is in Appendix B, valid sequents are themselves schemas. The proof uses an induction approach, with Theorem 4.2 covering the inductive step.

Definition 4.7. Let $\mathbb{S}$ be a set of schemas and let $\langle \kappa_1', \ldots, \kappa_n', \alpha \Vdash \gamma' \rangle$ be a sequent for multi-space system $\mathcal{M}$. Then $\langle \kappa_1', \ldots, \kappa_n', \alpha \Vdash \gamma' \rangle$ is **valid over** $\mathbb{S}$ provided either

1. $\langle \kappa_1', \ldots, \kappa_n', \alpha' \Vdash \gamma' \rangle \in \mathbb{S}$, or

2. there exists $\langle \kappa_1, \ldots, \kappa_n, \alpha \Vdash \gamma \rangle \in \mathbb{S}$ and a sequent, $\langle \kappa'_1, \ldots, \kappa'_n, \alpha'' \Vdash \gamma'' \rangle$, for $\mathcal{M}$ such that $\langle \kappa'_1, \ldots, \kappa'_n, \alpha'' \Vdash \gamma'' \rangle$ is valid over $\mathbb{S}$ and it is an application of $\langle \kappa_1, \ldots, \kappa_n, \alpha \Vdash \gamma \rangle$ to $\langle \kappa'_1, \ldots, \kappa'_n, \alpha' \Vdash \gamma' \rangle$.

A sequent is **valid** if it is valid for some set, $\mathbb{S}$, of schemas.

Theorem 4.3 tells us that we can use schemas to prove that, given an instantiation of the assumptions, $(\kappa_1, \ldots, \kappa_n, \alpha)$, we can find an instantiation of the assumptions together with the goal, $(\kappa_1, \ldots, \kappa_n, \alpha \cup \gamma)$.

Theorem 4.3. *Let $\mathbb{S}$ be a set of schemas and let $\langle \kappa'_1, \ldots, \kappa'_n, \alpha' \Vdash \gamma' \rangle$ be a sequent for multi-space system $\mathcal{M}$. If $\langle \kappa'_1, \ldots, \kappa'_n, \alpha' \Vdash \gamma' \rangle$ is valid over $\mathbb{S}$ then it is also a schema for $\mathcal{M}$.*

Theorem 4.4. *Let $\langle \kappa_1, \ldots, \kappa_n, \alpha \Vdash \gamma \rangle$ be a valid sequent for multi-space system $\mathcal{M}$. If $(\kappa_1, \ldots, \kappa_n, \alpha)$ is instantiatable in $\mathcal{M}$ then so is $(\kappa_1, \ldots, \kappa_n, \alpha \cup \gamma)$.*

Example B.1, found in Appendix B, shows a sequence of schema applications in order to derive the validity of a given sequent.

4.1 Instantiation: determining pattern instantiatability

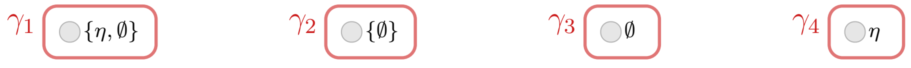
Here we address the problem of how to determine if a pattern graph in a construction space, $\mathcal{C} = (T, C, G)$, can be instantiated. Recall that the realm, G, of $\mathcal{C}$ is a graph that abides by the constraints set out by the type system, T, and the constructor specification, C. That is, whenever a vertex in the realm, G, is labelled by a constructor, c, its input and output vertices must be labelled by subtypes of those specified by the signature of c. However, not every graph that abides by the constraints lives in G. This is simply another way of saying that not every pattern graph, κ, for $\mathcal{C}$ can be instantiated in $\mathcal{C}$. Moreover, though a good modelling principle is that it should be *easy* to decide whether κ can be instantiated, this is not necessarily the case. Here we state the instantiation problem:

Problem Given a pattern graph, γ, for space $\mathcal{C}$, is it possible to instantiate γ in $\mathcal{C}$?

Solution If we determine that $\langle \ \Vdash \gamma \rangle$ is a schema for multi-space system $\mathcal{M} = (\mathcal{C})$ then γ can be instantiated in $\mathcal{C}$.

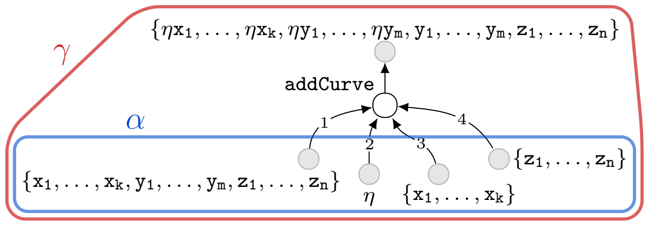
Now we can look back at the original instantiation problem proposed in Section 4. We can solve using schemas from Examples 4.8, 4.9, and 4.10, presented next.

Example 4.8 (Singleton instantiation)**.** For any label, η, the sequents $\langle \ \Vdash \gamma_1 \rangle$, $\langle \ \Vdash \gamma_2 \rangle$, $\langle \ \Vdash \gamma_3 \rangle$ and $\langle \ \Vdash \gamma_4 \rangle$, pictured below, are schemas for unary multi-space system $\mathcal{D}$.



Example 4.9 (Bottom-up instantiation of an `addCurve` configuration)**.** Provided that `addCurve` takes as inputs: a diagram of type $\{\mathtt{x_1}, \ldots, \mathtt{x_k}, \mathtt{y_1}, \ldots, \mathtt{y_m}, \mathtt{z_1}, \ldots, \mathtt{z_n}\} \leq \mathtt{diagram}$, a label $\eta \leq \mathtt{label}$, an *in* region $\{\mathtt{x_1}, \ldots, \mathtt{x_k}\} \leq \mathtt{region}$, and an *out* region $\{\mathtt{z_1}, \ldots, \mathtt{z_n}\} \leq \mathtt{region}$, where η does not appear in $\{\mathtt{x_1}, \ldots, \mathtt{x_k}, \mathtt{y_1}, \ldots, \mathtt{y_m}, \mathtt{z_1}, \ldots, \mathtt{z_n}\}$; that is, if η is not a

label in any of the words, *then* we can infer the output. This is captured by the family of schemas $\langle \alpha \Vdash \gamma \rangle$, pictured below, for unary multi-space system $\mathcal{D}$:



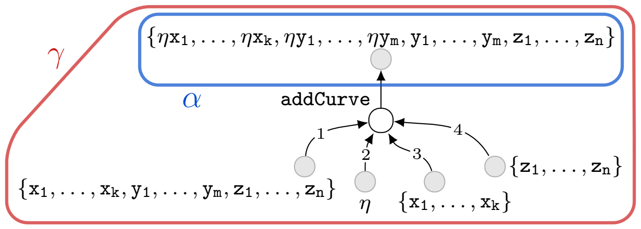
γ
$\{\eta\mathtt{x}_1, \ldots, \eta\mathtt{x}_k, \eta\mathtt{y}_1, \ldots, \eta\mathtt{y}_m, \mathtt{y}_1, \ldots, \mathtt{y}_m, \mathtt{z}_1, \ldots, \mathtt{z}_n\}$
addCurve
α
1
2
3
4
$\{\mathtt{x}_1, \ldots, \mathtt{x}_k, \mathtt{y}_1, \ldots, \mathtt{y}_m, \mathtt{z}_1, \ldots, \mathtt{z}_n\}$
η
$\{\mathtt{x}_1, \ldots, \mathtt{x}_k\}$
$\{\mathtt{z}_1, \ldots, \mathtt{z}_n\}$

Note that in the output, with type $\{\eta\mathtt{x}_1, \ldots, \eta\mathtt{x}_k, \eta\mathtt{y}_1, \ldots, \eta\mathtt{y}_m, \mathtt{y}_1, \ldots, \mathtt{y}_m, \mathtt{z}_1, \ldots, \mathtt{z}_n\}$, the curve labelled by η:

- *contains* every zone, $\mathtt{x}_\mathtt{i}$, of the *in* region,
- *excludes* every zone, $\mathtt{z}_\mathtt{i}$, of the *out* region, and
- *cuts* each zone, $\mathtt{y}_\mathtt{i}$, in two: one contained ($\eta\mathtt{y}_\mathtt{i}$), and one excluded ($\mathtt{y}_\mathtt{i}$).

If we see the constructor `addCurve` as a program, this family of schemas represents knowledge about how to *compute* this program.

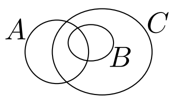
Example 4.10 (Top-down instantiation of an `addCurve` configuration)**.** Provided a diagram of type $\{\eta\mathtt{x}_1, \ldots, \eta\mathtt{x}_k, \eta\mathtt{y}_1, \ldots, \eta\mathtt{y}_m, \mathtt{y}_1, \ldots, \mathtt{y}_m, \mathtt{z}_1, \ldots, \mathtt{z}_n\} \leq \mathtt{diagram}$, where η is a label that does not appear in either $\mathtt{x}_1, \ldots, \mathtt{x}_k, \mathtt{y}_1, \ldots, \mathtt{y}_m, \mathtt{z}_1, \ldots, \mathtt{z}_n$ then we can decompose $\{\eta\mathtt{x}_1, \ldots, \eta\mathtt{x}_k, \eta\mathtt{y}_1, \ldots, \eta\mathtt{y}_m, \mathtt{y}_1, \ldots, \mathtt{y}_m, \mathtt{z}_1, \ldots, \mathtt{z}_n\}$ by removing η. This is captured by the family of schemas $\langle \alpha \Vdash \gamma \rangle$, pictured below, for unary multi-space system $\mathcal{D}$:



γ
$\{\eta\mathtt{x}_1, \ldots, \eta\mathtt{x}_k, \eta\mathtt{y}_1, \ldots, \eta\mathtt{y}_m, \mathtt{y}_1, \ldots, \mathtt{y}_m, \mathtt{z}_1, \ldots, \mathtt{z}_n\}$
α
addCurve
1
2
3
4
$\{\mathtt{x}_1, \ldots, \mathtt{x}_k, \mathtt{y}_1, \ldots, \mathtt{y}_m, \mathtt{z}_1, \ldots, \mathtt{z}_n\}$
η
$\{\mathtt{x}_1, \ldots, \mathtt{x}_k\}$
$\{\mathtt{z}_1, \ldots, \mathtt{z}_n\}$

This family of schemas represents knowledge about how to *parse* an Euler diagram.

The schemas above are sufficient for solving the instantiation problem. Figure 1 shows one way of doing it. The resulting construction graph encodes two ways of building the diagram shown here (right).



The following theorem is simply a corollary of Theorem 4.4.

Theorem 4.5. *Let $\mathbb{S}$ be a set of schemas and let $\langle \Vdash \gamma \rangle$ be a sequent for construction space $\mathcal{C}$ that is valid over $\mathbb{S}$. Then γ can be instantiated in $\mathcal{C}$.*

The significance of this theorem is that we can use schemas iteratively to derive new knowledge about which pattern graphs can be instantiated in $\mathcal{C}$. We have seen that schemas can be *applied* to derive such new knowledge.

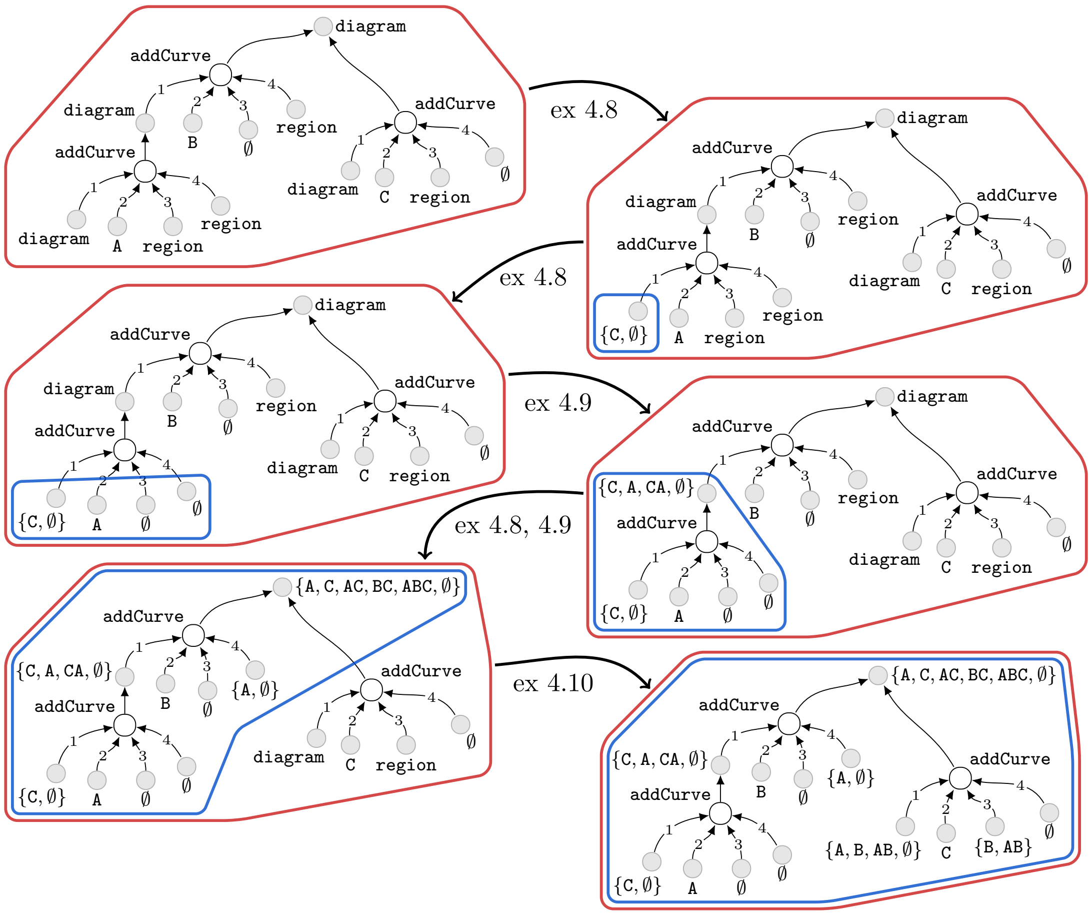


Figure 1: Solving an instantiation problem with forward-applications.

4.2 Inference: Determining Whether Meta-Properties Hold

Another important application of schemas is for identifying meta-properties across construction spaces in a multi-space system, $\mathcal{M}$. That is, we can use schemas to derive whether a specific property of tokens in $\mathcal{M}$ holds, provided that this property is expressed in the meta-space, $\mathcal{G}$. Thus, in this context, schemas are like inference rules for the relations expressible in $\mathcal{G}$.

Problem Take pattern graphs $\kappa_1, ..., \kappa_n$ for construction spaces $\mathcal{C}_1, \ldots, \mathcal{C}_n$, respectively. Assume they can be simultaneously instantiated to satisfy some constraint P. Can they also be instantiated to satisfy some constraint Q?

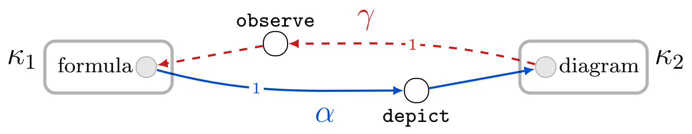
Solution Take a meta-space, $\mathcal{G}$, for which $(\mathcal{C}_1, \ldots, \mathcal{C}_n, \mathcal{G})$ is a multi-space system, where the constraints P and Q can be encoded as α and γ, respectively. Determine whether $\langle \kappa_1, ..., \kappa_n, \alpha \Vdash \gamma \rangle$ is a schema for $(\mathcal{C}_1, \ldots, \mathcal{C}_n, \mathcal{G})$.

Schemas for $(\mathcal{C}_1, \ldots, \mathcal{C}_n, \mathcal{G})$ can model inference rules if the antecedent and consequent, which are graphs in $\mathcal{G}$, model relations between the tokens of spaces $\mathcal{C}_1, \ldots, \mathcal{C}_n$. For any schema, $\langle \kappa_1, \ldots, \kappa_n, \alpha \Vdash \gamma \rangle$, where $\langle \kappa_1, \ldots, \kappa_n, \alpha \Vdash \gamma \rangle$ is monotonic for *every* graph

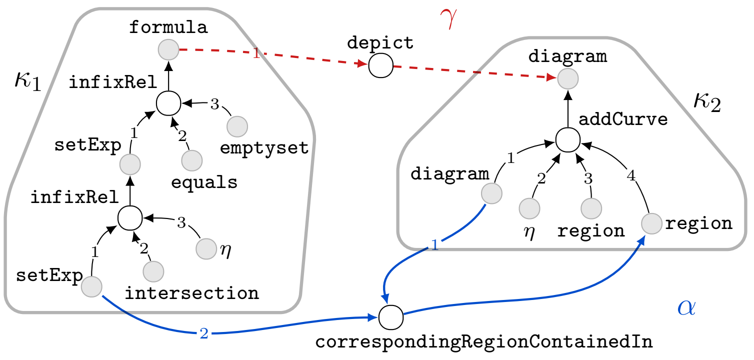
$(\kappa'_1, \ldots, \kappa'_n, \alpha')$, we have an analogous concept to logical monotonicity, which is often expected in logical calculi. Given a schema, $\langle \kappa_1, \ldots, \kappa_n, \alpha \Vdash \gamma \rangle$, if we can instantiate $(\kappa_1, \ldots, \kappa_n, \alpha)$ we must, by definition, be able to instantiate $(\kappa_1, \ldots, \kappa_n, \alpha \cup \gamma)$. Thus, if α encodes a relation, P, between the tokens of the graphs $\kappa_1, \ldots, \kappa_n$, and γ encodes another relation, Q, between the tokens of the graphs $\kappa_1, \ldots, \kappa_n$, then the schema $\langle \kappa_1, \ldots, \kappa_n, \alpha \Vdash \gamma \rangle$ encodes the fact that for any instantiation of $\kappa_1, \ldots, \kappa_n$ that satisfies P, we can find an instantiation that also satisfies Q. Importantly, by definition, schemas do not need to be monotonic. The implications of this are yet to be studied, but in principle this allows us to encode non-monotonic logics for inference in multi-spaces.

Now we can look back at the specific inference problem described at the start of Section 4. The following examples describe schemas that can be used to solve this problem.

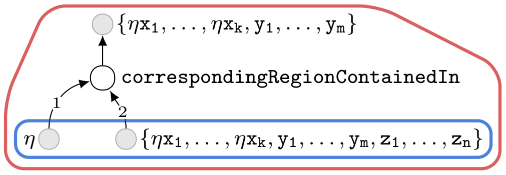
Example 4.11 (Observe and depict are dual)**.** If a diagram depicts a formula, then the same formula can be observed from the diagram. Thus $\langle \kappa_1, \kappa_2, \alpha \Vdash \gamma \rangle$, depicted below, is a schema.

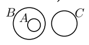


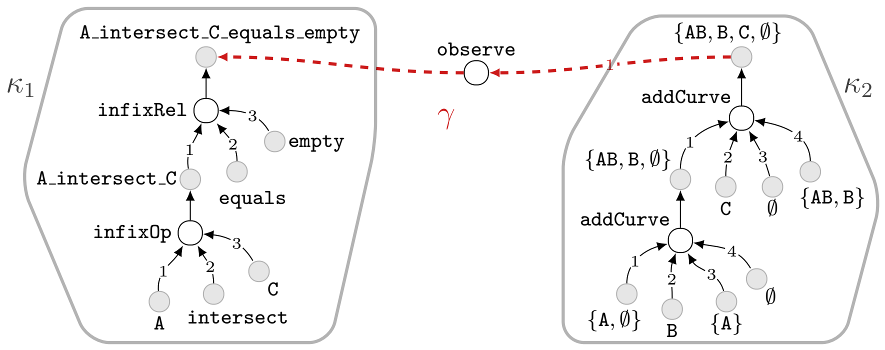
Example 4.12 (Depicting disjoint sets)**.** To depict an expression of the form $x \cap \eta = \emptyset$, where x is a set expression and η is a variable, it suffices to take a diagram, d, and add η in such a way that the region corresponding to x in d is part of the *out* regions of the curve labelled by η (based on Stapleton et al. (2010)). Thus, the following figure describes a family of schemas, one for each η, subtype of `var` in SET ALGEBRA, and subtype of `label` in EULER DIAGRAMS.



Example 4.13 (Set expressions and their corresponding regions)**.** This schema tells you how to propagate from the input to the output of the `correspondingRegionContainedIn` constructor. For every $\mathtt{x}_1, \ldots, \mathtt{x}_k, \mathtt{y}_1, \ldots, \mathtt{y}_m, \mathtt{z}_1, \ldots, \mathtt{z}_n$ and η the following is a schema:

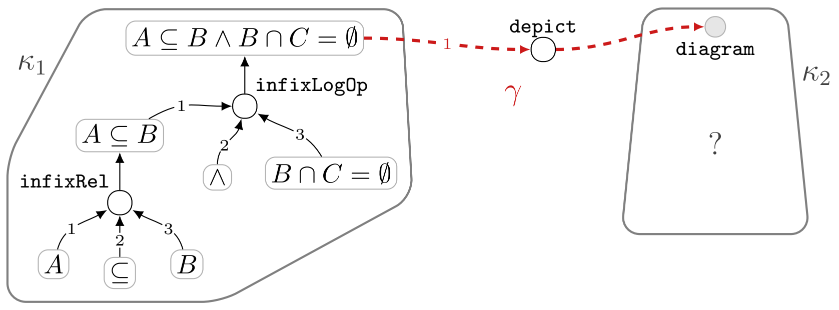

$\{\eta\mathtt{x}_1, \ldots, \eta\mathtt{x}_k, \mathtt{y}_1, \ldots, \mathtt{y}_m\}$
correspondingRegionContainedIn
1
2
η
$\{\eta\mathtt{x}_1, \ldots, \eta\mathtt{x}_k, \mathtt{y}_1, \ldots, \mathtt{y}_m, \mathtt{z}_1, \ldots, \mathtt{z}_n\}$

It is easy to see that a sequence of three applications of the schemas from Examples 4.11, 4.12 and 4.13 is sufficient to solve the inference problem (redrawn below). This simply means that the statement $A \cap C = \emptyset$ can be observed from the diagram :



4.3 Transfer Schemas: Abducting Structure

As we have seen, schemas are used to determine whether a relation holds between some tokens in a multi-space, *given* a context structure graph where such tokens live. For example, we can use a set of schemas to determine whether a sequent, $\langle \kappa_1, \kappa_2, \alpha \Vdash \gamma \rangle$, is itself a schema. In this case κ_1 and κ_2 are the given context for the tokens. Now consider the case where we know κ_1 and we want to *find* some yet-unknown κ_2 such that $\langle \kappa_1, \kappa_2, \alpha \Vdash \gamma \rangle$ holds. For instance, we may be given a set-theoretic formula with structure κ_1, and we may want to find an Euler diagram with structure κ_2 that depicts it, as drawn below[11]:



Thus, the problem in this case is stated roughly as follows: we are given a source graph, κ_1, an assumption graph, α (empty in this case), a goal graph, γ, and an *initial* target graph, κ_2' (with a single vertex) which we want to *reify*[12] into some graph κ_2 in such a way that $\langle \kappa_1, \kappa_2, \alpha \Vdash \gamma \rangle$ is provably a schema. As we will see, the *application* of *transfer schemas* will differ from the application of schemas in that transfer schema applications reify a target graph.

The next problem statement formalises this in a more general way, where an arbitrary number of the given context graphs is allowed reification.

11. For illustration purposes we are not showing the structure of $B \cap C = \emptyset$ here.
12. Recall that reifying means adding context and specialising types.

Problem Take pattern graphs $\kappa_1, ..., \kappa_n$ for construction spaces $\mathcal{C}_1, \ldots, \mathcal{C}_n$, respectively. Take a set $\sigma \subseteq \{1, \ldots, n\}$ that indicates the target. Is it possible to reify the target graphs, κ_i with $i \in \sigma$, in such a way that any instantiation of them satisfying P ensures an instantiation of them satisfying Q?

Solution Encode P and Q as pattern graphs α and γ respectively, for some multi-space system $(\mathcal{C}_1, \ldots, \mathcal{C}_n, \mathcal{G})$. Determine whether there exists a sequent, $\langle \kappa'_1, ..., \kappa'_n, \alpha' \Vdash \gamma' \rangle$, and a reification function $f \colon (\kappa_1, \ldots, \kappa_n, \alpha \cup \gamma) \to (\kappa'_1, \ldots, \kappa'_n, \alpha' \cup \gamma')$ such that

1. the restrictions $f|_\alpha \colon \alpha \to \alpha'$, $f|_\gamma \colon \gamma \to \gamma'$ and every $f|_{\kappa_j} \colon \kappa_j \to \kappa'_j$ for $j \notin \sigma$ are label-preserving isomorphisms (up to the tokens shared with $\bigcup_{i \in \sigma} \kappa_i$), and
2. $\langle \kappa'_1, ..., \kappa'_n, \alpha' \Vdash \gamma' \rangle$ is a schema.

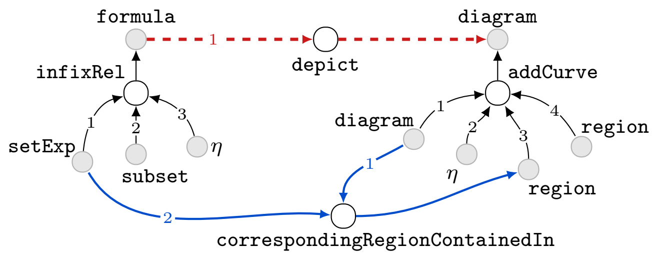
As we saw, schemas can capture invariants across multi-space systems. Now we define σ-transfer schema as a schema paired with a set, σ. This set simply provides information about which dimensions will be targetted for transferring structure to them.

Definition 4.8. A **σ-transfer schema** for $\mathcal{M}$ is a pair consisting of a schema, $\langle \kappa_1, \ldots, \kappa_n, \alpha \Vdash \gamma \rangle$, for $\mathcal{M}$, and a set, $\sigma \subseteq \{1, \ldots, n\}$.

Applying a σ-transfer schema, $\langle \kappa_1, \ldots, \kappa_n, \alpha \Vdash \gamma \rangle$, to a sequent, $\langle \kappa'_1, \ldots, \kappa'_n, \alpha' \Vdash \gamma' \rangle$, will be similar to applying it as a schema, except that we are allowed to reify the graphs indexed by σ.

Let us begin by illustrating with some examples of schemas where $n = 2$, that is, schemas of the form $\langle \kappa_1, \kappa_2, \alpha \Vdash \gamma \rangle$. Then, we will see that choosing a σ means selecting the target to produce structure while applying them.

Example 4.14 (Depicting subsets)**.** To depict an expression of the form $x \subseteq \eta$, where η is a variable, it suffices to take a diagram, d, and make sure that η is drawn so that the region corresponding to x is part of the *in* regions of the curve labelled by η (based on Stapleton et al. (2010)). This means that **for every** η the following is a schema[13]:

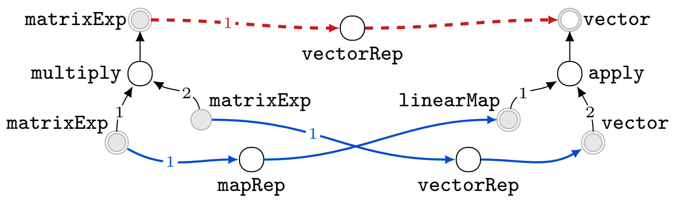


Using this schema as a $\{2\}$-transfer schema would produce a diagram from a given formula[14].

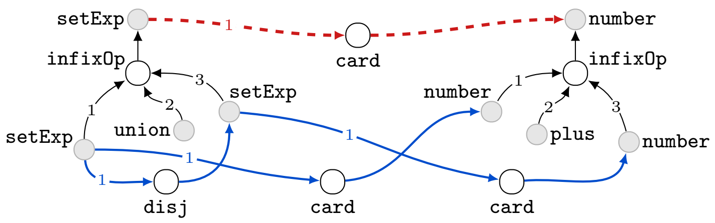
13. Hereafter we will draw sequents and schemas of the form $\langle \kappa_1, \kappa_2, \alpha \Vdash \gamma \rangle$ with κ_1 on the left, κ_2 on the right, α with thick blue arrows, and γ with dashed red arrows.

14. Moreover, using this schema as a $\{1\}$-transfer schema can be used for producing a formula for a given diagram. Using it as $\{1, 2\}$-transfer schema can be used to produce both the given assumptions and goals. Using it as a $\emptyset$-transfer schema is no different to using it as a schema.

Example 4.15 (Matrices and linear maps)**.** As we hinted in Example 3.4, the homomorphism between matrices and linear maps can be represented as a schema.



Example 4.16 (Cardinality)**.** The relation that links every finite set to its cardinality is respected along some *corresponding* operations. We can encode these properties as schemas, using constructors `card` and `disj` in an meta-space between some set-theory space and some number-theory space. For example, the diagram below depicts a schema which states that the union of disjoint sets is analogous to addition. The blue graph encodes the antecedent: that A and B are disjoint and that $|A| = n$ and $|B| = m$, and the red graph encodes the consequent: that $|A \cup B| = n + m$.



These examples capture homomorphisms which, as we have seen in Section 4.2, can be used for proving whether some specified relation holds. This is ideal to use them as σ-transfer schemas to produce a desirable structure in the target graphs specified by σ. As we will see, *structure transfer* is a calculus that uses transfer schemas to *abduct*[15] the structure of some target graphs in order to ensure that the specified relations hold. Our abductive approach is based on reification of the selected target graphs. Given some sequent, $\langle \kappa_1, ..., \kappa_n, \alpha \Vdash \gamma \rangle$, and a set $\sigma \subseteq \{1, ..., n\}$, we will be allowed to reify those graphs indexed by σ to ensure that the sequent is valid. The operation that does precisely this is called a *σ-reification*.

Definition 4.9. Let $\langle \kappa_1, ..., \kappa_n, \alpha \Vdash \gamma \rangle$ and $\langle \kappa'_1, ..., \kappa'_n, \alpha' \Vdash \gamma' \rangle$ be sequents for multi-space system $\mathcal{M}$ and let $\sigma \subseteq \{1, ..., n\}$. Then $\langle \kappa'_1, ..., \kappa'_n, \alpha' \Vdash \gamma' \rangle$ is a **σ-reification** of $\langle \kappa_1, ..., \kappa_n, \alpha \Vdash \gamma \rangle$ provided there exists a reification function $f \colon (\kappa_1, ..., \kappa_n, \alpha \cup \gamma) \to (\kappa'_1, ..., \kappa'_n, \alpha' \cup \gamma')$ such that $f|_\alpha \colon \alpha \to \alpha'$, $f|_\gamma \colon \gamma \to \gamma'$ and every $f|_{\kappa_j} \colon \kappa_j \to \kappa'_j$ where $j \notin \sigma$, are a label-preserving isomorphisms up to the tokens of $\bigcup_{i \in \sigma} \kappa_i$. In this context we call $\{\kappa_j : j \notin \sigma\}$ the **source** and $\{\kappa_i : i \in \sigma\}$ the **target** of the σ-reification.

A σ-reification can only modify the structure and types that appear in the target specified by σ. Everything else must be mapped through a label-preserving isomorphism, which means neither the structure nor the types can be changed. A σ-reification is not a deductive operation as the context is specialised and enlarged[16]. However, it is constrained to ensure

15. In logic, abduction is an operation by which, given a set of statements (e.g., a set of observations), a sufficient set of axioms (or a model) is produced that entails the set of statements.

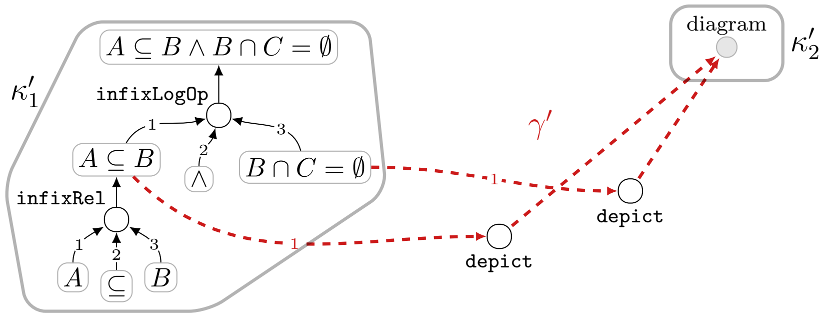
16. This is analogous to adding constraints and assumptions to a conjecture.

that the speculative structure introduced in the target graphs does not go beyond them into the source graphs or the antecedent.

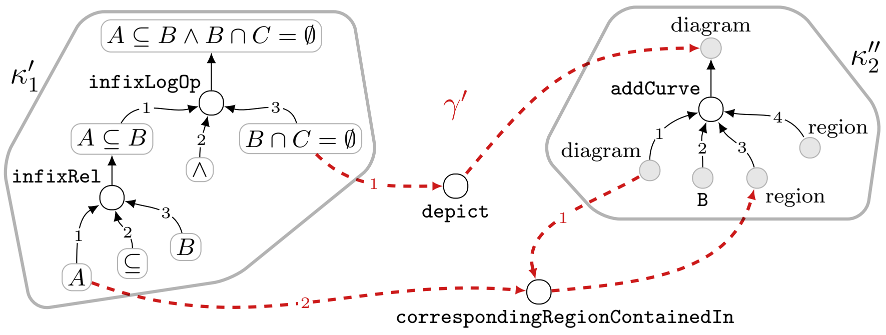
We can define the notion of application of σ-transfer schemas, which allows us to reify the target graphs.

Definition 4.10. A **σ-transfer schema application** of $\langle \kappa_1, ..., \kappa_n, \alpha \Vdash \gamma \rangle$ to $\langle \kappa'_1, ..., \kappa'_n, \alpha' \Vdash \gamma' \rangle$ is any application of $\langle \kappa_1, ..., \kappa_n, \alpha \Vdash \gamma \rangle$ to a σ-reification of $\langle \kappa'_1, ..., \kappa'_n, \alpha' \Vdash \gamma' \rangle$.

Example 4.17. We saw in Example 4.7 how applying a schema to a sequent yields a sequent, here renamed $\langle \kappa'_1, \kappa'_2, \alpha' \Vdash \gamma' \rangle$, with empty α', as visualised below.



Note that this sequent is *not* a schema, given that not every instantiation of κ'_2 will depict $A \subseteq B$ and $B \cap C = \emptyset$. However, what we want to know is whether there *exists* an instantiation of κ'_2 that satisfies the conditions encoded by γ'. Transfer schema applications can help us here. The schema from Example 4.14 tells us that an Euler diagram depicts a subset expression when it is built in a particular way, so we can use this information to build the diagram[17]. Note that, with the given κ'_2 we would not be able to apply the schema. However, we can apply the schema as a $\{2\}$-transfer schema, allowing us to reify κ'_2. This $\{2\}$-reification *unblocks* the sequent so that we can apply the schema. The figure below shows the result of reifying κ'_2 into some graph κ''_2 and applying the schema – that is, the result of applying it (backwards) as a $\{2\}$-transfer schema. Intuitively, the application of the $\{2\}$-transfer schema gives us sufficient structure to match the context, and reduces the goal of depicting $A \subseteq B$ to the goal of ensuring that the a region corresponding to A is contained in the *in* region of B.



17. Note that, given also the graph that constructs $B \cap C = \emptyset$ (which we omit for simplicity), we have a choice whether to use the schema for $A \subseteq B$ or to use the schema from Example 4.12. The choice here is only for demonstration purposes.

Below we define *validity modulo* σ – that is, when a sequent is one σ-reification away from becoming valid. Crucially, we can prove that a sequence of σ-transfer schema applications can be used to derive whether a sequent is valid modulo σ.

Definition 4.11. A sequent, $\langle \kappa_1, ..., \kappa_n, \alpha \Vdash \gamma \rangle$, is **valid modulo** σ provided there exists a valid sequent, $\langle \kappa'_1, ..., \kappa'_n, \alpha' \Vdash \gamma' \rangle$, that is a σ-reification of $\langle \kappa_1, ..., \kappa_n, \alpha \Vdash \gamma \rangle$.

The next theorem establishes that if we σ-apply transfer schemas sequentially to a sequent and reach a valid sequent this means that the original sequent is valid modulo σ.

Theorem 4.6. *Let* $\langle \kappa_1, ..., \kappa_n, \alpha \Vdash \gamma \rangle$ *be a sequent for* $\mathcal{M}$*, let* $\sigma \subseteq \{1, \dots, n\}$*, and let* $\mathbb{T}$ *be a set of* σ*-transfer schemas. Assume we apply these schemas sequentially, starting with* $\langle \kappa_1, ..., \kappa_n, \alpha \Vdash \gamma \rangle$ *and ending with* $\langle \kappa'_1, ..., \kappa'_n, \alpha' \Vdash \gamma' \rangle$*. If all schemas in* $\mathbb{T}$ *are monotonic for* $(\kappa'_1, ..., \kappa'_n, \alpha')$ *and* $\langle \kappa'_1, ..., \kappa'_n, \alpha' \Vdash \gamma' \rangle$ *is a valid sequent, then* $\langle \kappa_1, ..., \kappa_n, \alpha \Vdash \gamma \rangle$ *is valid modulo* σ*.*

The significance of this theorem is the basis for *structure transfer*: given a sequent, $\langle \kappa_1, ..., \kappa_n, \alpha \Vdash \gamma \rangle$, and $\sigma \subseteq \{1, \dots, n\}$, we can apply σ-transfer schemas iteratively in such a way that, in each application, the target is reified enough for the schema to be applicable. If after such a sequence of applications we manage to obtain a valid sequent, $\langle \kappa'_1, ..., \kappa'_n, \alpha' \Vdash \gamma' \rangle$, then the target graphs (those indexed by σ) are a transformation of the source graphs (those not indexed by κ_i).

We have seen that applying a transfer schema is just like applying a schema, but with the additional power to reify the target graphs while everything else is only mapped through label-preserving isomorphisms. Such modifications of a sequent make a difference when the target pattern graph of a transfer schema could not be matched before the σ-reification but can be matched after it.

5. Structure Transfer: an algorithmic approach

Structure transfer is a calculus by which we obtain a σ-reification of a given sequent that makes it valid for a multi-space $\mathcal{M}$. All the ingredients for this have already been presented. Specifically, Theorem 4.6 asserts that we can find such σ-reification by σ-applying transfer schemas sequentially. So let us review the conceptual tools we presented before, now under an algorithmic lens.

A fundamental property of the algorithms presented here is that they are non-deterministic. Specifically, in our pseudo-code, we include the use of an operator **find** which may get zero, one, or more results, in which case we are dealing respectively with either a dead end, a deterministic path or a branching of the search space. Thus, ultimately, structure transfer is a calculus, meaning that it establishes a set of rules and procedures that can be applied in a variety of ways. To develop specific methods and strategies for applying the rules of structure transfer appropriate heuristics must be used. In this paper we do not deal with such strategies and heuristics.

5.1 Procedures for Schema Application

Algorithms 1 and 2 produce forward and backward applications of a schema $\langle \kappa_1, \dots, \kappa_n, \alpha \Vdash \gamma \rangle$ to a sequent $\langle \kappa'_1, \dots, \kappa'_n, \alpha' \Vdash \gamma' \rangle$ for a multi-space $\mathcal{M}$. Recall that the definition of a schema application involves finding a refinement of the schema to resemble the sequent in question, with a monotonicity constraint. In this paper we do not deal with the general problem of determining monotonicity, and most examples we have presented are trivially monotonic. Thus, the algorithms assume that knowledge about monotonicity is given, through some set $\mathbb{Q}$ whose elements are pairs of the form $(\texttt{sc}, (\kappa_1, \ldots, \kappa_n, \alpha))$ where $\texttt{sc}$ is a schema and $(\kappa_1, \ldots, \kappa_n, \alpha)$ is pattern graph for which the schema is monotonic. If a schema, $\texttt{sc}$, is monotonic in general then $\mathbb{Q}$ has all pairs of the form $(\texttt{sc}, (\kappa_1, \ldots, \kappa_n, \alpha))$.

The approach of Algorithm 1 to forward-apply $\langle \kappa_1, \ldots, \kappa_n, \alpha \Vdash \gamma \rangle$ to $\langle \kappa'_1, \ldots, \kappa'_n, \alpha' \Vdash \gamma' \rangle$ involves first finding whether $(\kappa_1, \ldots, \kappa_n, \alpha)$ can be reified into $(\kappa'_1, \ldots, \kappa'_n, \alpha')$, and if so, we can loosen the consequent, γ, which yields a δ, which can be added as a new assumption to obtain the result of the application (provided the monotonicity condition is met).

Algorithm 1 Forward application of schema to sequent

```
1: procedure applySchemaFWD($\langle \kappa_1, \ldots, \kappa_n, \alpha \Vdash \gamma \rangle, \langle \kappa'_1, \ldots, \kappa'_n, \alpha' \Vdash \gamma' \rangle, \mathbb{Q}$)
2:     find reification $f\colon (\kappa_1, \ldots, \kappa_n, \alpha) \to (\kappa'_1, \ldots, \kappa'_n, \alpha')$, if none found fail
3:     find $\delta$ and partial function $\ell\colon \gamma \to \delta$ s.t:
4:         • $\ell|_\gamma\colon \gamma \to \ell[\gamma]$ is a loosening map up to the tokens of $(\kappa_1 \cup \cdots \cup \kappa_n \cup \alpha) \cap \gamma$
5:         • $\ell$ is compatible with $f$, and
6:         • $\delta \subseteq \alpha' \cup \ell[\gamma]$
7:     if $(\langle f[\kappa_1], \ldots, f[\kappa_n], f[\alpha] \Vdash \ell[\gamma] \rangle, (\kappa'_1, \ldots, \kappa'_n, \alpha')) \in \mathbb{Q}$ then
8:         return $\langle \kappa'_1, \ldots, \kappa'_n, \alpha' \cup \delta \Vdash \gamma' \rangle$
```

Conversely, the approach of Algorithm 2 to backward-apply $\langle \kappa_1, \ldots, \kappa_n, \alpha \Vdash \gamma \rangle$ to $\langle \kappa'_1, \ldots, \kappa'_n, \alpha' \Vdash \gamma' \rangle$ involves first finding whether γ can be loosened into γ' (up to some tokens) and then finding whether $(\kappa_1, \ldots, \kappa_n, \alpha)$ can be reified into some $(\kappa'_1, \ldots, \kappa'_n, \alpha' \cup \delta)$. Here we note that the condition in line 7 means δ must contain every part of γ' that was not mapped by the loosening of γ or is already in α'.

Algorithm 2 Backward application of schema to sequent

```
1: procedure applySchemaBWD($\langle \kappa_1, \ldots, \kappa_n, \alpha \Vdash \gamma \rangle, \langle \kappa'_1, \ldots, \kappa'_n, \alpha' \Vdash \gamma' \rangle, \mathbb{Q}$)
2:     find partial function $\ell\colon \gamma \to \gamma'$ s.t.
3:         • $\ell|_\gamma\colon \gamma \to \ell[\gamma]$ is a loosening map up to the tokens of $(\kappa_1 \cup \cdots \cup \kappa_n \cup \alpha) \cap \gamma$
4:         if none found fail
5:     find $\delta$ and reification $f\colon (\kappa_1, \ldots, \kappa_n, \alpha) \to (\kappa'_1, \ldots, \kappa'_n, \alpha' \cup \delta)$ s.t.
6:         • $f$ is compatible with $\ell$, and
7:         • $\gamma' \subseteq \alpha' \cup \delta \cup \ell[\gamma]$
8:         if none found fail
9:     if $(\langle f[\kappa_1], \ldots, f[\kappa_n], f[\alpha] \Vdash \ell[\gamma] \rangle, (\kappa'_1, \ldots, \kappa'_n, \alpha' \cup \delta)) \in \mathbb{Q}$ then
10:        return $\langle \kappa'_1, \ldots, \kappa'_n, \alpha' \Vdash \delta \rangle$
```

Algorithm 3, for general schema involves choosing between forward or backward applications. Again note the use of the **find** operator which identifies potential branching places in the search space.

Now, given procedures for applying schemas, we can check for validity with Algorithm 4.

Algorithm 3 Application of schema to sequent

```
1: procedure applySchema($\texttt{sc}, \texttt{sq}, \mathbb{Q}$)
2:     find sq' s.t.
3:         sq' = applySchemaFWD($\texttt{sc}, \texttt{sq}, \mathbb{Q}$) or sq' = applySchemaBWD($\texttt{sc}, \texttt{sq}, \mathbb{Q}$)
4:     return sq'
```

Algorithm 4 Validity of sequent

```
1: procedure validSequent($\texttt{sq}, \mathbb{S}, \mathbb{Q}$)
2:     if $\texttt{sq} \in \mathbb{S}$ then
3:         return true
4:     else
5:         find $\texttt{sc} \in \mathbb{S}$ and sq' s.t. sq' = applySchema($\texttt{sc}, \texttt{sq}, \mathbb{Q}$)
6:             return validSequent(sq', $\mathbb{S}, \mathbb{Q}$)
7:             if none found fail
```

5.2 Procedures for Transfer Schema applications

To apply a σ-transfer schema to a sequent means to produce a σ-reification of the sequent so that the schema is applicable. Thus, Algorithm 5 starts by reifying every target graph, κ'_i, with $i \in \sigma$, to some graph κ''_i in such a way that κ''_i is also a reification of κ_i (lines 2 to 4). This adds the context to every κ'_i necessary for the schema to be applicable. Next, we find the label-preserving isomorphism of the rest of the sequent (lines 5 to 9), and return the schema application (line 10).

Algorithm 5 Application of transfer schema to sequent, for $\sigma = \{i_1, \ldots, i_k\}$

```
1: procedure applyTransferSchema($\{i_1, \ldots, i_k\}, \langle \kappa_1, \ldots, \kappa_n, \alpha \Vdash \gamma \rangle, \langle \kappa'_1, \ldots, \kappa'_n, \alpha' \Vdash \gamma' \rangle, \mathbb{Q}$)
2:     find $\kappa''_{i_1}, \ldots, \kappa''_{i_k}$ and homomorphism $f \colon (\kappa'_{i_1}, \ldots, \kappa'_{i_k}) \to (\kappa''_{i_1}, \ldots, \kappa''_{i_k})$ s.t. for all $1 \le l \le k$,
3:         • $f|_{\kappa'_{i_l}} \colon \kappa'_{i_l} \to \kappa''_{i_l}$ is a reification function, and
4:         • $\kappa''_{i_l}$ is a reification of $\kappa_{i_l}$
5:     let $\{j_1, \ldots, j_{n-k}\} = \{i, \ldots, n\} \setminus \{i_1, \ldots, i_k\}$
6:     find $\kappa''_{j_1}, \ldots, \kappa''_{j_{n-k}}, \alpha'', \gamma''$ and isomorphism
7:           $h \colon (\kappa'_{j_1}, \ldots, \kappa'_{j_{n-k}}, \alpha', \gamma') \to (\kappa''_{j_1}, \ldots, \kappa''_{j_{n-k}}, \alpha'', \gamma'')$ s.t.
8:         • $h$ is label-preserving up to the tokens of $\kappa'_{i_1} \cup \cdots \cup \kappa'_{i_k}$ and
9:         • $h$ is compatible with $f$.
10:    return applySchema($\langle \kappa_1, \ldots, \kappa_n, \alpha \Vdash \gamma \rangle, \langle \kappa''_1, \ldots, \kappa''_n, \alpha'' \Vdash \gamma'' \rangle, \mathbb{Q}$)
```

Finally, Algorithm 6, for structure transfer, relies on the recursive, non-deterministic, application of σ-transfer schemas until a valid sequent is found, which, as we know from Theorem 4.6, means the original sequent was valid modulo σ. If such a valid sequent is found, this is the transformation of the original we want, so we call this a `full` transformation. Otherwise we return a `partial`. With a `full` sequent we can ensure any instantiation of the assumptions implies that there is an instantiation of the goal, and with a `partial` one we cannot. Of course, given the non-deterministic nature of the procedure, `partial` transformations can be useful and have different heuristic values, not discussed in this paper.

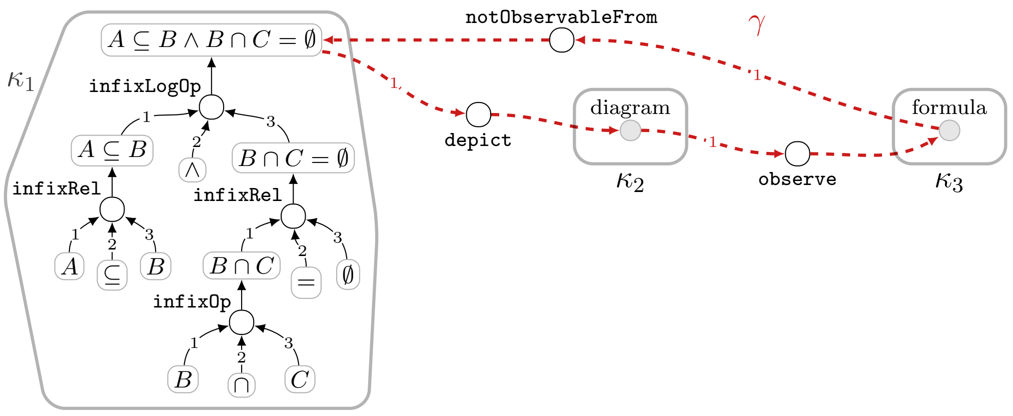
Algorithm 6 Structure transfer with target specified by σ

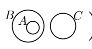
```
1: procedure STRUCTURETRANSFER($\sigma$, sq, $\mathbb{S}$, $\mathbb{Q}$)
2:     if VALIDSEQUENT(sq, $\mathbb{S}$, $\mathbb{Q}$) then
3:         return (sq, full)
4:     else
5:         find sc $\in \mathbb{S}$ and sq' s.t. sq' = APPLYTRANSFERSCHEMA($\sigma$, sc, sq, $\mathbb{Q}$)
6:             return STRUCTURETRANSFER($\sigma$, sq', $\mathbb{S}$, $\mathbb{Q}$)
7:             if none found return (sq, partial)
```

5.3 Structure transfer for the depict-and-observe process

We started this paper with an informal presentation of the depict-and-observe process, wherein we take a formula in Set Algebra, then we *depict* it in Euler Diagrams, and then we produce an *observation* from the diagram, which is itself a formula in Set Algebra. Now we shall see that this problem can be easily encoded and then solved with structure transfer.

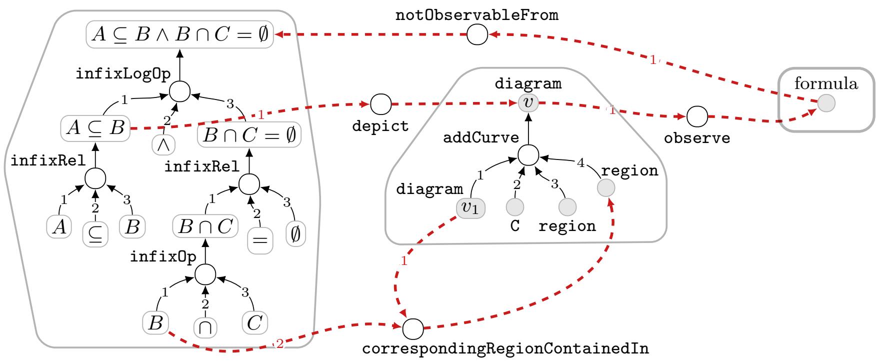
Problem encoding The multi-space we will work on is $(\mathcal{C}, \mathcal{D}, \mathcal{C}, \mathcal{G})$ where $\mathcal{C}$ encodes Set Algebra, $\mathcal{D}$ encodes Euler Diagrams and $\mathcal{G}$ encodes relations across and within the spaces. Given expression $A \subseteq B \wedge B \cap C = \emptyset$, we want to find a diagram that depicts it and a set expression that can be observed from the diagram. In other words, given the sequent $\langle \kappa_1, \kappa_2, \kappa_3, \alpha \Vdash \gamma \rangle$, visualised below, we want to find a $\{2,3\}$-reification of it that makes it valid. Note that we can even encode the relation of *not being observable from* to prevent a trivial observation.



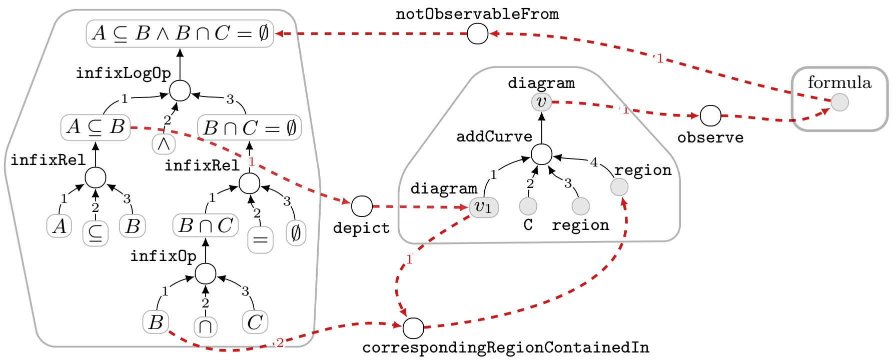
Solution approach The first step is to apply a sequence of $\{2,3\}$-transfer schemas[18], similar to those of examples 4.7 and 4.17 in order to reify κ_2 to a pattern graph, κ_2'. As we will see, this results in a construction of a diagram with type $\{\texttt{AB}, \texttt{B}, \texttt{C}, \emptyset\}$ (recall this is the type of ). Step-by-step, this involves first breaking down the goal of depicting $A \subseteq B \wedge B \cap C = \emptyset$ into two goals, one for each of the conjuncts, using the schema from Example 4.3. Then, we apply the schema from example 4.12 (as a $\{2,3\}$-transfer schema)

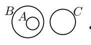
18. Note that for schemas to be applicable in this setting they need to be schemas for the space $(\mathcal{C}, \mathcal{D}, \mathcal{C}, \mathcal{G})$. We previously presented some relevant schemas for space $(\mathcal{C}, \mathcal{D}, \mathcal{G})$ which technically need to be *lifted* to $(\mathcal{C}, \mathcal{D}, \mathcal{C}, \mathcal{G})$. From a theoretical perspective, lifting schemas is a trivial operation, not discussed in this paper.

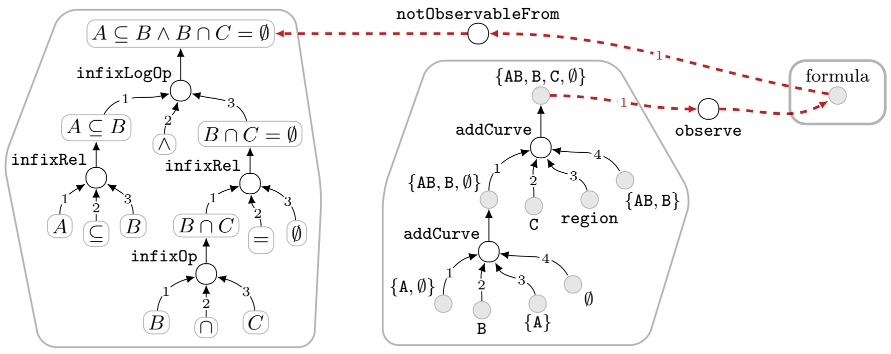
to reify κ_2 into a graph (unlabelled in the figure below) constrained by a goal which specifies disjointness[19]:



Next, we can reduce the goal stating that v must depict $A \subseteq B$ to one that states that v_1 depicts $A \subseteq B$, using a simple schema not presented here (included in Appendix C). This results in the following:

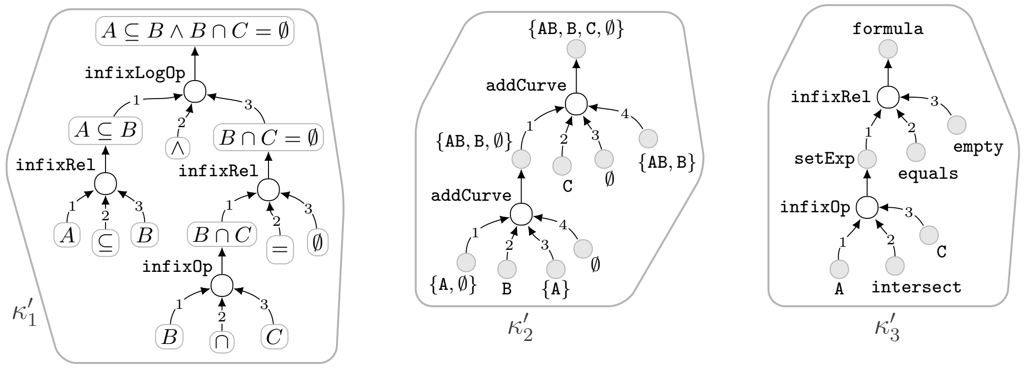


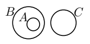
Then, an application of the schema from Example 4.14 adds constraints on the structure of v_1. Then, the goals for constructor `correspondingRegionContainedIn` can be satisfied by applications of the schema from Example 4.13, which effectively specialises the types of the Euler diagram pattern until v has type $\{\mathtt{AB}, \mathtt{B}, \mathtt{C}, \emptyset\}$, which we know is the type of . At this stage we have:



19. The graphs κ_1 and κ_3 remain label-isomorphic in this step.

The next step is to perform more applications to reify κ_3 in order to satisfy both remaining goals. There are many ways of doing this, but we can do it using schemas 4.11 and then 4.12 to obtain a pattern graph which constructs $A \cap B = \emptyset$. The goal `notObservableFrom` can be discharged with more applications of schemas included in Appendix C, noting that this can only be done if κ'_3 is distinct from $A \subseteq B$ nor $B \cap C = \emptyset$. The result is a sequent, $\langle \kappa'_1, \kappa'_2, \kappa'_3, \alpha' \Vdash \gamma' \rangle$, with empty α' and γ', which is clearly valid.



The pattern graphs κ'_2 and κ'_3 are precisely patterns whose instantiations in $\mathcal{D}$ and $\mathcal{C}$ (the construction spaces for EULER DIAGRAMS and SET ALGEBRA) construct  and, respectively, $A \cap C = \emptyset$. The pattern κ'_1 is label-isomorphic to our original κ_1, as we only performed $\{2, 3\}$-transfer schema applications.

5.4 Conclusion on structure transfer

Structure transfer allows us to transform a given representation by exploiting transfer schema applications. It starts with a sequent $\langle \kappa_1, \ldots, \kappa_n, \alpha \Vdash \gamma \rangle$ where $(\kappa_1, \ldots, \kappa_n)$ sets a structural context, α encodes some assumptions and γ encodes some goals. A target is specified by $\sigma \subseteq \{1, \ldots, n\}$, and an application of structure transfer given σ returns a sequent $\langle \kappa'_1, \ldots, \kappa'_n, \alpha' \Vdash \gamma' \rangle$ where the target specified by σ is our sought-after transformation of the source. In the `full` case, the resulting sequent is known to be valid, and in the `partial` case it may or may not be valid, so heuristics are needed to determine whether the transformation is useful for the given task. If the resulting sequent is valid it ensures the instantiatability of $(\kappa'_1, \ldots, \kappa'_n, \alpha' \cup \gamma')$ given an instantiation of $(\kappa'_1, \ldots, \kappa'_n, \alpha')$, which may not be known, but which we may try to solve as the instantiation problem presented in Section 4.1 simply by applying schemas to, say, the graph $\kappa'_1 \cup \cdots \cup \kappa'_n \cup \alpha'$. Although it may not seem so, the general instantiation problem has its own challenges that go beyond the methods in this paper. Specifically, a monotonicity condition needs to be satisfied for every schema application. While checking for monotonicity is trivial for any meta-space $\mathcal{G}$ that encodes a typical monotonic logic, if the space where we are trying to determine instantiatability is a grammar like SET ALGEBRA, where even simple instantiation schemas are not monotonic (see Example 4.4), then monotonicity needs to be determined case-by-case. This is a research and implementation challenge for the future. In this paper we will not address directly the general case for checking the instantiability of arbitrary pattern graphs in multi-spaces.

We have an implementation of the core notions of RST and structure transfer, called Oruga (Raggi, Stapleton, Jamnik, Stockdill, Garcia, & Cheng, 2022; Raggi, 2022). The implementation of Oruga assumes that any meta-space encoded is monotonic, and it does not deal with the instantiation problem that comes after structure transfer. The search space is navigated with heuristics and parameters that can be changed according to the task, but there is still much work left to enable multiple strategies. The main challenges for this and any implementations are:

1. strategies for navigating the structure transfer search space,
2. strategies for inference about monotonicity, and more generally for determining instantiatability,
3. a language for expressing and declaring schemas. For instance, if we want to express families of schemas, such as those presented in Examples 4.2, 4.8, 4.9, 4.10, 4.12 and 4.14 we need a powerful (meta-)type theory.

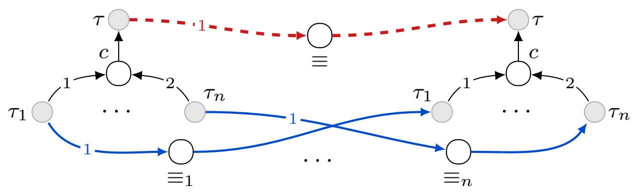
6. Related concepts and applications

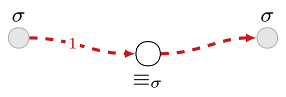
Representational Systems Theory provides new foundations for thinking about structures that are familiar to logicians, computer scientists, linguists and cognitive scientists. In particular, we claim that constructions in construction spaces generalise syntax trees, allowing for some atypical (but useful) structures. Now, one of the first things that we must do when providing a new and exciting structure that *generalises* over something familiar, is to show that the good-old familiar techniques are still available in the new structures.

Sections 3 and 4.3 provide evidence that RST can be used for one of its main motivating problems, which is that of producing transformations across representational systems. Now, it happens to be that structure transfer, when restricted, looks like some known techniques in formal methods.

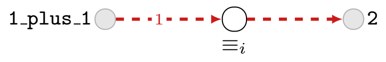
6.1 Formal methods: rewriting, abstraction and refinement

One technique generalised by structure transfer is *term rewriting*. In formal theories with equality, rewriting is defined as the process by which, given $t_1 = t_2$, we can replace t_1 for t_2 within a term. For this, it is not difficult to define the notion of *reflexive* schemas. Given some types $\tau, \tau_1, \ldots, \tau_n$ and notions of equivalence, $\equiv, \equiv_1, \ldots, \equiv_n$, for each of them, captured in a meta-space, we (can) introduce schemas of the form:





for every constructor, c, and every *minimal* type, σ. Of course, this requires the type system to have such minimal types. Moreover, such a set of schemas can be enriched with knowledge about the behaviour of the equivalence relations, such as:

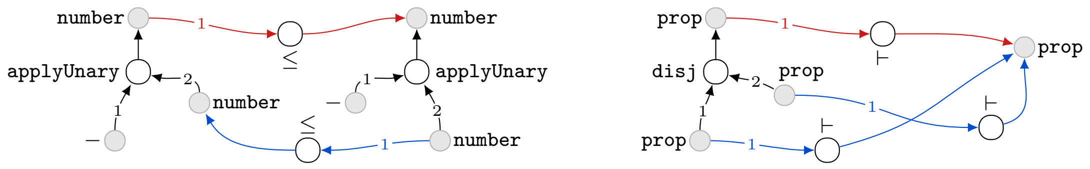


A set of reflexive schemas with some additional ones like this one above results in a *rewrite system*. Given a sequent of the form $\langle \kappa_1, \kappa_2, \alpha \Vdash \gamma \rangle$ where κ_1 encodes, say $2(1+1) \leq x$, and

κ_2 contains a single token t_2, with empty α and γ encoding the equivalence of $2(1+1) \leq x$ and t_2, structure transfer can give us a $\{2\}$-reification yielding κ_2' which encodes $2(2) \leq x$.

6.1.1 Abstraction and refinement

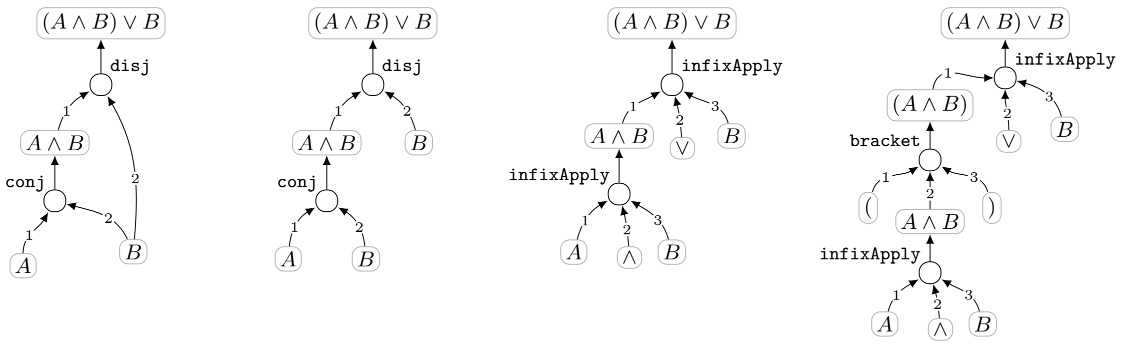
Structure transfer can handle reasoning by substitution with ad-hoc equalities, but it can handle what Coen (2004) calls *sub-equational rewriting*, which refers to rewriting where terms are replaced for non equal ones but where a transitive relation holds (e.g., entailment $\vdash$, or the order of integers $\leq$).



The result of sub-equational rewriting is not an equivalent term, but one that stands in a desired relation with the source term.

And finally, it should be clear that the relations encoded in $\mathcal{I}$ need not be transitive, so structure transfer is able to satisfy arbitrary relations, which is similar to the *transfer* tactic introduced by Huffman and Kunčar (2013), in Isabelle/HOL. Their *transfer rules* are analogous to transfer schemas, but of course, the former is limited to terms and relations in HOL, with some additional limitations on the structure of the source and target terms. The mechanism of the transfer tactic, and similar tools (e.g., Cohen, Dénès, and Mörtberg (2013)) are used for data *abstraction* and *refinement* (Tabareau, Tanter, & Sozeau, 2018; Delaware, Pit-Claudel, Gross, & Chlipala, 2015; Abrial, Butler, Hallerstede, Hoang, Mehta, & Voisin, 2010). Abstract data-types are useful for human-level specification, reasoning and verification, and more refined data-types are necessary at the implementation-level for computation. Formal links between various levels of abstraction are necessary and widely used in formal verification. For example, sets are useful for specifying programs, but their specific implementation can vary. For instance, a very simple refinement of a set is a list, and a translation of various expressions between these data-types is possible. For example, it is not hard to see that with an appropriate set of transfer schemas, we can derive that $(\{1\} \cup (\texttt{set\_of L})) \cup (\{4\} \cup \emptyset)$ is the set of list `append` (`insert` 1 (`rev L`)) (`insert` 4 []).

The concepts of abstraction and refinement have applications beyond formal verification. For example, within RST, when trying to develop or specify a representational system, we have modelling choices, such as whether to include parentheses as tokens, or whether to model some operation directly as a constructor or to model it as a token. Decisions concerning how much to abstract or refine a model usually depend on the purpose of the model. Each model may have its benefits and drawbacks depending on its use-case. For example, if we needed to assess a representation based on how a novice may interact with it, or if we simply need to know how much ink is needed to print it, we may need to capture the symbols at a high granularity. However, if we want a compact way of capturing the semantics, a low granularity may be more desirable. If this modelling variability is expected, it is imperative that tools for translating between them can readily be created. Suffice it to say, we can use structure transfer for this. Below, the proposition $(A \wedge B) \vee B$ is modelled in four ways, ordered in decreasing level of abstraction. The left-most model does not even distinguish between two distinct tokens of the same type, while the right-most model captures often-ignored tokens such as parentheses.



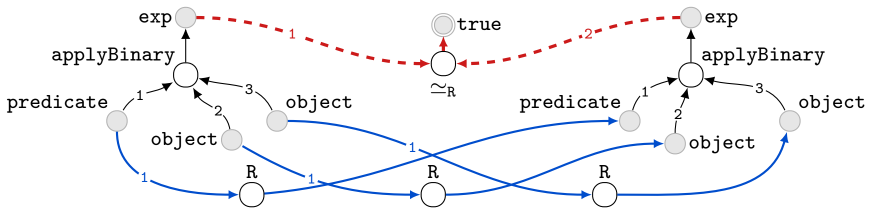
It is not difficult to see that we can define transfer schemas between three such construction spaces, so that we can exploit transformations between them.

6.2 Analogy: structure-mapping and anti-unification

Finally, we turn our attention towards mechanisms more philosophically similar, than formally similar. Structure transfer, as outlined here, has clear parallels to analogy, specifically *structure-mapping* by Gentner (1983), Falkenhainer, Forbus, and Gentner (1989) and related approaches, such as *anti-unification* by Krumnack, Schwering, Gust, and Kühnberger (2007), Schmidt, Krumnack, Gust, and Kühnberger (2014). The most important distinction between structure-mapping and structure transfer is that the latter is a procedure for *applying* a known mapping rather than *discovering* it. Here we explore briefly how analogy can be modelled in terms of schemas, and further we will suggest how our tools could be used for discovering analogies.

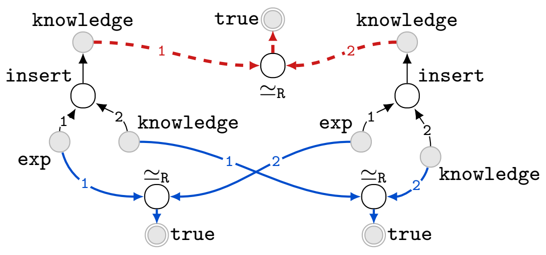
The core idea of structure-mapping is that an analogy between two domains is characterised by similarity at the level of relations between objects rather than at the level of the objects themselves or their attributes (i.e., unary properties). Thus, it is about how a seemingly arbitrary mapping at the object level (e.g., electron $\mapsto$ planet, atom's nucleus $\mapsto$ sun) is meaningful because it preserves some relations between the objects (e.g., the atom's nucleus and the electron attract each other like the sun and a planet attract each other). Moreover, Gentner's *systematicity principle* favours finding coherent *systems* of relationships, not just one-off relationships.

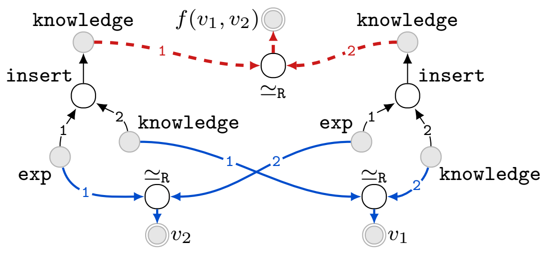
It is easy to see that invariants across domains, as those that characterise an analogy according to Gentner can be encoded with some simple schemas, such as the one drawn below, which simply captures the rule: *two statements are analogous modulo* $\mathtt{R}$ *(denoted* $\simeq_{\mathtt{R}}$*) if their arguments are related by* $\mathtt{R}$. If $\mathtt{R}$ maps `nucleus` to `sun` and `electron` to `planet` then the statements `attracts`(`nucleus`, `electron`) and `attracts`(`sun`, `planet`) are analogous modulo $\mathtt{R}$. The generalisation of this rule can be captured as a schema, drawn below:



This is not dissimilar to the idea that two statements are analogous if they can be anti-unified. Structure-mapping (Gentner, 1983) requires that knowledge for both domains be encoded with identical relations, such as *attracts*, while higher order versions (Krumnack et al., 2007; Schmidt et al., 2014) can map the predicates, as the schema above does.

To capture that two knowledge bases are analogous we might use schemas such as the one below (left), stating that two knowledge bases are analogous if their individual statements are analogous, or a *fuzzy* version (right) wherein the truth value (*strength*) of the analogy is a function of the values of the arguments:





Other rules, such as '$t_1 = t_2$ implies $t_1 \simeq_R t_2$' may also be encoded.

In this context, the question that structure-mapping aims to solve, given a pair of knowledge bases t_1 and t_2, is whether we can find a mapping R that ensures $t_1 \simeq_R t_2$ (or has maximal value in the fuzzy case). Our proposal for approaching this problem using the notions presented in this paper is to use schemas abductively: that is, encode knowledge bases κ_1 and κ_2, noting that each knowledge base can be constructed in many ways. Try to show that $\langle \kappa_1, \kappa_2, \alpha \Vdash \gamma \rangle$ is valid using Algorithm 4, for some γ that encodes the relation $\simeq_R$ between the pair of knowledge bases, for some variable R. Without knowing R we will not be able to prove validity, but the remaining goals, γ, may be used to inform how a good R may look like. In the fuzzy case we may try to find an R that maximises the truth value of the analogy.

A consequence of using this approach for finding an analogy is that once we have discovered a mapping R we can use structure transfer to transform arbitrary statements from the language of one knowledge base to the other, regardless of whether the statement or its resulting transformation, lives in the knowledge base. Thus, structure transfer can be used as a mechanism for the actual *transfer* of knowledge that happens once a mapping is grasped (Gick & Holyoak, 1983).

6.3 Last remarks on the applications of structure transfer

We have shown how structure transfer generalises various known procedures. Of course, all our claims of generality are in terms of computability and not complexity or efficiency. Our notion of structure transfer is defined by a search space, which may be infinite, and we do not provide explicit strategies or heuristics for traversing the space.

The fact that structure transfer is, in principle, a really powerful tool depending on the setting in which it is used calls for the *tactification* of it, wherein depending on the task, we can use a specific version of it. For example, for certain scenarios we might consider sets of transfer schemas where all elements are reflexive except for one (for standard rewriting), for others we might consider quasi-reflexive ones (for analogy). Or we may conceive of decision procedures where structure transfer is applied iteratively, modifying the set of transfer or inference schemas.

7. Conclusion

Raggi et al. (2023) introduced Representational Systems Theory motivated by the prospect of understanding the structure of representations in order to facilitate their analysis, use and transformations. One of the main innovations of RST was the notion of a construction space, where the structure of representations is captured through a graph-theoretic generalisation of syntax trees, along with an unassuming type structure.

All the content of this paper builds on the foundation provided by the concept of a construction space. From the abstract nature of this concept we get representational generality. We extended the theory to include the notions of schemas. As we showed, schemas can be applied like logical rules to determine the validity of sequents, and in their transfer version, to generate structure in some target spaces, to satisfy some specified constraints. This is the key of structure transfer. Crucially, this means that we can use knowledge about the preservation of information across construction spaces to produce re-representations through *any* relation about which we have some knowledge.

We showed that structure transfer has a wide range of applications for producing transformations across systems – which is valuable given the heterogeneous nature of our symbolic systems. In particular, we showed how it can be used to produce diagrams from sentences, and sentential observations from diagrams, it can be used for data abstraction and refinement, and for modelling and enacting analogies. Moreover, the procedures presented in this paper are based on an extensible knowledge base, as opposed to hard-coded transformations, and the knowledge can be encoded in any variety of logics expressible as multi-spaces with the use of pattern graphs.

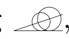
Our vision is that the ideas in this paper are used to build tools that not only *manage* heterogeneity but *exploit* it. Exploiting the intrinsic heterogeneity of our symbolic systems is a huge and important challenge for science and communication, and we have provided some foundations and methods to do this.

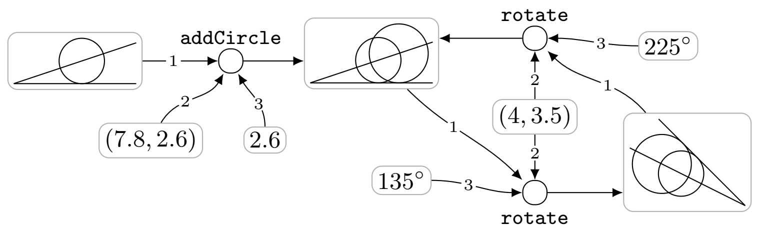
Acknowledgments

We thank Atsushi Shimojima and Jean Flower for their helpful comments. The research is funded by the UK EPSRC, grant numbers EP/R030642/1, EP/T019603/1, EP/T019034/1 and EP/R030650/1.

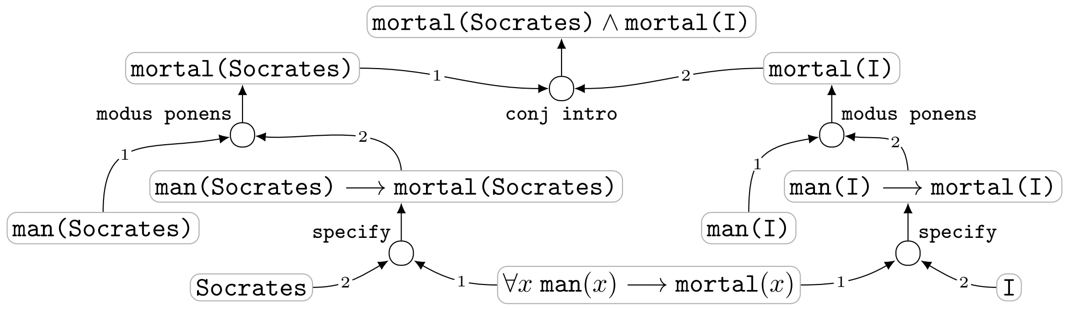
Appendix A. Representational Systems Theory

The next two examples – on geometric constructions and proofs – complement that on matrix algebra in Section 3, demonstrating what can be modelled with construction spaces.

Example A.1 (Geometric Constructions)**.** Constructors can encode geometric operations, with the tokens encoding points (e.g. $(7.8, 2.6)$), magnitudes (e.g. 2.6), and geometric figures (e.g. circles). The graph below shows two ways of constructing , with two rotations resulting in a cycle. Note that the tokens that represent points (e.g. $(7.8, 2.6)$), magnitudes (e.g. 2.6), and angles (e.g., $135°$), are not themselves graphical tokens but necessary pieces of information to build the constructors' outputs.

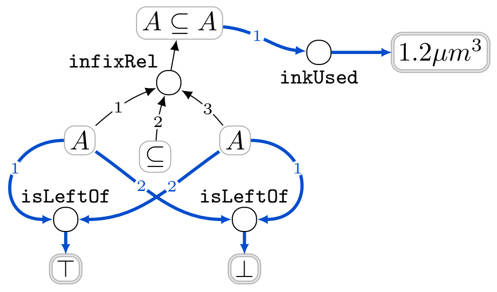


Example A.2 (Proofs/Arguments)**.** We can use constructors to encode inference rules, such as modus ponens, conjunction introduction, or universal specification. Given a constructor vertex, its output is the conclusion of the input premises given the inference rule:



The next example focuses on using a construction system to encode low-level properties of tokens using a meta-space.

Example A.3 (Modelling low-level properties of symbols)**.** Consider the construction space, $\mathcal{C}$, for Set Algebra. An meta-space, $\mathcal{I}$, for $\mathcal{C}$ can be defined by adding two types of meta-tokens: a boolean one (tokens $\top$ and $\bot$ of types `true` and `false`) for determining truth, and a numerical one for determining quantities. The meta-constructor `isLeftOf` captures whether one token is to the left of the other, and `inkUsed` captures the amount of ink, in cubic micrometers, used to print the token. Here the graph with black arrows belongs to $\mathcal{C}$ and the graph with thick blue arrows belongs to $\mathcal{I}$.



This example highlights that the meta-space may encode information about tokens not available at the type-system level. For example, both occurrences of A in $A \subseteq A$ might be indistinguishable at the type level, yet we can identify different properties of them.

Appendix B. Sequents and Schemas

We restate and prove lemma 4.3.

Lemma B.1. *Let $\langle \kappa_1, \ldots, \kappa_n, \alpha \Vdash \gamma \rangle$ be a sequent for multi-space system $\mathcal{M}$ and let $\langle \kappa'_1, \ldots, \kappa'_n, \alpha' \Vdash \gamma' \rangle$ be a weakening of $\langle \kappa_1, \ldots, \kappa_n, \alpha \Vdash \gamma \rangle$ with map r. If $\langle \kappa_1, \ldots, \kappa_n, \alpha \Vdash \gamma \rangle$ is a schema for $\mathcal{M}$ then so is $\langle r[\kappa_1], \ldots, r[\kappa_n], r[\alpha] \Vdash r[\gamma] \rangle$.*

Proof. Assume that $\langle \kappa_1, ..., \kappa_n, \alpha \Vdash \gamma \rangle$ is a schema for $\mathcal{M}$ and let $s\colon (r[\kappa_1], ..., r[\kappa_n], r[\alpha]) \to (\kappa''_1, ..., \kappa''_n, \alpha'')$ be an instantiatable specialisation function. We show that we can extend s to map $r[\gamma]$ to some instantiatable specialisation in the meta-space, $\mathcal{G}$. To simplify notation, we start by defining r_a and r_g to be the functions $r|_{(\kappa_1,...,\kappa_n,\alpha)}\colon (\kappa_1, ..., \kappa_n, \alpha) \to (r[\kappa_1], ..., r[\kappa_n], r[\alpha])$ and, resp., $r|_\gamma\colon \gamma \to r[\gamma]$. We have it that r_a is a specialisation of $(\kappa_1, ..., \kappa_n, \alpha)$ in $\mathcal{M}$. Therefore, it must be that $s \circ r_a\colon (\kappa_1, ..., \kappa_n, \alpha) \to (\kappa''_1, ..., \kappa''_n, \alpha'')$ is an instantiatable specialisation function. Given that $\langle \kappa_1, ..., \kappa_n, \alpha \Vdash \gamma \rangle$ is a schema, there exists an instantiatable specialisation function $s'\colon (\kappa_1, ..., \kappa_n, \alpha \cup \gamma) \to (\kappa''_1, ..., \kappa''_n, \alpha'' \cup \gamma'')$, in $\mathcal{M}$, that extends $s \circ r_a$. Setting s'' to be the restriction of s' to domain $r_g^{-1}[\gamma']$, we have $s''\colon r_g^{-1}[\gamma'] \to \gamma'''$, where γ''' is the subgraph of γ'' that ensures s'' is surjective. We show that

$$s \cup (s'' \circ r_g^{-1})\colon (r[\kappa_1], ..., r[\kappa_n], r[\alpha_n] \cup r[\gamma']) \to (\kappa''_1, ..., \kappa''_n, \alpha'' \cup \gamma''')$$

is an instantiatable specialisation function that extends s. Firstly, by construction, $s \cup (s'' \circ r_g^{-1})$ is a function. Clearly the instantiatablilty of $(\kappa''_1, ..., \kappa''_{n-1}, \alpha'' \cup \gamma'')$ in $\mathcal{M}$ implies the instantiatablility of $(\kappa''_1, ..., \kappa''_n, \alpha'' \cup \gamma''')$ in $\mathcal{M}$. We already have it, by assumption, that s is a specialisation in $\mathcal{M}$. What remains is to show that $s'' \circ r_\gamma^{-1}\colon r[\gamma] \to \gamma'''$ is a specialisation in $\mathcal{G}$. Well, the function r_g is a generalisation in $\mathcal{G}$, so its inverse, r_g^{-1}, is a specialisation in $\mathcal{G}$. In addition, s'' is a restriction of a specialisation in $\mathcal{G}$ and, thus, s'' is itself a specialisation in $\mathcal{G}$. Trivially, then, $s'' \circ r_g^{-1}\colon r[\gamma] \to \gamma'''$ is a specialisation in $\mathcal{G}$, as required. Therefore, $s \cup (s'' \circ r_g^{-1})$ is an instantiatable specialisation that extends s. Hence $\langle r[\kappa_1], ..., r[\kappa_n], r[\alpha] \Vdash r[\gamma] \rangle$ is a schema. □

We restate and prove theorem 4.1

Theorem B.1. *Let $\langle \kappa_1, ..., \kappa_n, \alpha \Vdash \gamma \rangle$ be a schema for multi-space system $\mathcal{M}$ with refinement $\langle \kappa'_1, ..., \kappa'_n \alpha' \Vdash \gamma' \rangle$. Then $\langle \kappa'_1, ..., \kappa'_n \alpha' \Vdash \gamma' \rangle$ is a schema for $\mathcal{M}$.*

Proof. Given that $\langle \kappa_1', ..., \kappa_n', \alpha' \Vdash \gamma' \rangle$ is a refinement of $\langle \kappa_1, ..., \kappa_n, \alpha \Vdash \gamma \rangle$, we know that there exists a refinement map, $r\colon (\kappa_1, ..., \kappa_n, \alpha \cup \gamma) \to (\kappa_1', ..., \kappa_n, \alpha' \cup \gamma')$. By lemma 4.3, we know that $\langle r[\kappa_1], ..., r[\kappa_n], r[\alpha] \Vdash r[\gamma] \rangle$ is a schema. By definition 4.5, $\langle r[\kappa_1], ..., r[\kappa_n], r[\alpha] \Vdash r[\gamma] \rangle$ is monotonic for $(\kappa_1', ..., \kappa_n', \alpha')$. Therefore $\langle \kappa_1', ..., \kappa_n', \alpha' \Vdash r[\gamma] \rangle$ is a schema. This implies that for any instantiatable specialisation, $s\colon (\kappa_1', ..., \kappa_n', \alpha') \to (\kappa_1'', ..., \kappa_n'', \alpha'')$, there exists an instantiatable specialisation, say $s'\colon (\kappa_1', ..., \kappa_n', \alpha' \cup r[\gamma]) \to (\kappa_1'', ..., \kappa_n'', \alpha'' \cup \gamma'')$, that extends s. By definition 4.5, $\gamma' \subseteq \alpha' \cup r[\gamma]$, so s' induces an instantiatable specialisation of $(\kappa_1', ..., \kappa_n', \alpha' \cup \gamma')$. Hence $\langle \kappa_1', ..., \kappa_n', \alpha' \Vdash \gamma' \rangle$ is a schema for $\mathcal{M}$. □

We prove two lemmas that immediately entail theorem 4.2.

Lemma B.2. *Let $\langle \kappa_1, ..., \kappa_n, \alpha \Vdash \gamma \rangle$ be a schema and $\langle \kappa_1', ..., \kappa_n', \alpha' \Vdash \gamma' \rangle$ be a sequent for multi-space system $\mathcal{M}$. If $\langle \kappa_1', ..., \kappa_n', \alpha' \cup \delta \Vdash \gamma' \rangle$ is a δ-forward application of $\langle \kappa_1, ..., \kappa_n, \alpha \Vdash \gamma \rangle$ to $\langle \kappa_1', ..., \kappa_n', \alpha' \Vdash \gamma' \rangle$ and a schema for $\mathcal{M}$ then $\langle \kappa_1', ..., \kappa_n', \alpha' \Vdash \gamma' \rangle$ is also a schema for $\mathcal{M}$.*

Proof. Suppose that $s\colon (\kappa_1', ..., \kappa_n', \alpha') \to (\kappa_1'', ..., \kappa_n'', \alpha'')$ is an instantiatable specialisation function. We must show that there exists an instantiatable specialisation function, $s'\colon (\kappa_1', ..., \kappa_n', \alpha' \cup \gamma') \to (\kappa_1'', ..., \kappa_n'', \alpha'' \cup \gamma'')$, that extends s. By theorem 4.1, we know that $\langle \kappa_1', ..., \kappa_n', \alpha' \Vdash \delta \rangle$ is a schema because it is a refinement of $\langle \kappa_1, ..., \kappa_n, \alpha \Vdash \gamma \rangle$. Therefore, there exists an instantiatable specialisation function, $s''\colon (\kappa_1', ..., \kappa_n', \alpha' \cup \delta) \to (\kappa_1'', ..., \kappa_n'', \alpha'' \cup \delta')$, that extends s. It is given that $\langle \kappa_1', ..., \kappa_n', \alpha' \cup \delta \Vdash \gamma' \rangle$ is also a schema, so there must also exist an extension of s'' to some instantiatable specialisation function, say $s'''\colon (\kappa_1', ..., \kappa_n', \alpha' \cup \delta \cup \gamma') \to (\kappa_1'', ..., \kappa_n'', \alpha'' \cup \delta'' \cup \gamma''')$. Restricting the domain of s''' to $(\kappa_1', ..., \kappa_n', \alpha' \cup \gamma')$ yields an instantiatable specialisation function that extends s, as required. Hence $\langle \kappa_1', ..., \kappa_n', \alpha' \Vdash \gamma' \rangle$ is a schema. □

Lemma B.3. *Let $\langle \kappa_1, ..., \kappa_n, \alpha \Vdash \gamma \rangle$ be a schema and $\langle \kappa_1', ..., \kappa_n', \alpha' \Vdash \gamma' \rangle$ be a sequent for multi-space system $\mathcal{M}$. If $\langle \kappa_1', ..., \kappa_n', \alpha' \Vdash \delta \rangle$ is a δ-backward application of $\langle \kappa_1, ..., \kappa_n, \alpha' \Vdash \gamma \rangle$ to $\langle \kappa_1', ..., \kappa_n', \alpha' \Vdash \gamma' \rangle$ and a schema for $\mathcal{M}$ then $\langle \kappa_1', ..., \kappa_n', \alpha' \Vdash \gamma' \rangle$ is also a schema for $\mathcal{M}$.*

Proof. Suppose that $s\colon (\kappa_1', ..., \kappa_n', \alpha') \to (\kappa_1'', ..., \kappa_n'', \alpha'')$ is an instantiatable specialisation function. We must show that there exists an instantiatable specialisation function, $s'\colon (\kappa_1', ..., \kappa_n', \alpha' \cup \gamma') \to (\kappa_1'', ..., \kappa_n'', \alpha'' \cup \gamma')$, that extends s. It is given that $\langle \kappa_1', ..., \kappa_n', \alpha' \Vdash \delta \rangle$ is a schema, so there exists an extension of s to some instantiatable specialisation function, $s''\colon (\kappa_1', ..., \kappa_n', \alpha' \cup \delta) \to (\kappa_1'', ..., \kappa_n'', \alpha'' \cup \delta')$. By theorem 4.1, we know that $\langle \kappa_1', ..., \kappa_n, \alpha' \cup \delta \Vdash \gamma' \rangle$ is a schema because it is a refinement of $\langle \kappa_1, ..., \kappa_n, \alpha \Vdash \gamma \rangle$. Therefore, we can extend s'' to an instantiatable specialisation function, $s'''\colon (\kappa_1', ..., \kappa_n', \alpha' \cup \delta \cup \gamma') \to (\kappa_1'', ..., \kappa_n'', \alpha'' \cup \delta' \cup \gamma'')$. Notably, since the domain, $(\kappa_1', ..., \kappa_n', \alpha' \cup \delta \cup \gamma')$ of s''' can be restricted to $(\kappa_1', ..., \kappa_n', \alpha' \cup \gamma')$, we have the existence of an extension of s to an instantiatable specialisation, $s'\colon (\kappa_1', ..., \kappa_n', \alpha' \cup \gamma') \to (\kappa_1'', ..., \kappa_n'', \alpha'' \cup \gamma''')$, for some γ'''. Hence $\langle \kappa_1', ..., \kappa_n', \alpha' \Vdash \gamma' \rangle$ is a schema. □

Theorem 4.2, restated below, is a corollary of the above two lemmas.

Theorem B.2. *Let $\langle \kappa_1, \ldots, \kappa_n, \alpha \Vdash \gamma \rangle$ be a schema and $\langle \kappa_1', \ldots, \kappa_n', \alpha' \Vdash \gamma' \rangle$ be a sequent for multi-space system $\mathcal{M}$. If $\langle \kappa_1', \ldots, \kappa_n', \alpha'' \Vdash \gamma'' \rangle$ is an application of $\langle \kappa_1, \ldots, \kappa_n, \alpha \Vdash \gamma \rangle$ to $\langle \kappa_1', \ldots, \kappa_n', \alpha' \Vdash \gamma' \rangle$ and a schema for $\mathcal{M}$ then $\langle \kappa_1', \ldots, \kappa_n', \alpha' \Vdash \gamma' \rangle$ is also a schema for $\mathcal{M}$.*

Proof. A schema application is either a forward or a backward application, respectively covered by lemmas B.2 and B.3. □

We now restate and prove theorem 4.3.

Theorem B.3. *Let $\mathbb{S}$ be a set of schemas and let $\langle \kappa_1', ..., \kappa_n', \alpha' \Vdash \gamma' \rangle$ be a sequent for multi-space system $\mathcal{M}$. If $\langle \kappa_1', ..., \kappa_n', \alpha' \Vdash \gamma' \rangle$ is valid over $\mathbb{S}$ then it is also a schema for $\mathcal{M}$.*

Proof. The proof is by induction, over the depth of the recursion used to establish the validity of $\langle \kappa_1', ..., \kappa_n', \alpha' \Vdash \gamma' \rangle$. The base case, where $\langle \kappa_1', ..., \kappa_n', \alpha' \Vdash \gamma' \rangle \in \mathbb{S}$, is trivial since $\mathbb{S}$ is a set of schemas. Assume, for any sequent, $\langle \kappa_1', ..., \kappa_n', \alpha' \Vdash \gamma' \rangle$, valid over $\mathbb{S}$ at depth k that $\langle \kappa_1', ..., \kappa_n', \alpha' \Vdash \gamma' \rangle$ is a schema. Let $\langle \kappa_1', ..., \kappa_n', \alpha' \Vdash \gamma' \rangle$ be a valid sequent at depth $k+1$. We must show that $\langle \kappa_1', ..., \kappa_n', \alpha' \Vdash \gamma' \rangle$ is a schema. Since $\langle \kappa_1', ..., \kappa_n', \alpha' \Vdash \gamma' \rangle$ is valid, there exists $\langle \kappa_1, ..., \kappa_n, \alpha \Vdash \gamma \rangle \in \mathbb{S}$ and a sequent, $\langle \kappa_1', ..., \kappa_n', \alpha'' \Vdash \gamma'' \rangle$, for $\mathcal{M}$ such that

1. $\langle \kappa_1', ..., \kappa_n', \alpha'' \Vdash \gamma'' \rangle$ is valid at depth k, and
2. $\langle \kappa_1', ..., \kappa_n', \alpha'' \Vdash \gamma'' \rangle$ is an application of $\langle \kappa_1, ..., \kappa_n, \alpha \Vdash \gamma \rangle$ to $\langle \kappa_1', ..., \kappa_n', \alpha' \Vdash \gamma' \rangle$.

By the inductive assumption, $\langle \kappa_1', ..., \kappa_n', \alpha'' \Vdash \gamma'' \rangle$ is a schema. Thus, by theorem 4.2, we deduce that $\langle \kappa_1', ..., \kappa_n', \alpha' \Vdash \gamma' \rangle$ is a schema for $\mathcal{M}$, as required. □

Example B.1. In Figure 2, on the left, we show 3 schemas, π, ρ, and σ, for Set Algebra. Schema π encodes the fact that any type we *input* into constructor `infixOp` leads to a potential instantiation of the output in a manner that is consistent with the types of the inputs. Schema ρ, with empty antecedent, encodes the fact that type `B` can be instantiated – that is, that there exist a token, B, of type `B`. Finally, schema σ encodes the fact that the `union` type can be instantiated (with a token of the form $\cup$).

Suppose we want to know whether a pattern graph, γ, as shown in the pattern graph (centre top), can be instantiated. We represent this task with a sequent $\langle \alpha \Vdash \gamma \rangle$ where α is the empty graph and γ is the pattern graph we in question. Using schemas A, B and C we can show that γ is instantiatble by applying schemas in a backward manner, in forward manner, or a combination of the two. Below we show one particular sequence of schema applications that works. The first step involves applying A in a backward manner. remove the graph of γ corresponding to γ and add back the part corresponding to α. Notice that, in the end, we reach a sequent where α contains γ, showing that the starting sequent is a valid instantiation sequent. As we will show, this means γ is instantiatable.

Example B.2. In Figure 2, on the left, we show 3 schemas, π, ρ and σ, for Set Algebra, two of which – ρ and σ – have an empty antecedent graph. Schema π, $\langle \pi_\alpha \Vdash \pi_\gamma \rangle$, encodes the fact that any tokens, drawn from Set Algebra, of the three *input* types of the constructor `infixOp` leads to an instantiation of the output. Schema ρ, which is $\langle \ \Vdash \rho_\gamma \rangle$, encodes the fact that type `B` can be instantiated – that is, that there exist a token, B, of type `B`. Finally, schema σ, which is $\langle \ \Vdash \sigma_\gamma \rangle$, encodes the fact that the `union` type can be instantiated (with a token of the form $\cup$).

Suppose we want to know whether a pattern graph, such as γ_1 shown centre top, can be instantiated. We represent this task with a sequent $\langle \kappa_1 \Vdash \gamma_1 \rangle$ where κ_1 is the empty graph and γ_1 is the pattern graph we in question. Using schemas π, ρ and σ we can show

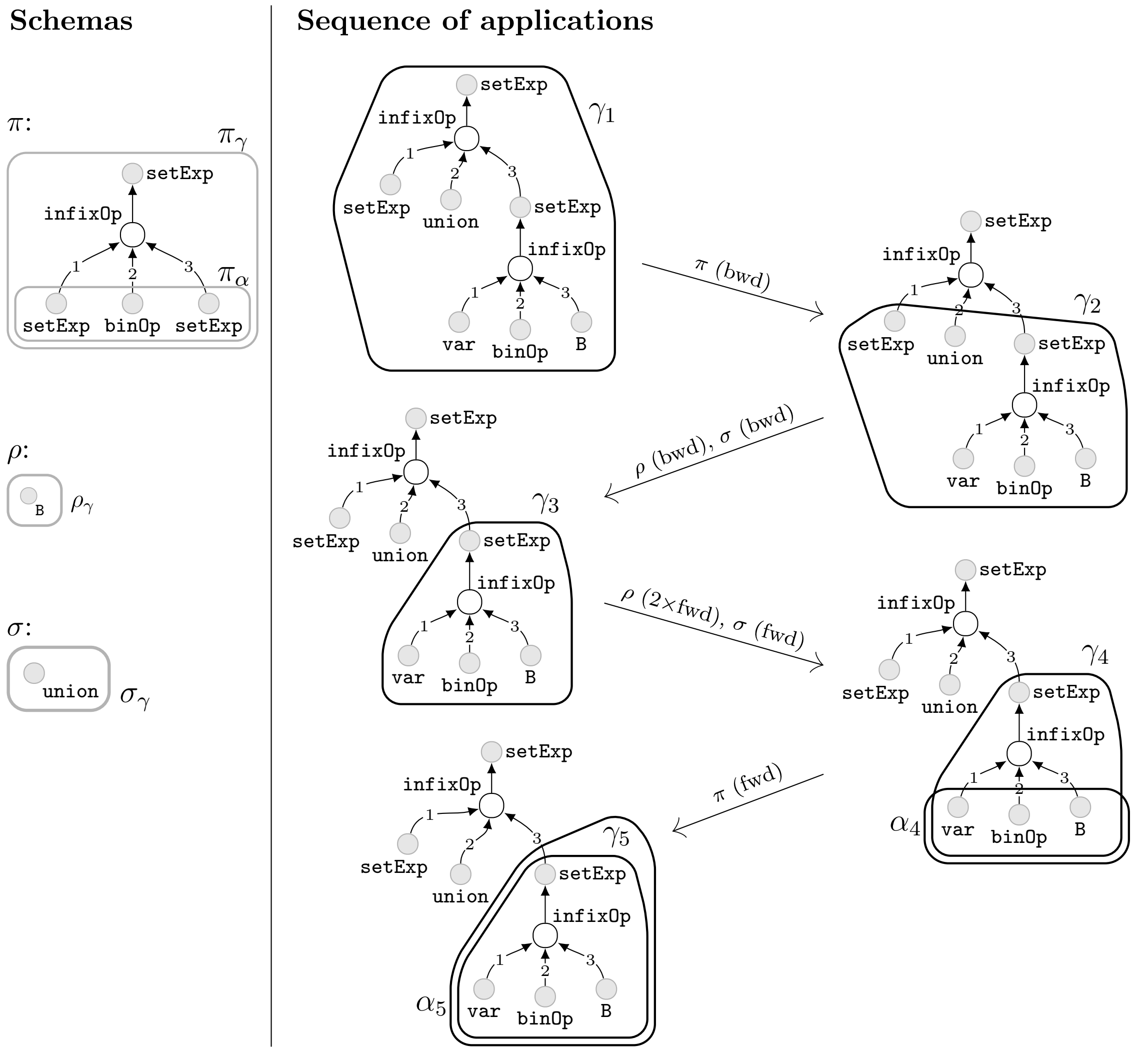


Figure 2: A sequence of schema applications

that γ_1 is instantiatable by applying them in a backward manner, in forward manner, or a combination of the two.

Below we show one particular sequence of applications. The first step involves applying π in a backward manner. We start with the weakening of $\langle \pi_\alpha \Vdash \pi_\gamma \rangle$ to $\langle \alpha^\pi \Vdash \gamma_1 \rangle$, where some map, f, maps π_γ to the top part of γ_1; this gives $\gamma_1 \backslash f[\pi_\gamma]$ as the γ_1-extender, highlighted in the figure. We then have a backward application of $\langle \pi_\alpha \Vdash \pi_\gamma \rangle$ to $\langle \ \Vdash \gamma_1 \rangle$ being $\langle \ \Vdash (\gamma_1 \backslash f[\pi_\gamma]) \cup \alpha^\pi \rangle$, which is the next sequent in the diagram, namely $\langle \ \Vdash \gamma_2 \rangle$. Essentially, this backwards application removed – from γ_1 – the subgraph of γ_1 it that was mapped to by π_γ and added back the part, α^π, that was mapped to by π_α. Notice that, after all seven applications, we reach the sequent $\langle \kappa_5 \Vdash \gamma_5 \rangle$, where the assumption, κ_5 is a subgraph of γ_5 (in fact, they are equal). Trivially, this sequent is a schema. Theorem 4.3 allows us to deduce that the original sequent, $\langle \Vdash \gamma_1 \rangle$, is also a schema. Therefore, it must be that γ_1 can be instantiated, since the empty graph is itself instantiatable.

We restate and prove Theorem 4.6.

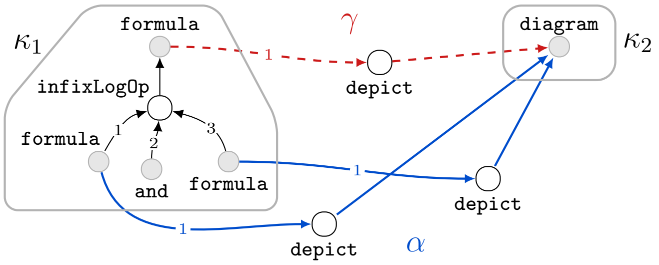
Theorem B.4. *Let $\langle \kappa_1, ..., \kappa_n, \alpha \Vdash \gamma \rangle$ be a sequent for $\mathcal{M}$, let $\sigma \subseteq \{1, \ldots, n\}$, and let $\mathbb{T}$ be a set of σ-transfer schemas. Assume we apply these schemas sequentially, starting with $\langle \kappa_1, ..., \kappa_n, \alpha \Vdash \gamma \rangle$ and ending with $\langle \kappa_1', ..., \kappa_n', \alpha' \Vdash \gamma' \rangle$. If all schemas in $\mathbb{T}$ are monotonic for $(\kappa_1', ..., \kappa_n', \alpha')$ and $\langle \kappa_1', ..., \kappa_n', \alpha' \Vdash \gamma' \rangle$ is a valid sequent, then $\langle \kappa_1, ..., \kappa_n, \alpha \Vdash \gamma \rangle$ is valid modulo σ.*

Proof. Suppose we σ-apply transfer schemas m times to $\langle \kappa_1, ..., \kappa_n, \alpha \Vdash \gamma \rangle$ to end with a valid sequent, $\langle \kappa_1', ..., \kappa_n', \alpha' \Vdash \gamma' \rangle$. This gives rise to m σ-reifications, that is m reifications, $f_1, \ldots, f_m$. The composition of reifications is a reification. Then, it is easy to see that we can apply $f_m \circ \cdots \circ f_1$ to obtain some sequent $\langle \kappa_1'', ..., \kappa_n'', \alpha'' \Vdash \gamma'' \rangle$ and then we can apply the schemas to obtain a sequent which is label-isomorphic to $\langle \kappa_1', ..., \kappa_n', \alpha' \Vdash \gamma' \rangle$. This is because if a monotonic schema is applicable to a sequent then it is applicable to any reification of the context. Therefore, $\langle \kappa_1'', ..., \kappa_n'', \alpha'' \Vdash \gamma'' \rangle$ is valid and thus $\langle \kappa_1, ..., \kappa_n, \alpha \Vdash \gamma \rangle$ is valid modulo σ. □

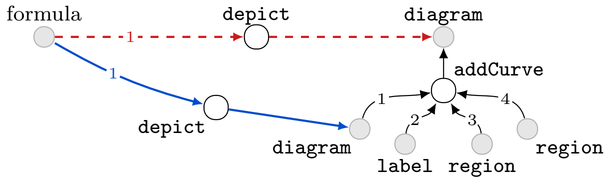
Appendix C. Appendix: Structure transfer worked example

The following transfer schemas characterise the notion of depiction.

Schema 1. From example 4.3. Let $\mathcal{M} = (\mathcal{C}, \mathcal{D}, \mathcal{G})$ be a multi-space where $\mathcal{C}$ encodes Set Algebra, $\mathcal{D}$ encodes Euler Diagrams and $\mathcal{G}$ encodes relations across and within the spaces. The sequent $\langle \kappa_1, \kappa_2, \alpha \Vdash \gamma \rangle$ for $\mathcal{M}$, depicted below, is as schema. It captures the intuition that a diagram depicts a conjunction if it depicts both conjuncts.

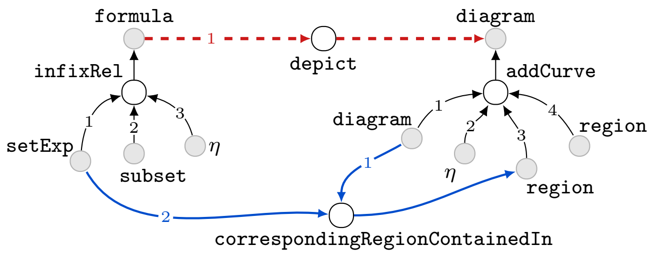


Schema 2. If a diagram depicts a formula then adding another curve preserves the depiction.



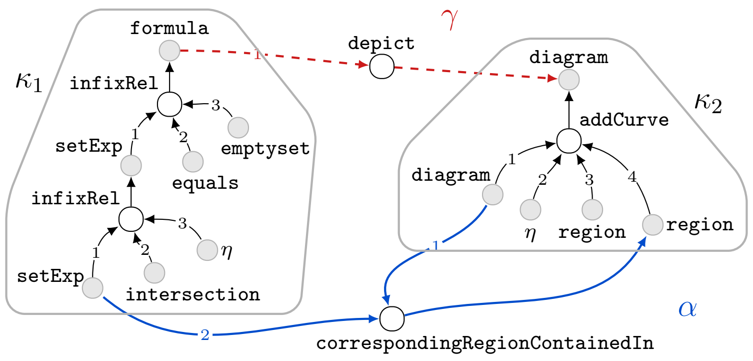
Schema 3 (Depicting subsets)**.** From example 4.14. To depict an expression of the form $x \subseteq \eta$, where η is a variable, it suffices to take a diagram, d, and make sure that η is

drawn so that the region corresponding to x is part of the *in* regions of the curve labelled by η (based on (Stapleton et al., 2010)). This means that **for every** η the following is a schema[20]:

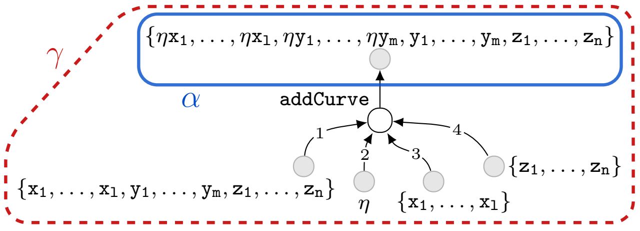


Using this schema as a $\{2\}$-transfer schema would produce a diagram from a given formula[21].

Schema 4 (Depicting disjoint sets)**.** From example 4.12. To depict an expression of the form $x \cap \eta = \emptyset$, where x is a set expression and η is a variable, it suffices to take a diagram, d, and add η in such a way that the region corresponding to x in d is part of the *out* regions of the curve labelled by η (based on (Stapleton et al., 2010)). Thus, the following figure describes a family of schemas, one for each η, subtype of `var` in Set Algebra, and subtype of `label` in Euler Diagrams.



Schema 5 (Top-down instantiation of an `addCurve` configuration)**.** From example 4.10. Provided a diagram of type $\{\eta\mathtt{x}_1, \ldots, \eta\mathtt{x}_l, \eta\mathtt{y}_1, \ldots, \eta\mathtt{y}_m, \mathtt{y}_1, \ldots, \mathtt{y}_m, \mathtt{z}_1, \ldots, \mathtt{z}_n\} \leq$ `diagram`, where η is a label that does not appear in either $\mathtt{x}_1, \ldots, \mathtt{x}_l, \mathtt{y}_1, \ldots, \mathtt{y}_m, \mathtt{z}_1, \ldots, \mathtt{z}_n$ then we can decompose $\{\eta\mathtt{x}_1, \ldots, \eta\mathtt{x}_l, \eta\mathtt{y}_1, \ldots, \eta\mathtt{y}_m, \mathtt{y}_1, \ldots, \mathtt{y}_m, \mathtt{z}_1, \ldots, \mathtt{z}_n\}$ by removing η. This is captured by the family of schemas $\langle \alpha \Vdash \gamma \rangle$, pictured below, for unary multi-space system $\mathcal{D}$:

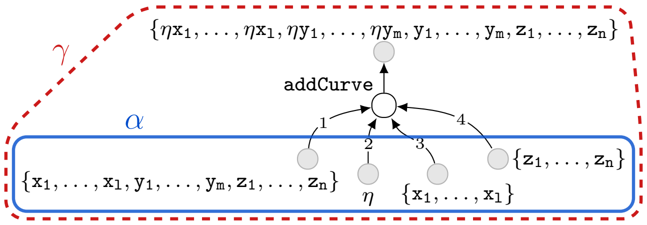


$\{\eta x_1, \ldots, \eta x_l, \eta y_1, \ldots, \eta y_m, y_1, \ldots, y_m, z_1, \ldots, z_n\}$
γ
α
addCurve
$\{z_1, \ldots, z_n\}$
$\{x_1, \ldots, x_l, y_1, \ldots, y_m, z_1, \ldots, z_n\}$
η
$\{x_1, \ldots, x_l\}$

20. Hereafter we will draw sequents and schemas of the form $\langle \kappa_1, \kappa_2, \alpha \Vdash \gamma \rangle$ with κ_1 on the left, κ_2 on the right, α with thick blue arrows, and γ with dashed red arrows.
21. Moreover, using this schema as a $\{1\}$-transfer schema can be used for producing a formula for a given diagram. Using it as $\{1, 2\}$-transfer schema can be used to produce both the given assumptions and goals. Using it as a $\emptyset$-transfer schema is no different to using it as a schema.

This family of schemas represents knowledge about how to *parse* an Euler diagram.

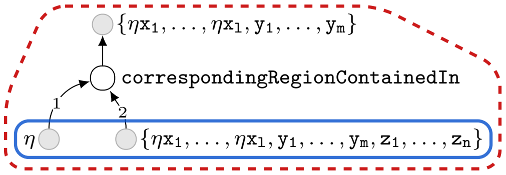
Schema 6 (Bottom-up instantiation of an `addCurve` configuration)**.** From example 4.9. Provided that `addCurve` takes as inputs: a diagram of type $\{\mathtt{x}_1, \dots, \mathtt{x}_l, \mathtt{y}_1, \dots, \mathtt{y}_m, \mathtt{z}_1, \dots, \mathtt{z}_n\} \leq$ `diagram`, a label $\eta \leq$ `label`, an *in* region $\{\mathtt{x}_1, \dots, \mathtt{x}_l\} \leq$ `region`, and an *out* region $\{\mathtt{z}_1, \dots, \mathtt{z}_n\} \leq$ `region`, where η does not appear in $\{\mathtt{x}_1, \dots, \mathtt{x}_l, \mathtt{y}_1, \dots, \mathtt{y}_m, \mathtt{z}_1, \dots, \mathtt{z}_n\}$, i.e. it is not a label in any of the words, *then* we can infer the output. This is captured by the family of schemas $\langle \alpha \Vdash \gamma \rangle$, pictured below, for unary multi-space system $\mathcal{D}$:


γ
$\{\eta\mathtt{x}_1, \dots, \eta\mathtt{x}_l, \eta\mathtt{y}_1, \dots, \eta\mathtt{y}_m, \mathtt{y}_1, \dots, \mathtt{y}_m, \mathtt{z}_1, \dots, \mathtt{z}_n\}$
addCurve
α
1
2
3
4
$\{\mathtt{z}_1, \dots, \mathtt{z}_n\}$
$\{\mathtt{x}_1, \dots, \mathtt{x}_l, \mathtt{y}_1, \dots, \mathtt{y}_m, \mathtt{z}_1, \dots, \mathtt{z}_n\}$
η
$\{\mathtt{x}_1, \dots, \mathtt{x}_l\}$

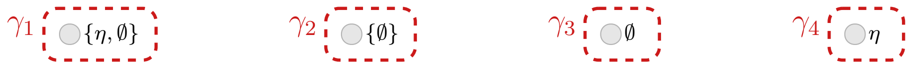
Note that in the output, $\{\eta\mathtt{x}_1, \dots, \eta\mathtt{x}_l, \eta\mathtt{y}_1, \dots, \eta\mathtt{y}_m, \mathtt{y}_1, \dots, \mathtt{y}_m, \mathtt{z}_1, \dots, \mathtt{z}_n\}$, the zones of the *in* region, $\{\mathtt{x}_1, \dots, \mathtt{x}_l\}$, are immersed in η, the zones of the *out* region, $\{\mathtt{z}_1, \dots, \mathtt{z}_n\}$, are disjoint of η, and the remaining ones, $\{\mathtt{y}_1, \dots, \mathtt{y}_m\}$, are each split in two: one inside, and one outside of η.

If we see the constructor `addCurve` as a program, this family of schemas represents knowledge about how to *compute* this program.

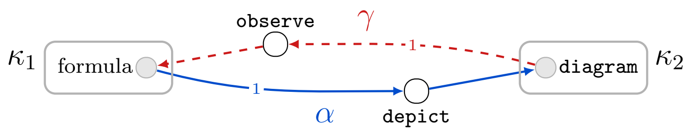
Schema 7 (Set expressions and their corresponding regions)**.** From Example 4.13. This schema tells you how to propagate from the input to the output of the `corresponding-RegionContainedIn` constructor. For every $\mathtt{x}_1, \dots, \mathtt{x}_l, \mathtt{y}_1, \dots, \mathtt{y}_m, \mathtt{z}_1, \dots, \mathtt{z}_n$ and η the following is a schema:


$\{\eta\mathtt{x}_1, \dots, \eta\mathtt{x}_l, \mathtt{y}_1, \dots, \mathtt{y}_m\}$
correspondingRegionContainedIn
1
2
η
$\{\eta\mathtt{x}_1, \dots, \eta\mathtt{x}_l, \mathtt{y}_1, \dots, \mathtt{y}_m, \mathtt{z}_1, \dots, \mathtt{z}_n\}$

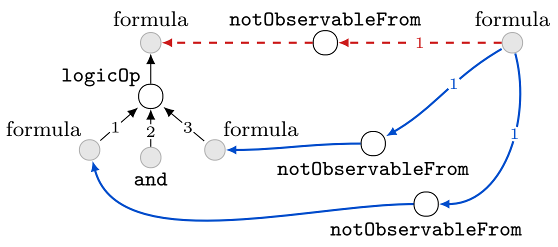
Schema 8 (Singleton instantiation)**.** From Example 4.8. For any label, η, the sequents $\langle \ \Vdash \gamma_1 \rangle$, $\langle \ \Vdash \gamma_2 \rangle$, $\langle \ \Vdash \gamma_3 \rangle$ and $\langle \ \Vdash \gamma_4 \rangle$, pictured below, are schemas for unary multi-space system $\mathcal{D}$.



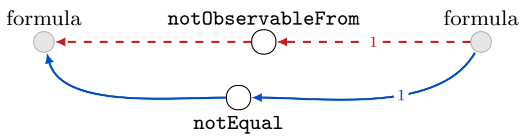
Schema 9 (Observe and depict are dual)**.** From example 4.11. If a diagram depicts a formula then the same formula can be observed from the diagram. Thus $\langle \kappa_1, \kappa_2, \alpha \Vdash \gamma \rangle$, depicted below, is a schema.



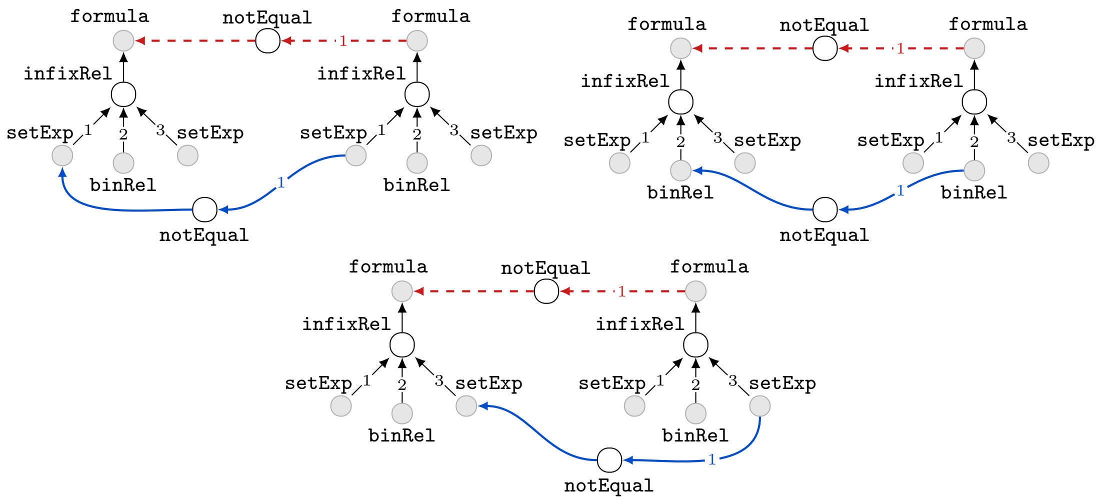
Schema 10. A formula is not observable from a conjunct of formulas if it is not observable from either conjunct.



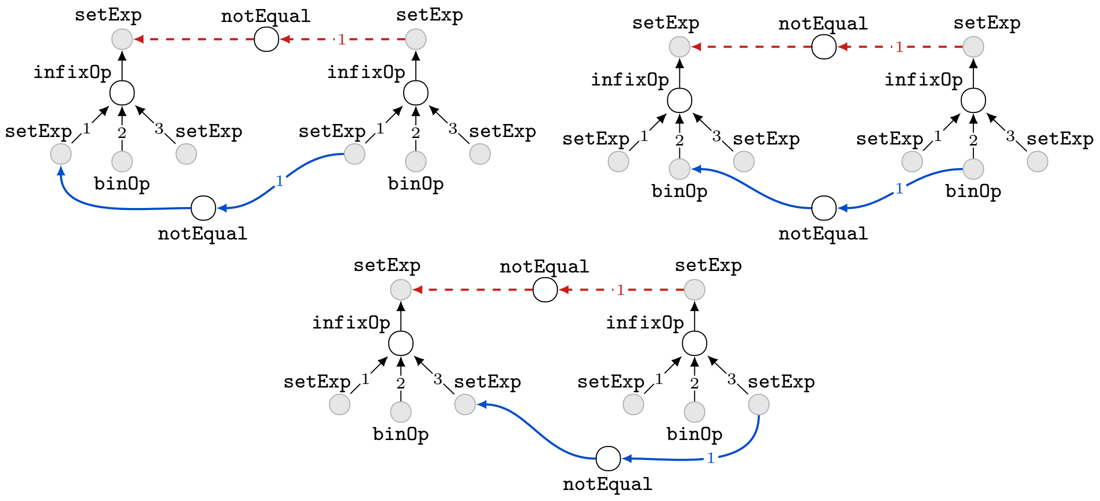
Schema 11. A formula is not observable from another if they are not *syntactically* equal.



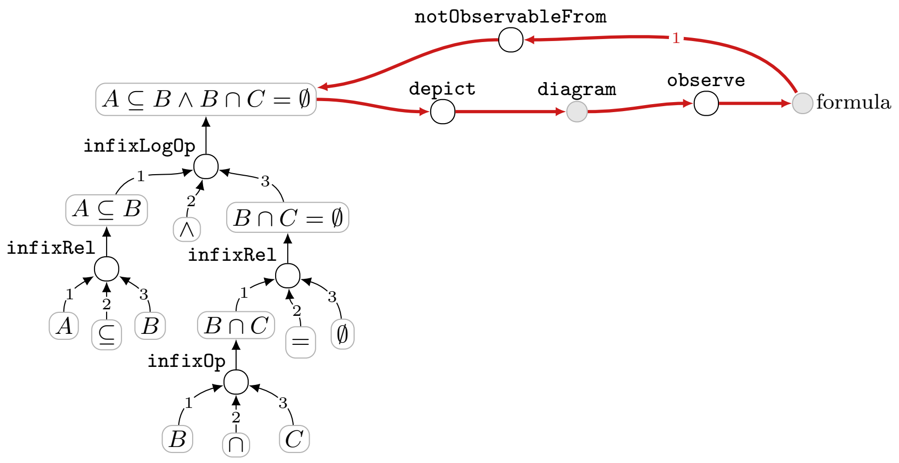
Schema 12. A pair of atomic formulas are not equal if any part of them is not equal. Thus we have three schemas:



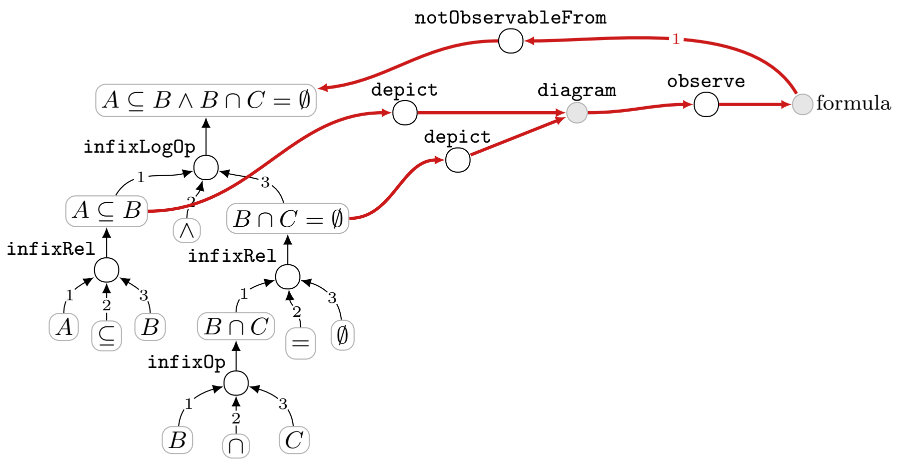
Schema 13. A pair of set expressions are not equal if any part of them is not equal. Thus we have three schemas:



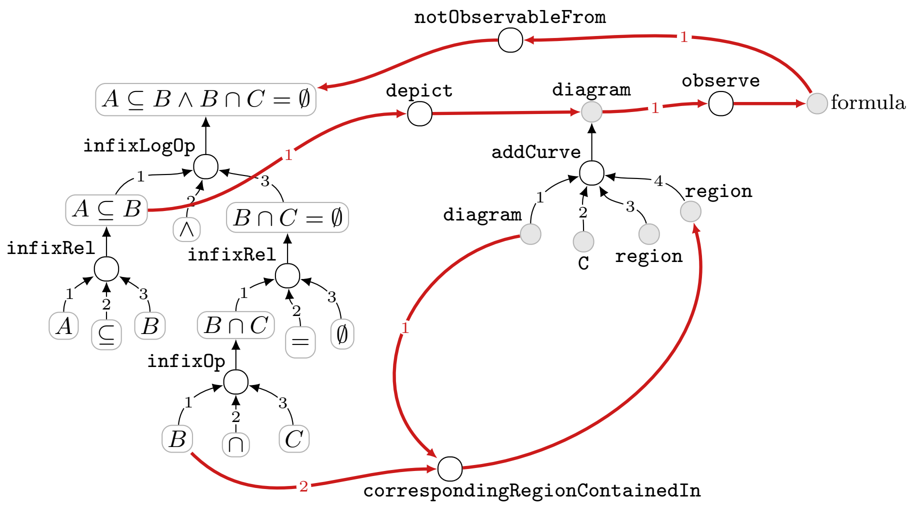
Problem encoding The multi-space we will work on is $(\mathcal{C}, \mathcal{D}, \mathcal{C}, \mathcal{G})$ where $\mathcal{C}$ encodes Set Algebra, $\mathcal{D}$ encodes Euler Diagrams and $\mathcal{G}$ encodes relations across and within the spaces. Given expression $A \subseteq B \wedge B \cap C = \emptyset$, we want to find a diagram that depicts it and a set expression that can be observed from the diagram. In other words, given the sequent $\langle \kappa_1, \kappa_2, \kappa_3, \alpha \Vdash \gamma \rangle$, visualised below, we want to find a $\{2,3\}$-reification of it that witnesses its validity modulo $\{2,3\}$. Note that we encode the relation of *not being observable from* to prevent a trivial observation.



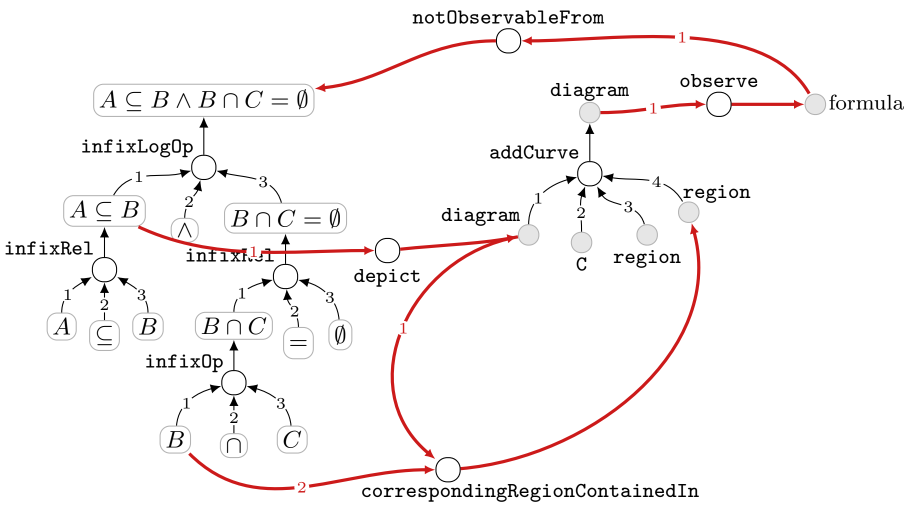
Apply schema 1 (depict conjunction):



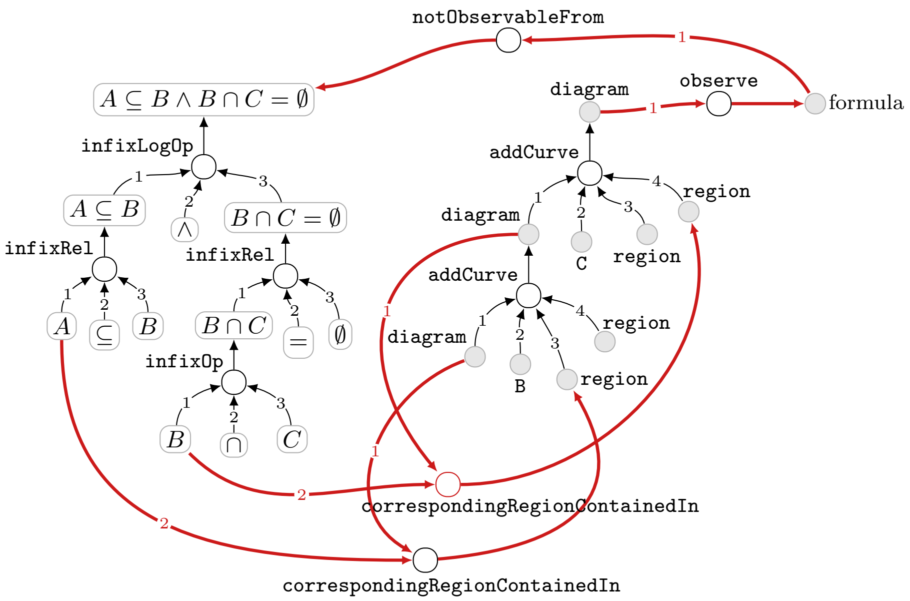
Apply schema 4 (depict disjoint relation):



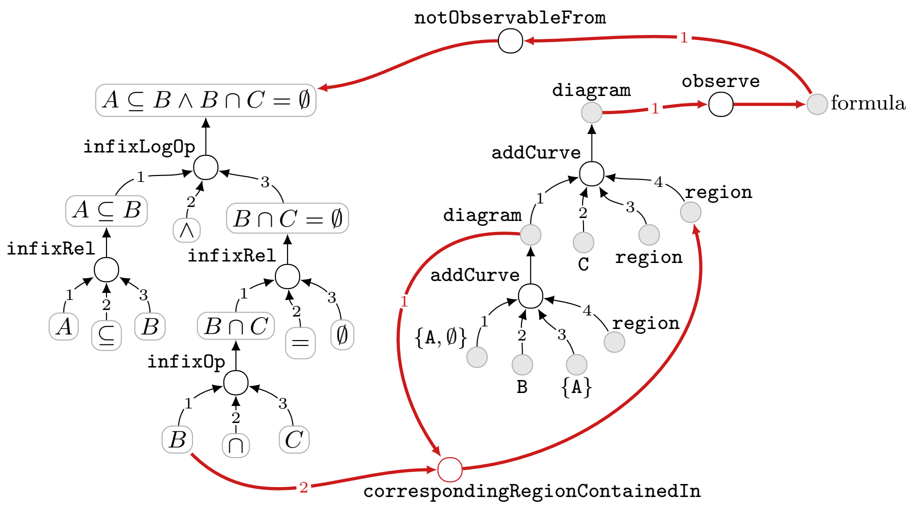
Apply Schema 2 (pass depict to the first argument of `addCurve`):



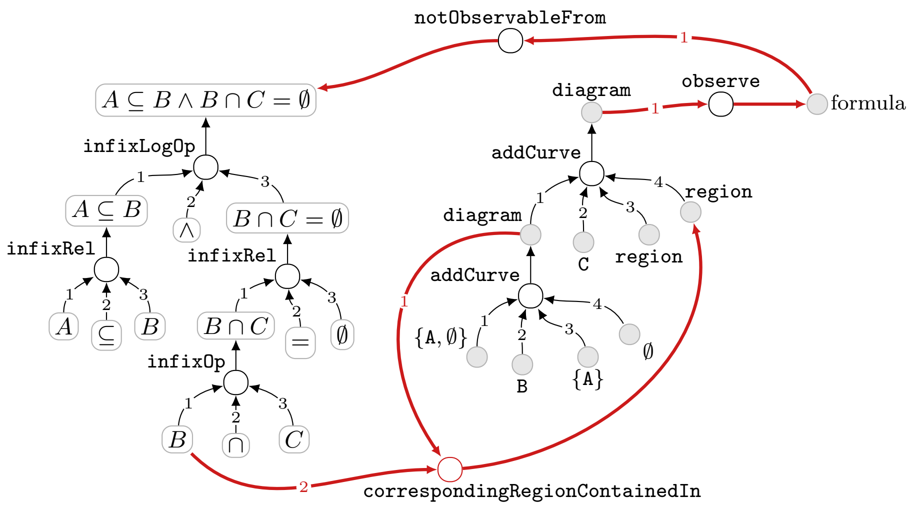
Apply Schema 3 (depict subset relation):



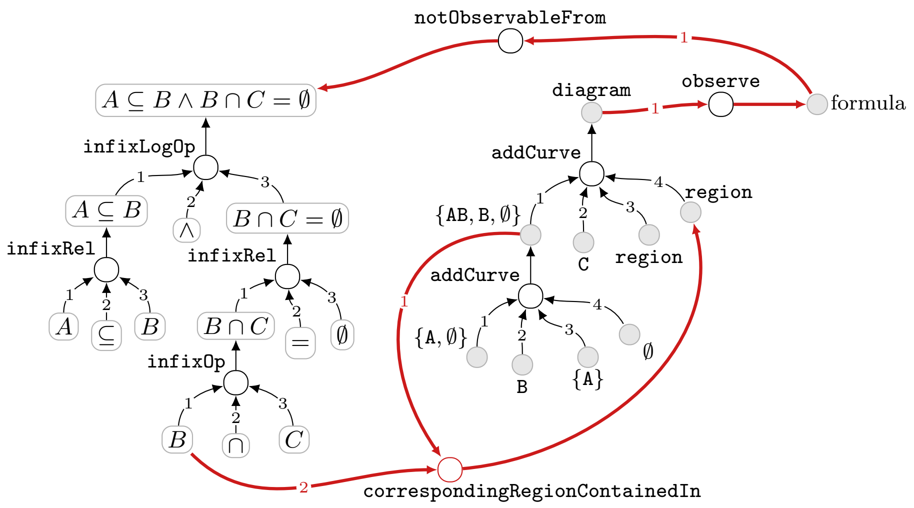
Apply Schema 7 (compute `correspondingRegionContainedIn`):



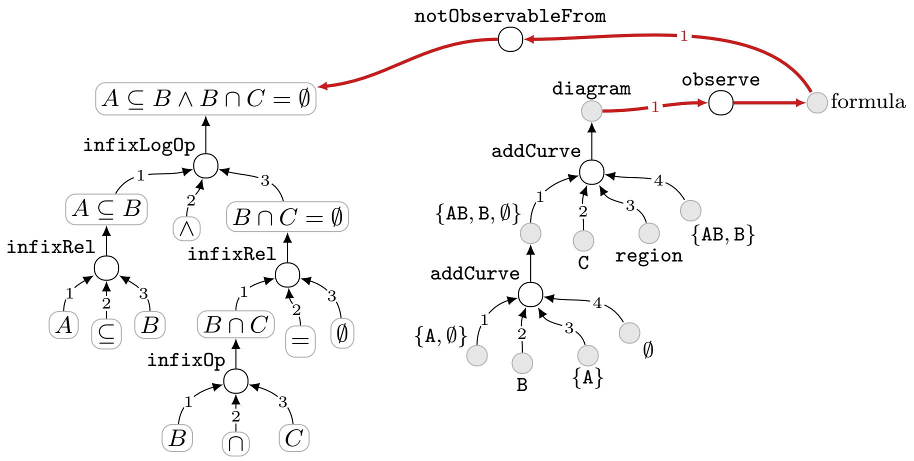
Apply Schema 8 (specialise region to $\emptyset$):



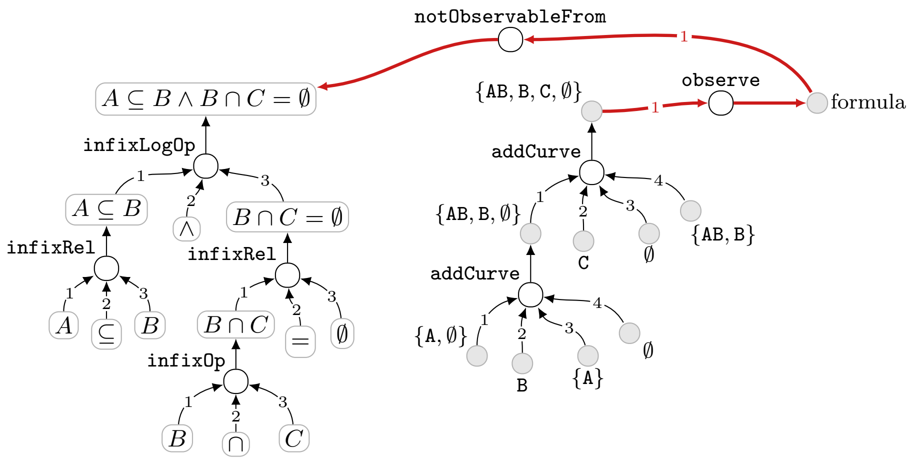
Apply Schema 6 (specialise type of diagram given inputs of `addCurve`):



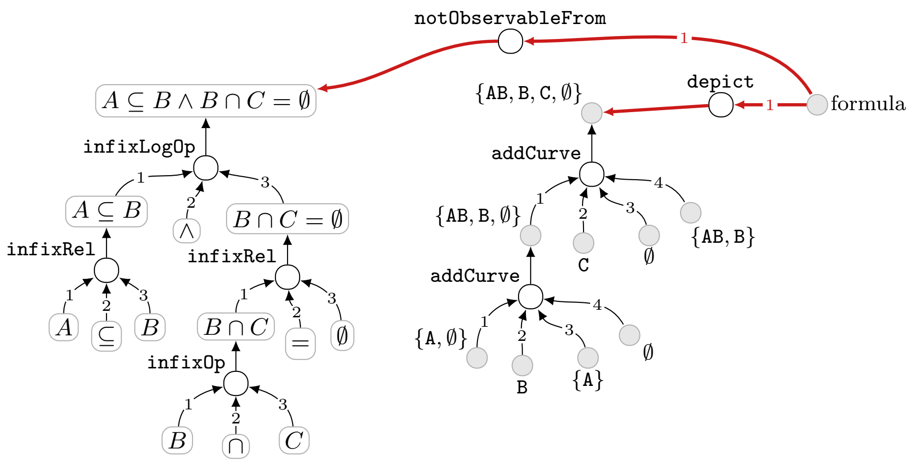
Apply Schema 7 again (compute `correspondingRegionContainedIn`):



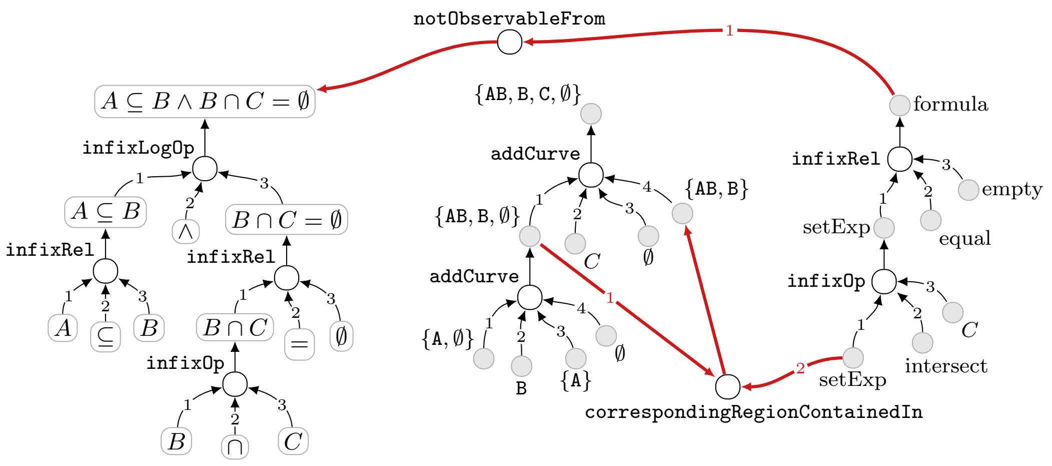
Apply Schema 8 (specialise region to $\emptyset$) and then Schema 6 (specialise type of diagram):



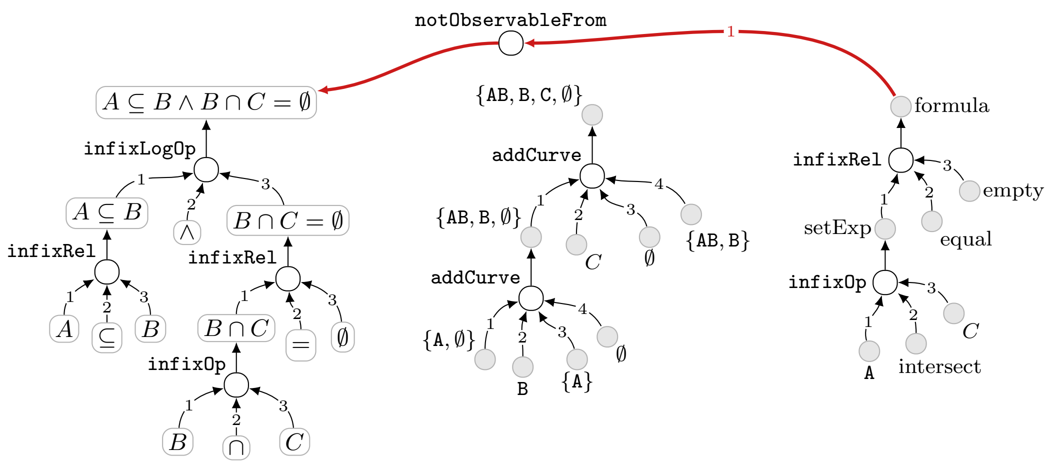
Apply Schema 9 (flip depict and observe):



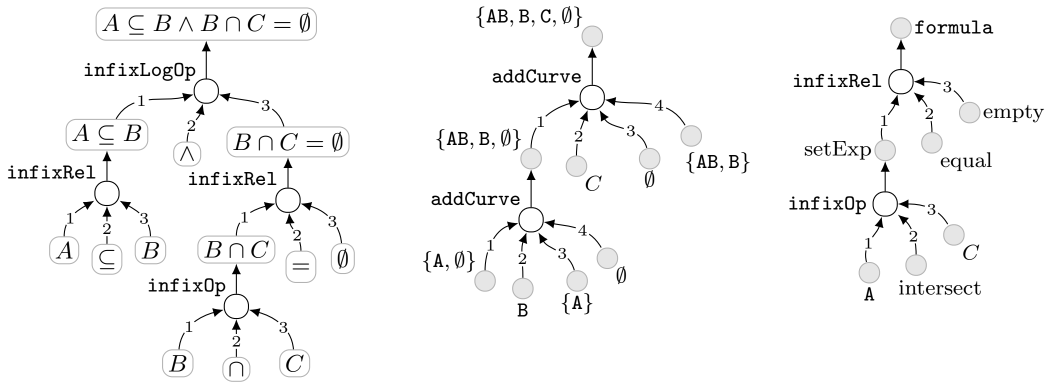
We could observe something trivial, but we can also allow the schema application to add structure to the set expression. Thus, we apply Schema 4:



And now we can apply Schema 7 (compute `correspondingRegionContainedIn` noting that during the application we could specialise setExp with either A or B):



And finally we can apply Schemas 10, 11 and 12 to discharge the `notObservableFrom` goal. Note that this would not have been possible if B had been chosen.



References

Abrial, J.-R., Butler, M., Hallerstede, S., Hoang, T. S., Mehta, F., & Voisin, L. (2010). Rodin: an open toolset for modelling and reasoning in Event-B. *International journal on software tools for technology transfer*, *12*(6), 447–466.

Ainsworth, S. (1999). The functions of multiple representations. *Computers & education*, *33*(2-3), 131–152.

Barwise, J., & Etchemendy, J. (1990). *Visualization in mathematics*, chap. Visual information and valid reasoning, pp. 8–23. Mathematical Association of America.

Barwise, J., & Etchemendy, J. (1992). Hyperproof: Logical reasoning with diagrams. In Chandrasekaran, B., & Simon, H. (Eds.), *Reasoning with Diagrammatic Representations*, pp. 80–84. AAAI press.

Barwise, J., & Etchemendy, J. (2019). Visual information and valid reasoning. In *Philosophy and the Computer*, pp. 160–182. Routledge.

Barwise, J., & Seligman, J. (1997). *Information flow: the logic of distributed systems*. Cambridge University Press.

Blake, A., Stapleton, G., Rodgers, P., & Touloumis, A. (2021). Evaluating free rides and observational advantages in set visualizations. *Journal of Logic, Language and Information*, *30*(3), 557–600.

Cheng, P., Garcia, G. G., Raggi, D., Stockdill, A., & Jamnik, M. (2021). Cognitive properties of representations: A framework. In *International Conference on Theory and Application of Diagrams*, pp. 415–430. Springer.

Cheng, P. C.-H. (2002). Electrifying diagrams for learning: principles for complex representational systems. *Cognitive Science*, *26*(6), 685–736.

Cheng, P. C.-H. (2011). Probably good diagrams for learning: Representational epistemic recodification of probability theory. *Topics in Cognitive Science*, *3*(3), 475–498.

Cheng, P. C.-H., Lowe, R. K., & Scaife, M. (2001). *Cognitive Science Approaches To Understanding Diagrammatic Representations*, pp. 79–94. Springer Netherlands, Dordrecht.

Coen, C. S. (2004). A semi-reflexive tactic for (sub-) equational reasoning. In *International Workshop on Types for Proofs and Programs*, pp. 98–114. Springer.

Cohen, C., Dénès, M., & Mörtberg, A. (2013). Refinements for free!. In *International Conference on Certified Programs and Proofs*, pp. 147–162. Springer.

Delaware, B., Pit-Claudel, C., Gross, J., & Chlipala, A. (2015). Fiat: Deductive synthesis of abstract data types in a proof assistant. *Acm Sigplan Notices*, *50*(1), 689–700.

Falkenhainer, B., Forbus, K. D., & Gentner, D. (1989). The structure-mapping engine: Algorithm and examples. *Artificial Intelligence*, *41*(1), 1–63.

Gentner, D. (1983). Structure-mapping: A theoretical framework for analogy. *Cognitive science*, *7*(2), 155–170.

Gick, M. L., & Holyoak, K. J. (1983). Schema induction and analogical transfer. *Cognitive psychology*, *15*(1), 1–38.

Huffman, B., & Kunčar, O. (2013). Lifting and transfer: A modular design for quotients in Isabelle/HOL. In *International Conference on Certified Programs and Proofs*, pp. 131–146. Springer.

Krumnack, U., Schwering, A., Gust, H., & Kühnberger, K.-U. (2007). Restricted higher-order anti-unification for analogy making. In *Australasian Joint Conference on Artificial Intelligence*, pp. 273–282. Springer.

Kutz, O., Mossakowski, T., & Lücke, D. (2010). Carnap, goguen, and the hyperontologies: Logical pluralism and heterogeneous structuring in ontology design. *Logica Universalis*, *4*(2), 255–333.

Lakoff, G., & Johnson, M. (2008). *Metaphors we live by.* University of Chicago press.

Lakoff, G., & Núñez, R. (2000). *Where mathematics comes from*, Vol. 6. New York: Basic Books.

Larkin, J., & Simon, H. (1987). Why a diagram is (sometimes) worth ten thousand words. *Journal of Cognitive Science*, *11*, 65–99.

Mossakowski, T., Maeder, C., & Lüttich, K. (2007). The Heterogeneous Tool Set, Hets. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 519–522. Springer.

Newell, A., Simon, H. A., et al. (1972). *Human problem solving*, Vol. 104. Prentice-hall Englewood Cliffs, NJ.

Palmer, S. (1978). Fundamental aspects of cognitive representation. In Rosch, E., & Lloyd, B. (Eds.), *Cognition and Categorization*, pp. 259–303. Lawrence Elbaum Associates.

Polya, G. (1957). *How to Solve It.* Princeton University Press.

Rabe, F., & Kohlhase, M. (2013). A scalable module system. *Information and Computation*, *230*, 1–54.

Raggi, D., Stapleton, G., Stockdill, A., Jamnik, M., Garcia, G. G., & Cheng, P. (2020). How to (re)represent it?. In *32nd International Conference on Tools with Artificial Intelligence*, pp. 1224–1232. IEEE.

Raggi, D. (2022). An implementation based on RST. https://github.com/danielraggi/rep2rep.

Raggi, D., Bundy, A., Grov, G., & Pease, A. (2016). Automating change of representation for proofs in discrete mathematics (extended version). *Mathematics in Computer Science*, *10*(4), 429–457.

Raggi, D., Stapleton, G., Jamnik, M., Stockdill, A., Garcia, G. G., & Cheng, P. C.-H. (2022). Oruga: An avatar of representational systems theory. In *CEUR Workshop Proceedings.*

Raggi, D., Stapleton, G., Stockdill, A., Jamnik, M., Garcia, G. G., & Cheng, P. C.-H. (2023). *Representational Systems Theory: A Unified Approach to Encoding, Analysing and Transforming Representations.* CSLI Press.

Schmidt, M., Krumnack, U., Gust, H., & Kühnberger, K.-U. (2014). Heuristic-driven theory projection: An overview. *Computational approaches to analogical reasoning: Current trends*, *1*(548), 163–194.

Shimojima, A. (1999). The graphic-linguistic distinction exploring alternatives. *Artificial Intelligence Review*, *13*, 313–335.

Stapleton, G., Jamnik, M., & Shimojima, A. (2017). What makes an effective representation of information: A formal account of observational advantages. *Journal of Logic, Language and Information*, *26*(2), 143–177.

Stapleton, G., Rodgers, P., Howse, J., & Zhang, L. (2010). Inductively generating Euler diagrams. *IEEE Transactions on Visualization and Computer Graphics*, *17*(1), 88–100.

Tabareau, N., Tanter, É., & Sozeau, M. (2018). Equivalences for free: Univalent parametricity for effective transport. *Proceedings of the ACM on Programming Languages*, *2*(ICFP), 1–29.

Wetzel, L. (2018). Types and Tokens. In *The Stanford Encyclopedia of Philosophy* (Fall 2018 edition). Metaphysics Research Lab, Stanford University.