PYVIVID; A CONCRETE FRAMEWORK FOR MECHANIZED DIAGRAMMATIC REASONING

By

Nicholas Marton

A Thesis Submitted to the Graduate

Faculty of Rensselaer Polytechnic Institute

in Partial Fulfillment of the

Requirements for the Degree of

MASTER OF SCIENCE

Major Subject: COMPUTER SCIENCE

Approved by the
Examining Committee:

Selmer Bringsjord
Thesis Adviser

Jim Hendler, Member

Carlos Varela, Member

Bram Van Heuveln, Member

Rensselaer Polytechnic Institute
Troy, New York

April 2016
(For Graduation May 2016)

CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGMENTS

First and foremost, I would like to thank my thesis advisor Prof. Selmer Bringsjord for his continuous support of my studies and research, for his patience, enthusiasm, motivation, and for his immense knowledge. His guidance has helped me immeasurably in all the time of research and writing of this thesis. His spearheading of investigation of axiomatic physics, in collaboration with the Rensselaer Polytechnic Institute, is much appreciated.

Besides my advisor, I would like to thank the rest of my thesis committee: Prof. James A. Hendler, Prof. Carlos A. Varela, and Prof. Bram van Heuveln, for their insightful comments, difficult questions, and encouragement.

I would also like to extend my sincere gratitude to Dr. John Licato for offering me a position within the Rensselaer Artificial Intelligence and Reasoning (RAIR) Laboratory, which led to the inception of the work examined in this thesis.

I would like to thank Dr. Konstantine Arkoudas for paving the way for research concerning diagrammatic reasoning within a heterogeneous, natural deduction setting.

Finally, I am grateful for the financial support the Air Force Office of Scientific Research (AFOSR) and Office of Naval Research (ONR) have provided to Prof. Bringsjord's RAIR Lab for research and development in the intersection of computational logic and the particular sciences.

ABSTRACT

In 2007, the abstract reasoning framework Vivid was introduced by [Arkoudas and Bringsjord, 2009] for the purpose of mechanized heterogeneous natural deduction that combines symbolic and diagrammatic reasoning. We introduce an implementation of the Vivid framework in the Python programming language, pyVivid, the first such concrete framework capable of arbitrary formal diagrammatic reasoning. Furthermore, we present a protocol that allows for the extension of the pyVivid library, opening up the possibility of the incorporation of diagrammatic reasoning to other programmers.

1. Introduction

Diagrams are a pervasive and valuable tool extensively used in a wide variety of fields. To elaborate on the areas mentioned by [Arkoudas and Bringsjord, 2009], for example, there are numerous diagrams used in the field of statistics such as bar charts, pie charts and lorenz curves [Gastwirth, 1971]; in the field of physics, free-body, energy-level, vector fields and Minkowski [Parker and Schmieg, 1970] diagrams are prevalent; in the biological realm, phylogenetic trees [Fitch and Margoliash, 1967], ribbon diagrams and cell migration diagrams are used to encapsulate a huge amount of variability succinctly and in an intuitive way; circuit diagrams, process flow diagrams and phase diagrams are utilized throughout different fields of engineering; diagrams are widespread in the field computer science where their usage ranges across a broad set of subfields like firmware interface flowcharts in computer architecture, network topology diagrams in computer networking, UML diagrams (which are used specifically to capture and intuitively explain more abstract features of larger software architectures [Berardi et al., 2005]) and hash function diagrams in cryptography among others; there are even algorithms that can be represented as graphical models (and thus as diagrams) such as bayesian networks [Jordan, 1998]. One of the major reasons of the prevalence of diagrams across such a large amount of fields of study is the inherent ability of diagrams to capture incomplete information. System containing incomplete information pervade across a multitude of fields, yet there is no general framework for dealing with them in a rigorous way. Additionally, diagrams are reasoned over and used by humans to reason with, but there is no rigorous framework for regimenting that, and enabling an artificial intelligence to do it.

Roughly, there are two approaches to reasoning over incomplete information. In the first approach, the objective is the quantification of indefiniteness; in this approach, probabilistic or fuzzy models of uncertainty are usually used. This approach has been rigorously investigated with various probabilistic models and it has even been shown that the theory of lower and upper previsions appears to be general enough to model the most typical types of uncertainty [Walley, 1996]. The models used in this approach, however, are inherently limited by the amount of information from which their estimations are made. The second, much less investigated, approach is the use of deductive reasoning. In this case, formal conclusions about the incomplete information are shown to necessarily follow some set of premises (most likely composed from the incomplete information itself). This approach is severely limited by the lack of sound frameworks capable of dealing with incomplete information. Vivid aims to address this problem; however, the framework itself provides no complete concrete implementation of the inference mechanisms therein currently.

The Vivid framework, presented by [Arkoudas and Bringsjord, 2009], is a domain-independent framework for mechanized heterogeneous natural deduction that combines diagrammatic and symbolic reasoning presented in the form of a family of denotational proof languages (DPLs). Formal proof construction and proof checking, in a matter that includes diagrams, is a main purpose for Vivid. The framework also provides a novel form structure, *named system states*,

designed specifically to deal with incomplete information in the form of undetermined diagrams. These named system states can also be refined gradually as more information is obtain through the use of the inference rules provided in the framework, permitting deductive reasoning over incomplete information; an approach which needs to undergo more investigation. Additionally, in the Vivid framework, the general inference mechanisms introduced extend the assumption-base semantics of DPLs, allowing for the valid extraction of information from diagrams and incorporation of sentential information into diagrams. This ability to combine heterogeneous knowledge cannot be understated. Traditional logic makes use of symbolic formalisms captured entirely in linguistic expressions. Diagrams when applied in a useful manner, however, almost certainly have some structural similarities to what they are meant to be representing; they analogically representative of the information contained therein. This feature is incredibly important in the sense that intuition can be drawn very easily from well constructed diagrams. Additionally, a lot of these diagrams can be used during the course of constraint programming [1].

While Vivid seems like a candidate for a deductive approach to reasoning over incomplete information as well as a powerful tool for heterogeneous inference, there is no widly available implementation until now. Additionally, pyVivid has the ability to link to outside resources and can easily be extended to handle any potential diagrams. With the introduction of the pyVivid library, the functionality of the Vivid framework is now freely available to programmers and logicians alike.

In the next section, we summarize the definitions and notations introduced in the original Vivid framework. Then, we provide an overview of the formal semantics of the framework. In section 3, we first describe why the Python programming language was chosen for the creation of the pyVivid library, then we provide an overview of the available features of the pyVivid library and finally we describe the process of extending pyVivid with arbitrary Python objects. Section 4 demonstrates multiple usages of the pyVivid library in the context of proof verification. Finally, in section 5, we discuss related and future work.

[1] We're indebted to Dr. Carlos Varela for pointing this out; it is a topic that will be addressed in more detail in the future.

2. Vivid: The Formal Framework

The Vivid framework is substantial in its notation and definitions and intricate in its semantics. Furthermore, as many of the components of the pyVivid architecture seek to directly encode the definitions provided in Vivid, it is practical to understand the underlying framework before using the pyVivid library. Therefore, we dedicate the rest of this chapter to understanding the most applicable notations, definitions and semantics of the Vivid framework, summarized directly from [Arkoudas and Bringsjord, 2009], with respect to the pyVivid library.

2.1 Definitions and Notation

To understand the formal semantics of the diagrammatic deductions introduced in Vivid, it is necessary to understand the framework's native definitions and notation. While the Vivid framework introduces an extensive amount of different definitions and notational conveniences, some are more applicable, with respect to the pyVivid library, than others. What follows is a description of the more important and widely used definitions and notations therein:

- The **set-theoretic difference** between any two sets A and B, denoted by $A \setminus B$, is defined as follows: $A \setminus B = \{x \in A \mid x \notin B\}$.
- For any set A, $\mathcal{P}_{fin}(A)$ denotes the set of all finite subsets of A.
- For an arbitrary relation $R \subseteq A_1 \times \cdots \times A_n$, $D(R)$ denotes the set $\{A_1, \ldots, A_n\}$.
- An **attribute structure** is a pair $\mathcal{A} = (\{A_1, \ldots, A_k\}; \mathcal{R})$ consisting of a finite collection of sets $A_1, \ldots, A_k$ called **attributes** (where each A_i has a unique label l_i corresponding to it) and a countable collection $\mathcal{R}$ of computable relations, with $D(R) \subseteq \{A_1, \ldots, A_k\}$ for each $R \in \mathcal{R}$. When the relations of $\mathcal{A}$ are immaterial, we identify $\mathcal{A}$ with its attributes and write $\mathcal{A}$ as $l_1 : A_1, \ldots, l_k : A_k$, where l_i is the label of A_i.
- An **attribute system** based on some attribute structure $\mathcal{A}$ is a pair $\mathcal{S} = (\{s_1, \ldots, s_n\}; \mathcal{A})$ consisting of a finite number $n > 0$ of objects $s_1, \ldots, s_n$ and the attribute structure $\mathcal{A}$.
- A **state** of a system $\mathcal{S} = (\{s_1, \ldots, s_n\}, \{A_1, \ldots, A_k\})$ is a set of functions $\sigma = \{\delta_1, \ldots, \delta_k\}$ where each δ_i is a function from $\{s_1, \ldots, s_n\}$ to the set of non-empty finite subsets of A_i (where each δ_i is referred to as the state's **ascription** into A_i), i.e.,

  $$\delta_i : \{s_1, \ldots, s_n\} \to \mathcal{P}_{fin}(A_i) \setminus \{\emptyset\}.$$

  As an additional convention, given a state σ, attribute label l_i and object s_j, we write $\sigma(l_i, s_j)$ for $\delta_i(s_j)$ i.e., the value of the ascription δ_i for the object s_j in the state σ.
- If an ascription δ_i maps every object to a singleton, that is, if $|\delta_i(s_j)| = 1$ for every $j = 1, \ldots, n$, δ_i is referred to as a **valuation** and a **world** w is a state in which every ascription is a valuation.

- A state σ' of an attribute system $\mathcal{S} = (\{s_1, \ldots, s_n\}; l_1 : A_1, \ldots, l_k : A_k)$ is an **extension** of another state σ, written $\sigma' \sqsubseteq \sigma$, iff $\sigma'(l_i, s_j) \subseteq \sigma(l_i, s_j)$ for every $i = 1, \ldots, k$ and $j = 1, \ldots, n$. A state σ' is a **proper extension** of a state σ, denoted $\sigma' \sqsubset \sigma$ iff $\sigma' \sqsubseteq \sigma$ and $\sigma \not\sqsubseteq \sigma'$.

- A list of $m \geq 1$ attribute-object pairs $[(l_1; s_1), \ldots, (l_m; s_m)]$ is **homogeneous** iff $l_1 = \cdots = l_m$ and $s_1 = \cdots = s_m$, i.e., iff all m pairs are identical.

- Let $\sigma_1, \ldots, \sigma_m \sqsubset \sigma, m \geq 1$. A list of m attribute-object pairs $L = [(l_1; s_1), \ldots, (l_m; s_m)]$ **spans** the states $\sigma_1, \ldots, \sigma_m$ with respect to σ iff $\sigma_i(l_i, s_i) \subset \sigma(l_i, s_i)$ for every $i = 1, \ldots m$. Additionally, a list of m attribute-object pairs $L = [(l_1; s_1), \ldots, (l_m; s_m)]$ **properly spans** $\sigma_1, \ldots, \sigma_m$ w.r.t. σ iff for every sublist of $[i_1 \cdots i_{m'}]$ of $[i, \ldots, m]$ such that $[L(i_1), \cdots, L(i_{m'})]$ is homogeneous, we have

$$\left[\bigcup_{j=1}^{m'} \sigma_{i_j}(l_{i_j}, s_{i_j})\right] \subset \sigma(l_{i_1}, s_{i_1}).$$

  Equivalently L does not properly span $\sigma_1, \ldots, \sigma m$ with respect to σ iff for some such sublist we have

$$\sigma_{i_1}(l_{i_1}, s_{i_1}) \cup \cdots \cup \sigma_{i_{m'}}(l_{i_{m'}}, s_{i_{m'}}) = \sigma(l_{i_1}, s_{i_1}).$$

- Let $\sigma_1, \ldots, \sigma_m, \sigma' \sqsubset \sigma, m \geq 1$. We refer to σ' as an **alternate extension of σ w.r.t.** $\sigma_1, \ldots, \sigma_m$, written $Alt(\sigma, \{\sigma_1, \ldots, \sigma_m\}, \sigma')$ iff there is a list $L = [(l_1; s_1) \cdots (l_m; s_m)]$ properly spanning $\sigma_1, \ldots, \sigma_m$ w.r.t. σ such that for every attribute l and object s we have

$$\sigma'(l, s) = \sigma(l, s) \setminus \bigcup_{i \in Pos((l;s),L)} \sigma_i(l, s)$$

  where, $Pos(x, L) = \{i \in \{1, \ldots, n\} \mid x = x_i\}$. We write $\mathbf{AE}(\{\sigma_1, \ldots, \sigma_m\}, \sigma)$ for the set of all alternate extensions of σ w.r.t. $\sigma_1, \ldots, \sigma_m$.

- By **vocabulary**, we mean a first-order vocabulary Σ = (C;R;V) consisting of a set of constant symbols C; a set of relation symbols R; and a set of variables V.

- An **attribute interpretation** of some first-order vocabulary Σ into an attribute structure $\mathcal{A} = (\{l_1 : A_1, \ldots, l_k : A_k\}; \mathcal{R})$ is a mapping I that assigns, to each relation symbol $R \in$ R of arity n:

  1. A relation $R^I \in \mathcal{R}$ of some arity m, called the **realization** of R:

$$R^I \subset A_{i_1} \times \cdots \times A_{i_m}$$

     (where it is possible for $m \neq n$); and

  2. a list of m pairs

$$[(l_{i_1}; j_1) \cdots (l_{i_m}; j_m)]$$

     called the **profile** of R and denoted by $Prof(R)$, with $1 \leq j_x \leq n$ for each $x = 1, \ldots, m$.

- A **constant assignment** is a partial function ρ from the constants C of some vocabulary Σ to the objects $\{s_1, \ldots, s_n\}$ of some attribute system $\mathcal{S} = (\{s_1, \ldots, s_n\}; \mathcal{A})$. We write $Dom(\rho)$ for the domain of a constant assignment ρ, i.e., the set of all and only those constant symbols for which ρ is defined. A total constant assignment is written as $\widehat{\rho}$, with the hat indicating that the mapping is total. Additionally, two constant assignments ρ_1 and ρ_2 have a **conflict** iff there is some $c \in Dom(\rho_1) \cap Dom(\rho_2)$ such that $\rho_1(c) \neq \rho_2(c)$.

- A **variable assignment** is a total function χ from the variables V of some vocabulary Σ to the objects $\{s_1, \ldots, s_n\}$ of some attribute system $\mathcal{S} = (\{s_1, \ldots, s_n\}; \mathcal{A})$.

- A **Formula** F is defined over a vocabulary Σ as usual, with a term t being either a variable or constant symbol.

- A **named state** is a pair $(\sigma; \rho)$ consisting of a state σ and a constant assignment ρ.

- A named state $(\sigma'; \rho')$ is an **extension** of another named state $(\sigma; \rho)$, written $(\sigma'; \rho') \sqsubseteq (\sigma; \rho)$, iff $\sigma' \sqsubseteq \sigma$ and $\rho' \supseteq \rho$. Additionally, $(\sigma'; \rho')$ is a **proper extension** of $(\sigma; \rho)$, written $(\sigma'; \rho') \sqsubset (\sigma; \rho)$, iff $(\sigma'; \rho') \sqsubseteq (\sigma; \rho)$ and either $\sigma' \sqsubset \sigma$ or $\rho' \supset \rho$. Further, $(\sigma'; \rho')$ is a **finite extension** of $(\sigma; \rho)$, denoted $(\sigma'; \rho') \overset{\infty}{\sqsubseteq} (\sigma; \rho)$, iff $(\sigma'; \rho') \sqsubseteq (\sigma; \rho)$ and the difference $\rho' \setminus \rho$ is finite.

- A named state $(\sigma; \rho)$ is a **world** iff σ is a world and ρ is total.

- An **assumption base** β is a finite set of formulae.

- A **context** is a pair $\gamma = (\beta; (\sigma; \rho))$ consisting of an assumption base β and a named state $(\sigma; \rho)$.

- We define $\mathcal{V}^I_{(w;\rho)/\chi}(F)$ to assign a truth value to a formula F, w.r.t. a given a world w (of an attribute system $\mathcal{S} = (\{s_1, \ldots, s_n\}; \mathcal{A})$), along with a constant assignment ρ and variable assignment χ, as follows:
  First the constants **true** and **false** are self-evaluating:

$$\mathcal{V}^I_{(w;\rho)/\chi}[\mathbf{true}] = \mathbf{true} \quad \text{and} \quad \mathcal{V}^I_{(w;\rho)/\chi}[\mathbf{false}] = \mathbf{false}.$$

  Next, consider an atomic formula $R(t_1, \ldots, t_n)$, where R is a relation symbol of arity n and profile

$$[(l_{i_1}; j_1), \ldots, (l_{i_m}; j_m)].$$

  We have:

$$\mathcal{V}^I_{(\sigma;\rho)/\chi}[R^I(t_1, \ldots, t_n)] = \begin{cases} \mathbf{unknown} & \text{if } \exists\ k \in \{1, \ldots, m\}\ .\ t^{\rho,\chi}_{j_k} \uparrow; \\ \mathbf{true} & \text{if } R^I(w(l_{i_1}, t^{\rho,\chi}_{j_1}), \ldots, w(l_{i_m}, t^{\rho,\chi}_{j_m})); \\ \mathbf{false} & \text{if } \neg R^I(w(l_{i_1}, t^{\rho,\chi}_{j_1}), \ldots, w(l_{i_m}, t^{\rho,\chi}_{j_m})). \end{cases}$$

  where $t^{\rho,\chi} \uparrow$ indicates that $t^{\rho,\chi}$ is undefined (respectively $t^{\rho,\chi} \downarrow$ indicates that $t^{\rho,\chi}$ is defined). Vivid assigns truth values in accordance with the strong three-valued Kleene scheme.

- We define $I_{(\sigma;\rho)/\chi}(F)$ to assign a truth value to a formula F, given an arbitrary named state $(\sigma;\rho)$ (of an attribute system $\mathcal{S} = (\{s_1,\ldots,s_n\};\mathcal{A})$) along with a variable assignment χ as follows:

$$I_{(\sigma;\rho)/\chi}(F) = \begin{cases} \textbf{true} & \mathcal{V}^I_{(w;\rho)/\chi}[F] = \textbf{true} \text{ for every world } w \sqsubseteq \sigma \\ \textbf{false} & \mathcal{V}^I_{(w;\rho)/\chi}[F] = \textbf{false} \text{ for every world } w \sqsubseteq \sigma \\ \textbf{unknown} & \text{otherwise} \end{cases}$$

- For any given F, ρ, and χ, the **basis** of F w.r.t. ρ, and χ, denoted $\mathcal{B}(F,\rho,\chi)$ is a set of attribute-object pairs (or an error token ∞) defined on F by structural recursion. We present only the clause of the definition dealing with atomic formulae:

$$\mathcal{B}(R(t_1,\ldots,t_n),\rho,\chi) = \begin{cases} \{(l_{i_1};t^{\rho,\chi}_{j_1}),\ldots,(l_{i_m};t^{\rho,\chi}_{j_m})\} & \text{if } t^{\rho,\chi}_{j_k}\downarrow \text{ for every } k\in\{1,\ldots,m\}; \\ \infty & \text{otherwise.} \end{cases}$$

- A world $(w;\widehat{\rho})$ **satisfies a formula** F w.r.t. a variable assignment χ, denoted $(w;\widehat{\rho}) \models_\chi F$, iff

$$\mathcal{V}^I_{(w;\rho)/\chi}[F] = \textbf{true}.$$

- A world $(w;\widehat{\rho})$ **satisfies a named state** $(\sigma;\rho)$, denoted $(w;\widehat{\rho}) \models (\sigma;\rho)$, iff $(w;\widehat{\rho}) \sqsubseteq (\sigma;\rho)$.

- A world $(w;\widehat{\rho})$ **satisfies a context** $\gamma = (\beta;(\sigma;\rho))$ w.r.t. a given variable assignment χ, denoted $(w;\widehat{\rho}) \models_\chi \gamma$, iff $(w;\widehat{\rho}) \models_\chi F$ for every $F \in \beta$ and $(w;\widehat{\rho}) \models (\sigma;\rho)$.

- **A context γ entails a formula** F, denoted $\gamma \models F$, iff $(w;\widehat{\rho}) \models_\chi \gamma$ implies $(w;\widehat{\rho}) \models_\chi F$ for all worlds $(w;\widehat{\rho})$ and variable assignments χ.

- **A context γ entails a named state** $(\sigma';\rho')$, denoted $\gamma \models (\sigma';\rho')$, iff for all worlds $(w;\widehat{\rho})$ and variable assignments χ, we have $(w;\widehat{\rho}) \models (\sigma';\rho')$ whenever $(w;\widehat{\rho}) \models_\chi \gamma$.

- Let $(\sigma_1;\rho_1),\ldots,(\sigma_m;\rho_m),(\sigma';\rho') \overset{\infty}{\sqsubset} (\sigma;\rho), m \geq 1$. We say that $(\sigma';\rho')$ is an **alternate extension** of $(\sigma;\rho)$ w.r.t. $(\sigma_1;\rho_1),\ldots,(\sigma_m;\rho_m)$, denoted

$$\mathit{Alt}((\sigma;\rho),\{(\sigma_1;\rho_1),\ldots,(\sigma_m;\rho_m)\},(\sigma';\rho')),$$

  iff $\mathit{Dom}(\rho') = \mathit{Dom}(p_1) \cup \cdots \cup \mathit{Dom}(p_m)$ and there is a subset $S \subseteq \{1,\ldots,m\}$ such that:

  1. ρ' conflicts with p_i iff $i \in S$; and
  2. if $S \neq \{1,\ldots,m\}$ then $\mathit{Alt}(\sigma,\{\sigma_i \mid i \in \{1,\ldots,m\} \setminus S\},\sigma')$, while if $S = \{1,\ldots,m\}$ then $\sigma' = \sigma$.

- Suppose that $(\sigma_1;\rho_1),\ldots,(\sigma_m;\rho_m) \sqsubset (\sigma;\rho)$ and let β be any assumption base. We say that $(\sigma;\rho)$ **entails** $(\sigma_1;\rho_1),\ldots,(\sigma_m;\rho_m)$ **w.r.t.** β, denoted $(\sigma;\rho) \Vdash_\beta \{(\sigma_1;\rho_1),\ldots,(\sigma_m;\rho_m)\}$, iff for every $(\sigma';\rho')$ such that

$$\mathit{Alt}((\sigma;\rho),\{(\sigma_1;\rho_1),\ldots,(\sigma_m;\rho_m)\},(\sigma';\rho')),$$

the following holds for all χ:

$$I_{(\sigma';\rho')/\chi}\left(\bigwedge_{F \in \beta} F\right) = \textbf{false}.$$

Additionally, formalizing an argument into an instance of Vivid is potentially confusing, so we now provide an example of the process of formulating an instance of Vivid from an informal argument.

Consider the following scenario:

Three men (whom we refer to as m_1, m_2, and m_3) stand in a queue waiting to be served. Now if we consider what it means for some person to be in front of another, and given that m_1 is in front of m_2 and that m_2 is in front of m_3, it is obvious that m_1 will be served before m_3. We now demonstrate the process of formalization for this scenario.

We can formulate this in Vivid as follows:

To represent position, we use the attribute *position* which, in our scenario, can take on values in $\{1, 2, 3\}$. Additionally, we can represent what it means to be in front of another person with a relation; we call this relation R_1 and define it as follows:

$$R_1(p_i, p_j) \Leftrightarrow p_i < p_j \text{ where } R_1 \subseteq \mathit{position} \times \mathit{position}.$$

Then we have the attribute structure

$$\mathcal{A} = (\mathit{position} : \{1, 2, 3\}; \{R_1\})$$

Now, we define the vocabulary $\Sigma = (\{m_1, m_2, m_3\}, \{InFrontOf\}, \{\})$ and the attribute interpretation I of Σ into $\mathcal{A}$ as the mapping from the $InFrontOf$ relation symbol of arity 2 to the relation R_1 by defining the realization R_1 and profile $[(\mathit{position}, 1), (\mathit{position}, 2)]$ (in the case of multiple relation symbols, it is convenient to define an "interpretation table" which summarizes the mappings as quadruples of relation symbols, arities, realizations and profiles). The definitions for $\mathcal{A}, \Sigma$ and I form a formal instance of the Vivid language.

Now, consider the attribute system composed of $\mathcal{A}$ and three objects s_1, s_2, s_3; we can create a state to represent our scenario as follows:

$$\sigma : \mathit{position}(s_1) = \{1\}, \mathit{position}(s_2) = \{2\}, \mathit{position}(s_3) = \{3\}$$

or equivalently, as

$$\sigma(\mathit{position}, m_1) = \{1\}, \sigma(\mathit{position}, m_2) = \{2\}, \sigma(\mathit{position}, m_3) = \{3\}.$$

With the constant assignment ρ, mapping m_1 to s_1, m_2 to s_2 and m_3 to s_3, and assumption base containing the formulae $InFrontOf(m_1, m_2)$ and $InFrontOf(m_2, m_3)$ (that is, the assumption base $\beta = \{InFrontOf(m_1, m_2), InFrontOf(m_2, m_3)\}$) we then have the ingredients to create a context $\gamma = (\beta; (\sigma; \rho))$ for our scenario. Now, we can infer $InFrontOf(m_1, m_3)$ from the context γ by a simple application of the **observe** rule (with $F = InFrontOf(m_1, m_3)$) defined in the following section.

2.2 Formal Semantics

The two syntactic proof categories of the Vivid framework are sentential and diagrammatic; sentential deductions are used to derive formulae, while diagrammatic deductions derive diagrams. The letters D and Δ are used to denote sentential and diagrammatic deductions, respectively. The formal evaluation semantics of the framework are given by axioms and rules establishing judgments of the following form:

$$\gamma \vdash D \rightsquigarrow F$$

and

$$\gamma \vdash \Delta \rightsquigarrow (\sigma; \rho),$$

read as:

"In the context γ, deduction D (Δ) derives F (respectively, $(\sigma; \rho)$)".

Although both sentential and diagrammatic deductions are included in the framework, most of the deductions that fall into the sentential category are straightforward generalizations of the standard $\mathcal{NDL}$ semantics given in [Arkoudas, 2001a]. Indeed, for readers familiar with natural deduction in its traditional forms, sentential deduction in vivid will have a familiar feel. The only sentential forms not presented in $\mathcal{NDL}$ are **observe**, **cases by**, and $\Delta; D$. The semantics of the **observe** form are as follows:

$$\frac{}{(\beta; (\sigma; \rho)) \vdash \textbf{observe } F \rightsquigarrow F} \quad [Observe]$$
$$\text{provided that } I_{(\sigma;\rho)/\chi}(F) = \textbf{true} \text{ for all } \chi$$

Additionally, we briefly mention the **sentential-to-sentential** form of case reasoning. In the sentential-to-sentential form, first a disjunction $F_1 \vee F_2$ is noted to hold, then a formula G is shown to be entailed either way, entitling us to conclude G. The sentential-to-sentential form is captured syntactically as a rule application:

$$\textbf{cases } F_1 \vee F_2, F_1 \Rightarrow G, F_2 \Rightarrow G$$

and the semantics of the rule application is as follows:

$$\frac{}{(\beta \cup \{F_1 \vee F_2, F_1 \Rightarrow G, F_2 \Rightarrow G\}; (\sigma; \rho)) \vdash \textbf{cases } F_1 \vee F_2, F_1 \Rightarrow G, F_2 \Rightarrow G \rightsquigarrow G}$$

The semantics of **cases by** and $\Delta; D$ (as well as the semantics of the diagrammatic forms of the framework) are given below in Figures 2.1 and 2.2 respectively.

These formal semantics can all be understood in the same way; they all consider some context γ, some rule application, and either some derived named state or sentential form. For example, [*Absurdity*] can be understood as follows: Given some context $\gamma = (\beta; (\sigma; \rho))$, if some formula always evaluates to **false** in the named state of the context, then any arbitrary named state can be derived from γ. Intuitively, this captures the notion of a contradiction; if there is no world derivable from γ in which some formula contained in the assumption base of γ can be true, then we can claim anything from the context. As an additional example, consider the [$Widening$] rule. It can be understood as follows: given a context $\gamma = (\beta; (\sigma; \rho))$, we can derive any named state $(\sigma'; \rho')$ provided that the named state $(\sigma'; \rho')$ can be extended to the named state $(\sigma; \rho)$. Intuitively, we are effectively "losing" information about a particular diagram with this rule.

$$\frac{\begin{array}{c}(\beta \cup \{F_1, \ldots, F_k\}; (\sigma_1; \rho_1)) \vdash D_1 \rightsquigarrow F \\ \vdots \\ (\beta \cup \{F_1, \ldots, F_k\}; (\sigma_n; \rho_n)) \vdash D_n \rightsquigarrow F\end{array}}{\begin{array}{l}(\beta \cup \{F_1, \ldots, F_k\}; (\sigma; \rho)) \vdash \textbf{cases by } F_1, \ldots, F_k\text{: } (\sigma_1; \rho_1) \rightarrow D_1 \mid \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad \ldots \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad (\sigma_n; \rho_n) \rightarrow D_n \rightsquigarrow F \\ \text{provided } (\sigma; \rho) \Vdash\!\vdash_{\{F_1, \ldots, F_k\}} \{(\sigma_1; \rho_1), \ldots, (\sigma_n; \rho_n)\}\end{array}} \quad [C_3]$$

Figure 2.1: The semantics of diagrammatic-to-sentential case reasoning [Arkoudas and Bringsjord, 2009].

$$\frac{}{(\beta \cup \{F_1, \ldots, F_n\}; (\sigma; \rho)) \vdash (\sigma'; \rho') \textbf{ by thinning with } F_1, \ldots, F_n \rightsquigarrow (\sigma'; \rho') \text{ provided } (\sigma; \rho) \Vdash\!\vdash_{\{F_1, \ldots, F_n\}} (\sigma'; \rho')} \quad [\textit{Thinning}]$$

$$\frac{}{(\beta; (\sigma; \rho)) \vdash (\sigma'; \rho') \textbf{ by widening } \rightsquigarrow (\sigma'; \rho') \text{ provided } (\sigma; \rho) \sqsubseteq (\sigma'; \rho')} \quad [\textit{Widening}]$$

$$\frac{}{(\beta \cup \{\textbf{false}\}; (\sigma; \rho)) \vdash (\sigma'; \rho') \textbf{ by absurdity } \rightsquigarrow (\sigma'; \rho')} \quad [\textit{Absurdity}]$$

$$\frac{}{(\beta; (\sigma; \rho)) \vdash \textbf{claim } (\sigma; \rho) \rightsquigarrow (\sigma; \rho)} \quad [\textit{Diagram-Reiteration}]$$

$$\frac{\begin{array}{c}(\beta \cup \{F_1, \ldots, F_k\}; (\sigma_1; \rho_1)) \vdash \Delta_1 \rightsquigarrow (\sigma'; \rho') \\ \vdots \\ (\beta \cup \{F_1, \ldots, F_k\}; (\sigma_n; \rho_n)) \vdash \Delta_n \rightsquigarrow (\sigma'; \rho')\end{array}}{(\beta \cup \{F_1, \ldots, F_k\}; (\sigma; \rho)) \vdash \textbf{cases by } F_1, \ldots, F_k\text{: } (\sigma_1; \rho_1) \rightarrow \Delta_1 \mid \cdots \mid (\sigma_n; \rho_n) \rightarrow \Delta_n \rightsquigarrow (\sigma'; \rho') \text{ provided } (\sigma; \rho) \Vdash\!\vdash_{\{F_1, \ldots, F_k\}} \{(\sigma_1; \rho_1), \ldots, (\sigma_n; \rho_n)\}} \quad [C_1]$$

$$\frac{(\beta \cup \{F_1 \vee F_2, F_1\}; (\sigma; \rho)) \vdash \Delta_1 \rightsquigarrow (\sigma'; \rho') \quad (\beta \cup \{F_1 \vee F_2, F_2\}; (\sigma; \rho)) \vdash \Delta_2 \rightsquigarrow (\sigma'; \rho')}{(\beta \cup \{F_1 \vee F_2\}; (\sigma; \rho)) \vdash \textbf{cases } F_1 \vee F_2\text{: } F_1 \rightarrow \Delta_1 \mid F_2 \rightarrow \Delta_2 \rightsquigarrow (\sigma'; \rho')} \quad [C_2]$$

$$\frac{(\beta; (\sigma; \rho)) \vdash D \rightsquigarrow F \quad (\beta \cup \{F\}; (\sigma; \rho)) \vdash \Delta \rightsquigarrow (\sigma'; \rho')}{(\beta; (\sigma; \rho)) \vdash D; \Delta \rightsquigarrow (\sigma'; \rho')} \quad [D; \Delta]$$

$$\frac{(\beta; (\sigma; \rho)) \vdash \Delta \rightsquigarrow (\sigma'; \rho') \quad (\beta; (\sigma'; \rho')) \vdash D \rightsquigarrow F}{(\beta; (\sigma; \rho)) \vdash \Delta; D \rightsquigarrow F} \quad [\Delta; D]$$

$$\frac{(\beta; (\sigma; \rho)) \vdash \Delta_1 \rightsquigarrow (\sigma_1; \rho_1) \quad (\beta; (\sigma_1; \rho_1)) \vdash \Delta_2 \rightsquigarrow (\sigma_2; \rho_2)}{(\beta; (\sigma; \rho)) \vdash \Delta_1; \Delta_2 \rightsquigarrow (\sigma_2; \rho_2)} \quad [\Delta; \Delta]$$

$$\frac{(\beta; (\sigma; \rho)) \vdash D_1 \rightsquigarrow F_1 \quad (\beta \cup \{F_1\}; (\sigma; \rho)) \vdash D_2 \rightsquigarrow F_2}{(\beta; (\sigma; \rho)) \vdash D_1; D_2 \rightsquigarrow F_2} \quad [D; D]$$

$$\frac{(\beta \cup \{\exists\, x\, .\, F, F[z/x]\}; (\sigma; \rho)) \vdash \Delta[z/w] \rightsquigarrow (\sigma'; \rho')}{(\beta \cup \{\exists\, x\, .\, F\}; (\sigma; \rho)) \vdash \textbf{pick-witness } w \textbf{ for } \exists\, x\, .\, F \;\; \Delta \rightsquigarrow (\sigma'; \rho')} \quad [\textit{EI}/\Delta]$$
provided z is fresh

Figure 2.2: The formal semantics of diagrammatic deductions [Arkoudas and Bringsjord, 2009].

3. pyVivid: An Implementation

There were a few key considerations that went into the inception of the architecture for the implementation of the Vivid framework. These considerations ranged from the more immediate considerations like "which programming language should be used for the implementation?" to the more long term considerations like "what design paradigms should be used to ensure that the library can be continuously developed and modified to provide new and additional support and optimizations?". These questions inspired reflection over many different factors, but after a large amount of deliberation, a few select paradigms were chosen to guide the development of the pyVivid architecture, shown in figure 3.1. Among them, the most important were ease of use, extensibility and stability. The full pyVivid package can be found at `https://github.com/nickmarton/pyVivid` [Marton, 2015]. In what follows, we dive deeper into the paradigms used and choices made behind the implementation of pyVivid. Additionally, we provide an overview of the features included in the pyVivid library and explain how other programmers can easily extend the library itself for added functionality.

3.1 Why Python?

After due consideration, the architecture for the Vivid framework was written in the Python programming language for a select set of reasons. Python is an interpreted, object-oriented, high-level programming language with dynamic semantics. The syntax of Python is incredibly easy to learn and it emphasizes readability and therefore reduces the cost of program maintenance. Python also supports the use of so called modules and packages, the latter being analogous to libraries, to encourage modularity and code reuse. Additionally, the Python interpreter along with Python's extensive standard library are freely distributed for all major platforms [van Rossum, 1995].

One of the biggest reasons for the selection of the Python programming language as the programming language to use for the implementation of Vivid is the ease of use associated with Python; specifically, Python is easy to learn, easy to read and easy to maintain. In a framework as extensive as Vivid, these features are incredibly useful. The semantics of Vivid are very abstract and encoding them then should be done as concisely as possible as to avoid obfuscation. The syntax of Python, along with the fact that Python is dynamically typed, allows for code to be written in an exceptionally readable fashion; in some cases various code segments can even be represented roughly by natural language through the use of the built-in functions and operators of Python (e.g., membership can be determined through the use of the "`in`" operator). This readability is diminished in statically typed languages like C++, where verbosity (from the explicit declaration of types) is imposed necessarily by the syntax of the language. Furthermore, Vivid is targeted specifically for logicians; the average programmer would not be able to leverage the framework fully without potentially dedicating a substantial amount of time to learning a sufficient amount of background knowledge in the domain of formal logic. Additionally, as not all logicians are experienced programmers, the readability of the source code of the implementation

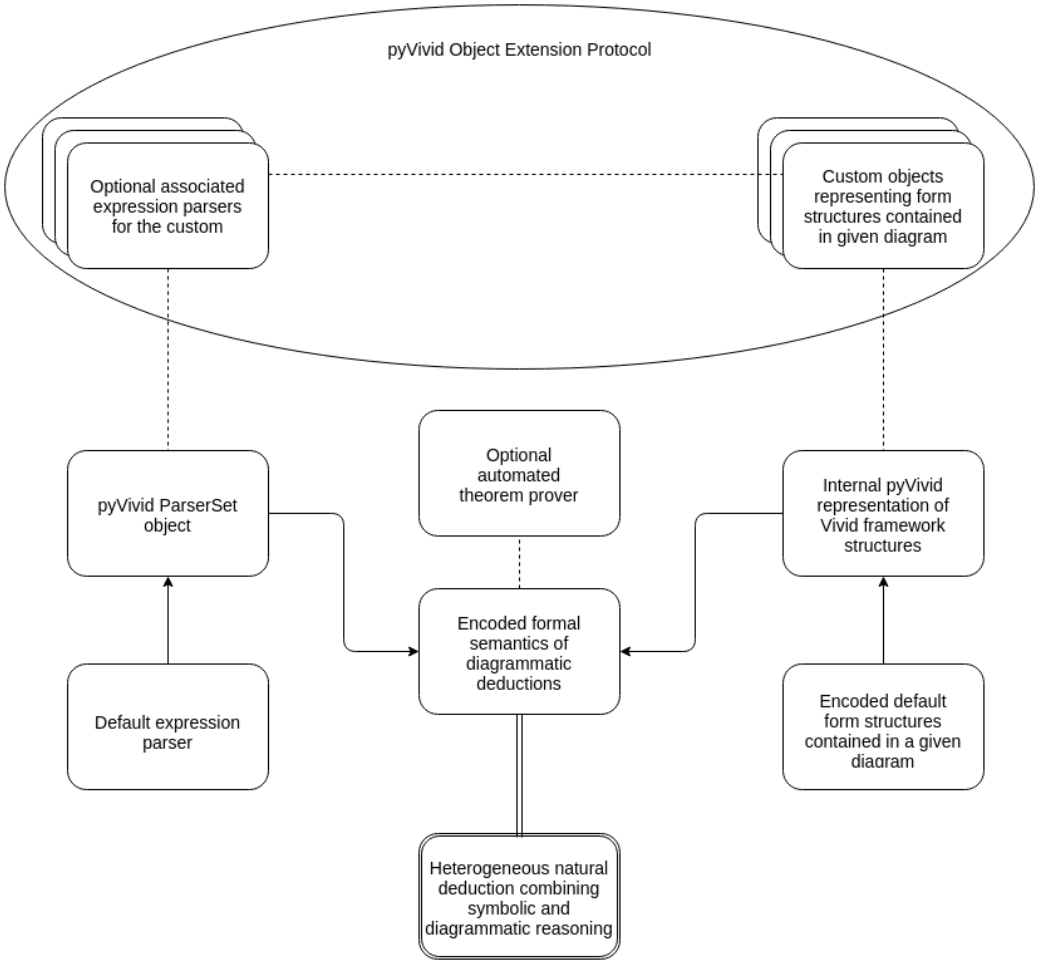

Figure 3.1: An overview of the architecture of pyVivid. Dashed lines represent optional behaviour flowing between components while solid lines represent necessary Behaviour flowing between components in the direction of the arrowhead. The POE Protocol, which we elaborate on later in this chapter, is wrapped in the oval at the top; this protocol is used to expand pyVivid's ability to assign truth values as well as add more supported types to pyVivid. As one can see towards the center of the figure, pyVivid has optional compatibility with automated theorem provers.

is that much more important to those who seek to extend the library.

Another major factor contributing to the decision of choosing Python as the programming language for the Vivid framework is the potential rate of development. While software written in Python is generally slower than other imperative, object-oriented languages, it can be develop at a much more accelerated rate and further, Python source code can be later compiled directly to C code via the Cython library [Behnel et al., 2011]. Statements in Python can also be written in a very compact way when the built-in functions and syntax are used properly. Additionally, since there is no compilation step, the edit-test-debug cycle is incredibly fast. Indeed, Python is a very attractive language for the purpose of Rapid Application Development (RAD), inspiring the creation of Python packages for RAD across many different domains (for example CherryPy for rapid Python web application development [Hellegouarch, 2007] and HTSeq for rapid de-

velopment of scripts which work with high-throughput sequencing data [Anders et al., 2014]). Put frankly, a full implementation of Vivid requires a sizable architecture and a great deal of code. Python, then, is a natural choice as a lot can be accomplished in a small amount of time; certainly, there is a lot to be done to implement Vivid in its entirety.

Unit testing is becoming more and more necessary in the software world. It finds problems early in the development cycle, facilitates change and simplifies integration. Numerous packages are also provided in Python for the purpose of testing the functionality of code. In fact, Python comes with a standard package for unit testing, the unittest package (more information for general unit testing within Python can be found in [Kuhlman, 2009]). Other external packages like pytest offer even more advanced testing functionality. The amount of external support played a big part in the decision to use Python. The pyVivid library comes equipped with an extensive set of unit tests, through the pytest package, for each and every function and class included therein, truly bolstering the stability and providing a safety net for source level modifications in the future.

3.2 An Overview of Available Features

The pyVivid library provides a concrete and fully tested implementation of the Vivid framework with an emphasis on both the semantic forms that fall under the diagrammatic category and the structures that allow for reasoning over incomplete information. As the library is substantial in size, an overview of the features that it provides is a useful summary to review before diving into the library. In this section, we provide an overview of the various features provided in pyVivid along with high level descriptions of the classes within.

3.2.1 Supported Types

At the heart of the pyVivid library is the ValueSet class. The ValueSet class provides support for many of the Python programming language's built-in types (e.g., integers, floats, longs, strings, etc.) as well as for other objects. It is through the instantiations of the ValueSet class (henceforth referred to as "ValueSets" or "ValueSet objects") that all of the underlying functionality of the library is provided; these objects act as containers for both the possible values of attributes and the values of the ascriptions of objects in named system states and are pervasive throughout many other areas of the library. In Python terminology, ValueSet objects are attributes of many of the other classes in the pyVivid architecture. Additionally, the ValueSet class has the ability to be extended to include arbitrary objects through the pyVivid Object Extension (POE) protocol explained in the following section. The Point class along with the LineSegment class are two such examples of the POE protocol put into practice.

3.2.2 Attributes and Relations

Instances of the Attribute and Relation classes (henceforth referred to as "Attributes" or "Attribute objects" and "Relations" or "Relation objects" respectively) form the building blocks of the more advanced structures used in the pyVivid library. Each Attribute object has an associated label which serves as the object's alias; these labels are used in numerous other locations

as well, for instance in the creation of the ascriptions of system states and named system states. Relation objects are equally important; the definitions of the Relation objects are used to create the expressions evaluated (after a few substitutions) when determining the truth value of a given formula; the assignment of a truth value to a formula is one of the most pivotal components of the pyVivid library as almost all of the diagrammatic inference rules make use of the assignment of truth values to formulas in one way or another.

3.2.3 Attribute Structures and Systems

The AttributeStructure class and AttributeSystem class in the pyVivid library implement the attribute structure and attribute system structures presented in the Vivid framework respectively. More formally, each instance of the AttributeStructure class (henceforth referred to as an "AttributeStructure" or "AttributeStructure object") is a pair $\mathcal{A} = (\{A_1, \ldots, A_k\}; \mathcal{R})$ consisting of a set of unique Attribute objects $\{A_1, \ldots, A_k\}$ and a set of unique Relation objects $\mathcal{R} = \{R_1, \ldots, R_l\}$; these AttributeStructure objects are used to create interpretations that enable us to interpret first-order signatures into named system states in the case of incomplete or missing information. Each instance of the AttributeSystem class (henceforth referred to as an "AttributeSystem" or "AttributeSystem object") is a pair $\mathcal{S} = (\{s_1, \ldots, s_n\}; \mathcal{A})$ consisting of a finite number $n > 0$ of objects $s_1, \ldots, s_n$ (maintained as a list of strings in Python terminology) and an AttributeStructure object $\mathcal{A}$. These AttributeSystem objects are required for the creation of instances of the State and NamedState classes, two classes required by the pyVivid library to perform diagrammatic deduction.

3.2.4 Vocabularies

The Vocabulary class provides the pyVivid library with first-order vocabularies consisting of a set of constant symbols, a set of relation symbols and a set of variables. The instances of the Vocabulary class, Vocabulary objects, are used in the creation of both constant and variable assignments as well as in the creation of attribute interpretations. In fact, changes to the Vocabulary object to which an object holds a reference to is even propagated to other objects that make use of the same Vocabulary object, even if the object holds that reference through another object like instances of the NamedState and Context classes do. As all conceivable use cases of pyVivid are for the verification of a proof (at least while Vivid remains a type-α DPL [Arkoudas, 2001a]), the library actively enforces the use of a single Vocabulary object throughout all objects involved in the verification process in an effort to reduce verbosity and increase ease of use. To that end, one can assume that a reference to the underlying Vocabulary object used during proof verification is never broken regardless of the intermediate functions being called or objects being created.

3.2.5 Constant and Variable Assignments

As mentioned in the previous section, formally, a constant assignment is a partial function ρ from the constants C of some vocabulary Σ to the objects $\{s_1, \ldots, s_n\}$ of some attribute system $\mathcal{S} = (\{s_1, \ldots, s_n\}; \mathcal{A})$ and a variable assignment is a total function χ from the variables V of some vocabulary Σ to the objects $\{s_1, \ldots, s_n\}$ of some attribute system $\mathcal{S} = (\{s_1, \ldots, s_n\}; \mathcal{A})$.

The ConstantAssignment and VariableAssignment classes implement these definitions respectively. Instances of the ConstantAssignment and VariableAssignment classes (referred to as "ConstantAssignments" or "ConstantAssignment objects" and "VariableAssignments" or "VariableAssignment objects" respectively) act as a single ρ or χ. These objects also hold a reference to the vocabulary (represented by a Vocabulary object) which they are defined over; in this way, ConstantAssignments and VariableAssignment objects are kept up-to-date about any changes made to the vocabulary (e.g. a constant symbol being added).

3.2.6 Named States

Instances of the NamedState class (henceforth referred to as "NamedStates" or "NamedState objects") are the principal structures for modelling incomplete information in the form of undetermined diagrams. Each NamedState object is a pair $(\sigma; \rho)$ consisting of an AttributeSystem object $\mathcal{S}$ (along with a State object σ of that AttributeSystem which is represented as a base class to the NamedState) and an instance of the ConstantAssignment class ρ. NamedState objects provide functionality for the more central operations involved in the semantics of diagrammatic deductions like determining entailment of other NamedState objects w.r.t. an assumption base and determining satisfaction of a given formula, named state, or context (provided that the NamedState object performing the computation is a world in the case of determining satisfaction). They also come equipped with the ability to generate the alternate extensions of themselves w.r.t. other NamedState objects as well as the ability to determine if a NamedState is a valid alternate extension of another. NamedState objects are also a principle component of the Context class, another class central to the library's ability to perform diagrammatic deductions. Additionally, NamedStates are capable of adding new objects (optionally mapped to a corresponding constant symbol in the underlying constant assignment) and ascriptions of those objects at any time; a useful feature for when more information about a particular diagram is learned and a constant symbol must dynamically come to denote an object during the course of deduction.

3.2.7 Attribute Interpretations

To interpret first-order languages into system states with pyVivid, it is necessary to create an instance of the AttributeInterpretation class first (henceforth referred to as an "AttributeInterpretation" or an "AttributeInterpretation object"). As mentioned by [Arkoudas and Bringsjord, 2009], attribute interpretations compile atomic formulae over system objects via profiles into atomic formulae over selected attribute values of some of (or possibly all of) those objects. The profiles specified dictate which attributes of which objects are selected in the compilation process. The AttributeInterpretation class is central to the pyVivid library; it is a prerequisite to almost every function acting to implement the semantics of diagrammatic deduction as it allows for the assignment of a truth value to a formula. Each AttributeInterpretation object stores a mapping of a Vocabulary object into an AttributeStructure object by creating a table with the pertinent information of the interpretation, including the profiles (which are specified at the time of construction). They also hold a reference to the Vocabulary object; this way, AttributeInterpretation objects are kept up to date with respect to any modifications made to

the underlying Vocabulary object.

3.2.8 Formulae and Assumption Bases

Instances of the Formula class (referred to as "Formulae" or "Formula objects") store a set of terms and are defined over some Vocabulary object and thus store a reference to that specific Vocabulary. Formula objects are used during the interpretation process where first-order languages are interpreted into system states; to that end, each Formula object can perform a function to assign a truth value to itself (with respect to an AttributeInterpretation object, NamedState object and VariableAssignment object). A more complete description of the truth value assignment process can be found in Chapter 10 of Appendix A. The instances of the AssumptionBase class (referred to as "AssumptionBase objects" or "AssumptionBases") are simply an implementation of the assumption base structure defined in the Vivid framework; that is, they are a finite set of Formula objects. AssumptionBases are one of the two components required to create a context and as such they play a big part in the implementation of the semantics of diagrammatic deduction.

3.2.9 Contexts

Context objects, instances of the Context class, are ubiquitous throughout the implementation of the semantics of diagrammatic deductions. Each Context object represents a single context structure; a pair $\gamma = (\beta; (\sigma; \rho))$ consisting of an assumption base β (given by an AssumptionBase object) and a named state $(\sigma; \rho)$ (given by an NamedState object). In an effort for all objects used in the proof verification process to remain consistent, one caveat is imposed during the creation of a Context object: the underlying Vocabulary of the AssumptionBase object must be the same Vocabulary as the underlying Vocabulary object of the NamedState object. As mentioned in section 2, the evaluation semantics of the Vivid framework requires the establishment of judgments of the following form:

"In the context γ, deduction $D(\Delta)$ derives F (respectively, $(\sigma; \rho)$)"

Thus, every function corresponding to some rule of diagrammatic deduction takes as a parameter a Context object. Additionally, Context objects are capable of determining whether or not a given Formula object or NamedState object is entailed by the Context object itself; in fact these functions aid in the process of performing diagrammatic deductions in multiple places.

3.2.10 Diagrammatic Deductions

The pyVivid library provides direct implementations for each of the rules of diagrammatic deduction of the Vivid framework. Specifically, the rules of $[Thinning]$, $[Widening]$, $[Absurdity]$, $[Diagram - Reiteration]$, $[C_1]$, $[C_2]$ and $[C_3]$ in Figures 2.1 and 2.2 all have corresponding functions in the `inference_rules` module (where $[C_1]$, $[C_2]$ and $[C_3]$ correspond to `diagrammatic_to_diagrammatic`, `sentential_to_diagrammatic` and `diagrammatic_to_sentential` respectively). Performing any of the four types of deduction sequencing rules, $[D; D]$, $[D; \Delta]$, $[\Delta; D]$ and $[\Delta; \Delta]$ is easily accomplished by simpling proceeding sequentially from the intermediate result as the proof verification is done in a procedural way; one can simply verify a step holds and thread the conclusion into the next function call.

It is worth mentioning that pyVivid does not provide a native automated theorem prover for purely symbolic expressions. Purely symbolic expressions, however, are contained solely by the definition attribute of Relation objects. For this reason, pyVivid allows for the exporting and importing of the definitions of a Relation object or the set of Relation objects contained in any pyVivid object corresponding to some form structure in Vivid. With this functionality, the $[EI/\Delta]$ rule of Figure 2.2 can be accomplished by exporting a given Relation object (corresponding to some formula F via an attribute interpretation), picking a valid witness through the use of an external theorem prover, then importing the witness back into a Relation object. Additional functions are provided (with respect to named states) for the $[Observe]$ and $[Sentential - to - Sentential]$ (the last of the 4 types of case reasoning) rules.

3.3 Extending pyVivid

One of the key paradigms guiding the development of the pyVivid library is extensibility. While many of the primitive built-in types of the Python programming language are supported by pyVivid, there are plenty of cases where primitive types are not enough. For these cases, the natural thing to do would be to use an object to represent what a primitive cannot. Engineering functions which account for each and every possible object would require code covering an intractably large amount of edge cases, obfuscating the source code immensely. While the results of the completely general solution are tempting, they are simply not feasible without jeopardizing both the readability and the extensibility of the library. However, with a small amount of effort on the programmer's part, support for an arbitrary object can be installed in pyVivid through the POE protocol. In the rest of this section, we examine the process of adding support for a class to pyVivid.

3.3.1 The POE Protocol

A few steps must be taken to ensure the correct installation of an object into the pyVivid library. Specifically, support for an object is added to the library through the ValueSet class, as ValueSet objects are used to dictate the different values an Attribute can take on and therefore the different values ascriptions of objects can take on in named system states. In what follows, we enumerate the steps of this process, which we dub the **POE protocol**. Additionally, the Point and LineSegment objects provide examples of the protocol in action.

1. Before support for the new class can be added to the ValueSet class, it is necessary to implement some of the so called "magic methods" of the new class; these functions can be identified by the leading and trailing double underscores (e.g. "`__init__`") and are not invoked directly. Specifically, the magic methods "`__deepcopy__`", "`__eq__`", "`__str__`" and "`__hash__`" must be implemented by the new class. These magic methods correspond to functions and operators being called behind the scenes for each supported type of the ValueSet class. In this way, no assumptions about the kind of objects that will be extending the ValueSet class are made and generality is preserved.

2. Next, an identifier must be added to the new class. This identifier should be of the form "`_is_`[class name]" where the name of the new class is substituted for [class name] (e.g. for

the Point class, the identifier is "_is_Point").

3. Now, the identifier of the new class must be added to the "_object_types" class attribute of the ValueSet class. This can be accomplished simply by adding the identifier directly to "_object_types" in the source code or by a call to the ValueSet class method "add_object_types" during run time. This is the last step required in all cases.

4. In some cases, the "__le__" magic method (representing the subset operator) and the "__sub__" magic method (representing the set theoretic difference operator) of the ValueSet class need to take into account more than simple equality. In these cases, these magic methods must be modified to support the new objects directly. For example, the Point class supports the notion of a generic point (i.e., the point has a defined dimension, but no defined coordinates). Then, a collection of Point objects with defined coordinates that share the same dimension as a generic Point object are a subset of that generic Point object and should be treated as such. Luckily, the extension of the "__le__" and "__sub__" magic methods is made easy by the usage of the class identifier (as both methods immediately split their respective ValueSet parameters by type). The subset of a generic point example given above can be found in the source code.

5. During the assignment of truth values, the typical expression being evaluated is composed of a combination of mathematical or logical operators (for a more detailed example of such an expression, refer to section 13.3 of Appendix A). However, sometimes this is not enough. For example, during the assignment of truth values which involve Point objects, those objects are required to call functions local to the Point class itself. To reconcile this, a "PointParser" is provided to the ParserSet object that is created during the assignment process. If the new objects are required to call functions local to their respective class, a parser class must be created for the object and added to the list of parsers defined in the constructor of the ParserSet class. The parser class for the new object must also implement the "__call__" magic method. The PointParser class and the respective unit tests of the class provides an example of how to implement a parser capable of calling functions local to the class of objects it is parsing for.

4. Examples

Recall that a concrete instance of Vivid is obtained by specifying a vocabulary Σ = (C,R,V), an attribute structure $\mathcal{A} = (l_1 : \{A_1, \ldots, l_k : A_k\}, \mathcal{R})$ and an interpretation I of $\mathcal{R}$ into $\mathcal{A}$. This is handled in pyVivid by constructing a Vocabulary object, AttributeStructure object and an AttributeInterpretation object, however each of these objects requires some additional work in order to be constructed properly. From there, the proof verification process begins via the use of the supported inference rules contained in pyVivid. We now turn to some examples to examine the process of proof verification in pyVivid sketched in other papers and clarify the usage of the library's internal classes, beginning with a simple toy proof taken from the original paper introducing Vivid [Arkoudas and Bringsjord, 2009].

4.1 A Simple Toy Proof

We now turn to the simple one line proof given by Example 13 in [Arkoudas and Bringsjord, 2009]. To paraphrase, consider a vocabulary Σ_{clock} = (C_{clock}, R_{clock}, V_{clock}) where the set of constant symbols is $\mathrm{C}_{clock} = \{\mathrm{c}_1, \mathrm{c}_2\}$, the set of variables is $\mathrm{V}_{clock} = \emptyset$, (i.e., the empty set) and the set of relation symbols is $\mathrm{R}_{clock} = \{Ahead\}$ where *Ahead* is binary. Now, consider the attribute structure

$$\mathcal{A}_{clock} = (hours : \{0, \ldots, 23\}, minutes : \{0, \ldots, 59\}; \{R_3\})$$

where $R_3 \subseteq hours \times minutes \times hours \times minutes$, defined as follows:

$$R_3(h_1, m_1, h_2, m_2) \Leftrightarrow h_1 > h_2 \vee (h_1 = h_2 \wedge m_1 > m_2).$$

We define an interpretation I_{clock} of Σ_{clock} into $\mathcal{A}_{clock}$ by specifying a unique relation and a unique profile for the *Ahead* relation symbol in R_{clock}. We set $Ahead^{I_{clock}} = R_3$ and

$$Prof(Ahead) = [(hours, 1), (minutes, 1), (hours, 2), (minutes, 2)].$$

Now consider a system of two clocks s_1 and s_2 with the attribute structure given by $\mathcal{A}_{clock}$. We define the constant assignment ρ by mapping the constant symbols c_1 and c_2 to the objects s_1 and s_2 respectively. Now let σ be the state depicted by the following picture:

Now suppose that we are given the premise $Ahead(c_1, c_2)$ indicating that the time of s_1 is ahead of the time of s_2. From the state σ and the aforementioned premise, the following diagram, called σ' should be able to be inferred:

Portions of this chapter previously appeared as: Sen, A., Bringsjord, S., Marton, N. and Licato, J., Toward diagrammatic automated discovery in axiomatic physics. in *2nd International Conference on Logic, Relativity and Beyond*, (Budapest, Hungary, 2015), Gergely Székely, 1-12.

Portions of this chapter have been submitted to: Sen, A., Peveler, M., Marton, N., Ghosh, R., Licato, J., Radke, R.J., Woodstock, T.K.A.E. and Bringsjord, S., Towards the cognitive classroom: mathematical physics. in *6th European Immersive Education Summit*, (Padua, Italy, 2016), Journal of Immersive Education, 1-14.

That is, we should be able to conclude the exact time of s_1, since, given that s_1 is ahead of s_2, the hour displayed by it cannot possibly be 4 or 5; it must, therefore, be 6. This can be done in Vivid with the following one-line proof:

$$(\sigma'; \rho) \textbf{ by thinning with } \mathit{Ahead}(c_1, c_2).$$

We now demonstrate how to verify this simple proof with the pyVivid library:

Before the proof can be constructed, it is necessary to define a formal instance of Vivid within pyVivid. To that end, first, Attribute objects corresponding to the *hours* and *minutes* attributes are constructed along with a Relation object corresponding to R_3 above. These objects are then fed into the constructor of an AttributeStructure object which we denote "`attribute_structure`". Next, a Vocabulary object corresponding to Σ_{clock}, which we denote "`vocabulary`", is constructed with the constant symbols c_1 and c_2 along with the *Ahead* relation symbol (via a RelationSymbol object). The AttributeInterpretation object, which we denote "`attribute_interpretation`" is then constructed after creating $Prof(\mathit{Ahead})$. This concrete instance of Vivid is shown in figure 4.1.

```
hours = vivid.Attribute("hours", [vivid.Interval(0, 23)])
minutes = vivid.Attribute("minutes", [vivid.Interval(0, 59)])
R3 = vivid.Relation(
    "R3(h1,m1,h2,m2) <=> h1 > h2 or (h1 = h2 and m1 > m2)",
    ["hours", "minutes", "hours", "minutes"], subscript=3)

attribute_structure = vivid.AttributeStructure(
    hours, minutes, R3)

Ahead = vivid.RelationSymbol("Ahead", 2)

vocabulary = vivid.Vocabulary(["c1", "c2"], [Ahead], [])

profiles = [
    [Ahead, ("hours", 1), ("minutes", 1), ("hours", 2),
     ("minutes", 2)]]

attribute_interpretation = vivid.AttributeInterpretation(
    vocabulary, attribute_structure, {Ahead: 3}, profiles)
```

Figure 4.1: Construction of a vivid instance in pyVivid for the simple toy proof outlined in section 4.1.

We proceed by creating the system of two clocks along with the constant assignment ρ and use them in the creation of the NamedState object $(\sigma; \rho)$, which we denote "`named_state`". We also create the NamedState object that we will be verifying, $(\sigma'; \rho)$, which we denote "`refined_named_state`". The last step is the application of the thinning rule, but for that we must first construct a Context object as thinning is performed in some context. We create a Context object using the `named_state` variable and a formula for *Ahead* represented by a

Formula object wrapped in an AssumptionBase object. Formally, we are creating the context $(\{Ahead(c_1, c_2)\}; (\sigma; \rho))$. Finally, we can verify that the named state $(\sigma'; \rho)$ is inferable by calling pyVivid's thinning function. This process is shown in Figure 4.2.

```
attribute_system = vivid.AttributeSystem(
    attribute_structure, ["s1", "s2"])

p = vivid.ConstantAssignment(vocabulary, attribute_system,
                             {"c1": "s1", "c2": "s2"})

ascriptions = {("hours", "s1"): [4, 5, 6], ("minutes", "s1"): [28],
               ("hours", "s2"): [5], ("minutes", "s2"): [45]}
named_state = vivid.NamedState(attribute_system, p, ascriptions)

refined_ascriptions = {("hours", "s1"): [6], ("minutes", "s1"): [28],
                       ("hours", "s2"): [5], ("minutes", "s2"): [45]}
refined_named_state = vivid.NamedState(attribute_system, p,
                                       refined_ascriptions)

ahead_formula = vivid.Formula(vocabulary, "Ahead", "c1", "c2")

context = vivid.Context(
    vivid.AssumptionBase(ahead_formula), named_state)

assert vivid.thinning(context,
                      refined_named_state,
                      vivid.AssumptionBase(ahead_formula),
                      attribute_interpretation)
```

Figure 4.2: Construction of the simple toy proof in pyVivid outlined in section 4.1.

4.2 The NEAT Theorem in Special Relativity

[Sen et al., 2015a] used pyVivid to formally prove Theorem 2.2 from Logic of Space-time and Relativity Theory [2] [Andréka et al., 2007] (abbreviated as "Theorem NEAT" where NEAT is an acronym for "No Event At Two Places"), which states that no observer observes the same event at two different space-time locations in models of the field and light axioms of special relativity (To our knowledge, this informal heterogeneous proof has never been verified begin this work). The full proof is rather verbose so we simply cover the construction of the concrete instance of the vivid language used therein by translating the attribute structure $\mathcal{A}$, vocabulary Σ and interpretation I of Σ into $\mathcal{A}$ into pyVivid code.

To reiterate, the attribute structure used in the proof of Theorem NEAT, which we denote $\mathcal{A}_{NEAT}$, is defined as follows:

$$\mathcal{A}_{NEAT} = (\{position : \mathbb{R}^n,\ slope : [0, 1],\ line\_positions : \mathbb{R}^{n^*}\}; R_1, R_2, R_3)$$

where the attributes of $\mathcal{A}_{NEAT}$ are *position*, *slope* and *line_positions* with ranges $\mathbb{R}^2$, $[0, 1]$, and $\mathbb{R}^{2^*}$ (where * is taken to mean the set of all lists of the elements of $\mathbb{R}^2$) respectively,

[2] This theorem states that no observer is able to observe the same event from two distinct space-time locations in models of the field and light axioms of special relativity [Sen et al., 2015a].

and the relations of $\mathcal{A}_{NEAT}$ are $R_1 \subseteq \mathbb{R}^2 \times \mathbb{R}^{2^*}$ defined as $R_1(p_1, [p'_1, ..., p'_k]) \Leftrightarrow p_1 \in [p'_1, ..., p'_k]$, $R_2 \subseteq \mathbb{R}^2 \times \mathbb{R}^2 \times \mathbb{R}^{2^*}$ defined as $R_2(p_1, p_2, [p'_1, ..., p'_k]) \Leftrightarrow p_1 = p_2 \vee (p_1 \in [p'_1, ..., p'_k] \wedge p_2 \in [p'_1, ..., p'_k])$, and $R_3 \subseteq [0, 1]$ defined as $R_3(s) \Leftrightarrow s = 1$.

The vocabulary of the proof of theorem NEAT, Σ_{NEAT}, consists of the constant symbols $\mathrm{C}_{NEAT} = \{x, y, m, l_1, l_2\}$, the relation symbols $\mathrm{R}_{NEAT} = \{through, observes, slope\_of\_one\}$ (where *through* and *observes* are binary and *slope_of_one* is unary) and the variables $\mathrm{V}_{NEAT} = \emptyset$ (i.e., the empty set).

Finally, the attribute interpretation I_{NEAT} of Σ_{NEAT} into A_{NEAT} used in the proof of the NEAT theorem is summarized in Table 4.1. The pyVivid code constructing this instance of Vivid is shown in Figure 4.3.

Table 4.1: The attribute interpretation I_{NEAT} used during the proof of NEAT.

Symbol	Arity	Realization	Profile
through	2	R_1	$[(position, 1), (line\_positions, 2)]$
observes	3	R_2	$[(position, 1), (position, 2), (line\_positions, 3)]$
slope_of_one	1	R_3	$[(slope, 1)]$

Before we proceed, there is something to note about this code excerpt. In the definition parameters of the first two Relation objects, the right hand sides of these definitions mimic python style syntax for calling a function. This is exactly the case and in fact during truth value assignment, these functions are called by the Point class through the PointParser object included in the ParserSet (more detail on this process can be found in Section 10.1 and 13.1 of Appendix A).

4.3 Clocks Move Out of Sync in Special Relativity

We now dive into a more extensive proof and showcase the visual explanatory power of the Vivid framework and the pyVivid library. Specifically, we will walk through a proof demonstrating that clocks move out of sync in the realm of axiomatic special relativity, first shown in [Sen et al., 2015b].

4.3.1 The Informal Proof

We begin with the diagram of Figure 4.12. At the origin, we assume that the clock of the stationary observer m_1 and that of the three moving observers, m_2, m_3 and m_4, are all set to 0. Hence $t = t' = 0$ where t and t' are the clocks in the stationary and moving frames of reference respectively. Thus, the clocks of m_1, m_2, m_3 and m_4 are all in sync.

Exactly at that moment, m_2 sends out a beam of light c_1 (where the speed of light c is represented visually by a slope with the absolute value of 1) in the direction of m_3, intersecting at point p. Figure 4.14 represents this (where additionally the point at the origin i_1 is labelled, which lies on the worldline of observer m_2 and shares its coordinates with the stationary observer m_1).

Now, coincidentally, the observer m_4 also sends out a beam of light c_2 towards m_3, which reaches m_3 at exactly the same moment as the light bean from m_2. Figure 4.16 represents this

```
position = vivid.Attribute("position", [vivid.Point("x", "x")])
slope = vivid.Attribute("slope", [vivid.Interval(0.0, 1.0)])
line_positions = vivid.Attribute(
    "line_positions",
    [vivid.LineSegment(vivid.Point("x","x"), vivid.Point("x","x"))])

R1 = vivid.Relation("R1(p1,l) <=> is_on(p1,l)",
    ["position", "line_positions"], subscript=1)
R2 = vivid.Relation("R2(p1,p2,l) <=> can_observe(p1,p2,l)",
    ["position", "position", "line_positions"],
    subscript=2)
R3 = vivid.Relation("R3(s) <=> s = 1", ["slope"], subscript=3)

attribute_structure = vivid.AttributeStructure(
    position, slope, line_positions, R1, R2, R3)

C, V = ["x", "y", "m", "l1", "l2"], []

through = vivid.RelationSymbol("through", 2)
observes = vivid.RelationSymbol("observes", 3)
slope_of_one = vivid.RelationSymbol("slope_of_one", 1)
R = [through, observes, slope_of_one]

vocabulary = vivid.Vocabulary(C, R, V)

profiles = [
    [through, ("position", 1), ("line_positions", 2)],
    [observes, ("position", 1), ("position", 2),
     ("line_positions", 3)],
    [slope_of_one, ("slope", 1)]]

mapping = {through: 1, observes: 2, slope_of_one: 3}

attribute_interpretation = vivid.AttributeInterpretation(
    vocabulary, attribute_structure, mapping, profiles)
```

Figure 4.3: Construction of a vivid instance in pyVivid for the NEAT proof in section 4.2.

new information; the position from which c_2 was sent out is labelled by both q and i_2. We do not know when and where m_4 sent out the light beam c_2, but we do know that the absolute value of the slope must be 1, so it is clearly here where it intersects with the worldline of m_4.

Solving for q yields the values $x = 2.\overline{66},\ t = 1.\overline{33}$, which is not equal to $(x = 2, t = 0)$. Then, we can reason that, since at $(2, 0)$, the clock of m_4 was in sync with the clock of m_2, and m_2's clock was in sync with m_1's clock (and they all read 0), and since a light beam sent out from some other point q from m_4 reached m_3 at the same location in space-time as a lightbeam sent out from m_2 at $t' = 0$, it must be that $t' = 0$ at q (that is, the time in the frame of reference of $m_2 - m_3 - m_4$ is 0 at q), but yet it is clearly not $t = 0$ at q (rather the time in the frame of reference of m_1 is $1.\overline{33}$, as shown above), and thus the clocks in the two reference frames are no longer in sync. Hence, moving clocks go out of sync.

4.3.2 Specifying an Instance of the Vivid Language

We now formulate an instance of Vivid for transforming the informal proof into a formal, machine-checkable one:

1. First, we provide an Attribute Structure $\mathcal{A}$:

   We choose the attributes *speed*, *spacetime_position*, and *worldline* with ranges $[0,1]$, $\mathbb{R}^2$, and $\mathbb{R}^{2^*}$ (where * is taken to mean the set of all lists of the elements of $\mathbb{R}^2$), respectively. Furthermore, we choose the relations

   (a) $R_1 \subseteq \mathbb{R}^2 \times \mathbb{R}^{2^*} \times \mathbb{R}^{2^*}$ defined as
   $R_1(sp, [p_1, ..., p_k], [p'_1, ..., p'_k]) \Leftrightarrow sp \in [p_1, ..., p_k] \wedge sp \in [p'_1, ..., p'_k]$,

   (b) $R_2 \subseteq [0,1]$ defined as $R_2(v) \Leftrightarrow v = 1$,

   (c) $R_3 \subseteq \mathbb{R}^2 \times \mathbb{R}^2$ defined as $R_3(sp, sp') \Leftrightarrow sp \neq sp'$,

   (d) $R_4 \subseteq [0,1] \times [0,1]$ defined as $R_4(v, v') \Leftrightarrow v = v'$ and

   (e) $R_5 \subseteq \mathbb{R}^2 \times \mathbb{R}^2$ defined as $R_5(t, t') \Leftrightarrow t \neq t'$, where t and t' represent the time coordinate of a space-time position.

   Then, in Vivid's native notation, our attribute structure is

   $$\mathcal{A} = (\{speed : [0,1],\ spacetime\_position : \mathbb{R}^2,\ worldline : \mathbb{R}^{2^*}\}; R_1, R_2, R_3, R_4, R_5),$$

   where R_1, R_2, R_3, R_4 and R_5 are defined as above.

2. Next, we provide a Vocabulary Σ = (C,R,V):

   We define C as $\{i_1, i_2, p, q, m_1, m_2, m_3, m_4, c_1, c_2\}$,

   R as the set containing:

   (a) $meets(sp, m, m')$, which holds when the worldlines (observers) m and m' intersect at the spacetime coordinate sp,

   (b) $speed\_of\_light(v)$, which holds when the (scaled) speed v is 1,

   (c) $not\_same\_location(sp, sp')$, which holds when spacetime coordinates sp and sp' are not equal,

   (d) $in\_same\_frame(v, v')$, which holds when velocities v and v' are equal, and

   (e) $clocks\_unequal(i, i')$, which holds when the times t and t' of two spacetime coordinates are not equal.

   and

   V as $\emptyset$, the empty set.

3. Lastly, we provide an interpretation I of Σ into $\mathcal{A}$. We summarize this interpretation in the compiled interpretation table given by Table 4.2.

Table 4.2: The attribute interpretation I used in the clocks move out of sync proof.

Symbol	Arity	Realization	Profile
$meets$	3	R_1	$[(spacetime\_position, 1),$ $(worldline, 2), (worldline, 3)]$
$speed\_of\_light$	1	R_2	$[(speed, 1)]$
$not\_same\_location$	2	R_3	$[(spacetime\_position, 1),$ $(spacetime\_position, 2)]$
in_same_frame	2	R_4	$[(speed, 1), (speed, 2)]$
$clocks\_unequal$	2	R_5	$[(spacetime\_position, 1),$ $(spacetime\_position, 2)]$

4.3.3 A Formal Proof in Vivid

Before proceeding with the formal proof, we first clarify what it means for clocks to be out of sync by defining additional definitions and a conjecture using the relations symbols defined in Σ above:

1. The in_sync definition defined as:

$$in\_same\_frame(v, v') \vee (meets(sp, m, m') \wedge \neg in\_same\_frame(v, v')) \vee (meets(i, m, c) \wedge meets(p, c, c') \wedge meets(i', c', m') \wedge speed\_of\_light(c) \wedge speed\_of\_light(c'))$$

   taken to mean informally that there are some coordinates where two observer's respective time (clock) components are in sync if the observers m and m' are in the same frame (that is, their respective velocities v and v' are equal), or if m and m' are not in the same frame but their respective worldlines meet at a certain coordinate sp, or if m sends out a beam of light c from coordinate i on its worldline and m' sends out a beam of light c' from coordinate i' on its worldline and c and c' meet at a certain coordinate p.

2. The $can\_observe\_photon\_at$ definition defined as:

$$meets(sp, c, m) \wedge speed\_of\_light(c)$$

   taken to mean informally that an observer m can observe a photon at sp if a beam of light c intersects the worldline of m at sp.

3. The out_of_sync conjecture defined as:

$$clocks\_unequal(i, i') \wedge in\_same\_frame(v, v') \vee (meets(sp, m, m') \wedge \neg in\_same\_frame(v, v')) \vee (meets(i, m, c) \wedge meets(p, c, c') \wedge meets(i', c', m') \wedge speed\_of\_light(c) \wedge speed\_of\_light(c'))$$

   which can be understood in the same way as the in_sync definition with the additional condition that the time (clock) components of i and i' are not equal.

The formal Vivid proof is given by Figure 4.4.

One important distinction must be made in the proof given by 4.4. When thinning is applied with the $can\_observe\_photon\_at$ definition, we are not considering $can\_observe\_photon\_at$ as a formula

Begin with the diagram Δ_0 (Figure 4.12)

Δ_1 (Figure 4.13) by $[Diagram - Reiteration]$
observe $(meets(i_1, m_1, m_2) \wedge \neg in\_same\_frame(v_1, v_2))$
observe $in\_same\_frame(v_2, v_3)$
observe $in\_same\_frame(v_2, v_4)$
observe $in\_same\_frame(v_3, v_4)$
Δ_2 (Figure 4.14) by $[Diagram - Reiteration]$
observe $(meets(p, c_1, m_3) \wedge speed\_of\_light(c_1)$
Δ_3 (Figure 4.15) by $[Thinning]$ with *can_observe_photon_at* definition
Δ_4 (Figure 4.16) by $[Diagram - Reiteration]$
observe $(meets(q, c_2, m_4) \wedge speed\_of\_light(c_2)$
Δ_5 (Figure 4.17 by $[Thinning]$ with the *can_observe_photon_at* definition
observe $not\_same\_location(q, q')$ ()where $q' = (2, 0))$
observe $not\_same\_location(p, q)$,
observe $(meets(i_1, m_2, c_1) \wedge meets(p, c_1, c_2) \wedge meets(i_2, c_2, m_4) \wedge speed\_of\_light(c_1) \wedge speed\_of\_light(c_2))$
observe $clocks\_unequal(i_1, i_2)$
disjunction introduction yields $in\_same\_frame(v_2, v_4) \vee (meets(p, m_2, m_4) \wedge \neg in\_same\_frame(v_2, v_4)) \vee (meets(i_1, m_2, c_1) \wedge meets(p, c_1, c_2) \wedge meets(i_2, c_2, m_4) \wedge speed\_of\_light(c_1) \wedge speed\_of\_light(c_2))$
conjunction introduction yields $clocks\_unequal(i_1, i_2) \wedge (in\_same\_frame(v_2, v_4) \vee (meets(p, m_2, m_4) \wedge \neg in\_same\_frame(v_2, v_4)) \vee (meets(i_1, m_2, c_1) \wedge meets(p, c_1, c_2) \wedge meets(i_2, c_2, m_4) \wedge speed\_of\_light(c_1) \wedge speed\_of\_light(c_2)))$
Then, the *out_of_sync* conjecture is satisfied and hence, clocks move out of sync.
□

Figure 4.4: The formal vivid proof of the clocks move out of sync conjecture [Sen et al., 2015b].

in the formulas $f_1, \ldots, f_n$ of the proviso of thinning. The *can_observe_photon_at* definition is satisfied by the preceding observation, and in the case of the applications of thinning above, we simply reiterate the preceding diagram.

4.3.4 Constructing the Proof in pyVivid

Now that a formal proof of clocks moving out of sync has been presented in an instance of the Vivid language, it can be implemented and verified in pyVivid. In what follows, the proof given above is reconstructed through the pyVivid library step by step:

The first step is to construct the concrete instance of Vivid from within pyVivid by creating an AttributeStructure object, a Vocabulary object and an AttributeInterpretation object in accordance with the attribute structure $\mathcal{A}$, vocabulary Σ, and attribute interpretation I, as defined above. This construction is given by figure 4.5. It is worth noting here that only the constants representing m_1, m_2, m_3 and m_4 are defined initially in the Vocabulary object, as during the course of deduction, a constant symbol can dynamically come to denote an object as more information is obtained about the diagram.

Before preceding any farther, we first define the LineSegment and Point objects corresponding to the points and line segments used in the formal proof. Additionally, we create the NamedState

```
spacetime_loc = vivid.Attribute(
    "spacetime_loc", [vivid.Point("x", "x")])
speed = vivid.Attribute("speed", [vivid.Interval(0.0, 1.0)])
worldline = vivid.Attribute(
    'worldline', [vivid.LineSegment(vivid.Point('x', 'x'),
                                    vivid.Point('x', 'x'))])

R1 = vivid.Relation("R1(sp, w1, w2) <=> meets(sp, w1, w2)",
                    ["spacetime_loc", "worldline", "worldline"],
                     subscript=1)
R2 = vivid.Relation("R2(v) <=> v = 1", ["speed"], subscript=2)
R3 = vivid.Relation("R3(sp1, sp2) <=> not_same_point(sp1, sp2)",
                    ["spacetime_loc", "spacetime_loc"], subscript=3)
R4 = vivid.Relation("R4(v1, v2) <=> v1 = v2",
                    ["speed", "speed"], subscript=4)
R5 = vivid.Relation("R5(sp1, sp2) <=> clocks_unequal(sp1, sp2)",
                    ["spacetime_loc", "spacetime_loc"], subscript=5)

attribute_structure = vivid.AttributeStructure(
    spacetime_loc, worldline, speed, R1, R2, R3, R4, R5)

meets = vivid.RelationSymbol("meets", 3)
speed_of_light = vivid.RelationSymbol("speed_of_light", 1)
not_same_location = vivid.RelationSymbol("not_same_spacetime", 2)
in_same_frame = vivid.RelationSymbol("in_same_frame", 2)
clocks_unequal = vivid.RelationSymbol("clocks_unequal", 2)

vocabulary = vivid.Vocabulary(
    ["m1", "m2", "m3", "m4"],
    [meets, speed_of_light, not_same_location, in_same_frame,
     clocks_unequal], [])

profiles = [
  [meets, ("spacetime_loc", 1), ("worldline", 2), ("worldline", 3)],
  [speed_of_light, ("speed", 1)],
  [not_same_location, ("spacetime_loc", 1), ("spacetime_loc", 2)],
  [in_same_frame, ("speed", 1), ("speed", 2)],
  [clocks_unequal, ("spacetime_loc", 1), ("spacetime_loc", 2)]]

attribute_interpretation = vivid.AttributeInterpretation(
    vocabulary, attribute_structure,
    {meets: 1, speed_of_light: 2, not_same_location: 3,
     in_same_frame: 4, clocks_unequal: 5}, profiles)
```

Figure 4.5: A formal vivid instance in pyVivid for the proof in section 4.3.

object corresponding to Δ_0 by first creating the AttributeSystem $\mathcal{S}$ (with objects corresponding to m_1, m_2, m_3 and m_4) the named state is derived from, along with the ConstantAssignment object ρ. These definitions and declarations are shown in figure 4.6.

From diagram Δ_0, we label the point at the origin i_1 which lies on the worldline of observer m_2 and shares its coordinate with the stationary observer m_1. This is done through the `add_object` function of the NamedState class, which provides support for adding an object to the system of the named state. We then use the diagrammatic rule [*Diagram − Reiteration*] to derive Δ_1 in Figure 4.13; the pyVivid function for the rule, `diagram_reiteration`, requires a Context object as input, so we wrap `delta_0` in a Context constructor with an empty AssumptionBase object

```
m1_segment = vivid.LineSegment(
    vivid.Point(0.0, 0.0), vivid.Point(0.0, 0.0))
m2_segment = vivid.LineSegment(
    vivid.Point(-1.0, -2.0), vivid.Point(3.0, 6.0))
m3_segment = vivid.LineSegment(
    vivid.Point(0.0, -2.0), vivid.Point(4.0, 6.0))
m4_segment = vivid.LineSegment(
    vivid.Point(1.0, -2.0), vivid.Point(5.0, 6.0))
c1_segment = vivid.LineSegment(
    vivid.Point(-2.0, -2.0), vivid.Point(6.0, 6.0))
c2_segment = vivid.LineSegment(
    vivid.Point(0.0, 4.0), vivid.Point(6.0, -2.0))
i1_pos = vivid.Point(0.0, 0.0)
p_pos = vivid.Point(2.0, 2.0)
q_pos = vivid.Point(2.6666666667, 1.3333333333)
i2_pos = vivid.Point(2.6666666667, 1.3333333333)
q_prime_pos = vivid.Point(2.0, 0.0)

objects = ["m1", "m2", "m3", "m4"]
attribute_system = vivid.AttributeSystem(
    attribute_structure, objects)
const_mapping = {"m1": "m1", "m2": "m2", "m3": "m3", "m4": "m4"}
p = vivid.ConstantAssignment(vocabulary, attribute_system,
                             const_mapping)

ascr = {
    ("worldline", "m1"): [m1_segment], ("speed", "m1"): [0.0],
    ("worldline", "m2"): [m2_segment], ("speed", "m2"): [1.0],
    ("worldline", "m3"): [m3_segment], ("speed", "m3"): [1.0],
    ("worldline", "m4"): [m4_segment], ("speed", "m4"): [1.0]}

delta_0 = vivid.NamedState(attribute_system, p, ascr)
```

Figure 4.6: Extra definitions and the declaration of Δ_0 used in the formal pyVivid proof corresponding to section 4.3.

before calling the function. Additionally, we proceed by using the **observe**[3] rule to extract sentential information from Δ_1. Specifically, we extract the first four observations in the proof given above:

observe $(meets(i_1, m_1, m_2) \wedge \neg in\_same\_frame(v_1, v_2))$,

observe $in\_same\_frame(v_2, v_3)$,

observe $in\_same\_frame(v_2, v_4)$, and

observe $in\_same\_frame(v_3, v_4)$.

Then the clocks of $m_1 - m_4$ are all in sync (in sync clocks satisfy transitivity). These observations are performed in the context of Δ_1 represented by the NamedState object `delta_1` by first constructing a Formula object for each observation, then applying the `observe` function in the pyVivid library (where we again wrap the NamedState object in a Context object). This process is shown in Figure 4.7.

Next, we claim observer m_2 sends out a beam of light c_1 from the origin which intersects

[3]This rule is used to extract sentential information from diagrams. This is similar to what we do when we just look at something.

```
delta_0.add_object("i1", {("spacetime_loc", "i1"): [i1_pos]},
                           constant_symbol="i1")
delta_1 = vivid.diagram_reiteration(
    vivid.Context(vivid.AssumptionBase(vocabulary), delta_0))

meets_i1_m1_m2 = vivid.Formula(vocabulary, "meets", "i1", "m1", "m2")
same_frame_v1_v2 = vivid.Formula(vocabulary,
                                 "in_same_frame", "m1", "m2")
same_frame_v2_v3 = vivid.Formula(vocabulary,
                                 "in_same_frame", "m2", "m3")
same_frame_v2_v4 = vivid.Formula(vocabulary,
                                 "in_same_frame", "m2", "m4")
same_frame_v3_v4 = vivid.Formula(vocabulary,
                                 "in_same_frame", "m3", "m4")

delta_1_context = vivid.Context(
    vivid.AssumptionBase(vocabulary), delta_1)

assert (vivid.observe(delta_1_context, meets_i1_m1_m2,
                      attribute_interpretation) and
        not vivid.observe(delta_1_context, same_frame_v1_v2,
                          attribute_interpretation))

assert vivid.observe(delta_1_context, same_frame_v2_v3,
                     attribute_interpretation)

assert vivid.observe(delta_1_context, same_frame_v2_v4,
                     attribute_interpretation)

assert vivid.observe(delta_1_context, same_frame_v3_v4,
                     attribute_interpretation)
```

Figure 4.7: The declaration of Δ_1 along with instances of the observe rule from within pyVivid.

m_3 at p and use $[Diagram - Reiteration]$ to derive Δ_2 (Figure 4.14). To proceed, we again make an observation: **observe** $meets(p, c_1, m_3) \wedge speed\_of\_light(c_1)$ where p is the point of intersection between c_1 and m_3. Further, we check if the $[Thinning]$ rule holds only after asserting that the *can_observe_photon_at* definition holds, then rename Δ_2 to Δ_3 through the $[Diagram - Reiteration]$ rule (Figure 4.15). These steps are shown in Figure 4.8.

Next, we claim observer m_4 sends out a beam of light c_2. We mark their intersection with q and denote the time at q in m_4's frame of reference as i_2 and use $[Diagram - Reiteration]$ to derive Δ_4 (Figure 4.16). We then make another observation: **observe** $meets(q, m_4, c_2) \wedge speed\_of\_light(c_2)$ where q is the point of intersection between c_2 and m_4. Further, we again perform $[Thinning]$ after asserting that the *can_observe_photon_at* definition holds, then rename Δ_4 to Δ_5 through the $[Diagram - Reiteration]$ rule (Figure 4.17). These steps are shown in figure 4.9.

```
delta_1.add_object("p", {("spacetime_loc", "p"): [p_pos]},
                   constant_symbol="p")
delta_1.add_object("c1", {("worldline", "c1"): [c1_segment],
                          ("speed", "c1"): [1.0]},
                   constant_symbol="c1")

delta_2 = vivid.diagram_reiteration(
    vivid.Context(vivid.AssumptionBase(vocabulary), delta_1))

meets_p_c1_m3 = vivid.Formula(vocabulary, 'meets', 'p', 'c1', 'm3')
speed_of_light_c1 = vivid.Formula(vocabulary, 'speed_of_light', 'c1')

delta_2_context = vivid.Context(
    vivid.AssumptionBase(vocabulary), delta_2)

assert (vivid.observe(delta_2_context, meets_p_c1_m3,
                      attribute_interpretation) and
        vivid.observe(delta_2_context, speed_of_light_c1,
                      attribute_interpretation))

assert vivid.thinning(delta_2_context, delta_2)

delta_3 = vivid.diagram_reiteration(delta_2_context)
```

Figure 4.8: Proceeding from Δ_1 to Δ_3 from within pyVivid.

Proceeding, the following four observations are made, shown below:

observe $not\_same\_location(q, q')$ where $q' = (2, 0)$ (that is, where m_4's clock reads 0),

observe $not\_same\_location(p, q)$,

observe $meets(i_1, m_2, c_1) \wedge meets(p, c_1, c_2) \wedge meets(i_2, c_2, m_4) \wedge$
$speed\_of\_light(c_1) \wedge speed\_of\_light(c_2)$,

observe $clocks\_unequal(i_1, i_2)$

```
delta_3.add_object("c2", {('worldline', 'c2'): [c2_segment],
                          ("speed", "c2"): [1.0]},
                   constant_symbol="c2")
delta_3.add_object("i2", {("spacetime_loc", "i2"): [i2_pos]},
                   constant_symbol="i2")
delta_3.add_object("q", {("spacetime_loc", "q"): [q_pos]},
                   constant_symbol="q")
delta_3.add_object("q_prime",
                   {("spacetime_loc", "q_prime"): [q_prime_pos]},
                   constant_symbol="q_prime")

delta_4 = vivid.diagram_reiteration(
    vivid.Context(vivid.AssumptionBase(vocabulary), delta_3))

meets_q_c2_m4 = vivid.Formula(vocabulary, 'meets', 'q', 'c2', 'm4')
speed_of_light_c2 = vivid.Formula(vocabulary, 'speed_of_light', 'c2')

delta_4_context = vivid.Context(
    vivid.AssumptionBase(vocabulary), delta_4)

assert (vivid.observe(delta_4_context, meets_q_c2_m4,
                      attribute_interpretation) and
        vivid.observe(delta_4_context, speed_of_light_c2,
                      attribute_interpretation))

assert vivid.thinning(delta_4_context, delta_4)

delta_5 = vivid.diagram_reiteration(delta_4_context)
```

Figure 4.9: Proceeding from Δ_3 to Δ_5 from within pyVivid.

In the previous set of observations, the third assertion satisfies the *in_sync* definition and thus

$$in\_same\_frame(v_2, v_4) \vee (meets(p, m_2, m_4) \wedge \neg in\_same\_frame(v, v')) \vee (meets(i_1, m_2, c_1) \wedge meets(p, c_1, c_2) \wedge meets(i_2, c_2, m_4) \wedge speed\_of\_light(c_1) \wedge speed\_of\_light(c_2))$$

holds from `disjunction introduction`. We determine the truth value of this disjunction, then, using the standard `conjunction introduction` rule with this and the last of the previous observations, we obtain

$$clocks\_unequal(i_1, i_2) \wedge (in\_same\_frame(v_2, v_4) \vee (meets(p, m_2, m_4) \wedge \neg in\_same\_frame(v, v)) \vee (meets(i_1, m_2, c_1) \wedge meets(p, c_1, c_2) \wedge meets(i_2, c_2, m_4) \wedge speed\_of\_light(c_1) \wedge speed\_of\_light(c_2)))$$

Hence, the *out_of_sync* conjecture is satisfied, and thus, clocks move out of sync; these concluding steps are shown in Figure 4.11.

□

```
meets_i1_m2_c1 = vivid.Formula(vocabulary, 'meets', 'i1', 'm2', 'c1')
meets_p_c1_c2 = vivid.Formula(vocabulary, 'meets', 'p', 'c1', 'c2')
meets_i2_c2_m4 = vivid.Formula(vocabulary, 'meets', 'i2', 'c2', 'm4')
not_same_loc_q_qprime = vivid.Formula(
    vocabulary, 'not_same_spacetime', 'q', 'q_prime')
not_same_loc_p_q = vivid.Formula(
    vocabulary, 'not_same_spacetime', 'p', 'q')
clocks_unequal_i1_i2 = vivid.Formula(
    vocabulary, 'clocks_unequal', 'i1', 'i2')

delta_5_context = vivid.Context(
    vivid.AssumptionBase(vocabulary), delta_5)

assert vivid.observe(delta_5_context, not_same_loc_q_qprime,
                     attribute_interpretation)

assert vivid.observe(delta_5_context, not_same_loc_p_q,
                     attribute_interpretation)

o_meets_i1_m2_c1 = vivid.observe(
    delta_5_context, meets_i1_m2_c1, attribute_interpretation)
o_meets_p_c1_c2 = vivid.observe(
    delta_5_context, meets_p_c1_c2, attribute_interpretation)
o_meets_i2_c2_m4 = vivid.observe(
    delta_5_context, meets_i2_c2_m4, attribute_interpretation)
o_speed_of_light_c1 = vivid.observe(
    delta_5_context, speed_of_light_c1, attribute_interpretation)
o_speed_of_light_c2 = vivid.observe(
    delta_5_context, speed_of_light_c2, attribute_interpretation)

assert (meets_i1_m2_c1 and meets_p_c1_c2 and meets_i2_c2_m4 and
        o_speed_of_light_c1 and o_speed_of_light_c2)

o_clocks_unequal_i1_i2 = vivid.observe(
    delta_5_context, clocks_unequal_i1_i2, attribute_interpretation)
assert o_clocks_unequal_i1_i2
```

Figure 4.10: The last observations of the proof of section 4.3 from within pyVivid.

```
meets_p_m2_m4 = vivid.Formula(vocabulary, 'meets', 'p', 'm2', 'm4')

o_meets_p_m2_m4 = vivid.observe(delta_5_context, meets_p_m2_m4,
                                attribute_interpretation)
o_same_frame_v2_v4 = vivid.observe(delta_5_context, same_frame_v2_v4,
                                   attribute_interpretation)

disjunction = (o_same_frame_v2_v4 or
               (o_meets_p_m2_m4 and not o_same_frame_v2_v4) or
               (o_meets_i1_m2_c1 and o_meets_p_c1_c2 and
                o_meets_i2_c2_m4 and o_speed_of_light_c1 and
                o_speed_of_light_c2))
assert disjunction

out_of_sync = o_clocks_unequal_i1_i2 and disjunction
assert out_of_sync
```

Figure 4.11: The last steps of the proof of section 4.3 from within pyVivid.

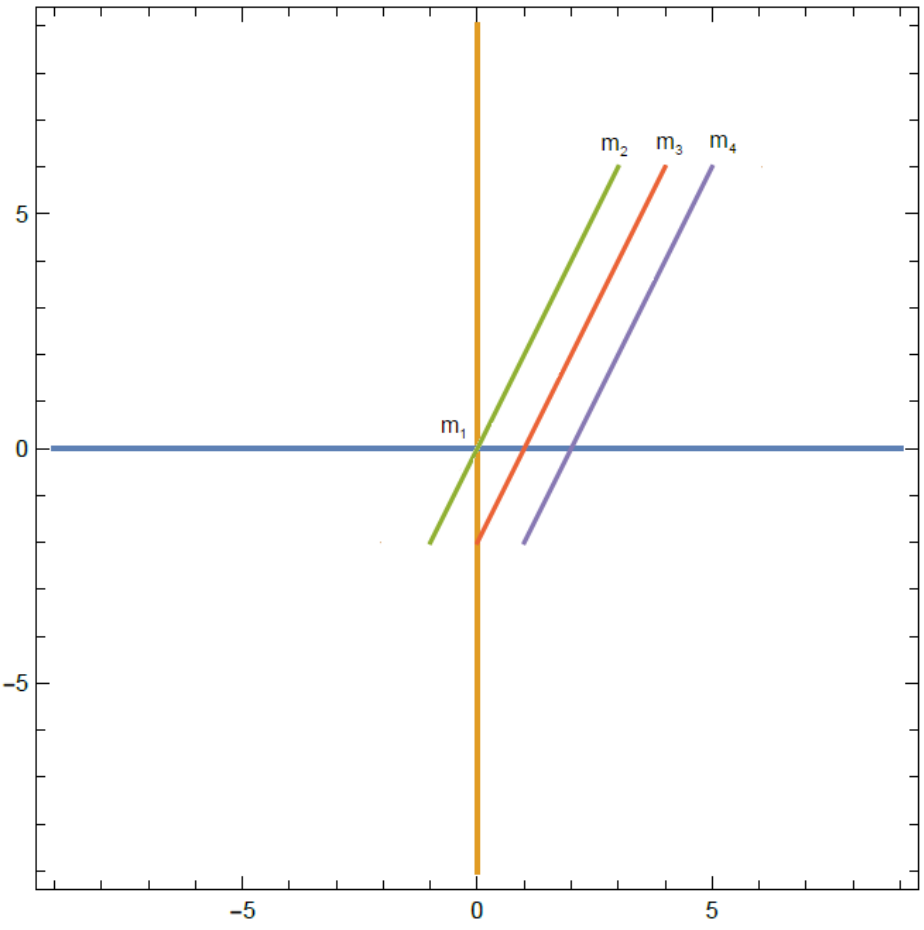

Figure 4.12: A pictorial representation of the named state Δ_0 [Sen et al., 2015b].

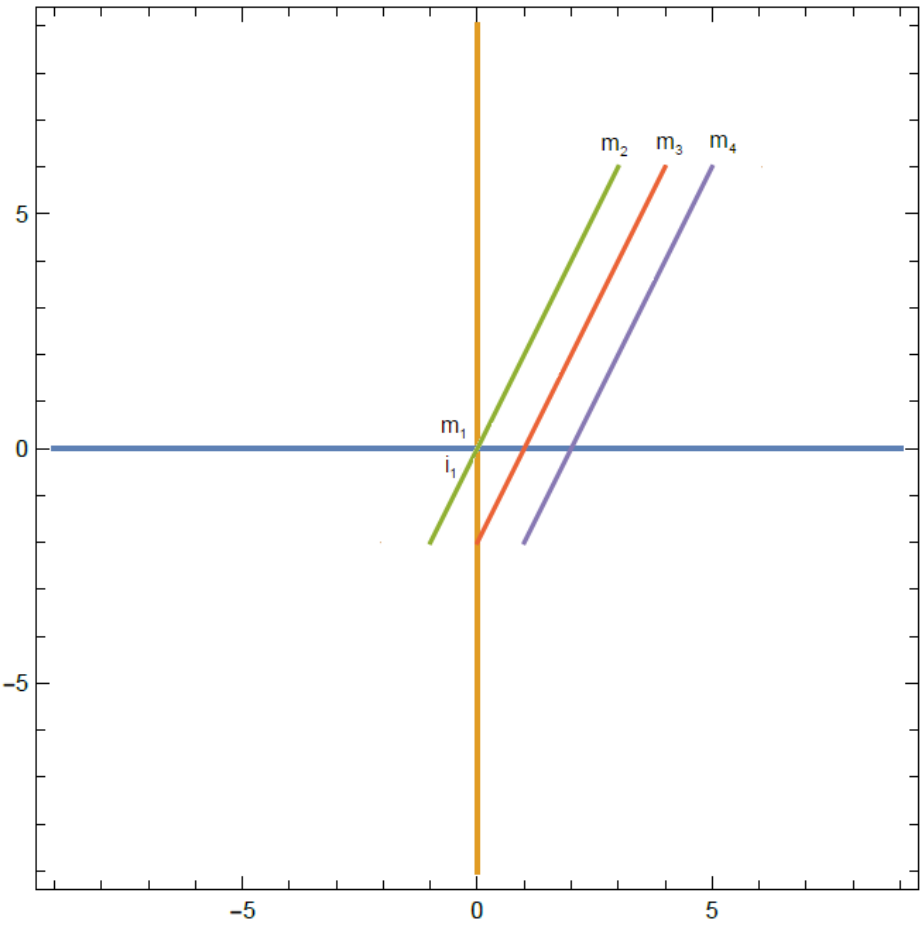

Figure 4.13: A pictorial representation of the named state Δ_1 [Sen et al., 2015b].

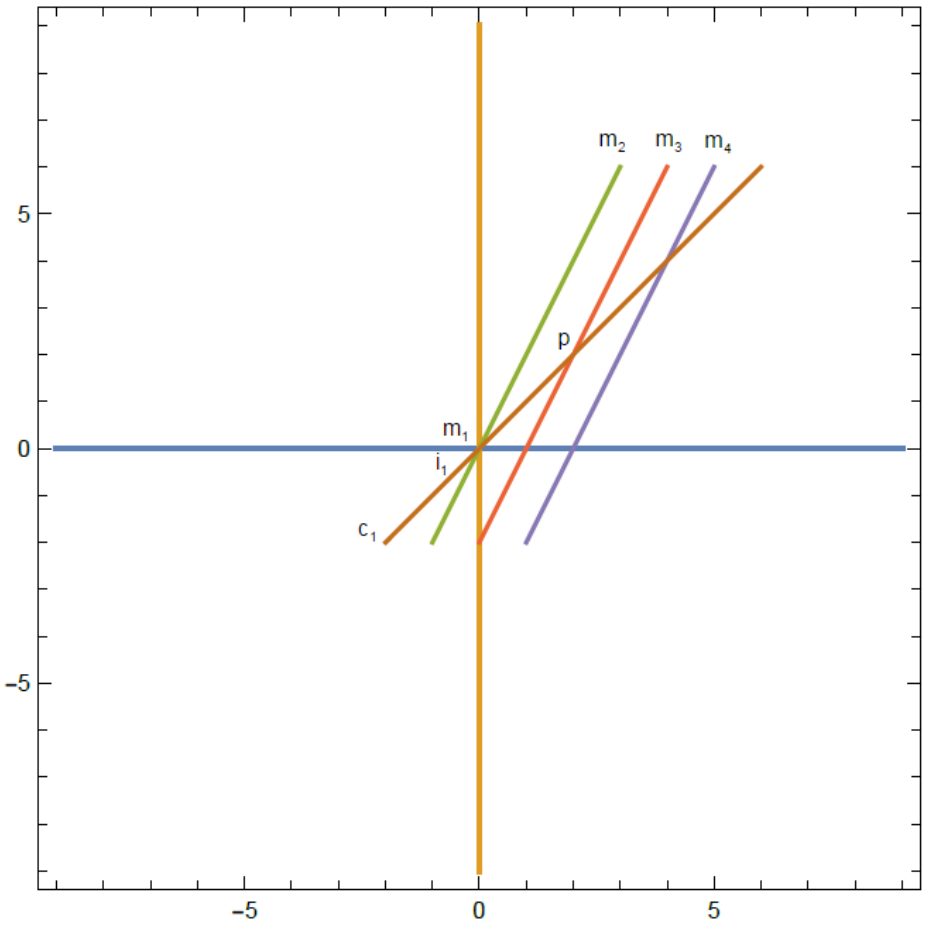

Figure 4.14: A pictorial representation of the named state Δ_2 [Sen et al., 2015b].

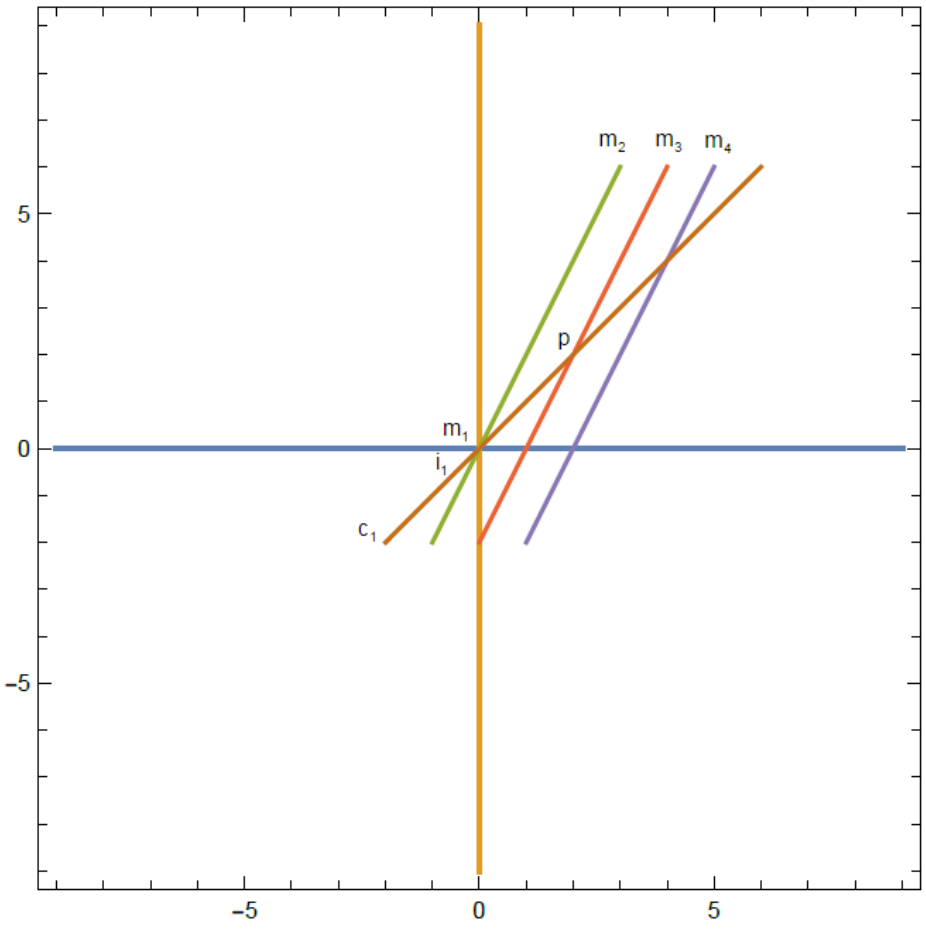

Figure 4.15: A pictorial representation of the named state Δ_3 [Sen et al., 2015b].

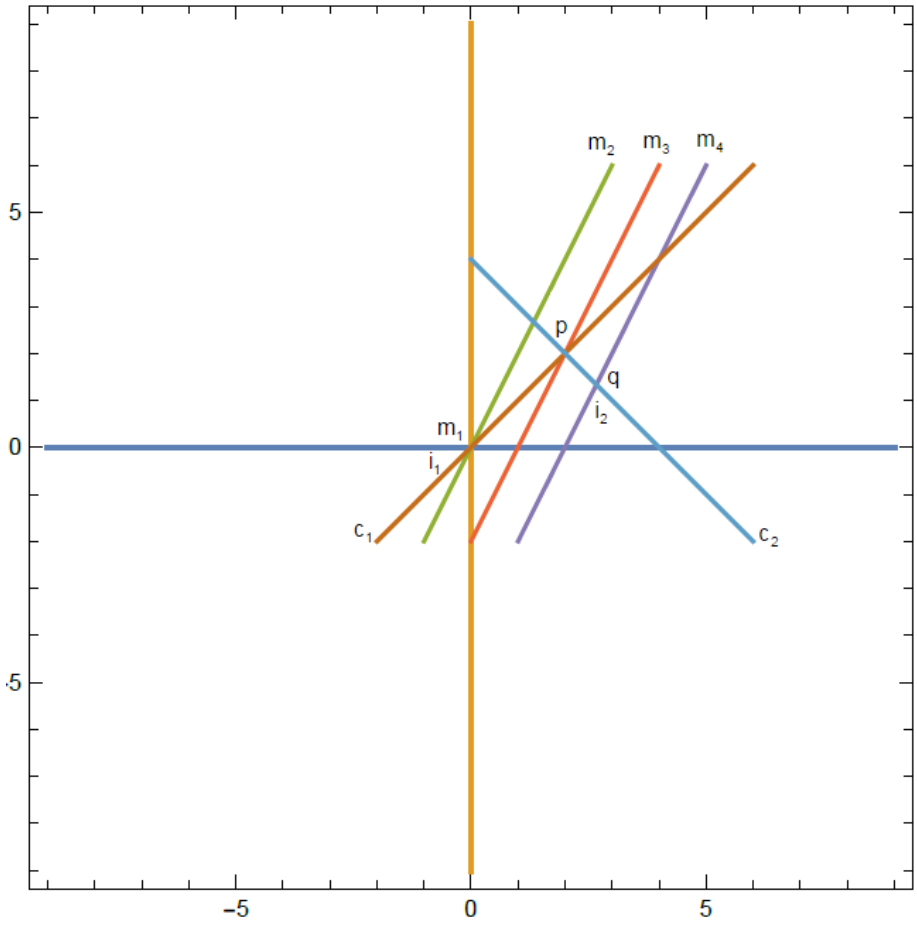

Figure 4.16: A pictorial representation of the named state Δ_4 [Sen et al., 2015b].

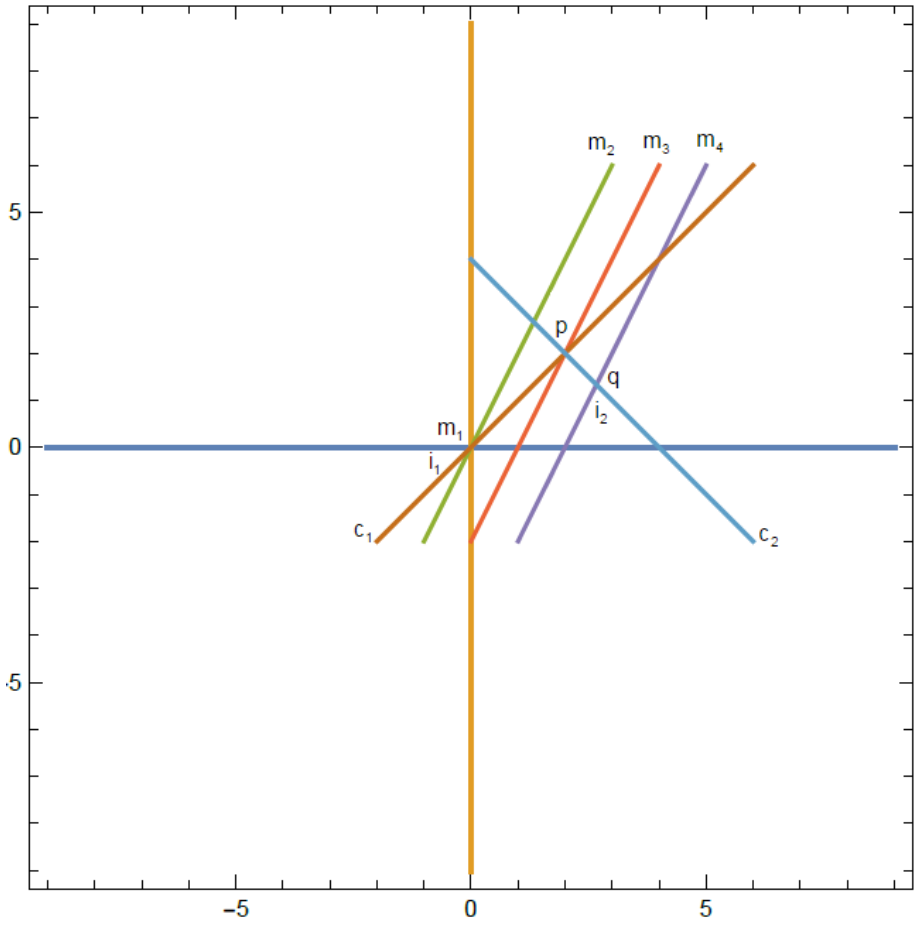

Figure 4.17: A pictorial representation of the named state Δ_5 [Sen et al., 2015b].

5. Conclusion

Paraphrasing [Arkoudas and Bringsjord, 2009], a tremendous amount of software is provided today for the purpose of symbolic reasoning: automated theorem provers, proof verification systems, model finders, planners and other tools exist in a multitude of different programming languages. Furthermore, symbolic reasoning is still an ongoing area of research and much of the software implementing the tools of symbolic reasoning are still being developed and used across a multitude of fields such as robotics, linguistics and software verification. Despite many researchers realizing the advantages of the analogical representations that diagrams provide, little research has been done towards the formalization of the diagrammatic inference, particularly with respect to the area of formal proofs. Even less has been accomplished in the way of a concrete system for diagrammatic reasoning. The pyVivid library works to remedy this huge discrepancy. It is the first public library capable of domain-independent heterogeneous reasoning which freely combines symbolic and diagrammatic logic. Furthermore, pyVivid was designed with ease of use in mind, targeting a large group of logicians and programmers.

As one of the key usages of the Vivid framework is dealing with incomplete information, great effort was applied to representing incomplete information easily and sufficiently, resulting in NamedState objects within the pyVivid library. These objects make it quick and easy to gradually eliminate uncertainty and the inherent objects of the library allow for the combining of diagrammatic and symbolic reasoning in an uncomplicated, procedural way, allowing for the derivation of new information. Additionally, pyVivid makes no assumptions on the particular class of diagrams being used for proof verification and thus none of the functionality behind the library is obfuscated by particulars; remaining general in the application of the functions corresponding to the semantics of diagrammatic deductions is one of the major goals of pyVivid.

One next step for pyVivid is the development of a graphical interface for the pyVivid library. An interface capable of displaying pictorial representations of the diagrams, exactly as they appear, and performing diagrammatic deductions over these representation would extend the audience of pyVivid to an even larger group. An effort will also be made to expand pyVivid's ability to incorporate more general constructs. Generality has not been explored very much from within pyVivid and exploration of the subject could lead to some very interesting results. Specifically, the incorporation of recursive and infinite structures would certainly expand pyVivid's domain of representable proofs. With the right choice of representations, it may be possible to represent infinite structures from within the current version of pyVivid, but our work has been restricted to finite cases. Computational tractability w.r.t. representing and reasoning over these (possibly) infinite structures must also be investigated; luckily, the POE protocol can be leveraged to perform these tests. Additionally, the introduction of unrestricted naming and computation conventions would extend pyVivid into a type-ω DPL [4] framework [Arkoudas, 2001b], capable of proof search.

[4] Type-ω Denotational Proof Languages are languages for both sound proof presentation and sound proof searching [Arkoudas, 2001b].

BIBLIOGRAPHY

Anders, S., Pyl, P.T. and Huber, W. Htseq - a python framework to work with high-throughput sequencing data. *Bioinformatics*, *31* (2), 166–169.

Andréka, H., Madarász, J.X. and Németi, I. Logic of space-time and relativity theory. in Aiello, M., Pratt-Hartmann, I. and Van Benthem, J. eds. *Handbook of Spatial Logics*. Springer Netherlands, Dordrecht, 2007, 607-711.

Arkoudas, K. Type-α DPLs. *AI Memo*, *2001* (25), 1–2.

Arkoudas, K. Type-ω DPLs. *AI Memo*, *2001* (27), 1–4.

Arkoudas, K. and Bringsjord, S. Vivid: an ai framework for heterogeneous problem solving. *Artificial Intelligence*, *173* (15), 1367–1405.

Behnel, S., Bradshaw, R., Citro, C., Dalcin, L., Seljebotn, D.S. and Smith, K. Cython: the best of both worlds. *Computing in Science & Engineering*, *13* (2), 31–39.

Berardi, D., Calvanese, D. and De Giacomo, G. Reasoning on uml class diagrams. *Artificial Intelligence*, *168* (1), 70–118.

Fitch, W.M. and Margoliash, E. Construction of phylogenetic trees. *Science*, *155* (3760), 279–284.

Gastwirth, J.L. A general definition of the lorenz curve. *Econometrica*, *39* (6), 1037–1039.

Hellegouarch, S. Cherrypy essentials: rapid python web application development. Packt Publishing Ltd, Birmingham, 2007.

Jordan, M.I. Learning in graphical models. Springer Netherlands, Dordrecht, 1998.

Kuhlman, D. A python book: beginning python, advanced python, and python exercises. ReadCycle, Cincinnati, 2009.

Marton, N. Pyvivid - a concrete package for heterogeneous natural deduction that combines diagrammatic and symbolic reasoning. Retrieved May 2, 2016 from https://github.com/nickmarton/pyVivid.

Parker, L. and Schmieg, G.M. A useful form of the minkowski diagram. *American Journal of Physics*, *38* (11), 1298–1302.

Sen, A., Bringsjord, S., Marton, N. and Licato, J., Toward diagrammatic automated discovery in axiomatic physics. in *2nd International Conference on Logic, Relativity and Beyond*, (Budapest, Hungary, 2015), Gergely Székely, 1-12.

Sen, A., Peveler, M., Marton, N., Ghosh, R., Licato, J., Radke, R.J., Woodstock, T.K.A.E. and Bringsjord, S., Towards the cognitive classroom: mathematical physics. in *6th European Immersive Education Summit*, (Padua, Italy, 2016), Journal of Immersive Education, 1-14.

Van Rossum, G. The python tutorial. Retrieved May 2, 2016 from https://docs.python.org/2/tutorial/.

Walley, P. Measures of uncertainty in expert systems. *Artificial Intelligence*, *83* (1), 1–58.

APPENDIX A
pyVivid Documentation

The pyVivid package provides a tremendously large code base with an architecture spanning over 20 separate files. Despite best efforts to provide self-documenting code in pyVivid, the underlying abstract framework of Vivid is extremely dense and developing a complete understanding of the mechanics behind it takes time and practice, even for experienced logicians. Therefore, an in depth explanation of the functionality of the architecture is needed to fully leverage pyVivid. Appendix A provides comprehensive documentation of the functionality behind each component involved in the pyVivid architecture, along with an abstract description of the more complex functions therein in the native notation of the Vivid framework.

A1 Basic Components

A1.1 The Interval object

This subsection introduces the Interval class.

class `interval.`**`Interval`**(*inf*, *sup*)

Intervals are over natural or real values.

Variables

- **infimum** – The infimum of the interval.
- **supremum** – The supremum of the interval.
- **type** – The type of the Interval object (int, float, or long).
- **_is_Interval** – An identifier to use in place of `type` or `isinstance`.

`__and__`(*other*)

Overloaded `&` operator; return the intersection of two Interval objects.

Raises ValueError Intervals must overlap to take the intersection.

`__contains__`(*key*)

Determine if the calling Interval contains an int, float, long, or another Interval.

Parameters key (*int|float|long|Interval*) – The value to check for membership in the calling Interval.

`__deepcopy__`(*memo*)

Deepcopy an Interval object via the `copy.deepcopy` method.

`__eq__`(*other*)

Determine if two Interval objects are equal via the == operator.

`__ge__`(*other*)

Overloaded >= operator for Interval. Determine if the calling Interval is greater than another Interval; that is, the infimum of the Interval in `other` parameter is strictly less than the calling Interval's infimum, and the supremum of the interval in `other` parameter is less than the calling Interval's supremum: $(o_{inf}, (s_{inf}, o_{sup}), s_{sup})$.

`__getitem__`(*index*)

Retrieve the infimum or supremum of the calling Interval via indexing (e.g. `Interval[0]`).

Raises

- **IndexError** – Index must be either "`0`" or "`1`".
- **TypeError** – Index must be an `int`.

`__gt__`(*other*)

Overloaded > operator for Interval. Determine if the calling Interval is strictly greater than another interval; that is, the infimum of the calling Interval is strictly greater than the supremum of the Interval in `other` parameter: $(o_{inf}, o_{sup})(s_{inf}, s_{sup})$.

`__hash__`()

Hash implementation for set functionality of Interval objects.

`__init__`(*inf*, *sup*)

Construct an Interval object.

Parameters

- **inf** (*int|float|long*) – The value to use as the infimum of the Interval.
- **sup** (*int|float|long*) – The value to use as the supremum of the Interval.

Raises

- **ValueError** – The infimum must be strictly less than the supremum.
- **TypeError** – The infimum and supremum provided must be ints, floats, or longs and their types must match.

__le__(*other*)
Overloaded <= operator for Interval. Determine if the calling Interval is less than another Interval; that is, the supremum of the calling Interval is greater than the Interval in `other`'s infimum, less than the Interval in `other`'s supremum and the infimum of the calling Interval is strictly less than the infimum of the Interval in `other` parameter: $(s_{inf}, (o_{inf}, s_{sup}), o_{sup})$.

__lt__(*other*)
Overloaded < operator for Interval. Determine if the calling Interval is strictly less than another interval; that is, the supremum of the calling Interval is strictly less than the infimum of the Interval in `other` parameter: $(s_{inf}, s_{sup})(o_{inf}, o_{sup})$.

__ne__(*other*)
Determine if two Interval objects are not equal via the `!=` operator.

__or__(*other*)
Overloaded `|` operator; return the union of two Interval objects.

Raises ValueError Intervals must overlap to take the union.

__repr__()
Return a string representation of the Interval object.

__str__()
Return a readable string representation of the Interval object.

_key()
Private key function for hashing.

Returns 2-tuple consisting of (infimum, supremum).

Return type `tuple`

static collapse_intervals(*intervals*)
Collapse a list of overlapping intervals.

Parameters intervals (*list*) – A list of intervals to collapse.

Returns A new list of totally disjoint, collapsed Intervals.

Return type `list`

Raises TypeError `interval` parameter must be a `list` containing only Interval objects.

discretize(*jump=None*)
Return all values within the range of the calling interval.

Parameters jump (*None|int|float|long*) – The jump to use after each value. Defaults to `1`, `1.0` and `1` for int, float, and long Intervals respectively.

Returns A list of discrete values contained in the calling Interval with a step size of `jump`.

Return type `list`

Raises TypeError If a jump is provided, it must be an `int`, `float`, or `long` and match the type of the calling Interval.

A1.2 The Point object

This subsection introduces the Point class.

class point.**Point**(**coordinates*)

Point objects represent a point of N_d cartesian space. Point objects are immutable.

Variables

- **is_generic** – Whether or not the Point object is generic (i.e., the coordinates have not been defined).
- **coordinate** – The coordinate of the Point object.
- **dimension** – The dimension of space the Point object exists in.
- **_is_Point** – An identifier to use in place of `type` or `isinstance`.

__deepcopy__(*memo*)

Deepcopy a Point object via the `copy.deepcopy` method.

__eq__(*other*)

Determine if two Point objects are equal via the == operator.

__getitem__(*key*)

Retrieve the ith cooridinate from a Point object via indexing (e.g., `Point[i]`).

Raises

- **TypeError** – `key` parameter must be an `int`.
- **IndexError** – `key` parameter must be within the set $\{0, \ldots, d\}$ where d is the dimension of the Point object.

__hash__()

Hash implementation for set functionality of Point objects.

__init__(**coordinates*)

Construct a Point object.

Parameters **coordinates** (*strs*|*floats*) – The values to use as the coordinates of the Point object. At least one coordinate must be provided; to create a generic point object, pass values of `"x"`.

Raises

- **ValueError** – At least one coordinate must be provided.
- **TypeError** – All cooridnates must be either strings (equal to `"x"`) or floats.

__ne__(*other*)

Determine if two Point objects are not equal via the != operator.

__repr__()

Return a string representation of the Point object.

__str__()

Return a readable string representation of the Point object.

_key()

Private key function for hashing.

Returns d-tuple consisting of coordinates.

Return type `tuple`

can_observe(*spacetime_loc*, *worldline_start*, *worldline_end*)

Determine if the calling Point object can observe the spacetime location represented by the Point object in `spacetime_loc` parameter on the worldline segment determined by the Point objects in `worldline_start` and `worldline_end` parameters.

Returns Whether or not the calling Point object can observe `spacetime_loc` through the worldine defined by the Point objects in the `worldline_start` and `worldline_end` parameters.

Return type `bool`

clocks_unequal(*other*)
Determine if the clocks of two spacetime locations are unequal wherein the last coordinate of each represents time.

Returns Whether or not the calling Point object's last coordinate is equal to the last coordinate of the Point object in `other` parameter.

Return type `bool`

Raises ValueError Dimensions of the Point object contained in `other` parameter and the calling Point must match.

is_on(*endpoint_1*, *endpoint_2*)
Determine if the calling Point object lies on line segment defined by the endpoint Point objects provided in the `endpoint_1` and `endpoint_2` parameters.

Raises ValueError The calling Point object and the Point objects in the `endpoint_1` and `endpoint_2` parameters must all be in the same dimension of space and no Point object involved can be generic.

Returns Whether or not the calling Point object lies on the line segment.

Return type `bool`

meets(*worldline_1_start*, *worldline_1_end*, *worldline_2_start*, *worldline_2_end*)
Determine if the worldline segments defined by the Point objects in the `worldline_1_start` and `worldline_1_end` parameters and the `worldline_2_start` and `worldline_2_end` parameters meet at the calling Point object's coordinates.

Returns Whether or not worldines meet at the calling Point object's coordinates.

Return type `bool`

Raises ValueError All Point objects must have the same dimension and cannot be generic.

not_same_point(*other*)
Determine if the calling Point object and the Point object contained in `other` parameter are not the same.

Returns Whether or not the calling Point object is unequal to the Point object in `other` parameter.

Return type `bool`

static unstringify(*point_string*)
Reconstruct a Point object from its string representation.

Returns Point object reconstructed from string representation.

Return type Point

Raises ValueError The string must match the form given by `str(Point)` or `repr(Point)`, i.e., $P(c_1, \ldots, c_d)$.

A1.3 The LineSegment object

This subsection introduces the LineSegment class.

class `line_segment.LineSegment`(*start_point*, *end_point*)
Each LineSegment object represents a line segment in N_d cartesian space. LineSegment objects are immutable.

Variables

- **is_generic** – Whether or not the LineSegment object is generic (i.e., the set of coordinates have not been defined).
- **start_point** – The first endpoint of the LineSegment.
- **end_point** – The second endpoint of the LineSegment.
- **dimension** – The dimension of space the LineSegment object exists in.
- **_is_LineSegment** – An identifier to use in place of `type` or `isinstance`.

__contains__(*key*)
Determine if a Point object or LineSegment object given in the `key` parameter is on the calling LineSegment object via the `in` operator.

Parameters **key** (*Point | LineSegment*) – The Point or LineSegment to check for membership in the calling LineSegment.

Raises **TypeError** `key` parameter must be a Point or LineSegment object.

__deepcopy__(*memo*)
Deepcopy a LineSegment object via the `copy.deepcopy` method.

__eq__(*other*)
Determine if two LineSegment objects are equal via the == operator.

__ge__(*other*)
Overloaded >= operator for LineSegment object. Determine if the calling LineSegment object is a superset of the LineSegment object contained in `other` parameter where we view both LineSegment as a set of possible Point objects.

__getitem__(*key*)
Retrieve the *i*th endpoint from a LineSegment object via indexing (e.g., `LineSegment[i]`).

Raises

- **TypeError** – `key` parameter must be an `int`.
- **IndexError** – `key` parameter must be either `0` or `1`.

__gt__(*other*)
Overloaded > operator for LineSegment object. Determine if the calling LineSegment object is a strict superset of the LineSegment object contained in `other` parameter where we view both LineSegment as a set of possible Point objects.

__hash__()
Hash implementation for set functionality of LineSegment objects.

__init__(*start_point*, *end_point*)
Construct a LineSegment object.

Parameters

- **start_point** (*Point*) – The starting point of the LineSegment object.
- **end_point** (*Point*) – The end point of the LineSegment object.

Raises **ValueError** both `start_point` and `end_point` parameters must be Point objects, agree on their gnerality, share the same dimension and be unequal.

__le__(*other*)
Overloaded <= operator for LineSegment object. Determine if the calling LineSegment object is a subset of the LineSegment object contained in `other` parameter where we view both LineSegment as a set of possible Point objects.

__lt__(*other*)
Overloaded < operator for LineSegment object. Determine if the calling LineSegment object

is a strict subset of the LineSegment object contained in `other` parameter where we view both LineSegment as a set of possible Point objects.

__ne__(*other*)
Determine if two LineSegment objects are not equal via the != operator.

__repr__()
Return a string representation of the LineSegment object.

__str__()
Return a readable string representation of the LineSegment object.

_key()
Private key function for hashing.

Returns 2-tuple consisting of Point objects.

Return type `tuple`

static meets(*point*, *line_1*, *line_2*)
Determine if LineSegment objects given by the `line_1` and `line_2` parameters meet at the Point object given by `point` parameter.

Parameters

- **point** (*Point*) – The potential intersection point of the two LineSegment objects.
- **line_1** (*LineSegment*) – The first LineSegment of the possible intersecting pair of line segments.
- **line_2** (*LineSegment*) – The second LineSegment of the possible intersecting pair of line segments.

static unstringify(*line_segment_string*)
Reconstruct a LineSegment object from its string representation.

Returns LineSegment object reconstructed from string representation.

Return type LineSegment

Raises ValueError The string must match the form given by `str(LineSegment)` or `repr(LineSegment)`.

A1.4 The ValueSet object

This subsection introduces the ValueSet class.

The ValueSet class supports any object provided they implement `__deepcopy__`, `__eq__`, `__str__`, `__hash__`, and provide a parser for truth value evaluation. Additionally, `__le__` in ValueSet class must be extended to support the object if the object uses a non-standard form of equality, (e.g. Point objects are subset of generic Point of same dimension).

class `valueset`**.ValueSet**(*valueset*)
ValueSet class.

The ValueSet class uses the `total_ordering` decorator so strict subsets, supersets and strict supersets are also available via the <, >=, and > operators respectively, despite the lack of magic functions for them.

Variables

- **_base_types** – The literal types supported by the ValueSet class.
- **_object_types** – The object types supported by the ValueSet class.
- **values** – The values contained in the ValueSet object.

- **_is_ValueSet** – An identifier to use in place of `type` or `isinstance`.

__add__(*other*)
Overloaded + operator for ValueSet. Take the union of two ValueSet objects or add a single element to the calling Valueset object. If adding an object, the object must be within `_object_types`.

__contains__(*key*)
Overloaded `in` operator for ValueSet. Determine if a value is contained in the calling ValueSet object.

Parameters **key** – The item to test for membership in the calling ValueSet object.

__deepcopy__(*memo*)
Deepcopy a ValueSet object via the `copy.deepcopy` method.

__eq__(*other*)
Determine if two ValueSet objects are equal via the == operator.

__getitem__(*key*)
Retrive the value located at the index given by `key` parameter via indexing (e.g. `ValueSet[key]`).

Parameters **key** (*int*) – The index to use for retrieval.

Raises

- **IndexError** – `key` index must be in $\{0, \ldots, n-1\}$ where n is the number of values contained in the calling ValueSet object.
- **TypeError** – `key` must be an int.

__iadd__(*other*)
Overloaded += operator for ValueSet. Take the union of two ValueSet objects or add a single element to the calling Valueset object. If adding an object, the object must be within `_object_types`.

__init__(*valueset*)
Construct a ValueSet object.

Parameters **valueset** (*list|set*) – The values to place in the ValueSet object. These values are passed through the `_parse` function before being stored.

Raises **TypeError** `valueset` parameter must be a `list` or `set`.

__iter__()
Provide an iterator for ValueSet objects (e.g. "`for value in ValueSet:`").

__le__(*other*)
Overloaded <= operator for ValueSet object. Determine if the calling ValueSet object is a subset of the ValueSet object contained in `other` parameter.

__len__()
Determine the length of a ValueSet object via the `len` built-in function e.g.(`len(ValueSet)`).

__ne__(*other*)
Determine if two ValueSet objects are not equal via the != operator.

__nonzero__()
Determine if a ValueSet object is falsy (e.g. `if ValueSet` or `if not ValueSet`).

__repr__()
Return a string representation of the ValueSet object.

__setitem__(*key*, *value*)
Assign a value in `value` parameter to a ValueSet object at the index given by `key` parameter (e.g. `ValueSet[key] = value`).

Parameters

- **key** (*int*) – The index to use for assignment.
- **value** – The value to assign at index given by `key`.

Raises

- **AttributeError** – An invalid/unsupported object is given as `value` parameter.
- **IndexError** – `key` index must be in $\{0, \ldots, n-1\}$ where n is the number of values contained in the ValueSet object.
- **TypeError** – `key` must be an `int` and value in `value` parameter must be in `ValueSet._base_types` or `ValueSet._object_types`.
- **ValueError** – Duplicate values are not allowed in a ValueSet object.

__str__()
Return a readable string representation of the ValueSet object.

__sub__(*other*)
Overloaded − operator for ValueSet. The − operator functions as the set-theoretic difference.

static _parse(*values*)
Parse a `list` into the standard format used by ValueSet objects. Any `int`s, `long`s, and `float`s are absorbed into an Interval object if they are contained by that Interval and the base types contained in `values` are sorted.

Returns Filtered, sorted values in the ValueSet standard format.

Return type `list`

Raises TypeError `values` parameter must be either a `list` or `set`.

static _split_by_types(*values*)
Split an iterable object by the types of elements within it and return a defaultdict where keys are the types composing the iterable object and the values are lists of the values of the iterable object falling into those types.

Parameters values (*list|set|ValueSet|...*) – An iterable object to split.

Returns A defaultdict of lists where keys correspond to `ValueSet._base_type` and `ValueSet._object_types` present in `values` parameter.

Return type `defaultdict(list)`

Raises

- **AttributeError** – Only objects contianing a single identifier in `_object_types` are supported.
- **TypeError** – An invalid type exists in the iterable object.

classmethod add_object_type(*object_identifier*)
Add compatibility for an object to the ValueSet class. This is part of the vivid object extension protocol.

Parameters object_identifier (*str*) – The identifier used by each instance of the new class. This must be of the form: `"_is_Object"` (e.g. `"_is_Point"` or `"_is_Interval"`).

A2 Attributes and Relations

A2.1 The Attribute object

This subsection introduces the Attribute class.

class `attribute.`**`Attribute`**(*label*, *value_set*)

An Attribute is a finite set A with an associated label l.

Variables

- **label** – The associated label l of the Attribute A.
- **value_set** – A ValueSet object functioning as the set of values that the attribute can take on (e.g {small,large}).
- **_is_Attribute** – An identifier to use in place of `type` or `isinstance`.

`__add__`(*other*)

Combine an Attribute object with another Attribute object, a Relation object, an AttributeStructure object or an AttributeSystem object via the + operator.

Parameters other (*Attribute|Relation|AttributeStructure|AttributeSystem*) – The object to combine with the Attribute. If an Attribute, Relation, or AttributeStructure object is provided, an AttributeStructure object is returned; if an AttributeSystem object is provided, an AttributeSystem object is returned.

Raises TypeError `other` parameter must be an Attribute, Relation, AttributeStructure, or AttributeSystem object.

`__deepcopy__`(*memo*)

Deepcopy an Attribute object via the `copy.deepcopy` method.

`__eq__`(*other*)

Determine if two Attribute objects are equal via the == operator.

`__hash__`()

Hash implementation for set functionality of Attribute objects.

`__init__`(*label*, *value_set*)

Construct an Attribute object.

Parameters

- **label** (*str*) – The label l to associate with the Attribute object.
- **value_set** (*list|ValueSet*) – The set of values the Attribute object can take on.

Raises TypeError `label` parameter must be a string and `value_set` parameter must be either a ValueSet object or a list.

`__ne__`(*other*)

Determine if two Attribute objects are not equal via the != operator.

`__repr__`()

Return a string representation of the Attribute object.

`__str__`()

Return a readable string representation of the Attribute object.

`_key`()

Private key function for hashing.

Returns 2-tuple consisting of (label, valueset)

Return type `tuple`

A2.2 The Relation object

This subsection introduces the Relation class.

class `relation.`**`Relation`**(*definition*, *D_of_r*, *subscript*)

Relation objects represent logical relations used in AttributeStructure objects.

Variables

- **definition** – A string representation of the Relation object's definition with form `Rn(a,...)<=>...` where n is a positive integer; whitespace is ignored.
- **DR** – DR represents $D(R) \subseteq \{A_1, \ldots, A_n\}$; held as a list of strings corresponding to the labels of some set of Attributes objects; no assumptions are made on the labels of the attributes.
- **subscript** – The subscript of the relation.
- **_is_Relation** – An identifier to use in place of `type` or `isinstance`.

`__add__`(*other*)

Combine a Relation object with an Attribute object, an AttributeStructure object or an AttributeSystem object via the + operator.

Parameters other (*Attribute|AttributeStructure|AttributeSystem*) – The object to combine with the Attribute. If an Attribute or AttributeStructure object is provided, an AttributeStructure object is returned; if an AttributeSystem object is provided, an AttributeSystem is returned.

Raises TypeError `other` parameter must be an Attribute, AttributeStructure, or AttributeSystem object.

`__deepcopy__`(*memo*)

Deepcopy a Relation object via the `copy.deepcopy` method.

`__eq__`(*other*)

Determine if two Relation objects are equal via the == operator.

`__init__`(*definition*, *D_of_r*, *subscript*)

Construct a Relation object.

Parameters

- **definition** (*str*) – The definition of the logical relation; valid definitions have the form: `Rn(a,...)  <=>  ...` where all whitespace is ignored.
- **D_of_r** (*list*) – A list of strings representing $D(R) \subseteq \{A_1, \ldots, A_n\}$.
- **subscript** (*int*) – The subscript of the relation; must match subscript in definition.

Raises

- **TypeError** – `definition` parameter must be a `str`, `D_of_r` parameter must be a `list` containing only `str`s and `subscript` parameter must be an `int`.
- **ValueError** – `definition` parameter must be correctly formatted, the number of parameters provided in `definition` must match the length of `D_of_r` and `subscript` parameter must match subscript provided in `definition` parameter.

`__ne__`(*other*)

Determine if two Relation objects are not equal via the != operator.

`__repr__`()

Return a string representation of the Relation object.

__str__()
Return a readable string representation of the Relation object.

get_DR(*string=False*)
Return $D(R)$ of the calling Relation object. If `string` parameter is set to True, return a `str` representation of $D(R)$.

Parameters **string** (*boolean*) – A boolean value for whether or not to return a string representation of $D(R)$.

Returns A representation of $D(R)$.

Return type `str|list`

get_arity()
Return the arity of the calling Relation object.

Returns The length of $D(R)$.

Return type `int`

static is_valid_definition(*definition*)
Determine if a given definition in `definition` parameter is valid. A definition is valid when it is of the form `Rs(x1,...,xn)` <=> <expression>.

The important thing here is the left hand side and the marker `"<=>"`. Everything on the right hand side of `"<=>"` is ignored as far as the definition's validity is concerned; whether or not it is evaluatable is left to `Formula.assign_truth_value()` as it is only during the assignment of a truth value that the expression comes into play. All whitespace is trimmed immediately so arbitrary spacing is allowed.

Parameters **definition** (*str*) – The definition to verify.

Returns Whether or not `definition` is valid.

Return type `bool`

set_DR(*DR*)
Set $D(R)$ to `DR` parameter.

Parameters **DR** (`list`) – The list of strings to set the calling Relation object's $D(R)$ to.

Raises

- **TypeError** – $D(R)$ is not a `list` of `str`s.
- **ValueError** – The cardinality of $D(R)$ must match argument cardinality in Relation object's definition member.

set_definition(*definition*)
Set the definition of the Relation object to `definition` parameter after ensuring that it conforms to required format.

Parameters **definition** (*str*) – The new definition of the Relation object.

Raises

- **TypeError** – `definition` parameter must be a `str`.
- **ValueError** – `definition` must conform to valid definition rules.

A3 Attribute Structures

A3.1 The AttributeStructure object

This subsection introduces the AttributeStructure class.

class `attribute_structure.`**`AttributeStructure`**(**args*)

Analogous to the definition of attribute structures given in [Arkoudas and Bringsjord, 2009], an AttributeStructure object consists of a finite set of Attribute objects $A_1, \ldots, A_k$; and a countable collection $\mathcal{R}$ of computable Relation objects with $D(R) \subseteq \{A_1, \ldots, A_k\}$ for each $R \subseteq \mathcal{R}$. i.e.,

$$\mathcal{A} = (\{A_1, \ldots, A_k\}; \mathcal{R})$$

The AttributeStructure class uses the `total_ordering` decorator so strict subsets, supersets and strict supersets are also available via the <, >=, and > operators respectively, despite the lack of magic functions for them.

Variables

- **attributes** – A list of Attribute objects (i.e., $A_1, \ldots, A_k$); always maintained as a list.
- **relations** – A dictionary of relations (i.e., $\mathcal{R}$).
- **_is_AttributeStructure** – An identifier to use in place of `type` or `isinstance`.

`__add__`(*other*)

Add an Attribute, Relation, AttributeStructure, or AttributeSystem object via the + operator.

Parameters other (*Attribute|Relation|AttributeStructure|AttributeSystem*) – The object to combine with the AttributeStructure. If an Attribute, Relation, or AttributeStructure object is provided, an AttributeStructure object is returned; if an AttributeSystem object is provided, an AttributeSystem object is returned.

Raises

- **TypeError** – `other` parameter must be an Attribute, Relation, AttributeStructure, or AttributeSystem object.
- **ValueError** – Duplicate Attribute labels are not permitted, duplicate subscripts are not permitted and every Relation's $D(R)$ must be a subset of Attribute labels in the AttributeStructure.

`__contains__`(*key*)

Determine if Attribute, Relation, Attribute corresponding to a label in the `key` parameter, or Relation corresponding to a subscript in the `key` parameter is contained by AttributeStructure via `in` operator.

Parameters key (*Attribute|Relation|str|int*) – The key to use when checking for membership.

Raises TypeError `key` must be an Attribute object, Relation object, `str`, or `int`.

`__deepcopy__`(*memo*)

Deepcopy an AttributeStructure object via the `copy.deepcopy` method.

`__eq__`(*other*)

Determine if two AttributeStructure objects are equal via the == operator.

`__getitem__`(*key*)

Retrieve a reference to the Attribute object or Relation object in the AttributeStructure via the key provided in the `key` parameter provided.

Parameters **key** (*Attribute|Relation|str|int*) – The Attribute object, Relation object, label, or subscript to use when attempting to find the corresponding Attribute object or Relation object.

Raises

- **KeyError** – Attribute object or Relation object provided in the `key` parameter not found in the AttributeStructure, no Attribute object with label provided in the `key` parameter found in the AttributeStructure, or no Relation object with subscript provided in the `key` parameter found in the AttributeStructure.
- **TypeError** – `key` is not an Attribute object, Relation object, `int`, or `str`.

__iadd__(*other*)

AAdd an Attribute, Relation, AttributeStructure, or AttributeSystem object via the += operator.

Parameters **other** (*Attribute|Relation|AttributeStructure|AttributeSystem*) – The object to combine with the AttributeStructure. If an Attribute, Relation, or AttributeStructure object is provided, an AttributeStructure object is returned; if an AttributeSystem object is provided, an AttributeSystem object is returned.

Raises

- **TypeError** – `other` parameter must be an Attribute, Relation, AttributeStructure, or AttributeSystem object.
- **ValueError** – Duplicate Attribute labels are not permitted, duplicate subscripts are not permitted and every Relation's $D(R)$ must be a subset of Attribute labels in the AttributeStructure.

__init__(**args*)

Construct an AttributeStructure object.

Parameters **args** (*Attribute|Relation*) – Any amount of Attribute and Relation objects.

Raises

- **TypeError** – all optional positional arguments provided must be Attribute or Relation objects.
- **ValueError** – Duplicate Attribute labels are not permitted, Duplicate Relation subscripts are not permitted, and each Relation object's $D(R)$ must be a subset of the cartesian product of some combination of the labels of the Attributes provided.

__isub__(*other*)

Remove Attribute's or Relation's via − operator. If an AttributeStructure object is provided, all Attribute objects and Relation objects within that AttributeStructure object will be removed from the calling AttributeStructure.

Parameters **other** (*Attribute|Relation|AttributeStructure*) – The Attribute, Relation, or AttributeStructure object to remove.

Raises

- **KeyError** – Invalid Attribute or Relation object provided in `other` parameter.
- **TypeError** – Only Attribute, Relation, or AttributeStructure objects can be removed.
- **ValueError** – Some Attribute or Relation object provided in AttributeStructure object in `other` parameter not found in calling AttributeStructure object or some Relation object's $D(R)$ is invalid after an Attribute object is removed.

__le__(*other*)
Overloaded <= operator. Determine if the calling AttributeStructure object is a subset of the AttributeStructure object contained in `other` parameter.

__ne__(*other*)
Determine if two AttributeStructure objects are not equal via the != operator.

__repr__()
Return a string representation of the AttributeStructure object.

__str__()
Return a readable string representation of the AttributeStructure object.

__sub__(*other*)
Remove Attribute's or Relation's via − operator. If an AttributeStructure object is provided, all Attribute objects and Relation objects within that AttributeStructure object will be removed from the calling AttributeStructure.

Parameters **other** (*Attribute|Relation|AttributeStructure*) – The Attribute, Relation, or AttributeStructure object to remove.

Raises

- **KeyError** – Invalid Attribute or Relation object provided in `other` parameter.
- **TypeError** – Only Attribute, Relation, or AttributeStructure objects can be removed.
- **ValueError** – Some Attribute or Relation object provided in AttributeStructure object in `other` parameter not found in calling AttributeStructure object or some Relation object's $D(R)$ is invalid after an Attribute object is removed.

get_cardinality()
Return the cardinality of the calling AttributeStructure object.

Returns The cardinality of the AttributeStructure object, i.e., the amount of Attribute objects contained therein.

Return type `int`

get_labels()
Return the labels of the Attribute objects within the calling AttributeStructure object.

Returns A list of the labels of the Attribute objects in the calling AttributeStructure object.

Return type `list`

get_subscripts()
Return the subscripts of the Relation objects within the calling AttributeStructure object.

Returns A list of the subscripts of the Relation objects in this AttributeStructure object.

Return type `list`

A4 Attribute Systems

A4.1 The AttributeSystem object

This subsection introduces the AttributeSystem class.

class `attribute_system.`**`AttributeSystem`**(*attribute_structure*, *objects*)

Analogous to the definition of attribute systems in [Arkoudas and Bringsjord, 2009], an AttributeSystem object, based on the AttributeStructure object $\mathcal{A}$ is a pair

$$\mathcal{S} = (\{s_1, \ldots, s_n\}; \mathcal{A})$$

consisting of a finite number $n > 0$ of objects $s_1, \ldots, s_n$ (represented as `strs`) and $\mathcal{A}$.

The AttributeSystem class uses the `total_ordering` decorator so strict subsets, supersets and strict supersets are also available via the <, >=, and > operators respectively, despite the lack of magic functions for them.

Variables

- **attribute_structure** – The AttributeStructure of the AttributeSystem.
- **objects** – The objects of the AttributeSystem; held as a list of `strs`.
- **_is_AttributeSystem** – An identifier to use in place of `type` or `isinstance`.

`__add__`(*other*)

Add an Attribute, Relation, AttributeStructure, or AttributeSystem object via the + operator.

Parameters other (*Attribute|Relation|AttributeStructure|AttributeSystem*) – The object to combine with the AttributeSystem. An AttributeSystem object is always returned regardless of the type of `other` parameter.

Raises

- **TypeError** – `other` parameter must be an Attribute, Relation, AttributeStructure, or AttributeSystem object.
- **ValueError** – Cannot add AttributeSystems with overlapping objects.

`__contains__`(*key*)

Determine if the Attribute object, Relation object, AttributeStructure object, or `str` in the `key` parameter is contained by the calling AttributeSystem object via `in` operator.

Parameters key (*Attribute|Relation|AttributeStructure|str*) – The key to use when checking for membership.

Raises TypeError `key` parameter must be an Attribute object, Relation object, AttributeStructure object, or `str`.

`__deepcopy__`(*memo*)

Deepcopy an AttributeSystem object via the `copy.deepcopy` method.

`__eq__`(*other*)

Determine if two AttributeSystem objects are equal via == operator.

`__getitem__`(*key*)

Retrieve a reference to the Attribute, Relation, or object in the AttributeSystem by indexing with the key provided in `key` parameter (e.g., `"AttributeSystem[key]"`).

Parameters key (*Attribute|Relation|str*) – The Attribute, Relation or name of the object to get the reference of from the calling AttributeSystem object.

Raises TypeError `str` in `key` does not match any object contained in the calling AttributeSystem object.

__iadd__(*other*)
Add an Attribute, Relation, AttributeStructure, or AttributeSystem object via the += operator.

Parameters other (*Attribute|Relation|AttributeStructure|AttributeSystem*) – The object to combine with the AttributeSystem. An AttributeSystem object is always returned regardless of the type of `other` parameter.

Raises

- **TypeError** – `other` parameter must be an Attribute, Relation, AttributeStructure, or AttributeSystem object.
- **ValueError** – Cannot add AttributeSystems with overlapping objects.

__init__(*attribute_structure*, *objects*)
Construct AttributeSystem object.

Parameters

- **attribute_structure** (*AttributeStructure*) – an AttributeStructure object to use as the attribute structure $\mathcal{A}$ of the AttributeSystem object.
- **objects** (*list*) – A list of `str`s denoting the objects of the AttributeSystem.

Raises

- **TypeError** – `objects` parameter must be a `list` and `attribute_structure` parameter must be an AttributeStructure object.
- **ValueError** – all objects provided in `objects` parameter must be unique non-empty `str`s.

__isub__(*other*)
Remove an Attribute, Relation, AttributeStructure, or AttributeSystem object via the −= operator. In the case of AttributeStructure and AttributeSystem objects being provided to `other` parameter, remove all of their consituent parts from the calling AttributeSystem.

Parameters other (*Attribute|Relation|AttributeStructure|AttributeSystem*) – The object to remove from the AttributeSystem. An AttributeSystem object is always returned regardless of the type of `other` parameter.

Raises

- **TypeError** – `other` parameter must be an Attribute, Relation, AttributeStructure, or AttributeSystem object.
- **ValueError** – Cannot remove objects not present in this AttributeSystem.

__le__(*other*)
Determine if the calling AttributeSystem object is a subset of the AttributeSystem object contained in the `other` parameter via <= operator.

__ne__(*other*)
Determine if two AttributeSystem objects are not equal via != operator.

__repr__()
Return a string representation of the AttributeSystem object.

__str__()
Return a readable string representation of the AttributeSystem object.

__sub__(*other*)
Remove an Attribute, Relation, AttributeStructure, or AttributeSystem object via the − operator. In the case of AttributeStructure and AttributeSystem objects being provided to `other` parameter, remove all of their consituent parts from the calling AttributeSystem.

Parameters other (*Attribute|Relation|AttributeStructure|AttributeSystem*) – The object to remove from the AttributeSystem. An AttributeSystem object is always returned regardless of the type of `other` parameter.

Raises

- **TypeError** – `other` parameter must be an Attribute, Relation, AttributeStructure, or AttributeSystem object.
- **ValueError** – Cannot remove objects not present in this AttributeSystem.

`get_power``()`

Get the power of the calling AttributeSystem object, i.e., $n \cdot |\mathcal{A}|$.

Returns The power of the calling AttributeSystem object: $n \cdot |\mathcal{A}|$

Return type `int`

`is_automorphic``()`

Determine if the calling AttributeSystem object is automorphic.

Returns Whether or not the calling AttributeSystem object is automorphic; i.e., some (perhaps all) of the objects $s_1, \ldots, s_n$ are contained by at least one of the ValueSets of the Attribute objects of the underlying AttributeStructure object $\mathcal{A}$.

Return type `bool`

A5 States

A5.1 The State object

This subsection introduces the State class.

class `state.`**`State`**(*attribute_system*, *ascriptions={}*)

Analogous to the definition of states in [Arkoudas and Bringsjord, 2009], each State object is a state of an AttributeSystem; that is a set of functions $\sigma = \{\delta_1, \ldots, \delta_k\}$, where each δ_i is a function from $\{s_1, \ldots, s_n\}$ to the set of all non-empty finite subsets of A_i, i.e.,

$$\delta_i : \{s_1, \ldots, s_n\} \rightarrow \mathcal{P}_{fin}(A_i) \setminus \emptyset.$$

The State class uses the `total_ordering` decorator so proper extensions, contravariant extensions and contravariant proper extensions are also available via the <, >=, and > operators respectively, despite the lack of magic functions for them.

Variables

- **attribute_system** – A copy of the AttributeSytem object $\mathcal{S}$ that the State object comes from.
- **ascriptions** – The ascriptions of the state (i.e., the set of attribute-object pairs and their corresponding ValueSet objects) $\delta_i,\ i = 1, \ldots, k$.
- **_is_State** – An identifier to use in place of `type` or `isinstance`.

`__deepcopy__`(*memo*)

Deepcopy a State object via the `copy.deepcopy` method.

`__eq__`(*other*)

Determine if two State objects are equal via the == operator.

`__getitem__`(*key*)

Retrive the ascription δ_i or ascription of a particular object $\delta_i(s_j)$ (that is, the ValueSet corresponding to the attribute-object pair) given by `key` parameter via indexing (e.g. `State[key]`).

Raises

- **KeyError** – `key` parameter must be a valid Attribute label or valid attribute-object pair in the underlying AttributeSystem $\mathcal{S}$ of the State object.
- **TypeError** – `key` parameter must be a `str` or `tuple` containing only `str`s.

`__init__`(*attribute_system*, *ascriptions={}*)

Construct a State object.

Parameters

- **attribute_system** (*AttributeSystem*) – The AttributeSystem object $\mathcal{S}$ from which the State comes from.
- **ascriptions** (`dict`) – An optional dictionary of attribute-object pairs $\delta_i(s_j)$ to use as ascriptions; if some attribute-object pair is not provided, the full ValueSet of the Attribute object corresponding to the attribute label in the attribute-object pair is used.

Raises TypeError `attribute_system` parameter must be an AttributeSystem object and `ascriptions` parameter must be a `dict`.

`__le__`(*other*)

Overloaded <= operator for State; Determine if the calling State object is an extension of the State object in `other` parameter.

Raises

- **TypeError** – `other` parameter must be a State object.
- **ValueError** – The state object in the `other` parameter must share the same underlying AttributeSystem object $\mathcal{S}$ as the calling State object.

__ne__(*other*)
Determine if two State objects are not equal via the != operator.

__repr__()
Return a string representation of the State object.

__str__()
Return a readable string representation of the State object.

add_object(*obj*, *ascriptions=None*)
Add an object s' to the calling State object's underlying AttributeSystem $\mathcal{S}$ and optionally update any ascriptions of the new object $\delta_i(s')$ provided.

Parameters

- **obj** (`str`) – The new object s' to add to the State.
- **ascriptions** (`dict`) – The optional ValueSets to assign to the attribute-object pairs corresponding to the new object $\delta_i(s')$.

Raises

- **TypeError** – `obj` parameter must be a non-empty `str` and if `ascriptions` parameter is provided, it must be a `dict`.
- **ValueError** – Duplicate objects cannot be added and all ascriptions provided must be from an existing label of an Attribute object in the underlying AttributeSystem object $\mathcal{S}$ to s'.

get_alternate_extensions(**states*)
Return all alternate extensions of the calling State object with respect to State objects $\sigma_1, \ldots, \sigma_m$ provided by optional positional arguments of `states` parameter, i.e., generate $\mathbf{AE}(\{\sigma_1, \ldots, \sigma_m\}, \sigma')$.

Parameters **states** (*State*) – The states $\{\sigma_1, \ldots, \sigma_m\}$ to use for the derivation of the alternate extensions of the calling State object.

Returns $\mathbf{AE}(\{\sigma_1, \ldots, \sigma_m\}, \sigma')$.

Return type `list`

Raises

- **TypeError** – all optional positional arguments must be State objects.
- **ValueError** – at least one State object must be provided in optional positional arguments and all provided State objects must be proper extensions of the calling State object.

get_worlds()
Return a list of all possible worlds $(w; \widehat{\rho})$ derivable from the calling State object.

Returns all worlds $(w; \widehat{\rho})$ derivable from this State object.

Return type `list`

is_alternate_extension(*s_prime*, **states*)
Determine if the State object in `s_prime` parameter is an alternate extension of the calling State object w.r.t. the State objects $\sigma_1, \ldots, \sigma_m$ provided by optional positional arguments of `states` parameter, i.e., evaluate *Alt*$(\sigma, \{\sigma_1, \ldots, \sigma_m\}, \sigma')$.

Parameters

- **s_prime** (*State*) – The State object to verify as the alternate extension, σ'

- **states** (*State*) – The states $\{\sigma_1, \ldots, \sigma_\mathrm{m}\}$ to use for the derivation of the alternate extensions of the calling State object.

Returns The result of the evaluate of $Alt(\sigma, \{\sigma_1, \ldots, \sigma_m\}, \sigma')$.

Return type `bool`

Raises TypeError `s_prime` parameter must be a State object.

`is_disjoint`(*other*)
Determine if the calling State object is disjoint from the State object in the `other` parameter.

Returns Whether or not the calling State object and State object contained in the `other` parameter are disjoint.

Return type `bool`

`is_valuation`(*label*)
Determine if the ascription δ_i corresponding to the `label` parameter is a valuation in the calling State object; that is $|\delta_i(s_j)|= 1$ for every $j = 1, \ldots, n$.

Parameters **label** (`str`) – The label corresponding to the ascription δ_i to check for valuation in the calling State object.

Returns Whether or not the ascription δ_i corresponding to `label` is a valuation in the calling State object.

Return type `bool`

`is_world`()
Determine if the calling State object σ is a world; that is every ascription δ_i of σ is a valuation.

Returns Whether or not the calling State object is a world.

Return type `bool`

static `join`(*s1*, *s2*)
Join two State objects if possible, i.e., return $\sigma_1 \sqcup \sigma_2$.

Parameters

- **s1** (*State*) – The left operand σ_1 to use for the join operation.
- **s2** (*State*) – The right operand σ_2 to use for the join operation.

Raises ValueError `s1` and `s2` parameters must share the same underlying AttributeSystem $\mathcal{S}$.

`set_ascription`(*ao_pair*, *new_valueset*)
Set an ascription of an object (that is, $\delta_i(s_j)$) in the calling State object, given by the `ao_pair` parameter, to the ValueSet object provided in the `new_valueset` parameter.

Parameters

- **ao_pair** (`tuple`) – The attribute-object pair $\delta_i(s_j)$ to use as a key for the ascription `dict` member in the calling State object.
- **new_valueset** (*ValueSet*) – The new ValueSet object to assign to the corresponding attribute-object pair $\delta_i(s_j)$ given by the `ao_pair` paramater.

Raises

- **TypeError** – `ao_pair` parameter must be a `tuple` and `new_valueset` parameter must be a `list`, `set`, or ValueSet object.
- **ValueError** – `ao_pair` parameter must be a 2-tuple (`str`,`str`) and `new_valueset` parameter must be a non-empty subset of the ValueSet object of the Attribute object corresponding to the attribute in the `ao_pair` parameter.

- **KeyError** – `ao_pair` must be a key in the `ascriptions` member of this State object.

A6 Vocabularies

A6.1 The RelationSymbol object

This subsection introduces the RelationSymbol class.

class `relation_symbol.`**`RelationSymbol`**(*name*, *arity*)

The RelationSymbol class is used solely by the Vocabulary class.

Variables

- **name** – A `str` designating the name of the RelationSymbol object.
- **arity** – An `int` designating the arity of the RelationSymbol object.
- **_is_RelationSymbol** – An identifier to use in place of `type` or `isinstance`.

`__deepcopy__`(*memo*)

Deepcopy a RelationSymbol object via the `copy.deepcopy` method.

`__eq__`(*other*)

Determine if two RelationSymbol objects are equal via the == operator.

`__hash__`()

Hash implementation for set functionality of RelationSymbol objects.

`__init__`(*name*, *arity*)

Construct a RelationSymbol object.

Parameters

- **name** (`str`) – The name of the RelationSymbol object.
- **arity** (`int`) – The arity of the RelationSymbol object.

Raises

- **TypeError** – `name` parameter must be a `str` and `arity` parameter must be an `int`.
- **ValueError** – `arity` must be positive.

`__ne__`(*other*)

Determine if two RelationSymbol objects are not equal via the != operator.

`__repr__`()

Return a string representation of the RelationSymbol object.

`__str__`()

Return a readable string representation of the RelationSymbol object.

`_key`()

Private key function for hashing.

Returns 2-tuple consisting of (`name`, `arity`)

Return type `tuple`

A6.2 The Vocabulary object

This subsection introduces the Vocabulary class.

class `vocabulary.`**`Vocabulary`**(*C*, *R*, *V*)

Analogous to the definition of vocabularies in [Arkoudas and Bringsjord, 2009], the Vocabulary class represents a first-order vocabulary $\Sigma = (\mathrm{C},\mathrm{R},\mathrm{V})$ consisting of a set of constant symbols C; a set of relation symbols R; and a set of variables V.

Variables

- **C** – The constants C of the vocabulary.
- **R** – The relation symbols R of the vocabulary.
- **V** – The variables V of the vocabulary.
- **_is_Vocabulary** – An identifier to use in place of `type` or `isinstance`.

__contains__(*key*)
Determine if the calling Vocabulary object contains the `str` or RelationSymbol object in the `key` parameter.

Parameters **key** (*RelationSymbol|str*) – The `str` or RelationSymbol object to test for membership in the calling Vocabulary object.

__deepcopy__(*memo*)
Deepcopy a Vocabulary object via the `copy.deepcopy` method.

__eq__(*other*)
Determine if two Vocabulary objects are equal via == operator.

__hash__()
Hash implementation for set functionality of Vocabulary objects.

__init__(*C*, *R*, *V*)
Construct a Vocabulary object. Each parameter `C`, `R` and `V` are sorted before being stored.

Parameters

- **C** (`list`) – The constants C of the Vocabulary object; held as a `list` of `str`s.
- **R** (`list`) – The relation symbols R of the Vocabulary object; held as a `list` of RelationSymbol objects.
- **V** (`list`) – The variables V of the Vocabulary object; held as a `list` of `str`s.

Raises

- **TypeError** – `C`, `R`, and `V` parameters must all be lists, `C` and `V` must be contain only `str`s and `R` must contain only RelationSymbol objects.
- **ValueError** – `C` and `V` cannot overlap and duplicates are not permitted in any of the lists.

__ne__(*other*)
Determine if two Vocabulary objects are not equal via `!=` operator.

__repr__()
Return a string representation of the Vocabulary object.

__str__()
Return a readable string representation of the Vocabulary object.

_key()
Private key function for hashing.

Returns 3-tuple consisting of (`C`, `R`, `V`)

Return type `tuple`

add_constant(*constant*)
Add a constant to this Vocabulary object's constants `C`.

Parameters **constant** (`str`) – The new constant to add to the Vocabulary's constants `C`.

Raises

- **TypeError** – `constant` parameter must be a `str`.
- **ValueError** – Duplicate symbols are not permitted.

add_variable(*variable*)

Add a variable to this Vocabulary object's variables `V`.

Parameters **variable** (`str`) – The new variable to add to the Vocabulary's variables `V`.

Raises

- **TypeError** – `variable` parameter must be a `str`.
- **ValueError** – Duplicate symbols are not permitted.

A7 Constant and Variable Assignments

A7.1 The Assignment Base Class

This subsection introduces the Assignment class.

class `assignment.`**`Assignment`**(*vocabulary*, *attribute_system*)

The Assignment class functions as a superclass for the ConstantAssignment and VariableAssignment classes.

Variables

- **vocabulary** – A reference to the Vocabulary object Σ that the Assignment is defined over.
- **attribute_system** – A copy of the AttributeSystem the Assignment originates from.
- **_is_Assignment** – An identifier to use in place of `type` or `isinstance`.

`__eq__`(*other*)

Determine if two Assignment objects are equal via the == operator.

`__init__`(*vocabulary*, *attribute_system*)

Construct a base Assignment.

Parameters

- **vocabulary** (*Vocabulary*) – The Vocabulary Σ the Assignment is defined over.
- **attribute_system** (*AttributeSystem*) – The AttributeSystem from which the objects in the Assignment come from.

Raises TypeError `vocabulary` parameter must be a Vocabulary object and `attribute_system` parameter must be an AttributeSystem object.

`__ne__`(*other*)

Determine if two Assignment objects are not equal via the != operator.

A7.2 The ConstantAssignment object

This subsection introduces the ConstantAssignment class.

class `constant_assignment.`**`ConstantAssignment`**(*vocabulary*, *attribute_system*, *mapping*)

Bases: `assignment.Assignment`

Analogous to the definition of constant assignments in [Arkoudas and Bringsjord, 2009], a ConstantAssignment is a partial function ρ from the constants C of some Vocabulary object Σ, to the objects $\{s_1, \ldots, s_n\}$ of some AttributeSystem $\mathcal{S}$.

The ConstantAssignment class uses the `total_ordering` decorator so strict subsets, supersets and strict supersets are also available via the <, >=, and > operators respectively, despite the lack of magic functions for them.

Variables

- **vocabulary** – A reference to the Vocabulary object Σ the ConstantAssignment is defined over.
- **attribute_system** – A copy of the AttributeSystem object $\mathcal{S}$ the ConstantAssignment originates from.
- **mapping** – The mapping C $\longmapsto \{s_1, \ldots, s_n\}$.

- **source** – The constants of Σ used in the partial mapping ρ.
- **target** – The objects of $\mathcal{S}$ used in the partial mapping ρ.
- **_is_ConstantAssignment** – An identifier to use in place of `type` or `isinstance`.

__deepcopy__(*memo*)
Deepcopy a ConstantAssignment object via the `copy.deepcopy` method. This does not break the reference to the underlying Vocabulary object Σ.

__eq__(*other*)
Determine if two ConstantAssignment objects are equal via the == operator.

__getitem__(*key*)
Retrive the object s_i mapped to the constant c_i given by `key` parameter via indexing (e.g. `ConstantAssignment[key]`).

Parameters **key** (*str*) – The constant c_i to use for retrieval.

Raises

- **KeyError** – The constant c_i given by the `key` parameter is not in this ConstantAssignment's `source` member.
- **TypeError** – `key` parameter must be a `str`.

__init__(*vocabulary*, *attribute_system*, *mapping*)
Construct a ConstantAssignment object.

Parameters

- **vocabulary** (*Vocabulary*) – The Vocabulary object Σ the ConstantAssignment is defined over.
- **attribute_system** (*AttributeSystem*) – The AttributeSystem object $\mathcal{S}$ from which the objects $\{s_1, \ldots, s_n\}$ in the ConstantAssignment come from.
- **mapping** (`dict`) – The mapping ρ from the constants C of the Vocabulary object Σ in the `vocabulary` parameter to the objects $\{s_1, \ldots, s_n\}$ of the AttributeSystem object $\mathcal{A}$ in the `attribute_system` parameter.

Raises

- **TypeError** – `vocabulary` parameter must be a Vocabulary object, `attribute_system` parameter must be an AttributeSystem object and `mapping` parameter must be a `dict` with `str` keys and values.
- **ValueError** – All keys in the `mapping` parameter must be in the Vocabulary object in the `vocabulary` parameter's `C` member and all values in the `mapping` parameter must be unique and match some object in the `object` member of the AttributeSystem object in the `attribute_system` parameter.

__lt__(*other*)
Overloaded < operator for ConstantAssignment. Determine if the calling ConstantAssignment object is a subset of the ConstantAssignment object in the `other` parameter.

Raises TypeError `other` parameter must be a ConstantAssignment object.

__ne__(*other*)
Determine if two ConstantAssignment objects are not equal via the != operator.

__repr__()
Return a string representation of the ConstantAssignment object.

__str__()
Return a readable string representation of the ConstantAssignment object.

add_mapping(*constant_symbol*, *obj*)
Extend the calling ConstantAssignment object by adding a new mapping from the constant c' in the `constant_symbol` parameter to the object o' in the `obj` parameter.

Raises

- **TypeError** – Both `constant_symbol` and `obj` parameters must be `str`s.
- **ValueError** – The constant c' in the `constant_symbol` parameter must be in the `C` member of the underlying Vocabulary object Σ, the object o' in the `obj` parameter must be in the objects of the `objects` member of the underlying AttributeSystem object $\mathcal{S}$ and neither the constant c' nor the object o' may be a duplicate.

get_domain()
Get the set of all and only those constant symbols for which the calling ConstantAssignment object ρ is defined w.r.t. the `C` member of the `vocabulary` member of ρ.

Returns The list of constants for which ρ is defined.

Return type `list`

in_conflict(*other*)
Check if the calling ConstantAssignment object ρ_1 is in conflict with the ConstantAssignment object ρ_2 provided in the `other` parameter.

Returns Whether or not ρ_1 and ρ_2 are in conflict, that is, if there is some $c \in Dom(\rho_1) \cap Dom(\rho_2)$, such that $\rho_1(c) \neq \rho_2(c)$.

Return type `bool`

Raises TypeError `other` parameter must be a ConstantAssignment object.

is_total()
Determine if the calling ConstantAssignment object ρ is a total function $\widehat{\rho}$ from C $\longrightarrow$ $\{s_1, \ldots, s_n\}$.

Returns Whether or not the source of ρ spans the `C` member of Σ.

Return type `bool`

remove_mapping(*constant_symbol*, *obj*)
Extend the calling ConstantAssignment object by removing an existing mapping from the constant c' in the `constant_symbol` parameter to the object o' in the `obj` parameter.

Raises

- **TypeError** – both `constant_symbol` and `obj` parameters must be `str`s.
- **ValueError** – The constant c' in the `constant_symbol` parameter must be in the `C` member of the underlying Vocabulary object Σ, the object o' in the `obj` parameter must be in the objects of the `objects` member of the underlying AttributeSystem object $\mathcal{S}$ and the constant c' and the object o' must already be in the `source` and `target` members of the calling ConstantAssignment object respectively.

A7.3 The VariableAssignment object

This subsection introduces the VariableAssignment class.

class variable_assignment.**VariableAssignment**(*vocabulary*, *attribute_system*, *mapping*, *dummy=False*)
Analogous to the definition of variable assignments in [Arkoudas and Bringsjord, 2009], a VariableAssignment is a total function χ from the variables V of some Vocabulary object Σ, to the objects $\{s_1, \ldots, s_n\}$ of some AttributeSystem $\mathcal{S}$.

Variables

- **vocabulary** – A reference to the Vocabulary object Σ the ConstantAssignment is defined over.
- **attribute_system** – A copy of the AttributeSystem object $\mathcal{S}$ the ConstantAssignment originates from.
- **mapping** – The mapping V $\longrightarrow \{s_1, \ldots, s_n\}$.
- **source** – The variables of Σ used in the total mapping χ.
- **target** – The objects of $\mathcal{S}$ used in the total mapping χ.
- **_is_VariableAssignment** – An identifier to use in place of type or isinstance.

The VariableAssignment class uses the `total_ordering` decorator so strict subsets, supersets and strict supersets are also available via the <, >=, and > operators respectively, despite the lack of magic functions for them.

`__deepcopy__` (*memo*)
Deepcopy a VariableAssignment object via the `copy.deepcopy` method. This does not break the reference to the underlying Vocabulary object Σ.

`__eq__` (*other*)
Determine if two VariableAssignment objects are equal via the == operator.

`__getitem__` (*key*)
Retrive the object s_i mapped to the variable v_i given by `key` parameter via indexing (e.g. `VariableAssignment[key]`).

Parameters **key** (*str*) – The variable v_i to use for retrieval.

Raises

- **KeyError** – The variable v_i given by the `key` parameter is not in this VariableAssignment's `source` member.
- **TypeError** – `key` parameter must be a `str`.

`__init__` (*vocabulary*, *attribute_system*, *mapping*, *dummy=False*)
Construct a VariableAssignment object.

Parameters

- **vocabulary** (*Vocabulary*) – The Vocabulary object Σ the VariableAssignment is defined over.
- **attribute_system** (*AttributeSystem*) – The AttributeSystem object $\mathcal{S}$ from which the objects $\{s_1, \ldots, s_n\}$ in the VariableAssignment come from.
- **mapping** (`dict`) – The mapping χ from the variables V of the Vocabulary object Σ in the `vocabulary` parameter to the objects $\{s_1, \ldots, s_n\}$ of the AttributeSystem object $\mathcal{A}$ in the `attribute_system` parameter.
- **dummy** (`bool`) – A flag for creating a dummy (i.e., empty) VariableAssignment object χ_{dummy}.

Raises

- **TypeError** – `vocabulary` parameter must be a Vocabulary object, `attribute_system` parameter must be an AttributeSystem object and `mapping` parameter must be a `dict` with `str` keys and values.
- **ValueError** – All keys in the `mapping` parameter must be in the Vocabulary object in the `vocabulary` parameter's `V` member and all values in the `mapping` parameter must be unique and match some object in the `object` member of the AttributeSystem object in the `attribute_system` parameter.

__ne__(*other*)
Determine if two VariableAssignment objects are not equal via the != operator.

__repr__()
Return a string representation of the VariableAssignment object.

__str__()
Return a readable string representation of the VariableAssignment object.

A8 Named States

A8.1 The NamedState object

This subsection introduces the NamedState class.

class `named_state.`**`NamedState`**(*attribute_system*, *p*, *ascriptions={}*)
Bases: `state.State`

Analogous to the definition of named states in [Arkoudas and Bringsjord, 2009], each NameState object wraps a pair $(\sigma; \rho)$ consisting of a state σ and a constant assignment ρ.

The NamedState class uses the `total_ordering` decorator so proper extensions, contravariant extensions and contravariant proper extensions are also available via the <, >=, and > operators respectively, despite the lack of magic functions for them.

Variables

- **attribute_system** – A copy of the AttributeSytem object $\mathcal{S}$ that the NamedState object comes from.
- **ascriptions** – The ascriptions of the named state (i.e., the set of attribute-object pairs and their corresponding ValueSet objects) $\delta_i,\ i = 1, \ldots, k$.
- **p** – The ConstantAssignment object ρ of the named state.
- **_is_NamedState** – An identifier to use in place of `type` or `isinstance`.

`__deepcopy__`(*memo*)
Deepcopy a NamedState object via the `copy.deepcopy` method. This does not break the reference to the underlying Vocabulary object Σ.

`__eq__`(*other*)
Determine if two NamedState objects are equal via the == operator.

`__init__`(*attribute_system*, *p*, *ascriptions={}*)
Construct a NamedState object.

Parameters

- **attribute_system** (*AttributeSystem*) – The AttributeSystem object $\mathcal{S}$ from which the NamedState object comes from.
- **p** (*ConstantAssignment*) – The ConstantAssignment object ρ of the NamedState object.
- **ascriptions** (`dict`) – An optional dictionary of attribute-object pairs $\delta_i(s_j)$ to use as ascriptions; if some attribute-object pair is not provided, the full ValueSet of the Attribute object corresponding to the attribute label in the attribute-object pair is used.

Raises

- **TypeError** – `p` parameter must be a ConstantAssignment object.
- **ValueError** – The AttributeSystem object provided in the `attribute_system` parameter and the AttributeSystem object of the ConstantAssignment object in the `p` parameter must match.

`__le__`(*other*)
Overloaded <= operator for NamedState; Determine if the calling NamedState object $(\sigma; \rho)$ is an extension of the NamedState object $(\sigma'; \rho')$ in the `other` parameter; that is, if $(\sigma; \rho) \sqsubseteq (\sigma'; \rho')$.

Raises TypeError `other` parameter must be a NamedState object.

__ne__(*other*)
Determine if two NamedState objects are not equal via the != operator.

__repr__()
Return a string representation of the NamedState object.

__str__()
Return a readable string representation of the NamedState object.

_generate_variable_assignments()
Generate all possible VariableAssignment objects χ derivable from the calling NamedState object i.e., find all combinations of objects not in the calling NamedState object's ConstantAssignment ρ and variables in the underlying Vocabulary Σ object (a reference to it is held by ρ). If no VariableAssignments can be created, a dummy VariableAssignment χ_{dummy} is returned.

Returns A generator for all derivable VariableAssignment objects χ.

Return type `generator`

add_object(*obj*, *ascriptions=None*, *constant_symbol=None*)
Add an object s' to the calling NamedState object's underlying AttributeSystem $\mathcal{S}$, optionally update any ascriptions of the new object $\delta_i(s')$ provided and optionally bind it to a constant c' given by the `constant_symbol` parameter (if c' does not exist in the underlying Vocabulary object Σ, it will be added to Σ and furthermore all objects holding a reference to Σ will receive the update).

Parameters

- **obj** (`str`) – The new object s' to add to the NamedState.
- **ascriptions** (`dict`) – The optional ValueSets to assign to the attribute-object pairs corresponding to the new object $\delta_i(s')$.
- **constant_symbol** (`str`) – The optional constant c' to bind to the new object s'.

Raises

- **TypeError** – `obj` parameter must be a non-empty `str`, if `ascriptions` parameter is provided, it must be a `dict` and if `constant_symbol` parameter is provided, it must be a `str`.
- **ValueError** – Duplicate objects cannot be added, c' cannot be bound already and all ascriptions provided must be from an existing label of an Attribute object in the underlying AttributeSystem object $\mathcal{S}$ to s'.

get_named_alternate_extensions(**named_states*)
Obtain all alternate extensions of the calling NamedState object w.r.t. the NamedState objects $(\sigma_1; \rho_1), \ldots, (\sigma_m; \rho_m)$ provided in the `named_states` parameter, i.e., compute **AE**(Σ_i, σ) for the various applicable *i* according to algorithm.

Returns all alternative extensions of this NamedState object $(\sigma; \rho)$ w.r.t. the NamedState objects $(\sigma_1; \rho_1), \ldots, (\sigma_m; \rho_m)$ provided as optional positional arguments in the `named_states` parameter.

Return type `list`

Raises

- **TypeError** – `ns_prime` and all arguments provided as optional positional arguments to the `named_states` parameter must be NamedState objects.
- **ValueError** – At least one NamedState object must be provided in `named_states` parameter and all NamedState objects in `ns_prime` and `named_states` parameters must be proper extensions of this NamedState.

get_worlds()

Return a generator for the generation of all possible worlds $(w; \widehat{\rho})$ derivable from the calling NamedState object.

Returns A generator for the generation of all possible worlds $(w; \widehat{\rho})$ derivable from this NamedState object.

Return type `generator`

is_exhaustive(*basis*, **named_states*)

Determine if on some basis (i.e., a set of attribute-object pairs $\delta_i(s_j)$), a set of NamedState objects $(\sigma_1; \rho_1), \ldots, (\sigma_n; \rho_n)$ is exhaustive w.r.t the calling NamedState object $(\sigma; \rho)$, that is, if the ascriptions (ob objects in the basis) of the states $\sigma_1, \ldots, \sigma_n$ span the ascriptions of σ.

Parameters

- **basis** (`list`) – A list of attribute-object pairs $\delta_i(s_j)$.
- **named_states** (*NamedState*) – Any positive amount of NamedState objects $(\sigma_1; \rho_1), \ldots, (\sigma_n; \rho_n)$.

Returns Whether or not the NamedState objects $(\sigma_1; \rho_1), \ldots, (\sigma_n; \rho_n)$ provided as optional positional arguments are exhaustive w.r.t. the calling NamedState object $(\sigma; \rho)$ on the basis provided in the `basis` parameter.

Return type `bool`

Raises ValueError `basis` parameter cannot be empty and at least one NamedState object must be provided to `named_states` parameter.

is_named_alternate_extension(*ns_prime*, **named_states*)

Determine if the NamedState object in the `ns_prime` parameter $(\sigma'; \rho')$ is an alternate extension of the calling NamedState object $(\sigma; \rho)$ w.r.t. the NamedState objects provided in the `named_states` parameter $(\sigma_1; \rho_1), \ldots, (\sigma_m; \rho_m)$, i.e., evaluate $\mathit{Alt}((\sigma; \rho), \{(\sigma_1; \rho_1), \ldots, (\sigma_m; \rho_m)\}, (\sigma'; \rho'))$

Returns The result of the evaluation of $\mathit{Alt}((\sigma; \rho), \{(\sigma_1; \rho_1), \ldots, (\sigma_m; \rho_m)\}, (\sigma'; \rho'))$

Return type `bool`

Raises

- **TypeError** – `ns_prime` and all arguments provided as optional positional arguments to `named_states` parameter must be NamedState objects.
- **ValueError** – At least one NamedState object must be provided to `named_states` parameter and all NamedState objects in `ns_prime` and `named_states` parameters must be proper extensions of the calling NamedState object.

is_named_entailment(*assumption_base*, *attribute_interpretation*, **named_states*)

Determine if the calling NamedState object $(\sigma; \rho)$ entails the NamedState objects $(\sigma_1; \rho_1), \ldots, (\sigma_m; \rho_m)$ provided as optional positional arguments to the `named_states` parameter w.r.t. the AssumptionBase object β in the `assumption_base` parameter, using the AttributeInterpretation object I provided in the `attribute_interpretation` parameter to resolve truth values of the Formula objects contained therein, i.e., evaluate $(\sigma; \rho) \Vdash_\beta \{(\sigma_1; \rho_1), \ldots, (\sigma_m; \rho_m)\}$.

Parameters

- **assumption_base** (*AssumptionBase*) – The AssumptionBase object β to use when evaluating $I_{(\sigma'; \rho')/\chi}\left(\bigwedge_{F \in \beta} F\right) =$ **false** for all χ.
- **attribute_interpretation** (*AttributeInterpretation*) – The AttributeInterpretation object I to use to resolve the truth values of Formula objects in the AssumptionBase object β in `assumption_base` parameter.

- **named_states** (*NamedState*) – Any amount of NamedState objects $(\sigma_1; \rho_1), \ldots, (\sigma_m; \rho_m)$ to check for entailment.

Returns Whether or not $(\sigma; \rho) \Vdash_\beta \{(\sigma_1; \rho_1), \ldots, (\sigma_m; \rho_m)\}$

Return type `bool`

Raises

- **TypeError** – `assumption_base` parameter must be an AssumptionBase object, `attribute_interpretation` parameter must be an AttributeInterpretation object, and all optional positional arguments in `named_states` parameter must be NamedState objects.
- **ValueError** – All NamedState objects provided as optional positional arguments to the `named_states` parameter $(\sigma_1; \rho_1), \ldots, (\sigma_m; \rho_m)$ must share the same underlying Vocabulary object Σ, have equivalent AttributeSystem objects $\mathcal{S}$ and be proper extensions of the calling NamedState object $(\sigma; \rho)$, that is, $(\sigma_i; \rho_i) \sqsubset (\sigma; \rho)$ for $i = 1, \ldots, m$.

is_world()

Determine if the calling NamedState object $(\sigma; \rho)$ is a world; that is every ascription δ_i of σ is a valuation and ρ is total (that is, $\rho \to \widehat{\rho}$ holds).

Returns Whether or not the calling NamedState object is a world.

Return type `bool`

satisfies_context(*context*, *X*, *attribute_interpretation*)

Determine if this NamedState object $(w; \widehat{\rho})$ (which must be a world) satisfies the given Context object in the `context` parameter $\gamma = (\beta; (\sigma; \rho))$ w.r.t. a given VariableAssignment object χ and given AttributeInterpretation object I, i.e., $(w; \widehat{\rho}) \models_\chi \gamma$ (the calling NamedState object $(w; \widehat{\rho})$ satisfies every Formula object in the AssumptionBase object of the Context object β and the calling NamedState object satisfies the NamedState object of the Context object $(\sigma; \rho)$, that is, $(w; \widehat{\rho}) \models_\chi F$ for every $F \in \beta$ and $(w; \widehat{\rho}) \models (\sigma; \rho)$).

Parameters

- **context** (*Context*) – The context γ to check for satisfaction.
- **X** (*VariableAssignment*) – The variable assignment χ
- **attribute_interpretation** (*AttributeInterpretation*) – The fixed attribute interpretation I to use for interpreting which constants and variables are substituted into the formula for evaluation.

Returns Whether or not $(w; \widehat{\rho}) \models_\chi \gamma$.

Return type `bool`

Raises

- **TypeError** – `context` parameter must be a Context object, `X` parameter must be a VariableAssignment object and `attribute_interpretation` must be an AttributeInterpretation object.
- **ValueError** – The calling NamedState object must be a world.

satisfies_formula(*formula*, *X*, *attribute_interpretation*)

Determine if the calling NamedState object $(w; \widehat{\rho})$ (which must be a world) satisfies the given Formula object F in the `formula` parameter w.r.t. the VariableAssignment object χ in the `X` parameter and given AttributeInterpretation I in the `attribute_interpretation` parameter, i.e., $(w; \widehat{\rho}) \models_\chi F$.

Parameters

- **formula** (*Formula*) – The formula F to check for satisfaction.

- **X** (*VariableAssignment*) – The variable assignment χ
- **attribute_interpretation** (*AttributeInterpretation*) – The fixed attribute interpretation I to use for interpreting which constants and variables are substituted into the formula F for evaluation.

Returns Whether or not $(w; \widehat{\rho}) \models_\chi F$.

Return type `bool`

Raises

- **TypeError** – `formula` parameter must be a Formula object, `X` parameter must be a VariableAssignment object and `attribute_interpretation` parameter must be an AttributeInterpretation object.
- **ValueError** – The calling NamedState object must be a world.

`satisfies_named_state`(*named_state*)

Determine if the calling NamedState object $(w; \widehat{\rho})$ (which must be a world) satisfies the given NamedState object $(\sigma; \rho)$ i.e., $(w; \widehat{\rho}) \models (\sigma; \rho)$.

Parameters **named_state** (*NamedState*) – The named state $(\sigma; \rho)$ to check for satisfaction.

Returns Whether or not $(w; \widehat{\rho}) \models (\sigma; \rho)$.

Return type `bool`

Raises

- **TypeError** – `named_state` parameter must be a NamedState object.
- **ValueError** – The calling NamedState object must be a world.

A9 Attribute Interpretations

A9.1 The AttributeInterpretation object

This subsection introduces the AttributeInterpretation class.

class `attribute_interpretation.`**`AttributeInterpretation`**(*vocabulary*, *attribute_structure*, *mapping*, *profiles*)

Analogous to the definition of attribute interpretations in [Arkoudas and Bringsjord, 2009], AttributeInterpretation objects build an interpretation table; that is, a mapping I that assigns, to each RelationSymbol object of a Vocabulary object Σ, $R \in$ R of arity n:

1. a relation $R^I \in \mathcal{R}$ of some arity *m*, called the **realization** of *R*:

$$R^I \subset A_{i_1} \times \cdots \times A_{i_m}$$

(where we might have $m \neq n$); and

2. a list of *m* pairs

$$[(l_{i_1}; j_1) \cdots (l_{i_m}; j_m)]$$

called the **profile** of *R* and denoted by $Prof(R)$, with $1 \leq j_x \leq n$ for $x = 1, \ldots, m$

Variables

- **vocabulary** – The Vocabulary object Σ of the interpretation.
- **attribute_structure** – The AttributeStructure object the interpretation is into.
- **mapping** – The mapping R $\rightarrow \mathcal{R}$.
- **profiles** – A list of profiles $[(l_{i_1}; j_1) \cdots (l_{i_m}; j_m)]$; one for each realization.
- **table** – The interpretation table of the attribute interpretation.
- **relation_symbols** – A copy of the RelationSymbol objects from Σ (for convenient access).
- **is_AttributeInterpretation** – An identifier to use in place of `type` or `isinstance`.

`__deepcopy__`(*memo*)

Deepcopy an AttributeInterpretation object via the `copy.deepcopy` method. This does not break the reference to the underlying Vocabulary object Σ.

`__eq__`(*other*)

Determine if two AttributeInterpretation objects are equal via the == operator.

`__init__`(*vocabulary*, *attribute_structure*, *mapping*, *profiles*)

Construct an AttributeInterpretation object.

Parameters

- **vocabulary** (*Vocabulary*) – The Vocabulary object Σ to define the AttributeInterpretation over.
- **attribute_structure** (*AttributeStructure*) – The AttributeStructure object for which to define the AttributeInterpretation into.
- **mapping** (`dict`) – The mapping from the RelationSymbol objects of the Vocabulary object Σ in the `vocabulary` parameter to the Relation objects of the AttributeStructure object in the `attribute_structure` parameter; the keys being RelationSymbol objects and values being `int`s corresponding to the subscripts of the AttributeStructure Relation objects.

- **profiles** (`list`) – A list of realizations corresponding to the mapping wherein each element is a list where the first element is the RelationSymbol and the following elements are 2-tuples $(l_{i_k}; j_k)$.

Raises

- **TypeError** – `vocabulary` parameter must be a Vocabulary object, `attribute_structure` parameter must be an AttributeStructure object, `mapping` parameter myst be a `dict` and `profile` parameter must be a `list`.
- **ValueError** – All keys in the `mapping` parameter must be RelationSymbol objects and all values must be unique `int`s, duplicate profiles are not permitted (determined by repeated RelationSymbol objects), the set of RelationSymbol's provided in the `mapping` parameter and the set of RelationSymbol's in the `vocabulary` parameter and the set of RelationSymbol's provided in the `profile` parameter must all be equal, all subscripts provided in the `mapping` parameter keys must be valid subscripts of Relation objects in the `attribute_structure` parameter, all l_{i_k} in $(l_{i_k}; j_k)$ 2-tuples provided in each realization of the `profile` parameter must be a label of some Attribute object in the `attribute_structure` parameter, and all j_k in $(l_{i_k}; j_k)$ 2-tuples provided in each realization of the `profile` parameter must be between 1 and the arity of the RelationSymbol object of the realization.

`__iter__``()`
Provide an iterator for the interpretation table of AttributeInterpretation objects.

(e.g. "`for entry in attribute_interpretation:`")

`__ne__`(*other*)
Determine if two AttributeInterpretation objects are not equal via the `!=` operator.

`__repr__``()`
Return a string representation of the AttributeInterpretation object.

`__str__``()`
Return a readable string representation of the AttributeInterpretation object.

A10 Formulae and Assumption Bases

A10.1 The Formula object

This subsection introduces the Formula class.

class `formula.`**`Formula`**(*vocabulary*, *name*, **terms*)

Formula objects are defined over some Vocabulary Σ. Formula objects are immutable.

Variables

- **vocabulary** – The underlying Vocabulary object Σ the Formula is defined over.
- **name** – The name of the Formula object.
- **terms** – The terms of the Formula object.
- **is_Formula** – An identifier to use in place of `type` or `isinstance`.

`__add__`(*other*)

Combine a Formula object and another Formula object or an AssumptionBase object into an AssumptionBase object via the + operator.

Raises

- **TypeError** – Only a Formula object or AssumptionBase object can be combined with a Formula object.
- **ValueError** – This Formula and `other` parameter must share the same underlying Vocabulary object Σ and duplicate Formula objects are not permitted (determined by name of the Formula object only).

`__deepcopy__`(*memo*)

Deepcopy a Formula object via the `copy.deepcopy` method. This does not break the reference to the underlying Vocabulary object Σ.

`__eq__`(*other*)

Determine if two Formula objects are equal via the == operator.

`__hash__`()

Hash implementation for set functionality of Formula objects.

`__init__`(*vocabulary*, *name*, **terms*)

Construct a Formula object.

Parameters

- **vocabulary** (*Vocabulary*) – The underlying Vocabulary object Σ the Formula is defined over.
- **name** (`str`) – The name (identifier) of the formula.
- **terms** (`str`) – Any amount of `str` constants and variables representing the terms of the formula.

Raises

- **TypeError** – `vocabulary` parameter must be a Vocabulary object.
- **ValueError** – `name` parameter must match some RelationSymbol object in the `vocabulary` parameter, at least one term must be provided and all terms provided must be in either the constants or variables of the `vocabulary` parameter Σ.

`__ne__`(*other*)

Determine if two Formula objects are not equal via the != operator.

__repr__()
Return a string representation of the NamedState object.

__str__()
Return a readable string representation of the NamedState object.

_key()
Private key function for hashing.

Returns tuple consisting of (name, $t_1, \ldots, t_n$)

Return type tuple

assign_truth_value(*attribute_interpretation*, *named_state*, *X*)
Assign a truth value in {**true**, **false**, **unknown**} to the calling Formula object F given an arbitrary NamedState object $(\sigma; \rho)$ in the named_state parameter and VariableAssignment object χ in the X parameter w.r.t. an AttributeInterpretation object I.

This function makes use of the ParserSet object; the ParserSet object is a key part in the vivid object extension protocol.

The assign_truth_value function works as follows:

1. Find the entry in the interpretation table of the AttributeInterpretation object I in the attribute_interpretation parameter and extract the corresponding profile and Relation object (the 3rd element of the corresponding row of the table is the identifier for the Relation object; e.g. $R_{subscript}$).

2. Substitute the terms of the Formula object F into the profile (the 2nd element of each pair in the profile corresponds to the index of the term in the F to use, shifted down by 1).

3. Using the ConstantAssignment object of the named_state parameter ρ and the VariableAssignment in the X parameter χ, substitute for each term now in the profile, the object corresponding to that term given by the mapping in ρ or χ (if the term is in neither ρ nor χ, "unknown" is returned as the truth value).

4. The profile now consists of the attribute-object pairs ($\delta_i(s_j)$ for some set of the possible values of i and j) to use in the Relation object's definition when creating the evaluatable expression. Now, all worlds $(w; \widehat{\rho})$ derivable from the NamedState are generated and the ValueSets of the attribute-object pairs in the profile (consisting of single elements) are extracted from the ascriptions of these worlds.

5. The single element ValueSets are zipped together with the arguments in the Relation object definition (the ith attribute-object pair of the profile is zipped with the ith argument of the definition) and these new argument-ValueSet pairs are used to substitute every occurance of each argument in the definition with the corresponding single element ValueSet creating a (hopefully) evaluatable expression (the RHS of the substituted definition) for each world $(w; \widehat{\rho})$.

6. Each parser in the ParserSet object will then try to evaluate the expression and save the truth value for each $(w; \widehat{\rho})$. If some expression is unevaluatable for all parsers in the ParserSet a ValueError is raised.

7. If the expression of every world $(w; \widehat{\rho})$ evaluates to True, the truth value returned is **true**, if the expression of every world evaluates to False, the truth value returned is **false** and if the expressions of any two worlds evaluate to different values, the truth value returned is **unknown**.

Returns A truth value in the set {**true**, **false**, **unknown**}

Return type bool | str

Raises

- **TypeError** – attribute_interpretation parameter must be an AttributeInterpretation object, named_state parameter must be a NamedState object and X parameter must be a VariableAssignment object.

- **ValueError** – This Formula object, the AttributeInterpretation object in the `attribute_interpretation` parameter, the NamedState object in the `named_state` parameter and the VariableAssignment object in the `X` parameter must all share the same underlying Vocabulary object (that is F, I, $(\sigma;\rho)$ and χ must all share the same Σ), the Formula object must match an entry in the interpretation table of the AttributeInterpretation I in the

`attribute_interpretation` parameter, the number of attribute-object pairs in the profile corresponding to the Formula must match the arity of the corresponding Relation object found in the table (where the Relation object is found in the AttributeStructure object in the AttributeSystem member of the `named_state` parameter), $1 \leq j_x \leq n$ for each j_x in the profile (where *n* is the arity of the RelationSymbol corresponding to the RelationSymbol object matching the Formula in the interpretation table, or equivalently, the number of terms in the Formula object) and a parser in the ParserSet object must be able to evaluate the expression obtained after substituting the objects of the AttributeSystem in the `named_state` parameter, corresponding to the terms of the Formula, into the Relation object's definition.

static `get_basis`(*constant_assignment*, *variable_assignment*, *attribute_interpretation*, **formulae*)

Get the basis of the Formula objects $F_1, \ldots, F_k$ provided as optional positional arguments in the `formulae` parameter w.r.t. the ConstantAssignment object ρ provided in the `constant_assignment` parameter, VariableAssignment object χ provided in the `variable_assignment` parameter, and the AttributeInterpretation object I provided in the `attribute_interpretation` parameter, i.e., compute $\mathcal{B}(F_1, \rho, \chi) \cup \cdots \cup \mathcal{B}(F_k, \rho, \chi)$.

Parameters

- **constant_assignment** (*ConstantAssignment*) – The ConstantAssignment object ρ to use to compile the profile corresponding to each Formula object $F_i, i = 1, \ldots, k$ into attribute-object pairs to consider for the basis.
- **variable_assignment** (VariableAssignment | `None`) – The VariableAssignment object χ to use to compile the profile corresponding to each Formula object $F_i, i = 1, \ldots, k$ into attribute-object pairs to consider for the basis or `None`.
- **attribute_interpretation** (*AttributeInterpretation*) – The AttributeInterpretation object I to use to determine the profiles corresponding to the Formula objects $F_1, \ldots, F_k$ provided (the profile is extracted from the interpretation table when the RelationSymbol matching the Formula object's name is found).
- **formulae** (*Formula*) – Any positive amount of Formula objects $F_1, \ldots, F_k$ to consider in the basis.

Returns A list of attribute-object pairs comprising the basis of the Formula objects $F_1, \ldots, F_k$ provided w.r.t. ρ and χ.

Return type `list`

Raises

- **TypeError** – `constant_assignment` parameter must be a ConstantAssignment object, and all optional positional arguments provided in the `formulae` parameter must be Formula objects.
- **ValueError** – At least one Formula object must be provided and all Formula objects provided must match some entry in the interpretation table of the AttributeInterpretation object I.

A10.2 The AssumptionBase object

This subsection introduces the AssumptionBase class.

class `assumption_base.`**`AssumptionBase`**(**formulae*)

Analogous to the definition of assumption bases in [Arkoudas and Bringsjord, 2009], an AssumptionBase object functions as a container for a finite set of Formula objects $F_1, \ldots, F_k$ over a single underlying Vocabulary Σ, denoted β.

Variables

- **formulae** – The set of Formula objects $F_1, \ldots, F_k$ contained in the AssumptionBase object β.
- **vocabulary** – The underlying Vocabulary object Σ the AssumptionBase object β is defined over.
- **_is_AssumptionBase** – An identifier to use in place of `type` or `isinstance`.

`__add__`(*other*)

Add all Formula objects in another AssumptionBase object or a single Formula object to the calling AssumptionBase object via the + operator.

Raises

- **TypeError** – Only Formula objects or AssumptionBase objects can be added to the calling AssumptionBase object.
- **ValueError** – Cannot add objects with different underlying Vocabulary objects and duplicate Formula objects are not permitted.

`__contains__`(*item*)

Overloaded `in` operator for AssumptionBase. Determine if a Formula object is contained in the calling AssumptionBase object.

Parameters **key** (Formula | `str`) – The Formula object or the name of a Formula object to test for membership in the calling AssumptionBase object.

`__deepcopy__`(*memo*)

Deepcopy an AssumptionBase object via the `copy.deepcopy` method. This does not break the reference to the underlying Vocabulary object Σ.

`__eq__`(*other*)

Determine if two AssumptionBase objects are equal via the == operator.

`__getitem__`(*key*)

Retrive the Formula object corresponding to the key given by the `key` parameter via indexing (e.g. `AssumptionBase[key]`).

Parameters **key** (`int` | `str` | Formula) – The key to use for indexing in the calling AssumptionBase object.

Raises

- **IndexError** – `int` key is out of range.
- **KeyError** – `key` parameter does not correspond to any Formula object in the AssumptionBase.
- **TypeError** – `key` parameter must be an `int`, `str`, or Formula object.

`__iadd__`(*other*)

Add all Formula objects in another AssumptionBase object or a single Formula object to the calling AssumptionBase object via the + operator.

Raises

- **TypeError** – Only Formula objects or AssumptionBase objects can be added to the calling AssumptionBase object.
- **ValueError** – Cannot add objects with different underlying Vocabulary objects and duplicate Formula objects are not permitted.

__init__(**formulae*)

Construct an AssumptionBase object.

Parameters formulae (*Formula|Vocabulary*) – Any amount of Formula objects $F_1, \ldots, F_k$ or a single Vocabulary object Σ no Formula objects are provided.

Raises

- **TypeError** – All optional positional arguments provided must be Formula objects or a single Vocabulary object Σ.
- **ValueError** – All Formula objects provided as optional positional arguments must share the same Vocabulary object Σ.

__iter__()

Provide an iterator for AssumptionBase objects (e.g. "`for formula in AssumptionBase:`").

__len__()

Determine the length of an AssumptionBase object β (i.e., the amount of Formula objects contained in β) via the `len` built-in function e.g.(`len(AssumptionBase)`).

__ne__(*other*)

Determine if two AssumptionBase objects are not equal via the `!=` operator.

__repr__()

Return a string representation of the AssumptionBase object.

__str__()

Return a readable string representation of the AssumptionBase object.

A11 Contexts

A11.1 The Context object

This subsection introduces the Context class.

class `context.`**`Context`**(*assumption_base*, *named_state*)

Analogous to the definition of contexts in [Arkoudas and Bringsjord, 2009], a Context wraps a pair composed of an AssumptionBase object β and a NamedState object $(\sigma; \rho)$, i.e., $\gamma = (\beta; (\sigma; \rho))$.

Variables

- **assumption_base** – The AssumptionBase object β of the Context object.
- **named_state** – The NamedState object $(\sigma; \rho)$ of the Context object.
- **is_Context** – An identifier to use in place of `type` or `isinstance`.

`__deepcopy__`(*memo*)

Deepcopy a Context object via the `copy.deepcopy` method. This does not break the reference to the underlying Vocabulary object Σ.

`__eq__`(*other*)

Determine if two Context objects are equal via the == operator.

`__init__`(*assumption_base*, *named_state*)

Construct a Context object.

Parameters

- **assumption_base** (*AssumptionBase*) – The AssumptionBase object β to use in the Context object.
- **named_state** (*NamedState*) – The NamedState object $(\sigma; \rho)$ to use in the Context object.

Raises

- **TypeError** – `assumption_base` parameter must be an AssumptionBase object and `named_state` parameter must be a NamedState object.
- **ValueError** – The underlying Vocabulary objects of the `assumption_base` and `named_state` parameters must be the same Vocabulary object Σ.

`__ne__`(*other*)

Determine if two Context objects are not equal via the != operator.

`__repr__`()

Return a string representation of the Context object.

`__str__`()

Return a readable string representation of the Context object.

`entails_formula`(*formula*, *attribute_interpretation*)

Determine if the calling Context object $\gamma = (\beta; (\sigma; \rho))$ entails the Formula object F provided in the `formula` parameter, using the AttributeInterpretation object I provided in the `attribute_interpretation` parameter to interpret truth values, i.e., determine if $\gamma \models F$.

Parameters

- **formula** (*Formula*) – The Formula object F to check for entailment.
- **attribute_interpretation** (*AttributeInterpretation*) – The AttributeInterpretation object I to use for the interpretation of truth values during the evaluation of $\gamma \models F$.

Returns Whether or not $\gamma \models F$, that is, whether or not $(w; \widehat{\rho}) \models_\chi \gamma$ implies $(w; \widehat{\rho}) \models_\chi F$ for all worlds $(w; \widehat{\rho})$ and variable assignments χ.

Return type `bool`

Raises

- **TypeError** – `formula` parameter must be a Formula object and `attribute_interpretation` parameter must be an AttributeInterpretation object.
- **ValueError** – The calling Context object and the Formula object F provided in the `formula` parameter must share the same underlying Vocabulary object Σ.

`entails_named_state`(*named_state*, *attribute_interpretation*)

Determine if the calling Context object $\gamma = (\beta; (\sigma; \rho))$ entails the NamedState object $(\sigma'; \rho')$ provided in the `named_state` parameter, using the AttributeInterpretation object I provided in the `attribute_interpretation` parameter to interpret truth values, i.e., determine if $\gamma \models (\sigma'; \rho')$.

Parameters

- **named_state** (*NamedState*) – The NamedState object $(\sigma'; \rho')$ to check for entailment.
- **attribute_interpretation** (*AttributeInterpretation*) – The AttributeInterpretation object I to use for the interpretation of truth values during the evauation of $\gamma \models (\sigma'; \rho')$.

Returns Whether or not $\gamma \models (\sigma'; \rho')$, that is for all worlds $(w; \widehat{\rho})$ and variable assignments χ, $(w; \widehat{\rho}) \models (\sigma'; \rho')$ whenever $(w; \widehat{\rho}) \models_\chi \gamma$.

Return type `bool`

Raises

- **TypeError** – `named_state` parameter must be a NamedState object and `attribute_interpretation` parameter must be an AttributeInterpretation object.
- **ValueError** – The calling Context object and the NamedState object $(\sigma'; \rho')$ provided in the `named_state` parameter must share the same underlying Vocabulary object Σ.

A12 Rules of Inference for Diagrammatic Deductions

This section introduces the rules of inference given in [Arkoudas and Bringsjord, 2009] for diagrammatic deductions which are supported by pyVivid.

A12.1 The [Thinning] Rule

`inference_rules.`**`thinning`**(*context*, *named_state*, *assumption_base=None*, *attribute_interpretation=None*)

Verify that the NamedState object $(\sigma'; \rho')$ in the `named_state` parameter can be obtained by thinning from the NamedState object $(\sigma; \rho)$ contained in the Context object $(\beta; (\sigma; \rho))$ in the `context` parameter w.r.t. the AssumptionBase object $\{F_1, \ldots, F_n\}$ given by the `assumption_base` parameter, using the AttributeInterpretation object I in the `attribute_interpretation` parameter to interpret truth values.

By Corollary 26, if $(\sigma; \rho) \Vdash_{\{F_1,\ldots,F_n\}} (\sigma'; \rho')$ then $(\{F_1, \ldots, F_n\}; (\sigma; \rho)) \models (\sigma'; \rho')$.

Then, by weakening, $(\beta \cup \{F_1, \ldots, F_n\}; (\sigma; \rho)) \models (\sigma'; \rho')$ and thinning holds, thus it suffices to show that a call to `entails_named_state` with context $(\{F_1, \ldots, F_n\}; (\sigma; \rho))$ and named state $(\sigma'; \rho')$, that is $(\sigma; \rho) \Vdash_{\{F_1,\ldots,F_n\}} (\sigma'; \rho')$, holds to show that thinning holds.

Parameters

- **context** (*Context*) – The Context object $(\beta; (\sigma; \rho))$.
- **named_state** (*NamedState*) – The NamedState object $(\sigma'; \rho')$
- **assumption_base** (AssumptionBase | `None`) – The set of Formula objects to thin with $\{F_1, \ldots, F_n\}$ if thinning is to be done with any Formula (i.e., $n > 0$), otherwise `None`.
- **attribute_interpretation** (AttributeInterpretation | `None`) – The AttributeInterpretation object I to use to interpret truth values if $n > 0$, otherwise `None`.

Returns Whether or not thinning holds, i.e., the result of $(\sigma; \rho) \Vdash_{\{F_1,\ldots,F_n\}} (\sigma'; \rho')$

Return type `bool`

Raises TypeError `context` parameter must be a Context object and `named_state` parameter must be a NamedState object.

A12.2 The [Widening] Rule

`inference_rules.`**`widening`**(*context*, *named_state*, *attribute_interpretation=None*)

Verify that the NamedState object $(\sigma'; \rho')$ in the `named_state` parameter can be obtained from the Context object $(\beta; (\sigma; \rho))$ in the `context` parameter by widening, using the AttributeInterpretation object I in the `attribute_interpretation` parameter to interpret truth values.

Parameters

- **context** (*Context*) – The Context object $(\beta; (\sigma; \rho))$.
- **named_state** (*NamedState*) – The NamedState object $(\sigma'; \rho')$
- **attribute_interpretation** (AttributeInterpretation | `None`) – The AttributeInterpretation object I to use to interpret truth values if widening should consider the AssumptionBase object of the `context` parameter, otherwise `None`.

Returns Whether or not the NamedState object $(\sigma'; \rho')$ in the `named_state` parameter can be obtained from the Context object $(\beta; (\sigma; \rho))$ in the `context` parameter by widening, i.e., whether or not $(\beta; (\sigma; \rho)) \models (\sigma'; \rho')$

Return type `bool`

Raises TypeError `context` parameter must be a Context object and `named_state` parameter must be a NamedState object.

A12.3 The [Observe] Rule

`inference_rules.`**`observe`**(*context*, *formula*, *attribute_interpretation*)

Determine if the Formula object F given by the `formula` parameter can be observed in the Context object $(\beta; (\sigma; \rho))$ given by the `context` parameter, using the AttributeInterpretation object I in the `attribute_interpretation` parameter to interpret truth values, i.e., determine if **observe** F holds in $(\beta; (\sigma; \rho))$.

Parameters

- **context** (*Context*) – The Context object $(\beta; (\sigma; \rho))$ in which the Formula object F can potentially be observed.
- **formula** (*Formula*) – The (potentially) observable Formula object F.
- **attribute_interpretation** (*AttributeInterpretation*) – The AttributeInterpretation object I to use to interpet truth values in the `context` and `formula` parameters.

Returns Whether or not **observe** F holds in $(\beta; (\sigma; \rho))$, that is, whether or not $(\beta; (\sigma; \rho)) \models F$.

Return type `bool`

A12.4 The [Absurdity] Rule

`inference_rules.`**`diagrammatic_absurdity`**(*context*, *named_state*, *attribute_interpretation*)

Verify that the NamedState object $(\sigma'; \rho')$ in the `named_state` parameter can be obtained from the Context object $(\beta; (\sigma; \rho))$ in the `context` parameter by absurdity, using the AttributeInterpretation object I provided in the `attribute_interpretation` parameter to interpet truth values.

To show $(\sigma'; \rho')$ **by absurdity**, we must show $(\beta \cup \{\textbf{false}\}; (\sigma; \rho)) \models (\sigma'; \rho')$.

By lemma 20, $(\beta \cup \{\textbf{false}\}; (\sigma; \rho)) \models (\sigma'; \rho')$, thus it suffices to show that a call to `entails_named_state` with context $(\beta; (\sigma; \rho))$ and named state $(\sigma'; \rho')$, that is, $(\beta; (\sigma; \rho)) \models (\sigma'; \rho')$ holds, implicitly assuming that some $F \in \beta$ evaulates to **false** (as then no world can satisify the context, i.e., for any world $(w; \widehat{\rho})$ derivable from the context $(\beta; (\sigma; \rho))$, $(w; \widehat{\rho}) \not\models_\chi (\beta; (\sigma; \rho))$ and thus `entails_named_state` will always hold yielding $(\sigma'; \rho')$ **by absurdity** regardless of the NamedState object $(\sigma'; \rho')$ provided in the `named_state` parameter)

Parameters

- **context** (*Context*) – The Context object $(\beta; (\sigma; \rho))$.
- **named_state** (*NamedState*) – The NamedState object $(\sigma'; \rho')$.
- **attribute_interpretation** (*AttributeInterpretation*) –

Returns Whether or not $(\sigma'; \rho')$ **by absurdity**, that is, whether or not $(\beta; (\sigma; \rho)) \models (\sigma'; \rho')$ holds.

Return type `bool`

Raises TypeError `context` parameter must be a Context object, `named_state` parameter must be a NamedState object and `attribute_interpretation` parameter must be an AttributeInterpretation object.

A12.5 The [Diagram-Reiteration] Rule

`inference_rules.`**`diagram_reiteration`**(*context*)

Perform $[Diagram-Reiteration]$ to retrieve the current diagram of the Context object $(\beta; (\sigma; \rho))$ provided in the `context` parameter, i.e., from lemma 19: $(\beta; (\sigma; \rho)) \models (\sigma; \rho)$.

Parameters **context** (*Context*) – The Context object $(\beta; (\sigma; \rho))$ from which to retrieve the current NamedState object $(\sigma; \rho)$.

Returns The NamedState object $(\sigma; \rho)$ of the Context object $(\beta; (\sigma; \rho))$ in `context` parameter.

Return type NamedState

A12.6 The [Sentential-to-Sentential] Rule

`inference_rules.`**`sentential_to_sentential`**(*context*, *F1*, *F2*, *G*, *attribute_interpretation*, *variable_assignment=None*)

Verify that a disjunction $F_1 \vee F_2$ holds in the Context object $(\beta; (\sigma; \rho))$ in the `context` parameter and that the Formula object G in the `G` parameter follows in either case, using the AttributeInterpretation object I in the `attribute_interpretation` parameter to interpret truth values.

To perform the **sentential-to-sentential** inference, first the disjunction $F_1 \vee F_2$ is verified. Then the truth values of $F_1 \Rightarrow G$ and $F_1 \Rightarrow G$ are determined. If either $F_1 \Rightarrow G$ or $F_1 \Rightarrow G$ do not hold, then **sentential-to-sentential** does not hold, otherwise, **sentential-to-sentential** holds.

Parameters

- **context** (*Context*) – The Context object $(\beta; (\sigma; \rho))$ in which the the Formula objects in the parameters `F1`, `F2` apply and in which the Formula object in the `G` parameter would follow.
- **F1** (*Formula*) – The left operand of the disjunction F_1.
- **F2** (*Formula*) – The right operand of the disjunction F_2.
- **G** (*Formula*) – The Formula object G potentially following the disjunction in either case.
- **attribute_interpretation** (*AttributeInterpretation*) – The AttributeInterpretation object I to use for interpeting truth values.
- **variable_assignment** (VariableAssignment | `None`) – The optional VariableAssignment object χ to consider in the interpretation of truth values.

Returns Whether or not **sentential-to-sentential** holds.

Return type `bool`

Raises ValueError The disjunction $F_1 \vee F_2$ does not hold.

A12.7 The [Diagrammatic-to-Diagrammatic] Rule

`inference_rules.`**`diagrammatic_to_diagrammatic`**(*context*, *inferred_named_state*, *named_states*, *attribute_interpretation*, *variable_assignment*, **formulae*)

Verify that on the basis of the present diagram $(\sigma; \rho)$ of the Context object $(\beta; (\sigma; \rho))$ in the `context` parameter and some set of Formula objects $F_1, \ldots, F_k, k \geq 0$ provided as optional positional arguments in the `formulae` parameter, that for each NamedState object

$(\sigma_1; \rho_1), \ldots, (\sigma_n; \rho_n), n > 0$, contained in the `named_states` parameter, a NamedState object $(\sigma'; \rho')$ provided in the `inferred_named_state` parameter can be derived in every one of these n cases.

This is rule $[C_1]$.

This function works as follows:

1. If $k > 0$ (i.e., if at least one Formula object is provided as an optional positional argument to the `formulae` parameter), compute the basis $\mathcal{B}(F_1, \rho, \chi) \cup \cdots \cup \mathcal{B}(F_k, \rho, \chi)$ of $F_1, \ldots, F_k$ and determine if the NamedState objects $(\sigma_1; \rho_1), \ldots, (\sigma_n; \rho_n)$ provided in the `named_states` parameter form an exhuastive set of possibilities on this basis.

2. Determine if the proviso $(\sigma; \rho) \Vdash_{\{F_1, \ldots, F_k\}} \{(\sigma_1; \rho_1), \ldots, (\sigma_n; \rho_n)\}$ (where $k \geq 0$) holds.

3. Return the evaluation of $(\beta \cup \{F_1, \ldots, F_k\}; (\sigma; \rho)) \models (\sigma'; \rho')$.

Parameters

- **context** (*Context*) – The Context object $(\beta; (\sigma; \rho))$ from which the present diagram $(\sigma; \rho)$ comes from.
- **inferred_named_state** (*NamedState*) – The NamedState object $(\sigma'; \rho')$ derivable in the $n > 0$ cases provided by the NamedState objects $(\sigma_1; \rho_1), \ldots, (\sigma_n; \rho_n)$ in the `named_state` parameter.
- **named_states** (`list`) – The NamedState objects $(\sigma_1; \rho_1), \ldots, (\sigma_n; \rho_n), n > 0$ functioning as the set of n exhaustive cases from which F can be derived.
- **attribute_interpretation** (*AttributeInterpretation*) – The AttributeInterpretation object I to use for the interpretation of truth values and the computation of the basis of $F_1, \ldots, F_k$.
- **variable_assignment** (VariableAssignment | `None`) – The VariableAssignment object χ to consider when computing the basis $\mathcal{B}(F_1, \rho, \chi) \cup \cdots \cup \mathcal{B}(F_k, \rho, \chi)$ of $F_1, \ldots, F_k$ or `None` if all terms of the $F_1, \ldots, F_k$ are in ρ.
- **formulae** (*Formula*) – The $k \geq 0$ Formula objects $F_1, \ldots, F_k$ to use in the computation of the basis, computation of the proviso and the evaluation of $(\beta \cup \{F_1, \ldots, F_k\}; (\sigma; \rho)) \models (\sigma'; \rho')$.

Returns The result of the evaluation of $(\beta \cup \{F_1, \ldots, F_k\}; (\sigma; \rho)) \models (\sigma'; \rho')$.

Return type `bool`

Raises ValueError If $k > 0$, the NamedState objects $(\sigma_1; \rho_1), \ldots, (\sigma_n; \rho_n), n > 0$ are not exhaustive on the basis of the Formula objects $F_1, \ldots, F_k$ or the proviso $(\sigma; \rho) \Vdash_{\{F_1, \ldots, F_k\}} \{(\sigma_1; \rho_1), \ldots, (\sigma_n; \rho_n)\}$ (where $k \geq 0$) does not hold.

A12.8 The [Sentential-to-Diagrammatic] Rule

`inference_rules.`**`sentential_to_diagrammatic`**(*context*, *F1*, *F2*, *named_state*, *attribute_interpretation*, *variable_assignment=None*)

Verify that a disjunction $F_1 \vee F_2$ holds in the Context object $(\beta; (\sigma; \rho))$ in the `context` parameter and that the NamedState object $(\sigma'; \rho')$ in the `named_state` parameter follows in either case, using the AttributeInterpretation object I in the `attribute_interpretation` parameter to interpet truth values.

This is rule $[C_2]$.

This function works as follows:

1. Verify that the disjunction $F_1 \cup F_2$ given by `F1` and `F2` parameters holds in the Context object $(\beta; (\sigma; \rho))$ given by `context` parameter.

2. Generate two new Context objects $\gamma_1 = (\beta_1; (\sigma; \rho))$ where $\beta_1 = \beta \cup F_1$ and $\gamma_2 = (\beta_2; (\sigma; \rho))$ where $\beta_2 = \beta \cup F_2$.

3. For all possible worlds $\left(w'; \widehat{\rho'}\right)$ and variable assignments χ of the NamedState object $(\sigma'; \rho')$, determine if the world $\left(w'; \widehat{\rho'}\right)$ satisfies both Context objects γ_1 and γ_2, that is $\left(w'; \widehat{\rho'}\right) \models \gamma_1 \bigwedge \left(w'; \widehat{\rho'}\right) \models \gamma_2$.

4. If any world $\left(w'; \widehat{\rho'}\right)$ and variable assignments χ of the NamedState object $(\sigma'; \rho')$ does not satisify both γ_1 and γ_2, then **sentential-to-diagrammatic** does not hold, otherwise, **sentential-to-diagrammatic** holds.

In this way, we capture the idea that any world $\left(w'; \widehat{\rho'}\right)$ and variable assignments χ of the NamedState object $(\sigma'; \rho')$ (and thus the NamedState object $(\sigma'; \rho')$ itself) follows from the context $(\beta \cup \{F_1 \vee F_2\}; (\sigma; \rho))$ in either case of the disjunction $F_1 \vee F_2$.

Parameters

- **context** (*Context*) – The Context object $(\beta; (\sigma; \rho))$ in which the Formula objects in the parameters `F1` and `F2` apply and in which the NamedState object $(\sigma'; \rho')$ in `named_state` parameter would follow.
- **F1** (*Formula*) – The left operand of the disjunction F_1.
- **F2** (*Formula*) – The right operand of the disjunction F_2.
- **named_state** (*NamedState*) – The NamedState object $(\sigma'; \rho')$ potentially following the disjunction in either case.
- **attribute_interpretation** (*AttributeInterpretation*) – The AttributeInterpretation object I to use for the interpretation of truth values.
- **variable_assignment** (VariableAssignment | `None`) – The optional VariableAssignment object χ to consider in the interpretation of truth values.

Returns Whether or not **sentential-to-diagrammatic** holds.

Return type `bool`

Raises ValueError The disjunction $F_1 \vee F_2$ does not hold.

A12.9 The [Diagrammatic-to-Sentential] Rule

`inference_rules.`**`diagrammatic_to_sentential`**(*context*, *F*, *named_states*, *attribute_interpretation*, *variable_assignment*, **formulae*)

Verify that on the basis of the present diagram $(\sigma; \rho)$ of the Context object $(\beta; (\sigma; \rho))$ in the `context` parameter and some set of Formula objects $F_1, \ldots, F_k, k \geq 0$ provided as optional positional arguments in the `formulae` parameter, that for each NamedState object $(\sigma_1; \rho_1), \ldots, (\sigma_n; \rho_n), n > 0$, contained in the `named_states` parameter, a Formula object F provided in the `F` parameter can be derived in every one of these n cases.

This is rule $[C_3]$.

This function works as follows:

1. If $k > 0$ (i.e., if at least one Formula object is provided as an optional positional argument to the `formulae` parameter), compute the basis $\mathcal{B}(F_1, \rho, \chi) \cup \cdots \cup \mathcal{B}(F_k, \rho, \chi)$ of $F_1, \ldots, F_k$ and determine if the NamedState objects $(\sigma_1; \rho_1), \ldots, (\sigma_n; \rho_n)$ provided in the `named_states` parameter form an exhuastive set of possibilities on this basis.

2. Determine if the proviso $(\sigma; \rho) \Vdash_{\{F_1, \ldots, F_k\}} \{(\sigma_1; \rho_1), \ldots, (\sigma_n; \rho_n)\}$ (where $k \geq 0$) holds.

3. Return the evaluation of $(\beta \cup \{F_1, \ldots, F_k\}; (\sigma; \rho)) \models F$.

Parameters

- **context** (*Context*) – The Context object $(\beta; (\sigma; \rho))$ from which the present diagram $(\sigma; \rho)$ comes from.
- **F** (*Formula*) – The Formula object F derivable in the $n > 0$ cases provided by the NamedState objects $(\sigma_1; \rho_1), \ldots, (\sigma_n; \rho_n)$ in the `named_state` parameter.
- **named_states** (`list`) – The NamedState objects $(\sigma_1; \rho_1), \ldots, (\sigma_n; \rho_n), n > 0$ functioning as the set of n exhaustive cases from which F can be derived.
- **attribute_interpretation** (*AttributeInterpretation*) – The AttributeInterpretation object I to use for the interpretation of truth values and the computation of the basis of $F_1, \ldots, F_k$.
- **variable_assignment** (VariableAssignment | `None`) – The VariableAssignment object χ to consider when computing the basis $\mathcal{B}(F_1, \rho, \chi) \cup \cdots \cup \mathcal{B}(F_k, \rho, \chi)$ of $F_1, \ldots, F_k$ or `None` if all terms of the $F_1, \ldots, F_k$ are in ρ.
- **formulae** (*Formula*) – The $k \geq 0$ Formula objects $F_1, \ldots, F_k$ to use in the computation of the basis, computation of the proviso and the evaluation of $(\beta \cup \{F_1, \ldots, F_k\}; (\sigma; \rho)) \models F$.

Returns The result of the evaluation of $(\beta \cup \{F_1, \ldots, F_k\}; (\sigma; \rho)) \models F$.

Return type `bool`

Raises ValueError If $k > 0$, the NamedState objects $(\sigma_1; \rho_1), \ldots, (\sigma_n; \rho_n), n > 0$ are not exhaustive on the basis of the Formula objects $F_1, \ldots, F_k$ or the proviso $(\sigma; \rho) \Vdash_{\{F_1, \ldots, F_k\}} \{(\sigma_1; \rho_1), \ldots, (\sigma_n; \rho_n)\}$ (where $k \geq 0$) does not hold.

A13 Parsers

A13.1 The ParserSet Object

parser_set module.

class `parser_set.`**`ParserSet`**

ParserSet class. The ParserSet object functions as a sequence/collection. The ParserSet class is part of the vivid object extension protocol.

Variables

- **parsers** – The parsers contained in the ParserSet object.
- **_is_ParserSet** – An identifier to use in place of `type` or `isinstance`.

`__getitem__`(*key*)

Retrive the parser located at the index given by `key` parameter via indexing (e.g. `ParserSet[key]`).

Parameters **key** (`int`) – The index to use for retrieval.

Raises **TypeError** `key` parameter must be an index.

`__init__`()

Construct a ParserSet object.

`__iter__`()

Provides an iterator for ParserSet (e.g. "`for parser in ParserSet:`").

`__len__`()

Determine the length of the ParserSet object via the `len` built-in function e.g.(`len(ParserSet)`).

A13.2 The PointParser Object

point_parser module.

class `point_parser.`**`PointParser`**

PointParser class. The PointParser class is used for parsing Point object related expressions.

Variables **_is_Parser** – An identifier to use in place of `type` or `isinstance`.

`__call__`(**args*)

Call PointParser object (e.g., `PointParser(expression)`).

`__init__`()

Construct a PointParser object.

`_eval`(*string*)

Try to evaluate given string (e.g., "`is_on(P(2.0,2.0),P(1.0,1.0),P(3.0,3.0))`").

Parameters **string** (`str`) – The expression to evaluate; the PointParser object unstringifies Point objects in `string` parameter and tries to call a function of the Point object (also given by `string` parameter) with unstringified Points as arguments.

Raises **ValueError** Function provided in `string` parameter is not a function in the Point class, some argument is not a Point after trying to unstringify or the `string` parameter is improperly formatted.

A13.3 The TruthValueParser Object

truth_value_parser module.

class `truth_value_parser.`**`TruthValueParser`**

TruthValueParser class. TruthValueParser provides parsing functionality for entirely mathematical/logical strings.

Variables **_is_Parser** – An identifier to use in place of `type` or `isinstance`.

`__call__`(**args*)

Call TruthValueParser object (e.g., `TruthValueParser(expression)`).

`__init__`()

Construct a TruthValueParser object.

`_eval`(*string*)

Try to evaluate given string in `string` parameter. (e.g.,"`(4 < 5 * cos(2 * PI) and 4*e^3 > 3 *(3 + 3))and!(2 < 3)`").

Parameters **string** (`str`) – The expression to evaluate.

A13.4 The LineSegmentParser Object

point_parser module.

class `line_segment_parser.`**`LineSegmentParser`**

LineSegmentParser class. The PointParser class is used for parsing LineSegment object related expressions (some of which can involve Point objects).

Variables **_is_Parser** – An identifier to use in place of `type` or `isinstance`.

`__call__`(**args*)

Call LineSegmentParser object (e.g., `LineSegmentParser(expression)`).

`__init__`()

Construct a LineSegmentParser object.

`_eval`(*string*)

Try to evaluate given string (e.g., "`is_on(P(2.0,2.0),P(1.0,1.0),P(3.0,3.0))`").

Parameters **string** (`str`) – The expression to evaluate; the LineSegmentParser object unstringifies LineSegment and Point objects in `string` parameter and tries to call a function of the LineSegment object (also given by `string` parameter) with unstringified objects as arguments.

Raises **ValueError** Function provided in `string` parameter is not a function in the LineSegment class, some argument is not a Point or LineSegment after trying to unstringify or the `string` parameter is improperly formatted.