pyVivid; a Concrete Framework for Mechanized Diagrammatic Reasoning

Nicholas Marton

Abstract

In 2007, the abstract semantic framework of Vivid was introduced by Arkoudas et al. [1] 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 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. For example, there are numerous diagrams used in the field of statistics such as bar charts, pie charts and lorenz curves [2]; in the field of physics, free-body, energy-level, vector fields and Minkowski [3] diagrams are prevalent; in the biological realm, phylogenetic trees [4], 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 [5]) 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 [6]. 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 and furthermore, there is no general solution in dealing with them.

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 appear to be sufficiently general to model the most common types of uncertainty [7]. The models used in this approach 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 concrete implementation of the inference mechanisms therein.

The Vivid framework, presented by Arkoudas et al. [1], 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). 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. While Vivid seems like a good candidate for a deductive approach to reasoning over incomplete information as well as a powerful tool for heterogeneous inference, no concrete implementation of the framework has been provided until now. 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 **??** demonstrates multiple usages of the pyVivid library in the context of proof verification. Finally, in section **??**, we discuss related and future work.

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 following section to understanding the most applicable notations, definitions and semantics of the framework 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 $\Sigma = (\mathrm{C;R;V})$ consisting of a set of constant symbols C; a set of relation symbols R; and a set of variables V.

- An **attribute interpretation** of 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 $R \in \mathrm{R}$ 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:
  $$\mathcal{V}^I_{(w;\rho)/\chi}[\mathbf{true}] = \mathbf{true} \text{ and } \mathcal{V}^I_{(w;\rho)/\chi}[\mathbf{false}] = \mathbf{false}.$$
  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).
- 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:

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

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:

$$\gamma \vdash D \leadsto F$$

and

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

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 [8]. 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:

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

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:

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

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 1 and 2 respectively.

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

Figure 1: The semantics of diagrammatic-to-sentential case reasoning.

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

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

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

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

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

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

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

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

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

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

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

Figure 2: The formal semantics of diagrammatic deductions.

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. Among them, the most important were ease of use, extensibility and stability. 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 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 the 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.

3.1.1 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 vivid object extension protocol explained in the following section. The Point class along with the LineSegment class are two such examples of this object extension protocol put into practice.

3.1.2 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, like 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.

3.1.3 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.

3.1.4 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, the Vocabulary object to which a constant assignment holds a reference is even propagated to other objects that make use of the constant assignment like instances of the NamedState and Context classes. As all conceivable use cases of pyVivid are for the verification of a proof (at least while Vivid remains a type-α DPL), 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.

3.1.5 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).

3.1.6 Named States

Instances of the NamedState class (henceforth referred to as NamedStates or NamedState objects) are the principal structure for modeling 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.

3.1.7 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). An attributeInterpretation compiles atomic formulae over system objects via profiles into atomic formulae over selected attribute values of (some 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 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.

3.1.8 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 **??**. 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.

3.1.9 Contexts

Context objects, instances of the Context class, are ubiquitous throughout the implementation of the semantics of diagrammatic deductions. Each Context objects 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.

3.1.10 Diagrammatic Deductions

The pyVivid library provides direct implementations for each 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 1 and 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 an AttributeStructure object (formatted in S-expression notation). With this functionality, the $[EI/\Delta]$ rule of figure 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 for the $[Observe]$ and $[Sentential - to - Sentential]$ (the last of the 4 types of case reasoning) rules as well as they are performed with respect to named states.

References

[1] Konstantine Arkoudas and Selmer Bringsjord. Vivid: An AI Framework for Heterogeneous Problem Solving. *Artificial Intelligence*, 173(15):1367–1405, 2009. The url http://kryten.mm.rpi.edu/vivid/vivid.pdf provides a preprint of the penultimate draft only. If for some reason it is not working, please contact either author directly by email.

[2] Joseph L. Gastwirth. A general definition of the lorenz curve. *Econometrica*, 39(6):1037–1039, 1971.

[3] L. Parker and G. M. Schmieg. A Useful Form of the Minkowski Diagram. *American Journal of Physics*, 38:1298–1302, November 1970.

[4] Walter M Fitch, Emanuel Margoliash, et al. Construction of phylogenetic trees. *Science*, 155(3760):279–284, 1967.

[5] Daniela Berardi, Diego Calvanese, and Giuseppe De Giacomo. Reasoning on uml class diagrams. *Artificial Intelligence*, 168(1):70–118, 2005.

[6] Michael Irwin Jordan. *Learning in graphical models*, volume 89. Springer Science & Business Media, 1998.

[7] Peter Walley. Measures of uncertainty in expert systems. *Artificial intelligence*, 83(1):1–58, 1996.

[8] Konstantine Arkoudas. Natural deduction language. 2004.