AN EXPLORATION IN FLUID LOGICS

By

Joshua Taylor

A Dissertation Submitted to the Graduate
Faculty of Rensselaer Polytechnic Institute
in Partial Fulfillment of the
Requirements for the Degree of

DOCTOR OF PHILOSOPHY

Major Subject: COMPUTER SCIENCE

Approved by the
Examining Committee:

Selmer Bringsjord
Thesis Adviser

Mark Goldberg, Member

Carlos Varela, Member

Nick Cassimatis, Member

Rensselaer Polytechnic Institute
Troy, New York

November 2014
(For Graduation December 2014)

Contents

List of Tables

List of Figures

Acknowledgments

This has been many years in the making, and I am grateful to my parents and my wife for their immeasurable patience and unceasing encouragement. This research would not have been possible without the help of my advisor and many colleagues at the RAIR Lab, as well as coworkers and teammates in the Decision Sciences and Analytics group at Assured Information Security. Additionally, the role of AFOSR in catalyzing and supporting basic research in the RAIR Lab relating to machine discovery and proof in the formal sciences, in ways relating to category theory, is acknowledged with gratitude.

Abstract

Human reasoning is heterogeneous. Whether reasoning formally or informally, human reasoners frequently and effortlessly switch between many problem representations, reapplying results and techniques from one domain in another. Research in artificial intelligence, on the other hand, often produces “mechanical savants” that can solve one problem incredibly well, but are completely inapplicable to any other. In this dissertation, we set out to investigate and implement formal methods and techniques that capture the heterogeneity of formal human reasoning.

We review the ways in which combinations of logical systems have been used in mathematics and artificial intelligence, and the types of results that have been realized though mappings between logics and proof systems. We identify areas of research that are especially promising in automated reasoning and the representation of logical systems: denotational proof languages and category theory. Denotational proof languages are a family of languages that integrate deduction and computation. Category theory is an abstract branch of mathematics that provides a high level of generality and unites, among other things, many different types of logical systems.

Fusing denotational proof languages and category theory, we develop categorical denotational proof languages. These are a variant of denotational proof languages that take proofs, realized as categorical arrows, rather than propositions, as a fundamental building block. We demonstrate that category theory is a suitable formalism for representing logical systems and the mappings between them, and that categorical denotational proof languages are an effective tool for specifying the relationships between logical systems and the transformations between them, and do so in a way that promotes proof reuse.

We designed and implemented a framework, programming language, and standard library for specifying and programming in categorical denotational proof languages. We encoded a number of logical systems, including several versions of the propositional calculus, and mappings between them, including a mapping based on a translation between axiomatic proofs into natural-deduction proofs, and a mapping based on the deduction theorem. We present examples illustrating how programs in this language achieve heterogeneous reasoning, and conclude with discussion of future work and applications.

Chapter 1

Introduction & Historical Review

1.1 Motivation

1.1.1 Human Reasoning is Heterogeneous

Human reasoning is undoubtedly heterogeneous. Such reasoning, whether formal or informal, incorporates multiple representations compatible and incompatible, complete and incomplete; and human reasoners are adept at combining the results (e.g., arguments, proofs, mental models, examples and counter-examples) into coherent structure. Some activities are more common in certain modes of reasoning than others. For instance, combining formal proofs and arguments may be more common in mathematical reasoning, while checking consistency with existing mental models may be more common in informal or non-deductive reasoning. The study of model theory explicitly looks to the connections between axiomatizations and their interpretations. Humans build a variety of types of mental models (and counterexamples) of propositions in informal contexts. Humans regularly construct and compare competing arguments (Pollock, 1992). Humans also reason using many types of representations (e.g., formulae, diagrams) and are able to bring these representations to bear despite surface-level incompatibilities (Barwise & Etchemendy, 1995).

1.1.2 Humans Employ Tools

Not only are human reasoners capable of reasoning with many types of representations, proof theories, argument modes, &c., but they also employ many types of tools to assist them in their reasoning.

Logic may be understood as the study of good reasoning. From Aristotle's syllogisms to the present day, philosophers have recognized, categorized, and codified valid and invalid patterns of reasoning. In the past hundred and fifty years, formal proof systems have proved both an interesting topic of study in their own right as well as an incredibly useful tool.

Algorithms, formal and informal, have played an important role in human reasoning, providing reusable techniques for answering questions. Closely related to the use of algorithms is the use of heuristics. While human intuition can help to make reasoning into something of art, good heuristics can always prove useful in reasoning tasks.

Human reasoners often use techniques and build tools that exploit correspondences; phonetic scripts and cartography are superb examples. Barwise & Etchemendy (1995) give several examples (one of which is discussed in Example 9 (p. 13)) of representations that depict some underlying reality through the use of diagrams, portions of which correspond to underlying features. A less formal type of reasoning that uses correspondences in representations is analogical reasoning (Sowa & Majumdar, 2003).

And, of course, another "tool" that humans use in both formal and informal reasoning is observation, both direct and indirect. While abstract reasoning can help to understand the consequences of certain premises, it is often observation that helps to determine what propositions to accept as premises.

1.1.3 Humans Automate and Combine Tools

Many of the tools used by human reasoners are informal in nature, but those that lend themselves to rigorous, mechanical analysis can often be automated, fully or in part. Automation is, properly speaking, a combination of formalisms, algorithms, and, sometimes, heuristics.

Automated theorem provers were one of the first applications of artificial intelligence in mathematics and logic, and the field of automated theorem proving has since blossomed into a rich and fruitful field. Indeed, the Logic Theorist (Newell & Simon, 1956; Newell et al., 1958), which was capable of proving many of the theorems derived in Whitehead & Russell's (1927) *Principia Mathematica*, was developed a year before the term "artificial intelligence" was coined.

Automated proof assistants, ranging from the simple proof checkers to complex interactive proof construction environments such as Isabelle (Paulson & Nipkow, 1994) and Coq (Bertot & Castéran, 2004) have also proved an important advance in automated reasoning.

Model finders such as Paradox (Claessen & Sorensson, 2003) can be used to find satisfying interpretations of sets of formulae that can serve as possible interpretations or as counterexamples, both of which can be an aid in developing theories that accurately represent the intents of human reasoners. (Arkoudas (2004) provides an illustrative example in which a model finder is used to find the "edge cases" that an axiom set in development does not properly address.) Model finders can also serve as consistency checking tools: if a model finder is sufficiently powerful but cannot find a satisfying model of a set of formulae, then the formulae must be inconsistent.

These are just a few instances of the ways in which human reasoners automate tools and techniques. In all of these cases, the different procedures that human reasoners employ produce different types of artifacts. Automated theorem provers produce proofs (often in proof systems that are not particularly amenable to human analysis); model finders produce (usually complete) models, consistency checkers may produce "yes" or "no" answers, or may be able to point out the specific location of an inconsistency in a set of formulae. Humans excel at taking these various kinds of results, putting them together, and drawing conclusions from them that no individual result could justify alone.

1.1.4 Our Motivation, In This Context

Yet, despite the growing use of increasingly powerful automated tools, this combining process seems to be uniquely human. It is a suitable, if formidable, challenge, then, to attempt to automate or reproduce mechanically at least certain aspects of this distinctly human process. Particularly, we should like to be able to enable machines to reason using multiple representations and reasoning techniques jointly and to subsequently combine the results. Though some of the most impressive work done by humans in combining representations and reasoning artifacts is seen in the informal domain, we will restrict our focus to formal, though not necessarily deductive, reasoning and representations.

1.2 Problem Statement

Problem Statement. Develop a formal system for: (i) specifying logical systems and the interactions and relationships between them and (ii) proof construction using multiple logical systems; formally analyze said system; implement in software.

To fulfill the problem statement, it will be necessary to have: (i) a rigorous and principled representation of logical systems; (ii) a rigorous and principled representation of the interactions and relationships between logical systems; and (iii) a user-interface or API for working with the resulting framework. Additionally, from a pragmatic standpoint, we should also like to be able to retrofit existing formalisms and automated tools for use in the new framework.

The next section will discuss some of the prior approaches toward representing logical systems and working with multiple related logical systems. Ultimately we shall decide upon using a combination of category theory and denotational proof languages to represent logical systems and the relationships between them, but the following discussion will influence this decision and inform the use of the chosen formalisms.

1.3 Related & Prior Work

There is a great deal of prior work that looks toward representing logical systems, and there are many cases through the history of mathematical logic of the relationships between them being considered. In this section we will discuss several *ad hoc* approaches. We do not use the term *ad hoc* here is any pejorative sense, but only to indicate that a particular approach was developed to address a particular case, and that the ability to generalize was not a priority.

1.3.1 *Ad Hoc* Approaches

Throughout the history of formal logic, mathematicians and logicians have studied the relationships between formal systems, including logical systems as well as other mathematical structures, directly using techniques such as embeddings, translations, and encodings. In the field of computer science, particularly within complexity theory, the use of reductions whereby a given problem of unknown complexity is reduced to another of known complexity (i.e., an instance of the given problem can be deterministically reformulated as an instance of the problem of known complexity) has long been used to gain understanding of the "hardness" of different computing tasks.

Example 1 (Intuitionistic Propositional Calculus). Kolmogorov (1925/1967) compares the classical propositional calculus developed by Hilbert with a weaker propositional calculus suitable for Brouwer's intuitionistic program. The latter Kolmogorov calls a "general logic of judgments," and the former a "special logic of judgments." The two logics differ in which axioms they take, particularly regarding negation. Kolmogorov's terminology refers to the applicability of the intuitionistic calculus to any type of proposition (especially in the domain of the infinitary) making it "general," whereas Hilbert's classical calculus is suitable to only certain types of propositions (those in the domain of the finitary), making it a "special" case. (The meanings of "finitary" and "infinitary" arise in intuitionistic mathematics, but are not essential to the present discussion.)

The intuitionistic system, which Kolmogorov calls $\mathfrak{B}$, for Brouwer, is given by a language of formulae built from propositional variables $A, B, C, \ldots$ and the logical connectives $\supset$ and $\sim$ in the usual way, and a proof calculus using *modus ponens* and subsitution of propositional variables. The system $\mathfrak{B}$ has

Axioms (1.1)–(1.4) for implication.

$$A \supset (B \supset A) \tag{1.1}$$

$$(A \supset (A \supset B)) \supset (A \supset B) \tag{1.2}$$

$$(A \supset (B \supset C)) \supset (B \supset (A \supset C)) \tag{1.3}$$

$$(B \supset C) \supset ((A \supset B) \supset (A \supset C)) \tag{1.4}$$

$\mathfrak{B}$ also includes Axiom (1.5) for the principle of contradiction.

$$(A \supset B) \supset ((A \supset \sim B) \supset \sim A) \tag{1.5}$$

The classical system, which Kolmogorov dubs $\mathfrak{H}$, for Hilbert, is simply $\mathfrak{B}$ with the addition of Axiom (1.6).

$$\sim\sim A \supset A \tag{1.6}$$

Kolmogorov proves that $\mathfrak{H}$ is equivalent to Hilbert's original system which had (1.1)–(1.4), but neither (1.5) nor (1.6), but rather (1.7) and (1.8) for negation.

$$A \supset (\sim A \supset B) \tag{1.7}$$

$$(A \supset B) \supset ((\sim A \supset B) \supset B) \tag{1.8}$$

Kolmogorov treated the logics $\mathfrak{H}$ and $\mathfrak{B}$ as calculi with different domains of applicability. With this understanding, $\mathfrak{H}$ is a calculus for whose sentences the double negation principle, Axiom (1.6), is applicable. That is, each sentence in $\mathfrak{H}$ represents a judgment which follows from its double negation. The principle of double negation does not hold in general for the sentences of $\mathfrak{B}$; i.e., there may be sentences ϕ in $\mathfrak{B}$ for which $\sim\sim\phi \supset \phi$ does not hold. Kolmogorov's insight lay in asking whether there are sentences in $\mathfrak{B}$ for which the principle of double negation must hold. Brouwer (1925) had shown that every negation has this property, by means of a proof in $\mathfrak{B}$ with the conclusion $\sim\sim\sim A \supset \sim A$. Thus if a sentence ϕ is of the form $\sim\psi$, then $\sim\sim\phi \supset \phi$. Kolmogorov further shows that if ϕ and ψ are sentences each of which follows from its double negation, then $\phi \supset \psi$ also follows from its double negation. Kolmogorov concludes that:

> The precise boundary of the domain in which the special logic of judgments is applicable has been found; this domain coincides with the domain in which the formula of double negation is applicable. (Kolmogorov, 1925/1967, p. 427)

Though this translation, Kolmogorov showed that despite the foundational differences between classical and intuitionistic logic, and that the intuitionistic school of thought explicitly rejected some classical principles, to every classical result there is an intuitionistic counterpart.

We will later develop more formal tools in which this type of result can be expressed (e.g., see Example 30 (p. 35)), but for now we can point out a simple corollary from Kolmogorov's result: any proof in $\mathfrak{H}$ can be translated into a $\mathfrak{B}$ proof by translating each formula ϕ to a formula ϕ' by: (i) replacing each occurrence of a propositional variable A with its double negation $\sim\sim A$; and (ii) replacing each use of the double negation axiom in the proof with $\sim\sim\phi \supset \phi$ with the corresponding derivation of derivation of $\sim\sim\phi \supset \phi$ in $\mathfrak{B}$. (Other steps of the proof are simply substitutions, instances of axioms common to $\mathfrak{H}$ and $\mathfrak{B}$, or applications of *modus ponens*.)

Example 2 (First-Order Semantics for Modal Logics). Saul Kripke (1963) unified the study of a number of modal logics when he presented frame semantics (now often called "Kripke semantics") for understanding these theories. The modal propositional logics with which he was concerned were those whose formulae included the propositional calculus (i.e., propositional variables, and some complete set of connectives, e.g., $\sim$ and $\supset$) as well as the modal operator $\Box$, where $\Box\phi$ is a sentence whose meaning depends on the interpretation of $\Box$. For instance, in alethic modal logic, the logic of necessity, $\Box\phi$ is read as "ϕ is necessary," and in deontic logic, the logic of obligation, it is read as "ϕ is obligatory."

Kripke's proposed semantics specified that a model structure consisted in a triple $\langle \mathbf{G}, \mathbf{K}, \mathbf{R} \rangle$ where $\mathbf{K}$ is a set of possible worlds, $\mathbf{G}$ is a special element of $\mathbf{K}$ (the "real world"), and $\mathbf{R}$ is a reflexive relation on $\mathbf{K}$ (accessibility between worlds, or relative possibility). (The reflexivity of $\mathbf{R}$ is not strictly necessary, and there are advantages to dropping it, as later authors did, but Kripke included this requirement.) A model coupled a model structure with a mapping of sentences and the elements of $\mathbf{K}$ to truth values (so as to speak of P being true in k_1, Q being false in k_2, and so on), such that the boolean connectives are interpreted in the obvious way (e.g., $P \,\&\, Q$ is true in k if and only if P and Q are true in k), and such that $\Box\phi$ is true in k if and only if ϕ is true in every k' such that $k\mathbf{R}k'$, i.e., if ϕ is true in every world k' accessible to k.

Though Kripke presented his semantic theory as just that, it is not difficult to understand a correspondence with a first-order theory whose domain is $\mathbf{K}$, and has a constant symbol $\mathbf{G}$, and a binary relation $\mathbf{R}$. Modal formulae can be translated into first order formulae in the theory by the translation τ, defined as follows:

$$\tau(\phi) = t(\phi, \mathbf{G})$$

$$t(\sim\phi, g) = \sim t(\phi, g)$$

$$t(\phi \supset \psi, g) = t(\phi, g) \supset t(\psi, g)$$

$$t(\Box\phi, g) = \forall h[g\mathbf{R}h \supset t(\phi, h)]$$

Though there are efficient tableaux reasoning algorithms for these modal logics, this translation allows the use of off-the-shelf general purpose first-order reasoners to be applied to problems in modal propositional logics. (Indeed, this is the approach used in at least one software system, viz., Slate (Bringsjord et al., 2007).)

By encoding the semantics of this family of non-classical logics using first-order logic, many results of first-order logic can immediately be carried over to modal logic. Additionally, automated reasoners for first-order logic (both syntactic and semantic, e.g., both proof finders and model builders) can be applied to modal logic. Conversely, specialized reasoners for modal logics can be applied to corresponding fragments of first-order logic.

Example 3 (Fragments of First-Order Logic, Description Logics). Blossoming in the 1980's and growing up through the present day (particularly in the Semantic Web), Description Logics (for a comprehensive reference, see Baader et al., 2003) have provided a concise syntax for certain types of knowledge representation tasks.

In an impressive survey, Hustadt et al. (2004) examine the relationship of various description logics with decidable fragments of first-order logic. Description logic formulae are partitioned into terminological axioms and assertional sentences. The former define and relate concepts (classes of entities) and roles (relationships among entities) while the latter assert individual membership in concepts and roles. While standard first-order logics tend to provide a minimal syntax for constructing formulae, leaving little that can be removed without severely restricting expressiveness and little that can be added that could not be expressed in terms of existing constructs, description logics are built on a richer, more granular, syntax. Description logics are typically given a set-based semantics by specifying a domain $\mathcal{D}$ and an interpretation function $\mathcal{I}$ that maps individual names to elements of $\mathcal{D}$, concept names to subsets of $\mathcal{D}$, and role names to subsets of $\mathcal{D} \times \mathcal{D}$.

Hustadt et al. (2004) define a particular description logic, which they call $\mathcal{DL}$, whose sublogics are subsequently examined. The syntax for $\mathcal{DL}$ is reproduced in Table 1.1. The semantics of $\mathcal{DL}$ is reproduced in part in Figure 1.1.

Table 1.1: Constructors of $\mathcal{DL}$. Term constructors are not necessarily independent. E.g., $\bot$ is redundant provided $\sim$ and $\top$. The variables C and D range over concept terms, R and S over role terms, and a and b over individual names.

Concept Terms		**Role Terms**	
$\top$	top concept	$\triangle$	top role
$\bot$	bottom concept	$\bigtriangledown$	bottom role
$C \sqcap D$	concept intersection	$\mathrm{id}(C)$	identity role on C
$C \sqcup D$	concept union	$R \sqcap S$	role intersection
$\sim C$	concept complement	$R \sqcup S$	role union
$\forall R.C$	universal restriction	$R \circ S$	role composition
$\exists R.\top$	limited existential restriction	$\sim R$	role complement
$\exists R.C$	existential restriction	$R^{\smile}$	role inverse
$\exists_{\geq n}R$, $\exists_{\leq n}R$	number restrictions	R^{+}	transitive role closure
$\exists_{\geq n}R.C$, $\exists_{\leq n}R.C$	qualified number restrictions	$R \upharpoonleft C$	domain restriction
$(R \subseteq S)$	inclusion role value maps	$R \downharpoonleft C$	range restriction
$R = S$	equality role value maps		

Terminological Sentences		**Assertional Sentences**	
$C \sqsubseteq D$ $(R \sqsubseteq S)$	concept (role) subsumption	$a{:}C$	concept membership
$C = D$ $(R = S)$	concept (role) equivalence	$(a, b){:}R$	role membership

$$
\begin{aligned}
\mathcal{I}(\top) &= \mathcal{D}\\
\mathcal{I}(\bot) &= \emptyset\\
\mathcal{I}(C \sqcap D) &= \mathcal{I}(C) \cap \mathcal{I}(D)\\
\mathcal{I}(C \sqcup D) &= \mathcal{I}(C) \cup \mathcal{I}(D)\\
\mathcal{I}(\sim C) &= \mathcal{D} \setminus \mathcal{I}(C)\\
\mathcal{I}(\forall R.C) &= \{\, x \in \mathcal{D} \mid \forall y((x, y) \in \mathcal{I}(R) \supset t \in \mathcal{I}(C))\,\}\\
\mathcal{I}(\exists R.C) &= \{\, x \in \mathcal{D} \mid \exists y((x, y) \in \mathcal{I}(R) \,\&\, y \in \mathcal{I}(C))\,\}\\
\mathcal{I}(\exists_{\geq n}R) &= \{\, x \in \mathcal{D} \mid n \leq |\{\, y \in \mathcal{D} \mid (x, y) \in \mathcal{I}(R)\,\}|\,\}\\
\mathcal{I}(R \subseteq S) &= \{\, x \in \mathcal{D} \mid \forall y((x, y) \in \mathcal{I}(R) \supset (x, y) \in \mathcal{I}(S))\,\}\\
\mathcal{I}(\triangle) &= \mathcal{D} \times \mathcal{D}\\
\mathcal{I}(\bigtriangledown) &= \emptyset\\
\mathcal{I}(R \sqcap S) &= \mathcal{I}(R) \cap \mathcal{I}(S)
\end{aligned}
$$

$$
\begin{array}{lll}
(\mathcal{D}, \mathcal{I}) \models a{:}C & \text{iff} & \mathcal{I}(a) \in \mathcal{I}(C)\\
(\mathcal{D}, \mathcal{I}) \models C \sqsubseteq D & \text{iff} & \mathcal{I}(C) \subseteq \mathcal{I}(D)\\
(\mathcal{D}, \mathcal{I}) \models C = D & \text{iff} & \mathcal{I}(C) = \mathcal{I}(D)\\
(\mathcal{D}, \mathcal{I}) \models (a, b){:}R & \text{iff} & (\mathcal{I}(a), \mathcal{I}(b)) \in \mathcal{I}(R)
\end{array}
$$

Figure 1.1: Set-theoretic semantics, in part, of $\mathcal{DL}$. $\mathcal{D}$ is a domain of individuals, and $\mathcal{I}$ which maps individual, role, and concept names to elements of $\mathcal{D}$, and complex role and concept expressions as defined here.

Table 1.2: Some decidable sublogics of $\mathcal{DL}$ can be obtained by restricting the available concept and role constructors.

$\mathcal{DL}$ **Sublogic**	**Concept Constructors**	**Role Constructors**
$\mathcal{ALC}$	$\sim, \sqcap, \exists\ (\top, \bot, \sqcup, \forall)$	
BML	$\sim, \sqcap, \exists\ (\top, \bot, \sqcup, \forall)$	$\sim, \sqcap\ (\triangledown, \triangle, \sqcup)$
$\mathcal{ALB}$	$\sim, \sqcap, \exists\ (\top, \bot, \sqcup, \forall)$	$\sim, \sqcap, ^{\smile}, \upharpoonleft\ (\triangledown, \triangle, \sqcup, \downharpoonright)$
Pierce Logic	$\sim, \sqcap, \exists\ (\top, \bot, \sqcup, \forall)$	$\sim, \sqcap, ^{\smile}, \circ, \mathrm{id}\ (\triangledown, \triangle, \sqcup, \upharpoonleft, \downharpoonright)$
PDL	$\sim, \sqcap, \exists\ (\top, \bot, \sqcup, \forall)$	$\sqcup, \circ, ^{+}, \mathrm{id}(C)\ (\triangledown)$

Table 1.3: Partial translation of $\mathcal{DL}$ sentences into first-order formula. $\Pi(\phi)$ translates the $\mathcal{DL}$ sentence ϕ into a first-order formula. $\pi'(R,X,Y)$ produces a formula that indicates that the terms denoted by X and Y stand in relation T. $\pi(C,X)$ produces a formula that indicates that the term denoted by X is a member of the concept C.

Concept and Role Translations	
$\pi(A,X) = Q_A(X)$	$\pi(C \sqcap D, X) = \pi(C,X) \,\&\, \pi(D,X)$
$\pi(\sim C, X) = \sim \pi(C,X)$	$\pi(C \sqcup D, X) = \pi(C,X) \vee \pi(D,X)$
$\pi(\top, X) = \top$	$\pi(\forall R.C, X) = \forall y(\pi'(R,X,y) \supset \pi(C,X))$
$\pi(\bot, X) = \bot$	$\pi(\exists R.C, X) = \exists y(\pi'(R,X,y) \,\&\, \pi(C,y))$
$\pi'(P,X,Y) = Q_P(X,Y)$	$\pi'(R \sqcap S, X, Y) = \pi'(R,X,Y) \,\&\, \pi'(S,X,Y)$

Sentence Translations	
$\Pi(C \sqsubseteq D) = \forall x(\pi(C,x) \supset \pi(D,x))$	$\Pi(R \sqsubseteq S) = \forall xy(\pi'(R,x,y) \supset \pi(S,x,y))$
$\Pi(C = D) = \forall x(\pi(C,x) \leftrightarrow \pi(D,x))$	$\Pi(R = S) = \forall xy(\pi'(R,x,y) \leftrightarrow \pi'(S,x,y))$
$\Pi(a{:}C) = \pi(C,a)$	$\pi((a,b){:}R) = \pi'(R,a,b)$

The richer syntax of description logics facilitates exploration of sub-languages and extensions through the addition and removal of concept and term constructors. For instance, Hustadt et al. (2004) point out that the logics shown in Table 1.2 are obtained from $\mathcal{DL}$ by removing certain concept or role constructors.

Based on Figure 1.1, it is obvious that the semantics of description logics can be expressed using first-order logic. In particular, the terminological and assertional sentences of a given description logic can be recast as first-order formulae. Given a first-order predicate symbol Q_C for each concept symbol C and a first-order binary relation symbol Q_R for each role symbol R, the translation given in Table 1.3 maps $\mathcal{DL}$ sentences to first-order formulae.

Hustadt et al. (2004) make a number of connections between sublogics of $\mathcal{DL}$ and decidable fragments of first-order logic. We summarize two to give a feel for the types of results that can be obtained using these translation-based methods. These are both based on the translation Π defined in Table 1.3 applied to sublogics of $\mathcal{DL}$ defined in Table 1.2.

- The translation Π maps $\mathcal{ALC}$ into the guarded fragment of first-order logic, GF. GF is decidable, and its complexity is known. This provides an upper bound on reasoning complexity for $\mathcal{ALC}$, and also allows any reasoner for GF to be applied to (the translation of) $\mathcal{ALC}$ formulae.

- Π maps $\mathcal{ALB}$ sentences into the two-variable fragment of first-order logic, FO^2, whose complexity is known. Again, this provides upper bounds for the complexity for $\mathcal{ALB}$ reasoning as well as usable automated reasoners.

As with the previous example, a careful examination of the relationships between a family of non-traditional logics and first-order logic gives benefits in automated reasoning to both domains: first-order theorem provers and model finders can be applied to (translations of) description logics, and specialized reasoning techniques developed for description logics can be applied to fragments of first-order logic. Additionally, completeness and efficiency results derived in the study of description logics can be carried over to corresponding first-order fragments.

Example 4 (Relating Ontologies with Bridging Axioms). A very important *ad hoc* technique from a practical standpoint that has been used in relating and combining multiple knowledge representation and reasoning systems is that of bridging axioms. A bridging axiom is simply an axiom that relates terms each of which originate in different ontologies (Dou et al., 2005). In real-world applications, the development of an interlingua, i.e., a logical language into which all the relevant logical systems can be translated, is infeasible, and a much more realistic approach is to relate terms from different ontologies incrementally. The creation of an appropriate set of bridging axioms for a given circumstance is something of an art, though there has been work toward automating the process (Dou & McDermott, 2006).

My own earlier work (Taylor, 2007) examined principled ways of creating bridging axioms and organizing them in such as a way as to include only those necessary for a particular reasoning task. This resulted in the construction of translation graphs, such as the simple one shown in Figure 1.2.

While earlier examples highlighted the benefit of different types of logics and exploiting the similarities between them, the present example points to the importance of and the need to work with multiple logical systems of the same species. While it is common to speak of "first-order logic" as though it were one system, in reality every practical application of first-order logic requires a custom vocabulary and

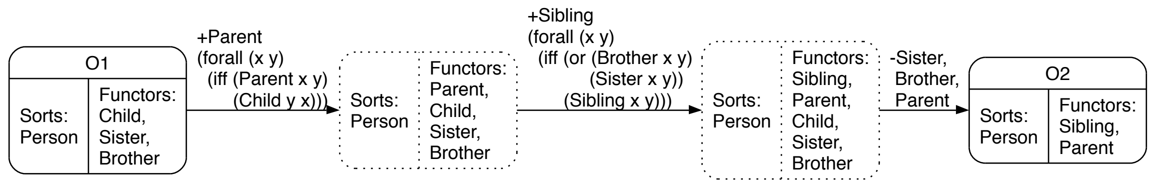


Figure 1.2: A translation graph relating two genealogical ontologies. Two intermediate ontologies are constructed, each of which has a vocabulary distinct from $\mathcal{O}_1$ and $\mathcal{O}_2$. In general, reasoning with information from multiple ontologies requires the axioms along the paths connecting the ontologies.

set of axioms in addition to the standard logical connectives, axioms, and inference rules. The use of bridging axioms and translation graphs are techniques that can help in situations where two or more existing logical systems must be brought together.

Example 5 (KIF (Knowledge Interchange Format)). The Knowledge Interchange Format, KIF (Genesereth & Fikes, 1997), was developed to provide a standard syntax for expressing ontologies and instance data. A common format reduces the interoperability challenge to that of discovering or engineering appropriate bridging axioms and efficient reasoning.

While the concept of a standard representation for information is not in and of itself a particularly interesting accomplishment, KIF is notable for several reasons:

- KIF provides numerous constructs particularly suited to the challenges faced by knowledge engineers, e.g., special defining operators such as defining-axiom, defobject, deffunction, and defrelation.
- KIF provides built-in libraries for common representation tasks and data structures including numbers, lists, and sets.
- KIF was developed by an Interlingua Working Group as part of the DARPA Knowledge Sharing Effort. This indicates the recognition by government agencies of the importance of knowledge sharing and interchange.

The KIF effort addressed some of the same goals that bridging axioms and translation graphs did, viz., achieving interoperability. However, where bridging axioms and translation graphs addressed the connections between logical systems at a very high level, KIF provided a concrete syntax for the exchanging formulae and bridging axioms. Where the high-level techniques could assist in determining a plan for how multiple systems might be joined up, KIF answered many of the practical engineering challenges inherent to implementation of such a joined up system.

Example 6 (RDF, OWL, &c.). Where the existence of KIF demonstrated the need for knowledge sharing and interoperability within the government and defense communities, Semantic Web technologies including RDF and OWL show that industry and academia also recognize the same needs. The Semantic Web languages, particular the Web Ontology Language (OWL) family (Bechofter et al., 2004) draw heavily upon Description Logics.

The development of the Semantic Web and Semantic Web Languages combine some of the most important facets of description logics and knowledge interchange efforts. The OWL family of logics fall mostly with the expressiveness of description logics, and so reasoners with acceptable complexities can be applied (whereas KIF could provide no such guarantees). RDF and OWL also have syntax based in XML (as well as representations in other forms) which alleviate many problems relating to namespaces and common vocabulary terms often occurred in simpler representations.

Example 7 (IKRIS). In 2004, the Disruptive Technology Office (DTO) (which subsequently evolved into the Advanced Research and Development Activity (ARDA), and finally the Intelligence Advanced Research Projects Activity (IARPA)) sponsored a challenge workshop, *IKRIS: Interoperable Knowledge Representation for Intelligence Support* (Thurman et al., 2006), with the purposes of enabling interoperability between knowledge representation and reasoning systems developed for and used by the intelligence community and developing a knowledge representation suitable for the tasks performed by those within the intelligence community. The IKRIS workshop produced two knowledge representation languages, both extensions of Common Logic (ISOCommonLogic, 2007): the IKRIS Knowledge Language (IKL) (Hayes, 2006; Hayes & Menzel, 2006); and the IKRIS Context Logic (ICL) (Cheikes, 2006).

The IKRIS workshop was prompted by many of the same types of needs that brought about the KIF effort. The recognition that many of the knowledge representations and databases used within the intelligence and defense communities had no capabilities for interoperability, coupled with a heightened emphasis on information sharing between agencies, motivated the development of a formalism for relating these systems. The variety of types of information at hand required logics much more expressive than description logics and, at least superficially, first-order logic.

1.3.2 Heterogeneous Logic

Barwise & Etchemendy (1996) present heterogeneous logics as framework for unifying different approaches to reasoning with multiple logical systems, particularly for reasoning with both linguistic (i.e., sentence-based) and diagrammatic logics. They place particular emphasis on systems with representations that are homomorphic, i.e., representation schemes which themselves have some of the same structures as the information that they represent. For instance, the placement of numbers on a number line provides a homomorphic representation of the less-than and greater-than relations, whereas a sentential representation using the relation symbols $<$ and $>$ does not, as shown in Figure 1.3. Barwise & Etchemendy provide an informal ontology for homomorphic representations, and for combinations of logical systems that do not employ an interlingua or "universal scheme of representation." Rather, the logical systems which they combine are united by a common underlying semantic structure which each representation describes incompletely. (Note that this is in distinct contrast with some earlier examples (e.g., KIF) wherein languages were developed expressly for the purpose of developing interlinguae.)

$$\begin{aligned} x &< y \\ y &< z \\ x &> w \end{aligned}$$

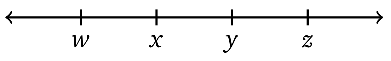


Figure 1.3: Non-homomorphic and homomorphic representations for the ordering relations on numbers. Axioms would allow the inference of $w < z$ from the formulae on the left, but the same relation can be observed directly from the number line.

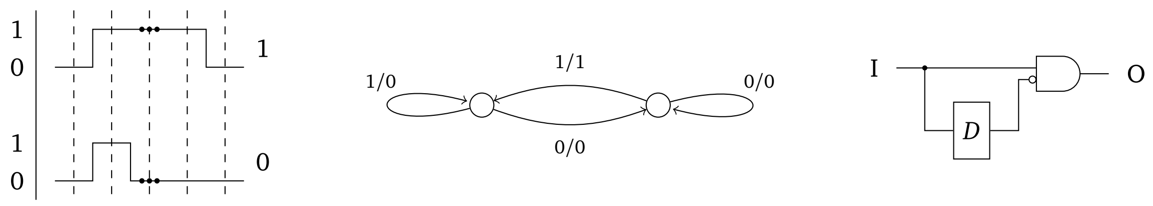


Figure 1.4: Three types of diagrams, viz.: timing diagrams, state diagrams, and circuit diagrams, are all used to describe a unit pulser. Each diagram depicts aspects of the underlying pulser, but no representation represents all aspects of the pulser.

Example 8 (Unit Pulsers). Barwise & Etchemendy (1996) cite Johnson et al.'s (1996) use of various kinds of diagrams in electronic hardware design as a natural use of heterogeneous reasoning incorporating linguistic (i.e., natural language) and diagrammatic representations. In the design of a unit pulser, for instance, three types of diagrams, shown in Figure 1.4, are employed, viz.: state diagrams, timing diagrams, and circuit diagrams. Each type of diagram represents some aspect of the hardware under development, but no type of diagram captures the entire design:

> There are hundreds of different relationships that figure into the design of a new chip or other electronic device. These relationships typically cluster into families, depending on what perspective one takes for the moment. ... Engineers have solved this representation problem with three separate representational systems: state charts for the representation of control, circuit diagrams for the representation of gate information, and timing diagrams for the representation of timing. (Barwise & Etchemendy, 1996, p. 183)

Example 9 (Hyperproof). Barwise & Etchemendy's (1994) Hyperproof is a superb example of the use of heterogeneous logic to combine sentential and diagrammatic logics for reasoning about a blocks-world environment. Hyperproof is a software and textbook package that provides a proof environment which combines a Fitch-style natural-deduction proof system with a simple diagrammatic logic. (We describe Fitch-style systems later, in Example 14 (p. 21).) Both representations are incomplete representations of a blocks-world.

The language of the Fitch-style natural-deduction proof system is a typical first-order language for a blocks world. The typical boolean connectives and first-order quantifiers are available and sentences are built in the standard way. The predicates and relation symbols used for describing the blocks world are: (unary) Small, Medium, Large, Tet, Cube, and Dodec; (binary) =, Larger, LeftOf, FrontOf, Adjoins, SameSize, SameShape, SameRow, and SameColumn; and (ternary) Between. The inference rules in the sentential proof system are typical of a Fitch-style natural deduction proof system, including introduction and elimination rules for each connective.

The diagrammatic representation consists of a grid on which the tokens of sixteen different types are placed. Each token has a particular size (small, medium, large, or unknown) and a particular shape (cube, tetrahedron, dodecahedron, or unknown). (The unknown shape is represented by sack, and the unknown size by a cylinder with a question mark.) The shape, size, and placement of a token on the grid represents the shape, size, and placement of a corresponding block in a blocks world.

Note at this point that neither the sentential nor the diagrammatic representations can represent all the relevant aspects of the underlying blocks world. For instance, there is no way to represent the absolute position of a block with the sentential representation. Yet there is no way of representing, for instance, the fact that a given block is to the left of another with the diagrammatic representation without specifying absolute positions of the two blocks.

After defining a blocks-world semantics, definitions of satisfiability and consequence relations for sentences and diagrams are straightforward. (It should be noted, however, that care must be taken with the diagrams to distinguish between what a diagram does not show to hold and what a diagram shows not to hold. The underlying logic has three values: true, false, and indeterminate.) This requires nothing out of the ordinary for the sentential representation, but there are some points of interest for the diagrammatic representation, including that: every diagram is consistent; and if each of two diagrams d and d' is a logical consequence of the other, then $d = d'$. We mention these points only to highlight the fact that Hyperproof's diagrammatic representation has characteristics that distinguish it not only from Hyperproof's sentential representation, but from typical sentential representations in general.

The most interesting aspect of Hyperproof from the perspective of the present work, however, is the relationship and interaction between the sentential and the diagrammatic proof systems. This interaction is achieved through the use of two inference rules specific to the combined proof environment, **Observe** and **Cases Exhaustive**.

A user with a diagram uses the **Observe** rule to derive sentences which, based on the diagram, must be true. For instance, if a diagram shows that a is a tetrahedron, then the user is justified in using the rule **Observe** to derive the sentence Tet(a). Thus **Observe** allows a user with a diagram to obtain new sentences.

The **Cases Exhaustive** rule allows a user in possession of a diagram d and a sentence to construct a number of diagrams $d_1, \ldots, d_n$ each more specific than d and to mark that set of diagrams as an exhaustive set of extensions of d with respect to the sentence. The diagrammatic rule **Merge** can subsequently be used to derive a new diagram which is like d but extended with any information which is common to all of $d_1, \ldots, d_n$. This combination of **Cases Exhaustive** with **Merge** allows a user with a diagram and a sentence to obtain new diagrams.

Hyperproof is an important example of the combination of two logical systems with different characteristics. We note that the two logical systems can exist on their own and be used independently for reasoning about blocks-world environments, but that their combination, when augmented with appropriate inference rules for relating the different representations, can achieve results that neither system alone could. Importantly, Hyperproof uses no interlingua or common representation scheme, but bases the combination of logical systems only on the presence of a common underlying semantics.

Chapter 2

Theory

In this chapter, we will review the historical treatments of logical systems that are most important to our present efforts. Particularly, we will briefly cover and give examples of the two major types of proof calculi (viz., axiomatic and natural deduction), we will examine a somewhat more recent category-theoretic approach to proof systems (deductive systems and categories), and we will review the formal calculus underlying the denotational proof languages (the $\lambda\mu$-calculus).

2.1 Traditional Treatments

A logic, or logical system, is defined by specifying its language of formulae and its rules of proof construction. In fact, since some proof construction rules might appeal to entities other than formulae, an even more general definition would have a logic defined by a set of proof construction rules, but in practice the proof construction rules require the notion of formula, so we maintain the distinction between languages and proof systems.

Logical languages are classes whose members are propositions expressing formulae. The languages of propositional logics are typically specified by defining a set of atomic formulae, and a set of logical connectives by which formulae can be combined into compound formulae. The languages of first-order logics are usually more complicated, defining first a set of terms, including variables, constants, and compound terms, and then a set of atomic formulae, and again a set of logical connectives and quantifiers by which compound formulae are formed. Modal logics, both propositional and first-order, introduce intensional operators by which other types of sentences can be constructed. For the most part, we will not be concerned in any depth with the particular languages of the logics we will consider, these being well-studied and not directly relevant to the issues we will treat. It is worth noting, however, that the formulae of many logics are decidable sets; i.e., the class of formulae is in fact a set, and it is decidable whether an expression is in fact a formula. This shall be the case for all logics we consider herein.

Proof construction rules provide a way of combining formulae and, sometimes, other relevant entities, into proofs expressing chains of reasoning. There are many kinds of proof systems, and though our primary concern is not with any one type in particular, there are several dominant traditions by which most proof systems are influenced. Two such traditions are the axiomatic and natural-deduction approaches.

2.1.1 Axiomatic Proof Systems

An axiomatic, or Hilbert-style,[1] proof system begins with the specification of a logical language. The specification determines the class of well-formed formulae of the language, and is typically given by some set of rules or a grammar, though the only requirement is that the set of well-formed formulae is decidable. A subset of the well-formed formulae are then taken as axioms. Axioms are typically specified schematically, e.g., an axiomatic proof system for the propositional calculus might have the axiom schema $\phi \supset (\psi \supset \phi)$, all of whose instances (i.e., formulae obtained by uniformly substituting formulae of the language for ϕ and ψ) are axioms of the proof system. Finally, an axiomatic proof specifies a (usually small) number of inference rules by which formulae may be derived from other formulae. A proof is simply a sequence of formulae in which each formulae is either an axiom or a formulae justified by an inference rule on the basis of prior formulae in the proof. Many axiomatic proof systems for the propositional calculus adopt the sole rule *modus ponens* by which ψ is derived from instances of $\phi \supset \psi$ and ϕ.

Example 10 (Axiomatic Propositional Calculus). An axiomatic proof system for the propositional calculus discovered by Łukasiewicz (1948) uses three axiom schemata and one inference rule. The language of well-formed formulae is given as follows. A set of propositional variables $\{p, q, r, \ldots\}$ is fixed, each of whose members is a formula. Additionally, if ϕ and ψ are formulae, then so too are $\sim\phi$ and the material conditional $\phi \supset \psi$. It is common to define other connectives in terms of $\sim$ and $\supset$. For instance, a conjunction $\phi \,\&\, \psi$ may be defined as $\sim(\phi \supset \sim\psi)$. The disjunction $\phi \vee \psi$ might be defined as $\sim\phi \supset \psi$.

The axioms of the propositional calculus are the instances of three schemata:

$$\phi \supset (\psi \supset \phi) \tag{PC1}$$

$$(\phi \supset (\psi \supset \rho)) \supset ((\phi \supset \psi) \supset (\phi \supset \rho)) \tag{PC2}$$

$$(\sim\psi \supset \sim\phi) \supset (\phi \supset \psi) \tag{PC3}$$

A proof is simply a sequence of formulae each of which is either an axiom (i.e., an instance of one of

[1]The term "Hilbert-style proof calculus" or "Hilbert system" is common, but systems of this type were developed and used by many mathematicians prior to (and after) the development of natural-deduction proof calculi.

the axiom schemata) or the result of an application of *modus ponens* to two earlier formulae in the proof. In this axiomatic system, we may construct a proof of the formula $p \supset p$:

1.	$p \supset ((p \supset p) \supset p)$	(PC1)
2.	$(p \supset ((p \supset p) \supset p)) \supset ((p \supset (p \supset p)) \supset (p \supset p))$	(PC2)
3.	$((p \supset (p \supset p)) \supset (p \supset p))$	*modus ponens* 1, 2
4.	$p \supset (p \supset p)$	(PC1)
5.	$p \supset p$	*modus ponens* 4, 3

An important meta-logical result in some axiomatic systems is the deduction theorem. The deduction theorem states that if a formulae ψ can be proved from the axioms and inference rules of the logical system as well as an additional premise ϕ, then the conditional $\phi \supset \psi$ can be proved in the logical system alone. For instance, taking $\sim\sim\sim p \supset \sim p$ as a premise, we can prove $p \supset \sim\sim p$:

1.	$\sim\sim\sim p \supset \sim p$	premise
2.	$(\sim\sim\sim p \supset \sim p) \supset (p \supset \sim\sim p)$	(PC3)
3.	$p \supset \sim\sim p$	*modus ponens* 1, 2

Then, according to the deduction theorem, there is a proof in the axiomatic system of the conditional

$$(\sim\sim\sim p \supset \sim p) \supset (p \supset \sim\sim p)$$

that does not make use of any premises.

More generally, when a formula ϕ is provable in the logical system with a set of premises Γ, we write $\Gamma \vdash \phi$. When ϕ can be proved without the use of any premises, we write $\emptyset \vdash \phi$, or just $\vdash \phi$, and ϕ is said to be a theorem. Restated with this notation, the deduction theorem states that if $\Gamma \cup \{\phi\} \vdash \psi$, then $\Gamma \vdash \phi \supset \psi$.

Example 11 (Axiomatic **T**). Alethic modal logics deal with sentences of the form "it is necessary that ..." and "it is possible that ... ." We now consider an example of a simple axiomatic proof system for the alethic modal logic, **T**. The set of formulae of **T** is a superset of the formulae of the propositional calculus. Again, we fix a set of propositional variables, $\{p, q, r, \dots\}$, each of which is a formula. Furthermore, if ϕ and ψ are formulae, then so are $\sim\phi$, $\phi \supset \psi$, and $\Box\phi$. The latter is read as "it is necessary that ϕ," or "necessarily ϕ." **T** includes an axiomatization of the propositional calculus (e.g., the one discussed in the previous example, but any suitable axiomatization will do), and two additional axiom schemata:

$$\Box(\phi \supset \psi) \supset (\Box\phi \supset \Box\psi) \tag{dist}$$

$$\Box\phi \supset \phi \tag{M}$$

Like the propositional calculus, **T** includes *modus ponens* as an inference rule, and also adds the necessitation rule,

$$\frac{\vdash \phi}{\Box\phi}$$

which states that if ϕ is a theorem, then so too is $\Box\phi$.

Example 12 (Axiomatic First-Order Logic). The propositional calculus dealt with propositional variables and **T** extended the logical language to include assertions concerning necessity. The language of first-order logic is more complex, but permits expressions concerning individuals and quantification thereover.

Central to the language of first-order logic are terms. We fix a set of variables, $\{ v_0, v_1, v_2, \dots \}$, each of which is a term. Intuitively, a term denotes an individual.

We also have a set of relation, or predicate, symbols. For each non-negative integer n, we have a set of relation symbols of arity n, $\{p^n_0, p^n_1, \dots \}$. For any collection of terms $\{ t_1, t_2, \dots, t_n \}$, the expression $p^n_i(t_1, \dots, t_n)$ is a formula. Other formulae are built from the logical connectives $\sim$ and $\supset$ as before; if ϕ and ψ are formulae, then so too are $\sim\phi$ and $\phi \supset \psi$. Additionally, for any variable x, $(\forall x)\phi$ is a formula if and only if ϕ is a formula. We define free and bound variables in the typical way. We will write $\phi(x)$ to indicate that x may appear free in ϕ and $\phi(a/x)$ to indicate the formulae obtained from ϕ by substituting all free occurrences of x with a.

The axiomatic treatment of first-order logic has the axiom schemata of the propositional calculus as well as one additional axiom schema for the universal quantifier, given now.

$$[(\forall x)\phi(x)] \supset \phi(a/x) \quad \text{where } a \text{ does not become bound in } \phi \tag{FOL}$$

Additionally, there is one new inference rule.

$$\frac{\phi \supset \psi(x)}{\phi \supset (\forall x)\psi(x)}\ \forall \text{ rule} \quad \text{where } x \text{ does not appear free in } \phi$$

2.1.2 Natural Deduction

The axiomatic approach to logical systems is relatively minimal, a property which leads to certain pleasant mathematical properties. However, the small size of the logical system tends to make proofs rather large and unwieldy. For instance, the proof of $p \supset p$ given above requires five lines, and the derivation is not particularly intuitive. The theorem, however, is intuitively acceptable almost immediately.

Gentzen (1935) developed the first natural-deduction proof systems as alternatives to the axiomatic systems that would more closely follow the actual processes of mathematical reasoning. Natural-deduction systems, like axiomatic systems, begin by specifying a logical language, i.e., a set of well-formed formulae, but, in contrast, have few, if any, axiom schemata, instead incorporating a larger set of rules. In contrast to axiomatic approaches which begin with instances of axioms (and only add the notion of premises later), natural-deduction proof systems are concerned with assumptions from the very beginning. The process of constructing proofs in a natural-deduction system is that of establishing judgments of the form $\Gamma \vdash \phi$ based on the prior establishment of other judgements.

Prawitz (1965) provides a thorough coverage of natural-deduction proof systems for a number of logics, including first-order logic, second-order logic, intuitionistic logic, and modal logic, and examines the concept of proof normalization, as well.

Example 13 (Natural-Deduction Propositional Calculus (Tree Style)). We consider a natural-deduction proof system for the propositional calculus. The language is the same as that defined in Example 10 (p. 17). The logical system has no axiom schemata, but has several introduction and elimination inference rules for each of the logical connectives (i.e., $\sim$, &, $\vee$, and $\supset$). For instance, if p and q have been derived with respect to some set of premises, then $p \mathbin{\&} q$ can be derived using that same set of premises using the conjunction introduction rule, $\&I$. Schematically,

$$\dfrac{\begin{array}{cc}\vdots & \vdots\\ p & q\end{array}}{p \mathbin{\&} q}\;\&I$$

Premises are introduced with the $[\cdot]$ notation. For instance, the following derivation of $p \mathbin{\&} r$ is based on two premises: p and $q \mathbin{\&} r$:

$$\dfrac{[p] \qquad \dfrac{[q \mathbin{\&} r]}{r}\;\&E_2}{p \mathbin{\&} r}\;\&I_1$$

Some rules discharge previously introduced assumptions. For instance, when a formulae ψ has been derived using an assumption ϕ, conditional introduction is used to discharge the assumption ϕ and conclude the conditional $\phi \supset \psi$. Using this rule, the conditional $p \supset (q \vee p)$ can be established by deriving $q \vee p$ from an assumption p and then discharging the assumption with conditional introduction:

$$\dfrac{\dfrac{[p]}{q \vee p}\;\vee I_2}{p \supset (q \vee p)}\;\supset I$$

$$\dfrac{\begin{matrix}[\sim\phi]\\ \vdots\\ \psi \mathbin{\&} \sim\psi\end{matrix}}{\phi}\sim E \qquad \dfrac{\begin{matrix}[\phi]\\ \vdots\\ \psi \mathbin{\&} \sim\psi\end{matrix}}{\sim\phi}\sim I$$

$$\dfrac{\phi \mathbin{\&} \psi}{\phi}\,\&E_1 \quad \dfrac{\phi \mathbin{\&} \psi}{\psi}\,\&E_2 \qquad \dfrac{\phi \quad \psi}{\phi \mathbin{\&} \psi}\,\&I$$

$$\dfrac{\phi \vee \psi \quad \begin{matrix}[\phi]\\ \vdots\\ \rho\end{matrix} \quad \begin{matrix}[\psi]\\ \vdots\\ \rho\end{matrix}}{\rho}\vee E \qquad \dfrac{\phi}{\phi \vee \psi}\vee I_1 \quad \dfrac{\psi}{\phi \vee \psi}\vee I_2$$

$$\dfrac{\phi \quad \phi \supset \psi}{\psi}\supset E \qquad \dfrac{\begin{matrix}[\phi]\\ \vdots\\ \psi\end{matrix}}{\phi \supset \psi}\supset I$$

Figure 2.1: Natural-deduction rules for the propositional calculus.

The inference rules used in this natural-deduction proof system are given in Figure 2.1.

Proofs of the judgments $\vdash p \supset p$ and $\{\sim\sim\sim p \supset \sim p\} \vdash p \supset \sim\sim p\}$ are now given for comparison with their axiomatic counterparts developed earlier in Example 10 (p. 17).

$$\dfrac{[p]}{p \supset p}\supset I \qquad \dfrac{\dfrac{\dfrac{[p] \quad \dfrac{[\sim\sim\sim p] \quad [\sim\sim\sim p \supset \sim p]}{\sim p}\supset E}{p \mathbin{\&} \sim p}\,\&I}{\sim\sim p}\sim E}{p \supset \sim\sim p}\supset I$$

Example 14 (Natural-Deduction Propositional Calculus (Fitch Style)). The tree-form presentation of natural-deduction proofs in the previous example is common. Another common form was developed by Fitch (1952) and uses conditional subproofs for structuring and regulating the scope of assumptions. Fitch's treatment also incorporates more of the linear structure of the presentation of proofs. A Fitch style proof is essentially a hierarchy of subproofs. Within each subproof is a sequence of steps, each of which is either an application of a rule to previous steps, or a nested subproof. As an example, consider the following Fitch-style derivation of $p \supset (q \supset r)$ from the premise $(p \,\&\, q) \supset r$.

```
1  │ (p & q) ⊃ r
   ├──
2  │ │ p
   │ ├──
3  │ │ │ q
   │ │ ├──
4  │ │ │ p & q          &I 2, 3
5  │ │ │ r              ⊃E 1, 4
6  │ │ q ⊃ r            ⊃I 3–5
7  │ p ⊃ (q ⊃ r)        ⊃I 2–6
```

The first line introduces the premise, or assumption, $(p \,\&\, q) \supset r$. The second line introduces p, and the third line introduces q. The fourth line is in the scope of the assumptions p and q and uses conjunction introduction to produce $p \,\&\, q$, citing the second and third lines. The fifth line uses conditional elimination, or *modus ponens* to derive r from the initial assumption and the derived conjunction $p \,\&\, q$. At this point, the subproof that began on the third line is terminated. The sixth line uses conditional introduction to cite the entire subproof that ran from the third line to the fifth line, which started with the assumption q and ended with the conclusion r, to produce the conditional $q \supset r$. At this point the subproof that introduced the assumption p is terminated, and the seventh line uses conditional introduction, citing the just-terminated subproof, to introduce the conditional $p \supset (q \supset r)$.

We have used the same names for rules used in this Fitch-style presentation of the propositional calculus as those used in the tree-style presentation, and the intended meanings should be obvious. The Fitch-style presentation of the propositional calculus is quite intuitive and readable. Specifying the mechanics of proof construction is somewhat more complicated, as the lines citeable by inference rules are restricted by assumption scope. For instance, it would not be permissible for the sixth line to conclude $p \,\&\, q$ by conjunction introduction citing the second and third lines, for the scope of the assumption q has been terminated. Intuitively, rules may only cite earlier lines at a level equal or higher than their own.

We shall not spend a great deal of time working with Fitch-style proofs, as their mechanics tend to be somewhat complicated, but they will prove quite useful in comprehending the scope of different reasoning contexts. (In this case, the “new context” is that of having an additional premise.)

Example 15 (Natural-Deduction **T** (Tree Style)). We now consider a natural-deduction treatment of the modal logic **T**. We use the same basic language as that used in the natural deduction propositional calculus (i.e., a set of propositional formulae, each of which is a formulae, and we take all the logical connectives $\sim$, &, $\vee$, and $\supset$, as primitive). In addition, if ϕ is a formulae, then so too is $\Box\phi$.

This natural-deduction treatment of **T** has no axioms, but incorporates the rules of the propositional calculus just developed, as well as two additional rules for handling modalities.

The first inference rule, necessity elimination, or $\Box E$, is straightforward:

$$\frac{\Box\phi}{\phi}\ \Box E$$

Necessity elimination captures the behavior of (M).

The second modal inference rule, necessity introduction, or $\Box I$, has the following form:

$$\frac{\begin{array}{c}\vdots\\ \phi\end{array}}{\Box\phi}\ \Box I$$

This appears to say that when ϕ is derivable, so is $\Box\phi$. It is the case that if ϕ is a theorem, then $\Box\phi$ should be too, but what happens in the case that the derivation of ϕ includes an assumption? We certainly should not be able to infer $\Box\phi$ by first assuming ϕ! Indeed, uses of $\Box I$ must satisfy an additional constraint, namely that any undischarged assumption used in the proof of ϕ must be of the form $\Box\psi$ and must be immediately used by necessity elimination.

Here is a proof demonstrating the use of $\Box E$ and $\Box I$ to show that $\{\Box\Box p, \Box(p \supset q)\} \vdash \Box(q \vee "r)$.

$$\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{[\Box\Box p]}{\Box p}\ \Box E}{p}\ \Box E \qquad \cfrac{[\Box(p \supset q)]}{p \supset q}\ \Box E}{q}\ \supset E}{q \vee r}\ \vee I_1}{\Box(q \vee r)}\ \Box I$$

Remark 1 (Other Modal Natural Deduction Rules). Numerous variations of the natural-deduction modal rules have been proposed, but most share common themes. We required that the undischarged assumptions in a necessity introduction subproof all be modal and that necessity elimination was used immediately after assumption. Satre (1972) surveyed a variety of proposed sequent-based natural deduction rules for a number of modal logics gives several that are essentially equivalent to the one we provided, including these two (the subscripts appear in Satre's (1972) list and are provided for reference):

$$\frac{\Gamma \vdash \Box\Delta \quad \Delta \vdash B}{\Gamma \vdash \Box B}\ \Box I_1 \qquad\qquad \frac{\Delta \vdash B}{\Delta \vdash \Box B}\ \Box I_{11}\ \text{provided that all hypothesis of } \Delta \text{ are of the form of either } \Box C \text{ or } \Diamond C \text{ for some wff } C$$

Prawitz (1965) relaxes the restriction on undischarged assumptions through the use of "essentially modal formulae." Undischarged assumptions no longer need be modal formulae, but the inferences drawn from them must, at some point before the final conclusion, be modal formulae. Bierman & de Paiva (2000, p. 390) take a different approach, wherein all undischarged assumptions must be modal, and all are discharged by the necessity introduction rule. This comparison is shown in the following; Prawitz's rule is shown on the left, Bierman & de Paiva's on the right.

$$\begin{array}{c} \begin{array}{ccc} \Delta_1 & & \Delta_k \\ \vdots & & \vdots \\ \Box B_1 & \cdots & \Box B_k \end{array} \\ \vdots \\ \dfrac{A}{\Box A}\ \Box I \end{array} \qquad\qquad \dfrac{\begin{array}{ccc} \vdots & & \vdots \\ \Box A_1 & \cdots & \Box A_k \end{array} \quad \begin{array}{c} [\![\Box A_1^{x_1} \ \cdots \ \Box A_k^{x_k}]\!] \\ \vdots \\ B \end{array}}{\Box B}\ \Box I_{x_1,\cdots,x_k}$$

Example 16 (Natural Deduction **T** (Fitch Style)). Konyndyk (1986) presents a Fitch-style proof system for the modal logic **T**, and it will prove instructive to examine this system here. The Fitch-style proof system for **T** builds upon the Fitch-style propositional calculus system by extending the language such that where ϕ is a formula, so too is $\Box\phi$, and extends the propositional calculus with three new inference rules. The first, necessity elimination, is the same as that given for the tree-based presentation:

$$\begin{array}{r|ll} 1 & \Box\phi & \\ 2 & \vdots & \\ 3 & \phi & \Box E\ 1 \end{array}$$

The other two rules are **T**-reiteration and necessity introduction, and both refer to the final new construct of this proof calculus, the modal subproof. In addition to the conditional introduction subproofs already seen, the Fitch-style proof system for **T** has modal subproofs which permit reasoning in the context of what is necessary. A modal subproof introduces a new "scope" and is visually similar to a conditional subproof, but does not introduce any assumptions. Modal subproofs are used with two inference rules: (i) necessity introduction or $\Box I$; and (ii) **T** reiteration.

The necessity introduction rule cites a necessity subproof and derives $\Box\phi$ when one of the formulae derived in the necessity subproof is ϕ:

$$\begin{array}{r|l} 1 & \begin{array}{c|l} \Box & \vdots \\ & \phi \end{array} \\ 2 & \\ 3 & \Box\phi \qquad \Box I\ 1\text{–}2 \end{array}$$

The necessity subproof intuitively represents reasoning about things which are necessary, and necessity introduction is the mechanism by which these necessary results are extracted from the subproof. In order to ensure that the reasoning occurring within a necessity subproof cites only necessary things, rules used within a necessity subproof, with one exception, cannot cite formulae outside of the subproof.

The **T** reiteration rule is the sole exception to the restriction on what can be cited in a necessity introduction subproof. Using **T** reiteration, we may cite a formulae $\Box\phi$ that occurs one level outside the necessity subproof and reiterate ϕ inside the subproof:

```
1  | □φ
2  |□| ⋮
3  | | φ        T reit 1
```

Using this proof system, we can recreate the tree-based derivation of $\{ \Box\Box p, \Box(p \supset q) \} \vdash \Box(q \vee r)$:

```
1  | □□p
2  | □(p ⊃ q)
   |----
3  |□| □p         T reit 1
4  | | p          □E 3
5  | | p ⊃ q      T reit 2
6  | | q          ⊃E 4,5
7  | | q ∨ r      ∨I 6
8  | □(q ∨ r)     □I 3–7
```

Recall that the necessitation rule in the axiomatic presentation of **T** affirmed that any theorem is necessary. That is, if $\vdash \phi$, then $\Box\phi$. Since conditional subproofs can also occur within a necessity subproof, any theorem can be derived within a necessity subproof. For instance, consider the theorem $((p \,\&\, q) \supset r) \supset (p \supset (q \supset r))$, which we derive within a necessity subproof and to which we apply necessity introduction.

```
1  |□| | (p & q) ⊃ r
   | | |----
2  | | | | p
   | | | |----
3  | | | | | q
   | | | | |----
4  | | | | | p & q                     &I 2, 3
5  | | | | | r                         ⊃E 1, 4
6  | | | | q ⊃ r                       ⊃I 3–5
7  | | | p ⊃ (q ⊃ r)                   ⊃I 2–6
8  | | ((p & q) ⊃ r) ⊃ (p ⊃ (q ⊃ r))   ⊃I 1–7
9  | □((p & q) ⊃ r) ⊃ (p ⊃ (q ⊃ r))   □I 1–8
```

Example 17 (First-Order Logic (Tree & Fitch Style)). We now look at tree and Fitch-style natural-deduction proof systems for first-order logic. The language is exactly the same as the language of the axiomatic counterparts. Using the same natural-deduction pattern of introduction and elimination rules already developed, we expect to see an introduction and elimination rule for the new logical symbol, the universal quantifier $\forall$. The elimination rule is straightforward: from a universal $(\forall x)\phi(x)$ we may infer any instance of ϕ with a term t substituted for x.

$$\dfrac{(\forall x)\phi(x)}{\phi(t/x)}\ \forall E$$

The corresponding introduction rule is somewhat more complex, in that it places an extra constraint on assumptions appearing earlier in the proof.

$$\dfrac{\begin{matrix}\vdots\\ \phi(a/x)\end{matrix}}{(\forall x)\phi(x)}\ \forall I$$

when a does not appear free in any undischarged assumption in the proof of $\phi(a/x)$

The constraint that a does not appear free in any undischarged assumption of the proof of $\phi(a/x)$ ensures that a is acting as an "arbitrary name," i.e., one about which no extra assumptions have been made.

Here is a proof of $(\forall z)(p(z) \supset r(z))$ from $(\forall x)(p(x) \supset q(x))$ and $(\forall y)(q(y) \supset r(y))$. Note that in the use of universal introduction, there was an assumption in which a appeared free, namely $p(a)$, but that it was discharged by the conditional introduction that produced $p(a) \supset r(a)$.

$$\dfrac{\dfrac{\dfrac{\dfrac{[p(a)] \quad \dfrac{[(\forall x)(p(x) \supset q(x))]}{p(a) \supset q(a)}\ \forall E}{q(a)}\ \supset R \qquad \dfrac{[(\forall y)(q(y) \supset r(y))]}{q(a) \supset r(a)}\ \forall E}{r(a)}\ \supset E}{p(a) \supset r(a)}\ \supset I}{(\forall z)(p(z) \supset r(z))}\ \forall I$$

The Fitch-style treatment of first-order logic adopts universal elimination in the obvious form, but introduces a new type of subproof to handle universal introduction. In **T**, the necessity subproof provided a way by which to reason in a new context, that of things that are necessary. In first order logic, we have "arbitrary individual" subproofs in which a designated name is understood to denote an arbitrary individual. Let us examine the same proof in the Fitch-style proof system:

1	$\vert\ (\forall x)(p(x) \supset q(x))$	
2	$\vert\ \underline{(\forall y)(q(y) \supset r(y))}$	
3	$\vert\ {}^{a}\vert\ p(a) \supset q(a)$	$\forall E$ 1
4	$\vert\ \ \vert\ \vert\ \underline{p(a)}$	
5	$\vert\ \ \vert\ \vert\ q(a)$	$\supset E$ 3, 4
6	$\vert\ \ \vert\ \vert\ q(a) \supset r(a)$	$\forall E$ 2
7	$\vert\ \ \vert\ \vert\ r(a)$	$\supset E$ 5, 6
8	$\vert\ \ \vert\ p(a) \supset r(a)$	$\supset I$ 4–7
9	$\vert\ (\forall z)(p(z) \supset r(z))$	$\forall I$ 3–8

While the "arbitrary individual" subproof make no restrictions on what lines rules may cite, it is required that the name a does not appear outside of the subproof that introduces it.

Remark 2 (Comparing Tree- and Fitch-Style Proof Systems). In the preceding example we have presented in parallel both tree-style and Fitch-style proof systems for the propositional calculus, for first-order logic, and for the alethic modal logic **T**. In all three logics, the set of theorems provable in each proof calculus is the same (though we did not prove this), and we express no preference between the two.

However, several remarks are in order. From a practical standpoint, the tree-style proof systems are somewhat easier to specify. The Fitch-style systems tend to involve more complicated constraints on what can and cannot happen within various types of subproofs. The counterpart of these constraints in the tree-style proofs are those constraints on the applicability of inference rules. However, the Fitch-style proofs tend to better visually indicate different types of reasoning contexts, e.g., the scope of an assumption or name, or the extent of a "necessary" context. There is one respect in particular where the Fitch-style proofs seem superior to the tree-style proofs. Consider the following Fitch-style proof in the propositional calculus.

1	$\vert\ \vert\ \underline{p}$	
2	$\vert\ \vert\ p \vee q$	$\vee I$ 1
3	$\vert\ \vert\ p \vee r$	$\vee I$ 1
4	$\vert\ \vert\ (p \vee q) \,\&\, (p \vee r)$	$\& I$ 2, 3
5	$\vert\ p \supset ((p \vee q) \,\&\, (p \vee r))$	$\supset I$ 1–4

The assumption p is cited twice and discharged once. In a tree-style proof, conditional introduction must be allowed to discharge multiple (identical) assumptions at once, as in the following tree-style proof of the same theorem.

$$\dfrac{\dfrac{\dfrac{[p]}{p \vee q}\vee I_1 \qquad \dfrac{[p]}{p \vee r}\vee I_1}{(p \vee q) \mathbin{\&} (p \vee r)}\,\&I}{p \supset ((p \vee q) \mathbin{\&} (p \vee r))}\supset I$$

This is not a problem, *per se*, but it does illustrate that the most intuitive way of verbalizing a deduction may correspond more closely to top-to-bottom reading of a Fitch-style proof than of some reading of the corresponding tree-style proof.

2.2 Category Theoretic Treatments

Category theory is a mathematical formalism similar to abstract algebra, but is sufficiently general to formalize a very large number of mathematical constructions. Category theory's importance, however, lies not just in its usefulness as a general structure to describe mathematical conceptions, but in the importance it places on examining the relationships between those structures. The use of category theory to represent proof systems was pioneered largely by Lambek (1968).[2]

2.2.1 Deductive Systems

The categorical treatment of logical systems is a natural progression from a treatment of logical system as *deductive systems*. Before presenting deductive systems, we first need an auxiliary notion of a (directed) graph.

Definition 18 (Graph)**.** A graph consists in a collection of objects and a collection of directed edges (or arrows) among the objects. An arrow f from an object A to an object B is written $f : A \to B$ (Lambek & Scott, 1988, p. 5).

By imposing two simple constraints on a graph and viewing objects as formulae and arrows as proofs, we obtain deductive systems.

Definition 19 (Deductive System)**.** A deductive system is a graph which has: an arrow $\mathrm{id}_A : A \to A$ for each object A; a composite arrow $g \circ f : A \to C$ for each pair of arrows $f : A \to B$ and $g : B \to C$ (Lambek & Scott, 1988, p. 47).

We are often concerned with demonstrating the existence of particular arrows. Toward this end we

[2]The 1968 paper establishes precedent, but later works, (e.g., Lambek, 1989), are more approachable for the casual reader.

usually present families of arrows schematically. For instance, the schema

$$\mathrm{id}_A : A \to A$$

represents the class of identity arrows, where A is understood as a variable to be replaced by an object. Thus by this schema we may demonstrate the existence of an arrow $\mathrm{id}_B : B \to B$ in a deductive system with an object B. Similarly, the composition schema

$$\frac{f : A \to B \quad g : B \to C}{g \circ f : A \to C}$$

allows us to demonstrate the existence of an arrow $q \circ p : a \to c$ given two arrows $p : a \to b$ and $q : b \to c$.

In identifying arrows with proofs and objects with logical representations, we impose two constraints on proof systems that were not present before. The first is that there are "identity proofs," and the second is that proofs compose.

The requirement that there are "identity proofs" may be somewhat unusual, but is not controversial: we are usually willing to accept that a formula or sentence is provable from itself.

The requirement that proofs compose may also be somewhat unusual, but only because proof composition occurs so frequently and so naturally in both formal and informal reasoning that it may seem odd to make it explicit. Any time an appeal is made to well-known (and oft-derived) propositions such as the infinitude of the primes or the irrationality of $\sqrt{2}$, proof composition is occurring. Sometimes proofs are even "primed" for composition; a computer scientist who proves a result contingent on the assumption that $P \neq NP$ may implicitly claim that the result is seen to be theorem when composed with a yet unknown proof of the assumption. Alternatively, a proof that some assumption leads to a proposition widely supposed to be contradictory is often presented to support the hypothesis that the assumption is contradictory; this is based on another implicit composition.

Deductive systems make proof composition explicit. If there is a proof g of C from B and f of B from A, we are typically comfortable asserting the existence of some proof $g \circ f$ of C from A. That is, for any proofs $g : B \to C$ and $f : A \to C$ there is a proof $g \circ f : A \to C$.

Note that we are now in a position to distinguish different proofs that have the same premise and conclusion. For instance, given the proofs $f : A \to B$ and $g, g' : B \to C$, both $g \circ f : A \to C$ and $g' \circ f : A \to C$ are proofs of C from A, but because we can reference the proofs by name, we can ask questions such as whether $g' \circ f = g \circ f$, that is, whether the proofs are equivalent. Classical logic has traditionally been concerned with the question of whether one proposition entails another, and has been less concerned with the proofs that demonstrate the entailment. In a deductive system (and more generally, in categorical proof theory) proofs, realized as arrows, are themselves objects of study.

Example 20 (Positive Intuitionistic Propositional Calculus). A deductive system with an object $\top$ (truth) and objects $A \,\&\, B$ and $A \supset B$ whenever A and B are objects, along with the following arrows, is a positive intuitionistic propositional calculus:

$$A \xrightarrow{\bigcirc_A} \top$$

$$A \,\&\, B \xrightarrow{\pi_{A,B}} A \qquad A \,\&\, B \xrightarrow{\pi'_{A,B}} B \qquad \frac{C \xrightarrow{f} A \quad C \xrightarrow{g} B}{C \xrightarrow{\langle f,g\rangle} A \,\&\, B}$$

$$\frac{C \,\&\, A \xrightarrow{h} B}{C \xrightarrow{h^*} A \supset B} \qquad (A \supset B) \,\&\, A \xrightarrow{\epsilon_{A,B}} B$$

From top to bottom, and left to right, these arrows six arrow types correspond to the six inference rules of the positive intuitionistic propositional calculus: (i) truth (i.e., $\top$, is provable from any proposition A); (ii) A is provable from $A \,\&\, B$; (iii) B is provable from $A \,\&\, B$; (iv) given proofs of A and B from C, there is a proof of $A \,\&\, B$ is also provable from C; (v) given a proof of B from $C \,\&\, A$, there is proof of $A \supset B$ from C; (vi) B is provable from $(A \supset B) \,\&\, A$. Common logical results can be reproduced using the deductive system. For instance, the following is a demonstration that $B \,\&\, A$ is provable form $A \,\&\, B$, i.e., that there is an arrow $A \,\&\, B \to B \,\&\, A$, particularly, $\langle \pi_{A,B}, \pi'_{A,B} \rangle$.

$$\frac{A \,\&\, B \xrightarrow{\pi_{A,B}} A \qquad A \,\&\, B \xrightarrow{\pi'_{A,B}} B}{A \,\&\, B \xrightarrow{\langle \pi_{A,B}, \pi'_{A,B}\rangle} B \,\&\, A}$$

As another example, we may show the associativity of conjunction. For brevity, we abbreviate $(A\&B)\&C$ with ϕ.

$$\frac{\dfrac{\phi \xrightarrow{\pi_{A\&B,C}} A \,\&\, B \qquad A \,\&\, B \xrightarrow{\pi_{A,B}} A}{\phi \xrightarrow{\pi_{A,B} \circ \pi_{A\&B,C}} A} \qquad \dfrac{\dfrac{\phi \xrightarrow{\pi_{A\&B,C}} A \,\&\, B \qquad A \,\&\, B \xrightarrow{\pi'_{A,B}} B}{\phi \xrightarrow{\pi'_{A,B} \circ \pi_{A\&B,C}} B} \qquad \phi \xrightarrow{\pi'_{A\&B,C}} C}{\phi \xrightarrow{\langle \pi'_{A,B} \circ \pi_{A\&B,C}, \pi'_{A\&B,C}\rangle} B \,\&\, C}}{(A \,\&\, B) \,\&\, C \xrightarrow{\langle \pi_{A,B} \circ \pi_{A,B}, \langle \pi'_{A,B} \circ \pi_{A\&B,C}, \pi'_{A\&B,C}\rangle\rangle} A \,\&\, (B \,\&\, C)} \tag{2.1}$$

We will reference the schema from (2.1) again later, and call it $\alpha_{A,B,C}$. That is,

$$\alpha_{A,B,C} : (A \,\&\, B) \,\&\, C \to A \,\&\, (B \,\&\, C)$$

Remark 3 (Concerning Notation). As seen in (2.1), the full notation for arrow labels can quickly become very dense. As a matter of notational convenience, we will often leave subscripts implicit when they are clear from context (e.g., $\pi : A \,\&\, B \to A$ rather than $\pi_{A,B} : A \,\&\, B \to A$), and will often write the composition

$g \circ f$ as gf. In order to conserve vertical space, we will often use $f : A \to B$ in preference to

$$A \xrightarrow{f} B$$

This makes the derivation of some arrows clearer. For instance, leaving the derivation and subscripts of $\pi\pi \colon (A \,\&\, B) \,\&\, C \to A$ implicit, the derivation of (2.1) becomes:

$$\dfrac{\pi\pi \colon (A \,\&\, B) \,\&\, C \to A \qquad \dfrac{\pi'\pi \colon (A \,\&\, B) \,\&\, C \to B \qquad \pi' \colon (A \,\&\, B) \,\&\, C \to C}{\langle \pi'\pi, \pi' \rangle \colon (A \,\&\, B) \,\&\, C \to B \,\&\, C}}{\alpha_{A,B,C} \equiv \langle \pi\pi, \langle \pi'\pi, \pi' \rangle \rangle \colon (A \,\&\, B) \,\&\, C \to A \,\&\, (B \,\&\, C)} \tag{2.2}$$

2.2.2 Categories

Deductive systems provide a consistent notation for logical systems, and have the benefit of asking us to consider the possible equivalence of proofs, and requiring us to consider the composition of proofs. However, given the existence of identity arrows as well as a proof composition operator, there are certain equivalences that we might like to take for granted. Particularly, we might like to know that identity proofs actually behave as identities for composition. Additionally, composition in many applications is associative, and it would be reasonable to consider whether proof theory is such an application. Imposing these requirements, we obtain categories.

Definition 21 (Category)**.** A category is a deductive system in which the following equivalences hold for all $f : A \to B$, $g : B \to C$, and $h \colon C \to D$.

$$(h \circ g) \circ f = h \circ (g \circ f) \qquad\qquad \mathrm{id}_B \circ f = f = f \circ \mathrm{id}_A$$

That is, composition is associative and identity arrows are identities.

2.2.2.1 Mathematical Structures as Categories

One of the most significant benefits of working with logical systems as categories is that there is a large literature within category theory on various topics in mathematics, including logic. The abundance of categorical literature is a result of the widespread applicability of category theory to various domains. Many mathematical structures can be naturally represented as categories, and categorical results are easily transferred between these domains.

Example 22 (Preorders). A set P with a relation $\leq$ is a category whose objects are the elements of P and which has an arrow $A \to B$ if and only if $A \leq B$ and which has at most one arrow from any object to another. To confirm that a preorder is, or gives rise to, a category, we must check that it has identity arrows and composition, and that composition behaves properly with respect to the identity arrows. For each object A, since $A \leq A$ there is an arrow from $A \to A$, and since there is at most one arrow from any one object to another, this arrow must be taken as id_A. For any arrows $f : A \to B$ and $g : B \to C$, we must consider whether there is a composite $g \circ f$. The existence of f indicates that $A \leq B$, and of g that $B \leq C$. Since $\leq$ is transitive, it must be that $A \leq C$, and thus that there is an arrow $A \to C$. Since this is the only such arrow, it will be taken as the composite. Similar reasoning shows that the only candidate for $\mathrm{id}_B \circ f$ and for $f \circ \mathrm{id}_A$ is f, and so composition respects identities. To show that composition is associative, it suffices to note that where the composite $(h \circ g) \circ f : A \to D$ is defined (for an appropriate $h : C \to D$), it is the only $A \to D$ arrow and so must be $h \circ (g \circ f)$ as well.

Example 23 (Natural Numbers). The natural numbers give rise to a category $\mathscr{N}$ with a single object and whose arrows are the natural numbers $0, 1, \ldots$ with addition $+$ as the composition operator. This case is an interesting contrast with that of preorders, in that for a preorder $\langle P, \leq \rangle$, the objects of the category were simply the elements of P, but here there is just one object in the category, but many arrows. Yet the process for checking whether the resulting structure is a category is the same. We denote the single object by $*$ and observe that to each natural number n there is a corresponding arrow $n : * \to *$. While every arrow has $*$ as its source and target, taking $+$ as the composition operator, we see that $\mathrm{id}_* = 0$, since 0 is the identity for $+$. Similarly, we know that $+$ is associative and total.

Example 24 (Deductive Systems as Categories). Deductive systems give rise to categories when the appropriate identities are imposed. This example points out that we can obtain a category by taking a deductive system and imposing an equivalence relationship on its arrows such that the identity arrows are identities with respect to composition and such that the composition operator is associative. What this means in practice is that given a deductive system, we no longer consider individual proofs of one formula from another, but rather equivalence classes of proofs. In the example shown so far (the positive intuitionistic propositional calculus) these equivalence classes are not particularly interesting. However, more significant results arise when other categorical notions are brought to bear on categories obtained from deductive systems.

2.2.2.2 Categorical Constructions

The power of category theory comes not only from providing a convenient and uniform way of representing many different types of mathematical structures, but also in giving a framework in which to define common mathematical structures at a sufficiently abstract level. When mathematical constructions are expressed using category theory, they can be immediately realized in numerous categories and categorical results are immediately applicable.

Example 25 (Products). A great number of mathematical structures have a notion of product and many of these can be unified by treating the mathematical structure as a category and examining the more abstractly defined categorical product.

A product of objects A and B is an object $A \times B$ with arrows $\pi_{A,B} : A \times B \to A$ and $\pi'_{A,B} : A \times B \to B$ such that for any arrows $f : D \to A$ and $g : D \to B$ there is a unique arrow $\langle f, g \rangle$, such that $f = \pi_{A,B} \circ \langle f, g \rangle$ and $g = \pi'_{A,B} \circ \langle f, g \rangle$.

$$\begin{tikzcd}
 & D \arrow[dl, "f"'] \arrow[d, dashed, "\langle f, g \rangle"] \arrow[dr, "g"] & \\
A & A \times B \arrow[l, "\pi_{A,B}"] \arrow[r, "\pi'_{A,B}"'] & B
\end{tikzcd}$$

The figure here, called a commutative diagram, is common in category theory and serves to identify a number of arrows. In commutative diagrams, all paths from one object to another denote the same arrow. Thus in this figure, $f = \pi_{A,B} \circ \langle f, g \rangle$ and $g = \pi'_{A,B} \circ \langle f, g \rangle$. That an arrow, in this case $\langle f, g \rangle$, is drawn with a dashed line indicates that it is unique. That is, the diagram asserts that (for every pair of arrows f and g) there is a unique arrow $\langle f, g \rangle$ such that $f = \pi_{A,B} \circ \langle f, g \rangle$.

We can now consider examples of products in various categories.

Example 26 (Preorders). In preorders, products are greatest lower bounds. If an object C is such that there are arrows $x : C \to A$ (so $C \leq A$) and $y : C \to B$ (so $C \leq B$) then C is a lower bound of A and B. Then the product $A \times B$ is a lower bound of A and B by virtue of the projections p_1 and p_2. The requirement that for any D with arrows $f : D \to A$ and $g : D \to B$ there is a unique arrow $\langle f, g \rangle : D \to A \times B$ ensures that $A \times B$ is the greatest lower bound of A and B. Suppose that some C ($\neq A \times B$) is the greatest lower bound of $A \times B$. Then there are arrows $f' : C \to A$ and $g' : C \to B$ and thus a unique arrow $\langle f', g' \rangle : C \to A \times B$, but this would mean that $C \leq A \times B$. Since $A \times B$ is also a lower bound, we have $A \times B \leq C$. But then, contrary to assumption, $A \times B = C$. Then $A \times B$ is the greatest lower bound of A and B.

Example 27 (Sets). In the category of sets (objects are sets and arrows are set functions), the Cartesian product of sets S and T is a categorical product of S and T. We say "a categorical product" rather than "the categorical product," as every set isomorphic to $S \times T$ is a categorical product of S and T.

Given an isomorphism $i : X \to S \times T$, the projections of X as a product of S and T are simply $\pi_{A,B} i : X \to A$ and $\pi'_{A,B} i : X \to B$. For any arrows $f : U \to A$ and $g : U \to B$, $i\langle f, g \rangle$ is unique because i is an isomorphism. Thus X is also a categorical product of S and T.

Example 28 (Logics). In categories arising from deductive systems for logics with conjunctions, it is common to impose an equivalence relation on proofs that make conjunctions categorical products. A product of formulae A and B is the formula $A\&B$ with arrows $\pi_{A,B} : A\&B \to A$ (left conjunction elimination) and $\pi'_{A,B} : A \& B \to B$ (right conjunction elimination) such that if f and g are proofs of A and B from some formula D, there is a unique proof $\langle f, g\rangle$ of $A \& B$ from D. What is actually happening here is the observation that logical conjunction seems like a kind of product, and we adopt the necessary proof equivalences to make it so. Typically the arrow $\langle f, g\rangle$ is understood as the proof that combines the results of the proofs f and g using conjunction introduction. Accepting $A \& B$ as the product of A and B has the effect of equating the following proofs.

$$\dfrac{\dfrac{A \& (B \& C)}{B \& C}\,\&E_R}{B}\,\&E_L \qquad\qquad \dfrac{\dfrac{\dfrac{A \& (B \& C)}{A}\,\&E_L \quad \dfrac{\dfrac{A \& (B \& C)}{B \& C}\,\&E_R}{B}\,\&E_L}{A \& B}\,\&I}{B}\,\&E_R$$

The equivalence is reasonable, since the proof on the right derives B, then uses conjunction introduction to derive $A \& B$, and then immediately uses conjunction elimination to re-derive B. The proof on the left might be understood as a normalized form of the proof on the right. This example highlights the point that arrows in logic categories are equivalence classes of proofs.

Remark 4 (Categorical Constructions are Unique up to Isomorphism). As demonstrated in the category of sets and in the categorical presentation of logical systems, categorical products are unique up to isomorphism. This is typical of categorical definitions; constructions are often specified up to isomorphism.

2.2.2.3 Relationships Between Categories

Categories obtained from deductive systems provide a simple, yet powerful, mechanism for representing logical systems, and many of the common points of interest in logics can be expressed categorically. The language of categories, then, seems to fulfill our first requirement, that of representing logical systems.

The second requirement, that of representing the interactions between logical systems and the relationships among these interactions, will now be addressed through functors and natural transformations. Functors, or category homomorphisms, provide a way of understanding relationships between logical systems represented as categories, and natural transformations provide a way of understanding relationships between functors.

Definition 29 (Functor). A functor $F : \mathscr{C} \to \mathscr{D}$ is a mapping of $\mathscr{C}$ objects and arrows to $\mathscr{D}$ objects and arrows such that for any $\mathscr{C}$ arrows $f : A \to B$ and $g : B \to C$, there are $\mathscr{D}$ objects and a $\mathscr{D}$ arrow such that:

$$F(f) : F(A) \to F(B)$$

and which respects identity and composition:

$$F(\mathrm{id}_A) = \mathrm{id}_{F(A)}$$
$$F(g \circ_{\mathscr{C}} f) = F(g) \circ_{\mathscr{D}} F(f)$$

It is illustrative to consider the realization of functors between categories that represent logical systems. Since objects in these categories are formulae, the object mapping of a functor is a formula translation. Similarly, since arrows are proofs, the arrow mapping of a functor is a proof translation. Formulae translations have long been used in the study of logics, particularly in examining their expressiveness, and proof translations have, though to a lesser extent, also been significant.

Example 30 (Relating Classical and Intuitionistic Propositional Logic). Prawitz & Malmnäs (1968) consider the relationships between minimal, intuitionistic, and classical propositional logics, and develop several formulae translations in their study of interpretability and interpretability with respect to derivability.

A logical system S_1 is said to be interpretable in a logical system S_2 by a formula translation $\mathcal{F}$ when

$$\vdash_{\mathsf{S}_1} A \quad \text{if and only if} \quad \vdash_{\mathsf{S}_2} \mathcal{F}(A).$$

S_1 is said to be interpretable with respect to derivability by $\mathcal{F}$ just in case

$$\Gamma \vdash_{\mathsf{S}_1} A \quad \text{if and only if} \quad \mathcal{F}(\Gamma) \vdash_{\mathsf{S}_2} \mathcal{F}(A),$$

where $\mathcal{F}(\Gamma)$ is the set $\{\, \mathcal{F}(\gamma) \mid \gamma \in \Gamma \,\}$.

Prawitz & Malmnäs define a translation $\sim\sim$ by letting $A^{\sim\sim}$ be the formula that results from inserting two negation signs before each part (roughly, subformula) of A. They then prove that classical logic is interpretable with respect to derivability in intuitionistic logic and in minimal logic:

$$\Gamma \vdash_{\mathsf{C}} A \quad \text{if and only if} \quad \Gamma^{\sim\sim} \vdash_{\mathsf{I}} A^{\sim\sim} \tag{2.3}$$
$$\Gamma \vdash_{\mathsf{C}} A \quad \text{if and only if} \quad \Gamma^{\sim\sim} \vdash_{\mathsf{M}} A^{\sim\sim} \tag{2.4}$$

We need not reproduce their entire proof here; it suffices to point out that their proof is based on examining the relationships between intuitionistic proofs and classical proofs, just as they did:

> By observing, first, that the rule for eliminating double negation is intuitionistically valid when the conclusion is a negation, and, second, that every intuitionistic inference rule continues to be intuitionistically valid after the $\sim\sim$-translation, one immediately obtains a proof of Theorem A. (Prawitz & Malmnäs, 1968, p. 220)

The approach taken by Prawitz & Malmnäs is rigorous, but *ad hoc* in the sense described in § 1.3.1.

To demonstrate the simplicity of the category theory-based approach, we recast their Theorem A in categorical terms.

First, the derivability relation $\{ B_1, \dots, B_n \} \vdash A$ relates sets of formulae with individual formulae, but when logical systems are represented as categories, there are no sets of formulae, so we must understand the relationship $\vdash A$ to stand for the existence of an arrow $\top \to A$, the relationship $\{ B \} \vdash A$ for the existence of an arrow $B \to A$, and the relationship $\{ B_1, \dots, B_n \} \vdash A$ for the existence of an arrow $B_1 \,\&\, \cdots \,\&\, B_n \to A$.

The definition of interpretability and interpretability with respect to derivability must remain the same, but now (2.3) can be proved in a somewhat more general fashion.

Proof. Let I be the category whose objects are formulae freely generated from propositional variables and the boolean connectives, and whose arrows are generated by schemata for intuitionistically valid inference rules. Let C be the category whose objects are the same as I's and whose arrows are generated by schemata for intuitionistically valid inference rules as well as a schema for double negation elimination. Let I' be the category whose objects are formulae freely generated from the double negation of propositional variables and the boolean connectives, and whose arrows are generated by intuitionistically valid inference rule schemata. Note that every object (arrow) of I' is also an object (arrow) of I.

There are functors $\mathcal{F}\colon \mathsf{C} \to \mathsf{I}'$ and $\mathcal{F}'\colon \mathsf{I}' \to \mathsf{C}$ whose formulae mappings are $\sim\sim$ and the its inverse ($\sim\sim$ is readily verified to be invertible) and whose arrow mappings are the proof translations described by Prawitz & Malmnäs. The existence of these functors show that $\Gamma \vdash_{\mathsf{C}} A$ implies $\mathcal{F}(\Gamma) \vdash_{\mathsf{I}'} \mathcal{F}(A)$ and vice versa. Then C is interpretable in I' by $\mathcal{F}$. □

Slightly more work is needed to show that $\mathcal{F}(\Gamma) \vdash_{\mathsf{I}} \mathcal{F}(A)$ implies $\mathcal{F}(\Gamma) \vdash_{\mathsf{I}'} \mathcal{F}(A)$, but our purpose here has been only to demonstrate that earlier *ad hoc* approaches fit within the categorical framework.

Example 31 (A Deduction Theorem for Propositional Calculus). Lambek & Scott (1988, pp. 51–52) present a deduction theorem for the positive, propositional, intuitionistic calculus using a categorical representation of the calculus; specifically, the category that arises from the deductive system given in Example 20. In traditional treatments, the deduction theorem states that if a sentence ψ is provable using the assumptions $\Gamma \cup \{ \phi \}$, then the conditional $\phi \supset \psi$ is provable using just the assumptions Γ, i.e., that

$$\Gamma \cup \{ \phi \} \vdash \psi \quad \text{implies} \quad \Gamma \vdash \phi \supset \psi.$$

Categorically, the result is that if an arrow $\phi(x)\colon B \to C$ can be demonstrated by positing the existence of an arrow $x\colon \top \to A$ in a category $\mathscr{L}$, then there is a proof of an arrow $f : A \,\&\, B \to C$ which does not depend on x.

Table 2.1: Lambek & Scott (1988) partition the arrows of $\mathscr{L}[x\colon \top \to A]$ into five classes and map each $B \to C$ arrow of $\mathscr{L}[x]$ to an $A \,\&\, B \to C$ arrow of $\mathscr{L}$. The notation $\alpha_{A,B,C}$ used in the last case was introduced in Example 20 (p. 30) in Equation (2.1).

Arrow in $\mathscr{L}[x]$	**Arrow in $\mathscr{L}$**
$k\colon B \to C$ (an arrow in $\mathscr{L}$)	$k\pi'\colon A \,\&\, B \to C$
$x\colon \top \to A$	$\pi\colon A \,\&\, \top \to A$
$\langle \psi(x), \chi(x)\rangle\colon B \to C' \,\&\, C''$	$\langle \kappa(\psi(x)), \kappa(\chi(x))\rangle\colon A \,\&\, B \to C' \,\&\, C''$
$\chi(x) \circ \psi(x)\colon B \to C$	$\kappa(\chi(x)) \circ \langle \pi_{A,B}, \kappa(\psi(x))\rangle\colon A \,\&\, B \to C$
$\psi(x)^*\colon B \to C' \supset C''$	$(\kappa(\psi(x) \circ \alpha_{A,B,C'}))^*\colon A \,\&\, B \to C' \supset C''$

Table 2.2: The arrow mapping for the functor K maps each $B \to C$ arrow in $\mathscr{L}[x]$ to a $A \,\&\, B \to A \,\&\, C$ arrow in $\mathscr{L}$. Since K is a functor, the fourth case, composition, is forced. The first case covers identity arrows, and requires that conjunction is a product in order that $\langle \pi_{A,B}, \mathrm{id}_B \circ \pi'_{A,B}\rangle = \mathrm{id}_{A\&B}$.

Arrow in $\mathscr{L}[x]$	**Arrow in $\mathscr{L}$**
$k\colon B \to C$ (an arrow in $\mathscr{L}$)	$\langle \pi_{A,B}, k \circ \pi'_{A,B}\rangle$
$x\colon \top \to A$	$\langle \mathrm{id}_A, \mathrm{id}_A\rangle \circ \pi'_{\top,A}$
$\langle \psi(x), \chi(x)\rangle\colon B \to C' \,\&\, C''$	$\langle \pi_{A,B}, \langle \pi'_{A,C'} \circ K(\psi), \pi'_{A,C''} \circ K(\chi)\rangle\rangle$
$\chi(x) \circ \psi(x)$	$K(\chi(x)) \circ K(\psi(x))$
$\psi(x)^*\colon B \to C' \supset C''$	$\langle \pi_{A,B}, (\pi'_{A,C''} \circ K(\psi(x)) \circ \alpha_{A,B,C'})\rangle$

More formally, let $\mathscr{L}$ be a category arising from the deductive system given in Example 20 (p. 30) (i.e., whose arrows are generated by those schemata, and for which the appropriate equations on arrows hold) for some given set of propositional variables. Then for a propositional variable A, let $\mathscr{L}[x\colon \top \to A]$ (or just $\mathscr{L}[x]$) be the category like $\mathscr{L}$ but with an additional arrow and the additional arrows arising from it through the arrow schemata. The categorical version of the deduction theorem says, then, that for every arrow $\phi(x)\colon B \to C$ in $\mathscr{L}[x\colon \top \to A]$, there is an arrow $f: A \,\&\, B \to C$ in $\mathscr{L}$. Lambek & Scott prove this directly using the case analysis shown in Table 2.1.

While the arrow mapping κ given in Table 2.1 maps each arrow of $\mathscr{L}[x]$ to an arrow of $\mathscr{L}$, κ does not determine a functor. Indeed, in the fourth case κ maps $\chi(x) \circ \psi(x)\colon B \to C$ to $\kappa(\chi(x)) \circ \langle \pi_{A,B}, \kappa(\psi(x))\rangle$, but a functor would respect composition and map to $\kappa(\chi(x)) \circ \kappa(\psi(x))$. Another way to perceive that κ does not determine a functor is to note that κ does not map identity arrows to identity arrows since κ maps every $B \to B$ arrow in $\mathscr{L}[x]$ to an $A \,\&\, B \to B$ arrow in $\mathscr{L}$.

However, we can define a formula and arrow mapping K similar to κ which is, in fact, a $\mathscr{L}[x] \to \mathscr{L}$ functor. First, K takes each formula B of $\mathscr{L}[x]$ to $A \,\&\, B$. Before defining the arrow mapping, we note that K will take each arrow $\phi(x)\colon B \to C$ of $\mathscr{L}[x]$ to an arrow $K(\phi(x))\colon A\&B \to A \,\&\, C$. By virtue of the arrow $\pi'\colon A \,\&\, C \to C$, we have that $\pi' \circ K(\phi(x))\colon A \,\&\, B \to C$, and thus that using the functor K we can identify an $A \,\&\, B \to C$ arrow in $\mathscr{L}$ for each $B \to C$ arrow in $\mathscr{L}[x]$. K's arrow mapping is given in Table 2.2.

Given the existence of a functor from $\mathscr{L}[x]$ to $\mathscr{L}$ with an object mapping that takes ϕ to $A \,\&\, \phi$, the deduction theorem is a simple corollary: For any $\mathscr{L}[x]$ arrow $f : B \to C$, there is an arrow $\pi'_{A,C} \circ K(f) : A \& B \to C$ in $\mathscr{L}$.

2.2.2.4 Relationships between Functors

There are many relationships that can hold between formula translations. For instance, one translation may preserve more information than another, or one might be more general than another. Amongst proof translations, too, there are interesting relationships. For instance, one proof translation may preserve more proof structure than another. Functors are the categorical version of formula and proof translations, but it is natural transformations that capture the relationships that hold between functors.

Definition 32 (Natural Transformation)**.** A natural transformation $\eta : F \to G$ between two $\mathscr{C} \to \mathscr{D}$ functors is a collection of $\mathscr{D}$ arrows indexed by $\mathscr{C}$ elements such that the following diagram commutes.

$$\begin{tikzcd}
F(A) \arrow[r, "\eta_A"] \arrow[d, "F(f)"'] & G(A) \arrow[d, "G(f)"] \\
F(B) \arrow[r, "\eta_B"'] & G(B)
\end{tikzcd}$$

While the definition of natural transformations is short, it is worth examining in closer detail. The objects $F(A)$, $F(B)$, $G(A)$, and $G(B)$ in the commutative diagram are all objects in the category $\mathscr{D}$ where F and G are $\mathscr{C} \to \mathscr{D}$ functors. Similarly, both $F(f)$ and $G(f)$ are $\mathscr{D}$ arrows. Finally, the arrows η_A and η_B, then, must also be $\mathscr{D}$ arrows. A natural transformation, then, is a collection of $\mathscr{D}$ arrows that relate the image of $\mathscr{C}$ objects and arrows under $\mathscr{C} \to \mathscr{D}$ functors. In categories for logics, all these objects are $\mathscr{D}$ formulae and these arrows are $\mathscr{D}$ proofs. A natural transformation for logic categories, then, is a collection of $\mathscr{D}$ proofs indexed by $\mathscr{C}$ formulae.

Example 33 (Weakening). Classical logics, and many non-classical logics, are monotone: the addition of additional premises does not invalidate any conclusions drawn from previous accepted premises. (There are logics which do not have this property; defeasible logics are an important example in which arguments can be defeated by competing arguments (Pollock, 1987, 1992). More generally, such logics, including default logics and abductive logics, are called non-monotonic.)

In our treatment of logics as categories, we might express the monotonicity of a logic as the existence of a particular class of natural transformations. We consider two functors. The first is the identity functor $\mathrm{id}_{\mathscr{C}} : \mathscr{C} \to \mathscr{C}$ which maps each $\mathscr{C}$ object and arrow to itself. That $\mathrm{id}_{\mathscr{C}}$ preserves identity arrows and compositions is a trivial observation. The second is the functor $A \times - : \mathscr{C} \to \mathscr{C}$ which maps each object X to the product $A \times X$, and each arrow $f : X \to Y$ to the arrow $\mathrm{id}_A \times f : A \times X \to A \times Y$. Recall that $\mathrm{id}_A \times f$

is shorthand for the product arrow $\langle \mathrm{id}_A\pi, f\pi'\rangle$. We should verify that $A\times -$ preserves composites and identities. Identities are preserved:

$$\begin{aligned} A\times -(\mathrm{id}_X) &= \mathrm{id}_A \times \mathrm{id}_X \\ &= \langle \mathrm{id}_A\pi_{A,X}, \mathrm{id}_X\pi'_{A,X}\rangle \\ &= \langle \pi_{A,X}, \pi'_{A,X}\rangle \\ &= \mathrm{id}_{A\times X} \\ &= \mathrm{id}_{A\times -(X)} \end{aligned}$$

Composites are preserved:

$$\begin{aligned} F(gf) &= \mathrm{id}_A \times gf \\ &= \mathrm{id}_A\mathrm{id}_A \times gf \\ &= (\mathrm{id}_A \times g)\circ(\mathrm{id}_A \times f) \\ &= (A\times -(g))\circ(A\times -(f)) \end{aligned}$$

Now we may observe that the following diagram commutes:

$$\begin{tikzcd} A\times X \arrow[r, "\pi'_{A,X}"] \arrow[d, "\mathrm{id}_A\times f"'] & X \arrow[d, "f"] \\ A\times Y \arrow[r, "\pi'_{A,Y}"'] & Y \end{tikzcd}$$

This is precisely the required diagram to show that $\eta_X = \pi'_{A,X}$ are the components of a natural transformation $A\times - \to \mathrm{id}_{\mathscr{C}}$. Then we may say that if, for every $\mathscr{C}$ object A there is a natural transformation $\eta\colon A\times - \to \mathrm{id}_{\mathscr{C}}$ with component $\pi'_{A,X}$ at X, then $\mathscr{C}$ is monotonic.

This example is not complex, since the functors and natural transformations involved are relatively simple; but it illustrates the way in which simple categorical constructions capture significant properties of logical systems. Indeed, we observe that any logical system represented by a category where the addition of an assumption is captured by a categorical product must be monotonic.

2.2.3 Denotational Proof Languages

Arkoudas (2000) developed and introduced denotational proof languages (DPLs) as a family of languages for proof construction and computation (for proof search) with a well-defined denotational semantics. Programs in denotational proof languages are evaluated with respect to an assumption base to produce a value, their denotation. Denotational proof languages are united by a common underlying formalism, the $\lambda\mu$-calculus, an extension of the familiar λ-calculus. In practice, denotational proof languages provide simple, concise, and elegant proof-construction environments.

Before delving into the technical foundations upon which denotational proof languages are built, we first consider a small proof in a simple DPL.

Example 34 (Syntax of Classical Natural Deduction, $\mathcal{CND}$). A proof in a DPL for classical natural deduction, $\mathcal{CND}$, of $P \,\&\, Q \supset Q \,\&\, P$:

```
assume P & Q in
  begin
    right-and P & Q;
    left-and P & Q;
    both Q, P
  end
```

We begin by considering the body within the top level **assume** $P \,\&\, Q$ **in** … form, that is, the **begin** … **end** block. The **begin** … **end** pair serves to group its contents as a single deduction. The three inner deductions, delimited by the composition operator ";", are evaluated in sequence. Each deduction in the proof is evaluated with respect to an assumption base. The first deduction, **right-and** $P \,\&\, Q$ is evaluated and, if the proposition $P \,\&\, Q$ is in the current assumption base, produces the right conjunct, Q. The second deduction, **left-and** $P \,\&\, Q$, when evaluated in an assumption base containing $P \,\&\, Q$, produces the left conjunct, P. The third deduction, **both** Q, P, when evaluated in an assumption base containing Q and P, produces the conjunction $Q \,\&\, P$. The composition operator between two deductions has the effect that after the first is evaluated with respect to a particular assumption base, β, to produce some result, ϕ, the second deduction is evaluated with respect to β extended with ϕ, i.e., $\beta \cup \{\phi\}$.

Let us consider the evaluation of the entire **begin** … **end** deduction in an assumption base containing the proposition $P \,\&\, Q$, say $\beta \cup \{P \,\&\, Q\}$. The evaluation of **right-and** $P \,\&\, Q$ with respect to $\beta \cup \{P \,\&\, Q\}$ produces Q. Due to the composition, the next deduction, **left-and** $P \,\&\, Q$, is evaluated in the assumption base $\beta \cup \{P \,\&\, Q, P\}$ and produces the proposition Q. The third deduction, **both** Q, P, is evaluated in the assumption base which incorporates this new result, i.e., $\beta \cup \{P \,\&\, Q, Q, P\}$, and so produces the conjunction $Q \,\&\, P$. Then the denotation of the **begin** … **end** deduction, when evaluated in an assumption base containing $P \,\&\, Q$, is the proposition $Q \,\&\, P$.

The evaluation semantics of **assume** $P \,\&\, Q$ **in** … is only slightly more complicated. In general, the evaluation of **assume** ϕ in D in an assumption base β is the proposition $\phi \supset \psi$ when the evaluation of D in $\beta \cup \{\phi\}$ produces ψ. Thus, in the present case, the evaluation of the **assume** … form in an assumption base β will evaluate the **begin** … **end** block in $\beta \cup \{P \,\&\, Q\}$ which will produce $Q \,\&\, P$, and the final result is the conditional $P \,\&\, Q \supset Q \,\&\, P$.

The $\lambda\mu$-calculus provides a common foundation for denotational proof languages, and is an extension of the λ-calculus, which is often used as a foundation for computation (and thus, programming languages). The $\lambda\mu$-calculus extends the λ-calculus by adding a syntactic category for deductions, as well as an additional type of expression, that of methods, i.e., abstractions over deductions. The syntax of the $\lambda\mu$-calculus is straightforward. The syntax also includes provisions for special deductive forms.

Remark 5 (Concerning other $\lambda\mu$-calculi). Another extension of the λ-calculus called the $\lambda\mu$-calculus was introduced earlier in an article by Parigot (1992), *$\lambda\mu$-calculus: an algorithmic interpretation of classical natural deduction*. Both are related to deduction and extend the λ-calculus, but aside from these similarities, they are unrelated, distinct formalisms. In later publications, Arkoudas uses the name $\lambda\phi$-calculus to avoid confusion (Arkoudas, 2001a, p. 1, footnote 1).

Definition 35 ($\lambda\mu$-calculus syntax)**.** In the following, D ranges over *Ded*, the class of deductions, E over *Exp*, the class of expressions, and M and N over *Phr*, the class of phrases (a phrase is a deduction or an expression, i.e., $\mathit{Phr} = \mathit{Ded} \cup \mathit{Exp}$).

$$\begin{aligned} E &::= c \mid I \mid \mu\,\overrightarrow{I}.D \mid \lambda\,\overrightarrow{I}.E \mid \mathbf{app}(E, \overrightarrow{M}) \\ D &::= \mathbf{dapp}(E, \overrightarrow{M})\,\{ \mid kwd_1(\overrightarrow{\Xi_1}) \mid \cdots \mid kwd_n(\overrightarrow{\Xi_n})\,\} \\ M &::= E \mid D \end{aligned}$$

Syntactically, the $\lambda\mu$-calculus is very similar to the pure λ-calculus. Abstractions over deductions, called methods, are formed using μ in the same way that functions are abstracted from expressions using λ. Function application is written as $\mathbf{app}(E, \overrightarrow{M})$ as opposed to the more conventional $(E\ \overrightarrow{M})$, but this notation is a better parallel with method application which is written as $\mathbf{dapp}(E, \overrightarrow{M})$.

The semantics of the $\lambda\mu$-calculus is denotational whereas the usual semantics for the λ-calculus is given by a number of reduction rules. In the denotational semantics, a phrase M is evaluated with respect to an assumption base β to produce a value v. That M produces v when evaluated "in" or "under" the assumption base β is denoted by

$$\beta \vdash M \leadsto v$$

The semantics for the (pure) $\lambda\mu$-calculus are given by the rules presented in Figure 2.2. Most of the rules specify behavior similar to the λ-calculus, but several rules (viz.: R7, R10, R11, and R12) provide semantics particularly important for proof languages. To specify a useful system with the $\lambda\mu$-calculus, it remains to specify a set of constants which is partitioned into a set of primitive methods, a set of primitive functions, and a set of primitive values. A subset of the primitive values are designated as sentences, and

$$\frac{}{\beta \vdash \mathbf{app}(\lambda \overrightarrow{I}.E, \overrightarrow{M}) \leadsto E[\overrightarrow{M}/\overrightarrow{I}]}\ [\text{R1}] \qquad \frac{}{\beta \vdash \mathbf{dapp}(\mu \overrightarrow{I}.D, \overrightarrow{M}) \leadsto D[\overrightarrow{M}/\overrightarrow{I}]}\ [\text{R2}]$$

$$\frac{\beta \vdash E \leadsto E'}{\beta \vdash \mathbf{app}(E, \overrightarrow{M}) \leadsto \mathbf{app}(E', \overrightarrow{M})}\ [\text{R3}] \qquad \frac{\beta \vdash E \leadsto E'}{\beta \vdash \mathbf{dapp}(E, \overrightarrow{M}) \leadsto \mathbf{dapp}(E', \overrightarrow{M})}\ [\text{R4}]$$

$$\frac{\beta \vdash M_i \leadsto M_i'}{\beta \vdash \mathbf{app}(E, M_1, \ldots, M_i, \ldots, M_k) \leadsto \mathbf{app}(E, M_1, \ldots, M_i', \ldots, M_k)}\ [\text{R5}]$$

$$\frac{\beta \vdash E_i \leadsto E_i'}{\beta \vdash \mathbf{dapp}(E, M_1, \ldots, E_i, \ldots, M_k) \leadsto \mathbf{dapp}(E, M_1, \ldots, E_i', \ldots, M_k)}\ [\text{R6}]$$

$$\frac{\beta \vdash D_i \leadsto S \quad \beta \cup \{S\} \vdash \mathbf{dapp}(E, M_1, \ldots, S, \ldots, M_k) \leadsto N}{\beta \vdash \mathbf{dapp}(E, M_1, \ldots, D_i, \ldots, M_k) \leadsto N}\ [\text{R7}]$$

$$\frac{\beta \vdash E \leadsto N}{\beta \vdash \lambda \overrightarrow{I}.E \leadsto \lambda \overrightarrow{I}.N}\ [\text{R8}] \qquad \frac{\beta \vdash D \leadsto N}{\beta \vdash \mu \overrightarrow{I}.E \leadsto \mu \overrightarrow{I}.N}\ [\text{R9}]$$

$$\frac{\beta \vdash D \leadsto S}{\beta \cup \beta' \vdash D \leadsto S}\ [\text{R10}] \qquad \frac{\beta \vdash M_1 \leadsto M_2 \quad \beta \vdash M_2 \leadsto M_3}{\beta \vdash M_1 \leadsto M_3}\ [\text{R11}]$$

$$\frac{}{\{S\} \vdash \mathbf{dapp}(\mathsf{claim}, S) \leadsto S}\ [\text{R12}]$$

Figure 2.2: Rules specifying the semantics of the $\lambda\mu$-calculus. Most rules ensure that λ and μ abstractions behave in the "usual" way, e.g., R1–R6. R7 is special in that it ensure that deduction applications are executed in an assumption base that has been extended with the results of any deductive arguments. R12 specifies the semantics of the primitive method claim.

assumption bases are sets of sentences. The semantics of primitive methods must be specified on a case by case basis, and while there are certain constraints placed on primitive methods, we need not concern ourselves with them here.

Given the general semantics of the $\lambda\mu$-calculus, we can now examine the specification of a particular denotational proof language as a $\lambda\mu$-system. We shall examine the DPL introduced in Example 34, a natural deduction proof system for classical propositional calculus.

Example 36 (Semantics of Classical Natural Deduction, $\mathcal{CND}$). A $\lambda\mu$-system is specified by fixing a set of primitive methods, primitive functions, and the primitive values (including sentences) of that system. The primitive values of $\mathcal{CND}$ are the sentences built up from propositional variables and boolean connectives in the usual way. $\mathcal{CND}$ has no primitive functions. $\mathcal{CND}$ has one special deductive syntax, **assume**, and a number of primitive methods. The semantics of **assume** and the primitive methods are given in Figure 2.3. (Note that we are really presenting two versions of $\mathcal{CND}$. The first, given in Example 34, is to be seen as syntactic sugar for the second, given here. Alternatively, one might view the latter as a sort of assembly to which the former is compiled.)

$$\beta \cup \{ P \supset Q, P \} \vdash \textbf{dapp}(\text{modus-ponens}, P \supset Q, P) \rightsquigarrow Q$$
$$\beta \cup \{ P \supset Q, \sim Q \} \vdash \textbf{dapp}(\text{modus-tollens}, P \supset Q) \rightsquigarrow \sim Q \sim P$$
$$\beta \cup \{ \sim\sim P \} \vdash \textbf{dapp}(\text{double-negation}, \sim\sim P) \rightsquigarrow P$$
$$\beta \cup \{ P_1, P_2 \} \vdash \textbf{dapp}(\text{both}, P_1, P_2) \rightsquigarrow P_1 \,\&\, P_2$$
$$\beta \cup \{ P_1 \,\&\, P_2 \} \vdash \textbf{dapp}(\text{left-and}, P_1 \,\&\, P_2) \rightsquigarrow P_1$$
$$\beta \cup \{ P_1 \,\&\, P_2 \} \vdash \textbf{dapp}(\text{right-and}, P_1 \,\&\, P_2) \rightsquigarrow P_2$$
$$\beta \cup \{ P_1 \} \vdash \textbf{dapp}(\text{left-either}, P_1, P_2) \rightsquigarrow P_1 \vee P_2$$
$$\beta \cup \{ P_2 \} \vdash \textbf{dapp}(\text{right-either}, P_1, P_2) \rightsquigarrow P_1 \vee P_2$$
$$\beta \cup \{ P_1 \vee P_2, P_1 \supset Q, P_2 \supset Q \} \vdash \textbf{dapp}(\text{cases}, P_1 \vee P_2, P_1 \supset Q, P_2 \supset Q) \rightsquigarrow Q$$
$$\beta \cup \{ P_1 \supset P_2, P_2 \supset P_1 \} \vdash \textbf{dapp}(\text{equivalence}, P_1 \supset P_2, P_2 \supset P_1) \rightsquigarrow P_1 \longleftrightarrow P_2$$
$$\beta \cup \{ P_1 \longleftrightarrow P_2 \} \vdash \textbf{dapp}(\text{left-iff}, P_1 \longleftrightarrow P_2) \rightsquigarrow P_1 \supset P_2$$
$$\beta \cup \{ P_1 \longleftrightarrow P_2 \} \vdash \textbf{dapp}(\text{right-iff}, P_1 \longleftrightarrow P_2) \rightsquigarrow P_2 \supset P_1$$
$$\beta \cup \{ P, \sim P \} \vdash \textbf{dapp}(\text{absurd}, P, \sim P) \rightsquigarrow \textbf{false}$$
$$\frac{\beta \cup \{ P \} \vdash D \rightsquigarrow Q}{\beta \vdash \textbf{assume}(P, D) \rightsquigarrow Q}$$

Figure 2.3: The semantics of $\mathcal{CND}$'s primitive methods and special deductive form **assume**.

Example 37 (Athena, Integrating Computation with DPLs). One of the stated benefits of using DPLs was the ability to integrate computation with deduction. The $\mathcal{CND}$ language has no primitive functions defined for computation, and so cannot demonstrate these features, so we turn our attention to Athena (Arkoudas, 2005a). Athena is a DPL for many-sorted first-order logic, and has a Lisp-like syntax. Most of Athena's primitive methods are shared with $\mathcal{CND}$, and we need not cover all the details of Athena here.

First, we consider a straightforward Athena proof of the conditional $(A \,\&\, (B \,\&\, C)) \supset (C \,\&\, A)$. The proof uses the **left-and**, **right-and**, and **both** rules that we have already described, as well as an **assume** form. In Athena, **seq** takes the place of **begin** … **end** pairs.

```
(assume (and A (and B C))
  (seq
    (!left-and (and A (and B C)))
    (!right-and (and A (and B C)))
    (!right-and (and B C))
    (!both C A)))
=>
(if (and A (and B C))
    (and C A))
```

In this proof, each reasoning step is explicit, and there are no extra steps taken. The use of **seq** to sequence proof steps makes the proof more readable, but it also increases the scope in which intermediate derivations can be cited. A more compact version would be:

```
(assume (and A (and B C))
  (!both (!right-and (!right-and (and A (and B C))))
         (!left-and (and A (and B C)))))
=>
(if (and A (and B C))
    (and C A))
```

A more generally applicable proof strategy that could have been used to prove this same result would use the method **decompose**. **decompose** takes as arguments a formula and a method. Using **left-and** and **right-and** recursively, **decompose** infers all the conjunctive subformulae of its first argument and then applies the method in the resulting assumption base.

The proof using the **decompose** method might be implemented as the following Athena proof.

```
(assume (and A (and B C))
  (!decompose (and A (and B C))
              (method ()
                (!both C A))))
=>
(if (and A (and B C))
    (and C A))
```

The behavior of **decompose** is as follows. When **decompose** is called with arguments f, a formula, and m, a method of no arguments, it first examines f. If f is not a conjunction, then **decompose** simply invokes m with no parameters. Otherwise, f should be in the assumption base, and **left-and** is used to extract its left conjunct, l. **decompose** is then called recursively with l and a new method that will (eventually) infer the f's right conjunct and invoke m. The method **decompose** can be implemented simply:

```
(define (decompose formula M)
  (dmatch formula
    ((and P Q)
     (!decompose (!left-and formula)
                 (method ()
                   (!decompose (!right-and formula)
                               M))))
    (_ (!M))))
```

In this example, this has the effect of evaluating the method $\mu()$.**dapp**(both, C, A) in an assumption base containing $A \,\&\, (B \,\&\, C)$, A, $B \,\&\, C$, B, and C. Since the assumption base contains C and A, the method produces the sentence $C \,\&\, A$, and the entire proof produces $(A \,\&\, (B \,\&\, C)) \supset (C \,\&\, A)$.

It is worth noting that both of the proofs given in the preceding example produce the same formula under all assumption bases. Yet their text is quite different, and their observational equivalence is not apparent unless one is familiar with the definition of **decompose**. An even more significant difference is that the second proof performs some “extra” work. This is not necessarily bad, in and of itself (indeed, any proof-search automation is bound to perform some unnecessary work), but it is a significant difference, and the DPL framework makes no provision for determining which work was actually useful.

2.3 Summary

We have reviewed a number of approaches to representing logical systems, including both traditional approaches oriented mainly toward the construction of proofs and deductions (axiomatic and natural-deduction proof systems) as well as approaches that attempt to generalize the idea of logical systems and make them an object of study unto themselves (category-theory-based approaches). We have also seen an approach (denotational proof languages) that uses a powerful formalism to generalize the notions of proof language and computation (the $\lambda\mu$-calculus) while still providing a convenient notation for manually constructing proofs and automated deductive procedures. In the sequel, we shall focus primarily on these latter two formal systems (the category theoretic treatment of logics, and denotational proof languages) as we attempt to combine the benefits of the category-theoretic approach (a rigorous and principled way of specifying logical systems and the interactions between them) and of denotational proof languages (convenient proof construction and support for proof automation), to order to achieve the goals of our problem statement.

Chapter 3

Results

In this chapter we describe and define a family of categorical DPLs, discuss their implementation, and demonstrate their use, including the ways in which interoperability between logics occurs. It is precisely when logical systems can be implemented and connected in this way that we call them *fluid logics*. In this initial investigation, we implement some of the structures used in logics that are available in the Slate (Bringsjord et al., 2007, 2008) courseware. We also implement and demonstrate two fluid logics and mappings between them, namely the mapping from a logical system with an additional hypothesis to the logical system without (this is a form of the deduction theorem), and the mapping from a natural-deduction proof system to an axiomatic system.

3.1 Overview

We originally envisioned that the implementation of fluid logics via a framework for categorical denotational proof languages would encompass a significant variation or departure from the $\lambda\mu$-calculus that underlies traditional denotational proof languages as presented by Arkoudas (2000, chapter 8). After some initial experimentation, however, it became clear that a traditional denotational proof language in which the "propositions" were the arrows of a category, along with a powerful macro system, provides the desired features of a true "categorical denotational proof language." As such, we implemented a $\lambda\mu$-calculus interpreter in the style of Scheme (Sussman & Steele, 1998; Kelsey et al., 1998). The resulting language is very similar to Arkoudas's Athena (Arkoudas, 2005a), but provides a `define-macro` similar to Common Lisp's `defmacro`, allowing many of the forms that Arkoudas described as syntactic sugar, including advanced features such as pattern matching, to be implemented directly in the language as macros. To support interoperability with Java, we also added support for the JScheme's JavaDot notation (Anderson et al., 2001, § Java Access).

3.2 Implementing the $\lambda\mu$-Calculus

We implemented the $\lambda\mu$-calculus using Java as a host language. A detailed description of the language syntax and semantics is given in Appendix A, but readers having some familiarity with Athena, Scheme, or Common Lisp, will probably understand most of the following sections. The implemented system is not a unique DPL, but rather the underlying framework in which DPLs can be constructed. While our primary concern is with categorical DPLs for fluid logics, we can just as easily implement traditional DPLs. We continue by implementing three traditional DPLs, and then moving on to categorical DPLs.

3.3 Implementing Denotational Proof Languages

The current work is aimed at implementing categorical denotational proof languages, but the programming language at hand is suitable for implementing standard denotational proof languages as well. As such, we begin with three examples of three standard denotational proof languages. The first is for a simple formal system introduced by Hofstadter (1979). The second and third are natural deduction and axiomatic presentations of the propositional calculus.

Example 38 (Hofstadter's MIU-system). In his modern classic, *Gödel, Escher, Bach: an Eternal Golden Braid*, Hofstadter (1979) introduces his readers to axiomatic proof systems though the simple MIU-system. Strings of the system are generated from the three-letter alphabet, **M**, **I**, and **U**. Viewed as a proof system, there is one axiom, **MI**, and there are four inference rules. Intuitively, these are as follows. By rule 1, one may append a **U** to a string ending in **I**. By rule 2, one may append Γ to a string $\mathbf{M}\Gamma$. By rule 3, any occurrence of **III** in a string may be replaced by **U**. By rule 4, any occurrence of **UU** may be dropped from a string. Formally, where Γ and Λ stand for any strings of the MIU-system, (including the empty string),

$$\frac{}{\mathbf{MI}}\,A_1 \qquad \frac{\Gamma\,\mathbf{I}}{\Gamma\,\mathbf{IU}}\,R_1 \qquad \frac{\mathbf{M}\,\Gamma}{\mathbf{M}\,\Gamma\,\Gamma}\,R_2 \qquad \frac{\Gamma\,\mathbf{III}\,\Gamma}{\Gamma\,\mathbf{U}\,\Lambda}\,R_3 \qquad \frac{\Gamma\,\mathbf{UU}\,\Lambda}{\Gamma\,\Lambda}\,R_4$$

The MIU-system can be implemented as a denotational proof language, adopting Java's `String`s as propositions. Checking the preconditions of the rules is a simple matter of string manipulation. The axiom A_1 and the rules R_1 and R_2 can be implemented as primitive methods straightforwardly. The method `a1` simply returns the string **MI**. The method `r1` takes a string argument and returns the string concatenated with **U** if the string is in the assumption base and ends with **I**, or throws an appropriate exception otherwise. Method `r2` takes a string argument and returns the string concatenated with the string's suffix after the initial character if the string is in the assumption base and begins with **M**, and throws an appropriate exception otherwise.

```
(define-primitive-method (a1)
    ;; /MI
    "MI")

(define-primitive-method (r1 string)
    ;; xI/xIU
    (check
     ((~ (.contains (ab) string))
      (error string " is not in the assumption base."))
     ((~ (.endsWith string "I"))
      (error  string " does not end with I."))
     (else
      (.concat string "U"))))

(define-primitive-method (r2 string)
    ;; Mx/Mxx
    (check
     ((~ (.contains (ab) string))
      (error string " is not in the assumption base."))
     ((~ (.startsWith string "M"))
      (error string " does not begin with M."))
     (else
      (.concat string (.substring string 1)))))
```

In the final two methods we must make an aesthetic decision. For any given string, there may be multiple positions to which R_3 and R_4 could be applied. That is, there may be multiple occurrences of the substring **III** that R_3 could replace with **U**, or multiple occurrences of the substring **UU** that R_4 could replace. One approach to implementing these methods has the methods take both the premise and the desired conclusion as arguments. The methods then check whether the latter is a valid conclusion of the rule when applied to the former, and that the former is, in fact, in the assumption base. As a matter of taste and programming style, we prefer an alternative wherein the user is not required to specify (or in some cases, even *know*) the conclusion in advance. As such, we implement R_3 and R_4 as methods that take an index and a string, and check whether **III** or **UU** appear in the string at the specified index, respectively.

```
(define-primitive-method (r3 position string)
    ;; xIIIy/xUy
    (check
     ((~ (.contains (ab) string))
      (error string " is not in the assumption base."))
     ((~ (equals "III" (.substring string position (+ 3 position))))
      (error string " does not contain III at position " position "."))
     (else
      (.concat (.substring string 0 position)
               (.concat "U" (.substring string (+ 3 position)))))))

(define-primitive-method (r4 position string)
    ;; xUUy/xy
    (check
     ((~ (.contains (ab) string))
      (error string " is not in the assumption base."))
     ((~ (equals "UU" (.substring string position (+ 2 position))))
      (error string " does not contain UU at position " position "."))
     (else
47          (.concat (.substring string 0 position)
48                   (.substring string (+ 2 position))))))
```

In fewer than fifty lines we have implemented Hofstadter's MIU-system! The formal system is so easily implemented because it is rather simple, and because we were able to leverage Java's built in string type. In fact, almost all the Java interoperability in this example was used for interacting with Java's strings, which provide `endsWith`, `concat`, `startsWith`, and so on. The other use of Java interoperability is in `(.contains (ab) string)`. The function `ab` returns the current assumption base. The assumption base is a `java.lang.Collection`, and so implements the `contains` method. (This, in turn, imposes the restriction that the objects we use as "propositions" must implement `equals` in a meaningful way. Java's string class implements `.equals` for string equality. In our logical formulae presented later, we intern formulae so that Java's default test of object identity is sufficient.)

Hofstadter presents the following derivation of the string **MUIIU**, which we will recreate in two different ways. First. Hofstadter's derivation:

(1)	**MI**	axiom
(2)	**MII**	from (1) by rule II
(3)	**MIIII**	from (2) by rule III
(4)	**MIIIIU**	from (3) by rule II
(5)	**MUIU**	from (4) by rule III
(6)	**MUIUUIU**	from (5) by rule II
(7)	**MUIIU**	from (6) by rule IV

A minimal translation of this proof calls `a1` to derive **MI**, passes the result to `r2`, and so on, all in line 54, ultimately deriving **MUIIU**:

```
52  (define (example-a)
53      ;; β ⊢ !example-a ⇝ MUIIU
54      (!r4 3 (!r2 (!r3 1 (!r1 (!r2 (!r2 (!a1))))))))
```

Recall that the semantics of **dapp** ([R7] from Figure 2.2 (p. 42)) is such that the evaluation of a method body occurs in an assumption base augmented with any arguments produced by deductions:

$$\frac{\beta \vdash D_i \rightsquigarrow S \quad \beta \cup \{S\} \vdash \mathbf{dapp}(E, M_1, \ldots, S, \ldots, M_k) \rightsquigarrow N}{\beta \vdash \mathbf{dapp}(E, M_1, \ldots, D_i, \ldots, M_k) \rightsquigarrow N} \text{ [R7]}$$

When `example-a` is called under assumption base β, `a1` is called with β and produces **MI**. The rightmost application of `r2` is called with **MI** in the assumption base $\beta \cup \{\textbf{MI}\}$, and produces **MII**. Note that the second application of `r2` is not evaluated under the assumption base $\beta \cup \{\textbf{MI}, \textbf{MII}\}$, but rather $\beta \cup \{\textbf{MII}\}$. This is a result of the semantics of the $\lambda\mu$-calculus rule [R7].

It is possible, however, to sequence the deductions in such a way that at each method application, the results of all the previous deductions are available in the assumption base. This can be achieved manually through the careful construction and application of μ-expressions, but also through the more convenient `dlet*` form, as in `example-b`:

```
58 (define (example-b)
59     ;; β ⊢ !example-b ⇝ MUIIU
60     (dlet* ((mi (!a1))
61             (mii (!r2 mi))
62             (miiii (!r2 mii))
63             (miiiiu (!r1 miiii))
64             (muiu (!r3 1 miiiiu))
65             (muiuuiu (!r2 muiu)))
66            (!r4 3 muiuuiu)))
```

When `example-b` is called in an assumption base β, `a1` is called on line 60 in β and produces **MI**. The call to `r2` on line 61 occurs in the assumption base $\beta \cup \mathbf{MI}$ and produces **MII**. It is in line 62 that the behavior of `example-a` and `example-b` differ. In line 62, `r2` is called in the assumption base $\beta \cup \{\mathbf{MI}, \mathbf{MII}\}$. The utility of constructions such as `dlet*` is in implementing higher-order methods that evaluate deductions to "extend" an assumption base and then call methods under the extended assumption base.

Example 39 (Athena's Propositional Calculus). As an interesting example, we can reproduce the fragment of the DPL used by Athena for the propositional calculus. We define the actual sentences of the propositional calculus in Java as data structures. The definition is fairly routine, and is given in Section E.1 (p. 113). The same appendix includes the definitions in the standard language for interfacing with the Java structures, as well as extensions to the pattern matching system that allows convenient access to the propositional structures.

All DPLs have a claim method that produces a proposition just in case it is already in the assumption base. Its possible implementations are trivial, but we include our definition of `claim` here.

```
(define (ab-check p)
    ;; Returns p if p is in the assumption base,
    ;; and throws an error otherwise
    (check
     ((.contains (ab) p) p)
     (else (error p " is not in the assumption base"))))

(define-primitive-method (claim p)
    ;; Derives p if p is in the assumption base.  Every DPL
    ;; has a claim method.
    (ab-check p))
```

The more interesting part of this example, however, is in the implementation of the primitive methods that make up Athena's natural-deduction system for the propositional calculus. These are similar enough to the methods of $\mathcal{CND}$ shown in Figure 2.3 (p. 43) that we need not discuss most of them, except to note a common implementation pattern. Most of the primitive methods depend only on checking that a number of sentences are already present in the assumption base, and then simply returning an easily constructed sentence based on the method arguments.

```
(define-primitive-method (true-intro)
    ;; /⊤
    TRUE)

(define-primitive-method (absurd p np)
    ;; p,∼p/⊥
    (match (list (ab-check p) (ab-check np))
     ((list x (not x)) FALSE)
     (_ (error "absurd cannot be applied to " p " and " np))))

(define-primitive-method (left-and formula)
    ;; p&q/p
    (match (ab-check formula)
      ((and p q) p)
      (_ (error formula " is not a conjunction"))))

(define-primitive-method (right-and formula)
    ;; p&q/q
    (match (ab-check formula)
      ((and p q) q)
      (_ (error formula " is not a conjunction"))))

(define-primitive-method (both p q)
    ;; p,q/p&q
    (and (ab-check p) (ab-check q)))

(define-primitive-method (left-or p q)
    ;; p/p∨q
    (or (ab-check p) q))

(define-primitive-method (right-or p q)
    ;; q/p∨q
    (or p (ab-check q)))

(define-primitive-method (cd or-p-q if-p-r if-q-r)
    ;; p∨q,p⊃r,q⊃r/r
    (match (list (ab-check or-p-q)
                 (ab-check if-p-r)
                 (ab-check if-q-r))
     ((list (or p q) (if p r) (if q r)) r)
     (_ (error "malformed application of cd to " or-p-q ", "
               if-p-r ", and " if-q-r))))

(define-primitive-method (mp if-p-q p)
    ;; p⊃q,p/q
    (match (list (ab-check if-p-q) (ab-check p))
     ((list (if p q) p) q)
     (_ (error "malformed application of mp to " if-p-q
               " and " p))))

;; Implementing assume is a bit tricky, because there is no sound way
;; to unconditionally add something to an assumption base.  We define
;; a primitive method %assume using another primitive method
;; %%force that unconditionally derives its argument.  %assume
;; and %%force are declared within a lexical scope introducing
;; a shared secret %%key, so that %%force cannot be used anywhere
;; but in %assume.

(let ((%%key (gensym))) ; begin scope of %%key

  (define-primitive-method (%%force p key)
      (check
       ((== key %%key) p)
       (else (error "%%force cannot be used"))))

  (define-primitive-method (%assume p d)
      ;; %assume returns the conditional that has the antecedent p,
      ;; and the consequent that is the result of invoking d in an
      ;; assumption base that includes p
      (if p (!(mu (_) (!d))
              (!%%force p %%key))))

) ; end scope of %%key

(define-macro (assume form env)
    (destructuring-bind (_ proposition . body) form
      `(!%assume ,proposition (mu () ,@body))))

(define-primitive-method (dn nnp)
    ;; ~~p/p
    (match (ab-check nnp)
     ((not (not p)) p)
     (_ (error formula " is not a double negation"))))

(define-primitive-method (%suppose-absurd p d)
    (match (!%assume p d)
     ((if p (equals FALSE)) (not p))
     (_ (error "malformed application of %suppose-absurd"))))

(define-macro (suppose-absurd form env)
    (destructuring-bind (_ p . body) form
      `(!%suppose-absurd ,p (mu () ,@body))))
```

Example 40 (Axiomatic Propositional Calculus). Using the very same data structures from the previous example, it is possible to implement a denotational proof language for the propositional calculus using an axiomatization of the propositional calculus. This axiomatic presentation also makes use of the source code from Section E.1 (p. 113), but differs in the definition of the primitive methods. All but one of the (non-claim) primitive methods take propositions as arguments and return instantiations of axiom schemata with the given propositions. The only inference rule that depends on the content of the assumption base is `modus-ponens`, which is defined similarly to `mp` from the previous example.

```
(define-primitive-method (then-1 p q)
    ;; p ⊃ (q ⊃ p)
    (if p
        (if q p)))

(define-primitive-method (then-2 p q r)
    ;; (p ⊃ (q ⊃ r)) ⊃ ((p ⊃ q) ⊃ (p ⊃ r))
    (if (if p (if q r))
        (if (if p q)
            (if p r))))

(define-primitive-method (and-1 p q)
    ;; (p & q) ⊃ p
    (if (and p q) p))

(define-primitive-method (and-2 p q)
    ;; (p & q) ⊃ q
    (if (and p q) q))

(define-primitive-method (and-3 p q)
    ;; p ⊃ (q ⊃ (p & q))
    (if p (if q (and p q))))

(define-primitive-method (or-1 p q)
    ;; p ⊃ (p ∨ q)
    (if p (or p q)))

```

```
(define-primitive-method (or-2 p q)
    ;; q ⊃ (q ∨ q)
    (if q (or p q)))

(define-primitive-method (or-3 p q r)
    ;; (p ⊃ r) ⊃ ((q ⊃ r) ⊃ ((p ∨ q) ⊃ r))
    (if (if p r)
        (if (if q r)
            (if (or p q)
                r))))

(define-primitive-method (not-1 p q)
    ;; (p ⊃ q) ⊃ ((p ⊃ ∼q) ⊃ ∼p)
    (if (if p q)
        (if (if p (not q))
            (not p))))

(define-primitive-method (not-2 p q)
    ;; p ⊃ (∼p ⊃ q)
    (if p (if (not p) q)))

(define-primitive-method (not-3 p)
    ;; p ∨ ∼p
    (or p (not p)))

(define-primitive-method (modus-ponens antecedent conditional)
    ;; p, p ⊃ q / q
    (match (list (ab-check antecedent)
                 (ab-check conditional))
     ((list p (if p q)) q)
     (_ (error "malformed application of modus-ponens to antecedent "
               antecedent " and conditional " cond "."))))
```

The presentations of the propositional calculus in Examples 39 and 40 are equivalent in that exactly the same set of theorems is derivable from each system. Yet they clearly differ in that the deductions that derive any given theorem are very different. The proof that these approaches are equivalent is well known, and has been given for a variety of axiomatic and natural deduction systems, not just the two shown above. The implementation of the two systems, despite the fact that they use the same underlying data structures to represent propositions, does not provide constructions for deriving a sentence using one set of primitive methods given a derivation using the other.

It is precisely this type of consideration that leads us to develop categorical denotational proof languages, whose objects do preserve enough information about the way in which a sentence is derived so as to guide the re-derivation in another system.

Remark 6 (Type-α and Type-ω DPLs). Traditional (i.e., non-categorical) DPLs have no provision for distinguishing one deduction of a proposition from another. The value returned by method application is a proposition; no trace of the underlying primitive methods that were used is preserved. However, it is worth noting that Arkoudas distinguished between two types of DPLs: type-α and type-ω (Arkoudas, 2001a,b). The difference is that type-α DPLs lack the abstraction mechanisms provided by λ and μ expressions. Every proof in a type-α DPL, then, is based only on primitive methods and deduction composition. Every type-ω DPL contains a type-α DPL that can be obtained simply by removing abstractions, and every successful deduction in a type-ω DPL gives rise to a corresponding type-α deduction that is essentially the "trace" of primitive method calls and compositions used in the deduction. Arkoudas calls this "trace" of a deduction its *certificate*. In our categorical DPLs, deductions produce arrows from categories, which serve as their own certificates.

3.4 Implementing Categories

We have seen three examples of how the present work supports the creation of denotational proof languages, especially by exploiting data structures and algorithms available in the host language, Java. Our approach for fluid logics is to define categorical denotational proof languages, first defining categorical structures, that is, objects, arrows, and categories, as Java classes, and then implementing primitive methods based on them.

Many features of popular logical systems can be expressed categorically in such a manner as to abstract away one or more implementation details. For instance, Example 28 (p. 33) showed that logical conjunction is captured by the categorical notion of products. If a category has products, then, given any two objects of the category, A and B, we may obtain from the category two projections, π and π', the domain of each of which is the product of A and B (typically denoted $A \times B$, but this is simply convention) and the codomains of which are A and B, respectively. Additionally, for any pair of arrows $f : C \to A$ and $f : C \to B$, we may obtain a product arrow $\langle f, g \rangle : C \to A \times B$. When treating logical systems as categories, we now know of three proofs in logical systems that have products, even if we do not know the details of the objects of the system.

3.4.1 Graphs & Categories

The primary data structures needed to implement logical systems categorically are directed graphs and categories. The Graph and Category interfaces are shown in Figure 3.1. Each instance of Graph implements operations to check whether an arbitrary Java object is an arrow (i.e., an edge) in the graph, and to extract the domain and codomain of any arrows (i.e., edges) in the graph. Each instance of Category implements operations to support the additional structure that Categories add to graphs. Specifically, categories provide operations to create identity and composite arrows, test whether objects are identity and composite arrows, and to extract the components of a composite arrow. No operation is needed for determining the object associated with an identity arrow, since Graph's domain and codomain operations will return it.

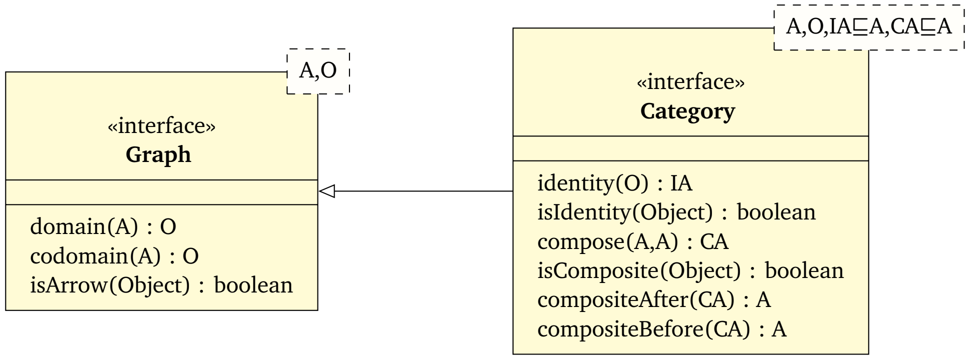


Figure 3.1: The Graph and Category interfaces provide the operations needed to work with directed graphs and the basic arrows common to all categories.

Using Java interoperability and the interfaces defined here, we can begin to define category-agnostic methods in a categorical DPL. For instance, while working with any category, we may, given an object in that category, retrieve the identity arrow for that object. Similarly, we may compose any compatible arrows in a category. All the methods declared by the Graph and Category interfaces require an instance of the graph or category. We shall very often be concerned with working in multiple categories, and with switching between them. One approach toward achieving this keeps explicit references to category instances, and passes these to primitive methods that accept categories as arguments. This approach would define a primitive method `identity` as follows.

```
(define-primitive-method (identity category object)
  (.identity category object))
```

A second approach would treat category references implicitly, and keep the category provided to the Java `.identity` method hidden from the user, stored in a dynamically scoped variable. With this approach, `identity` would be defined as follows (using the convention from Common Lisp that dynamically scoped variables have "earmuffs," i.e., initial and final asterisks).

```
(define-primitive-method (identity object)
  (.identity *category* object))
```

We prefer the second approach, as most of the code written by users will focus on deductions in just one category, and it is inconvenient to manually pass category arguments to every method application. However, the language at present does not support true dynamic variables. As a workaround, we define a function, `current-category`, which returns the "current working category," using which `identity` may defined as follows.

```
(define-primitive-method (identity object)
  (.identity (current-category) object))
```

Using this technique, we define primitive methods for `identity` (as just shown) and `compose`. We discuss "cross-category reasoning" and the dynamic redefinition of `current-category` more fully in Section 3.5.5 (p. 71).

Example 41 (Natural Numbers). As demonstrated in Example 23 under addition have a category structure. The category has a single object, which we denote by $\star$. The identity arrow on $\star$ is *zero*, and the category has an arrow $\mathit{one}\colon \star \to \star$. The composition operator is addition. The following Java class, `NaturalNumbers`, implements the category interface. Arrows of the category are instances of the Java `Integer` class (but only non-negative `Integers` are arrows). A locally defined enumeration, `Star`, with a single element, provides the single object of the category. Every arrow in the category can be viewed as the composite of some other arrows, and so the implementation of `compositeAfter` and `compositeBefore` are somewhat arbitrary.

```java
package edu.rpi.cs.tayloj.fluid;

import edu.rpi.cs.tayloj.fluid.category.Category;
public class NaturalNumbers
     implements Category<NaturalNumbers.Star,Integer,Integer,Integer> {
  public NaturalNumbers() {}

  public enum Star { STAR }

  public Star domain( Integer n ) { return Star.STAR; }
  public Star codomain( Integer a ) { return Star.STAR; }

  public Integer identity( Star o ) { return 0; }
  public Integer compose( Integer n, Integer m ) { return n + m; }

  public Integer zero() { return 0; }
  public Integer one() { return 1; }

  public boolean isArrow( Object object ) {
    return ( object instanceof Integer ) && ((Integer) object) >= 0;
  }

  public boolean isIdentity( Object object ) {
    return zero().equals( object );
  }

  /** Every arrow can be represented as a sum of 0 and itself. */
  public boolean isComposite( Object object ) { return isArrow( object ); }
  public Integer compositeAfter( Integer h ) { return 0; }
  public Integer compositeBefore( Integer h ) { return h; }
}
```

With the Java implementation of $\mathcal{N}$ in place, the corresponding denotational proof language is a simple matter of making `current-category` return an instance of `NaturalNumbers` and defining primitive methods that wrap `zero` and `one`.

```scheme
(let ((NN (edu.rpi.cs.tayloj.fluid.NaturalNumbers.)))
  (define (current-category)
    NN))

```

```
(define-primitive-method (zero)
  ;; zero: ⋆ → ⋆
  (.zero (current-category)))

(define-primitive-method (one)
  ;; one: ⋆ → ⋆
  (.one (current-category)))
```

The natural numbers category is not really a proof system, but it is a category with arrows having some structure, and is sufficient to demonstrate how we can build more complex methods up from the primitive methods. In $\mathcal{N}$, we may "derive" natural numbers by constructing them as sums (compositions). The following code shows a method that derives the natural number n using a number of compositions linear in n, as well as a method that derives n using a number of compositions logarithmic in n.

```
(define (linear-prove n)
  (dcheck
   ((== n 0) (!zero))
   ((== n 1) (!one))
   (else (!compose (!one) (!linear-prove (- n 1))))))

(define (log-prove n)
  (dcheck
   ((== n 0) (!zero))
   ((== n 1) (!one))
   (else (dlet* ((half (!log-prove (/ n 2)))
                 (whole (!compose half half)))
                (dcheck
                 ((== 0 (% n 2)) (!claim whole))
                 (else (!compose (!one) whole)))))))
```

Simple timing shows the marked difference between the performance of the two. Since each application of a primitive method corresponds to a single step in a proof, we may view `linear-prove` as creating longer, or less efficient, proofs than `log-prove`.

```
> (dtime (!linear-prove 3294))
Evaluation required 0.931015613 seconds

3294
> (dtime (!log-prove 3294))
Evaluation required 0.005282027 seconds

3294
```

3.4.2 Categorical Constructions

We implemented Java interfaces for a number of the categorical features that capture properties of logical systems. In particular, we implemented interfaces for categories with initial and terminal objects, products and coproducts, and exponential objects. We briefly discussed the significance of products in Example 25 (p. 32), but have not yet discussed these other categorical constructions. We now describe these constructs, as well as the Java interfaces that declare the operations needed to work with them.

3.4.2.1 Initials & Terminals

A terminal object in a category is an object, typically denoted $\top$ or $\mathbf{1}$, such that for every object A in the category, there is exactly one arrow $\top_A \colon A \to \top$, called the terminal arrow for A. Note that the terminal arrow for $\top$, $\top_\top$, must be the same as $\mathrm{id}_\top$. In logical systems, the terminal object represents truth, and the existence of terminal arrows corresponds to the principle that truth is a consequence of every formula.

$$\begin{tikzcd} A \arrow[r, dashed, "\top_A"] & \top \end{tikzcd} \tag{3.1}$$

Initial objects are the dual of terminal objects, and are typically denoted by $\perp$ or $\mathbf{0}$. In a category with an initial object, $\perp$, for each object A there is exactly one arrow $\perp_A \colon \perp \to A$. In logical systems, the initial object represents falsehood, and the existence of initial arrows corresponds to the principle that everything follows from a contradiction. Just as $\mathrm{id}_\top = \top_\top$, so also $\mathrm{id}_\perp = \perp_\perp$.

$$\begin{tikzcd} \perp \arrow[r, dashed, "\perp_A"] & A \end{tikzcd} \tag{3.2}$$

Initial objects also play a role in representing negation. Exponentials (§ 3.4.2.3) will be used to represent conditionals, and will adopt the tradition of taking the negation of a formula, $\sim A$, as shorthand for the conditional $A \supset \perp$.

From an implementation standpoint, a category with an initial (terminal) object must provide an operation to obtain the initial (terminal) arrows for objects in the category. Once an initial (terminal) arrow is obtained, the initial (terminal) object can be obtained by taking the domain (codomain) of the arrow. For convenience, our interfaces also provide operations for obtaining the initial (terminal) object, checking whether an arbitrary Java `Object` is the initial (terminal) object, and determining whether an arbitrary Java `Object` is an initial (terminal) arrow. The Java interfaces `HasInitial` and `HasTerminal` extend the `Graph` interface, and are depicted in Figure 3.2.

Remark 7 (Uniquesness up to isomorphism and Java Interfaces). As noted earlier in Remark 4 (p. 34), most categorical constructions are unique only up to isomorphism. This is the case with initial and terminal objects. A category may actually have more than one initial or terminal object, but they will isomorphic. Suppose that a category has two initial objects, $\perp$ and $\perp'$. There is exactly one arrow from $\perp$ to $\perp'$, and thus $\mathrm{id}_\perp = \perp'_\perp \circ \perp_{\perp'}$ and $\mathrm{id}_{\perp'} = \perp_{\perp'} \circ \perp'_\perp$.

In Java's type system, however, a class cannot implement an interface more than once, so the `HasInitial` and `HasTerminal` can only be used to identity *one* initial and *one* terminal in a category. Because these are objects are unique up to isomorphism, we do not view this as a significant limitation. This analysis applies to other categorical constructions as well.

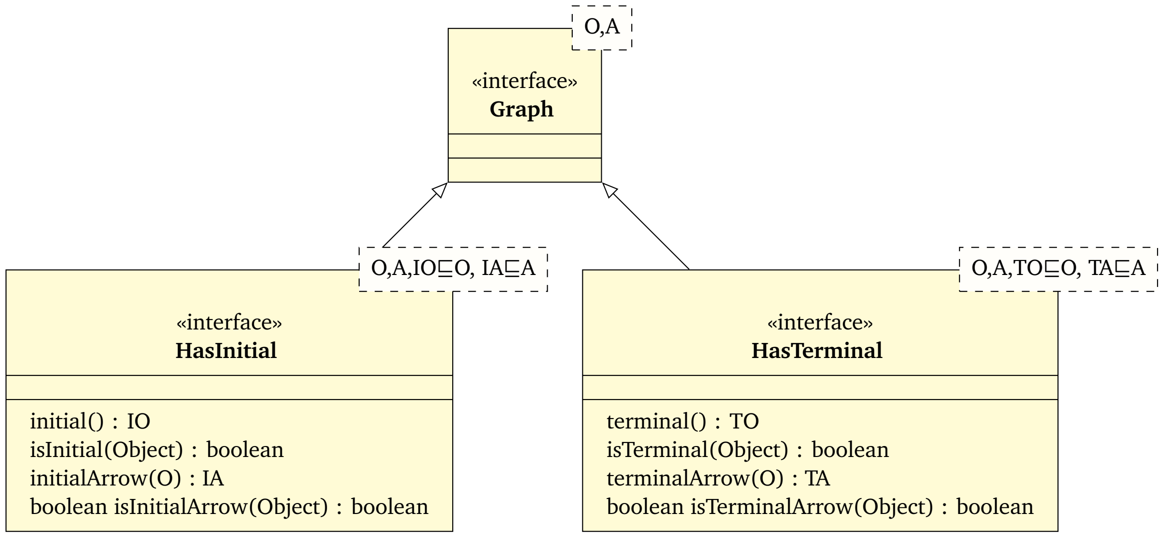


Figure 3.2: The HasInitial and HasTerminal interfaces declare all the operations needed for retrieving the initial and terminal objects from categories that have them, as well as locating initial and terminal arrows for objects in the category.

3.4.2.2 Products & Coproducts

Products, previously discussed in Example 25 (p. 32), and coproducts are categorical constructions that correspond, in most logical systems, to conjunctions and disjunctions, respectively. In a category with products, the product of objects A and B, typically denoted $A \times B$, is an object such that there are projection arrows $\pi_{A,B} : A \times B \to A$ and $\pi'_{A,B} : A \times B \to B$, and that for any arrows $f : C \to A$ and $g : C \to B$, there is a unique arrow $\langle f, g \rangle : C \to A \times B$.

$$\begin{tikzcd}
 & C \arrow[ld, "f"'] \arrow[d, dashed, "\langle f,g \rangle"] \arrow[rd, "g"] & \\
A & A \times B \arrow[l, "\pi_{A,B}"] \arrow[r, "\pi'_{A,B}"'] & B
\end{tikzcd} \tag{3.3}$$

The left and right projection arrows $\pi_{A,B}$ and $\pi'_{A,B}$ correspond to the logical proofs that the conjuncts of a conjunction follow from the conjunction. The product arrow $\langle f, g \rangle$ corresponds to principle that the product of A and B follows from anything from which A and B each follow.

Given two arrows $f : A \to C$ and $g : B \to D$, there is an arrow $\langle f\pi, g\pi' \rangle : A \times B \to C \times D$. This arrow is also denoted as $f \times g : A \times B \to C \times D$ in the literature, and we adopt this convention as well.

Coproducts, also known as sums, are the dual of products and correspond to disjunction in logical systems. In a category with coproducts, the coproduct of objects A and B, denoted $A + B$, is an object such that there are injection arrows $\iota_{A,B} : A \to A + B$ and $\iota'_{A,B} : B \to A + B$, and that for any pair of arrows

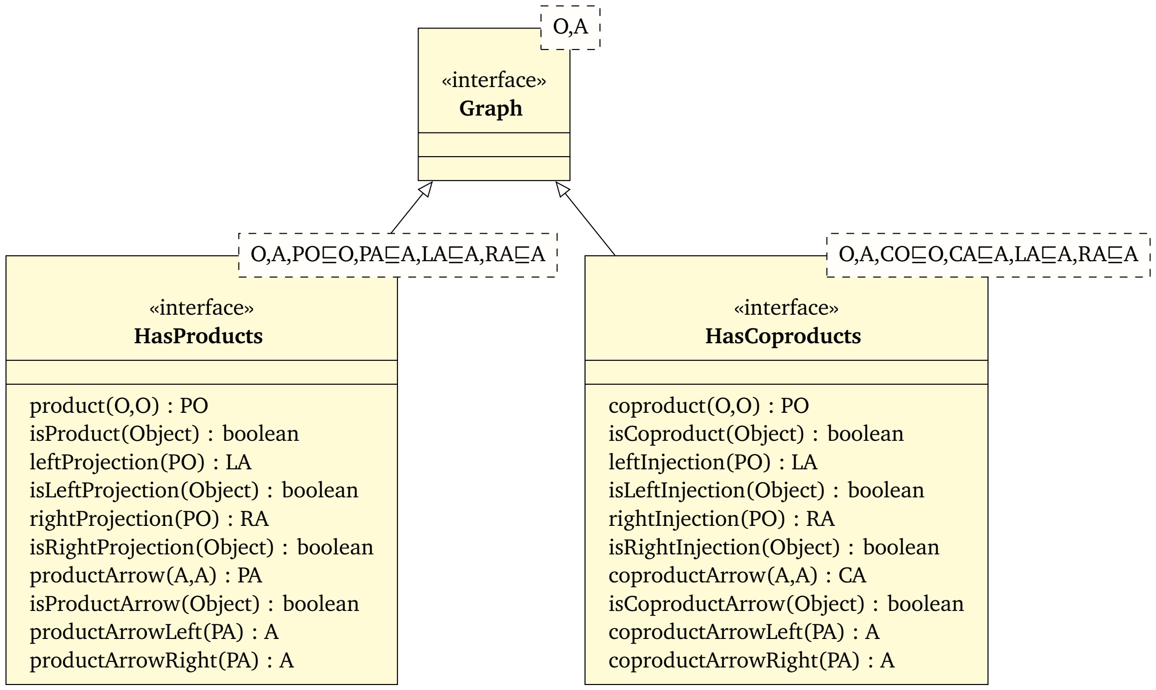

O,A
«interface»
Graph
O,A,PO⊑O,PA⊑A,LA⊑A,RA⊑A
«interface»
HasProducts
product(O,O) : PO
isProduct(Object) : boolean
leftProjection(PO) : LA
isLeftProjection(Object) : boolean
rightProjection(PO) : RA
isRightProjection(Object) : boolean
productArrow(A,A) : PA
isProductArrow(Object) : boolean
productArrowLeft(PA) : A
productArrowRight(PA) : A
O,A,CO⊑O,CA⊑A,LA⊑A,RA⊑A
«interface»
HasCoproducts
coproduct(O,O) : PO
isCoproduct(Object) : boolean
leftInjection(PO) : LA
isLeftInjection(Object) : boolean
rightInjection(PO) : RA
isRightInjection(Object) : boolean
coproductArrow(A,A) : CA
isCoproductArrow(Object) : boolean
coproductArrowLeft(PA) : A
coproductArrowRight(PA) : A

Figure 3.3: The HasProducts and HasCoproducts interfaces declare all the operations needed for retrieving product and coproduct objects and arrows from a category.

$f : A \to C$ and $g : B \to C$, there is a unique coproduct arrow $[f, g] : A + B \to C$. The injection arrows correspond to proofs of disjunction introduction, whereby a disjunction may be inferred from either of its disjuncts. The coproduct arrow is a proof by cases; given proofs of a proposition from each disjunct, that proposition follows from the disjunction as a whole.

$$\begin{tikzcd}
A \arrow[r, "\iota_{A,B}"] \arrow[dr, "f"'] & A+B \arrow[d, dashed, "{[f,g]}"] & B \arrow[l, "\iota'_{A,B}"'] \arrow[dl, "g"] \\
& C &
\end{tikzcd} \tag{3.4}$$

In notation mirroring that for products, given arrows $f : A \to C$ and $g : B \to D$, there is an arrow $[\iota f, \iota' g] : A + B \to C + D$ that can be abbreviated $f + g : A + B \to C + D$. This abbreviation is also used in the literature, but is somewhat less common than the $f \times g$ notation used with products.

Implementations of categories with products and coproducts need to provide access to the product and coproduct objects of the category, the projections and injections, and the product and coproduct arrows. Additionally, the interfaces `HasProducts` and `HasCoproducts` provide predicates for checking whether Java `Objects` are these categorical objects or arrows. UML diagrams of these interfaces are given in Figure 3.3.

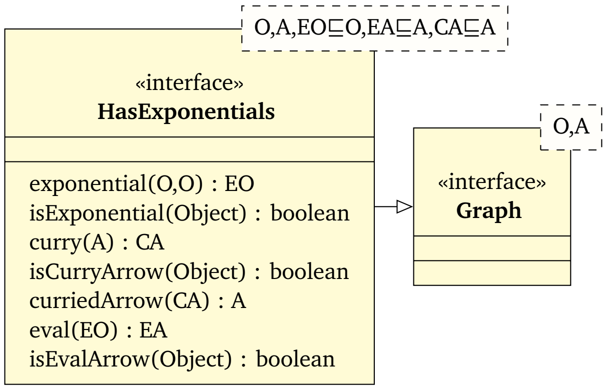


Figure 3.4: The HasExponentials interface declares all the operations needed for retrieving exponential objects, eval, and curry arrows from a category.

3.4.2.3 Exponentials

Exponentials are one of the most important constructions in categories that are related to logics (as well as programming languages). In logical categories, exponential objects play the role of conditionals, and in programming languages they are functions. An exponential in a category with products is an object C^B and an associated arrow $\text{eval}_{B,C}$ such that that the following diagram commutes:

$$\begin{tikzcd}
A \arrow[d, dashed, "\lambda f"'] & A \times B \arrow[d, dashed, "\lambda f \times \text{id}_B"'] \arrow[dr, "f"] & \\
C^B & C^B \times B \arrow[r, "\text{eval}_{B,C}"'] & C
\end{tikzcd} \tag{3.5}$$

In logical systems, exponential objects correspond to conditionals, where C^B is the conditional $B \supset C$. Indeed, where the product is interpreted as conjunction, $f : A \times B \to C$ is a proof of C from the conjunction of A and B, and $\lambda f : A \to C^B$ is the proof from A that $B \supset C$. The eval arrow is readily seen to be conditional elimination, i.e., *modus ponens*. Categories with exponential objects implement the `HasExponentials` interface, shown in Figure 3.4.

3.4.2.4 Double Negations

The categorical features described so far provide enough to reconstruct the features of the intuitionistic propositional calculus. All of the propositional constructors (&, $\vee$, $\supset$, and $\sim$— as an abbreviation for — $\supset \bot$) correspond nicely to categorical constructions. To move into the realm of classical propositional calculus, we define one more categorical interface, `HasDoubleNegation`, which provides an arrow schema:

$$\mathsf{dn}_\phi : \sim\sim\phi \to \phi \tag{3.6}$$

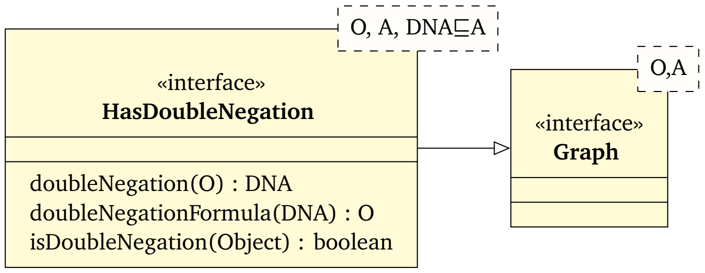


Figure 3.5: The HasDoubleNegation interface declares all the operations needed for obtaining double negation arrows from a category. (In actual implementation, negations (and double negations) are exponential objects ($\sim\phi$ abbreviates $\phi \supset \bot$), so HasDoubleNegation also extends HasTerminal and HasExponentials.)

Categories with double negation arrows implement the `HasDoubleNegation` interface, shown in Figure 3.5. An alternative approach to obtaining a category with classical semantics would be an arrow schema providing the excluded middle as a theorem:

$$\mathsf{em}_\phi : \top \to \phi \vee \sim\phi \tag{3.7}$$

This approach is demonstrably equivalent. A proof is given in Appendix D (p. 109).

3.4.3 Categories for Fluid Logics

The interfaces described in the previous section provide a useful abstraction layer that assists in implementing category-agnostic deductive methods. Some combinations of these interfaces correspond to types of categories that are of particular interest to logicians. These are particularly useful starting points for implementing fluid logics.

3.4.3.1 Cartesian & Bicartesian Closed Categories

A category is cartesian closed (Lambek & Scott, 1988) if it: (i) has a terminal object; (ii) has binary products; and (iii) has exponentials. We have seen how, in logical categories, terminal objects correspond to truth, products to conjunction, and exponentials to conditionals. A logic represented by a cartesian closed category, then, has all these features. The positive intuitionistic propositional calculus from Example 20 (p. 30) is an example of a logic whose category is cartesian closed.

A bicartesian closed category is simply a cartesian closed category with an initial object and coproducts. As logical systems, bicartesian closed categories have both truth and falsehood as sentences, conjunction and disjunction, and conditionals. The intuitionistic propositional calculus is an example of a logic whose categorical treatment is a bicartesian closed category. Bicartesian closed categories that additionally have

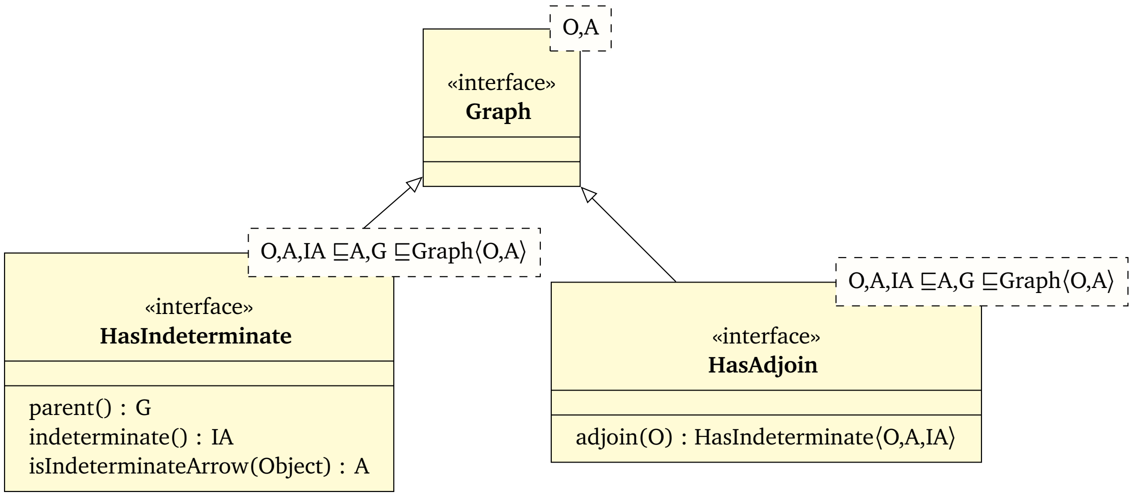


Figure 3.6: The HasIndeterminate interface declares all the operations needed for working with the indeterminate arrow x of a category $\mathscr{C}[x]$, and for retrieving the parent category, $\mathscr{C}$. Categories that can be extended with indeterminate arrows implement HasAdjoin, which can produce instances of HasIndeterminate.

double negation arrows capture classical propositional logic.

3.4.3.2 Indeterminates & Polynomial Categories

One of the initial motivations for category-based denotational proof languages was that the types of "reasoning contexts" so common in natural deduction proof systems, such as conditional or modal subproofs, seem to be captured very well by reasoning in "related" categories. For instance, let $\mathscr{C}$ be a category for the intuitionistic propositional calculus. Then, let $\mathscr{C}[x\colon \top \to A]$, or just $\mathscr{C}[x]$, be the category like $\mathscr{C}$, but with an additional arrow $x\colon \top \to A$, as well as any additional arrows required by the structure of the category (e.g., $\iota x\colon \top \to A \vee B$). The arrows of $\mathscr{C}[x]$, then, are those proofs that can be obtained in the intuitionistic propositional calculus, were it the case that A is a theorem demonstrated by proof x. Reasoning within $\mathscr{C}[x]$ is exactly like reasoning in a natural deduction conditional subproof beginning with the assumption A. We have already seen an example of this type of category with regard to the deduction theorem in Example 31 (p. 36). Categories that have indeterminate arrows implement the `HasIndeterminate` interface, shown in Figure 3.6. Lambek & Scott (1988) call these polynomial categories.

Remark 8 (Java's type system and the relationship between $\mathscr{C}$ and $\mathscr{C}[x]$). A Java object `cx` representing $\mathscr{C}[x]$ would implement the interface `HasIndeterminate`, and its `parent()` method should return an object `c` representing $\mathscr{C}$. Because $\mathscr{C}[x]$ contains all the arrows of $\mathscr{C}$, it should be the case that `c.isArrow(a)` implies `cx.isArrow(a)`. While the documentation of the interfaces explains this requirement, it cannot be enforced by Java's type system.

3.5 Fluid Logics for the Natural-Deduction and Axiomatic Propositional Calculi

We have now developed sufficient categorical structure to capture natural-deduction propositional logic, and can begin to examine the relationships between logical categories. We develop the natural-deduction propositional calculus and the axiomatic propositional calculus as our first fluid logics. We begin by defining categories for the axiomatic propositional calculus and the natural deduction propositional calculus, and proceed by demonstrating a proof mapping from the former to the latter.

3.5.1 Categorical Axiomatic Propositional Calculus

The axiomatic propositional calculus presented in Example 40 (p. 52) gives rise to a category with a very simple structure. The objects are the sentences of the propositional calculus, and each axiom schema σ in the calculus, there is an arrow schema for the category $\top \to \sigma$. For the modus ponens inference rule, there is a corresponding arrow rule. Additionally, we posit the existence of identity and composition arrows though, as we shall see, they serve little purpose in this category. The arrow schemata and rules for the category are:

$$\mathrm{id}_p : p \to p \qquad \frac{f : p \to q \quad g : q \to r}{gf : p \to r}$$

$$\mathrm{then}_1 : \top \to p \supset (q \supset p) \qquad \mathrm{then}_2 : \top \to (p \supset (q \supset r)) \supset ((p \supset q) \supset (p \supset r))$$

$$\mathrm{and}_1 : \top \to (p \,\&\, q) \supset p \qquad \mathrm{and}_2 : \top \to (p \,\&\, q) \supset q \qquad \mathrm{and}_3 : \top \to p \supset (q \supset (p \,\&\, q))$$

$$\mathrm{or}_1 : \top \to p \supset (p \vee q) \qquad \mathrm{or}_2 : \top \to q \supset (p \vee q) \qquad \mathrm{or}_3 : \top \to (p \supset r) \supset ((q \supset r) \supset ((p \vee q) \supset r))$$

$$\mathrm{not}_1 : \top \to (p \supset q) \supset ((p \supset \sim q) \supset \sim p) \qquad \mathrm{not}_2 : \top \to p \supset (\sim p \supset q) \qquad \mathrm{not}_3 : \top \to p \vee \sim p$$

$$\frac{f : \top \to p \supset q \quad g : \top \to p}{\mathrm{mp}_{f,g} : \top \to q}$$

We include identity and composite arrows because this is a category, but we note that the identity arrows are not particularly interesting, and that there are very few arrows in the category that can be composed because all arrow schemata produce arrows with the same domain. Additionally, while this category captures the propositional calculus insofar as that for every theorem ϕ, there is an arrow $\top \to \phi$, the objects of the category have very little interesting structure. For instance $p \,\&\, q$ is not an product of p and q in this category, for there are no projection arrows $p \,\&\, q \to p$ or $p \,\&\, q \to q$. Similarly, disjunctions are not coproducts, truth and falsehood are not terminal and initial objects, and conditionals are not exponentials. The implementation of the axiomatic propositional calculus as a categorical DPL is straightforward, and given in Section E.2 (p. 118).

3.5.2 Categorical Natural Deduction Propositional Calculus

The natural deduction propositional calculus given in Example 39 (p. 50) is readily captured by a bicartesian closed category with double negation arrows. (For the moment, we ignore the abuse of mapping the `(assume ...)` deductive form to curry arrows. We will develop a more natural mapping for `assume` later in § 3.6 (p. 72).) An implementation, then, needs only to provide a Java implementation of the appropriate interfaces, and to provide a mapping of the "standard" categorical names to names for the propositional calculus. The implementation and name mapping are given in Section E.3 (p. 126). The arrow schemata for the natural deduction category are as follows. These are familiar from preceding sections, and we do not repeat the uniqueness conditions of arrows here.

$$\mathsf{reit}(A) \equiv \mathrm{id}_A : A \to A \qquad \frac{f : A \to B \quad g : B \to C}{\mathsf{compose}(g,f) \equiv gf : A \to C}$$

$$\mathsf{true\text{-}intro}(A) \equiv \top_A : A \to \top \qquad \mathsf{false\text{-}elim}(A) \equiv \bot_A : \bot \to A$$

$$\mathsf{left\text{-}and}(A,B) \equiv \pi_{A,B} : A \,\&\, B \to A \qquad \mathsf{right\text{-}and}(A,B) \equiv \pi'_{A,B} : A \,\&\, B \to B$$

$$\frac{f : C \to A \quad g : C \to B}{\mathsf{both}(f,g) \equiv \langle f,g\rangle : C \to A \,\&\, B} \qquad \frac{f : A \to C \quad g : B \to C}{\mathsf{cd}(f,g) \equiv [f,g] : A \vee B \to C}$$

$$\mathsf{left\text{-}or}(A,B) \equiv \iota_{A,B} : A \to A \vee B \qquad \mathsf{right\text{-}or}(A,B) \equiv \iota'_{A,B} : B \to A \vee B$$

$$\mathsf{modus\text{-}ponens}(A,B) \equiv \mathsf{eval}_{A,B} : (A \supset B) \,\&\, A \to B \qquad \frac{f : A \,\&\, B \to C}{\mathsf{assume}(f) \equiv \lambda f : A \to B \supset C}$$

$$\mathsf{dn}(A) \equiv \mathsf{dn}_A : \sim\sim A \to A$$

3.5.3 Mapping Axiomatic PC to Natural Deduction PC

We now present a mapping from the axiomatic propositional calculus category to the natural deduction propositional calculus category. The mapping maps each proposition to itself, and because the axiomatic category has no non-trivial composition arrows, the mapping trivially preserves identity and composition arrows, and is thus a functor. We define the mapping by defining the corresponding natural deduction arrows for each of twelve axiomatic schemata.

Identity and composition arrows are the simplest, and map to themselves.

Conditional Arrows

The then$_1$ schema is easy to derive in the natural deduction category:

$$\text{then}_1 \mapsto \frac{\pi'\pi \colon (\top \,\&\, p) \,\&\, q \to p}{\lambda\lambda\pi'\pi \colon \top \to p \supset (q \supset p)}$$

The then$_2$ schema requires a bit more work, but is still relatively straightforward. First, note that $q \supset r$ and q are easy to derive:

$$\text{eval}(\pi'\pi\pi \times \text{id}) \colon ((\top \,\&\, (p \supset (q \supset r))) \,\&\, (p \supset q)) \,\&\, p \to q \supset r \tag{3.8}$$

$$\text{eval}(\pi'\pi \times \text{id}) \colon ((\top \,\&\, (p \supset (q \supset r))) \,\&\, (p \supset q)) \,\&\, p \to q \tag{3.9}$$

Composing eval with the product of these yields r, and currying three times provides the mapping for then$_2$:

$$\text{then}_2 \mapsto \frac{\text{eval}\langle(3.8),(3.9)\rangle \colon ((\top \,\&\, (p \supset (q \supset r))) \,\&\, (p \supset q)) \,\&\, p \to r}{\lambda\lambda\lambda\text{eval}\langle(3.8),(3.9)\rangle \colon \top \to (p \supset (q \supset r)) \supset ((p \supset q) \supset (p \supset r))}$$

Conjunction Arrows

Conjunction arrows have the simplest mappings because of the close connection between products and exponentials in cartesian categories.

$$\text{and}_1 \mapsto \frac{\pi\pi' \colon \top \,\&\, (p \,\&\, q) \to p}{\lambda(\pi\pi') \colon \top \to (p \,\&\, q) \supset p} \qquad \text{and}_2 \mapsto \frac{\pi'\pi' \colon \top \,\&\, (p \,\&\, q) \to q}{\lambda(\pi'\pi') \colon \top \to (p \,\&\, q) \supset q}$$

$$\text{and}_3 \mapsto \frac{\pi' \times \text{id} \colon (\top \,\&\, p) \,\&\, q \to p \,\&\, q}{\lambda\lambda(\pi' \times \text{id}) \colon \top \to p \supset (q \supset (p \,\&\, q))}$$

Disjunction Arrows

The first two disjunction arrows are trivial:

$$\text{or}_1 \mapsto \lambda(\iota\pi') \colon \top \to p \supset (p \vee q) \qquad \text{or}_2 \mapsto \lambda(\iota'\pi') \colon \top \to q \supset (p \vee q)$$

The or$_3$ is somewhat complicated by the fact that it captures “proof by cases,” but coproduct arrows don’t have an obvious way of incorporating “in-scope” assumptions into each case. First, we note that r can be derived from the assumptions $p \supset r$ and $q \supset r$, and either of p or q:

$$\text{eval}\langle\pi\pi',\pi\rangle \colon p \,\&\, ((p \supset r) \,\&\, (q \supset r)) \to r \tag{3.10}$$

$$\text{eval}\langle\pi'\pi',\pi\rangle \colon q \,\&\, ((p \supset r) \,\&\, (q \supset r)) \to r \tag{3.11}$$

These give rise to a useful coproduct arrow:

$$[(3.10),(3.11)]\colon (p \mathbin{\&} ((p \supset r) \mathbin{\&} (q \supset r))) \vee (q \mathbin{\&} ((p \supset r) \mathbin{\&} (q \supset r))) \to r \tag{3.12}$$

For any objects in a bicartesian closed category, there is an arrow that distributes products over coproducts. A derivation is given in Theorem 46 (p. 108). Using logical connectives, we take this instantiation:

$$\delta\colon (A \vee B) \mathbin{\&} C \to (A \mathbin{\&} C) \vee (B \mathbin{\&} C)$$

Composing an appropriate instantiation of δ with (3.12), we have

$$(3.12)\delta\colon (p \vee q) \mathbin{\&} ((p \supset r) \mathbin{\&} (q \supset r)) \to r \tag{3.13}$$

The commutativity of conjunction provides:

$$\langle \pi', \langle \pi'\pi\pi, \pi'\pi\rangle\rangle\colon ((\top \mathbin{\&} (p \supset r)) \mathbin{\&} (q \supset r)) \mathbin{\&} (p \vee q) \to (p \vee q) \mathbin{\&} ((p \supset r) \mathbin{\&} (q \supset r)) \tag{3.14}$$

Composing (3.13) and (3.14), and currying three times, we finally derive:

$$\mathrm{or}_3 \mapsto \lambda\lambda\lambda((3.13)(3.14))\colon \top \to (p \supset r) \supset ((q \supset r) \supset ((p \vee q) \supset r))$$

Negation Arrows

The not_1 schema is actually just a variant of then_2, in light of the fact that we take $\sim\phi$ as an abbreviation of $\phi \supset \bot$. In particular, r of then_1 is fixed as $\bot$, and the order of $p \supset (q \supset r)$ and $p \supset q$ is reversed:

$$\mathrm{then}_2\colon \top \to (p \supset (q \supset r)) \supset ((p \supset q) \supset (p \supset r))$$
$$\mathrm{not}_1\colon \top \to (p \supset q) \supset ((p \supset (q \supset \bot)) \supset (p \supset \bot))$$

Thus we have

$$\mathrm{eval}(\pi'\pi \times \mathrm{id})\colon ((\top \mathbin{\&} (p \supset q)) \mathbin{\&} (p \supset \sim q)) \mathbin{\&} p \to \sim q \tag{3.15}$$
$$\mathrm{eval}(\pi'\pi\pi \times \mathrm{id})\colon ((\top \mathbin{\&} (p \supset q)) \mathbin{\&} (p \supset \sim q)) \mathbin{\&} p \to q \tag{3.16}$$

and then

$$\mathrm{not}_1 \mapsto \lambda\lambda\lambda\mathrm{eval}\langle(3.15),(3.16)\rangle\colon \top \to (p \supset q) \supset ((p \supset \sim q) \supset \sim p)$$

The mapping for not_2 is simple:

$$\mathrm{not}_2 \mapsto \dfrac{\dfrac{\dfrac{\langle \pi', \pi'\pi\rangle \colon (\top \mathbin{\&} p) \mathbin{\&} {\sim} p \to {\sim} p \mathbin{\&} p \quad \mathrm{eval}\colon {\sim} p \mathbin{\&} p \to \bot}{\mathrm{eval}\langle \pi', \pi'\pi\rangle \colon (\top \mathbin{\&} p) \mathbin{\&} {\sim} p \to \bot} \qquad \bot_q \colon \bot \to q}{\bot_q \mathrm{eval}\langle \pi', \pi'\pi\rangle \colon (\top \mathbin{\&} p) \mathbin{\&} {\sim} p \to q}}{\lambda\lambda(\bot_q \mathrm{eval}\langle \pi', \pi'\pi\rangle) \colon \top \to p \supset ({\sim} p \supset q)}$$

The mapping for not_3 is more complicated. The approach is to derive an arrow $\top \to {\sim}{\sim}(p \vee {\sim} p)$ and compose it with $\mathsf{dn}\colon {\sim}{\sim}(p \vee {\sim} p) \to (p \vee {\sim} p)$. This case is covered in Lemma 48 (p. 110), and we will not repeat it here.

Modus Ponens Arrows

Modus ponens and composition are the only arrow rules with preconditions in the axiomatic propositional calculus, and composition arrows are trivially handled. Modus ponens arrows are only slightly more complicated. Given two arrows from the axiomatic category, $f \colon \top \to p \supset q$ and $g \colon \top \to p$, there are corresponding natural deduction arrows $f' \colon \top \to p \supset q$ and $g' \colon \top \to p$. Then:

$$\mathrm{mp}_{f,g} \mapsto \dfrac{\langle f', g'\rangle \colon \top \to (p \supset q) \mathbin{\&} p \quad \mathrm{eval}\colon (p \supset q) \mathbin{\&} p \to q}{\mathrm{eval}\langle f', g'\rangle \colon \top \to q}$$

3.5.4 Mapping in the Language

The preceding section shows that for each arrow in the axiomatic category, there is a corresponding in the natural deduction category. We note that because the mapping preserves identity arrows and composites, it is a functor, though we will take no more advantage of that fact here. We will now show the implementation of the mapping in the language.

First, we define a function (not a method) that accepts an arrow of the axiomatic category. The function should be called in a dynamic environment where `current-category` returns an instance of the axiomatic category (since matching patterns may expand to forms that depend on the current category), and returns a natural deduction method that derives the translation of the axiomatic arrow. This function is somewhat monolithic, but captures the mapping described in the previous section.

```
(define (ax2nd ax)
  ;; ax2nd should be called in a context where current-category
  ;; returns an axiomatic PC category (because the match patterns
  ;; depend on it), and returns a method that, when called in a
  ;; context where current-category returns a natural deduction PC
  ;; category, will derive the corresponding arrow.
  (match ax
   ((->identity p)
    (mu ()
        (!identity p)))
   ((->compose g f)
    (mu ()
        (!compose g f)))
  ((->then1 (if p (if q p)))
   (mu ()
        (!discharge* 2 (!right-and* (!left-and* (!identity (and (and TRUE p) q)))))))
  ((->then2 (if (if p (if q r))
                 (if (if p q)
                     (if p r))))
   (mu ()
        (dlet* ((X (!identity (and (and (and TRUE (if p (if q r))) (if p q)) p)))
                (if-q-then-r
                 (!mp*
                  (!right-and* (!left-and* (!left-and* X)))
                  (!right-and* X)))
                (q
                 (!mp*
                  (!right-and* (!left-and* X))
                  (!right-and* X))))
         (!discharge* 3 (!mp* if-q-then-r q)))))
  ((->and1 (if (and p q) p))
   (mu ()
        (!discharge (!left-and* (!right-and* (!identity (and TRUE (and p q))))))))
  ((->and2 (if (and p q) q))
   (mu ()
        (!discharge (!right-and* (!right-and* (!identity (and TRUE (and p q))))))))
  ((->and3 (if p (if q (and p q))))
   (mu ()
        (!discharge* 2 (!times
                        (!right-and TRUE p)
                        (!identity q)))))
  ((->or1 (if p (or p q)))
   (mu ()
        (!discharge (!left-or* (!right-and TRUE p) q))))
  ((->or2 (if q (or p q)))
   (mu ()
        (!discharge (!right-or* (!right-and TRUE q) p))))
  ((->or3 (if (if p r)
               (if (if q r)
                   (if (or p q)
                       r))))
   (mu ()
        (dlet
         ((pside (!identity (and (and (and TRUE (if p r)) (if q r)) p)))
          (qside (!identity (and (and (and TRUE (if p r)) (if q r)) q))))
         (!discharge* 2
                      (!compose
                       (!distribute p q (and (and TRUE (if p r)) (if q r)))
                       (!cd (!mp* (!right-and* (!left-and* (!left-and* pside)))
                                  (!right-and* pside))
                            (!mp* (!right-and* (!left-and* qside))
                                  (!right-and* qside))))))))
  ((->not1 (if (if p q)
                (if (if p (not q))
                    (not p))))
   (mu ()
        (dlet* ((X (!identity (and (and (and TRUE (if p q)) (if p (not q))) p)))
                (not-q (!mp*
                        (!right-and* (!left-and* X))
                        (!right-and* X)))
                (q (!mp
                    (!right-and* (!left-and* (!left-and* X)))
                    (!right-and* X))))
         (!discharge 3 (!mp* not-q q)))))
  ((->not2 (if p (if (not p) q)))
   (mu ()
        (dlet ((X (!identity (and (and TRUE p) (not p)))))
         (!discharge* 2
                      (!false-elim* (!mp* (!right-and* X)
                                          (!right-and* (!left-and* X)))
```

```
                                        q)))))
    ((->not3 (or p (not p)))
     (mu ()
         (dlet ((X (!identity (and (and TRUE (not (or p (not p)))) p))))
          (!dn* (!discharge
                 (!mp*
                  (!right-and TRUE (not (or p (not p))))
                  (!right-or* (!discharge
                               (!mp* (!right-and* (!left-and* X))
                                     (!left-or* (!right-and* X) (not p))))
                           p)))))))
    ((->modus-ponens p if-p-then-q)
     (let ((f (ax2nd if-p-then-q))
           (g (ax2nd p)))
       (mu ()
           (!mp* (!f) (!g)))))))
```

This function makes use of a number of auxiliary methods. For instance, `right-and*` handles the common task of obtaining a projection arrow and composing it with another. E.g., instead of `(!compose (!right-and a b) f)` for an arrow `f` with codomain `(and a b)`, we may simply use `(!right-and* f)`.

`ax2nd` is used within a natural-deduction method `%axiomatically` and the deductive form based on it, `axiomatically`. These can be used in a natural deduction proof to incorporate automatically axiomatic proofs.

```
(define (%axiomatically ax-method)
  ;; Temporarily redefine current-category to return an
  ;; instance of the axiomatic category, and call ax-method
  ;; within the definition, applying ax2nd to the result
  ;; to produce a natural deduction method.  Then, after
  ;; restoring the original category, invoke the natural
  ;; deduction method.
  (!(let ((cc (current-category))
          (ax (edu.rpi.cs.tayloj.fluid.calculi.impl.AxiomaticPCImpl.)))
      (define (current-category) ax)
      (try/finally
       (ax2nd (!ax-method))
       (define (current-category) cc)))))

(define-macro (axiomatically form env)
  (destructuring-bind (_ . body) form
    `(!%axiomatically (mu () ,@body))))
```

Using `axiomatically` we can compare the arrows of the axiomatic category with the arrows of the natural deduction category. The functions `in-ax` and `in-nd` change the current category, and `ax-derivation` and `nd-derivation` show the derivation of the arrow:

```
> (in-ax)                         # switch to the axiomatic category
> (!then1 p q)                    # an instantiation of the then1 schema
(->then1 TRUE (if p (if q p)))

> (in-nd)                         # switch to the natural deduction category
> (axiomatically (!then1 p q))  # the translation of the then1 schema
(->Curry TRUE (if p (if q p)))

> (nd-derivation (axiomatically (!then1 p q))) # its derivation
(->Curry TRUE (if p (if q p)))
  (->Curry (and TRUE p) (if q p))
    (->Composite (and (and TRUE p) q) p)
      (->Composite (and (and TRUE p) q) (and TRUE p))
        (->Identity (and (and TRUE p) q) (and (and TRUE p) q))
        (->LeftProjection (and (and TRUE p) q) (and TRUE p))
      (->RightProjection (and TRUE p) p)
```

Remark 9 (No Simplification of Arrows). In the present implementation, there is no simplification of equivalent arrows. For instance, the arrow $\pi\text{id}: (\top\&p)\&q \rightarrow \top\&p$ is not simplified to $\pi: (\top\&p)\&q \rightarrow \top\&p$. There are two reasons for this: (i) it is not particularly important for the present effort, especially as proof simplification and proof normalization using DPLs has been discussed elsewhere (e.g., Arkoudas, 2005b); and (ii) it is easier to guarantee termination of algorithms that consider arrows case by case when their forms are unique. The second point is particularly important in categories in which double elimination arrows are isomorphisms. Such categories collapse to preorders; there is at most one arrow $A \rightarrow B$ for any objects A and B (Low, 2011). By ignoring the equivalences among arrows, we greatly simplify the task of proof mapping.

3.5.5 Cross-Category Reasoning

The implementation of `axiomatically` in the previous section can be generalized to other types of cross-category reasoning for fluid logics. The general procedure seems to be:

1. Change the current category $\mathscr{C}$ to a new category $\mathscr{D}$.

2. Call some $\mathscr{D}$ method m to derive an arrow f.

3. Call a mapping function k with f to produce a $\mathscr{C}$ method g.

4. Change the current category back to $\mathscr{C}$.

5. Call g to produce the mapping of f.

This general procedure is easily implementable in the language and makes for seamless switches between fluid logics. From the categorical perspective, it is of interest because k may implement a $\mathscr{D} \rightarrow \mathscr{C}$ functor, or may implement a non-functor mapping. It has the additional advantage that when used to capture reasoning with different "contexts," there is typically more than one possible mapping between contexts.

For instance, the traditional natural-deduction rule of conditional introduction is based on the deduction theorem which states that if some conclusion C is derivable from the assumption B, then the conditional $B \supset C$ is derivable without the assumption of B. The deduction theorem is proved by demonstrating a general mapping from a proof with an assumption to a proof without the assumption. But there may well be numerous mappings that would suffice to demonstrate the deduction theorem. These mappings may have different proof-theoretic properties (e.g., one mapping may produce a proof linear in the size of the original, while another produces one quadratic in the size of the original). By explicitly providing a place for the mapping function k, this approach to cross-category reasoning is an excellent foundation for exploring proof mappings with fluid logics.

3.6 Completing Natural Deduction with a Deduction Theorem

In the previous section, we called the λ, or `curry`, method for the natural deduction calculus `assume`. This choice was based on the λ arrows capture the natural deduction rule of conditional introduction by "discharging" an assumption. Even so, obtaining an arrow $\lambda f : A \to B \supset C$ in a single step from an arrow $f : A \& B \to C$ is a very different process from assuming B and deriving C. A much more natural approach is an `assume` form (like the one given in Example 39 (p. 50)). In the categorical setting, adding an assumption A to a category $\mathscr{C}$ produces the category $\mathscr{C}[x]$ which is just like $\mathscr{C}$, but with the additional arrow $x : \top \to A$. For cartesian closed categories, for any arrow $f : B \to C$ in $\mathscr{C}[x]$, there is a unique arrow corresponding $f' : B \& A \to C$ in $\mathscr{C}$. By currying this arrow, we may obtain $\lambda(f') : B \to A \supset C$.

It is this property that we use to implement a `assume` form. Rather than implement the deduction theorem as stated above, we implement a variant which is actually a functor. (The version above is not a functor, as it does not have a consistent object mapping. For instance, it maps $f : B \to C$ to $f' : B \& A \to C$, but would also map $g : C \to B$ to $g' : C \& A \to B$. A functor must always map an object to the same object, but this mapping takes B in to $B \& A$ in the domain and to B in the codomain.) The functor maps each $\mathscr{C}[x]$ arrow $f : B \to C$ to a $\mathscr{C}$ arrow $f' : B \& A \to C \& A$. It is then easy to obtain $\lambda(\pi f') : B \to A \supset C$.

We begin with a functional implementation, `%assume`, that works when the current category implements `HasAdjoin`, which provides a method to obtain a category that implements `HasIndeterminate` (recall § 3.4.3.2 (p. 63) for these interfaces). This is an instance of the general process described in § 3.5.5 (p. 71); `m` is a $\mathscr{C}[x]$ method and `k` is deduction theorem mapping.

```
(define (%assume p d k)
  (dlet ((m (let* ((c (current-category))
                   (cx (.adjoin c p)))
              (define (current-category) cx)
              (try/finally
               (k (!d (!assumption)))
               (define (current-category) c)))))
   (!discharge (!left-and* (!m)))))
```

Now we need only choose a particular deduction theorem mapping, `dtm`, and we can define an `assume` form:

```
(define (%assume p d k)
  (dlet ((m (let* ((c (current-category))
                   (cx (.adjoin c p)))
              (define (current-category) cx)
              (try/finally
               (k (!d (!assumption)))
               (define (current-category) c)))))
   (!discharge (!left-and* (!m)))))
```

We now develop a functor F from $\mathscr{C}$ to $\mathscr{C}[x : \top \to A]$. Because $\mathscr{C}$ and $\mathscr{C}[x]$ have so much structure in common, this is actually much simpler than the mapping from the axiomatic category to the natural deduction category. This mapping is also a functor, as it preserves identities and composites. $\mathscr{C}$ and $\mathscr{C}[x]$ have exactly the same objects, the functor maps each $\mathscr{C}[x]$ object X to $X \mathbin{\&} A$ in $\mathscr{C}$. Identities and composites are simple:

$$F(\mathrm{id}_B) = \mathrm{id}_{F(B)} = \mathrm{id}_{B\&A} \qquad\qquad F(gf) = F(g)F(f)$$

For any arrow f that $\mathscr{C}$ and $\mathscr{C}[x]$ have in common (e.g., projection arrows, injection arrows, etc.) mapping is also very simple:

$$\begin{aligned} F(\pi) &= \pi \times \mathrm{id}_A & F(\pi') &= \pi' \times \mathrm{id}_A \\ F(\iota) &= \iota \times \mathrm{id}_A & F(\iota') &= \iota' \times \mathrm{id}_A \\ F(\mathrm{dn}) &= \mathrm{dn} \times \mathrm{id}_A & F(\mathrm{eval}) &= \mathrm{eval} \times \mathrm{id}_A \end{aligned}$$

Each of the remaining arrows types (viz., product arrows, coproduct arrows, and curry arrows) depend on other arrows that are "shorter" proofs. For $\mathscr{C}[x]$ arrows $f : B \to C$ and $g : B \to D$, the product arrow $\langle f, g\rangle : B \to C \mathbin{\&} D$ is mapped as follows by applying F to each of f and g to obtain $F(f) : B \mathbin{\&} A \to C \mathbin{\&} A$ and $F(g) : B \mathbin{\&} A \to D \mathbin{\&} A$. These can be combined in a number of ways to produce a $B \mathbin{\&} A \to (C \mathbin{\&} D) \mathbin{\&} A$ arrow, and we arbitrarily choose:

$$F(\langle f, g\rangle) = \langle\langle \pi F(f), \pi F(g)\rangle, \pi'\rangle : B \mathbin{\&} A \to (C \mathbin{\&} D) \mathbin{\&} A$$

Coproduct arrows are slightly more complicated, and make use of curry arrows. For $\mathscr{C}[x]$ arrows

$f : B \to D$ and $g : C \to D$, there is a coproduct arrow $[f, g] \colon B \vee C \to D$. The arrows f and g are mapped to $F(f) \colon B \,\&\, A \to D \,\&\, A$ and $F(g) \colon C \,\&\, A \to D \,\&\, A$. Each of these may be curried, giving $\lambda F(f) \colon B \to A \supset (D \,\&\, A)$ and $\lambda F(g) \colon C \to A \supset (D \,\&\, A)$. The coproduct of these, combined with id_A, and composed with an appropriate eval arrow provides $F([f, g])$:

$$F([f,g]) = \mathrm{eval}([\lambda F(f), \lambda F(g)] \times \mathrm{id}_A) \colon (B \vee C) \,\&\, A \to D \,\&\, A$$

The final case is a $\mathscr{C}[x]$ arrow $\lambda f : B \to C \supset D$ for a $\mathscr{C}[x]$ arrow $f : B \,\&\, C \to D$. F maps f to $F(f) \colon (B \,\&\, C) \,\&\, A \to D \,\&\, A$. Using properties of products, we can easily obtain:

$$\pi F(f) \langle \langle \pi\pi, \pi\pi' \rangle, \pi'\pi' \rangle \colon (B \,\&\, A) \,\&\, C \to D$$

Currying this provides:

$$\lambda(\pi F(f) \langle \langle \pi\pi, \pi\pi' \rangle, \pi'\pi' \rangle) \colon B \,\&\, A \to C \supset D,$$

from which we obtain:

$$F(\lambda f) = \langle \lambda(\pi F(f) \langle \langle \pi\pi, \pi\pi' \rangle, \pi'\pi' \rangle), \pi' \rangle \colon B \,\&\, A \to (C \supset D) \,\&\, A.$$

This mapping is implemented in the language by:

```
(define (dtm x)
  ;; A deduction theorem mapping to be called in category 𝒞[x],
  ;; (with assumption x: ⊤ → A) with a 𝒞[x] arrow
  ;; f: B → C.  Returns a method to be called in 𝒞
  ;; that will derive an arrow B&A → C&A.  (From that
  ;; arrow, it's trivial to derive an arrow B → A ⊃ C.)
  (let* ((a (codomain (!assumption)))
         (-xA (mu (f) (!times f (!identity a)))))
    (match
     x
     ((->true-intro b)  (mu () (!-xA (!true-intro b))))
     ((->false-elim b)  (mu () (!-xA (!false-elim b))))
     ((->left-and b c)  (mu () (!-xA (!left-and b c))))
     ((->right-and b c) (mu () (!-xA (!right-and b c))))
     ((->left-or b c)   (mu () (!-xA (!left-or b c))))
     ((->right-or b c)  (mu () (!-xA (!right-or b c))))
     ((->dn b)          (mu () (!-xA (!dn b))))
     ((->mp b c)        (mu () (!-xA (!mp b c))))
     ((->identity p)
      (mu () (!identity (and p a))))
     ((->compose g f)
      (let ((gm (dtm g))
            (fm (dtm f)))
        (mu () (!compose (!gm) (!fm)))))
     ((->both f g)
      (let ((fm (dtm f))
            (gm (dtm g)))
        (mu ()
            (dlet* ((ff (!fm))
                    (gg (!gm)))
             (!both (!both (!left-and* ff)
                           (!left-and* gg))
                    (!right-and* ff))))))
     ((->cd f g)
      (let ((fm (dtm f))
            (gm (dtm g)))
        (mu ()
            (dlet ((h (!-xA (!cd (!discharge (!fm))
                                 (!discharge (!gm))))))
             (!mp* (!left-and* h)
                   (!right-and* h))))))
     ((->discharge f)
      (let ((fm (dtm f)))
        (mu ()
            (dlet ((w (!fm)))
             (dmatch w
              ((-> (and (and b c) a) (and d a))
               (dlet* ((x (!identity (and (and b a) c)))
                       (y (!discharge
                           (!left-and*
                            (!compose
                             (!fm)
                             (!both (!both (!left-and* (!left-and* x))
                                           (!right-and* x))
                                    (!right-and* (!left-and* x))))))))
                      (!both y (!right-and b a)))))))))
     ((->assumption a)
      (mu ()
          (!both (!right-and TRUE a)
                 (!right-and TRUE a))))
     ;; any other arrow must be an indeterminate from an ancestor, so
     ;; it should be safe to claim it, since dtm should only be called
     ;; from within an (assume ...).
     ((-> _ _)
      (mu ()
          (!-xA (!claim x)))))))
```

With a proper assume form, we can derive theorems involving conditionals much more naturally. For instance, a proof of the theorem $(\sim a \vee b) \supset (a \supset b)$:

```
(define (example)
  ;; ⊤ → (∼a ∨ b) ⊃ (a ⊃ b)
  (assume ((or (not a) b) w)
   (!cd** w
          (assume ((not a) x)
           (assume (a y)
            (!false-elim* (!mp* x y) b)))
          (assume (b x)
           (assume (a _)
            (!claim x))))))

; > (!example)
; (->Curry TRUE (if (or (not a) b) (if a b)))
```

3.7 Comparing Translations

We now have two mappings into the natural deduction propositional calculus. One maps from axiomatic logic, and the other from a slightly different natural deduction propositional calculus (one extended with a hypothesis). Both the axiomatic and extended calculi can be used to prove the same theorems, but the mappings may produce different arrows in the natural deduction category (without hypotheses).

The function `nd-derivation` reconstructs the derivation of an arrow. We can use it to compare the translation strategies of `axiomatically` and `assume`. For instance, consider the theorem $b \supset (a \vee b)$:

```
> (nd-derivation (axiomatically (!or2 a b)))
(1 (->Curry TRUE (if b (or a b))))
  (2 (->Composite (and TRUE b) (or a b)))
    (3 (->RightInjection b (or a b)))
    (4 (->RightProjection (and TRUE b) b))

> (nd-derivation (assume (b x) (!right-or* x a)))
(1 (->Curry TRUE (if b (or a b))))
  (2 (->Composite (and TRUE b) (or a b)))
    (3 (->LeftProjection (and (or a b) b) (or a b)))
    (4 (->Composite (and TRUE b) (and (or a b) b)))
      (5 (->Product (and b b) (and (or a b) b)))
        (6 (->Composite (and b b) (or a b)))
          (7 (->RightInjection b (or a b)))
          (8 (->LeftProjection (and b b) b))
        (9 (->Composite (and b b) b))
           (10 (->Identity b b))
           (11 (->RightProjection (and b b) b))
      (12 (->Product (and TRUE b) (and b b)))
        (13 (->RightProjection (and TRUE b) b))
        (13 (->RightProjection (and TRUE b) b))
```

Let us consider these in the now-familiar tree-based notation. The first, which makes use of the manually defined `axiomatically` translation procedure, is a relatively straightforward derivation:

$$\dfrac{\dfrac{\pi'\colon \top \mathbin{\&} b \to b \qquad \iota'\colon b \to a \vee b}{\pi'\iota'\colon \top \mathbin{\&} b \to a \vee b}}{\lambda(\pi'\iota')\colon \top \to b \supset (a \vee b)}$$

Of course, the translation used by `axiomatically` is specifically designed to translate the axioms of the axiomatic propositional calculus, so this translation is essentially written by the creator of the translation.

The second is surprisingly complex, and big enough that we should break it into parts. The first part, in which we derive an arrow $\top \mathbin{\&} b \to (a \vee b) \mathbin{\&} b$:

$$\dfrac{\dfrac{\pi'\colon \top \mathbin{\&} b \to b \qquad \pi'\colon \top \mathbin{\&} b \to b}{\langle \pi', \pi' \rangle\colon \top \mathbin{\&} b \to b \mathbin{\&} b} \qquad \dfrac{\dfrac{\pi\colon b \mathbin{\&} b \to b \qquad \iota'\colon b \to a \vee b}{\iota'\pi\colon b \mathbin{\&} b \to a \vee b} \qquad \dfrac{\pi'\colon b \mathbin{\&} b \to b \qquad \mathrm{id}_b\colon b \to b}{\mathrm{id}_b\pi'\colon b \mathbin{\&} b \to b}}{\langle \iota'\pi, \mathrm{id}_b\pi' \rangle\colon b \mathbin{\&} b \to (a \vee b) \mathbin{\&} b}}{\langle \iota'\pi, \mathrm{id}_b\pi' \rangle\langle \pi', \pi' \rangle\colon \top \mathbin{\&} b \to (a \vee b) \mathbin{\&} b} \tag{3.17}$$

Reusing this derivation as a lemma, we may complete the proof:

$$\dfrac{\dfrac{\overset{\vdots\ (3.17)}{\langle \iota'\pi, \mathrm{id}_b\pi' \rangle\langle \pi', \pi' \rangle\colon \top \mathbin{\&} b \to (a \vee b) \mathbin{\&} b} \qquad \pi\colon (a \vee b) \mathbin{\&} b \to a \vee b}{\pi\langle \iota'\pi, \mathrm{id}_b\pi' \rangle\langle \pi', \pi' \rangle\colon \top \mathbin{\&} b \to a \vee b}}{\lambda(\pi\langle \iota'\pi, \mathrm{id}_b\pi' \rangle\langle \pi', \pi' \rangle)\colon \top \to b \supset (a \vee b)}$$

This proof uses the arrows present in a bicartesian closed category, and the translation embedded in the `assume` form. Most of the sentences in (3.17) are conjunctions that have b as a right conjunct. This is a result of the mapping from $\mathscr{C}[\top \to b]$ to $\mathscr{C}$ used in the implementation of `assume` in `(assume (b x)`

`(!right-or* x a)).`

Alternatively, we may consider the structure of the commutative diagrams from which these arrows could be extracted. The first proof corresponds to the rightmost arrow of the following diagram.

$$\begin{tikzcd}
\top \mathbin{\&} b \arrow[d, "\pi'"] & \top \arrow[dd, "\lambda(\iota'\pi')"] \\
b \arrow[d, "\iota'"] & \\
a \lor b & b \supset (a \lor b)
\end{tikzcd} \tag{3.18}$$

This proof takes a very direct approach and does not carry any "extra" information any father than necessary.

The second proof takes a somewhat more complicated approach, but its shape gives some insight into how the `assume` form carries its assumption through a proof.

$$\begin{tikzcd}
 & & \top \mathbin{\&} b \arrow[dl, "\pi'"'] \arrow[d, "\langle \pi', \pi' \rangle"] \arrow[dr, "\pi'"] & & & \top \arrow[dddd, "\lambda(\pi\langle \iota'\pi, \mathrm{id}_b\pi' \rangle \pi')"] \\
 & b & b \mathbin{\&} b \arrow[l, "\pi"'] \arrow[r, "\pi'"] \arrow[dl, "\pi"'] \arrow[dr, "\pi'"] \arrow[dd, "\langle \iota'\pi, \mathrm{id}_b\pi' \rangle"] & b & & \\
 & b \arrow[dl, "\iota'"'] & & b \arrow[dr, "\mathrm{id}_b"] & & \\
a \lor b & & (a \lor b) \mathbin{\&} b \arrow[ll, "\pi"] \arrow[rr, "\pi'"'] \arrow[d, "\pi"] & & b & \\
 & & a \lor b & & & b \supset (a \lor b)
\end{tikzcd} \tag{3.19}$$

The left side of the commutative diagram takes on a shape akin to a cartoon "Christmas tree," and we can observe that nodes on the top of the tree and on the tips of the branches on the left side are exactly the same as the nodes in the left hand chain of (3.18), viz.: $\top \mathbin{\&} b$, b and $a \lor b$. The tips of the right hand branches simply carry the assumption b through. Noting that $\mathrm{id}_b\pi' = \pi'$, we can even observe that every sloped arrow in the right hand branches will be π'.

3.8 Summary

Recall the problem statement from the introduction (§ 1.2 (p. 3)) and the stated goal to:

> Develop a formal system for: (i) specifying logical systems and the interactions and relationships between them and (ii) proof construction using multiple logical systems; formally

analyze said system; implement in software.

The approach and results demonstrated in this dissertation achieve these goals. The new framework and approach described herein are formal in nature, and provide the mechanisms for specifying logical systems using denotational proof languages, and more importantly, the categorical denotational proof languages that are the foundation of fluid logics. The interactions and relationships between logical systems is suitably captured in fluid logics by the categorical construction of functors and the types of mappings that we have demonstrated. We have demonstated that this approach, as implemented in software permits proof construction using multiple logical systems, and that the results proofs are amenable to formal analysis.

More specifically, in this chapter we have presented a portable and pragmatic implementation of a programming language based on the $\lambda\mu$-calculus, and an accompanying "standard library." We have shown that this programming language can be used to realize traditional denotational proof languages, and to greatly simplify their implementation (e.g., the MIU-system was implemented in fewer than fifty lines). By designing and implementing Java interfaces corresponding to the structures of category theory, we can just as easily implement categorical denotational proof languages. We demonstrated that we can relatively easily in the language implement both functor and non-functor mappings (viz.: the axiomatic to natural-deduction propositional calculus mapping and the deduction theorem functor mapping) and that these can be used as the "behind-the-scenes" implementation of syntactic constructions for seamlessly connecting different proof systems. Manual inspection and analysis of the resulting proofs can provide insight into the nature of these mappings.

In the final chapter, we dicuss future directions for this research and possible applications.

Chapter 4

Discussion & Future Work

In this brief chapter, we will discuss some future research and development that will make the initial work on fluid logics described in this dissertation more useful, as well as some potential application areas.

4.1 Slate

Slate (Bringsjord et al., 2008) is a software system for argument and proof construction developed over a number of years by several researchers affiliated with the Rensselaer Artificial Intelligence (AI) and Reasoning (RAIR) Laboratory. Slate, described in more detail in Appendix F (p. 155), is currently used in introductory logic courses at Rensselaer, and supports students in learning a variety of formal logics, including the propositional calculus, first-order predicate calculus, and several modal logics.

Through the progression of a course, students are often permitted to use reasoning “oracles” that serve as “shortcuts” over material covered earlier in the course. For instance, after the propositional calculus has been covered and students are learning first-order predicate calculus, they may be allowed to appeal to a propositional calculus oracle to apply, for instance, De Morgan’s Law, which is essentially propositional, in a proof for a proof in first-order logic.

Many of the oracles in Slate are based on the principle of a lossy translation from the sentences of one language into the language of another, and the application of automated reasoning procedures. For instance, a propositional calculus oracle might be applied to the first-order inquiry, “does $\forall x P(x)$ follow from $(\forall x P(x)) \,\&\, (\exists y Q(y))$?” In such a case, the premise is translated to $P_1 \,\&\, P_2$ and the query to P_1, and the oracle which can reason using only propositional calculus, would respond in the affirmative.

In the cases where the target logic is, in a sense, contained within the source logic, mapping the formulae in the proof in the target logic back to their counterparts in the source logic. For more complicated translations, however, this is not possible. For instance, when working with a modal logic, an oracle could translate premises and a conclusion into the first-order encoding of the Kripke semantics dicussed in Example 2 (p. 6). (Indeed, this is what the current implementation of Slate does.) An automated reasoner for first-order logic can then confirm or deny whether the conclusion follows from the premises, but obtaining a proof in the modal logic requires proof translation that the present work can enable. Figure F.7 (p. 163) shows how Slate currently displays the proof in first-order logic that justifies the modal logic oracle's claim; the ability to show a proof in the modal logic would be a great improvement.

We envision using the present work to augment the Slate system and provide much better feedback to students from the reasoning oracles. Students will benefit from intelligent courseware that can respond with more than a simple "correct" or "incorrect," but with explanations based in the logical system they are learning.

4.2 Natural Deduction for the Cognitive Event Calculus

The Cognitive Event Calculus ($\mathcal{CEC}$) (Arkoudas & Bringsjord, 2009) is a multi-modal first-order logic extension of the event calculus for reasoning about the knowledge and beliefs of multiple agents. The event calculus in $\mathcal{CEC}$ is standard, after the fashion of the original event calculue (Kowalski & Sergot, 1986), but the properties of the cognitive modalities is rather novel.

Epistemic and doxastic modal logics introduce modal operators to form formulae such as $K_i\phi$ and $\mathbf{B}_j\psi$, interpreted as "i knows that ϕ" and "j believes that ψ," respectively. In keeping with other modal logics, typical epistemic and doxastic logics incorporate logical omniscience: if an agent knows two propositions, then the agent also knows all of their consequences. This is a convenient fiction for logicians, but is not cognitively defensible. Arkoudas & Bringsjord's (2009) $\mathcal{CEC}$ earns the term "cognitive" by limiting the reasoning abilities of agents. For instance, this may mean that the $\mathcal{CEC}$ includes as a theorem every instance of modus ponens:

$$K_\alpha(\phi \supset \psi) \supset (K_\alpha\phi \supset K_\alpha\psi)$$

but not of modus tollens:

$$K_\alpha(\phi \supset \psi) \supset ((K_\alpha \sim\psi) \supset (K_\alpha \sim\phi))$$

We strongly expect that a natural-deduction proof calculus for the $\mathcal{CEC}$ could be constructed using modal subproofs that combine a weak, but cognitively plausible, logical system with suppositional reasoning. The cognitively plausible logical system could feature a logical language with cognitively based constraints (e.g., a maximum depth on subformulae, disallowing certain combinations of connectives,

&c..) with a proof system with cognitively oriented inference rules (e.g., including modus ponens and special cases of other inference rules). Such modal subproof would capture in a natural fashion the types of cognitive reasoning that is available in the $\mathcal{CEC}$, and could translated back into the axiomatic approach specified by the $\mathcal{CEC}$. Thus the calculus could be both more amenable the working logician, but still provably equivalent to the original.

4.3 A Richer Type System

While the intent of the present work was always to develop a framework for categorical denotational proof systems, the process of implementing the system has made it clear that no extension to the $\lambda\mu$-calculus is necessary for to do this; it is sufficient to adopt arrows are the fundamental "propositions" of a language, and the rest falls into place nicely. The use of Java interfaces and classes system to specify categorical structures ensures the ability to reuse methods in different logics, and with Java is useful in a more pragmatic sense due to the abundance of existing software written in Java.

The use of Java's type system to define categorical structures has come at a cost, however. As we approach more advanced categorical structures, Java's type system is not expressive enough to capture all the necessary properties. For instance, a monoidal category is a category $\mathscr{C}$ equipped with binary functor $\otimes : \mathscr{C} \times \mathscr{C} \to \mathscr{C}$ that satisfies several criteria (e.g., that $A \otimes (B \otimes C) \cong (A \otimes B) \otimes C$, though the specifics do not concern us here). A typical classical logic with both conjunction and disjunction gives rise to two monoidal categories: one with the functor & and one with the functor $\vee$. A Java class cannot implement a type-parameterized interface with different types, and so we cannot express in Java that, for instance, a bicartesian closed category is a monoidal category in two different ways.

Even in the cases where Java's type system allows us to express some of the constraints that should be imposed on a structure, it typically cannot be used to add other axiomatic constraints. For instance, we cannot impose that composition of an arrow f with an appropriate identity arrow yields f.

In the future, it may be worthwhile to implement a richer type systems in the new system rather than co-opting Java's type system to perform a task beyond its capabilities.

4.4 Idiomatic Programming

We have developed the new system following in the style of Arkoudas's (2005a) Athena, but with several significant departures. Rather than implementing special deductive forms, we added a general macro facility. In lieu of implementing a new denotational proof language for each selection of categories of interest, we simulated a dynamically scoped global variable keeping track of the current category. Significantly, rather than implementing categories and arrows (arrows are the analogue of propositions in Athena), we pushed arrow definition into the host language.

The biggest difference in "day-to-day" programming, however, is that the evaluation of a deduction now produces a proof, rather than a proposition. Athena proofs tend to be somewhat linear, in that a typical workflow is to get enough prerequisite propositions into the assumption base and only then evaluate a deduction that depends on them. Working "backward" is difficult in Athena; there is no convenient way to complete the final steps of a proof and then complete the beginning, except for deriving a series of conditionals and chaining them together. In our approach we are actually composing the ultimate proof from smaller proofs, and the order in which the smaller proofs are constructed is not important.

As an example, consider conjunction elimination. Athena has one right-and rule:

$$\beta \cup \{\, \phi \,\&\, \psi \,\} \vdash \mathbf{dapp}(\mathsf{right\text{-}and}, \phi \,\&\, \psi) \rightsquigarrow \psi$$

Our approach has one right-and rule as well, though it does not require any particular arrows to be in the assumption base:

$$\beta \vdash \mathbf{dapp}(\mathsf{right\text{-}and}, \phi, \psi) \rightsquigarrow \pi' : \phi \,\&\, \psi \to \psi$$

This primitive method works, but in practice we do not use it much, instead opting for two related rules *right-and and right-and*, each of which derives the appropriate π' projection arrow, but additionally composes it with another arrow (on the left or right, respectively):

$$\beta \cup \{ f : \psi \to \rho \,\} \vdash \mathbf{dapp}(\mathsf{*right\text{-}and}, \phi, f) \rightsquigarrow f\,\pi' : \phi \,\&\, \psi \to \rho$$
$$\beta \cup \{ f : \rho \to \phi \,\&\, \psi \,\} \vdash \mathbf{dapp}(\mathsf{right\text{-}and*}, f) \rightsquigarrow \pi' f : \rho \to \psi$$

That is, *right-and prepends to an existing proof, and right-and* appends to an existing proof. If we use only methods that append to existing proofs, our derivations tend to be very similar to corresponding derivations in Athena.

We have found this to be a useful technique in practice, but our experience with the system is still in its initial phases. It may turn out that other approaches may be better suited to some tasks. For instance, it may be more helpful to write deductive forms containing a number of derivations that are automatically composed. There are many possibilities to explore.

4.5 Improved Java Interoperability

The present language supports a limited form of compilation to an intermediate representation which is subsequently interpreted. In the long term, it may be beneficial to implement a proper compiler targeting the Java Virtual Machine. This would lead to a significant performance increase, but more importantly, it could allow interoperability in the form of Java code calling our system.

At present, the runtime of the language is exposed in very limited ways to Java code. A `eval` method can be used to evaluate snippets of code in our runtime (and this has proved very helpful in writing automated unit tests), but no "native" calls are possible. Such a compiler and interoperability layer would be a significant undertaking, but may prove necessary for any serious integration with other Java software (such as future versions of Slate).

Bibliography

Anderson, K., Hickey, T., & Norvig, P. (2001). *Reference manual for Jscheme*. Retrieved from http://jscheme.sourceforge.net/jscheme/doc/refman.html. (Date Last Accessed November 6, 2014.)

Arkoudas, K. (2000). *Denotational proof languages*. (Unpublished doctoral dissertation). Massachusetts Institute of Technology, Cambridge, MA.

Arkoudas, K. (2001a). *Type-alpha DPLs* (MIT AI Laboratory Memo No. AIM-2001-025). Cambridge, MA: Massachusetts Institute of Technology.

Arkoudas, K. (2001b). *Type-omega DPLs* (MIT AI Laboratory Memo No. AIM-2001-027). Cambridge, MA: Massachusetts Institute of Technology.

Arkoudas, K. (2004). Specification, abduction, and proof. In W. Fern (Ed.), Proceedings from ATVA 2004: *The Second International Symposium on Automated Technology for Verification and Analysis* (pp. 294–309). Berlin, Germany: Springer-Verlag.

Arkoudas, K. (2005a, March 29). *An Athena tutorial*. Retrieved from http://people.csail.mit.edu/kostas/dpls/athena/athenaTutorial.pdf. (Date Last Accessed November 6, 2014.)

Arkoudas, K. (2005b). Simplifying proofs in Fitch-style natural deduction systems. *Journal of Automated Reasoning*, *34*(3), 239–294.

Arkoudas, K., & Bringsjord, S. (2009). Propositional attitudes and causation. *International Journal of Software and Informatics*, *3*(1), 47–65.

Awodey, S. (2006). *Category theory*. New York, NY: Oxford University Press.

Baader, F., Calvanese, D., McGuinness, D. L., Nardi, D., & Patel-Schneider, P. F. (Eds.) (2003). *The description logic handbook: Theory, implementation and applications*. New York, NY: Cambridge University Press.

Barwise, J., & Etchemendy, J. (1995). Heterogeneous logic. In B. Chandrasekaran, J. Glasgow, & N. H. Narayanan (Eds.) *Diagrammatic reasoning: cognitive and computational perspectives* (pp. 211–234). Menlo Park, CA: AAAI Press.

Barwise, J., & Etchemendy, J. (1996). Heterogeneous logic. In Allwein, G., & Barwise, J. (Eds.) *Logical reasoning with diagrams* (pp. 179–200). New York, NY: Oxford University Press.

Barwise, K. J., & Etchemendy, J. (1994). *Hyperproof*. Stanford, CA: CSLI Press.

Bechofter, S., van Harmelen, F., Hendler, J., Horrocks, I., McGuinness, D. L., Patel-Schneider, P. F., & Stein, L. A. (2004, February 10). *OWL: Web ontology language reference* (W3C Recommendation). Cambridge, MA: W3C. Retrieved from http://www.w3.org/TR/owl-ref/. (Date Last Accessed November 6, 2014.)

Bertot, Y., & Castéran, P. (2004). *Interactive theorem proving and program development: Coq'Art: The calculus of inductive constructions*. New York, NY: Springer-Verlag.

Bierman, G. M., & de Paiva, V. C. V. (2000). On an intuitionistic modal logic. *Studia Logica*, *65*(3), 383–416.

Bringsjord, S., Arkoudas, K., Clark, M., Shilliday, A., Taylor, J., Schimanski, B., & Yang, Y. (2007). Reporting on some logic-based machine reading research. In Proceedings from *The 2007 AAAI Spring Symposium on Machine Reading* (pp. 23–28). Menlo Park, CA: AAAI Press.

Bringsjord, S., Shilliday, A., Taylor, J., Bello, P., Yang, Y., & Arkoudas, K. (2006). Harnessing intelligent agent technology to 'superteach' reasoning. *International Journal of Technology in Teaching and Learning*, *2*(2), 88–116.

Bringsjord, S., & Taylor, J. (2014). *Logic: A modern approach*. Unpublished manuscript. Departments of Cognitive Science and Computer Science, Rensselaer Polytechnic Institute, Troy, NY.

Bringsjord, S., Taylor, J., Shilliday, A., Clark, M., & Arkoudas, K. (2008). Slate: An argument-centered intelligent assistant to human reasoners. In F. Grasso, N. Green, R. Kibble, & C. Reed (Eds.) Proceedings from CMNA'08: *The Eighth International Workshop on Computational Models of Natural Argument* (pp. 1–10). Retrieved from http://www.cmna.info/CMNA8/programme/CMNA8-Bringsjord-etal.pdf. (Date Last Accessed November 6, 2014.)

Brouwer, L. E. J. (1925). Intuitionistische zerlegung mathematischer grundbegriffe. *Jahresbericht der Deutschen Mathematiker-Vereinigung*, *33*, 251–256.

Central Intelligence Agency. (2003). *Iraqi mobile biological warfare agent production plants* (General Report) Washington, DC: Central Intelligence Agency.

Chappell, A., Bringsjord, S., Shilliday, A., Taylor, J., & Wright, W. (2007). *Integration experiment with GeoTime, Slate, and VIKRS* (Conference Handout). Troy, NY: Rensselaer AI & Reasoning Laboratory, Rensselaer Polytechnic Institute.

Cheikes, B. A. (2006). *MITRE support to IKRIS* (MITRE Technical Report No. MTR060158) McLean, VA: The MITRE Corporation.

Chellas, B. F. (1980). *Modal logic: An introduction*. New York, NY: Cambridge University Press.

Chisholm, R. M. (1989). *Theory of knowledge* (3rd ed.). Englewood Cliffs, NJ: Prentice Hall.

Claessen, K., & Sorensson, N. (2003). New techniques that improve MACE-style finite model finding. In P. Baumgartner, & C. Fermueller (Eds.) Proceedings from the CADE-19 Workshop: *Model Computation — Principles, Algorithms, Applications* (pp. 11-27). Retrieved from http://www.cs.miami.edu/~geoff/Conferences/CADE/Archive/CADE-19/WS4/04.pdf. (Date Last Accessed November 6, 2014.)

Clark, M., Shilliday, A., Werner, D., & Bringsjord, S. (2007). *A primer on the use of arguments in Slate: Analyzing Pearl Harbor in late 1941* (Rensselaer AI & Reasoning Laboratory Technical Report). Troy, NY: Rensselaer AI & Reasoning Laboratory, Rensselaer Polytechnic Institute.

Dou, D., & McDermott, D. (2006). Deriving axioms across ontologies. In Proceedings from AAMAS'06: *The Fifth International Joint Conference on Autonomous Agents and Multiagent Systems* (pp. 952–954). New York, NY: ACM Press.

Dou, D., McDermott, D., & Qi, P. (2005). Ontology translation by ontology merging and automated reasoning. In *Ontologies for agents: Theory and experiences* (pp. 73–94). Berlin, Germany: Birkhäuser Basel.

Fikes, R. E., Ferrucci, D., & Thurman, D. A. (2005). Knowledge Associates for Novel Intelligence. In Proceedings from IA 2005: *The 2005 International Conference on Intelligence Analysis*. McLean, VA: The MITRE Corporation.

Fitch, F. B. (1952). *Symbolic logic: An introduction*. New York, NY: The Ronald Press Company.

Fitch, F. B. (1966). Natural deduction rules for obligation. *American Philosophical Quarterly*, *3*(1), 27–38.

Genesereth, M. R., & Fikes, R. E. (1997). *Knowledge Interchange Format version 3 reference manual* (Logic Group Technical Report No. Logic-92-1). Stanford, CA: Stanford Logic Group, Stanford University.

Gentzen, G. (1935). Untersuchungen über das logische Schließen. *Mathematische Zeitschrift*, *39*(1), 176–210, 405–431.

Goldblatt, R. (1984). *Topoi: The categorical analysis of logic*. Amsterdam, The Netherlands: Elsevier.

Hawthorn, J. (1990). Natural deduction in normal modal logic. *Notre Dame Journal of Formal Logic*, *31*(2), 263–273.

Hayes, P. (2006). *IKL guide* (IHMC Technical Report, IKRIS Interoperability Group Report). Pensacola, FL: Florida Institute for Human & Machine Cognition. Retrieved from http://www.ihmc.us/users/phayes/IKL/GUIDE/GUIDE.html. (Date Last Accessed November 6, 2014.)

Hayes, P., & Menzel, C. (2006). *IKL specification document* (IHMC Technical Report, IKRIS Interoperability Group Report). Pensacola, FL: Florida Institute for Human & Machine Cognition. Retrieved from http://www.ihmc.us/users/phayes/IKL/SPEC/SPEC.html. (Date Last Accessed November 6, 2014.)

Hofstadter, D. R. (1979). *Gödel, Escher, Bach: An eternal golden braid*. New York, NY: Vintage Books.

Hughes, F. J. (2003). *The art and science of the process of intelligence analysis: Case study #4, the sign of the crescent* (Training Case Study). Washington, DC: Joint Military Intelligence College.

Hustadt, U., Schmidt, R. A., & Georgieva, L. (2004). A survey of decidable first-order fragments and description logics. *Journal of Relational Methods in Computer Science*, *1*, 251–276.

ISO/IEC. (2007). *ISO/IEC 24707:2007(E): Information technology — Common Logic (CL): a framework for a family of logic-based languages*. Geneva, Switzerland: ISO/IEC.

Johnson, S. D., Barwise, J., & Allwein, G. (1996). Toward the rigorous use of diagrams in reasoning about hardware. In Allwein, G., & Barwise, J. (Eds.) *Logical reasoning with diagrams* (pp. 201–224). New York, NY: Oxford University Press.

Kapler, T., & Wright, W. (2005). GeoTime information visualization. *Information Visualization*, *4*(2), 136–146.

Kelsey, R., Clinger, W., Rees, J., Abelson, H., Dybvig, R. K., Haynes, C. T., ... Wand, M. (1998). Revised^5 report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, *11*(1), 7–105.

Kent (2005). *A tradecraft primer: Structured analytic techniques for improving intelligence analysis* (Tradecraft Review Volume 2, Number 2). Reston, VA: Sherman Kent School, Kent Center for Analytic Tradecraft.

Kolmogorov, A. N. (1967). On the principle of excluded middle. In J. van Heijenoort (Ed. & Trans.) *From Frege to Gödel: A source book in mathematical logic, 1879–1931* (pp. 414–437). Cambridge, MA: Harvard University Press. (Original work published 1925.)

Konyndyk, K. (1986). *Introductory modal logic*. Notre Dame, IN: University of Notre Dame Press.

Kowalski, R., & Sergot, M. (1986). A logic-based calculus of events. *New Generation Computing*, *4*(1), 67–94.

Kripke, S. A. (1963). Semantical considerations on modal logic. *Acta Philosophica Fennica*, *16*, 83–94.

Lambek, J. (1968). Deductive systems and categories i: Syntactic calculus and residuated categories. *Mathematical Systems Theory*, *2*(4), 287–318.

Lambek, J. (1989). On some connections between logic and category theory. *Studia Logica*, *48*(3), 269–278.

Lambek, J., & Scott, P. J. (1988). *Introduction to higher order categorical logic*. New York, NY: Cambridge University Press.

Low, Z. L. (2011). Bicartesian closed categories and logic. Presented at *The Cambridge University Part III Seminars Michaelmas in Logic and Category Theory*. Retrieved from http://zll22.user.srcf.net/talks/2011-12-01-CategoricalLogic.pdf. (Date Last Accessed November 6, 2014.)

Łukasiewicz, J. (1948). The shortest axiom of the implicational calculus of proposition. *Proceedings of the Royal Irish Academy, Section A: Mathematical and physical sciences*, *52*(3), 25–33. Reprinted as Łukasiewicz (1970).

Łukasiewicz, J. (1970). The shortest axiom of the implicational calculus of proposition. In L. Borowski (Ed.) *Jan Łukasiewicz, selected works* (pp. 295–305). Amsterdam, The Netherlands: North-Holland.

McCune, W. (2003). *Otter 3.3 reference manual* (Argonne National Laboratory Technical Memorandum 263). Argonne, IL: Argonne National Laboratory.

Newell, A., Shaw, J. C., & Simon, H. A. (1958). Elements of a theory of human problem solving. *Psychological Review*, *45*(3), 151–166.

Newell, A., & Simon, H. A. (1956). *The logic theory machine: A complex information processing system* (Technical Report No. P-868). Santa Monica, CA: Rand Corporation.

National Intelligence Council. (2007). *Iran: Nuclear intentions and capabilities* (National Intelligence Estimate). Washington, DC: National Intelligence Council.

Office of the Director of National Intelligence. (2008). *United States intelligence community information sharing strategy* (Report). Washington, DC: Office of the Director of National Intelligence.

Parigot, M. (1992). $\lambda\mu$-calculus: An algorithmic interpretation of classical natural deduction. In *Logic programming and automated reasoning* (pp. 190–201). Berlin, Germany: Springer Berlin Heidelberg.

Paulson, L. C., & Nipkow, T. (1994). *Isabelle: A generic theorem prover*. Berlin, Germany: Springer-Verlag.

Pollock, J. L. (1987). Defeasible reasoning. *Cognitive Science*, *11*(4), 481–518.

Pollock, J. L. (1992). How to reason defeasibly. *Artificial Intelligence*, *57*(1), 1–42.

Pollock, J. L. (1995). *Cognitive carpentry: A blueprint for how to build a person*. Cambridge, MA: MIT Press.

Prawitz, D. (1965). *Natural deduction: A proof-theoretical study*. Uppsala, Sweden: Almqvist & Wiksell.

Prawitz, D., & Malmnäs, P.-E. (1968). A survey of some connections between classical, intuitionistic and minimal logic. In H. A. Schmidt, K. Schütte, & H.-J. Thiele (Eds.) *Contributions to Mathematical Logic: Proceedings of the Logic Colloquium, Hannover 1966* (pp. 215–229). Amsterdam, The Netherlands: North-Holland.

Satre, T. W. (1972). Natural deduction rules for modal logic. *Notre Dame Journal of Formal Logic*, *13*(4), 461–475.

Shilliday, A., Bringsjord, S., Werner, D., & Clark, M. (2007a). *A Slate-based reconstruction of a CIA/DIA argument for Iraq having pre-invasion BW capability* (Rensselaer AI & Reasoning Laboratory Technical Report). Troy, NY: Rensselaer AI & Reasoning Laboratory, Rensselaer Polytechnic Institute.

Shilliday, A., Taylor, J., & Bringsjord, S. (2007b). Toward automated provability-based semantic interoperability between ontologies for the intelligence community. In K. S. Hornsby (Ed.) Proceedings from OIC-2007: *Ontology for the Intelligence Community: Towards Effective Exploitation and Integration of Intelligence Resources* (pp. 66–72). Aachen, Germany: CEUR-WS.org.

Shilliday, A., Taylor, J., Clark, M., & Bringsjord, S. (2010). Provability-based semantic interoperability for information sharing and joint reasoning. In L. Obrst, T. Janssen, & W. Ceusters (Eds.) *Ontologies and semantic technologies for intelligence* (pp. 109–128). Clifton, VA: IOS Press.

Siegel, N., Shepard, B., Cabral, J., & Witbrock, M. (2005). Hypothesis generation and evidence assembly for intelligence analysis: Cycorp's Noöscape application. In Proceedings from IA 2005: *The 2005 International Conference on Intelligence Analysis*. McLean, VA: The MITRE Corporation.

Siemens, D. F. (1977). Fitch-style rules for many modal logics. *Notre Dame Journal of Formal Logic*, *18*(4), 631–636.

Sowa, J. F., & Majumdar, A. K. (2003). Analogical reasoning. In A. de Moor, W. Lex, & B. Ganter (Eds.) Proceedings from ICCS 2003: *The Eleventh International Conference on Conceptual Structures: Conceptual Structures for Knowledge Creation and Communication* (pp. 16–36). New York, NY: Springer.

Stickel, M., Waldinger, R., & Chaudhri, V. (2000). *A guide to SNARK* (Technical Report). Menlo Park, CA: Artificial Intelligence Center, SRI International. Retrieved from http://www.ai.sri.com/snark/tutorial/. (Date Last Accessed November 6, 2014.)

Stickel, M., Waldinger, R., Lowry, M., Pressburger, T., & Underwood, I. (1994). Deductive composition of astronomical software from subroutine libraries. In Proceedings from CADE-12: *The Twelfth International Conference on Automated Deduction* (pp. 341–355). New York, NY: Springer.

Strzalkowski, T., Small, S., Hardy, H., Yamrom, B., Liu, T., Kantor, P., Ng, K. B., & Wacholder, N. (2005). HITIQA: A question answering analytical tool, In Proceedings from IA 2005: *The 2005 International Conference on Intelligence Analysis*. McLean, VA: The MITRE Corporation.

Suppes, P. (1957). *Introduction to logic*. Princeton, NJ: D. Van Nostrand.

Sussman, G. J., & Steele, G. L., Jr. (1998). Scheme: An interpreter for extended lambda calculus. *Higher Order Symbolic Computation*, *11*(4), 405–439.

Taylor, J. (2007). *Provability-based semantic interoperability between knowledgebases and databases via translation graphs* (Unpublished master's thesis). Rensselaer Polytechnic Institute, Troy, NY.

Taylor, J., Bringsjord, S., & Clark, M. (2010). *Getting started with Slate* (Rensselaer AI & Reasoning Laboratory Technical Report). Troy, NY: Rensselaer AI & Reasoning Laboratory, Rensselaer Polytechnic Institute.

Taylor, J., Clark, M., Shilliday, A., & Bringsjord, S. (2008). *On Slate's strength factors and the terms of likelihood and confidence used by the National Intelligence Council in National Intelligence Estimates* (Rensselaer AI & Reasoning Laboratory Technical Report). Troy, NY: Rensselaer AI & Reasoning Laboratory, Rensselaer Polytechnic Institute.

Taylor, J., Shilliday, A., & Bringsjord, S. (2007). Provability-based semantic interoperability via translation graphs. In J. L. Hainaut (Ed.) Proceedings from ONISW 2007: *The First International Workshop on Ontologies and Information Systems for the Semantic Web* (pp. 180–189). New York, NY: Springer.

Thurman, D. A., Chappell, A. R., & Welty, C. (2006). *Interoperable knowledge representation for intelligence support (IKRIS)* (MITRE Public Release Case No. 07-1111). McLean, VA: The MITRE Corporation.

Toulmin, S. E. (2003). *The uses of argument*. Cambridge, England: Cambridge University Press.

Whitehead, A. N., & Russell, B. (1927). *Principia mathematica*. Cambridge, England: Cambridge University Press.

Appendix A

Implementing the $\lambda\mu$-Calculus

The $\lambda\mu$-calculus is a very simple language, and we have taken a minimalist approach to its implementation, preferring to define higher level constructs using macros around the core of the language. The pure $\lambda\mu$-calculus was described in § 2.2.3 (p. 39), but the implemented language is necessarily more complex. We first implemented a purely functional core $\lambda\mu$-language, consisting of a reader, compiler, and evaluator, that has no provision for variable assignments, definitions, or mutable state. We then extended the core language with a top level environment, lexical bindings and assignment, a few special syntactic forms (e.g., `quote`), Java interoperability, and a direct conditional operator (i.e., a `cond` form that is implemented by a branch rather than Church-encoded booleans). This extension to the core we call the standard language. With Java interoperability in place, the standard language also has easy access to Java objects and thereby efficient record data types.

A.1 The Core Language

The core $\lambda\mu$-calculus is a useful building block for the implementation of the standard language, but is not a pragmatic $\lambda\mu$-calculus-based programming language. It has no primitive values, so no valid expressions can be written, and no primitive methods, so neither can any deductions be written. From an implementation standpoint, however, the core language implements the evaluation semantics of the $\lambda\mu$-calculus so that other languages based on it need not re-implement this core functionality. The lexical syntax and grammar of the core language are a subset of that provided by the standard language, and need not be described here.

A.2 The Standard Language

The standard language consists of the standard reader (i.e., the lexical processor that reads tokens from an input text, expands certain primitive types of syntactic sugar (e.g., `'object` for `(quote object)`) and constructs the list-based representation of the program text, standard compiler, and standard evaluator. We now describe the lexical syntax of the language and the evaluation semantics. The runtime "compiles" the source to an intermediate representation that is then interpreted. Parsing and compiling the $\lambda\mu$-calculus is somewhat more complicated than parsing and compiling pure λ-calculus, on account of the more complex syntax. The intermediate representation aids efficient evaluation, but is not novel; we will not discuss it.

A.2.1 Lexical Syntax

The lexical syntax of the standard language is similar to most languages in the Lisp family. Lists are delimited by parentheses, where a non-nil terminating atom may be written using a `dot`, as in `(a b . c)`, and strings are surrounded by double quotation marks (`"..."`). Parentheses, whitespace, quotation marks, and comments serve as lexeme delimiters. Comments are supported with the traditional syntax; a semicolon and any following characters on a line are comments. Sequences of non-delimiter characters are read and interned to produce symbols. One exception to the previous is support for traditional DPL notation (Arkoudas, 2000, § 8.6.1); the exclamation point terminates any previous character sequence, and simultaneously terminates its own sequence. That is, `(!prove)` is read as the list `(! prove)`, and `(!!!)` is read as the list `(! ! !)`. To support convenient quotation and quasiquotation, the following syntactic sugar is supported: `'object` is shorthand for `(quote object)`, `‘form` is shorthand for `(quasiquote form)`, and `,form` and `,@form` abbreviate `(unquote form)` and `(unquote-splicing form)`, respectively. Any character in the input stream may be escaped by preceding it with a backslash (\), and will not perform any special function in that position (e.g., `\;` does not introduce a comment, `\)` does not terminate a list).

In the core language, all character sequences are interned as symbols, so, for instance, the input `(x 23)` is a list of two symbols, the first with the name "x", and the second with the name "23". The standard language performs an additional step of processing, and recognizes several types of primitive values. If the sequence of characters can be parsed as an integral or floating point value, then that integral or floating point value is the literal in the source code. Otherwise, if the name has the form prescribed by JScheme's Java Dot Notation, then a corresponding function or class literal is used (see § A.3.1 (p. 101)).

A.2.2 Grammar

We have attempted to keep the grammar of our implemented language as close as possible to the grammar of the pure $\lambda\mu$-calculus, which we repeat here for convenience:

$$\begin{aligned} D &::= \mathbf{dapp}(E, \overrightarrow{M})\,\{\ \mid kwd_1(\overrightarrow{\Xi_1}) \mid \cdots \mid kwd_n(\overrightarrow{\Xi_n})\,\} \\ E &::= c \mid I \mid \mu\,\overrightarrow{I}.D \mid \lambda\,\overrightarrow{I}.E \mid \mathbf{app}(E, \overrightarrow{M}) \\ M &::= E \mid D \end{aligned}$$

The λ and μ expressions in this grammar are captured straightforwardly by the `lambda` and `mu` special forms, respectively. Identifiers are symbols, which are read according to the previous section. Constants, also described in the prior section, are numeric literals, strings, and the primitive Java Dot objects (functions implementing Java field accessors and methods, and class literals).

Function application is implemented in traditional fashion, `(E M...)`, rather than using an **app** operator. Instead of **dapp**, the exclamation point indicates method invocation, as in `(!  E M...)`. Since the exclamation point is given special lexical processing, the space preceding `E` is unnecessary, and the method application is typically written without it: `(!E M...)`.

While Arkoudas's grammar for the $\lambda\mu$-calculus includes provisions for syntactic deductions based on special keyword forms (e.g., $key_n(\overrightarrow{\Xi_n})$), the standard language's compiler performs macroexpansion, and we have come across no special deductive forms that cannot be realized through macroexpansion that produces expressions and deductions.

To support interoperability with Java, as well as mutable global and lexical environments, we do introduce several special forms, described in § A.2.4, that do not have direct counterparts in the pure $\lambda\mu$-calculus grammar.

A.2.3 Macroexpansion, Compilation, and Evaluation

The standard language compiler performs a transformation to a rudimentary intermediate representation. This representation corresponds almost directly the primitive language constructs (e.g., function application, method application, conditional execution), and does not merit much discussion here. The primary purpose of implementing an intermediate representation in what is, at the present time, a research language not excessively concerned with performance, is the efficient representation of lexical closures.

However, before translating the cons-based s-expressions into compiled form, the compiler performs macroexpansion, using macroexpansion functions. Macroexpansion functions are functions of two arguments, viz., the form to be expanded, and the lexical environment in which the expansion takes place.

When the compiler compiles a form whose `car` has a macro binding in the global environment, that is, a list whose first element is a symbol that the global environment maps to a macroexpansion function, the compiler calls the macroexpansion function with the entire form and with the current lexical environment. The macroexpansion function is responsible for returning the expansion of the form, which is simply another value. If the value returned by the macroexpansion function differs from the original form, the compiler repeats the macroexpansion process on the new form. Eventually, either the resulting form is a non-list, a list whose first element does not have a macro binding, or the macroexpansion function returns the same form, and the macroexpansion process terminates. (Technically speaking, bad macroexpansion functions could thwart the compilation process by always returning another macroexpandable form, but this is a pathological case.)

Macroexpansion ultimately reduces every form to one of five types: (i) literals; (ii) variables; (iii) function applications; (iv) method applications; and (v) special forms. Instances of the last case are described in the following section. The evaluator may be seen as a function, $\mathit{eval}\colon \mathit{value} \times \mathit{assumption} \times \mathit{environment} \to \mathit{value}$, mapping tuples of values, assumption bases, and environments to values.

literal Literals evaluate to themselves, and are returned by the evaluator as is; there is no dependency on *environment*. Literals include strings, numbers, and class literals in Java Dot notation. Other literal objects can be obtained using the `quote` special form.

variable Variables references are determined at compile time to be lexical or global references. If the variable is bound lexically, its value is retrieved from the lexical environment (a constant time lookup). Otherwise, the name refers to a binding in the global environment (which is based on a hash table, so lookup is typically constant, but slower than lookup in lexical environments). If a global variable is not bound in the global environment, the evaluator throws an exception. (This situation does not occur for lexical bindings, since all lexical binding constructs ensure that variables are bound to values.)

function application Each function application consists of a expression that must produce a function object and a (possibly empty) list of phrases whose values will be passed to the function object. The evaluator evaluates the function form first, then, from left to right, the phrases provided as arguments, each with the same assumption base and environment. After the function form and all argument phrases have been evaluated, the body of the function is evaluated under the same assumption base, and a lexical environment that includes the bindings of variables declared in the function's lambda-list to the appropriate values. (This is the realization of [R5] from Figure 2.2 (p. 42).) The binding of argument values to the variables declared in the function's lambda-list is discussed with the `lambda` special form.

method application Method applications are analogous to function applications, with the exception that the assumption base with which the method body is evaluated contains the values produced by the evaluation of every argument phrase that is a deduction. The method form and argument phrases are all evaluated with the same assumption base; only to the method body are additional lexical bindings and the extended assumption base visible. (This corresponds to [R7] from Figure 2.2 (p. 42).)

special forms The evaluation of special forms is described in the following section. Some of the forms are part of the essential core of the language (e.g., `lambda` and `mu`), while others support Java interoperability, or manipulation of the global or lexical environments.

Remark 10. Concerning Evaluation of Method Arguments In describing the evaluation of method applications above, we specified that in evaluating a method application in an assumption base β, we evaluate each argument phrase that is a deduction in β, and the results of these deductions are only available to the when evaluating the body of the method. The evaluation rule [R7] is actually less restrictive. Recall [R7]:

$$\frac{\beta \vdash D_i \rightsquigarrow S \quad \beta \cup \{S\} \vdash \mathbf{dapp}(E, M_1, \ldots, S, \ldots, M_k) \rightsquigarrow N}{\beta \vdash \mathbf{dapp}(E, M_1, \ldots, D_i, \ldots, M_k) \rightsquigarrow N}\ [\mathrm{R7}]$$

The semantics of [R7] actually permit evaluation of arguments in any order, and can allow the evaluation of an argument under an assumption base that contains results of other arguments already evaluated. Arkoudas (2000, §8.7) recognized this "ambiguity" and gave as an example that would lead to different results under different evaluation strategies the following example:

$$\{\sim\sim P, P \supset Q\} \vdash \mathbf{dapp}(\mathsf{both}, \mathbf{dapp}(\mathsf{mp}, P, P \supset Q), \mathbf{dapp}(\mathsf{dn}, \sim\sim P)) \rightsquigarrow \cdots$$

If the second argument, $\mathbf{dapp}(\mathsf{dn}, \sim\sim P)$, is evaluated first to produce P, then the first argument, $\mathbf{dapp}(\mathsf{mp}, P, P \supset Q)$, can be evaluated in the assumption base $\{\sim\sim P, P, P \supset Q\}$ and succeed. In fact, Arkoudas points out that for any particular evaluation strategy, there are examples such as this that will fail but would succeed under others. We hold with his conclusion:

> From a practical standpoint the issue is inconsequential. Just as languages such as Scheme or ML adopt an applicative-order strategy even though this might fail to accord with the formal semantics of the λ-calculus, in the same manner it is perfectly sensible for an implementation of the λμ-calculus to adopt a fixed argument-evaluation order for methods even if this might fail to accord with the formal semantics. In fact making an argument deduction contingent

on the result of a fellow argument deduction is patently bad style and very unlikely to be encountered in practice. (Arkoudas, 2000, § 8.7, p. 305)

A.2.4 Special Forms

In the tradition of the Lisp family of language, we define a number of special forms that cannot be defined in terms of other language constructs. The set of special forms is intended to be rather minimal, but there are more than what are present in most of the Lisp family of languages, for two reasons: (i) in languages based on the λ-calculus, lambda abstraction is sufficient to supply all variable bindings, but in the $\lambda\mu$-calculus, there must be abstraction over both expressions and deductions, so there are lambda expressions and mu expressions; and (ii) since the host language is based on Java, and because we wish to support Java interoperability for pragmatic reasons, our implementation needs constructs to support Java exceptions. Thus, we define ten special forms (`lambda`, `mu`, `throw`, `try/catch/finally`, `quote`, `cond`, `set`, `define`, `define-primitive-method`, and `define-macro`) and describe their syntax and semantics here.

A.2.4.1 lambda

```
lambda lambda-list expression
lambda-list ::= variable | ( variable* ) | ( variable+ . variable )
```

Lambda abstraction is present in the $\lambda\mu$-calculus, and lambda expressions are used in Lisp languages to denote anonymous functions. We adopt Scheme's convention for argument lists (Kelsey et al., 1998), wherein the first argument of a lambda expression may either be a variable, a proper list of variables, or an improper list of variables whose terminating atom is a variable. In the first case, the single variable is bound to a list of all arguments provided to the function. In the second case, the number of arguments the function is called with must be the same as the length of the lambda list. In the third case, the function must be called with at least as many arguments as are provided in the lambda list; the terminating variable is bound to a list containing any remaining arguments.

The `lambda` special form is stricter than the lambda form that is generally available in the language. Though it permits the flexible lambda lists described above, it may have only a single expression as its body. In the standard library, however, `lambda` is defined as a macro that can accept an arbitrary number of expressions in its body, and sequences them within a `seq` form. Macroexpansion functions are called with the entire form that is to be expanded, and can 'opt-out' of further macroexpansion by returning the unchanged form. The `lambda` macro does precisely this when the function body contains a single expression.

A.2.4.2 mu

```
mu mu-list deduction
mu-list ::= variable | ( variable* ) | ( variable+ . variable )
```

The special form `mu` is used to generate anonymous abstractions over deductions, which are called methods. The syntax of a mu expression is analogous to that of a lambda expression, with the exception that the body of a mu form must be a single deduction, not an expression. As is the case for lambda, the standard library defines a `mu` macro that accepts multiple deductions as its body, sequencing them with a `dseq` form.

A.2.4.3 throw

```
throw throwable
```

The `throw` special form is used to throw instances of the Java Throwable class, and is used for Java interoperability. Due to Java's strict typing, it is difficult for the standard language interpreter to throw the Throwable directly; instead, the interpreter throws an instance of WrappedEvaluatorException wrapping `throwable`. At the time of writing, it is somewhat unhelpful to use `throw`, but its counterpart, `try/catch/finally`, is more useful.

A.2.4.4 try/catch/finally

```
try/catch/finally expression catch-clause* finally-expression
catch-clause ::= ((class-literal*) catch-fn)
```

The `try/catch/finally` special form is used to execute code in a similar manner to Java's `try`, `catch`, and `finally` blocks. First, the `expression` is executed, and if it produces a value, then the `finally-expression` is executed. If the `finally-expression` throws an exception, then that exception is thrown from the `try/catch/finally` form. Otherwise, the value produced by `expression` is returned. If `expression` throws an exception, then each `catch-clause` is given the option to handle the exception, until one of the clauses does. A `catch-clause` handles the exception if the exception is an instance of the one of `class-literals`. If a matching `catch-clause` is found, its `catch-fn`, which should be a function that can be invoked with exactly one argument, is called with the exception. The result of `catch-fn` is the value returned by the `try/catch/finally` form. After the `catch-fn` returns, or if no matching `catch-clause` is found, or it `catch-fn` throws an exception, the `finally-expression` is executed. Finally, if a matching `catch-clause` was found and its `catch-fn` returned a value, then that value is returned. Otherwise, if there was a matching `catch-clause`, but its `catch-fn` threw an exception, that exception is thrown. If there was no matching `catch-clause`, then the exception thrown by `expression` is re-thrown.

A.2.4.5 quote

```
quote object
```

The `quote` special form produces the unevaluated `object`. As per tradition, the reader recognizes the shorthand `'object` for `(quote object)`.

A.2.4.6 cond

```
cond test-expr then-expr else-expr
```

The `cond` special form evaluates `test-expr` to produce a Java `Boolean` object (which may be cast from a Java `boolean`). If the value is `true`, then `then-expr` is evaluated and its value returned, otherwise `else-expr` is evaluated and its value returned. If the evaluation of any these expressions results in an exception being thrown, then that exception is thrown by the `cond` form.

A.2.4.7 set

```
set variable phrase
```

The `set` special form provides the update of lexical references. First, `phrase` is evaluated to produce a value, then that value is stored in `variable`. The presence of `set` in a program introduces mutable state in $\lambda\mu$-programs, and can significantly increase the complexity in reasoning about program semantics. It is expected that most code will not make much direct use of `set`, but may resort to it in order to achieve efficient implementations of algorithms that would be costly to evaluate otherwise.

A.2.4.8 define

```
define body
body ::= variable phase
       | (function-name . lambda-list) expression
       | (method-name . mu-list) deduction
```

The `define` special form is similar to `set`, but establishes bindings in the top level environment rather than a local lexical environment. In its first and simplest form, `define` assigns the value of `phrase` to the `variable`. This form can be used to define simple values, functions, and methods, as in

```
(define pi 3.14)
(define list (lambda ...))
(define prove (mu ...))
```

As a syntactic convenience, `define` also permits Scheme-style definitions for functions and methods:

```
(define (list . arguments) arguments)
(define (prove formula) ...)
```

In the case of a function definition, the first argument to `define` is a `cons` whose `car` is the name of the function and `cdr` is the lambda list of the function, and the following equivalence holds:

```
(define (name . lambda-list) body) == (define name (lambda lambda-list body)
```

Similarly, for method definition, the following equivalence holds:

```
(define (name . mu-list) body) == (define name (mu mu-list body))
```

This special syntax for function and method definitions is built into the compiler, and cannot be implemented as a macro, because the choice between whether the definition is a function definition or a method definition depends on the syntactic type of the `define`'s second argument (i.e., on whether it is an expression or a deduction), and this information is not available to macroexpansion functions.

Remark 11 (Nesting define). The `define` form always establishes bindings in the top-level environment. This a departure from Scheme, wherein nested definitions are local definitions. This is similar to Common Lisp, where `defun` always establishes bindings in the global environment.

A.2.4.9 define-primitive-method

The special form `define-primitive-method` is used to establish primitive method bindings in the global environment. It has the same syntax as `define`, but the value it establishes must be a function, i.e., a lambda expression. No counterpart for primitive functions is provided, as Java methods, invoked via Java interoperability, serve suitably for that purpose.

A.2.4.10 define-macro

The special form `define-macro` is used to define macroexpansion functions in the global environment. Its syntax is identical to that of `define`, but the value assigned to the name must be a function of exactly

Table A.1: The standard library defines a number of values, functions, macros, and match patterns. Most of these have counterparts in other Lisp-like languages, or to constructs in Athena.

Category	`Symbols`
I/O	`+out+, +err+`
Objects	`equals`
Boolean Logic	`true, false, ~, ||, &&`
Arithmetic	`+, -, *, /, %, ==, /=, <, <=, >, >=, 1+, 1-`
Bitwise Operators	`lognot, logand, logxor, logior`
Conses	`consp, atom, set-car, set-cdr`
Symbols	`intern, make-symbol, gensym, unintern, symbolp, symbol-name`
Lists	`endp, listp, foldl, foldr, every, some, notevery, notany, reverse, append, nthcdr, nth, member-if, member, list, list*, mapcar`
Association Lists	`pairlis, assoc, acons`
Functions	`complement`
Control Flow, Binding	`seq, dseq, progn, prog1, prog2, dprogn let, dlet, destructuring-bind, check, dcheck, dcond, try/catch, try/finally, try, dtry, foreach letrec, dletrec, nlet`
Pattern Matching	`match, dmatch, define-simple-match-expander, define-match-expander, define-match-alias`
Match Patterns	`equals, list, as, satisfies`
Arrays	`make-array, get-array, set-array`

two arguments, which should be the form to macroexpand and a lexical environment. The macroexpansion function must return another form, the macroexpansion of the form passed as the first argument. If the returned form is identical to the provided form, no further macroexpansion is performed. If the returned form is not identical to the provided form, then macroexpansion is performed on the returned form.

A.3 The Standard Library

Given `define` and `define-macro`, we have implemented a sizable standard library for our standard language. The best documentation for it is likely the `standard-lm.lib` code distributed with the language and included in the appendix (§ E.4 (p. 138)), but a brief summary of its contents is in order.

A.3.1 Java Dot Interoperability

In our standard language we have adopted JScheme's Java Dot Notation to support Java interoperability (Anderson et al., 2001, § Java Access). This syntax provides convenient and easy access to Java classes, objects, methods, and fields. The Java Dot Notation provides special syntax for the following types of references: (i) Java classes; (ii) constructors; (iii) static fields; (iv) instance fields; (v) static methods; and (vi) instance methods. When the reader postprocesses a symbol obtained from the lexer and determines whether it should be interpreted as a number (e.g., if all the characters in its name are numerals), symbol, or a Java Dot entity, it uses the following definitions to recognize Java Dot entities.

Java classes Java class literals are identified by fully qualified class names with a ".class" suffix. For instance, when the reader encounters the text `java.lang.String.class`, it returns the class literal for the Java `String` class.

constructors Constructors are identified by a final dot ".", as in `java.lang.String.`, though the particular constructor that will be called depends on the number of arguments provided to an invocation. The reader returns a form that will evaluate to a function that will find an invoke a suitable constructor for the provided arguments.

static and instance fields Static and instance field accessors are identified by a final dollar sign. If the first character of the identifier is a dot, then the identifier is an instance field accessor, and everything between is the name of the field. Otherwise it is a static field accessor, and the rest of the identifier is the fully qualified class name followed by a dot and the name of the field. For instance, `.foo$` is a function that retrieves the `foo` field of an instance, and `com.example.Bar.foo$` is a function that accesses the static `foo` field of the class `com.example.Bar`. These accessor functions internally use Java reflection, so `.foo$` correctly retrieves the value of a field `foo` from any instance.

static and instance methods Instance methods are identified by an initial dot in their name, e.g., `.doSomething`. The value of `.doSomething` is a function that should be invoked with at least one argument (the instance), and will retrieve a suitable method from the instance, based on the number of arguments and their types, and reflectively invoke it. Static methods are identified by the presence of a dot anywhere in their name (after class literals, constructors, static and instance fields, and instance methods have been considered). For instance `java.lang.Math.abs` is a function that should be invoked with one argument. The appropriate static method is determined at runtime based on the type of the arguments.

Appendix B

Adjunctions

In this chapter we briefly discuss a construction that is of great importance in category theory, and that may play a greater role in future work on fluid logics: adjunctions.

B.1 Review of Functors and Natural Transformations

Recall from Definition 29 (p. 34) that a functor F from category $\mathscr{C}$ to $\mathscr{D}$, denoted $F\colon \mathscr{C} \to \mathscr{D}$ is a category homomorphism that maps $\mathscr{C}$ objects and arrows to $\mathscr{D}$ objects and arrows in a way that preserves identity arrows and arrow composition. Along with functors, we also recall natural transformations, which were introduced in Definition 32 (p. 38). A natural transformation $\eta\colon F \to G$ between functors $F, G\colon \mathscr{C} \to \mathscr{D}$ is a collection of $\mathscr{D}$ arrows indexed by $\mathscr{C}$ objects such that the following commutes:

$$\begin{tikzcd}
F(A) \arrow[r, "\eta_A"] \arrow[d, "F(f)"'] & G(A) \arrow[d, "G(f)"] \\
F(B) \arrow[r, "\eta_B"'] & G(B)
\end{tikzcd}$$

These concepts are both important in category theory, but for the present effort, natural transformations are not used, and functors, as objects in a programming language, are not used much. Functors encapsulate the notion of a structure preserving map between categories, and are a concept with which we are quite concerned, but we will take the approach of implementing such mappings as deductions that produce an arrow of a category given an arrow of another. That is, in the present effort, we are more concerned with the implementation of functors as deductions.

B.2 Adjunctions

Two functors may be related by a natural transformation, as described in § 2.2.2.4 (p. 38), but there are other types of relationships that may hold between functors (and categories) as well. One of the most important in category theory is that of an adjunction, or adjointness. There are many equivalent ways of formally defining adjunction, and more thorough definitions may be found in the literature. The definition given here is most closely related to that given by Goldblatt (1984, ch. 16).

Definition 42 (Adjunction). An adjunction between categories $\mathscr{C}$ and $\mathscr{D}$ is given by two functors $F\colon \mathscr{C} \to \mathscr{D}$ and $G\colon \mathscr{D} \to \mathscr{C}$ and two natural transformations

$$\eta\colon \mathrm{id}_{\mathscr{C}} \to G \circ F$$
$$\epsilon\colon F \circ G \to \mathrm{id}_{\mathscr{D}}$$

such that for any $\mathscr{C}$ object A and $\mathscr{D}$ object B, if there is a $\mathscr{C}$ arrow $g\colon A \to G(B)$, then there is a unique $\mathscr{D}$ arrow $f\colon F(A) \to B$ such that the following diagram commutes:

$$\begin{tikzcd}
A \arrow[r, dashed, "\eta_A"] \arrow[dr, "g"'] & G(F(A)) \arrow[d, "G(f)"] & & F(A) \arrow[d, dashed, "f"] \\
& G(B) & & B
\end{tikzcd}$$

and dually, for a $\mathscr{D}$ arrow $f\colon F(A) \to B$, there is a unique $\mathscr{C}$ arrow $g\colon A \to G(B)$ the following diagram commutes:

$$\begin{tikzcd}
G(B) & & F(G(B)) \arrow[r, dashed, "\epsilon_B"] & B \\
A \arrow[u, dashed, "g"] & & F(A) \arrow[u, "F(g)"] \arrow[ur, "f"'] &
\end{tikzcd}$$

The functor F is said to be left adjoint to G, and G right adjoint to F. The natural transformation η is called the unit of the adjunction, and ϵ the counit.

Goldblatt (1984, p. 438) provides a particularly illuminating diagram, the essence of which we recreate here in Figure B.1. An adjunction establishes a type of equivalence between two categories, and generalizes many categorical constructions.

Example 43 (Exponentials as Adjunctions). There is a noticeable similarity between the diagrams in Definition 42 and the diagrams in definition of adjunction. Indeed, it turns out that exponentials are

$$\begin{tikzcd}
\mathscr{C} & \mathscr{D} \\
A \arrow[r, "F"] \arrow[d, dashed] & F(A) \arrow[d, dashed] \\
G(B) & B \arrow[l, "G"]
\end{tikzcd}$$

Figure B.1: An illustration of the structure of an adjunction. For any $\mathscr{C}$ object A and $\mathscr{D}$ object B, there is a bijection between the $\mathscr{C}$ arrows from A to $G(B)$ and the $\mathscr{D}$ arrows from $F(A)$ to B. (This is only part of the structure of the adjunction; the natural transformations of the adjunction ensure that certain diagrams involving the corresponding arrows commute.) While F and G are names commonly used in presenting adjunction, there is some variation in which categories are called $\mathscr{C}$ and $\mathscr{D}$.

the range of functors that are right adjoint to product functors (to be defined). The rest of this example closely follows the structure used by Awodey (2006, Example 9.7, p. 187).

In a category $\mathscr{C}$ with (binary) products, for any object B, we may define a functor, typically denoted $- \times B \colon \mathscr{C} \to \mathscr{C}$, that takes each object A to $A \times B$ and each arrow $f : A \to C$ to $f \times \mathrm{id}_B \colon A \times B \to C \times B$. Note that $- \times B$ is an endofunctor: both its domain and codomain are $\mathscr{C}$.

Under what conditions does $- \times B$ have a right adjoint? Let us consider what implications the existence of an adjoint, which for the moment we will call G, would be:

- First, since $A \times B$ is $- \times B(A)$, for any arrow $f : A \times B \to C$, there must be an arrow $A \to G(C)$. Imitating Figure B.1:

$$\begin{tikzcd}
\mathscr{C} & \mathscr{C} \\
A \arrow[r, "- \times B"] \arrow[d, dashed] & A \times B \arrow[d, "f"] \\
G(C) & C \arrow[l, "G"]
\end{tikzcd} \tag{B.1}$$

- Second, for any $\mathscr{C}$ object C, $G(C)$ must also be a $\mathscr{C}$ object, and $\mathrm{id}_{G(C)} \colon G(C) \to G(C)$ a $\mathscr{C}$ arrow. This ensure that there is an arrow $- \times B(G(C)) \to C$, that is, $G(C) \times B \to C$. Again, imitating Figure B.1:

$$\begin{tikzcd}
\mathscr{C} & \mathscr{C} \\
G(C) \arrow[r, "- \times B"] \arrow[d, "\mathrm{id}_{G(C)}"'] & G(C) \times B \arrow[d, dashed] \\
G(C) & C \arrow[l, "G"]
\end{tikzcd} \tag{B.2}$$

If we denote the right adjoint as $\cdot^{N}$ rather than $G(\cdot)$, so that $G(C)$ is now C^B, we see that the dotted arrows in (B.1) and (B.2) are precisely the arrows $\lambda f : A \to C^B$ and $\mathrm{eval}_{C,B} : C^B \times B \to C$! It is worthwhile to consider the unit and counit of this adjunction:

- The counit of the adjunction between $- \times C$ and $\cdot^C$ is a natural transformation $\epsilon \colon \cdot^C \circ - \times C \to \mathrm{id}_{\mathscr{C}}$ whose component at A is the arrow $\mathrm{eval}_A \colon A^C \times C \to A$.

- The unit of the adjunction is a natural transformation $\eta \colon \mathrm{id}_{\mathscr{C}} \to - \times C \circ \cdot^C$ whose component at A is an arrow $A \to (A \times C)^C$, which is simply the $\lambda \mathrm{id}_{A \times C} \colon A \to (A \times C)^C$.

It is important to note that each exponential object X^Y is the value of the object mapping of $\cdot^Y$. To say that a category with products has exponentials means that there is a functor $- \times Y$ for *every* object Y in the category. Each exponential object X^Y is the value of the object mapping of the right adjoint of the functor $- \times Y$ at X. That a category has exponentials does not simply assert the existence of one functor with a right adjoint, but the existence of one functor with a right adjoint per object of the category.

Example 44 (First-Order Quantification as Adjunction). For a given first order language $\mathcal{L}$, let $\mathscr{L}(x_1, \ldots, x_n)$ be the category whose objects are the formulae of $\mathscr{L}$ that contain at most $\{x_1, \ldots, x_n\}$ free. For formulae A and B, there is an arrow $A \to B$ if and only if B is derivable from A.

There is a trivial inclusion functor $U \colon \mathscr{L}(x_1, \ldots, x_n) \to \mathscr{L}(x_1, \ldots, x_n, y)$ which maps each object to itself (since a formulae with at most $x_1, \ldots, x_n$ free also has at most $x_1, \ldots, x_n, y$ free), and each arrow to itself, since each derivation in $\mathscr{L}(x_1, \ldots, x_n)$ also holds in $\mathscr{L}(x_1, \ldots, x_n, y)$.

It turns out that U has both right and left adjoints, which we will consider in turn. First, consider what it would mean for U to have a right adjoint:

- For each formula ϕ in which at most $x_1, \ldots, x_n$ appear free, for any derivable formula ψ in which at most $x_1, \ldots, x_n, y$ appear there, there is a corresponding formulae ψ' in which at most $x_1, \ldots, x_n$ appear free. This suggests that the right adjoint to $U \colon \mathscr{L}(x_1, \ldots, x_n) \to \mathscr{L}(x_1, \ldots, x_n, y)$ must somehow bind any free occurrence of y in ψ to produce ψ'. Since ϕ contains no free occurrence of y, but ψ may, this suggests that the binding operation is $(\forall y)$. Indeed, the familiar inference rule:

$$\frac{\phi \to \psi}{\phi \to (\forall y)\psi}\ \forall I \qquad \text{where } y \text{ does not appear free in } \phi$$

describes the necessary bijection between arrows when we recall that $U(\phi)$ is ϕ:

$$\frac{U(\phi) \to \psi}{\phi \to (\forall y)\psi}$$

The unit of this adjunction is trivial, since U is the inclusion functor. It is simply the family of arrows $\mathrm{id}_{(\forall y)\mathrm{id}_A} = (\forall y)\mathrm{id}_A$. The counit is a sort of specialized universal elimination arrow: $(\forall y)\phi \to \phi$.

- Duality suggests the left adjoint of U is the existential quantifier, $(\exists y)$, but this worth confirming. The bijection between arrows is now of the form (recalling that ψ and $U(\psi)$ are the same formula, and in which y does not appear free).

  $$\frac{(\exists y)\phi \to \psi}{\phi \to U(\psi)} \qquad \text{where } y \text{ does not appear free in } \psi$$

  which is the familiar existential elimination rule. The unit of this adjunction is a family of specialized existential introduction arrows, $\psi \to (\exists y)\psi$, and the counit is the family of identity arrows, $\mathrm{id}_{(\exists y)\phi} = (\exists y)\mathrm{id}_\phi$.

Just as a category having exponentials requires the consideration of functors $- \times X$ and $\cdot^X$ for each object X, a category having universal and existential quantification requires consideration of a large number of functors.

A more detailed treatment of quantifiers as adjoints can be found in Awodey (2006, § 9.5, Quantifiers as adjoints) and Goldblatt (1984, Chapter 15, Adjointness and Quantifiers). We have not touched upon, for instance, how the various categories $\mathscr{L}(\cdots)$ are combined into a single category for the logic $\mathcal{L}$.

Example 45 (Binary Products as Adjunction). For a category $\mathscr{C}$, consider the product category $\mathscr{C} \times \mathscr{C}$ the objects of which are pairs of $\mathscr{C}$ objects, and the arrows of which are pairs of $\mathscr{C}$ arrows. For instance, given $\mathscr{C}$ arrows $f : A \to B$ and $g : C \to D$, the category $\mathscr{C} \times \mathscr{C}$ has an arrow $(f, g) : (A, C) \to (B, D)$. The diagonal functor $\Delta : \mathscr{C} \to \mathscr{C} \times \mathscr{C}$. maps each $\mathscr{C}$ object A to (A, A) and each arrow $f : A \to B$ to $(f, f) : (A, A) \to (B, B)$. Arrow composition is defined where the composites of the left and right arrows is defined. That is, $(h, i) \circ (f, g) : (A, B) \to (C, D)$ is defined if and only if $h \circ f : A \to C$ and $i \circ g : B \to D$ are.

Consider the implications of Δ having a right adjoint $\times$:

- For each pair of $\mathscr{C}$ objects A and B, there is a $\mathscr{C}$ object $\times(A, B)$, which we denote more conventionally by $A \times B$; and

- for each pair of arrows $f : C \to A$ and $g : C \to B$, there is a unique arrow $\times(f, g) : C \to \times(A, B)$, often represented as $\langle f, g \rangle$.

It is readily seen that the diagonal functor has a right adjoint if and only if the category has binary products. The projections are determined by the counit of the adjunction, which is a family of arrows of the form $(\pi, \pi') : (A \times B, A \times B) \to (A, B)$, and which determines the projections of the product. The unit, for an object A, is simply $\langle \mathrm{id}_A, \mathrm{id}_A \rangle : A \to A \times A$.

Appendix C

Distribution Arrows

Theorem 46 (Products Distribute over Coproducts)**.** *In a bicartesian closed category, for any objects A, B, and C, there is a canonical arrow* $(A+B)\times C \to (A\times C)+(B\times C)$*.*

Proof. There are injections,

$$\iota\colon A\times C \to (A\times C)+(B\times C) \qquad \text{and} \qquad \iota'\colon B\times C \to (A\times C)+(B\times C),$$

that can be curried to produce

$$\lambda\iota\colon A \to ((A\times C)+(B\times C))^C \qquad \text{and} \qquad \lambda\iota'\colon B \to ((A\times C)+(B\times C))^C,$$

which give rise to a product arrow,

$$[\lambda\iota,\lambda\iota']\colon A\vee B \to ((A\times C)+(B\times C))^C.$$

This can be combined with id_C to produce

$$[\lambda\iota,\lambda\iota']\times \mathrm{id}_C\colon (A\vee B)\times C \to ((A\times C)+(B\times C))^C\times C,$$

which can be composed with an appropriate eval arrow to give the distribution arrow,

$$\mathrm{eval}([\lambda\iota,\lambda\iota']\times \mathrm{id}_C)\colon (A\vee B)\times C \to (A\times C)+(B\times C). \qquad \square$$

Appendix D

Equivalence of Double Negation and Excluded Middle

The essential difference between the intuitionistic and classical propositional calculi is that the classical propositional calculus accepts the law of excluded middle, $\phi \vee \sim\phi$, and its equivalents, notably the principle of double negation elimination, $\sim\phi \supset \phi$, whereas the intuitionistic propositional calculus does not. Typical categorical constructions admit presentations of intuitionistic logics, and an additional arrow schema is needed. The obvious candidates have these forms:

$$\mathsf{dn}_\phi : \sim\sim\phi \to \phi \tag{D.1}$$

$$\mathsf{em}_\phi : \top \to \phi \vee \sim\phi \tag{D.2}$$

It turns out, as we should hope, that these approaches are equivalent. In categorical proofs, it is often more convenient to work with the former than with the latter arrows, but either can be derived from the other, as we show in this chapter.

Lemma 47 (Double Negation from Excluded Middle). *A bicartesian closed category with excluded middle arrows also has double negation elimination arrows.*

Proof. We proceed by using a proof by cases. We show that in a bicartesian closed category, we have arrows $\phi \to (\sim\sim\phi \supset \psi)$ and $\sim\phi \to (\sim\sim\phi \supset \phi)$. We can compose the corresponding corproduce arrow with an excluded middle arrow of the form $\top \to \phi \vee \sim\phi$, and so obtain an arrow $\top \to \sim\sim\phi \supset \phi$.

For every object ϕ in a category with products, there is a left projection arrow, $\pi \colon \phi \,\&\sim\sim\phi \to \phi$. In a category with exponential objects, this arrow can be curried, yielding one of our cases:

$$\lambda\pi\colon \phi \to {\sim}{\sim}\phi \supset \phi \tag{D.3}$$

For the other case, we recall that we have adopted ${\sim}\phi$ as an abbreviation for the exponential $\phi \supset \bot$. This abbreviation equates the two arrows: $\mathsf{eval}\colon ((\phi \supset \bot) \supset \bot)\&(\phi \supset \bot) \to \bot$ and $\mathsf{eval}\colon {\sim}{\sim}\phi \& {\sim}\phi \to \bot$. Then the following diagram commutes:

$$\begin{tikzcd} {\sim}\phi \,\&\, {\sim}{\sim}\phi \arrow[r, "\langle \pi', \pi \rangle"] & {\sim}{\sim}\phi \,\&\, {\sim}\phi \arrow[r, "\mathsf{eval}"] & \bot \arrow[r, "\bot_\phi"] & \phi \end{tikzcd}$$

which can be curried to give:

$$\begin{tikzcd} {\sim}\phi \arrow[r, "\lambda(\bot_\phi \mathsf{eval}\langle \pi', \pi \rangle)"] & {\sim}{\sim}\phi \supset \phi \end{tikzcd} \tag{D.4}$$

This arrow is the other case.

If a category has these arrows, as well as excluded middle arrows generated by (D.2), then the following diagram commutes.

$$\begin{tikzcd} & \top \arrow[d, "\text{(D.2)}"] & \\ \phi \arrow[r, "\iota_1"] \arrow[dr, "\text{(D.3)}"'] & \phi \vee {\sim}\phi \arrow[d, dashed, "{[\text{(D.3)}, \text{(D.4)}]}"] & {\sim}\phi \arrow[l, "\iota_2"'] \arrow[dl, "\text{(D.4)}"] \\ & {\sim}{\sim}\phi \supset \phi & \end{tikzcd} \tag{D.5}$$

Uncurrying the central downward pointing arrow yields an arrow $\top \,\&\, {\sim}{\sim}\phi \to \phi$, which we adjoin to a product diagram, yielding the following commutative diagram:

$$\begin{tikzcd} & {\sim}{\sim}\phi \arrow[dl, "\top_{\sim\sim\phi}"'] \arrow[d, dashed, "\langle \top_{\sim\sim\phi}, \mathrm{id}_{\sim\sim\phi} \rangle"] \arrow[dr, "\mathrm{id}_{\sim\sim\phi}"] & \\ \top & \top \,\&\, {\sim}{\sim}\phi \arrow[l, "\pi"] \arrow[r, "\pi'"'] \arrow[d, "\lambda^{-1}\text{(D.5)}"] & {\sim}{\sim}\phi \\ & \phi & \end{tikzcd} \tag{D.6}$$

from which we may read off an arrow ${\sim}{\sim}\phi \to \phi$. □

Lemma 48 (Excluded Middle from Double Negation). *A bicartesian closed category with double negation arrows also has excluded middle arrows.*

Proof. To show that a bicartesian closed category with double negation arrows also has excluded middle arrows, we first show that there is an arrow $\top \to \sim\sim(\phi \vee \sim\phi)$. Then, we compose that arrow with the double negation arrow to obtain an excluded middle arrow.

Recalling again that $\sim\phi$ abbreviates $\phi \supset \bot$, we observe the following arrow:

$$\sim(\phi \vee \sim\phi) \,\&\, \phi \xrightarrow{\mathsf{id}\times\iota} \sim(\phi \vee \sim\phi) \,\&\, \phi \xrightarrow{\mathsf{eval}} \bot$$

which can be curried to produce:

$$\sim(\phi \vee \sim\phi) \to \sim\phi$$

This, in turn, can be composed with an injection to yield:

$$\sim(\phi \vee \sim\phi) \to \phi \vee \sim\phi \tag{D.7}$$

Then the following diagram commutes:

$$\begin{tikzcd}
 & \top \,\&\, \sim(\phi \vee \sim\phi) \arrow[d, "\pi'"] & \\
 & \sim(\phi \vee \sim\phi) \arrow[dl, "\mathsf{id}"'] \arrow[d, dashed, "\langle \mathsf{id}, (D.7) \rangle"] \arrow[dr, "(D.7)"] & \\
\sim(\phi \vee \sim\phi) & \sim(\phi \vee \sim\phi) \,\&\, (\phi \vee \sim\phi) \arrow[l, "\pi"] \arrow[r, "\pi'"'] \arrow[d, "\mathsf{eval}"] & \phi \vee \sim\phi \\
 & \bot &
\end{tikzcd} \tag{D.8}$$

Then, finally, taking the central downward pointing arrow of (D.8), currying it, and composing with the double negation arrow of (D.1), we have an arrow $\top \to \phi \vee \sim\phi$:

$$\top \xrightarrow{\lambda(D.8)} \sim\sim(\phi \vee \sim\phi) \xrightarrow{(D.1)} \phi \vee \sim\phi \tag{D.9}$$

Thus, if a bicartesian closed category has double negation arrows, it has excluded middle arrows. □

Theorem 49. *A bicartesian closed category has arrows of the form $\sim\sim\phi \to \phi$ for every object ϕ if and only* *if it has arrows of the form* $\top \to \phi \vee \sim \phi$.

Proof. Straightforward from Lemmata 47 and 48. □

Remark 12 (Applicable to all Bicartesian Closed Categories). While we first encounter this theorem in handling the classical propositional calculusm note that Theorem 49 holds in *every* bicartesian closed category. The proof given here could be used as the basis for a reusable method that depends only the interfaces specified for bicartesian closed categories.

Appendix E

Implementation Source Code

In this chapter, we provide the source for the standard library, as well as some of the logical structures (e.g., the sentences of the propositional calculus), interoperability layers, and categorical denotational proof languages used herein.

E.1 Structures for the Propositional Calculus

This section shows the Java source of the data structures implementing the language of the propositional calculus, and the interoperability layer in the standard language that provides convenient access to these structures.

E.1.1 PropositionalCalculus.java

The `PropositionalCalculus` class defines a `Proposition` interface and uses an enumeration and factory methods to simulate algebraic data types. Aggressive object interning ensures that each propositional sentence is represented by a unique object (e.g., every occurrence of $A \vee B$ is the same object).

```java
package edu.rpi.cs.tayloj.fluid.dpl;

import java.util.Arrays;

import edu.rpi.cs.tayloj.fluid.dpl.PropositionalCalculus.Proposition.Type;
import edu.rpi.cs.tayloj.fluid.util.Interner;

/**
 * PropositionalCalculus provides interfaces and static classes
 * implementing the sentences of propositional calculus, and
 * factory methods for creating instances of these classes:
 */
public class PropositionalCalculus {

    /**
     * A marker interface that all propositions implement.  Propositions are
     * essentially an algebraic datatype, with each instance having a specified
     * type and a list of arguments.
     */
    public interface Proposition {
        /**
         * Returns the type of the proposition.
         * @return the type of the proposition
         */
        Type getType();

        /**
         * Returns the arguments of the proposition.
         * @return the arguments of the proposition
         */
        Object[] getArguments();

        /**
         * An enumeration of the types of Propositions.
         */
        enum Type {
            Variable("variable",1),
            Conjunction("and",2),
            Disjunction("or",2),
            Conditional("if",2);

            private final String operator;
            private final int arity;

            /**
             * Return the operator for the proposition type.
             * @return the operator for the type
             */
            private String getOperator() { return operator; }

            /**
             * Returns the arity of the proposition type.
             * @return the arity of the type
             */
            private int getArity() { return arity; }

            Type( String operator, int arity ) {
                this.operator = operator;
                this.arity = arity;
            }
        }
    }

    private static class AbstractProposition implements Proposition {
        private final Type type;
        private final Object[] arguments;
        private final String toString;

        private AbstractProposition( Type type, Object... arguments ) {
            if ( type.getArity() != arguments.length ) {
                throw new IllegalArgumentException( "propositional type "+type+
                        " requires "+type.getArity()+" arguments, but was provided "+
                        arguments.length+": "+Arrays.deepToString( arguments ));
            }
            this.type = type;
            this.arguments = Arrays.copyOf( arguments, arguments.length );
            StringBuilder sb = new StringBuilder();
            sb.append( '(' ).append( type.getOperator() );
            for ( Object o : arguments )  {
                sb.append( ' ' ).append( o.toString() );
            }
            this.toString = sb.append( ')' ).toString();
        }

        @Override
        public Type getType() { return type; }

        @Override
        public String toString() { return toString; }

        @Override
        public Object[] getArguments() { return arguments; }
    }

    /**
     * Propositional variables are the atomic sentences of the
     * propositional calculus.
     */
    public static class PropositionalVariable extends AbstractProposition {
        PropositionalVariable( String name ) {
            super( Type.Variable, name );
        }

        @Override
        public String toString() {
            return (String) getArguments()[0];
        }
    }

    /**
     * Binary propositions are those sentences that are composed
     * from two other propositions.  These are conjunctions,
     * disjunctions, and conditionals.  Biconditionals and exclusive
     * disjunctions would be binary propositions if they were
     * implemented.
     */
    public interface BinaryProposition extends Proposition {
        /**
         * Returns the left formula of the binary proposition.
         * @return the left formula
         */
        Proposition getLeft();

        /**
         * Returns the right formula of the binary proposition.
         * @return the right formula
         */
        Proposition getRight();
    }

    /**
     * An implementation of {@link BinaryProposition} that provides a constructor
     * and a toString method (that depends on the implementation of an abstract
     * method in classes that extend this class).
     */
    private static class AbstractBinaryProposition extends AbstractProposition implements
     BinaryProposition {
        private AbstractBinaryProposition( Type type, Proposition left, Proposition right
    ) {
            super( type, left, right );
        }

        @Override
        public Proposition getLeft() { return (Proposition) getArguments()[0]; }

        @Override
        public Proposition getRight() { return (Proposition) getArguments()[1]; }
    }

    /**
```

```
     * Conjunctions are sentences of the form "A and B".
     */
    public static class Conjunction extends AbstractBinaryProposition {
        private Conjunction( Proposition left, Proposition right ) {
            super( Type.Conjunction, left, right );
        }
    }

    /**
     * Disjunctions are sentences of the form "A or B".
     */
    public static class Disjunction extends AbstractBinaryProposition {
        private Disjunction( Proposition left, Proposition right ) {
            super( Type.Disjunction, left, right );
        }
    }

    /**
     * Conditionals are sentences of the form "if A then B" or
     * "A implies B".
     */
    public static class Conditional extends AbstractBinaryProposition {
        private Conditional( Proposition left, Proposition right ) {
            super( Type.Conditional, left, right );
        }

        /** (if p FALSE) is abbreviated (not p). */
        @Override
        public String toString() {
            if ( FALSE().equals( this.getRight() )) {
                return "(not "+this.getLeft().toString()+")";
            }
            else {
                return super.toString();
            }
        }
    }

    /**
     * An interner that creates propositions, ensuring that for equivalent
     * arguments, the same proposition is returned.
     */
    private final static Interner<Proposition> INTERNER = new Interner<Proposition>() {
        @Override
        public Proposition create(Object... key) {
            switch ( (Type) key[0] ) {
            case Variable:
                return new PropositionalVariable( (String) key[1] );
            case Conditional:
                return new Conditional( (Proposition) key[1], (Proposition) key[2] );
            case Conjunction:
                return new Conjunction( (Proposition) key[1], (Proposition) key[2] );
            case Disjunction:
                return new Disjunction( (Proposition) key[1], (Proposition) key[2] );
            default:
                throw new IllegalArgumentException( key[0]+" is not a PropositionType" );
            }
        }
    };

    /**
     * Returns the propositional variable with the given name.
     * @param name the name of the propositional variable
     * @return the propositional variable
     */
    public static PropositionalVariable variable( String name ) {
        return (PropositionalVariable) INTERNER.intern( Type.Variable, name );
    }
```

```java

    /**
     * Returns the negation of a formula.
     * @param formula a Proposition
     * @return the negation of the formula
     */
    public static Conditional not( Proposition formula ) {
        return (Conditional) INTERNER.intern( Type.Conditional, formula, FALSE() );
    }

    /**
     * Returns the propositional variable TRUE
     * @return the variable TRUE
     */
    public static PropositionalVariable TRUE() {
        return (PropositionalVariable) INTERNER.intern( Type.Variable, "TRUE" );
    }

    /**
     * Returns the propositional variable FALSE
     * @return the variable FALSE
     */
    public static PropositionalVariable FALSE() {
        return (PropositionalVariable) INTERNER.intern( Type.Variable, "FALSE" );
    }

    /**
     * Returns the conjunction of two formulas.
     * @param left the left formula
     * @param right the right formula
     * @return the conjunction of the formulas
     */
    public static Conjunction and( Proposition left, Proposition right ) {
        return (Conjunction) INTERNER.intern( Type.Conjunction, left, right );
    }

    /**
     * Returns the disjunction of two formulas.
     * @param left the left formula
     * @param right the right formula
     * @return the disjunctio of the formulas
     */
    public static Disjunction or( Proposition left, Proposition right ) {
        return (Disjunction) INTERNER.intern( Type.Disjunction, left, right );
    }

    /**
     * Returns the implication of two formulas.
     * @param left the antecedent (the "if" part)
     * @param right the consequent (the "then" part)
     * @return the conditional
     */
    public static Conditional implies( Proposition left, Proposition right ) {
        return (Conditional) INTERNER.intern( Type.Conditional, left, right );
    }
}
```

E.1.2 pc.lm

The standard language code in `pc.lm` defines the interoperability layer for interacting with the interfaces defined in `PropositionalCalculus.java`.

```
(define (variable name)
    (edu.rpi.cs.tayloj.fluid.dpl.PropositionalCalculus.variable name))

(define-macro (declare-variable form env)
    (destructuring-bind (_ name) form
      `(define ,name (variable ,(symbol-name name)))))

(define-simple-match-expander (variable x) (formula)
  `((.isInstance
    edu.rpi.cs.tayloj.fluid.dpl.PropositionalCalculus$PropositionalVariable.class
    ,formula)
    ,formula))

(define TRUE (edu.rpi.cs.tayloj.fluid.dpl.PropositionalCalculus.TRUE))
(define FALSE (edu.rpi.cs.tayloj.fluid.dpl.PropositionalCalculus.FALSE))

(define (not formula)
    (edu.rpi.cs.tayloj.fluid.dpl.PropositionalCalculus.not formula))

(define-simple-match-expander (not p) (formula)
    `((&& (.isInstance
    edu.rpi.cs.tayloj.fluid.dpl.PropositionalCalculus$Conditional.class ,formula)
          (.equals FALSE (.getRight ,formula)))
      (.getLeft ,formula)))

(define (and left right)
    (edu.rpi.cs.tayloj.fluid.dpl.PropositionalCalculus.and left right))

(define-simple-match-expander (and p q) (formula)
    `((.isInstance edu.rpi.cs.tayloj.fluid.dpl.PropositionalCalculus$Conjunction.class
    ,formula)
      (.getLeft ,formula)
      (.getRight ,formula)))

(define (if left right)
    (edu.rpi.cs.tayloj.fluid.dpl.PropositionalCalculus.implies left right))

(define-simple-match-expander (if p q) (formula)
    `((.isInstance edu.rpi.cs.tayloj.fluid.dpl.PropositionalCalculus$Conditional.class
    ,formula)
      (.getLeft ,formula)
      (.getRight ,formula)))

(define (or left right)
    (edu.rpi.cs.tayloj.fluid.dpl.PropositionalCalculus.or left right))

(define-simple-match-expander (or p q) (formula)
    `((.isInstance edu.rpi.cs.tayloj.fluid.dpl.PropositionalCalculus$Disjunction.class
    ,formula)
      (.getLeft ,formula)
      (.getRight ,formula)))
```

E.2 An Axiomatic Propositional Calculus Categorical DPL

In this section, we define the Java and standard language implementations of a categorical DPL for an axiomatic presentation of the propositional calculus. We reuse the implementation of the propositional structures provided in the previous section.

E.2.1 AxiomaticPCImpl.java

The `AxiomaticPCImpl` class implements the categorical structure of the axiomatic propositional calculus. This class does not implement many categorical interfaces, because the axiomatic presentation actually has little categorical structure (e.g., conjunctions are not products).

```java
package edu.rpi.cs.tayloj.fluid.calculi.impl;

import static edu.rpi.cs.tayloj.fluid.dpl.PropositionalCalculus.and;
import static edu.rpi.cs.tayloj.fluid.dpl.PropositionalCalculus.implies;
import static edu.rpi.cs.tayloj.fluid.dpl.PropositionalCalculus.not;
import static edu.rpi.cs.tayloj.fluid.dpl.PropositionalCalculus.or;
import edu.rpi.cs.tayloj.fluid.calculi.AxiomaticPC;
import edu.rpi.cs.tayloj.fluid.calculi.impl.AxiomaticPCImpl.Arrow.ArrowType;
import edu.rpi.cs.tayloj.fluid.dpl.PropositionalCalculus;
import edu.rpi.cs.tayloj.fluid.dpl.PropositionalCalculus.Conditional;
import edu.rpi.cs.tayloj.fluid.dpl.PropositionalCalculus.Proposition;
import edu.rpi.cs.tayloj.fluid.util.Interner;

/**
 * An implementation of the axiomatic propositional calculus.  This implementation
 * is somewhat "quick and dirty", using some dynamic typing internally for convenience.
 */
public class AxiomaticPCImpl
implements AxiomaticPC
<
Proposition, AxiomaticPCImpl.Arrow,
AxiomaticPCImpl.Arrow, AxiomaticPCImpl.Composite,
AxiomaticPCImpl.Arrow, AxiomaticPCImpl.Arrow, AxiomaticPCImpl.Arrow,
AxiomaticPCImpl.Arrow, AxiomaticPCImpl.Arrow, AxiomaticPCImpl.Arrow,
AxiomaticPCImpl.Arrow, AxiomaticPCImpl.Arrow, AxiomaticPCImpl.Arrow,
AxiomaticPCImpl.Arrow, AxiomaticPCImpl.Arrow,
AxiomaticPCImpl.ModusPonens
>
{
    /**
     * Arrows in the propositional calculus category.  These are simply
     * tuples containing the type of the arrow, the domain, and codomain.
     */
    public static class Arrow {
        private final ArrowType type;
        private final Proposition domain;
        private final Proposition codomain;
        /**
         * Returns an arrow with the specified type, domain, and codomain.
         * @param type
         * @param domain
         * @param codomain
         */
        public Arrow( ArrowType type, Proposition domain, Proposition codomain ) {
            this.type = type;
            this.domain = domain;
            this.codomain = codomain;
        }

        /**
         * Returns an arrow with the specified type and codomain, and the domain {@link
     PropositionalCalculus#TRUE()}.
         * @param type
         * @param codomain
         */
        public Arrow( ArrowType type, Proposition codomain ) {
            this( type, PropositionalCalculus.TRUE(), codomain );
        }

        @Override
        public String toString() {
            return "(->"+type+" "+domain+" "+codomain+")";
        }

        /**
         * An enumeration of the types of arrows in the axiomatic propositional
         * calculus category. This includes the identity and composite arrows.
         */
        static enum ArrowType {
            Identity, Composite("compose"),
            Then1, Then2,
            And1, And2, And3,
            Or1, Or2, Or3,
            Not1, Not2, Not3,
            ModusPonens("modus-ponens");

            private final String operator;

            ArrowType( String operator ) {
                this.operator = operator;
            }

            ArrowType() {
                this.operator = null;
            }

            @Override
            public String toString() {
                return operator != null ? operator : super.toString().toLowerCase();
            }
        };
    }

    /**
     * Composite arrows in the axiomatic propositional calculus.
     */
    public static class Composite extends Arrow {
        private final Arrow g, f;
        Composite( Arrow g, Arrow f ) {
            super( ArrowType.Composite, f.domain, g.codomain );
            if ( f.codomain != g.domain ) {
                throw new IllegalArgumentException( "Cannot compose arrows "+g+" and
     "+f+"." );
            }
            else {
                this.g = g;
                this.f = f;
            }
        }
    }

    /**
     * ModusPonens arrows in the axiomatic propositional calculus.
     */
    public static class ModusPonens extends Arrow {
        private final Arrow antecedent, conditional;
        ModusPonens( Arrow antecedent, Arrow conditional ) {
            super( ArrowType.ModusPonens, antecedent.domain,
     ((Conditional)conditional.codomain).getRight() );
            if ( antecedent.domain != PropositionalCalculus.TRUE() ||
                    conditional.domain != PropositionalCalculus.TRUE() ||
                    antecedent.codomain != ((Conditional)conditional.codomain).getLeft()
     ) {
                throw new IllegalArgumentException( "Cannot apply modus ponens to
     "+antecedent+" and "+conditional );
            }
```

```
            else {
                this.antecedent = antecedent;
                this.conditional = conditional;
            }
        }

        /**
         * Returns the arrow whose codomain is the antecedent.
         * @return the arrow whose codomain is the antecedent
         */
        public Arrow getAntecedent() { return antecedent; }

        /**
         * Returns the arrow whose codomain is the conditional.
         * @return the arrow whose codomain is the conditional
         */
        public Arrow getConditional() { return conditional; }
    }

    private final static Interner<Arrow> INTERNER =
            new Interner<AxiomaticPCImpl.Arrow>() {
                @Override
                public Arrow create(Object... key) {
                    switch ( (ArrowType) key[0] ) {
                    case Identity:
                        Proposition p = (Proposition) key[1];
                        return new Arrow( ArrowType.Identity, p, p );
                    case Composite:
                        Arrow g = (Arrow) key[1];
                        Arrow f = (Arrow) key[2];
                        return new Composite( g, f );
                    case And1:
                        p = (Proposition) key[1];
                        Proposition q = (Proposition) key[2];
                        return new Arrow( ArrowType.And1, implies(and(p,q),p) );
                    case And2:
                        p = (Proposition) key[1];
                        q = (Proposition) key[2];
                        return new Arrow( ArrowType.And2, implies(and(p,q),q));
                    case And3:
                        p = (Proposition) key[1];
                        q = (Proposition) key[2];
                        return new Arrow( ArrowType.And3, implies(p,implies(q,and(p,q))));
                    case Not1:
                        p = (Proposition) key[1];
                        q = (Proposition) key[2];
                        return new Arrow( ArrowType.Not1,
     implies(implies(p,q),implies(implies(p,not(q)),not(p))));
                    case Not2:
                        p = (Proposition) key[1];
                        q = (Proposition) key[2];
                        return new Arrow( ArrowType.Not2, implies(p,implies(not(p),q)));
                    case Not3:
                        p = (Proposition) key[1];
                        return new Arrow( ArrowType.Not3, or(p,not(p)));
                    case Or1:
                        p = (Proposition) key[1];
                        q = (Proposition) key[2];
                        return new Arrow( ArrowType.Or1, implies(p,or(p,q)));
                    case Or2:
                        p = (Proposition) key[1];
                        q = (Proposition) key[2];
                        return new Arrow( ArrowType.Or2, implies(q,or(p,q)));
                    case Or3:
                        p = (Proposition) key[1];
                        q = (Proposition) key[2];
                        Proposition r = (Proposition) key[3];
                        return new Arrow( ArrowType.Or3,
implies(implies(p,r),implies(implies(q,r),implies(or(p,q),r))));
                    case Then1:
                        p = (Proposition) key[1];
                        q = (Proposition) key[2];
                        return new Arrow( ArrowType.Then1, implies(p,implies(q,p) ));
                    case Then2:
                        p = (Proposition) key[1];
                        q = (Proposition) key[2];
                        r = (Proposition) key[3];
                        return new Arrow( ArrowType.Then2,
implies(implies(p,implies(q,r)),implies(implies(p,q),implies(p,r))));
                    case ModusPonens:
                        return new ModusPonens( (Arrow) key[1], (Arrow) key[2] );
                    default:
                        throw new IllegalArgumentException( key[0]+" is not a legal
ArrowType" );
                    }
                }
            };

    @Override
    public Arrow identity(Proposition object) {
        return INTERNER.intern( ArrowType.Identity, object );
    }

    @Override
    public boolean isIdentity(Object object) {
        return object instanceof Arrow && ((Arrow)object).type == ArrowType.Identity;
    }

    @Override
    public Composite compose(Arrow g, Arrow f) {
        return (Composite) INTERNER.intern( ArrowType.Composite, g, f );
    }

    @Override
    public boolean isComposite(Object object) {
        return object instanceof Arrow && ((Arrow)object).type == ArrowType.Composite;
    }

    @Override
    public Arrow compositeAfter(Composite h) {
        return h.g;
    }

    @Override
    public Arrow compositeBefore(Composite h) {
        return h.f;
    }

    @Override
    public Proposition domain(Arrow arrow) {
        return arrow.domain;
    }

    @Override
    public Proposition codomain(Arrow arrow) {
        return arrow.codomain;
    }

    @Override
    public boolean isArrow(Object object) {
        return object instanceof Arrow;
    }

    @Override
    public Arrow then1(Proposition a, Proposition b) {
        return INTERNER.intern( ArrowType.Then1, a, b );
    }

    @Override
    public Arrow then2(Proposition a, Proposition b, Proposition c) {
        return INTERNER.intern( ArrowType.Then2, a, b, c );
    }

    @Override
    public Arrow and1(Proposition a, Proposition b) {
        return INTERNER.intern( ArrowType.And1, a, b );
    }

    @Override
    public Arrow and2(Proposition a, Proposition b) {
        return INTERNER.intern( ArrowType.And2, a, b );
    }

    @Override
    public Arrow and3(Proposition a, Proposition b) {
        return INTERNER.intern( ArrowType.And3, a, b );
    }

    @Override
    public Arrow or1(Proposition a, Proposition b) {
        return INTERNER.intern( ArrowType.Or1, a, b );
    }

    @Override
    public Arrow or2(Proposition a, Proposition b) {
        return INTERNER.intern( ArrowType.Or2, a, b );
    }

    @Override
    public Arrow or3(Proposition a, Proposition b, Proposition c) {
        return INTERNER.intern( ArrowType.Or3, a, b, c );
    }

    @Override
    public Arrow not1(Proposition a, Proposition b) {
        return INTERNER.intern( ArrowType.Not1, a, b );
    }

    @Override
    public Arrow not2(Proposition a, Proposition b) {
        return INTERNER.intern( ArrowType.Not2, a, b );
    }

    @Override
    public Arrow not3(Proposition a) {
        return INTERNER.intern( ArrowType.Not3, a );
    }

    @Override
    public ModusPonens modusPonens(Arrow antecedent, Arrow conditional) {
        return (ModusPonens) INTERNER.intern( ArrowType.ModusPonens, antecedent,
 conditional );
    }

    @Override
    public boolean isThen1(Arrow arrow) {
        return ArrowType.Then1.equals( arrow.type );
    }

    @Override
    public boolean isThen2(Arrow arrow) {
        return ArrowType.Then2.equals( arrow.type );
    }

    @Override
    public boolean isAnd1(Arrow arrow) {
        return ArrowType.And1.equals( arrow.type );
    }

    @Override
    public boolean isAnd2(Arrow arrow) {
        return ArrowType.And2.equals( arrow.type );
    }

    @Override
    public boolean isAnd3(Arrow arrow) {
        return ArrowType.And3.equals( arrow.type );
    }

    @Override
    public boolean isOr1(Arrow arrow) {
        return ArrowType.Or1.equals( arrow.type );
    }

    @Override
    public boolean isOr2(Arrow arrow) {
        return ArrowType.Or2.equals( arrow.type );
    }

    @Override
    public boolean isOr3(Arrow arrow) {
        return ArrowType.Or3.equals( arrow.type );
    }

    @Override
    public boolean isNot1(Arrow arrow) {
        return ArrowType.Not1.equals( arrow.type );
    }

    @Override
    public boolean isNot2(Arrow arrow) {
        return ArrowType.Not2.equals( arrow.type );
    }

    @Override
    public boolean isNot3(Arrow arrow) {
        return ArrowType.Not3.equals( arrow.type );
    }

    @Override
    public boolean isModusPonens(Arrow arrow) {
        return ArrowType.ModusPonens.equals( arrow.type );
    }
}
```

E.2.2 pc-axiomatic-cdpl.lm

The standard language code simply wraps the methods defined in the Java code, and defines pattern matching definitions.

```
(load "include/pc.lm")
(load "include/categories.lm")

(let ((%axpc (edu.rpi.cs.tayloj.fluid.calculi.impl.AxiomaticPCImpl.)))
  (define (current-category)
      %axpc))
```

```

(define-primitive-method (and1 p q)
    (.and1 (current-category) p q))

(define-primitive-method (and2 p q)
    (.and2 (current-category) p q))

(define-primitive-method (and3 p q)
    (.and3 (current-category) p q))

(define-primitive-method (or1 p q)
    (.or1 (current-category) p q))

(define-primitive-method (or2 p q)
    (.or2 (current-category) p q))

(define-primitive-method (or3 p q r)
    (.or3 (current-category) p q r))

(define-primitive-method (not1 p q)
    (.not1 (current-category) p q))

(define-primitive-method (not2 p q)
    (.not2 (current-category) p q))

(define-primitive-method (not3 p)
    (.not3 (current-category) p))

(define-primitive-method (then1 p q)
    (.then1 (current-category) p q))

(define-primitive-method (then2 p q r)
    (.then2 (current-category) p q r))

(define-primitive-method (modus-ponens a c)
    (.modusPonens (current-category) a c))

(define-simple-match-expander (->then1 p) (arrow)
  `((.isThen1 (current-category) ,arrow)
    (codomain ,arrow)))

(define-simple-match-expander (->then2 p) (arrow)
  `((.isThen2 (current-category) ,arrow)
    (codomain ,arrow)))

(define-simple-match-expander (->and1 p) (arrow)
  `((.isAnd1 (current-category) ,arrow)
    (codomain ,arrow)))

(define-simple-match-expander (->and2 p) (arrow)
  `((.isAnd2 (current-category) ,arrow)
    (codomain ,arrow)))

(define-simple-match-expander (->and3 p) (arrow)
  `((.isAnd3 (current-category) ,arrow)
    (codomain ,arrow)))

(define-simple-match-expander (->or1 p) (arrow)
  `((.isOr1 (current-category) ,arrow)
    (codomain ,arrow)))

(define-simple-match-expander (->or2 p) (arrow)
  `((.isOr2 (current-category) ,arrow)
    (codomain ,arrow)))

(define-simple-match-expander (->or3 p) (arrow)
  `((.isOr3 (current-category) ,arrow)
    (codomain ,arrow)))
```

```

(define-simple-match-expander (->not1 p) (arrow)
  `((.isNot1 (current-category) ,arrow)
    (codomain ,arrow)))

(define-simple-match-expander (->not2 p) (arrow)
  `((.isNot2 (current-category) ,arrow)
    (codomain ,arrow)))

(define-simple-match-expander (->not3 p) (arrow)
  `((.isNot3 (current-category) ,arrow)
    (codomain ,arrow)))

(define-simple-match-expander (->modus-ponens antecedent conditional) (arrow)
  `((.isModusPonens (current-category) ,arrow)
    (.getAntecedent ,arrow)
    (.getConditional ,arrow)))
```

E.3 A Natural-Deduction Propositional Calculus Categorical DPL

In this section, we define the Java and standard language implementations of a categorical DPL for a natural-deduction presentation of the propositional calculus. We reuse the implementation of the propositional structures provided earlier.

E.3.1 NaturalDeductionPCImpl.java

The `NaturalDeductionPCImpl` class implements the categorical structure of the natural-deduction propositional calculus. This class implements a number of categorical interfaces, because the natural-deduction calculus has the relatively rich categorical structure of a bicartesian closed category.

```java
package edu.rpi.cs.tayloj.fluid.calculi.impl;

import java.util.Arrays;
import java.util.Objects;

import edu.rpi.cs.tayloj.fluid.calculi.impl.NaturalDeductionPCImpl.Arrow;
import edu.rpi.cs.tayloj.fluid.calculi.impl.NaturalDeductionPCImpl.Arrow.Type;
import edu.rpi.cs.tayloj.fluid.category.BicartesianClosedCategory;
import edu.rpi.cs.tayloj.fluid.category.HasAdjoin;
import edu.rpi.cs.tayloj.fluid.category.HasDoubleNegation;
import edu.rpi.cs.tayloj.fluid.category.HasIndeterminate;
import edu.rpi.cs.tayloj.fluid.dpl.PropositionalCalculus;
import static edu.rpi.cs.tayloj.fluid.dpl.PropositionalCalculus.*;
import edu.rpi.cs.tayloj.fluid.dpl.PropositionalCalculus.Conditional;
import edu.rpi.cs.tayloj.fluid.dpl.PropositionalCalculus.Conjunction;
import edu.rpi.cs.tayloj.fluid.dpl.PropositionalCalculus.Disjunction;
import edu.rpi.cs.tayloj.fluid.dpl.PropositionalCalculus.Proposition;
import edu.rpi.cs.tayloj.fluid.dpl.PropositionalCalculus.PropositionalVariable;
import edu.rpi.cs.tayloj.fluid.util.Interner;

/**
 * An implementation of the classical natural deduction calculus as a bicartesian closed
 * category with double negation arrows.   This class implements {@link HasIndeterminate}
 * and {@link HasAdjoin}, but some methods defined by these interfaces will throw
 * {@link UnsupportedOperationException} for instances produced by the no-argument
    constructor.
 * However, {@link #adjoin(Proposition)} will return an instance for which those methods
 * will be properly supported.  This is sort of a violation of the Liskov Substitution
    Principle,
 * but it greatly simplifies implementation, and the effects of the violation are
    unlikely to be
 * observed in practice.
 */
public class NaturalDeductionPCImpl
implements
BicartesianClosedCategory<
Proposition,NaturalDeductionPCImpl.Arrow, // Object, Arrow
Arrow, Arrow, // Identity, Composite
PropositionalVariable, Arrow, // Terminal Type (TRUE), Terminal Arrow
Conjunction, Arrow, Arrow, Arrow, // Product Type, Product Arrow, Left Proj., Right Proj.
Conditional, Arrow, Arrow, //Exponential Type, Curry, Eval
PropositionalVariable, Arrow, // Initial Type (FALSE), Initial Arrow
Disjunction, Arrow, Arrow, Arrow // Coproduct Type, Coproduct Arrow, Left Inj., Right Inj.
>,
HasDoubleNegation<
Proposition, Arrow, // Object, Arrow
PropositionalVariable, Arrow, // Terminal, Terminal Arrow
Conditional, Arrow, Arrow, // Exponential, Curry Arrow, Eval Arrow
Arrow // Double Negation Arrow
>,
HasIndeterminate<Proposition, Arrow, Arrow>,
HasAdjoin<Proposition, Arrow, Arrow>
{
    private final Arrow indeterminate;
    private final NaturalDeductionPCImpl parent;

    private NaturalDeductionPCImpl( Proposition assumption, NaturalDeductionPCImpl parent
    ) {
        this.indeterminate = INTERNER.create( Type.Indeterminate, assumption, this );
        this.parent = parent;
    }

    /**
     * Returns a new natural deduction category without an indeterminate.
     */
    public NaturalDeductionPCImpl() {
        this.indeterminate = null;
        this.parent = null;
    }

    static class Arrow {
        private final Object[] arguments;
        private final Proposition domain;
        private final Proposition codomain;
        private final Type type;
        Arrow( Type type, Proposition domain, Proposition codomain, Object[] arguments ) {
            Objects.requireNonNull( type );
            Objects.requireNonNull( domain );
            Objects.requireNonNull( codomain );
            Objects.requireNonNull( arguments );
            this.type = type;
            this.domain = domain;
            this.codomain = codomain;
            this.arguments = Arrays.copyOf( arguments, arguments.length );
        }
        Type getType() { return type; }
        Proposition getDomain() { return domain; }
        Proposition getCodomain() { return codomain; }
        Object getArgument(int n ){ return n < arguments.length ? arguments[n] : null; }

        enum Type {
            Identity, Composite,
```

```java
            Initial, Terminal,
            Product, LeftProjection, RightProjection,
            Coproduct, LeftInjection, RightInjection,
            Curry, Eval,
            DoubleNegation,
            Indeterminate
        }

        @Override
        public String toString() {
            return "(->"+type+" "+domain+" "+codomain+")";
        }

        static boolean isArrowType( Object object, Type type ) {
            return object instanceof Arrow && ((Arrow)object).getType().equals( type );
        }
    }

    private static Interner<Arrow> INTERNER = new Interner<Arrow>() {
        @Override
        public Arrow create(Object... key) {
            switch ((Arrow.Type) key[0] ) {
            case Identity:
                Proposition p = (Proposition) key[1];
                return new Arrow( Type.Identity, p, p, key );
            case Composite:
                Arrow g = (Arrow) key[1];
                Arrow f = (Arrow) key[2];
                if ( !f.codomain.equals( g.domain ) ) {
                    throw new IllegalArgumentException( "Cannot compose: the codomain of "+f+" is not the domain of "+g+"." );
                }
                else {
                    return new Arrow( Type.Composite, f.domain, g.codomain, key );
                }
            case Initial:
                p = (Proposition) key[1];
                return new Arrow( Type.Initial, FALSE(), p, key );
            case Terminal:
                p = (Proposition) key[1];
                return new Arrow( Type.Terminal, p, TRUE(), key );
            case Product:
                f = (Arrow) key[1];
                g = (Arrow) key[2];
                if ( !f.domain.equals( g.domain ) ) {
                    throw new IllegalArgumentException( f+" and "+g+" have different domains." );
                }
                else {
                    return new Arrow( Type.Product, f.domain, and(f.codomain, g.codomain ), key );
                }
            case LeftProjection:
                Conjunction c = (Conjunction) key[1];
                return new Arrow( Type.LeftProjection, c, c.getLeft(), key );
            case RightProjection:
                c = (Conjunction) key[1];
                return new Arrow( Type.RightProjection, c, c.getRight(), key );
            case Coproduct:
                f = (Arrow) key[1];
                g = (Arrow) key[2];
                if ( !f.codomain.equals( g.codomain ) ) {
                    throw new IllegalArgumentException( f+" and "+g+" have different codomains." );
                }
                else {
                    return new Arrow( Type.Coproduct, or(f.domain,g.domain), f.codomain, key );
                }
            case LeftInjection:
                Disjunction d = (Disjunction) key[1];
                return new Arrow( Type.LeftInjection, d.getLeft(), d, key );
            case RightInjection:
                d = (Disjunction) key[1];
                return new Arrow( Type.RightInjection, d.getRight(), d, key );
            case Curry:
                f = (Arrow) key[1]; // f : p & q → r
                c = (Conjunction) f.domain;
                return new Arrow( Type.Curry, c.getLeft(), implies(c.getRight(),
    f.codomain), key );
            case Eval:
                Conditional ifPthenQ = (Conditional) key[1];
                return new Arrow( Type.Eval, and( ifPthenQ, ifPthenQ.getLeft() ),
    ifPthenQ.getRight(), key );
            case DoubleNegation:
                p = (Proposition) key[1];
                return new Arrow( Type.DoubleNegation, not(not(p)), p, key );
            case Indeterminate:
                p = (Proposition) key[1];
                return new Arrow( Type.Indeterminate, TRUE(), p, key );
            default:
                throw new IllegalArgumentException( key[0] + " is not a legal Arrow
    type." );
            }
        }
    };

    @Override
    public Arrow identity(Proposition object) {
        return INTERNER.intern( Type.Identity, object );
    }

    @Override
    public boolean isIdentity(Object object) {
        return Arrow.isArrowType( object, Type.Identity );
    }

    @Override
    public Arrow compose(Arrow g, Arrow f) {
        return INTERNER.intern( Type.Composite, g, f );
    }

    @Override
    public boolean isComposite(Object object) {
        return Arrow.isArrowType( object, Type.Composite );
    }

    @Override
    public Arrow compositeAfter(Arrow h) {
        return (Arrow) h.getArgument(1);
    }

    @Override
    public Arrow compositeBefore(Arrow h) {
        return (Arrow) h.getArgument(2);
    }

    @Override
    public Proposition domain(Arrow arrow) {
        return arrow.getDomain();
    }

    @Override
    public Proposition codomain(Arrow arrow) {
        return arrow.getCodomain();
    }
```

```java

    @Override
    public boolean isArrow(Object object) {
        return object instanceof Arrow;
    }

    @Override
    public PropositionalVariable terminal() {
        return PropositionalCalculus.TRUE();
    }

    @Override
    public boolean isTerminal(Object object) {
        return PropositionalCalculus.TRUE().equals( object );
    }

    @Override
    public Arrow terminalArrow(Proposition object) {
        return INTERNER.intern( Type.Terminal, object );
    }

    @Override
    public boolean isTerminalArrow(Object object) {
        return Arrow.isArrowType( object, Type.Terminal );
    }

    @Override
    public Conjunction product(Proposition left, Proposition right) {
        return and(left,right);
    }

    @Override
    public boolean isProduct(Object object) {
        return Conjunction.class.isInstance( object );
    }

    @Override
    public Arrow leftProjection(Conjunction product) {
        return INTERNER.intern( Type.LeftProjection, product );
    }

    @Override
    public boolean isLeftProjection(Object object) {
        return Arrow.isArrowType( object, Type.LeftProjection );
    }

    @Override
    public Arrow rightProjection(Conjunction product) {
        return INTERNER.intern( Type.RightProjection, product );
    }

    @Override
    public boolean isRightProjection(Object object) {
        return Arrow.isArrowType( object, Type.RightProjection );
    }

    @Override
    public Arrow productArrow(Arrow f, Arrow g) {
        return INTERNER.intern( Type.Product, f, g );
    }

    @Override
    public boolean isProductArrow(Object object) {
        return Arrow.isArrowType( object, Type.Product );
    }

    @Override
    public Arrow productArrowLeft(Arrow fg) {
        return (Arrow) fg.getArgument(1);
    }

    @Override
    public Arrow productArrowRight(Arrow fg) {
        return (Arrow) fg.getArgument(2);
    }

    @Override
    public Conditional exponential(Proposition base, Proposition exponent) {
        return PropositionalCalculus.implies( exponent, base );
    }

    @Override
    public boolean isExponential(Object object) {
        return PropositionalCalculus.Conditional.class.isInstance( object );
    }

    @Override
    public Arrow curry(Arrow g) {
        return INTERNER.intern( Type.Curry, g );
    }

    @Override
    public boolean isCurryArrow(Object object) {
        return Arrow.isArrowType( object, Type.Curry );
    }

    @Override
    public Arrow curriedArrow(Arrow curriedG) {
        return (Arrow) curriedG.getArgument(1);
    }

    @Override
    public Arrow eval(Conditional exponential) {
        return INTERNER.intern( Type.Eval, exponential );
    }

    @Override
    public boolean isEvalArrow(Object object) {
        return Arrow.isArrowType( object, Type.Eval );
    }

    @Override
    public PropositionalVariable initial() {
        return PropositionalCalculus.FALSE();
    }

    @Override
    public boolean isInitial(Object object) {
        return FALSE().equals( object );
    }

    @Override
    public Arrow initialArrow(Proposition object) {
        return INTERNER.intern( Type.Initial, object );
    }

    @Override
    public boolean isInitialArrow(Object object) {
        return Arrow.isArrowType( object, Type.Initial );
    }

    @Override
    public Disjunction coproduct(Proposition left, Proposition right) {
        return PropositionalCalculus.or( left, right );
    }

    @Override
    public boolean isCoproduct(Object object) {
        return PropositionalCalculus.Disjunction.class.isInstance( object );
    }

    @Override
    public Arrow coproductArrow(Arrow left, Arrow right) {
        return INTERNER.intern( Type.Coproduct, left, right );
    }

    @Override
    public boolean isCoproductArrow(Object object) {
        return Arrow.isArrowType( object, Type.Coproduct );
    }

    @Override
    public Arrow coproductArrowLeft(Arrow fg) {
        return (Arrow) fg.getArgument(1);
    }

    @Override
    public Arrow coproductArrowRight(Arrow fg) {
        return (Arrow) fg.getArgument(2);
    }

    @Override
    public Arrow leftInjection(Disjunction coproduct) {
        return INTERNER.intern( Type.LeftInjection, coproduct );
    }

    @Override
    public boolean isLeftInjection(Object object) {
        return Arrow.isArrowType( object, Type.LeftInjection );
    }

    @Override
    public Arrow rightInjection(Disjunction coproduct) {
        return INTERNER.intern( Type.RightInjection, coproduct );
    }

    @Override
    public boolean isRightInjection(Object object) {
        return Arrow.isArrowType( object, Type.RightInjection );
    }

    @Override
    public Arrow doubleNegation(Proposition object) {
        return INTERNER.intern( Type.DoubleNegation, object );
    }

    @Override
    public boolean isDoubleNegation(Object object) {
        return Arrow.isArrowType( object, Type.DoubleNegation );
    }

    @Override
    public Proposition doubleNegationFormula(Arrow a) {
        return (Proposition) a.getArgument(1);
    }

    /*
     * We lie a little bit when we have NaturalDeductionPCImpl implement
     * the HasIndeterminate and HasAdjoin methods.  The 0-argument constructor
     * leaves the fields for parent and indeterminate as null, which means
     * that some of these methods will throw UnsupportedOperationExceptions.
     * When the adjoin(Proposition) method is used, though, a NaturalDeductionPCImpl
     * is returned that does have the appropriate fields.  In the mapping from
     * the axiomatic category to the natural deduction category, we entirely
     * ignore these methods, but in the deduction theorem, we map from a
     * NaturalDeductionPCImpl (with a non-null parent) to its parent.
     */

    @Override
    public NaturalDeductionPCImpl adjoin(Proposition object) {
        return new NaturalDeductionPCImpl( object, this );
    }

    @Override
    public NaturalDeductionPCImpl parent() {
        if ( parent != null ) {
            return parent;
        }
        else {
            throw new UnsupportedOperationException( "This category does not have a
parent." );
        }
    }

    @Override
    public Arrow indeterminate() {
        if ( indeterminate != null ) {
            return indeterminate;
        }
        else {
            throw new UnsupportedOperationException( "This category does not have an
indeterminate." );
        }
    }

    @Override
    public boolean isIndeterminate(Object object) {
        return Arrow.isArrowType( object, Type.Indeterminate ) &&
                ((Arrow)object).getArgument(2) == this;
    }
}
```

E.3.2 pc-nd-cdpl.lm

The standard language code simply wraps the methods defined in the Java code, and defines pattern matchers.

```
(load "include/pc.lm")
(load "include/categories.lm")

(let ((%nd (edu.rpi.cs.tayloj.fluid.calculi.impl.NaturalDeductionPCImpl.)))
  (define (current-category)
      %nd))

(define false-elim initial)
(define-match-alias ->false-elim ->initial)

(define true-intro terminal)
(define-match-alias ->true-intro ->terminal)

(define left-and pi-left)
(define-match-alias ->left-and ->pi-left)

(define right-and pi-right)
(define-match-alias ->right-and ->pi-right)

(define both product-arrow)
```

```
(define-match-alias ->both ->product)

(define left-or iota-left)
(define-match-alias ->left-or ->iota-left)

(define right-or iota-right)
(define-match-alias ->right-or ->iota-right)

(define cd coproduct-arrow)
(define-match-alias ->cd ->coproduct)

(define mp eval)
(define-match-alias ->mp ->eval)

(define discharge curry)
(define-match-alias ->discharge ->curry)

(define dn double-negation)
(define-match-alias ->dn ->double-negation)

(define assumption indeterminate)
(define-match-alias ->assumption ->indeterminate)

;;; BEGIN ndAssumeFunction
(define (%assume p d k)
  (dlet ((m (let* ((c (current-category))
                   (cx (.adjoin c p)))
              (define (current-category) cx)
              (try/finally
               (k (!d (!assumption)))
               (define (current-category) c)))))
   (!discharge (!left-and* (!m)))))
;;; END ndAssumeFunction

;;; BEGIN ndAssume
(define-macro (assume form end)
  (destructuring-bind (_ (prop var) . body) form
    `(!%assume ,prop (mu (,var) ,@body) dtm)))
;;; END ndAssume

;;; BEGIN dtm
(define (dtm x)
  ;; A deduction theorem mapping to be called in category 𝒞[x],
  ;; (with assumption x: ⊤ → A) with a 𝒞[x] arrow
  ;; f: B → C.  Returns a method to be called in 𝒞
  ;; that will derive an arrow B & A → C & A.  (From that
  ;; arrow, it's trivial to derive an arrow B → A ⊃ C.)
  (let* ((a (codomain (!assumption)))
         (-xA (mu (f) (!times f (!identity a)))))
    (match
     x
     ((->true-intro b)  (mu () (!-xA (!true-intro b))))
     ((->false-elim b)  (mu () (!-xA (!false-elim b))))
     ((->left-and b c)  (mu () (!-xA (!left-and b c))))
     ((->right-and b c) (mu () (!-xA (!right-and b c))))
     ((->left-or b c)   (mu () (!-xA (!left-or b c))))
     ((->right-or b c)  (mu () (!-xA (!right-or b c))))
     ((->dn b)          (mu () (!-xA (!dn b))))
     ((->mp b c)        (mu () (!-xA (!mp b c))))
     ((->identity p)
      (mu () (!identity (and p a))))
     ((->compose g f)
      (let ((gm (dtm g))
            (fm (dtm f)))
        (mu () (!compose (!gm) (!fm)))))
     ((->both f g)
      (let ((fm (dtm f))
            (gm (dtm g)))
        (mu ()
            (dlet* ((ff (!fm))
                    (gg (!gm)))
             (!both (!both (!left-and* ff)
                           (!left-and* gg))
                    (!right-and* ff))))))
     ((->cd f g)
      (let ((fm (dtm f))
            (gm (dtm g)))
        (mu ()
            (dlet ((h (!-xA (!cd (!discharge (!fm))
                                 (!discharge (!gm))))))
             (!mp* (!left-and* h)
                   (!right-and* h))))))
     ((->discharge f)
      (let ((fm (dtm f)))
        (mu ()
            (dlet ((w (!fm)))
             (dmatch w
              ((-> (and (and b c) a) (and d a))
               (dlet* ((x (!identity (and (and b a) c)))
                       (y (!discharge
                           (!left-and*
                            (!compose
                             (!fm)
                             (!both (!both (!left-and* (!left-and* x))
                                           (!right-and* x))
                                    (!right-and* (!left-and* x))))))))
                      (!both y (!right-and b a))))))))
     ((->assumption a)
      (mu ()
          (!both (!right-and TRUE a)
                 (!right-and TRUE a))))
     ;; any other arrow must be an indeterminate from an ancestor, so
     ;; it should be safe to claim it, since dtm should only be called
     ;; from within an (assume ...).
     ((-> _ _)
      (mu ()
          (!-xA (!claim x)))))))
;;; END dtm

;; A very common pattern in taking an existing arrow $f:A \to B$
;; and composing it with the result of some method m that produces an
;; arrow $g:B \to C$.  We call a method that combines these m*.
;; For instance right-and* can be called with an arrow
;; $f:A \to B \& C$ and will return an arrow $\pi' f:A \to C$.
;; For some rules, it makes more sense to compose from the other side, and
;; these variants begin with *.

(define (false-elim* f p)
  ;; $f:x \to \bot / \bot_p f:x \to p$
  (!compose (!false-elim p)             ; $\bot_p:\bot \to p$
            f))                         ; $f:x \to bot$

(define (*true-intro f p)
  (!compose                             ; $f\top_p:p \to q$
   f                                    ; $f:\top \to q$
   (!true-intro p)))                    ; $\top_p:p \to \top$

(define (left-and* f)
  ;; $f:x \to p \& q / \pi f:x \to p$
  (dmatch f
   ((-> _ (and q r))
    (!compose (!left-and q r) f))))

(define (*left-and f q)
  ;; $f:p \to x / f\pi:p \& q \to x$
  (dmatch f
   ((-> p _)
    (!compose f (!left-and p q)))))

(define (right-and* f)
  ;; f : x → p & q/π′f : x → q
  (dmatch f
   ((-> _ (and q r))
    (!compose (!right-and q r) f))))

(define (*right-and f p)
  ;; f : q → x/f π′ : p & q → x
  (dmatch f
   ((-> q _)
    (!compose f (!right-and p q)))))

(define (times f g)
  ;; f : a → c, g : b → d/f & g : a & b → c & d
  (dmatch (list f g)
   ((list (-> a c) (-> b d))
    (!both
     (!compose f (!left-and a b))
     (!compose g (!right-and a b))))))

(define (left-or* f q)
  ;; f : x → p/ιf : x → p ∨ q
  (dmatch f
   ((-> _ p)
    (!compose (!left-or p q) f))))

(define (*left-or f)
  ;; f : p ∨ q → x/f ι : p → x
  (dmatch f
   ((-> (or p q) _)
    (!compose f (!left-or p q)))))

(define (right-or* f p)
  ;; f : x → q/ι′f : x → p ∨ q
  (dmatch f
   ((-> _ q)
    (!compose (!right-or p q) f))))

(define (*right-or f)
  ;; f : p ∨ q → x/f ι′ : q → x
  (dmatch f
   ((-> (or p q) _)
    (!compose f (!right-or p q)))))

(define (cd* h f g)
  ;; h : x → a ∨ b, f : a → c, g : b → c/[f, g]h : x → c
  (dmatch (list h f g)
   ((list (-> _ (or a b)) (-> a c) (-> b c))
    (!compose (!cd f g) h))))

(define (recharge f)
  ;; AKA uncurry.  Opposite of discharge.
  (dmatch f
   ((-> a (if b c))
    (!compose                       ; a & b → c
     (!mp c b)                      ; b ⊃ c & b → c
     (!times f (!identity b)))))) ; a & b → (b ⊃ c) & b

(define (cd** h f g)
  (dmatch (list h f g)
   ((list (-> x (or a b))
          (-> x (if a c))
          (-> x (if b c)))
    (!mp*                                     ; x → c
     (!compose                               ; $x \to x \supset c$
      (!cd                                   ; $a \vee b \to x \supset c$
       (!discharge (!*swap (!recharge f)))   ; $a \to x \supset c$
       (!discharge (!*swap (!recharge g))))  ; $b \to x \supset c$
      h)                                     ; $x \to a \vee b$
     (!identity x)))))                       ; $x \to x$

(define (mp* f g)
  ;; $f : a \to b \supset c, g : a \to b / \mathrm{eval}\langle f, g\rangle : a \to c$
  (dmatch (list f g)
   ((list (-> a (if b c)) (-> a b))
    (!compose (!mp c b) (!both f g)))))

(define (discharge* n f)
  ;; $f : a \,\&\, \cdots \to c / \lambda^n f : a \supset \cdots \supset c$
  (dcond (== 0 n)
   (!claim f)
   (!discharge* (- n 1)
                (!discharge f))))

(define (dn* f)
  ;; $f : p \to \sim\sim q / \mathrm{dn}_q f : p \to q$
  (dmatch f
   ((-> _ (not (not p)))
    (!compose (!dn p) f))))

(define (*dn f)
  ;; $f : p \to q / f\,\mathrm{dn}_p : \sim\sim p \to q$
  (!compose f (!dn (domain f))))

;; Some other categorical utilities for swapping the orders of
;; commuative operators.

(define (swap* f)
  ;; $f : x \to a \cdot b / gf : x \to b \cdot a$
  (dmatch f
   ((-> _ (and a b))
    (!both (!right-and* f)
           (!left-and* f)))
   ((-> _ (or a b))
    (!cd* f
          (!right-or b a)
          (!left-or b a)))))

(define (*swap f)
  ;; $f : a \cdot b \to x / f\,g : b \cdot a \to x$
  (dmatch f
   ((-> (and a b) _)
    (!compose f (!swap* (!identity (and b a)))))
   ((-> (or a b) _)
    (!compose f (!swap* (!identity (or b a))))))))

(define (distribute a b c)
  ;; $c \times (a + b) \to (c \times a) + (c \times b)$
  (!*swap
   (!compose (!eval (or (and c a) (and c b)) c)
             (!times (!cd (!discharge (!left-or* (!swap* (!identity (and a c))
                                                  (and c b)))
                          (!discharge (!right-or* (!swap* (!identity (and b c))
                                                   (and c a))))
                     (!identity c)))))

;; Examples

;;; BEGIN example1
(define (example)
  ;; $\top \to (\sim a \vee b) \supset (a \supset b)$
  (assume ((or (not a) b) w)
   (!cd** w
          (assume ((not a) x)
           (assume (a y)
            (!false-elim* (!mp* x y) b)))
          (assume (b x)
           (assume (a _)
            (!claim x))))))

; > (!example)
; (->Curry TRUE (if (or (not a) b) (if a b)))
;;; END example1



(declare-variable a)
(declare-variable b)
(declare-variable c)
```

E.4 The Standard Library

```
;; JavaDot Notation interoperability

(define (javaDotConstructor classname)
    (lambda args
      (apply javaConstructor (cons classname args))))

(define (javaDotInstanceMethod methodname)
    (lambda (instance . args)
      (apply javaInstanceMethod (cons instance (cons methodname args)))))

(define (javaDotInstanceField fieldname)
    (lambda (instance)
      (javaInstanceField instance fieldname)))

(define (javaDotStaticMethod class-and-method-name)
    (lambda args
      (apply javaStaticMethod (cons class-and-method-name args))))

(define (javaDotStaticField class-and-field-name)
    (javaStaticField class-and-field-name))

;; input and output streams

(define +out+ java.lang.System.out$)
(define +err+ java.lang.System.err$)

;; == and equals

(define (equals x y)
    (java.util.Objects.equals x y))

(define (== x y)
    (edu.rpi.cs.tayloj.fluid.standard.Util.eq x y))

;; Basic List Utilities

(define first car)
(define rest cdr)
(define (second l) (car (cdr l)))
(define (third l) (car (cdr (cdr l))))
(define (list . elements) elements)
(define nil '())

```

```
;; Boolean

(define true (equals nil nil))
(define false (equals nil true))

(define (~ exp)
    (cond exp false
          true))

(define-macro (|| form env)
    (list 'cond (second form)
          true
          (third form)))

(define-macro (&& form env)
    (list 'cond (list '~ (second form))
          false
          (third form)))

;; arithmetic

(define (+ . numbers)
    (foldl edu.rpi.cs.tayloj.fluid.standard.Util.add
           0 numbers))

(define (- x . xs)
    (cond (endp xs)
          (edu.rpi.cs.tayloj.fluid.standard.Util.subtract 0 x)
          (foldl edu.rpi.cs.tayloj.fluid.standard.Util.subtract
                 x xs)))

(define (* . numbers)
    (foldl edu.rpi.cs.tayloj.fluid.standard.Util.multiply
           1 numbers))

(define (/ x . xs)
    (cond (endp xs)
          (edu.rpi.cs.tayloj.fluid.standard.Util.divide 1 x)
          (foldl edu.rpi.cs.tayloj.fluid.standard.Util.divide
                 x xs)))

(define (% x y)
    (edu.rpi.cs.tayloj.fluid.standard.Util.modulo x y))

(define (== x . ys)
    (every (lambda (y)
             (edu.rpi.cs.tayloj.fluid.standard.Util.eq x y))
           ys))

(define (/= x . ys)
    ;; returns true if all elements are pairwise disjoint; This
    ;; requires an O(n^2) check.
    (cond (endp ys) true
          (&& (endp (member-if (lambda (y)
                                  (== x y))
                                ys))
              (apply /= ys))))

(define (< x . xs)
    (every (lambda (x y)
             (edu.rpi.cs.tayloj.fluid.standard.Util.lessThan x y))
           (cons x xs)
           xs))

(define (<= x . xs)
    (every (lambda (x y)
             (|| (< x y) (== x y)))
           (cons x xs)
           xs))

(define (> x . xs)
    (every (lambda (x y)
             (< y x))
           (cons x xs)
           xs))

(define (>= x . xs)
    (every (lambda (x y)
             (<= y x))
           (cons x xs)
           xs))

(define (lognot x)
    (edu.rpi.cs.tayloj.fluid.standard.Util.bitwiseNot x))

(define (logand . xs)
    (foldl edu.rpi.cs.tayloj.fluid.standard.Util.bitwiseAnd
           -1 xs))

(define (logxor . xs)
    (foldl edu.rpi.cs.tayloj.fluid.standard.Util.bitwiseXor
           0 xs))

(define (logior . xs)
    (foldl edu.rpi.cs.tayloj.fluid.standard.Util.bitwiseOr
           0 xs))

;; More Cons/List Utilities

(define (consp object)
    (.isInstance edu.rpi.cs.tayloj.fluid.sexp.Cons.class object))

(define (atom x)
    (~ (consp x)))

(define (set-car cons object)
    (.setCar cons object))

(define (set-cdr cons object)
    (.setCdr cons object))

;; Symbol utilities

(define (intern string)
    (edu.rpi.cs.tayloj.fluid.sexp.Symbol.intern string))

(define (make-symbol string)
    (edu.rpi.cs.tayloj.fluid.sexp.Symbol.makeSymbol string))

(define (gensym . args)
    (apply edu.rpi.cs.tayloj.fluid.sexp.Symbol.gensym args))

(define (unintern symbol)
    (edu.rpi.cs.tayloj.fluid.sexp.Symbol.unintern symbol))

(define (symbolp object)
    (.isInstance edu.rpi.cs.tayloj.fluid.sexp.Symbol.class object))

(define (symbol-name symbol)
    (.symbolName symbol))

;; List Utilities

(define (endp x)
    (equals nil x))

```

```
(define (listp x)
    (|| (endp x)
        (consp x)))

(define (foldl function init list)
    (cond (endp list)
          init
          (foldl function
                 (function init (first list))
                 (rest list))))

(define (complement function)
    (lambda args
      (~ (apply function args))))

(define (every function . lists)
    (cond (some endp lists) true
          (&& (apply function (mapcar first lists))
              (apply every
                     function
                     (mapcar rest lists)))))

(define (some function . lists)
    (~ (apply every (complement function) lists)))

(define (notevery function . lists)
    (apply some (complement function) lists))

(define (notany function . lists)
    (apply every (complement function) lists))

(define (reverse list)
    (foldl (lambda (acc x)
             (cons x acc))
           nil
           list))

;; Implementation of quasiquote (backquote) and unquote.  The reader
;; creates the appropriate (quasiquote ...)  and (unquote ...)  forms
;; from `form and ,form respectively, and we need only macroexpand.

(define (append x y)
  (cond (endp x) y
        (cons (car x)
              (append (cdr x) y))))

(define-macro (unquote form env)
    (error "unquote encountered outside backquote: " form))

(define-macro (quasiquote form env)
    (expand-qq (second form) 0))

(define (quasiquotep form)
    (&& (consp form)
        (== 'quasiquote (first form))
        (|| (== 2 (length form))
            (error "malformed quasiquote: " form))))

(define (unquote-splicing-p form)
    (&& (consp form)
        (== 'unquote-splicing (first form))
        (|| (== 2 (length form))
            (error "malformed unquote-splicing: " form))))

(define (unquotep form)
    (&& (consp form)
        (== 'unquote (first form))
        (|| (== 2 (length form))
            (error "malformed unquote: " form))))

(define (expand-qq form depth)
    (cond (quasiquotep form)
          (list 'list ''quasiquote (expand-qq (second form) (+ depth 1)))
          (cond (unquotep form)
                (cond (equals 0 depth)
                      (second form)
                      (list 'list ''unquote (expand-qq (second form) (- depth 1))))
                (cond (~ (consp form))
                      (list 'quote form)
                      (cond (unquote-splicing-p (car form))
                            (cond (equals 0 depth)
                                  (list 'append
                                        (second (car form))
                                        (expand-qq (cdr form) depth))
                                  (list 'list
                                        (list 'list ''unquote-splicing (expand-qq (second
    (car form)) (- depth 1)))
                                        (expand-qq (cdr form) depth)))
                            (list 'cons
                                  (expand-qq (car form) depth)
                                  (expand-qq (cdr form) depth)))))))

(define (1+ n) (+ n 1))
(define (1- n) (- n 1))

(define (length list)
    (foldl (lambda (len element)
             (1+ len))
           0
           list))

(define (nthcdr n list)
    (cond (equals 0 n)
          list
          (nthcdr (1- n)
                  (rest list))))

(define (nth n list)
    (first (nthcdr n list)))

(define (foldr function list init)
    (cond (endp list)
          init
          (function (first list)
                    (foldr function
                           (rest list)
                           init))))

(define (member-if predicate list)
    ;; member returns the first tail of list whose car satisfies the
    ;; predicate
    (cond (|| (endp list)
              (predicate (first list)))
          list
          (member-if predicate (rest list))))

(define (member element list)
    ;; member returns the first tail of the list whose car equals
    ;; element, or the empty list if element is not a member of list.
    (member-if (lambda (x)
                 (equals x element))
               list))

(define (some predicate list)
    (~ (endp (member-if predicate list))))

```

```
;; Association List Utilities

(define (pairlis keys values)
    (mapcar cons keys values))

(define (assoc item alist)
    (cond (endp alist)
          nil
          (cond (equals item (car (first alist)))
                (first alist)
                (assoc item (rest alist)))))

(define (acons key value alist)
    (list* (cons key value) alist))

;; Spreadable arglists

(define (spread-arglist arglist)
    (cond (~ (consp arglist))
          (error "not a proper list: " arglist)
          (cond (endp (cdr arglist))
                (car arglist)
                (cons (car arglist)
                      (spread-arglist (cdr arglist))))))

(define (list* e . es)
    (spread-arglist (cons e es)))

(define (equals x y)
    (java.util.Objects.equals x y))

(define (toString x)
    (java.util.Objects.toString x))

(define null
    ;; the superclass of Object is null
    (.getSuperclass java.lang.Object.class))

;; mapcar

(define (mapcar function list)
    ;; a primitive mapcar that takes a function and a single list is
    ;; used to 'bootstrap' the proper mapcar that takes any (positive,
    ;; non-zero) number of lists
    (cond (endp list) '()
          (cons (function (first list))
                (mapcar function (rest list)))))

((lambda (mapcar) ; the primitive mapcar
   (define (mapcar function list . lists)
       ;; a full-fledged mapcar that can operate on one or more
       (cond (|| (endp list) (some endp lists))
             '()
             (cons (apply function
                          (first list)
                          (mapcar first lists))
                   (apply mapcar
                          function
                          (rest list)
                          (mapcar rest lists)))))
   ) mapcar)

;; seq and dseq

(define (phrase-sequencer bind-one)
    (lambda (form env)
      (cond (== 1 (length form))
            (error "At least one phrase required: " form)
            ((lambda (sentinel)
               (foldr (lambda (phrase more-phrase)
                        (cond (== more-phrase sentinel)
                              phrase
                              `(,bind-one ((,(gensym) ,phrase))
                                          ,more-phrase)))
                      (rest form)
                      sentinel))
             (cons nil nil)))))

(define-macro seq (phrase-sequencer 'let))
(define-macro dseq (phrase-sequencer 'dlet))

(define-macro progn (phrase-sequencer 'let)) ; for the Common Lispers
(define-macro dprogn (phrase-sequencer 'dlet))

;; Anonymous functions/methods with sequenced body forms

(define (n-ary-greek greek sequencer form)
    (cond (== 3 (length form))
          form
          `(,greek ,(second form)
                   (,sequencer ,@(rest (rest form))))))

(define-macro (lambda form env)
    (n-ary-greek 'lambda 'progn form))

(define-macro (mu form env)
    (n-ary-greek 'mu 'dseq form))

;; let/dlet

(define-macro (let form env)
    `((lambda ,(mapcar first (second form))
        ,@(rest (rest form)))
      ,@(mapcar second (second form))))

(define-macro (dlet form env)
    (let ((vars (mapcar first (second form)))
          (values (mapcar second (second form)))
          (body (rest (rest form))))
      `(!(mu ,vars ,@body) ,@values)))

(define (bind* bind)
    (lambda (form env)
      (foldr (lambda (binding body)
               `(,bind (,binding) ,body))
             (second form)
             `(,bind () ,@(rest (rest form))))))

(define-macro let* (bind* 'let))
(define-macro dlet* (bind* 'dlet))

;; destructuring-bind

(define (destructure pattern object body-fn)
    (cond (symbolp pattern)
          (cond (== nil pattern)
                (body-fn)
                `(let ((,pattern ,object))
                   ,(body-fn)))
          (cond (consp pattern)
                (let ((obj-var (gensym)))
                  `(let ((,obj-var ,object))
                     ,(destructure (car pattern)
                                   `(car ,obj-var)
                                   (lambda ()
                                     (destructure (cdr pattern)
                                                    `(cdr ,obj-var)
                                                    body-fn)))))
                (error  "malformed destructure: " pattern object))))

(define-macro (destructuring-bind form env)
    (destructure (second form)
                 (third form)
                 (lambda ()
                   `(progn ,@(rest (rest (rest form)))))))

;; check

(define-macro (check form env)
    (destructuring-bind (check . cases) form
      (foldr (lambda (c1 else)
               (destructuring-bind (test . forms) c1
                 `(cond ,(cond (equals test 'else)
                               true
                               test)
                        (progn ,@forms)
                        ,else)))
             cases
             `(error "No case in " ',cases " succeeded."))))

(define-macro (dcheck form env)
    (destructuring-bind (_ . cases) form
      `(! ,(foldr (lambda (case else)
                    (destructuring-bind (test . deductions) case
                      `(cond ,(cond (equals test 'else)
                                    true
                                    test)
                             (mu () ,@deductions)
                             ,else)))
                  cases
                  `(mu ()
                       (!derror "No case in " ',cases " succeeded."))))))

;; letrec

(define-macro (letrec form env)
    (destructuring-bind (nil bindings . forms) form
      `(let ,(mapcar (lambda (binding)
                       (destructuring-bind (var val) binding
                         (list var nil)))
                     bindings)
         ,@(mapcar (lambda (binding)
                     `(set ,@binding))
                   bindings)
         ,@forms)))

(define-macro (dletrec form env)
    (destructuring-bind (_ bindings . deductions) form
      `(!(letrec ,bindings (mu () ,@deductions)))))

;; nlet

(define-macro (nlet form env)
    (destructuring-bind (nil name bindings . body) form
      (let ((fun `(lambda ,(mapcar first bindings) ,@body))
            (arguments (mapcar second bindings)))
        `(letrec ((,name ,fun))
                 (,name ,@arguments)))))

;; with-gensyms

(define-macro (with-gensyms form env)
    (destructuring-bind (wg vars . body) form
      `(let ,(mapcar (lambda (var)
                        `(,var (gensym)))
                      vars)
         ,@body)))

;; Pattern Matching

(letrec
 (
  ;; an association list mapping symbols to expander functions
  (expanders '())

  ;; store an expander function in expanders
  (set-expander
   (lambda (name matcher)
     (set expanders (acons name matcher expanders))))

  ;; retrieve an expander function from expanders
  (get-expander
   (lambda (name)
     (assoc name expanders)))

  ;; expand a number of patterns and objects
  (expand-matches
   (lambda (patterns objects vars fail-fn body-fn)
     (check
      ;; patterns and objects should run out at the same
      ;; time, at which point we execute the body
      ((&& (endp patterns) (endp objects))
       (body-fn vars))
      ;; if they don't it's an error
      ((|| (endp patterns) (endp objects))
       (error "mismatched patterns and objects: " patterns objects))
      ;; expand the first pattern with the first object-var where
      ;; the body will be generated by calling expand-matches
      ;; (via em) recursively.
      (else
       (expand-match (first patterns)
                     (first objects)
                     vars
                     fail-fn
                     (lambda (vars)
                       (expand-matches (rest patterns)
                                       (rest objects)
                                       vars
                                       fail-fn
                                       body-fn)))))))

  ;; expand a single pattern and object
  (expand-match
   (lambda (pattern object vars fail-fn body-fn)
     (check
      ;; _ is a wildcard that introduces no bindings
      ((== '_ pattern)
       (body-fn vars))
      ;; a symbol introduces a single binding
      ((symbolp pattern)
       (check
        ;; if not a member of vars, then bind it and call the body
        ;; with the extended list of vars.
        ((endp (member pattern vars))
         `(let ((,pattern ,object))
            ,(body-fn (list* pattern vars))))
        ;; but if it is a member of vars, then it's already been
        ;; bound, and we check that the previously established value
        ;; is the same as the present value.
        (else
         `(check
           ((~ (equals ,pattern ,object))
            ,(fail-fn))
           (else
            ,(body-fn vars))))))
      ;; a non cons, non symbol object is an equals literal
      ((~ (consp pattern))
       `(check
         ((~ (equals ',pattern ,object))
          ,(fail-fn))
         (else ,(body-fn vars))))
      ;; otherwise some (op .  patterns)
      (else
       (with-gensyms (objx)
         (destructuring-bind (op . patterns) pattern
           (let ((expander (get-expander op)))
             (check
              ((endp expander)
               (error "No expander for " op))
              (else
               `(let ((,objx ,object))
                  ,((cdr expander)
                    patterns objx vars
                    fail-fn body-fn))))))))))

;; matching is not significantly more complicated in the lambda-mu
;; calculus than in the pure lambda calculus, even though there are
;; both expressions and deductions in the lambda-mu calculus.  A given
;; matching construct XMATCH that has the syntax:
;;
;; (xmatch object
;;  (pattern1 phrases1...)
;;  ...
;;  (patternN phrasesN...))
;;
;; is compiled into a try expression that tries matching the object
;; to each patterni in succession.  When some patterni matches, the
;; try expression returns a callable (either anonymous function
;; (lambda () phrasesi) or an anonymous method (mu () phrasesi)).
;; The result is then invoked (either as a function application or
;; a method invocation) with zero arguments.  Returning the phasesi
;; in a callable in this way both ensures that the phrases keep the
;; lexical bindings established by the pattern matching, and
;; ensures that only errors thrown by the pattern matching code (as
;; opposed to the phrasesi) will be caught by the try expression.
;;
;; Matcher implements this technique.  Since deductions must be
;; syntactically recognizable, matcher takes an xapply argument which
;; is a list of symbols to splice into a call site (which makes the
;; call either an function application (an expression) or a method
;; invocation (a deduction)) and an xlambda (either the symbol lambda
;; or the symbol mu) for wrapping up the phrases.
(matcher
 (lambda (xapply xlambda)
   (lambda (form env)
     (destructuring-bind (nil object . clauses) form
       (with-gensyms (obj)
         ;; the outermost layer of xapply/xlambda (obj) here binds the
         ;; object that will be matched.  The call to the xlambda
         ;; produced by a matching clause is inside this scope, so
         ;; that if the object form is a deduction, its result is in
         ;; the assumption base when the body is evaluated.
         `(,@xapply (,xlambda (,obj)
            (,@xapply
             (try ,@(mapcar (lambda (clause)
                              (destructuring-bind (pattern . forms)
                                  clause
                                (expand-match
                                 pattern obj
                                 '()
```

```
                                    (lambda ()
                                      `(error ,object " did not match: " ',pattern))
                                    (lambda (vars)
                                      `(,xlambda () ,@forms)))))
                               clauses))))
             ,object))))))

 (make-simple-expander
  ;; make simple expander takes an integer (the number of expected
  ;; subpatterns) and a function of one argument (a symbol, the
  ;; variable bound to the object to be matched) that returns a list
  ;; whose first element is a test form and whose remaining elements
  ;; are accessors into an object satisfying the test form, and
  ;; returns an expander function.
  (lambda (arity get-test-and-parts)
    (lambda (subpatterns object vars fail body)
      (check
       ((== (length subpatterns) arity)
        (destructuring-bind (test . parts) (get-test-and-parts object)
          `(check
            ((~ ,test) ,(fail))
            (else ,(expand-matches
                    subpatterns parts
                    vars fail body)))))))))
 )

(define expand-match expand-match)
(define expand-matches expand-matches)

(define-macro match (matcher '() 'lambda))
(define-macro dmatch (matcher '(!) 'mu))

;; A match expander is a function of two arguments, the first of which
;; is a list of patterns, and the second of which is a symbol.  The
;; matcher must produce a list of forms, the first of which should
;; return false when evaluated if the value bound to the symbol is not
;; matchable by the operand.  The remaining elements in the list
;; should be forms that, when evaluated, produce objects that should
;; be be destructured for the patterns.  `define-match-expander'
;; simplifies the constuction of match expanders.

(define-macro (define-simple-match-expander form env)
    ;; define a simple match expander.
    ;;
    ;; (define-match-expander (op (pattern...)  object) .  body)
    ;;
    ;; defines an expansion function for patterns whose head is `op'.
    ;; The simple match expander is a more user-friendly interface
    ;; than the more complex `define-match-expander'.  The `body'
    ;; should produce a list of forms, the first of which should
    ;; return true if the `object' is matchable by the pattern, and
    ;; the rest of which should produce values to be matched to as
    ;; many patterns as are present in `patterns'.  The actual
    ;; patterns in a match form are not available in `body', but the
    ;; matcher defined by `define-simple-match-expander' will check
    ;; that the number of patterns in a match form is the same of the
    ;; number of patterns provided in the definition.
    ;;
    ;; Example:
    ;; (define-simple-match-expander (cons (kar kdr) kons)
    ;;  `((consp ,kons)
    ;;  (car ,kons)
    ;;  (cdr ,kons)))
    (destructuring-bind (_ (op . subpatterns) (object) . body) form
      (check
       ((consp subpatterns)
        (with-gensyms (%subpatterns)
          `(,set-expander
             ',op (,make-simple-expander
                   ,(length subpatterns)
                   (lambda (,object)
                     ,@body))))))))

 (define-macro (define-match-expander form env)
     ;; define a match expander.  define-match-expander defines an
     ;; expansion function for patterns whose head is ‘op’.  Within
     ;; the ‘body’, ‘patterns’ will be bound to a list of patterns
     ;; that should match the parts of the object being matched.
     ;; ‘object’ will be bound to a symbol whose that will be bound to
     ;; the object being matched.  ‘vars’ is a list of variables
     ;; already bound in the match pattern.  ‘fail’ is a function that
     ;; will produce a form that should be taken if the match fails,
     ;; and ‘body’ will produce the code that should be evaluated in
     ;; the scope of any introduced bindings.
     (destructuring-bind (_ (op subpatterns object vars fail bodyf) . body) form
       ‘(,set-expander
         ',op (lambda (,subpatterns ,object ,vars ,fail ,bodyf)
                ,@body))))

 (define-macro (define-match-alias form env)
   (destructuring-bind (_ alias original) form
     ‘(,set-expander ',alias (cdr (,get-expander ',original)))))
 )

(define-match-expander (equals %subpatterns %object %vars %fail %body)
    (destructuring-bind (%subpattern) %subpatterns
      ‘(check
        ((~ (equals ,%object ,%subpattern))
         ,(%fail))
        (else
         ,(%body %vars)))))

(define-match-expander (list %subpatterns %list %vars %fail %body)
    (check
     ;; If there is a list of patterns, match the first one against
     ;; first element of the list, and match the rest against the
     ;; rest.
     ((consp %subpatterns)
      ‘(check
        ((~ (consp ,%list))
         ,(%fail))
        (else
         ,(expand-match
           (first %subpatterns) ‘(first ,%list)
           %vars %fail (lambda (%vars)
                         (expand-match
                          (list* 'list (rest %subpatterns)) ‘(rest ,%list)
                          %vars %fail %body))))))
     ;; If there are no patterns, then the object should be the empty
     ;; list.
     ((endp %subpatterns)
      ‘(check
        ((~ (endp ,%list)) ,(%fail))
        (else ,(%body %vars))))
     ;; If there’s something else, then got here by a dotted list
     ;; pattern, and the whole of the remaining list should be matched
     ;; to it.
     (else
      (expand-match %subpatterns %list %vars %fail %body))))

(define-match-expander (as %subpatterns %object %vars %fail %body)
    ;; (as x pattern) binds the object as x, and also matches it into
    ;; pattern.  The usual use will have x as a variable, for holding
    ;; a reference to the object, but both x and pattern can be
    ;; arbitrary patterns.
    (destructuring-bind (var pattern) %subpatterns
      (expand-matches (list var pattern)
                      (list %object %object)
                      %vars %fail %body)))

(define-match-expander (satisfies %subpatterns %object %vars %fail %body)
    (destructuring-bind (test) %subpatterns
      `(check
        ((~ (,test ,%object))
         ,(%fail))
        (else
         ,(%body %vars)))))

(define-macro (dcond form env)
    (destructuring-bind (_ test dthen delse) form
      `(!(cond ,test
              (mu () ,dthen)
              (mu () ,delse)))))

;; Java interoperability

(define-macro (try/catch form env)
    ;; (try/catch exp .  catches) ===
    ;; (try/catch exp ,@catches nil)
    (destructuring-bind (t/c exp . catches) form
      `(try/catch/finally
        ,exp
        ,@catches
        nil)))

(define-macro (try/finally form env)
    ;; (try/finally exp .  forms) ===
    ;; (try/finally/catch exp (progn .  forms))
    (destructuring-bind (t/f exp . forms) form
      `(try/catch/finally ,exp (progn ,@forms))))

(define-macro (try form env)
    ;; The (try ...)  form is the DPL try that tries successive
    ;; expressions until one succeeds.  If the first expression throws
    ;; an exception, then the second is tried, then the third, and so
    ;; on until one finally works, or else if none works, an exception
    ;; is thrown with a message to that extent,
    (with-gensyms (e)
      (let ((sentinel (cons nil nil)))
        (foldr (lambda (exp more)
                 (check
                  ((== more sentinel) exp)
                  (else
                   `(try/catch/finally
                     ,exp
                     ((java.lang.Exception.class)
                      (lambda (,e) ,more))
                     nil))))
               (rest form)
               sentinel))))

(define (try-methods methods)
    ;; Apply each of the methods in the list in turn until one
    ;; produces an result without failing.  The method applications
    ;; are wrapped in identity calls because, according to the grammar
    ;; of the lambda-mu calculus, the body of a lambda function must
    ;; be an expression, and not a deduction.  This precludes a
    ;; function (lambda () (!some-method args)).  However, method
    ;; applications may be arguments to functions, so we *can* have
    ;; ((lambda (x) x) (!some-method args)).
    (check
     ((endp methods)
      (error "No methods to try"))
     (else
      (try/catch/finally
       ((lambda (x) x)
        (!(first methods)))
       ((java.lang.Exception.class)
        (lambda (e)
          (try-methods (rest methods))))
       nil))))

(define-primitive-method (%dtry methods)
    (try-methods methods))

(define-macro (dtry form env)
    (destructuring-bind (_ . deductions) form
      `(!%dtry
        (list ,@(mapcar (lambda (deduction)
                          `(mu () ,deduction))
                        deductions)))))

(define-macro (prog1 form env)
    (destructuring-bind (prog1 form . forms) form
      (let ((result (gensym)))
        `(let ((,result ,form))
           ,@forms
           ,result)))))

(define-macro (prog2 form env)
    (destructuring-bind (prog2 form1 form2 . forms) form
      `(seq ,form1
            (prog1 ,form2 ,@forms))))

;; throwing errors

(define (error . args)
    (throw (java.lang.RuntimeException. (apply strcat args))))

(define-primitive-method (derror . args)
    (throw (java.lang.RuntimeException. (apply strcat args))))

(define (strcat . args)
    ;; The java string concatenation operator should be implemented
    ;; something like this, except that it can coaslesce constants.
    (nlet app ((sb (java.lang.StringBuilder.))
               (args args))
          (check
           ((endp args) (.toString sb))
           (else (app (.append sb (first args))
                      (rest args))))))

;; Comments

(define-macro (comment form env) 'nil)

;; Arrays

(define (make-array . args)
    (apply java.lang.reflect.Array.newInstance args))

(define (get-array array index)
    (java.lang.reflect.Array.get array index))

(define (set-array array index object)
    (java.lang.reflect.Array.set array index object))

;; Java foreach

(define (%foreach-iterable fun iterable finish)
    (let ((i (.iterator iterable)))
      (nlet loop ()
            (check
             ((~ (.hasNext i))
              (finish))
             (else
              (fun (.next i))
              (loop))))))

(define (%foreach-array fun array finish)
    (let ((n (java.lang.reflect.Array.getLength array)))
      (nlet loop ((i 0))
            (check
             ((== i n)
              (finish))
             (else
              (fun (java.lang.reflect.Array.get array (.intValue i)))
              (loop (1+ i)))))))

(define (%foreach fun thing finish)
    (check
     ((.isArray (.getClass thing))
      (%foreach-array fun thing finish))
     (else
      (%foreach-iterable fun thing finish))))

(define-macro (foreach form env)
    (destructuring-bind (_ (item iterable . results) . body) form
      `(%foreach (lambda (,item) ,@body)
                 ,iterable
                 (lambda () nil ,@results))))

;; timing things

(define (invoke-with-timer function)
    (let ((begin (java.lang.System.nanoTime))
          (result (function))
          (end (java.lang.System.nanoTime)))
      (.println +err+ (strcat "Evaluation required "
                              (/ (- end begin) 1000000000.0) " seconds"))
      result))

(define-macro (time form env)
    (destructuring-bind (_ . body) form
      `(invoke-with-timer
        (lambda () ,@body))))

(define (dinvoke-with-timer method)
    (dlet ((begin (java.lang.System.nanoTime))
           (result (!method))
           (end (java.lang.System.nanoTime)))
          (dlet ((_ (.println +err+ (strcat "Evaluation required "
                                            (/ (- end begin) 1000000000.0)
                                            " seconds"))))
                (!claim result))))

(define-macro (dtime form env)
    (destructuring-bind (_ . body) form
      `(!dinvoke-with-timer
        (mu () ,@body))))

;; evaluator interface

(let ((+evaluator+ +evaluator+))
  (define (boundp symbol env)
      (cond (== env null)
            (.boundp +evaluator+ symbol)
            (.isBound env symbol)))

  (define (makunbound symbol)
      (.makunbound +evaluator+ symbol))

  (define (symbol-value symbol)
      (.symbolValue +evaluator+ symbol))

  (define (macro-function symbol)
      (.macroFunction +evaluator+ symbol))

  (define (macroboundp symbol)
      (.macroboundp +evaluator+ symbol))

  (define (macroexpand form env)
      (.macroexpand +evaluator+ form env))

  (define (macroexpand-1 form env)
      (.macroexpandOnce +evaluator+ form env))

  (makunbound '+evaluator+))

;; runtime interface

(let ((+runtime+ +runtime+))
  (define (load pathname)
      (.load +runtime+ pathname))

  (define (quit)
      ;; halt the REPL
      (.quit +runtime+))

  (define (halt)
      ;; stop loading a file
      (.halt +runtime+))

  (makunbound '+runtime+))

(define-macro (when form env)
    (destructuring-bind (_ test . forms) form
      `(cond ,test (progn ,@forms) nil)))

;; Automatically generating wrappers

(define (static-definitions class)
    ;; Returns a form that will define wrappers for all the public
    ;; static methods in the given class.
    (let* ((definitions '())
           (push (lambda (x) (set definitions (list* x definitions)))))
      (foreach (method (.getDeclaredMethods class) `(progn ,@definitions nil))
       (let ((mmods (logand (.getModifiers method)
                            (java.lang.reflect.Modifier.methodModifiers))))
         (when (&& (java.lang.reflect.Modifier.isPublic mmods)
                   (java.lang.reflect.Modifier.isStatic mmods))
           (push `(define (,(intern (.getName method)) . args)
                      (apply (javaDotStaticMethod
                              ,(strcat
                                (.getCanonicalName (.getDeclaringClass method))
                                "." (.getName method)))
                             args))))))))

(define-macro (define-static-methods form env)
    (destructuring-bind (_ class) form
      (static-definitions class)))

;; claim and ab-check

(define (ab-check p)
    ;; Returns p if p is in the assumption base,
    ;; and throws an error otherwise
    (check
     ((.contains (ab) p) p)
     (else (error p " is not in the assumption base"))))

(define-primitive-method (claim p)
    ;; Derives p if p is in the assumption base.  Every DPL
    ;; has a claim method.
    (ab-check p))
```

Appendix F

Slate

In this chapter we discuss Slate, a software system under development for a number of years, and one of the primary motivations for the current work in fluid logics. Slate is a argument construction environment originally developed for intelligence analysis, and has since evolved into a robust proof construction environment for students of formal logic. Slate is used in Rensselaer's introductory logic courses. The present work has not yet been applied to Slate, but Slate's use of proof translation, particularly in implementing automated "oracles," is a prime candidate for application of this technology. Figure F.7 (p. 163) shows an example where sophisticated proof translation would be a significant improvement.

F.1 History

Work on Slate began in the summer of 2003 under defense-related programs dedicated to improving the tools available to intelligence analysts (IAs). During this period, the intelligence community (IC) recognized an urgent need for information sharing on many levels. At the organizational level, there were policy limitations on information sharing between intelligence agencies that presented numerous difficulties, but even at the technological level, a lack of common representation schemata and storage technology prevented wide-scale information sharing even when permitted by policy. Another significant difficulty faced by the IC was understanding and documenting the reasoning of IAs that ultimately finds expression in reports issued to policy makers. A lack of clearly defined terminology for principles of probability and likelihood, and for documenting reasoning techniques were significant. In various research programs and with various collaborators, Slate served as a testbed for technologies designed to address these issues, including logical approaches culminating in "Provability-Based Semantic Interoperability" (PBSI), natural language processing efforts for question and answer (QA) systems for intelligence analysts, and argument-mapping techniques based on non-deductive inferences incorporating strength factors.

Policy changes in the early 2000's addressed the greater need for information sharing between intelligence agencies, particularly with respect to terrorism related data (Office of the Director of National Intelligence, 2008). Institution-level changes made it easier for information collected by one agency to find its way to analysts in other agencies, often with some "scrubbing" required (e.g., protection of original sources), but provisions for information sharing are insufficient to allow for all the desirable types of joint reasoning. Particularly, although a number of agencies may now be permitted to share data, different agencies have adopted as many different toolsets and knowledge representations schemata; much more data is now, technically, available, but is not usable by anyone but its original producer.

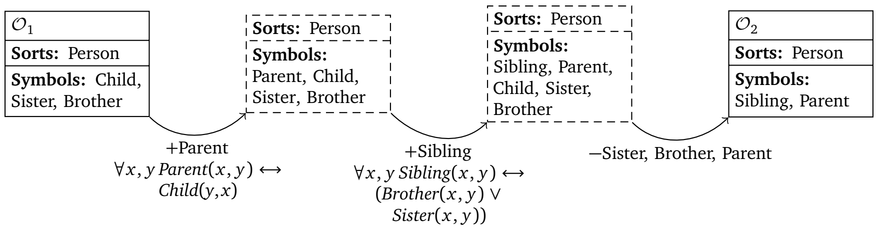
Even in cases where the conversions needed to make one agency's datasets usable by another, the "scrubbing" may require that another agency uses those converted datasets in different ways. For instance, assume agency A collected dataset D under a set of conditions C that makes D reliable and certain. A may be willing to draw strong conclusions from this data and to make recommendations accordingly. When A makes D available to other agencies, however, A may not be able to share the details of the conditions that make A confident in D. Other agencies necessarily reason differently over D than A does. The metadata that inform these kinds of considerations are sometimes available in structured form, are sometimes available in unstructured form, and are sometimes simply unavailable.

Researchers working to develop the tools and technologies necessary for the kinds of information sharing and joint reasoning desired faced two primary challenges:

Information Sharing Technologies must be in place to make information collected and maintained by many different agencies available to other agencies and in a form suitable for native tools. This requires translation of information expressed in one knowledge representation scheme to be dynamically translated into another. Often times complete translation is impossible, and suitable mechanisms for partial translations must be in place.

Joint Reasoning New tools must be developed that are aware of the many sources of information and that provide suitable ways of handling information that comes from different sources, has different provenance, and is of varying degrees of certainty, reliability, and likelihood.

RAIR Lab researchers addressed these challenges in two ways, developing new formalisms and strategies. These developments were first realized as modifications and enhancements to Slate. The need for information sharing led to the development of "provability-based semantic interoperability" (Shilliday et al., 2010) using translation graphs, a framework wherein translation is implemented as proof search. In the area of joint reasoning, Slate's argument mapping system was augmented with strength factors and



$\mathcal{O}_1$
Sorts: Person
Symbols: Child, Sister, Brother
Sorts: Person
Symbols: Parent, Child, Sister, Brother
Sorts: Person
Symbols: Sibling, Parent, Child, Sister, Brother
$\mathcal{O}_2$
Sorts: Person
Symbols: Sibling, Parent
+Parent
$\forall x, y\ Parent(x, y) \leftrightarrow Child(y, x)$
+Sibling
$\forall x, y\ Sibling(x, y) \leftrightarrow (Brother(x, y) \vee Sister(x, y))$
−Sister, Brother, Parent

Figure F.1: A vocabulary $\mathcal{O}_1$ with the terms "Child," "Sister," and "Brother" is gradually transformed into one $\mathcal{O}_2$, containing "Sibling," and "Parent." An axiom is associated with each of the first two modifications. The translation graph would be used to guide development of systems using information from $\mathcal{O}_1$ and $\mathcal{O}_2$.

other features to support the kinds of argumentation observed in intelligence analysis.

F.1.1 Provability-Based Semantic Interoperability & Translation Graphs

The main result in "provability-based semantic interoperability" (PBSI) is the use of translation graphs in describing relationships between ontologies and their use as guides to implementing systems that inter-operate semantically. In the PBSI approach, an ontology is a logical vocabulary in many-sorted logic along with associated axioms governing the terms of the vocabulary. Such a view of ontology is general enough to capture knowledge bases of traditional logic based approaches, relational databases, and contemporary Semantic Web ontologies. Even data sources not traditionally viewed as knowledge bases, e.g., environmental sensors, can be viewed as ontologies in this framework.

Using a small number of primitive operations, ontologies are reconstructed from an empty ontology by gradually adding vocabulary terms and axioms. Once a number of ontologies that are to be related have been reconstructed, the same primitive operations are used to transform the ontologies into each other. For example, two genealogical ontologies, once reconstructed in the PBSI framework, might be related by the translation graph shown in Figure F.1.

The term translation graph is actually misleading; although a translation graph may indicate how information expressed under one ontology might be translated into another, in many cases complete translation is not possible. The translation graph approach tends to make it easier to see how information can be shared even when complete translation is not possible.

The PBSI work was developed during the ARDA-sponsored Interoperable Knowledge Representation for Intelligence Support (IKRIS) Challenge Workshop. In the IKRIS workshop, several research groups worked to achieve interoperability between tools developed for intelligence analysis, specifically Slate, KANI (Fikes et al., 2005), and Noöscape (Siegel et al., 2005). This work is described in the final IKRIS report (Thurman et al., 2006). Later work focused on interoperability between Slate and Oculus Inc.'s GeoTime geospatial visualization system via the PBSI-based VIKRS translation services (Chappell et al., 2007; Kapler &

Wright, 2005). The techniques of PBSI were also applied to interoperability with natural language question and answer (QA) systems for intelligence analysis such as HITIQA (Strzalkowski et al., 2005) and Bringsjord et al.'s (2007) Solomon.

F.1.2 Argument Mapping

With increased information availability, tools for working with massive amounts of data become of the utmost importance. Needs commonly associated with massive datasets include visualization, organization, provenance tracking, consistency checking, and hypothesis generation. As we examined the documents used in training intelligence analysts (e.g., case studies such as Hughes's (2003) *Sign of the Crescent*), we recognized the great importance placed on argumentation. Argument maps in these materials were often expressed in tree form, influenced by so called "Toulmin diagrams" (Toulmin, 2003), not unlike arguments as presented in Slate. Common presentations lacked standard terminology for describing certainty, reliability, and estimates of likelihood, although the IC recognized the need and was moving toward a more consistent vocabulary (e.g., see *What We Mean When We Say: An Explanation of Estimative Language* (National Intelligence Council, 2007, p. 5)). Nonetheless, these reports still make liberal use of language expressing varying degrees of probability, certainty, and likelihood. For instance, the CIA's *Iraqi Mobile Biological Warfare Agent Production Plants* (Central Intelligence Agency, 2003) says (emphasis added):

> Analysis of the trailers reveals that they *probably* are ... plants described by the source. ... [Plates] on the fermentors list production dates of 2002 and 2003—*suggesting* Iraq continued to produce these units ... [The] layout and equipment are *consistent* with information provided ...

Focusing our efforts on incorporating the language of certainty and likelihood into Slate arguments, we turned to philosophical accounts of these topics as well as IC best practice (Kent), and developed an extension to Slate based on Chisholm's (1989) strength factors. Slate's strength factors are described in detail by Taylor et al. (2008), while examples of their use are given by Shilliday et al. (2007a) in a reconstruction of the CIA's argument regarding Iraqi biological weapons capabilities and by Clark et al. (2007) in a reconstructed analysis of the events leading up to the 1941 attack on Pearl Harbor. Figure F.2 shows manually assigned strength factors in a Slate argument and their propagation throughout the structure.

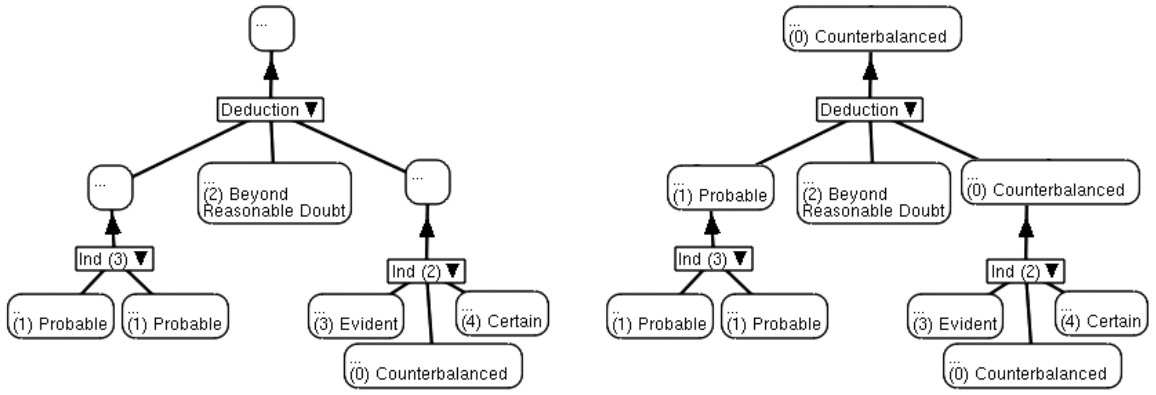


Figure F.2: The Slate argument structure on the left has strength factors assigned to the leaf nodes of the argument and to inferential links. The structure on the right has the same manually assigned strength factors, but the strength factors have been propagated through the argument. The labels on the inferential links indicate the mode of reasoning such as Induction and Deduction.

F.2 Proof Theory

Slate is first and foremost a system for argument construction. It has been used in the construction of non-deductive arguments (e.g., arguments using inductive or abductive inferences), but herein we are concerned with deductive arguments in formal proof calculi. Arguments in Slate are presented in a graphical natural deduction style similar to Gentzen-style proofs, but without the requirement that arguments be trees. For instance, a proof of the associativity of conjunction is rendered in Slate in Figure F.3.

The nodes in the proof graph in Figure F.3 each have the text { 1 } appearing below their formula. This indicates that they are in the scope of an assumption introduced by the formulae identified by 1. This style of assumption tracking is used in Suppes's (1957) classic textbook, and is particularly suitable for proofs in Slate where much less structure is imposed on the graphical layout of a proof than in many traditional systems. In the currently supported proof systems, **Assume** is the only inference rule that introduces assumptions, but a number of rules discharge them. Figure F.4 shows assumptions being discharged by disjunction elimination and conditional introduction. Also note the default behavior whereby the set of in-scope assumptions of a formulae is the union of the assumptions of its premises.

This assumption tracking method has been generalized to allow proof systems to define arbitrary attributes that may be associated with formulae in a proof. The natural deduction rules for boolean connectives (which are shared by Slate's proof systems for the propositional calculus, for first-order logic, for modal logics, &c.) define the "in-scope assumptions" attribute along with default and specialized mechanisms for computing the attribute value.

The general attribute framework has been used to implement the modal propositional logics supported

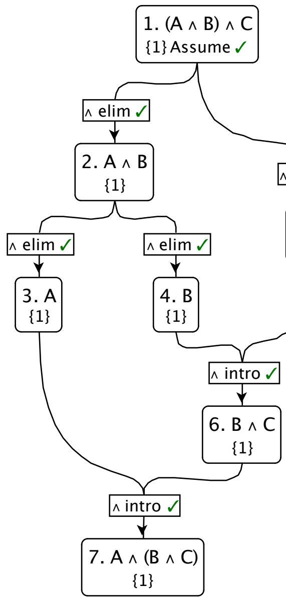


Figure F.3: A proof of the associativity of conjunction rendered in Slate illustrates that formulae in Slate proofs may be premises to more than one inference, in contrast to the tree structure of Gentzen-style proofs. Though formulae have identifiers which, by default, are numeric, less linear ordering is imposed in Slate proofs than in Fitch-style proofs.

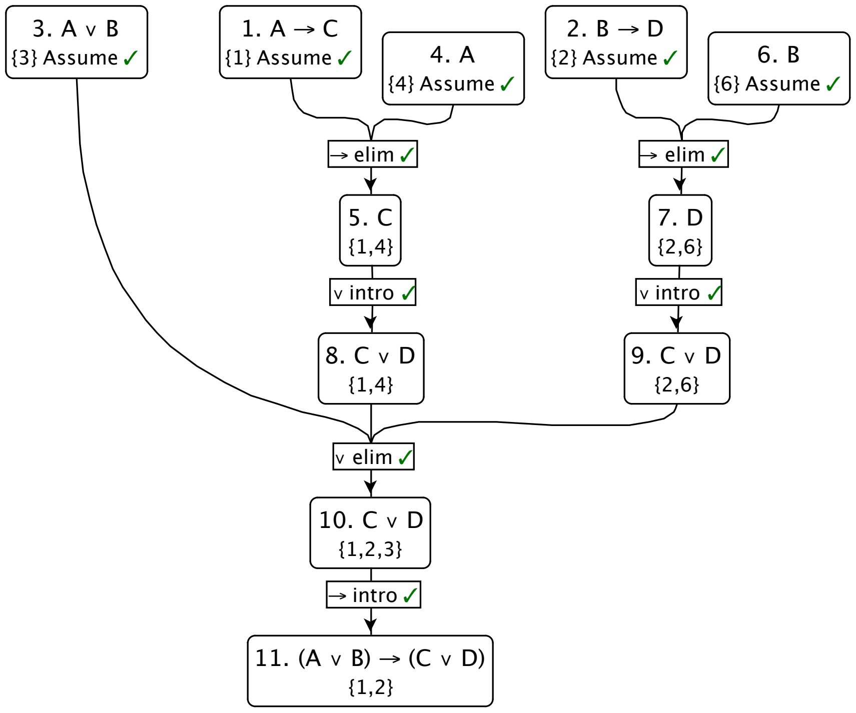


Figure F.4: In Slate, some rules, such as conditional introduction and disjunction elimination, discharge assumptions. This stands in contrast to the default behavior where the set of in-scope assumptions of a formula is the union of its premises' in-scope assumptions.

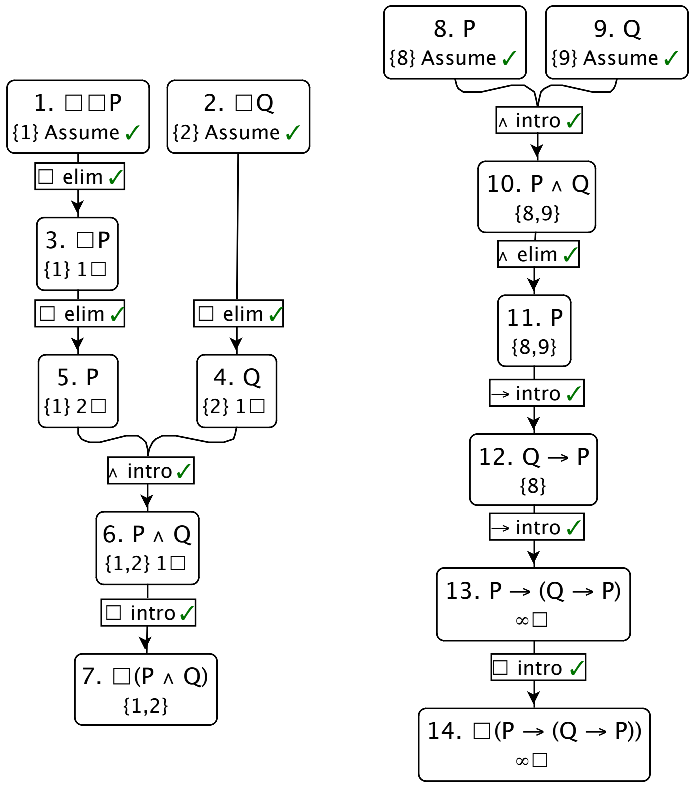


Figure F.5: By default, in **T**, a formula's necessity count is the minimum necessity count of its premises. Exceptions include necessity elimination which increases necessity count, necessity introduction which decreases it, and rules that discharge assumptions which have a more complicated behavior.

by Slate, by means of a necessity-count attribute. Natural deduction proof systems for modal logics are somewhat underrepresented in the usual literature (e.g., Chellas, 1980), but do have a long standing history (Fitch, 1952, 1966; Siemens, 1977), and, in fact, neither is our "necessity counting" approach without precedent (Hawthorn, 1990). As a testament to the utility of this approach to modal logics, we observe that most familiar modal logics (including **K**, **T**, **S4**, and **S5**) all admit proof systems that use the same set of inference rules and differ only in their default calculation of a formula's necessity count.

Slate's proof system for first-order logic is very similar a Gentzen-style first-order calculus. Particularly, universal introduction requires that the name serving to identify the "arbitrary individual" in its premise must not appear free any undischarged in-scope assumption. An analogous restriction holds for existential elimination. The set of free names appearing within in-scope assumptions could be taken as another attribute associated with formulae, though the first-order calculus is not actually implemented this way in Slate. Since each formula already has its set of in-scope assumptions as an attribute, it is straightforward to determine whether the relevant names are free in any of them on demand. Figure F.6 shows proofs using some of the first-order rules.

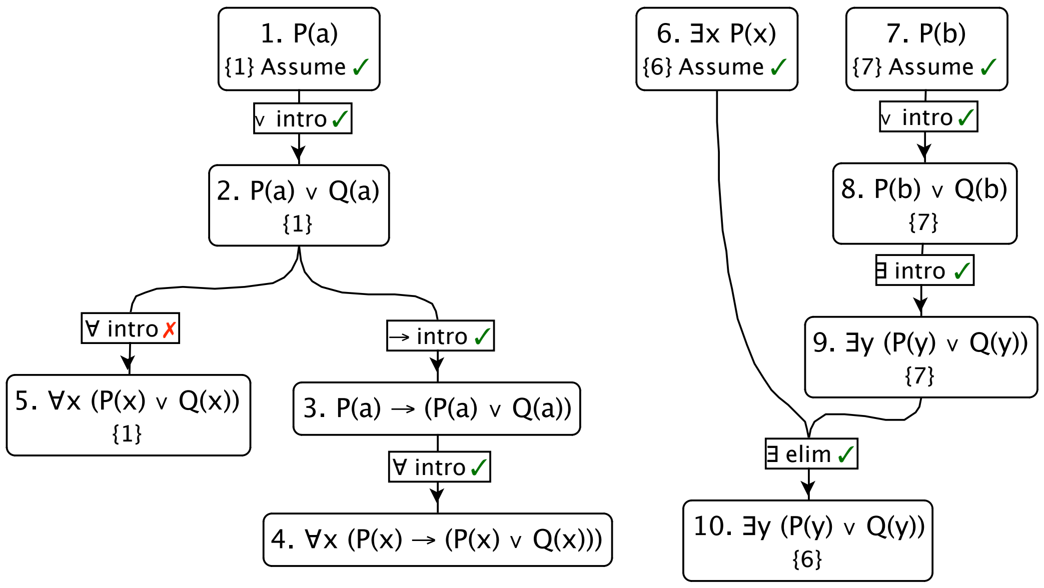
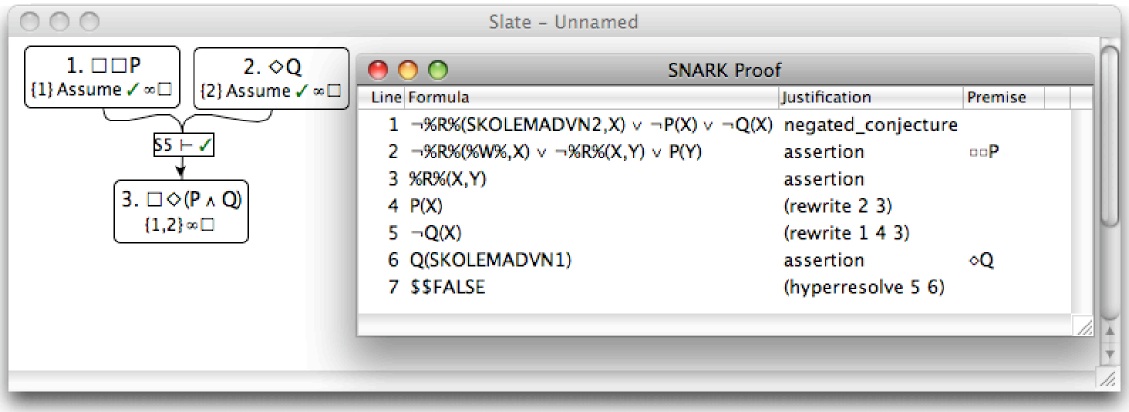


Figure F.6: Universal introduction and existential elimination require that the relevant name (of an arbitrary individual and of a witness, respectively) do not appear free in any undischarged assumptions of their premise. As a consequence, the attempt to infer $\forall x\,(P(x) \vee Q(x))$ from $P(a) \vee Q(a)$ fails since a appears free in the undischarged assumption $P(a)$.

Slate also incorporates rules that leverage automated theorem provers (e.g., McCune's (2003) Otter and Stickel et al.'s (1994) Snark (also see Stickel et al., 2000)) and SAT solvers and model finders (e.g., Claessen & Sorensson's (2003) Paradox). Correct applications of rules that apply theorem provers can be inspected to reveal the proof found by the theorem prover. Dually, examination of failed applications of rules that employ model finders exposes the countermodels that satisfy the premises of the application and the negated conclusion.

Most automated theorem provers are based on resolution rather than natural deduction (though there are notable exceptions, e.g., see Pollock's (1995) Oscar), and the proofs they generate are usually not particularly enlightening to human reasoners. This problem is exacerbated by the fact that, for some proof systems, formulae must be translated before an automated reasoner can be applied, and that the resulting proof may not be easily 'untranslated.' For instance, a first-order theorem prover can be used to prove results about propositional modal logic via the first-order encoding of Kripke-style semantics, but the resulting first-order proofs are obviously not proofs in the propositional modal system, and the correspondence may not be easily accessible. Figure F.7 shows a proof resulting from an application of **S5** $\vdash$ which translates propositional modal formulae into first-order formula to which it applies SNARK. The resolution-based proof in first-order logic is not likely to be at all enlightening to the casual user. It is hoped that the present research will help in the construction of proof translations that can make the products of automated reasoners more easily accessible.



Line	Formula	Justification	Premise
1	¬%R%(SKOLEMADVN2,X) ∨ ¬P(X) ∨ ¬Q(X)	negated_conjecture	
2	¬%R%(%W%,X) ∨ ¬%R%(X,Y) ∨ P(Y)	assertion	□□P
3	%R%(X,Y)	assertion	
4	P(X)	(rewrite 2 3)	
5	¬Q(X)	(rewrite 1 4 3)	
6	Q(SKOLEMADVN1)	assertion	◇Q
7	$$FALSE	(hyperresolve 5 6)	

Figure F.7: The **S5** ⊢ rule confirms that $\Box\Diamond(P \,\&\, Q)$ is derivable in **S5** from $\Box\Box P$ and $\Diamond Q$, but the resolution refutation based on the first-order encoding of the Kripke-style semantics of **S5** is hardly enlightening.

For a more complete summary of the proof systems available in Slate, including their particular syntax, inference rules, and the details of assumption tracking and necessity counting, the reader is directed to *Getting Started with Slate* (Taylor et al., 2010), a guide originally prepared for students using Slate in Rensselaer's introductory logic course, or Bringsjord & Taylor's (2014) textbook in preparation, *Logic: A Modern Approach*.