Deduction vis-a-vis Computation: the need for a formal language for proof engineering

Kostas Arkoudas

June 11, 1998

Abstract

We identify the need for a formal language for *proof engineering*, a term that we use to refer to the process of designing, developing and maintaining complex proofs. We start with a review and classification of the most prominent formal deductive platforms, explaining why they are inappropriate for proof engineering. Specifically, we argue against the traditional view of deductions as static objects such as sequences or trees of formulas, devoid of rich semantic content. We propose instead that, for the purposes of proof engineering, deductions should be written in modern, structured languages endowed with their own special brand of semantics. To motivate this view we introduce and justify a parallel with computation: deductions expressed in traditional frameworks are likened to computer programs written in extremely primitive machine languages; deductions expressed in the kind of framework we suggest are likened to programs written in high-level languages such as ML. Furthermore, we argue that modern systems such as HOL fail to lift the level of abstraction to a sufficiently high degree because they are foundationally based on the traditional, static view of deductions.

As a concrete solution we present a new language called Athena, featuring a two-layered syntax that enforces a crisp demarcation between computation and inference. The block structure of Athena's syntax proves ideal for formally capturing *the scope of an assumption*, a notion that is treated very awkwardly in traditional systems. A novel semantics is introduced for this language, based on the concept of an *assumption base*, a semantic abstraction which interacts with the syntactic notion of assumption scope much as environments interact with the notion of lexical scope in conventional programming language theory. The formal semantics of Athena are presented in the "natural dynamic" style of Kahn, and a polymorphic type system in the spirit of Hindley-Milner is introduced. The logic part of Athena is a natural deduction system that incorporates recursion, polymorphism, and higher-order values. We present several examples that illustrate how these standard programming tools, when coupled with the novel syntax and semantics of Athena, enable one to elegantly formulate sophisticated deductions and deduction schemas (which we call "methods") at a high level of abstraction. We prove Athena's soundness and completeness, discuss its extensional power, and conclude with a section on pragmatics that addresses certain efficiency issues.

1 Introduction

The notion of deduction, although fundamental in all mathematical and scientific thinking, has so far remained remarkably bare in formal treatments of the subject, especially in comparison to the notion of computation. Both concepts have been thoroughly investigated theoretically, but while countless formal languages have been developed for expressing computations at a high level of abstraction, little similar work has been done for deductions. Indeed, on the negative side we will argue for the thesis that *formal* proofs[1] are still expressed at the logical equivalent of assembly language.

This disparity might well be traceable to socio-economic factors: computers are tools of paramount financial importance in the modern business world, and it is only natural that a great deal of research would be aimed at discovering new and improved methodologies for programming them. The same is not the case for logic, but this is not to say that the notion of deduction is any less rich than that of computation, or any less in need of powerful abstractions. Accordingly, on the constructive side we will present Athena, a new formal language for expressing deductions which, in our view, confers the same benefits to logical reasoning that the move from assembly language to, say Pascal, confers to computer programming.

[1] We will use the terms "deduction" and "proof" interchangeably. Most authors—though not all—prefer to use the latter term in reference to a deduction from the empty set of premises.

Athena is a two-tiered language capable of expressing both computations and deductions, as well as integrating the two in an appropriate manner, but let us focus on the logic part for the moment. The main intuition behind the deductive framework of Athena can be stated as follows:

> *An Athena expression D represents a (possibly invalid) deduction. The value of D is simply the proposition deduced, if the deduction is valid, and a special "error" token otherwise.*

Thus, disregarding higher-order values for now, there are two kinds of values in the universe of Athena: propositions and the distinguished element *error*.

This is perfectly analogous to the notion of computation in conventional programming languages. Consider, for instance, a simple first-order functional programming language $\mathcal{L}$ for computing with integers. An expression E in $\mathcal{L}$, say `(+ (* 3 2) (- 8 5))` represents a (possibly invalid) computation. The *value* of E is simply the computed number, if the computation is valid, and an error token of some form or another otherwise. Thus, there are two kinds of values in the semantic universe of $\mathcal{L}$: numbers, and a special error token.

The analogy extends further to include functions. Expressions denoting *particular* (concrete) computations, such as `(+ (* 3 2) (- 8 5))` above, are often called **ground**. Ground expressions are extremely limited in their utility, and a language which restricted us to them would be of little interest and practical value—it would be no more than a simple calculator. A real programming language is born the minute we observe that particular constants in an expression E might be replaced by *variables* without changing the essence of the computation represented by E—the point at which we make the *abstraction* leap. An expression with variables such as `(+ (* x x) (* y y))` might be described as a *computation schema*, representing infinitely many concrete computations, corresponding to the infinitely many values that x and y may assume. This is precisely the idea behind functions (which are also known as "abstractions"). If we just tag the variables x and y above so as to formally mark them as *parameters*, by attaching them to some binding operator such as `lambda`, we get a function `(lambda (x y) (+ (* x x) (* y y)))` which can be applied to *any* two given numbers to initiate a computation that will produce the sum of their squares. Armed with (i) a solid initial supply of "primitive" functions (addition, multiplication, etc.), and (ii) an `if` or `cond` expression that can dynamically determine whether the actual argument(s) satisfy certain predicates and perform different computations accordingly (is the input `x` even? if yes, do this, else do that), we can, in the presence of unbounded recursion, write functions that perform any computation that can be performed mechanically (Church's thesis).

Likewise, what are traditionally known in logic as *deduction schemas* (deductions that use proposition metavariables rather than particular propositions) are really nothing but functions that may be applied to particular propositions (the "actual arguments", or rather "the actual premises", in our case) to derive from them a particular conclusion (the "value" of the function for the given inputs). This is precisely the role that the `method` construct plays in Athena. Armed with a solid initial supply of "primitive" methods (Modus Ponens, double negation, etc.), and a `match` expression that can dynamically determine the structure of the input propositions, we can, in the presence of recursion, write methods that perform any deduction that can be performed mechanically. This is an empirical statement concerning the extensional power of Athena akin to Church's thesis for a programming language. We will have more to say on these issues later. More importantly, we will see that the analogy we have been making extends perfectly well to the higher-order case.

For introductory purposes, this paper covers classic propositional logic only. Two forthcoming documents will treat classic and intuitionist predicate logic, propositional modal logic, finite type theory, and applications of our framework to verification of pragmatic software properties. The rest of this paper is organized as follows. Section 2 will survey and classify the major formalizations of deduction. Section 3 will criticize them, explaining why proof engineering should be shielded from the low level of detail inherent in such systems. In Section 4.1 we introduce the syntax of Athena. In Section 4.2 we give an informal exposition of the semantics, as well as numerous examples taken from transcripts of real sessions with an Athena interpreter. In Section 4.3 we introduce a minimal amount of "syntax sugar". In Section 4.4 we present several additional examples, including some that illustrate how recursion and higher-order values can be used to write powerful methods. In Section 4.5 we present the formal semantics of Athena. In Section 4.6 we introduce a polymorphic type system. In Section 4.7 we treat meta-theoretical issues: we prove the soundness and completeness of Athena, and discuss its extensional power. Finally, in Section 4.8 we show how to write "optimized methods" to achieve maximum efficiency.

2 Traditional formalizations of deduction

In this section we survey the most common existing formalizations of deduction. We begin by reviewing the fundamental ideas from an abstract perspective.

2.1 Deducibility relations and inference rules

Let S be a set whose elements are to be thought of as the sentences of some logic language (for our present purposes, propositional logic). A binary relation $\vdash$ from $\mathcal{P}(S)$ to S is called a **deducibility relation**[2] on S [15] iff it satisfies the following properties:

- **reflexivity** (a.k.a. *inclusiveness*, or *extensiveness*):

$$\Gamma \cup \{A\} \vdash A;$$

- **transitivity** (a.k.a. the *cut property*, or the *lemma generation property*):

$$\text{if } \Gamma \vdash A \text{ for every } A \in \Delta \text{ and } \Delta \vdash B, \text{ then } \Gamma \vdash B;$$

- **monotonicity** (a.k.a. *dilution*, or *weakening*, or *thinning*):

$$\text{if } \Gamma \vdash A \text{ then } \Gamma \cup \Delta \vdash A$$

for all $\Gamma, \Delta \subseteq S$ and $A, B \in S$. These properties were singled out by Alfred Tarski [31, 32] in the 1930s and have quite intuitive interpetations. Another property of interest is *compactness*: $\vdash$ is compact iff $\Gamma \vdash A$ only if $\Gamma' \vdash A$ for some finite subset $\Gamma' \subseteq \Gamma$.

The collection of all deducibility relations on S is a closure system [35], and hence deducibility relations may be generated by arbitrary subsets $\mathcal{R} \subseteq \mathcal{P}(S) \times S$. There are many ways to carve out such initial "kernel" relations. One of them is algorithmic [11]; e.g. we might say that $\{A_1, \ldots, A_n\} \vdash A$ iff a certain resolution engine can derive the empty clause from a given set of clauses representing the CNF of $A_1 \wedge \cdots \wedge A_n \wedge \neg A$; or if a tableau-construction algorithm produces a finished tableau for the root signed sentence $F[A_1 \wedge \cdots \wedge A_n \Rightarrow A]$ in which every branch is closed [27, 9]; or if the Davis-Putnam procedure [7] verifies the unsatisfiability of $\neg(A_1 \wedge \cdots \wedge A_n \Rightarrow A)$; and so on. A more customary (and more properly "logical") approach is to use *inference rules*. A **first-order inference rule** can be defined as an $(n+1)$-ary relation R on S, i.e. a subset of S^{n+1}, for some $n \geq 0$ which

[2] A.k.a. *derivability relation* [17], or in more semantically inclined contexts, *consequence relation* [26, 10].

we will call the **arity** of the rule. When $n = 0$ we say that R is an *axiom.* Thus an axiom is simply a set of sentences, which are said to be the axiom's *instances.* When $A_1 \cdots A_n \; R \; A$ we say that the *conclusion* A is obtained from the *premises* $A_1, \ldots, A_n$ through the rule R. We are using the term "first-order" to emphasize that the premises must be individual elements of S, that is, individual sentences.

An n-ary inference rule R is usually specified graphically as

$$\frac{\pi_1, \ldots, \pi_n}{\pi}$$

where the π's are patterns containing variables that may range over any specified domains. Each of these patterns is "of type S", that is, when all the pattern variables are properly instantiated the resulting expression denotes an element of S. The defined relation R is understood to contain all and only those tuples in S^{n+1} that *match* the compound pattern $(\pi_1, \ldots, \pi_n, \pi)$. Additional constraints may be imposed on the side, further delimiting the extension of R.

Now let $\mathcal{R}$ be any collection of first-order inference rules. We may refer to the pair $(S, \mathcal{R})$ as an *abstract first-order deduction system*; when S is fixed or immaterial we simply speak of $\mathcal{R}$ as a "deduction system". A typical example is the system $(\mathbf{P}, \mathcal{K})$ shown in Fig. 1. Here $\mathbf{P}$ consists of the sentences of propositional logic (using a generous supply of connectives: $\neg$, $\wedge$, $\vee$, $\Rightarrow$, $\Leftrightarrow$, $\top$, and $\bot$) and $\mathcal{K}$[3] comprises fourteen axioms and Modus Ponens. Note that each axiom either *introduces* or *eliminates* a connective. E.g. [A6] introduces $\wedge$ (and will thus be of relevance when we want to infer a conjunction), while [A7] and [A8] "eliminate" $\wedge$, allowing us to derive either the left or the right component of a conjunction. We find exactly the same arrangement for [A9]–[A11] (disjunctions), and [A12]–[A14] (biconditionals). Axioms [A3] and [A4] both introduce $\Rightarrow$. There are no axioms for eliminating $\Rightarrow$; this is accomplished by the MP rule. [A5] eliminates negations, but it can also introduce them when used in tandem with a theorem of the form $\neg\neg A \Rightarrow A$. Finally, [A1] and [A2] can be viewed as unconditional introductions and eliminations of $\top$ and $\bot$, respectively.

This neat and insightful classification of each axiom and rule according to whether it introduces or eliminates a connective dates back to the work of Gentzen [29]. A system that has intuitive axioms and rules for introducing and eliminating each connective is called a *natural deduction system.* $\mathcal{K}$ achieves this for every connective with one very important exception: the conditional connective $\Rightarrow$. In particular, the axioms that $\mathcal{K}$ offers for directly introducing conditionals, [A3] and [A4], fail to capture the way in which implications are established in informal mathematical arguments. This defect is not peculiar to $\mathcal{K}$; as we will explain shortly, *no* first-order deduction system can have a "natural" rule for establishing implications.[4] For this reason, first-order systems (a.k.a. *Frege* or *Hilbert* systems) cannot support natural deduction.[5]

---

[3] This system is a slightly modified version of one given by Kleene in his book "Mathematical Logic" [20].

[4] Although if one is content to construct proofs at the "meta" level, one can reap the benefits of a natural-style $\Rightarrow$-introduction rule by employing the so-called "deduction theorem": if B is *derivable* from $\Gamma \cup \{A\}$ then $A \Rightarrow B$ is derivable from Γ.

[5] It is sometimes claimed in the literature that the main source of difficulty in systems like $\mathcal{K}$ is the use of axioms rather than rules. In particular, the fact that axioms require *forward reasoning*, which engenders tough search problems (it is not easy to figure out where to start), whereas rules allow for backward reasoning, which is goal-oriented and focused. This is not true, however, because in the presence of Modus Ponens axioms and rules are completely interchangeable: to every axiom of the form $A_1 \Rightarrow (A_2 \Rightarrow \cdots (A_n \Rightarrow A_{n+1}) \cdots)$ we can associate an inference rule of the form

$$\frac{A_1, \ldots, A_n}{A_{n+1}}$$

and vice versa. Thus backwards reasoning can easily be simulated in axiomatic systems. For instance, to establish a conjunction $A \wedge B$ in $\mathcal{K}$ we can use axiom [A6] "backwards" to generate the subgoals A and B. If we manage to

[A1]	$\top$
[A2]	$\neg\bot$
[A3]	$A \Rightarrow (B \Rightarrow A)$
[A4]	$[A \Rightarrow (B \Rightarrow C)] \Rightarrow [(A \Rightarrow B) \Rightarrow (A \Rightarrow C)]$
[A5]	$(\neg A \Rightarrow B) \Rightarrow [(\neg A \Rightarrow \neg B) \Rightarrow A]$
[A6]	$A \Rightarrow (B \Rightarrow (A \wedge B))$
[A7]	$A \wedge B \Rightarrow A$
[A8]	$A \wedge B \Rightarrow B$
[A9]	$(A \Rightarrow C) \Rightarrow [(B \Rightarrow C) \Rightarrow (A \vee B \Rightarrow C)]$
[A10]	$A \Rightarrow A \vee B$
[A11]	$B \Rightarrow A \vee B$
[A12]	$(A \Rightarrow B) \Rightarrow [(B \Rightarrow A) \Rightarrow (A \Leftrightarrow B)]$
[A13]	$(A \Leftrightarrow B) \Rightarrow (A \Rightarrow B)$
[A14]	$(A \Leftrightarrow B) \Rightarrow (B \Rightarrow A)$
[MP]	$\dfrac{A \Rightarrow B \quad A}{B}$

Figure 1: Axioms and inference rules of a first-order deduction system for propositional logic.

A **deduction** of a sentence A from a set of sentences Γ in a system $\mathcal{R}$ (or an "$\mathcal{R}$-deduction of A from Γ" for short) is defined as a *sequence* $A_1, \ldots, A_k$ such that $A_k = A$ and each A_i is either an (instance of an) axiom, or a member of Γ (an *assumption*), or else follows from $n \geq 1$ earlier sentences through some n-ary inference rule $R \in \mathcal{R}$. We write $\Gamma \vdash_{\mathcal{R}} A$ to indicate that there is a deduction of A from Γ. The following is not difficult to prove:

Proposition 2.1 *$\vdash_{\mathcal{R}}$ is the least deducibility relation on S that includes every rule $R \in \mathcal{R}$.*

As an example, the following is a $\mathcal{K}$-deduction of $A \Rightarrow C$ from $\{A \Rightarrow B, B \Rightarrow C\}$:

1.	$(B \Rightarrow C) \Rightarrow (A \Rightarrow (B \Rightarrow C))$	[A3]
2.	$B \Rightarrow C$	Assumption
3.	$A \Rightarrow (B \Rightarrow C)$	1, 2, MP
4.	$(A \Rightarrow (B \Rightarrow C)) \Rightarrow ((A \Rightarrow B) \Rightarrow (A \Rightarrow C))$	[A4]
5.	$(A \Rightarrow B) \Rightarrow (A \Rightarrow C)$	4, 3, MP
6.	$A \Rightarrow B$	Assumption
7.	$A \Rightarrow C$	5, 6, MP

establish each of these then one application of MP to A and $A \Rightarrow (B \Rightarrow (A \wedge B))$ will yield $B \Rightarrow (A \wedge B)$, and a second application of MP to this and B will yield the desired $A \wedge B$. Suitability for backwards reasoning is determined by the subformula property, not by whether we are dealing with a rule or an axiom. What complicates the construction of deductions in systems like $\mathcal{K}$ is the fact that there is no intuitive rule for introducing conditionals (as will be attested by anyone who has ever tried to prove $A \Rightarrow A$ in such a system), not the preferential treatment given to axioms. The real impediment is the first-order limitation.

As defined above, deductions are **linear**: they are sequences of sentences. An alternative with a two-dimensional flavor is to define deductions as *trees*. In particular, we stipulate that any single-node tree containing a member A of a set of sentences Γ or an instance A of some axiom in $\mathcal{R}$ constitutes an $\mathcal{R}$-deduction of A from Γ; and that if the trees $T_1, \ldots, T_n$ are $\mathcal{R}$-deductions of $A_1, \ldots, A_n$ from Γ and $A_1 \cdots A_n \; R \; A$ for some $R \in \mathcal{R}$, then the tree whose root is A and whose subtrees are $T_1, \ldots, T_n$ is an $\mathcal{R}$-deduction of A from Γ.

It is clear that the two definitions are extensionally equivalent. And both allow us to reason about proofs inductively. Trees exhibit more structure and symmetry, and this makes them the usual choice in proof-theoretic studies [30]. On the other hand, there is a certain kind of redundancy in trees that is not found in sequences: a sentence will occur as many times in a deduction tree as it is used as a premise in the application of a rule. Assumptions, in particular, tend to be used more than once, and thus oftentimes the frontier of a deduction tree contains several repetitions.

2.2 Second-order inference rules

First-order inference rules cannot intuitively capture the derivation of conditionals. By far the most common and natural way of deriving a conditional $A \Rightarrow B$ is this: *take A as a working assumption and attempt to deduce B; if successfull, "discharge" A and infer $A \Rightarrow B$.* But observe that any inference rule which manages to encapsulate this principle must be a "meta-rule", in the sense that its premises must refer to entire deductions, not merely to sentences. Such a rule must, in effect, say "If there is a *deduction* of B from A, then infer $A \Rightarrow B$". A first-order rule is clearly unable to achieve this since, by definition, a first-order rule is a subset of S^{n+1} and thus its premises are always individual elements of S. What we need are second-order rules, i.e., rules whose premises can be more complicated objects capable of asserting the existence of deductions. If we make the leap to second-order systems then we can have intuitive introduction and elimination rules for *all* the usual logical connectives, including $\Rightarrow$. For this reason, most second-order systems are of the "natural deduction" variety. In an attempt to bring some order into the picture, we might partition second-order systems into those that are *sequent-based* and those that are *sentence-based*. In systems of the former kind the fundamental units of deduction are *sequents*,[6] usually represented as pairs (Γ, Δ) in *Gentzen-type* sequent-systems and pairs (Γ, A) in *natural-type* sequent systems,[7] where Γ and Δ are finite sets of sentences.[8] They are usually denoted by the more suggestive notations $\Gamma \rightarrow \Delta$ and $\Gamma \rightarrow A$. We call Γ the *antecedent*, and Δ and A the *succedent* of the respective sequents. Following custom, we write $\Gamma, A_1, \ldots, A_n$ for $\Gamma \cup \{A_1, \ldots, A_n\}$ (or more generally, Γ, Δ for $\Gamma \cup \Delta$) and $\rightarrow A$ for $\emptyset \rightarrow A$.

As their name suggests, "natural-type" sequent-based systems are natural deduction systems: they have intuitive introduction/elimination rules for every connective. A typical specimen is shown in Fig. 2; we will refer to it as "system $\mathcal{M}$". A sequent $A_1, \ldots, A_n \rightarrow A$ in such a system essentially means the same thing as $\{A_1, \ldots, A_n\} \models A$, or equivalently, $\models (A_1 \wedge \cdots \wedge A_n) \Rightarrow A$. This, however, is a matter of viewpoint. For instance, natural-type sequent-based systems naturally arise as "meta-systems" for Hilbert-style first-order systems such as $\mathcal{K}$ (in particular, once we have a constructive proof of the deduction theorem for $\mathcal{K}$ it becomes quite easy to devise an algorithm which converts every deduction in the system $\mathcal{M}$ to a corresponding deduction in $\mathcal{K}$), and from that perspective

[6] "Things that follow" in Latin.

[7] Sequents of the form (Γ, A) also arise in intuitionistic versions of the classic Gentzen calculi (the "LJ" versions of the "LK" systems).

[8] The use of sets is a fairly recent trend. Customarily finite *sequences* were used instead. This necessitated an additional group of so-called *structural rules*, namely *exchange* (a.k.a. *permutation*), which allows us to infer Γ, B, A, Δ from Γ, A, B, Δ; and *contraction*, allowing us to derive Γ, A, Δ from Γ, A, A, Δ. Working with sets allows us to do away with these rules.

$$[\text{MA1}]\ \frac{}{A \to A} \qquad [\text{MR1}]\ \frac{\Gamma \to A}{\Gamma, \Delta \to A}$$

$$[\text{MA2}]\ \frac{}{\to \top} \qquad [\text{MA3}]\ \frac{}{\to \neg\bot}$$

$$[\text{MR2a}]\ \frac{\Gamma, A \to B}{\Gamma \to A \Rightarrow B} \qquad [\text{MR2b}]\ \frac{\Gamma \to A \Rightarrow B \quad \Gamma \to A}{\Gamma \to B}$$

$$[\text{MR3a}]\ \frac{\Gamma, A \to B \quad \Gamma, A \to \neg B}{\Gamma \to \neg A} \qquad [\text{MR3b}]\ \frac{\Gamma \to \neg\neg A}{\Gamma \to A}$$

$$[\text{MR4a}]\ \frac{\Gamma \to A}{\Gamma \to A \vee B} \qquad [\text{MR4b}]\ \frac{\Gamma \to A}{\Gamma \to B \vee A} \qquad [\text{MR4c}]\ \frac{\Gamma, A \to C \quad \Gamma, B \to C}{\Gamma, A \vee B \to C}$$

$$[\text{MR5a}]\ \frac{\Gamma \to A \quad \Gamma \to B}{\Gamma \to A \wedge B} \qquad [\text{MR5b}]\ \frac{\Gamma \to A \wedge B}{\Gamma \to A} \qquad [\text{MR5c}]\ \frac{\Gamma \to A \wedge B}{\Gamma \to B}$$

$$[\text{MR6a}]\ \frac{\Gamma \to A \Rightarrow B \quad \Gamma \to B \Rightarrow A}{\Gamma \to A \Leftrightarrow B} \qquad [\text{MR6b}]\ \frac{\Gamma \to A \Leftrightarrow B}{\Gamma \to A \Rightarrow B} \qquad [\text{MR6c}]\ \frac{\Gamma \to A \Leftrightarrow B}{\Gamma \to B \Rightarrow A}$$

Figure 2: A natural-type sequent-based deduction system for classic propositional logic.

one might identify $\Gamma \to A$ with the assertion $\Gamma \vdash_{\mathcal{K}} A$. It is in this light that the second-order nature of these systems becomes evident:[9] a rule such as [MR2a] is now construed as saying "*if there is a $\mathcal{K}$-deduction of B from A and Γ*, infer $A \Rightarrow B$ from Γ". Of course, by the completeness of $\mathcal{K}$-like systems, such viewpoints are equivalent, since in fact $\{A_1, \ldots, A_n\} \vdash_{\mathcal{K}} A$ iff $(A_1 \wedge \cdots \wedge A_n) \Rightarrow A$ is valid. Natural-type sequent systems tend to be regarded as "formal" natural deduction alternatives to the visually—and hence informally—defined sentence-based systems to which we will turn shortly. They can be found in several Mathematical Logic texts [8, 1, 21, 37]. The HOL system [13] also uses sequent-based natural deduction.

In Gentzen-type sequent systems, of which a representative sample is shown in Fig. 3, a sequent $\Gamma \to \Delta$ is interpreted as the assertion that whenever all the members of Γ are true, at least one member of Δ is also true. Such systems feature a "left" and a "right" rule for each connective, serving to introduce the connective in the antecedent and in the succedent of a sequent, respectively. These rules exhibit a remarkable degree of symmetry and are further characterized by the very important *subformula property*: every sentence appearing in a premise of of a rule is a proper part of the "principal" sentence in the rule's conclusion. This is not the case when a cut rule[10] is included in the system, as was in Gentzen's original formulation, but in those cases it can be (constructively) proved that for

[9] For sentence-based systems the second-order element is apparent: conditional introduction rules in such systems explicitly hypothesize the existence of an appropriate deduction as a premise.

[10] A cut rule is essentially a syntactic rendition of the transitivity property of deducibility relations that allows for the use of lemmas. Modus Ponens is a restricted version of the cut.

$$\frac{}{\Gamma, A \to A}\ [\text{Init}]$$

$$\frac{}{\Gamma, \bot \to \Delta}\ [\bot\,\text{L}] \qquad \frac{}{\Gamma \to \Delta, \top}\ [\top\,\text{R}]$$

$$\frac{\Gamma \to \Delta, A}{\Gamma, \neg A \to \Delta}\ [\neg\,\text{L}] \qquad \frac{\Gamma, A \to \Delta}{\Gamma \to \Delta, \neg A}\ [\neg\,\text{R}]$$

$$\frac{\Gamma, A, B \to \Delta}{\Gamma, A \wedge B \to \Delta}\ [\wedge\,\text{L}] \qquad \frac{\Gamma \to \Delta, A, B}{\Gamma \to \Delta, A \wedge B}\ [\wedge\,\text{R}]$$

$$\frac{\Gamma, A \to \Delta \quad \Gamma, B \to \Delta}{\Gamma, A \vee B \to \Delta}\ [\vee\,\text{L}] \qquad \frac{\Gamma \to \Delta, A, B}{\Gamma \to \Delta, A \vee B}\ [\wedge\,\text{R}]$$

$$\frac{\Gamma \to \Delta, A \quad \Gamma, B \to \Delta}{\Gamma, A \Rightarrow B \to \Delta}\ [\Rightarrow\text{L}] \qquad \frac{\Gamma, A \to \Delta, B}{\Gamma \to \Delta, A \Rightarrow B}\ [\Rightarrow\text{R}]$$

$$\frac{\Gamma, A, B \to \Delta \quad \Gamma \to \Delta, A, B}{\Gamma, A \Leftrightarrow B \to \Delta}\ [\Leftrightarrow\text{L}] \qquad \frac{\Gamma, A \to \Delta, B \quad \Gamma, B \to \Delta, A}{\Gamma \to \Delta, A \Leftrightarrow B}\ [\Leftrightarrow\text{R}]$$

Figure 3: A cut-free Gentzen-type sequent calculus for classic propositional logic.

any deduction that uses the cut rule we can find one that does not; such results are known as *cut elimination theorems*, and allow us to present such systems in a cut-free form obeying the subformula property, as in Fig. 3. The subformula property has very profound ramifications for metatheory, especially for obtaining consistency results, but an important added bonus for computer scientists is that it renders a system amenable to *analytic* search methods for automatic proof construction of a very constructive flavor. By starting at the goal and using the rules backwards one will essentially construct a Beth tableau that will either establish the goal or *provide a counter-example*.

Both Gentzen-type and natural-type systems can be viewed as first-order systems on the set of all sequents ($\mathcal{P}_\infty(\mathbf{P}) \times \mathcal{P}_\infty(\mathbf{P})$ and $\mathcal{P}_\infty(\mathbf{P}) \times \mathbf{P}$, respectively, where $\mathcal{P}_\infty(\mathbf{P})$ is the set of all finite subsets of $\mathbf{P}$). Therefore, deductions in both systems can be cast either in a linear format (sequences of sequents) or a tree-based one (trees of sequents), although customarily trees are preferred for Gentzen deductions and sequences for natural-type deductions. For instance, in the system $\mathcal{M}$ we can define a deduction of a sequent $\Gamma \to A$ from a set of sequents Θ as a sequence $\Gamma_1 \to A_1, \ldots, \Gamma_k \to A_k$ such that $\Gamma_k = \Gamma, A_k = A$, and each sequent $\Gamma_i \to A_i$ is either an instance of a sequent axiom (an instance of [MA1], [MA2], or [MA3]), or a member of Θ, or else follows from earlier sequents via one of the rules [MR1]—[MR6c]. As a simple illustration, here is an $\mathcal{M}$-deduction of $A \wedge B, B \Rightarrow C \to A \wedge C$ from $\emptyset$:

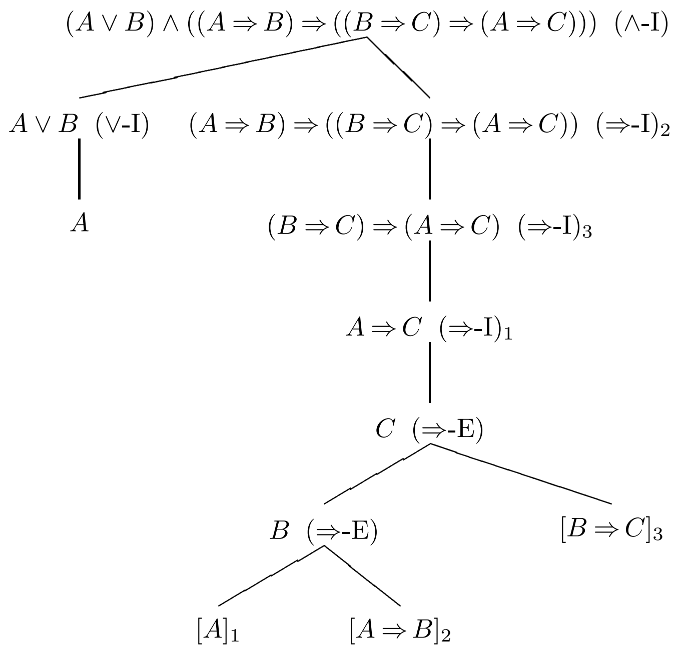


Figure 4: A deduction of $(A \vee B) \wedge ((A \Rightarrow B) \Rightarrow ((B \Rightarrow C) \Rightarrow (A \Rightarrow C)))$ from A in a sentence-based system, in tree form. The subscripts match assumption occurrences with rule applications that discharge them, an awkward treatment of assumption scope that makes these trees difficult to read.

1.	$A \wedge B \rightarrow A \wedge B$	[MA1]
2.	$A \wedge B, B \Rightarrow C \rightarrow A \wedge B$	1, [MR1]
3.	$A \wedge B, B \Rightarrow C \rightarrow B$	2, [MR5c]
4.	$B \Rightarrow C \rightarrow B \Rightarrow C$	[MA1]
5.	$A \wedge B, B \Rightarrow C \rightarrow B \Rightarrow C$	4, [MR1]
6.	$A \wedge B, B \Rightarrow C \rightarrow C$	5, 3, [MR2b]
7.	$A \wedge B, B \Rightarrow C \rightarrow A$	2, [MR5b]
8.	$A \wedge B, B \Rightarrow C \rightarrow A \wedge C$	7, 6, [MR5a]

In *sentence-based* second-order systems the basic deductive units are individual sentences. All systems in this category are of the "natural" kind, offering intuitive introduction and elimination rules for each connective. (This is in fact *the* class of systems that most authors refer to by the term "natural deduction systems", although, as we discussed above, the term also applies to natural-type sequent systems). As before, deductions in these systems can be formulated either as sequences or as trees, but here the distinction is much more significant because it determines the manner in which auxiliary assumptions are handled. In tree-based systems[11] all assumptions appear at the tree's frontier, and are discharged by tagging them and the internal node at which the discharge occurs with

[11] Originally invented by Gentzen and popularized by Prawitz [23], who proved an important normalization result for them analogous to the cut elimination theorems for Gentzen-type sequent systems.

a unique label. A typical representative of these systems is used in Van Dalen's "Logic and Structure" [34]. A sample deduction in such a system is shown in Fig. 4.[12]

Linear sentence-based second-order systems use nestable graphical devices such as boxes to model the scope of auxiliary assumptions.[13] Assumptions appearing at the "top" of a box B are active only inside B, including whatever boxes might be nested inside B to an arbitrary depth (modulo the obvious visual limitations). Usually every sentence is flanked by a number on its left, indicating its relative position in the deduction and also serving as a label for the sentence, and by a "justification" on its right, explaining whether the sentence is an assumption or whether it follows from earlier sentences through an inference rule. Systems of this kind are very popular in introductory logic books [2, 6, 28, 19]. A simple example:

1.	$A \Rightarrow B$	Assumption
2.	$B \Rightarrow C$	Assumption
3.	A	Assumption
4.	B	1,3,MP
5.	C	2,4,MP
6.	$A \Rightarrow C$	3,5,⇒-I
7.	$(B \Rightarrow C) \Rightarrow (A \Rightarrow C)$	2,6,⇒-I
8.	$(A \Rightarrow B) \Rightarrow ((B \Rightarrow C) \Rightarrow (A \Rightarrow C))$	1,7,⇒-I

Finally, Fig. 5 offers a convenient graphical summary of our classification.

[12] Traditionally, such trees are depicted upside-down, i.e. with the root at the bottom and growing upwards (needless to say, this is upside-down only relative to the familiar practice in Computer Science of drawing trees with their roots at the top). We feel that, tradition non-withstanding, there is little justification for this departure. In fact drawing deduction trees from top to bottom, aside from being visually consistent with the way all other trees are drawn in these fields, has at least two additional advantages, one psychological and the other conceptual. Firstly, when a deduction tree is drawn with the leaves at the bottom and the root at the top it resembles a pyramid-like edifice with axioms and assumptions at the foundation and the conclusion at the tip, which is a much more appropriate view than the opposite. Secondly, backward (analytic) and forward (synthetic) reasoning become respectively identified with top-down and bottom-up attempts to build a deduction tree. In the top-down approach we start at the root (the goal) and we apply rules backward (downward) in order to find children for the root, children for their children, and so on, until all the leaves are axioms or assumptions. In the bottom-up approach we start at the leaves (appropriately selected axioms and assumptions) and attempt to synthesize our way up to the desired goal by applying rules forward (upward). This clarifies the conceptual interplay between semantic tableaux and Gentzen tree-style deductions: tableau-construction algorithms simply build a Gentzen-style deduction tree from top to bottom!

[13] Other variations are also possible; e.g. Quine [25] uses a multiple-asterisk system for tagging auxiliary assumptions.

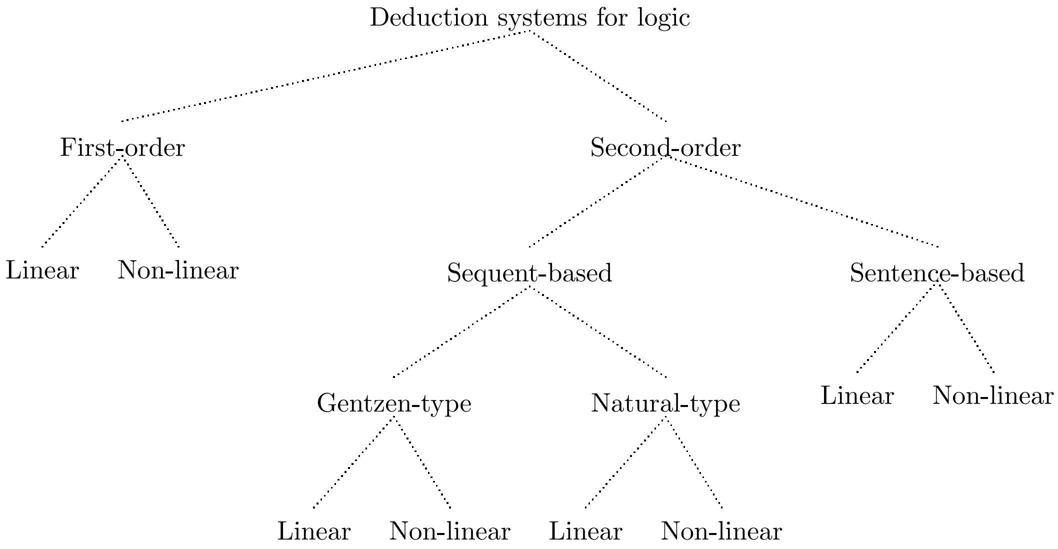


Figure 5: A classification of the major deductive frameworks.

3 Critique of traditional formalizations

The main problem with all of the above formalizations, we will now argue, is that they are too low-level. They are, so to speak, the Turing machines of the deducibility concept. As foundational formal analyses, they reduce deducibility to its most essential rudiments, just as Turing machines or straight-line GOTO languages with a total of three or four commands reduce computability to its most primitive elements. This inevitably means that reasoning *about* these systems is much easier than reasoning *in* them. Continuing our analogy, the *s-m-n* theorem might be easier to prove in the setting of a straight-line GOTO language rather than Modula-3, but when it comes to writing a spreadsheet or a debugger the situation is reversed; straight-line GOTO languages are totally unsuitable for developing, expressing, and maintaining realistic programs. In the same vein, some of the foregoing deductive frameworks are attractive for metatheory purposes precisely because of their austerity. Indeed, in later sections of this paper we will make good use of the definition of deductions as sequences of objects, e.g. when we present the semantics and type system of Athena, when we prove its soundness and completeness, etc. But for the purpose of developing, expressing, and maintaining realistic proofs, all of these frameworks are unsuitable. In fact we will argue that *even* for metatheory purposes most of these systems suffer from serious defects.

Let us begin with first-order systems. Clearly, the most serious problem of these is their inability to accomodate nested deductions based on auxiliary assumptions, a problem so severe that makes them practically unusable ("barbarously unintuitive", to quote Hodges [16]). But there are other problems, caused not by this inability but by the sequence/tree formats themselves. These problems thus carry over to second-order systems, but are best illusrated in the first-order case. Roughly, the analogy by which we wish to convey the problem is this: *Linear (sequence-based) deductions correspond to an extremely primitive form of nameless/stateless assembly language; while non-linear (tree-based) deductions correspond to dataflow graphs.*

Consider the linear version of the first-order system $\mathcal{K}$. A deduction of A from Γ in this system is a sequence of sentences $A_1, \ldots, A_k$ where each A_i is either a member of Γ, or an axiom instance, or else follows from previous sentences through MP. Now imagine a programming language in which a computation is represented as a sequence of numbers, each of which is either an input, or one of a certain set of constants, or the result of applying some operation such as addition or sign inversion to some previous numbers! The computation `(+ (* 3 3) (* 5 5))` would thus be the sequence

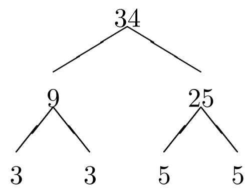
$$\begin{array}{c} 3 \\ 5 \\ 9 \\ 25 \\ 34 \end{array}$$

This would clearly be a horrid language for expressing computations. Of course we could try adorning this basic sceleton with some additional information in an attempt to enhance perspicuity, e.g. we could number the lines on the left and add "justifications" on the right, resulting in

1.	3	input
2.	5	input
3.	9	1, square
4.	25	2, square
5.	34	3, 4, add.

but this would amount to little more than sugarcoating. The point is that people do not write a program as one large sequence of data. They should not have to write their proofs in this manner either.

Consider next the tree-based version of $\mathcal{K}$. Continuing the deduction/computation juxtaposition, imagine as the analogue of this system a programming language in which a computation is expressed as a tree of data, in which every internal node is the result of applying some operation to its children. Thus the `(+ (* 3 3) (* 5 5))` computation might be expressed by the tree



This is essentially a dataflow graph (a tree, in this case). It has the slight advantage of relaxing the strict linear temporal ordering imposed by sequences, but it still has several drawbacks, some more serious than others. For instance, it is not indicated in the tree *which* operation yields each internal datum. This, however, can be easily remedied by tagging each intermediate node with the name of the appropriate operation. Secondly, there is a good deal of redundancy: a number will appear in the tree as many times as it is used, and in this respect the linear versions are much more economical. In the case of computations (i.e. for dataflow graphs) this can be fixed by using structure-sharing dags (directed acyclic graphs), but in the case of deductions trees ("logic-flow graphs") this solution is possible only for first-order systems. In second-order deduction trees the leaves represent sentence *occurrences* rather than just sentences, and their significance can vary depending on whether or not they are tagged (discharged); two edges pointing to two distinct occurrences of the same sentence cannot automatically be redirected and made to point to the same occurrence. As will be discussed shortly, the clumsiness with which assumption scope is handled in second-order sentence-based deduction trees is a very serious defect of these tree systems. But the most serious problem is again the extremely low level of representation. There is no naming mechanism, no abstraction mechanism, no concept of modularization and re-use, none of the familiar engineering principles. All of this, of course, follows trivially from the fact that the syntax and semantics of these trees are extremely primitive. The only thing you can "write" in this "language" is one big tree of formulas, and the only semantics is that the formula at every internal node must be obtainable from those at the children nodes through an inference rule. But people do not write computer programs as huge trees of data; they should not have to write deductions in this manner either. We need a much richer syntax with a much richer semantics.

Second-order systems have the substantial advantage of providing native support for nested deductions, but because they achieve this in a purely syntactic fashion, they pay a very heavy price. Sequent-based systems, in particular, pack all active assumptions (the so-called "context") in a single sequent, and thus explicitly mention all of them at every point, i.e. at every element of a sequence or node of a tree. This is wasteful, since in fact only second-order rules—such as conditional introduction—need to explicitly manipulate the context; and it clutters things up because the fundamental units manipulated by the rules are no longer single sentences but entire groups of them. Moreover, it is not a good model of informal human reasoning. When humans prove a claim they do not drag along all the relevant assumptions at each step of the way. Assumptions are introduced only once; from that point on they are only mentioned if and when they are needed. This is successfully captured in sentence-based second-order systems, but not in sequent-based systems.[14]

Sentence-based second-order systems have troubles of their own. Those that are tree-based are remarkably awkward in handling auxiliary assumptions. The tagging mechanism that such systems use to implement assumption introduction and discharge is non-local: *all* assumptions, auxiliary or nor, appear at the frontier of the tree. Thus the *scope* of an assumption, a concept that is crystal-clear in informal logical reasoning, is not emergent from the representation of a deduction. Linear systems, by using devices like boxes, fare better in this respect, but lack a formal foundation—a "box" drawn on a page is not a mathematically defined object, and is thus not amenable to formal analysis. Futhermore, although the requirement of labelling every line with a number serves as a (very primitive) naming mechanism, it is cumbersome and often unnecessary.

And in the end, over and above these problems, all second-order systems are inevitably plagued by the "data sequence/data-flow graph" syndrome we discussed above. A realistic deduction will either be one long unwieldy sequence of objects (sentences or sequents) or one large unwieldy "logic-flow graph". The use of boxes in linear natural deduction systems imparts some useful structure, but in the end all surveyed systems lack the following features which we claim necessary for proof engineering:

- an intuitive formalization of assumption scope that capitalizes on the block structure of nested deductions, but without requiring the explicit recording of the entire context as sequents do;
- naming mechanisms;
- abstraction and decomposition mechanisms;
- support for natural deduction (introduction/elimination rules);

[14] Indeed, in light of the normalization results obtained by Prawitz for sentence-based second-order systems (of the tree-based "natural deduction" variety, specifically) and the analytic tableau methods developed by Smullyan, there is not even a cogent metatheoretical or methodological reason for using sequents, other perhaps than historical considerations and the aesthetic value of the symmetry that is found in most such (Gentzen-type) systems. This is pointed out by Troelstra and Schwichtenberg in their recent "Basic Proof Theory" [33], pg. 51, although they also state that certain results are more easily obtainable for cut-free proofs in Gentzen systems rather than normal proofs in Prawitz systems.

- recursion; and
- a polymorphic static type discipline.

Now one might point out that most of these tools are already available in systems such as HOL. But the syntax and semantics of HOL are those of ML—a programming language for expressing computations, not deductions. In such systems *the deductive machinery itself is still based on one of the traditional low-level frameworks*. HOL, for instance, uses sequences of sequents. This is inadequate in many respects. For example, assumption scope is not captured by the syntax of HOL. E.g. in an HOL expression such as

```
let val x = ... (* some-proposition *) ...
 in
   ... (* body *) ...
end
```

the `body` coincides with the *computational scope* of `x`, not with its *logical scope*. In other words, `x` names an object that may be used for computation inside `body`, not a hypothesis that may be freely asserted in inferences made inside `body`. Unfortunately, the two scopes may well be different. When we enter the body of the `let`, the value assigned to `x` might not be an active assumption, and to make it an active assumption we have to explicitly insert it into the antecedent of the relevant sequent somewhere within `body`. Similarly, when we leave the body of the `let`, `x` might still be an active assumption, and we then have to "discharge" it by explicitly removing it from the antecedent.

The problem is that the language used here is a programming language, having programming language semantics. Thus the entire `let` expression represents a *computation*, not a deduction. It might be thought of as representing a deduction, if we look inside its body to check that it pushes sequents around in the right manner. But the extra level of indirection required by having to go through the ML semantics in order to understand that this is a computation that simulates a deduction is extraneous and distracting. What is needed is a language whose syntax and semantics are custom-made for deductions, not a programming language that is indirectly used to simulate low-level deductions.

To better illustrate the problem, imagine a system in which ML is used to simulate a low-level abstract RAM machine. All programming in this system must be done by writing ML code that cranks and runs the abstract machine in the desired fashion. Thus if you wanted to write a fast fourier transform in this system, you would still have to be conceptually involved at the level of the machine, that is, you would still be thinking in terms of registers and memory locations and program counters and so forth, only now you would have the luxury of writing ML functions to manipulate all these. But this is not much progress. The obvious question is: why not insulate the users *completely* from the machine language level and let them write their programs directly in ML? Likewise, if you want to write a complex proof in HOL you still have to think in terms of sequences of sequents, and in terms of explicitly moving formulas from the antecedent into the succedent, and so on, only now you have the luxury of writing ML functions to do all that. But this is not much progress. To achieve true abstraction we must completely isolate the users from the traditional low-level deductive frameworks on which systems such as HOL are based. That is exactly what Athena does.

4 Athena: a formal framework for computation and deduction

4.1 Syntax

Since we wish to avoid digressions into peripheral issues regarding concrete syntax (many of which are a matter of taste), we opt for a particularly simple choice: s-expressions. There are seven syntactic domains in the language: **Ide** (identifiers), **Var** (propositional variables, or simply "variables"), **Prop** (propositions), **Pat** (patterns), **Exp** (expressions), **Ded** (deductions), and **Phr** (phrases). We use the letters I, X, P, Q, E, D, and F, possibly adorned with subscripts, primes, etc., to denote arbitrary elements of these six domains, respectively.

I	$\in$	**Ide**	Identifiers
X	$\in$	**Var**	Propositional variables
P	$\in$	**Prop**	Propositions
Q	$\in$	**Pat**	Patterns
E	$\in$	**Exp**	Expressions
D	$\in$	**Ded**	Deductions
F	$\in$	**Phr**	Phrases
X	::=	`(var` I`)`	
P	::=	X	Atoms
	|	`true`	The constant "true"
	|	`false`	The constant "false"
	|	`(not` P`)`	Negations
	|	`(and` P_1 P_2`)`	Conjunctions
	|	`(or` P_1 P_2`)`	Disjunctions
	|	`(if` P_1 P_2`)`	Conditionals
	|	`(iff` P_1 P_2`)`	Biconditionals
Q	::=	I	Pattern variables
	|	X	Constant atom patterns
	|	`(val-of` I`)`	Bound-variable patterns
	|	`any`	The "match-all" pattern
	|	`true`	The constant pattern "true"
	|	`false`	The constant pattern "false"
	|	`(not` Q`)`	Negation patterns
	|	`(and` Q_1 Q_2`)`	Conjunction patterns
	|	`(or` Q_1 Q_2`)`	Disjunction patterns
	|	`(if` Q_1 Q_2`)`	Conditional patterns
	|	`(iff` Q_1 Q_2`)`	Biconditional patterns

Figure 6: Syntax of the Athena kernel, part 1.

E	$::=$	I	Variables
	$\mid$	X	Atomic propositions
	$\mid$	$\texttt{(function (}I_1 \cdots I_n\texttt{)}\ E\texttt{)}$	Functions
	$\mid$	$\texttt{(method (}I_1 \cdots I_n\texttt{)}\ D\texttt{)}$	Methods
	$\mid$	$\texttt{(}E\ F_1\ \cdots\ F_n\texttt{)}$	Function applications
	$\mid$	$\texttt{(clet ((}I_1\ F_1\texttt{)}\ \cdots\ \texttt{(}I_n\ F_n\texttt{))}\ E\texttt{)}$	"Name-and-compute"
	$\mid$	$\texttt{(cletrec ((}I_1\ E_1\texttt{)}\ \cdots\ \texttt{(}I_n\ E_n\texttt{))}\ E\texttt{)}$	Recursive `clet`
	$\mid$	$\texttt{(cmatch}\ F\ \texttt{(}Q_1\ E_1\texttt{)}\ \cdots\ \texttt{(}Q_n\ E_n\texttt{))}$	"Match-and-compute"
D	$::=$	$\texttt{(assume}\ F\ D\texttt{)}$	Hypothetical deductions
	$\mid$	$\texttt{(suppose-absurd}\ F\ D\texttt{)}$	Deductions by contradiction
	$\mid$	$\texttt{(apply-method}\ E\ F_1\ \cdots\ F_n\texttt{)}$	Method applications
	$\mid$	$\texttt{(dlet ((}I_1\ F_1\texttt{)}\ \cdots\ \texttt{(}I_n\ F_n\texttt{))}\ D\texttt{)}$	"Name-and-deduce"
	$\mid$	$\texttt{(dletrec ((}I_1\ E_1\texttt{)}\ \cdots\ \texttt{(}I_n\ E_n\texttt{))}\ D\texttt{)}$	Recursive `dlet`
	$\mid$	$\texttt{(dmatch}\ F\ \texttt{(}Q_1\ D_1\texttt{)}\ \cdots\ \texttt{(}Q_n\ D_n\texttt{))}$	"Match-and-deduce"
F	$::=$	$E \mid D$	

Figure 7: Syntax of the Athena kernel, part 2.

The syntactic domain **Ide** is primitive. The only assumption that we will make about it is that any two of its elements can be effectively tested for equality. The other six domains are defined by the productions shown in Fig. 6 and Fig. 7. This is the kernel of the language; a subsequent section will introduce a few additional syntactic forms as "syntax sugar".

We single out a set of keywords *TopName* = *LogOp* ∪ *PrimRule* where *LogOp* = {`true`, `false`, `not`, `and`, `or`, `if`, `iff`} (the "logical operators"), and *PrimRule* = {`claim`, `mp`, `dn`, `absurd`, `cd`, `both`, `left-and`, `right-and`, `either`, `equiv`, `left-iff`, `right-iff`} (the primitive inference rules). We stipulate that

- no element of *TopName* can occur bound in a phrase, i.e. as a `function` or `method` parameter, as a name introduced by a `clet`, `cletrec`, `dlet`, or `dletrec`, or as a pattern variable, where the variables of a pattern Q, denoted $PtVar(Q)$, are simply the identifiers that occur in Q (not counting the elements of *LogOp*, of course);
- the parameters of a function or a method must be distinct; and
- reserved words (`assume`, `dlet`, etc.) cannot be used as identifiers.

4.2 Informal semantics

The principal syntactic domain of Athena is that of phrases. A phrase is simply either an expression or a deduction. Note that it is immediately apparent from its form whether a given phrase is an expression or a deduction. Roughly, expressions are used to perform arbitrary computations while deductions are used to make logical inferences. Whereas any proposition whatsoever can be produced by evaluating an expression, only those propositions that are logical consequences of the contents of the *assumption base* can be obtained from evaluating a deduction. An assumption base is an abstraction, which we will concretely regard as a recursive set of Athena propositions (i.e. a subset of **Prop** with a computable characteristic function), that forms an integral component of the semantic model of the language. We will make the simplifying convention that *all* assumption bases contain the proposition `true`. Thus when we speak of "the empty assumption base" we will mean the singleton that contains `true`. This is not strictly necessary, but it is technically expedient as it simplifies a number of semantic issues whose treatment would otherwise prove tedious.

The stipulation that assumption bases must be recursive ensures that we can always determine effectively whether an arbitrary proposition is a member of a given assumption base. Of course this is a very liberal condition—it allows for assumption bases with astronomically complex membership problems. But remember that we are currently describing the theory of the language, in its full generality; and in theory there is no good reason to impose an arbitrary upper bound on the amount of effort that one needs to expend in order to settle a claim. In practice most deductions take place in finite assumption bases, usually either in the empty assumption base or in one that contains a small number of select axioms; in all such cases membership checks can be performed with negligible cost.

The basic values in the semantic universe of Athena are **propositions**.[15] A proposition is either an *atom* of the form `(var` I`)`, or one of the two logical constants `true` and `false`, or a *compound* proposition formed through one of the logical connectives `not`, `and`, `or`, `if`, or `iff`. An atom `(var` I`)` will be abbreviated as `?`I. Thus the following are all examples of well-formed propositions:

`?foo`	an atom
`(not ?foo)`	a negation
`(and ?x (not true))`	a conjunction
`(or ?x (not ?x))`	a disjunction
`(if ?foo (or false ?x))`	a conditional
`(iff (and ?x ?y) (and ?y ?x))`	a biconditional

As should be expected, the truth value of a proposition P is determined on the basis of an *interpretation* of its atoms, i.e. an assignment of either **t** ("true") or **f** ("false") to each atom that occurs in P. For instance, under the interpretation $\{$`?foo` $\mapsto \mathbf{t}\}$, the proposition `(not ?foo)` is false; under the interpretation $\{$`?foo` $\mapsto \mathbf{t}$, `?x` $\mapsto \mathbf{f}\}$, the proposition `(or ?foo ?x)` is true; under *any* interpretation of `?x` and `?y`, the proposition `(iff (and ?x ?y) (and ?y ?x))` is true; and so on.

There are also higher-order values, namely *function-values* and *method-values*.[16] A function-value, as its name suggests, is a function that maps a sequence of values (possibly including function- and/or

[15] Since we do not wish to delve into any metaphysical debates concerning the ontological status of propositions, we will simply identify them with the elements of the syntactic domain **Prop**. That is, we will take the "meaning" of a proposition $P \in$ **Prop** to be P itself. This is not unlike identifying natural numbers with numerals. The latter, though uncommon in practice, would be perfectly legitimate as long as we were only interested in isomorphically invariant numeric properties, i.e. properties that would not be contingent on any particular numeral representation (an example of a property that would be so contigent is "having an even number of digits"). This is rarely done, of course, because the natural numbers are well-behaved as abstract entities (in particular, they have clear identity conditions), and thus it is commonplace to posit the existence of a set of *abstract* numbers. Accordingly, in denotational semantics of programming languages there is usually a meaning function $\mathcal{M}$ from the set of all numerals to the set of natural numbers that maps a numeral such as `23` to the abstract entity that is the number twenty three. But the metaphysics of propositions are nowhere as clear. Exactly what abstract meanings should be attached to propositional sentences, if any, is a notoriously controversial topic in philosophical logic [14, 36]. Accordingly, we will adopt a "nominalist" standpoint and take the corresponding meaning function on **Prop** to be the identity mapping.

[16] We use the somewhat awkward terms "function-value" and "method-value" instead of just "function" and "method" because the latter are used at the syntax level, in reference to expressions of the form `(function (`$I_1 \cdots I_n$`)` E`)` and `(method (`$I_1 \cdots I_n$`)` D`)`, respectively.

method-values) either to a value (possibly a function- or a method-value), or to a distinguished error element, or to a distinguished "bottom" element that indicates non-termination. A method-value is also a function, but in addition to a sequence of values it also takes an assumption base as an extra argument; the value of the function is either a proposition, or again error or non-termination. Thus, unlike functions, methods cannot return higher-order values as results, although they can take such values as arguments.

We will now give an informal account of the semantics of the language by explaining how to obtain the value of a given phrase. Our description will amount to a high-level specification of an interpreter for Athena. The evaluation of a phrase F always takes place relative to a given *environment* ρ and a given *assumption base* β. An environment ρ is a computable mapping that maps each identifier I either to a value (and in that case we say that I is *bound* in ρ), or to a special "unbound" token (and we then say that I is *unbound* in ρ). The "domain" of ρ, denoted $\mathrm{Dom}(\rho)$, is the set of all identifiers that are bound in ρ. Given ρ, n distinct identifiers $I_1, \ldots, I_n$, and n values $V_1, \ldots, V_n$, we write $\rho \cdot [I_1 \mapsto V_1, \ldots, I_n \mapsto V_n]$ to denote the environment which maps I_j to $V_j, 1 \leq j \leq n$, and every other identifier I to $\rho(I)$. Finally, an assumption base, as we have already mentioned, is simply a recursive subset of **Prop** that contains at least the proposition `true`.

We consider the evaluation of expressions first. Given E, ρ, and β, the evaluation algorithm is driven by the structure of E:

- If E **is an identifier** I, then look up I in ρ. If I is unbound in ρ, raise an error; otherwise return the value $\rho(I)$.

  There are some pre-defined identifiers that are bound to specific values in any given environment, and cannot be given new values. These are the elements of *TopName*. For example, `false` is always bound to the proposition `false`; `and` is always bound to the function-value that takes a sequence of two propositions P_1 and P_2 and returns the conjunction `(and` P_1 P_2`)`; `mp` ("modus ponens") is always bound to a method-value that takes a sequence of two propositions P and P' and an assumption base β' and returns P_2 whenever (i) P and P' are of the form `(if` P_1 P_2`)` and P_1, respectively, and (ii) P and P' are both in β' (if either of these two conditions does not hold or if the input sequence of values contains anything other than two propositions P, P', then `mp` will generate an error); and so forth. An exhaustive description of the values associated with the identifiers in *TopName* can be found in Tables 1 and 2.

- If E **is a propositional variable** `?`I, then return the atomic proposition `?`I.

- If E **is a function** `(function (`$I_1 \cdots I_n$`)` E_b`)`, then return the unique function-value f for which $f(V_1, \ldots, V_n)$, for any given sequence of values $V_1, \ldots, V_n$, is that value which one obtains by evaluating E_b in $\rho \cdot [I_1 \mapsto V_1, \ldots, I_n \mapsto V_n]$ and β. Thus functions are statically scoped both with respect to environments and with respect to assumption bases.

- If E **is a method** `(method (`$I_1 \cdots I_n$`)` D`)`, then return the unique method-value that maps any given sequence of values $V_1, \ldots, V_n$ and assumption base β' to the value obtained by evaluating D in $\rho \cdot [I_1 \mapsto V_1, \ldots, I_n \mapsto V_n]$ and β'. Thus methods are statically scoped with respect to environments, but dynamically scoped with respect to assumption bases. The reason for this will become clear in the sequel.

- If E **is of the form** `(clet ((`I_1 F_1`)` $\cdots$ `(`I_n F_n`))` E_b`)`, then begin by evaluating F_1 in ρ and β, to obtain a value V_1. Then evaluate F_2 in $\rho \cdot [I_1 \mapsto V_1]$ and β to obtain a value V_2. Continue in this fashion until F_n is evaluated in $\rho \cdot [I_1 \mapsto V_1, \ldots, I_{n-1} \mapsto V_{n-1}]$ and β, resulting in a value V_n. If the evaluation of any F_i causes an error or diverges, then the evaluation of the entire

Name	Value
`true`	The proposition `true`.
`false`	The proposition `false`.
`not`	A function-value that takes a proposition P and returns the proposition `(not` P`)`.
`and`	A function-value that takes a sequence of two propositions P_1 and P_2 and returns the proposition `(and` P_1 P_2`)`.
`or`	A function-value that takes a sequence of two propositions P_1 and P_2 and returns the proposition `(or` P_1 P_2`)`.
`if`	A function-value that takes a sequence of two propositions P_1 and P_2 and returns the proposition `(if` P_1 P_2`)`.
`iff`	A function-value that takes a sequence of two propositions P_1 and P_2 and returns the proposition `(iff` P_1 P_2`)`.

Table 1: The values attached to the pre-defined identifiers in *LogOp*.

`clet` expression should produce the same error or diverge, respectively. If $V_1, \ldots, V_n$ have been successfully obtained, return the result of evaluating E_b in $\rho \cdot [I_1 \mapsto V_1, \ldots, I_n \mapsto V_n]$ and β.

- If E **is of the form** `(cletrec ((`I_1 E_1`)` $\cdots$ `(`I_n E_n`))` E_b`)`, then first evaluate $E_1, \ldots, E_n$, in that order, in $\rho' = \rho \cdot [I_1 \mapsto V_1, \ldots, I_n \mapsto V_n]$ and β, where V_i is the result of evaluating E_i in ρ' and β. Typically $E_1, \ldots, E_n$ will be functions or methods, but this is not strictly necessary, so if the evaluation of any E_i causes an error or diverges then the evaluation of the entire `cletrec` must also cause an error or diverge, respectively. If a value V_i is successfully obtained from the evaluation of each E_i, then return the result of evaluating E_b in ρ' and β.

- If E **is of the form** `(cmatch` F `(`Q_1 E_1`)` $\cdots$ `(`Q_n E_n`))`, then first evaluate F in ρ and β. If the result is not some proposition P, signal an error. Otherwise let Q_i be the first pattern amongst $Q_1, \ldots, Q_n$ that *matches* P under ρ and some propositional substitution θ (if no Q_i matches P raise an error), and return the result of evaluating E_i in the environment ρ augmented with the bindings of θ and in the assumption base β. (We will discuss the details of pattern matching later on; for now an intuitive understanding will suffice.)

Below we have printed the transcript of a simple interactive session with an Athena interpeter. A sequence of inputs is presented to the interpreter for evaluation at the prompt `>`. The interpreter evaluates each input in the current environment and assumption base and prints the result in the next line. An input is either an expression E or a definition `(define` I E`)` (for now we focus exclusively on expressions). In the latter case the interpreter evaluates E to obtain a value V and then augments the current environment with the binding $I \mapsto V$, so that subsequent inputs are evaluated in an environment in which I is bound to V. Thus the evaluation of an input expression E which has been preceded by n definitions `(define` I_1 E_1`)`,…,`(define` I_n E_n`)` is equivalent to the evaluation of the expression

`(clet ((`I_1 E_1`)` $\cdots$ `(`I_n E_n`))` E`)`

in the empty environment. The assumption base can be ignored in the following examples since for now we are not interested in evaluating deductions. Finally, note that the interpreter indicates the *type* of each result immediately after the colon `:`, and that some types contain *type variables* (e.g. the type of `id` below is `T -> T`, where `T` is a type variable).

```
>?foo
?foo:prop

>(define a ?foo)
a:prop

>(if a (or ?x ?foo))
(if ?foo (or ?x ?foo)):prop

>(define negate (function (p) (not p)))
negate:prop -> prop

>(negate true)
(not true):prop

>(negate (not a))
(not (not ?foo)):prop

>(negate (negate (or ?x a)))
(not (not (or ?x ?foo))):prop

>(clet ((p (and ?x (not false))))
   (negate p))
(not (and ?x (not false))):prop

>(define id (function (x) x))
id:T -> T

>(clet ((p (or ?x ?a)))
   ((id negate) (id p)))
(not (or ?x ?a)):prop

>(define t true)
t:prop

>(define g (function (p) (and t p)))
g:prop -> prop

>(g ?t)
(and true ?t):prop

>(define t false)
t:prop

>(g a)
```

```
(and true ?foo):prop

>(clet ((new-or or))
   ((id new-or) true false))
(or true false):prop

>(define reflect
   (cletrec
     ((reflect (function (prop)
                 (cmatch prop
                   ((not p) (not (reflect p)))
                   ((and p1 p2) (and (reflect p2) (reflect p1)))
                   ((or p1 p2) (or (reflect p2) (reflect p1)))
                   ((if p1 p2) (if (reflect p2) (reflect p1)))
                   ((iff p1 p2) (iff (reflect p2) (reflect p1)))
                   (p p)))))
        reflect))
reflect:prop -> prop

>(reflect true)
true:prop

>(reflect a)
?foo:prop

>(reflect (and ?x ?y))
(and ?y ?x):prop

>(reflect (if (and false ?x)
              (or ?foo true))))
(if (or true ?foo)
    (and ?x false)):prop

>(reflect (reflect (or ?x ?y)))
(or ?x ?y):prop
```

Next we describe the semantics of deductions by explaining how to evaluate an arbitrary $D \in \mathbf{Ded}$ in a given environment ρ and assumption base β. As before, our actions are dictated by the structure of D:

- If D **is a hypothetical deduction** (`assume` F D_b) then first evaluate F in ρ and β. If the result is *not* a proposition P, raise an error; otherwise, evaluate the deduction D_b in ρ and $\beta \cup \{P\}$. If that results in a proposition P', return the conditional (`if` P P'), otherwise raise an error.

- If D **is a deduction-by-contradiction** (`suppose-absurd` F D_b), then first evaluate F in ρ and β. If the result is *not* a proposition P, raise an error; otherwise, evaluate the deduction D_b in ρ and $\beta \cup \{P\}$. If that results in the proposition `false`, return the negation (`not` P), otherwise raise an error.

Name	Value
`claim` (Reiteration)	A method-value that takes a proposition P and an assumption base β and returns P if β contains P; in all other cases it generates an error.
`mp` (Modus Ponens)	A method-value that takes a sequence of two propositions P and P' of the form `(if` P_1 P_2`)` and P_1, respectively, and an assumption base β, and returns P_2 whenever both P and P' are in β; in all other cases it generates an error.
`dn` (Double negation)	A method-value that takes a proposition P of the form `(not (not` P'`))` and an assumption base β and returns P', provided P is in β; in all other cases it generates an error.
`absurd` (`false` introduction)	A method-value that takes a proposition P of the form `(and` P_1 `(not` P_1`))` and an assumption base β and returns `false`, provided that P is in β; in all other cases it generates an error.
`cd` (Constructive dilemma)	A method-value that takes a sequence of three propositions P, P', P'' of the form `(or` P_1 P_2`)`, `(if` P_1 P_3`)`, and `(if` P_2 P_3`)` respectively, and an assumption base β, and returns P_3 whenever all three of P, P', and P'' are in β; in all other cases it generates an error.
`both` (`and` introduction)	A method-value that takes a sequence of two propositions P_1 and P_2 and an assumption base β and returns the conjunction `(and` P_1 P_2`)`, provided both P_1 and P_2 are in β; in all other cases it generates an error.
`left-and` (`and` elimination)	A method-value that takes a proposition P of the form `(and` P_1 P_2`)` and an assumption base β and returns P_1, provided that P is in β; in all other cases it generates an error.
`right-and` (`and` elimination)	A method-value that takes a proposition P of the form `(and` P_1 P_2`)` and an assumption base β and returns P_2, provided that P is in β; in all other cases it generates an error.
`either` (`or` introduction)	A method-value that takes a sequence of two propositions P_1 and P_2 and an assumption base β and returns the disjunction `(or` P_1 P_2`)`, provided that at least one of P_1, P_2 is in β; in all other cases it generates an error.
`equiv` (`iff` introduction)	A method-value that takes a sequence of two propositions P and P' of the form `(if` P_1 P_2`)` and `(if` P_2 P_1`)`, respectively, and an assumption base β , and returns the biconditional `(iff` P_1 P_2`)` whenever both P_1 and P_2 are in β; in all other cases it generates an error.
`left-iff` (`iff` elimination)	A method-value that takes a proposition P of the form `(iff` P_1 P_2`)` and an assumption base β and returns `(if` P_1 P_2`)`, provided that P is in β; in all other cases it generates an error.
`right-iff` (`iff` elimination)	A method-value that takes a proposition P of the form `(iff` P_1 P_2`)` and an assumption base β and returns `(if` P_2 P_1`)`, provided that P is in β; in all other cases it generates an error.

Table 2: The values attached to the pre-defined identifiers in *PrimRule*.

- If D **is a method application** `(apply-method` E F_1 $\cdots$ F_n`)`, then begin by evaluating E in ρ and β. If the result is not a method-value m, raise an error. Otherwise evaluate each F_i in ρ and β to obtain a value V_i, with the understanding that if the evaluation of any F_i produces an error or diverges then the evaluation of the entire application must halt with the same error or diverge, respectively. Moreover, if F_i is a deduction then the result V_i must be a proposition, and should be inserted in a set called *ValidArgs*, which is initially empty. After every F_i has been successfully evaluated, return the result of applying the method-value m to the sequence $V_1, \ldots, V_n$ and the assumption base $\beta \cup \mathit{ValidArgs}$.

  Observe that the assumption base which is dynamically passed to the method-value m is the current assumption base β augmented with whatever argument values $V_i \in \mathbf{Prop}$ have been deductively derived. This means that method application is not merely a mechanism for altering the flow of control, as is function application, but it is also a mechanism for establishing intermediate conclusions ("lemmas"). The only other similar mechanism in the Athena kernel is the `dlet` construct, which we discuss below. This is no accident, for the semantics of `dlet`, as we will now explain, are identical to those of method applications.[17] The next two items discuss `dlet`s with only one binding, of the form `(`I E`)` and `(`I D`)`, respectively. The case of multiple bindings will then be reduced inductively to these two base cases.

- If D **is a deduction of the form** `(dlet ((`I E`))` D_b`)`, then first evaluate E in ρ and β to obtain a value V (if an error occurs instead, halt with that error). Then evaluate D_b in $\rho \cdot [I \mapsto V]$ and β and return the result.

  Thus `dlet`s allow for sequential evaluation.[18] In this case an expression E is evaluated first, presumably resulting in some value that will be of use inside D_b under the name I, and then the deduction D_b is evaluated. More interesting is the next case in which a deduction takes the place of E, resulting in the chained evaluation of two deductions. The general case, in which the `dlet` contains multiple bindings of identifiers to deductions, gives rise to a sequence of deduction evaluations that take place along a thread of monotonically increasing assumption bases.

- If D **is a deduction of the form** `(dlet ((`I D_1`))` D_2`)`, then first evaluate D_1 in ρ and β to get a proposition P (if an error occurs during the evaluation of D_1, halt in error). Then evaluate D_2 in $\rho \cdot [I \mapsto P]$ and $\beta \cup \{P\}$, returning that result as the final answer.

  The important point here is that P becomes logically available to the main deduction D_2, playing the role of a lemma. As we discussed above, the same effect can be achieved via argument evaluation in method applications—these are the only two mechanisms in the Athena kernel for establishing intermediate conclusions—but `dlet`s should be preferred because their syntactic form clarifies the exact logical scope of the lemma (as well as the lexical scope of the lemma's name).

  Finally, when a `dlet` contains more than one binding we proceed inductively:

- If D **is a deduction of the form** `(dlet ((`I_1 F_1`)` $\cdots$ `(`I_n F_n`))` D_b`)`, then return the result of evaluating `(dlet ((`I_1 F_1`)) (dlet ((`I_2 F_2`)` $\cdots$ `(`I_n F_n`))` D_b`))` in ρ and β. Thus

[17] In particular, a deduction `(dlet ((`I F`))` D`)` is equivalent to `(apply-method (method (`I`)` D`)` F`)`. This is similar to the usual desugaring of let expressions into functional applications, but with the added significance afforded by the role of the assumption base: if F is a deduction, then the proposition resulting from its evaluation will be incorporated into the assumption base in which the body D will be evaluated.

[18] We stress that `dlet`, as well as `clet`, is not merely a naming mechanism. In both cases the named phrases must be fully evaluated before the evaluation of the body can commence. In particular, if the evaluation of any named phrase diverges then the evaluation of the entire let phrase diverges, even if the name does not appear inside the body. This is a reflection of the call-by-value semantics of Athena.

in dlets with multiple bindings $(I_1\ F_1)\ \cdots\ (I_n\ F_n)$, the phrases $F_1, \ldots, F_n$ are evaluated sequentially, as in clets, with the value of F_j available to $F_{j+1}, \ldots, F_n$ and D_b under the name I_j.

The remaining two cases do not introduce any new ideas.

- If D **is of the form** (dletrec $((I_1\ E_1)\ \cdots\ (I_n\ E_n))\ D_b$), then, as in clet, begin by evaluating $E_1, \ldots, E_n$ in $\rho' = \rho \cdot [I_1 \mapsto V_1, \ldots, I_n \mapsto V_n]$ and β, where V_i is the result of evaluating E_i in ρ' and β. An error generated during the evaluation of an E_i must cause the evaluation of the entire dlet to fail. If the evaluation of $E_1, \ldots, E_n$ successfully produces values $V_1, \ldots, V_n$, then return the result of evaluating D_b in ρ' and β.
- If D **is of the form** (dmatch F $(Q_1\ D_1)\ \cdots\ (Q_n\ D_n)$), then first evaluate F in ρ and β. If the result is not some proposition P, generate an error. Otherwise let Q_i be the first pattern amongst $Q_1, \ldots, Q_n$ that matches P under ρ and some propositional substitution θ (if no pattern matches P raise an error), and return the result of evaluating D_i in the environment ρ augmented with the bindings of θ and in the assumption base β.

We will now present some intuition-building examples that illustrate the evaluation algorithm in practice. But first we introduce two syntactic conventions that we will use throughout:

1. We will abbreviate method applications (apply-method E $F_1\ \cdots\ F_n$) as (\$$E$ E $F_1\ \cdots\ F_n$).
2. We will write F! as an abbreviation for (\$claim F).

All of the phrases below are evaluated in the empty assumption base, unless we explicitly say otherwise. We also start out with an empty environment.

```
>($claim true)
Theorem:true

>true!
Theorem:true
```

The constant true may be freely claimed in all assumption bases.

```
>($claim ?foo)
Error: ?foo is not in the assumption base.

>false!
Error: false is not in the assumption base.

>(assume false
   false!)
Theorem:(if false false)
```

In the above example the claim `false!` is evaluated in an assumption base that contains the proposition `false`, hence it goes through fine. The resulting proposition is that `false` implies `false`: `(if false false)`.

```
>(define id-method
   (method (P)
     (assume P
       P!)))
id-method:prop ->> prop
```

Here we abstract the preceding deduction into a general method that can be applied to any proposition P to produce the tautology `(if` P P`)`.

```
>($id-method (and ?foo true))
Theorem:(if (and ?foo true) (and ?foo true))

>(dlet ((negate (function (p) (not p))))
   ($id-method (negate true)))
Theorem:(if (not true) (not true))
```

The next deduction is rather uninteresting, but helps to illustrate some basic points:

```
>(dlet ((not-not-true (suppose-absurd (not true)
                   ($absurd ($both true (not true))))))
   ($dn not-not-true))
Theorem:true
```

Here we begin with an auxiliary deduction that derives the proposition `(not (not true))` and binds the identifier `not-not-true` to it. This auxiliary deduction is made using the `suppose-absurd` form, which embodies the principle of "proof-by-contradiction" (*reductio ad absurdum*). We begin by making the supposition `(not true)`, as a hypothesis to be proven contradictory within the body of the `suppose-absurd`. The hypothesis `(not true)` is then inserted in the current assumption base and we proceed to evaluate the body of the `suppose-absurd`. The contradiction is easily derived there by applying `both` to `true` (which is in every assumption base) and `(not true)`, which is in the current assumption base. Now `both` is a primitive method that takes two propositions P_1, P_2, and if both of these are in the current assumption base (as is the case in this example), their conjunction `(and` P_1 P_2`)` is returned. This is then passed to `absurd`, which is another primitive method that takes a proposition of the form `(and` P `(not` P`))` as input (any other input is incorrect), and if this contradictory proposition is in the current assumption base then `absurd` returns `false`. This is precisely what `suppose-absurd` needs in order to yield the negation of its hypothesis, so in this case the entire `suppose-absurd` returns the proposition `(not (not true))`. This proposition then becomes the value of the identifier `not-not-true` and we jump into the body of the `dlet`. Further, because `(not (not true))` was deductively obtained, we incorporate it in our assumption base before we start evaluating the `dlet` body. The latter is simply the application of the "double negation" primitive rule `dn` to `not-not-true`. The `dn` rule requires one proposition as input, and if this proposition is in the current assumption base and has the form `(not (not` P`))`, then `dn` returns P. In this case `dn` is applied to `(not (not true))`, which is in the current assumption base, so the result of the entire `dlet` is simply the proposition `true`.

The following is a deduction of

```
(if (if ?A ?B)
    (if (not ?B) (not ?A)))
```

```
>(assume (if ?A ?B)
   (assume (not ?B)
     (suppose-absurd ?A
       ($absurd ($both ($mp (if ?A ?B) ?A)
                       (not ?B))))))
Theorem: (if (?A ?B)
             (if (?not B) (not ?A)))
```

We abstract this deduction into a method that takes a theorem of the form `(if` P_1 P_2`)` and logically derives the theorem `(if (not` P_2`) (not` P_1`))`, called the *contrapositive* of `(if` P_1 P_2`)`:

```
1. >(define contra-positive
2     (method (P)
3.      (dmatch P
4.        ((if p1 p2) (assume (not p2)
5.                      (suppose-absurd p1
6.                        ($absurd ($both ($mp P p1)
7.                                        (not p2)))))))))
   contra-positive:prop ->> prop
```

A couple of things should be noted here. First, the match expression contains only one clause, whose pattern is `(if p1 p2)`. This automatically means that if the discriminant, which in this case is the input proposition `P`, is not a conditional, then the match expression will fail and the entire application of the `contra-positive` method will generate an error. This is appropriate since, by definition, the contrapositive rule applies only to conditionals. So just as the application of the double negation rule to, say a conjunction, generates an error, the application of contrapositive to anything but a conditional should likewise generate an error. Secondly, notice that the type of `contra-positive` is written with a double-headed arrow, as `prop ->> prop`, rather than `prop -> prop`. This simply serves to indicate that `contra-positive` is a method rather than a function.

Let us now try to apply `contra-positive` to a conditional:

```
>($contra-positive (if ?x ?y))
Error: (if ?x ?y) is not in the assumption base.
```

It is not difficult to see what the problem is here if we only keep in mind that what we just attempted was to *deduce* `(if (not ?y) (not ?x))`, not merely to *compute* it. But this proposition does not in fact follow logically from the contents of the assumption base, for remember that we are currently working in the empty assumption base. Since Athena is logically sound (we will prove this later), it will never derive a conclusion that is not logically warranted, so we should have actually expected a problem. To be precise, the point at which the error occurs is when control is in line `6` of the body of `contra-positive`, in the application of `mp` to `P` and `p1`. The primitive rule `mp`, just like every other primitive rule in Athena except `either`, requires all of its arguments to be in the current assumption base in order to produce a result. This is the essence of how an inference rule is used in a deduction: when certain propositions *have been derived*, a certain other proposition is derived from them. In our case "have been derived" means "are in the assumption base". We emphasize that a primitive rule like `mp` does not merely require its arguments to follow logically from the contents of the assumption base; it requires them to *be in* the assumption base. For instance, suppose that our assumption base contains only two propositions, say `(and ?x ?y)` and `(if ?y ?foo)`. If we attempt to evaluate the method application `($mp (if ?y ?foo) ?y)` in this assumption base, we will get an error because the second argument `?y` is not in the assumption base, despite the fact that it follows from it. The fact that `?y` is not in the assumption base means that it has not been established, and for this reason `mp` acts skeptically. We might imagine `mp` telling us: "Well, I can see that `(if ?y ?foo)` holds, but you have yet to convince me about `?y`. If you can show me that `?y` holds as well, I will give you `?foo`.". What we should do in this case is *first* establish `?y` (which is really easy: we simply apply `right-and` to `(and ?x ?y)`), and *then* invoke `mp`. This is where deduction sequencing via `dlets` and method applications enters the picture. The following deduction does the trick:

```
>(dlet ((lemma ($right-and (and ?x ?y))))
   ($mp (if ?y ?foo) lemma))
Theorem:?foo
```

The same effect could be achieved via method application:

```
>($mp (if ?y ?foo) ($right-and (and ?x ?y)))
Theorem:?foo
```

since, as we explained in our discussion of method applications, if an argument to a method application is a deduction then the proposition obtained by evaluating that argument is inserted in the assumption base in which the body of the method is evaluated. A consequence of this is that `(dlet ((`I D_1`))` D_2`)` and `($(method (`I`)` D_2`)` D_1`)` have exactly the same meaning.

The situation we are discussing is no different than in other deductive systems. For instance, in graphical linear systems of the Kalish-Montague sort the erroneous evaluation of `($mp (if ?y ?foo) ?y)` in the assumption base {`(and ?x ?y)`, `(if ?y ?foo)`} is analogous to the following illegal deduction:

1.	$x \wedge y$	
2.	$y \Rightarrow foo$	
3.	foo	Modus Ponens, 2, y.

This is not a legal deduction because Modus Ponens is applied to y, which has not been previously derived. Indeed, this does not even "parse" in the Kalish-Montague world, i.e. it is not even well-formed, because *by definition* in such systems every single line in a legal deduction is of the form

Line number i. $\quad P \quad$ Justification: IR, $l_1, \ldots, l_k$

where P is the conclusion of this line, and IR is the name of some inference rule which is being applied to the propositions derived in the lines numbered $l_1, \ldots, l_k$, where the numbers $l_1, \ldots, l_k$ are all less than i. Thus the above is not well-formed because in the justification column of line 3, y appears in a place where a line number is expected. The correct way to express the desired inference in such a formalism would be

1.	$x \wedge y$	
2.	$y \Rightarrow foo$	
3.	y	And-simplification, 1.
4.	foo	Modus Ponens, 2, 3.

In summary, the user generally needs to establish the arguments to an inference rule as lemmas, with auxiliary deductions, and then apply the rule to the generated lemmas. In Athena such sequencing is done by using `dlets`, or method applications, or `dbegin` expressions (the latter will be introduced shortly as syntactic sugar). Semantically, the end result of such a deduction is a monotonic sequence of assumption bases

$$\beta_1 \subseteq \beta_2 \subseteq \cdots \subseteq \beta_n \subseteq \beta_{n+1}$$

where β_1 is the initial assumption base and β_{i+1} is β_i augmented with the result of applying a primitive method to some elements of β_i.

Returning to our original example, we can now see that before `contra-positive` can be applied to it, the proposition `(if ?x ?y)` must be derived. This is in fact impossible to do in the empty assumption base since `(if ?x ?y)` is not logically valid, but there are many assumption bases in which `(if ?x ?y)` could be derived, and then `contra-positive` could be successfully applied to it. Of course it is always possible to *compute* the contrapositive of a given conditional, regardless of what is in the assumption base:

```
>(define cp (function (P)
              (cmatch P
                ((if p1 p2) (if (not p2) (not p1))))))
cp:prop -> prop

>(cp (if ?x ?y))
(if (not ?y) (not ?x)):prop

>(cp (cp (if true false)))
(if (not (not true))
    (not (not false))):prop

>(cp (and true false))
Error: cmatch error---no match was found for (and true false).
```

However, not all methods require their input propositions to be facts (members of the assumption base at application time). Some methods can generate valid results regardless of whether or not their input propositions are in the assumption base. These are methods, then, whose results are independent of the contents of any particular assumption base because they are always valid, i.e. because they hold "in all possible worlds" (if we think of a given assumption base as "a world"). In other words, these are methods that produce tautologies. A trivial example is `(method (P) ($claim true))`. Another is `id-method`, which we defined earlier as `(method (P) (assume P ($claim P)))`. No matter what proposition P we supply to `id-method` as an argument, we will always get the tautology `(if` P P`)` as a result, which holds in all possible assumption bases. Another example is

```
>(define left-and-axiom
   (method (P1 P2)
     (assume (and P1 P2)
       ($left-and (and P1 P2)))))
```

which, applied to any propositions P_1, P_2 and any assumption base β, returns the tautology

$$(P_1 \wedge P_2) \Rightarrow P_1$$

which holds regarless of the contents of β. Additional examples of such methods can be found in the proof of the completeness theorem in the metatheory section; the readers will have no trouble constructing examples of their own.

We might say, then, that methods such as `id-method` or `left-and-axiom` are **logically independent**[19] of their arguments, while most other methods, such as `mp`, or `both`, or `contra-positive`, etc., are logically *dependent*, or **contingent**[20] on their arguments. Methods that are logically independent of their arguments can never produce a *contingent* result (a proposition is called "contingent" iff it is neither a tautology nor a contradiction, i.e. iff it is true under some interpretations and false under others), whereas a method that is contingent on (some of) its arguments might very well produce contingent results, which will hold owing to the contents of the particular assumption base in which the method was applied. For instance, the application of `mp` to `(if ?x ?y)` and `?x` in an assumption base that contains both of these propositions will yield the contingent proposition `?y`, which will hold in virtue of the fact that this particular assumption base ("world") contains `(if ?x ?y)` and `?x`. Perhaps *the* paradigmatic example of a contingent method is `claim`, which is obviously highly dependent on its argument. Another way to look at the distinction is to think of contingent methods such as `mp` as **truth preservers** and of non-contingent methods such as `id-method` as **truth generators**.

An *operational* characterization of methods that are logically independent of their arguments is this: in any chain of deductions spawned by an application of such a method m to propositions $P_1, \ldots, P_n$, no P_i is ever passed as an argument to a primitive method unless the assumption base at that particular point already contains P_i.[21] This means either that P_i is never logically used, or if it is, it is used in a safe-guarded context: *first* it is conditionally postulated, i.e. it is explicitly inserted in the assumption base by an

$$\texttt{(assume } P_i \ D\texttt{)}$$

deduction, and *then* it is used by a primitive method at some point during the evaluation of D. But nowhere in the entire call tree will P_i appear as an argument in a primitive method application whose outcome hinges on whether P_i was *in the particular assumption base in which the initial call* $\texttt{(\$}m \cdots P_1 \cdots P_n \cdots\texttt{)}$ *was made*.

Of course it is the user's responsibility to know whether a method is contingent on a certain argument and to ensure that it is always applied accordingly so as to avoid a run-time error, just as it is the programmer's responsibility to know whether a procedure might use a certain argument as a divisor and to ensure that such a procedure is never applied in a context in which that argument is zero.

In the remainder of this paper we will use a top-level directive `assert` that takes any number of propositions $P_1 \cdots P_n$ and mutates the current assumption base by adding $P_1, \ldots, P_n$ to it. This is not a part of the Athena language *per se*, but like `define`, it is a useful tool to have at the read-eval-print loop for experimentation, and in our case, for presenting illustrational examples.[22] We assume we begin with the empty assumption base (and the environment containing all the bindings of the preceding examples).

```
>($claim ?foo)
```

[19] The qualification provided by the adverb "logically" is necessary because a method might be *computationally* dependent on ("strict in") a certain argument without being logically dependent on it.

[20] Logical dependency of a method on its arguments might also be called "logical strictness" in analogy with the corresponding computational notion.

[21] An exception is applications of `either` to a pair of propositions one of which is a tautology, but this may be overlooked for our purposes.

[22] Needless to say, `assert` can easily engender an inconsistent assumption base—simply `(assert false)`. This is immaterial for our purposes, however, since we are only interested in explaining the evaluation of deductions, which has nothing to do with whether or not the assumption base is consistent.

```
Error: ?foo is not in the assumption base.

>(assert ?foo)
The proposition ?foo has been added to the assumption base.

>($claim ?foo)
Theorem:?foo

>(not ?foo)!
Error: (not ?foo) is not in the assumption base.
```

With the `contra-positive` method under our belt we can easily write a method that implements *Modus Tollens* (an inference rule that says "From $P_1 \Rightarrow P_2$ and $\neg P_2$ infer $\neg P_1$". We write

```
>(define modus-tollens
   (method (P1-gives-P2 not-P2)
     ($mp ($contra-positive P1-gives-P2) not-P2)))
modus-tollens:prop * prop ->> prop

>(assert (if ?A ?B)
         (not ?B))
The propositions (if ?A ?B), (not ?B) have been added to the assumption base.

>($modus-tollens (if ?A ?B) (not ?B))
Theorem:(not ?A)
```

We could have implemented Modus Tollens in a more verbose way by writing

```
>(define mt
   (method (P1 P2)
     (dmatch P1
       ((if x y)
          (dmatch P2
            ((not (val-of y)) ($mp ($contra-positive P1) P2)))))))
```

but the two versions are in fact extensionally identical, since all the matching performed by `mt` will be performed by `contra-positive` and `mp`. Another way to see that the matching is not really necessary is to observe that the pattern variables `x` and `y` are never used in the *result* part of a `dmatch` clause `(`*pattern result*`)`. Thus `mt` is somewhat less efficient than `modus-tollens`, although it is perhaps a little more self-documenting.

Next we observe that `modus-tollens` is not commutative:

```
>($modus-tollens (not ?B) (if ?A ?B))
Error: dmatch error---no match was found for (not ?B).
```

`modus-tollens` passes its first argument to `contra-positive`, which blows up on anything other than a conditional (recall from the body of `contra-positive` that its input proposition is matched against only one pattern: `(if p1 p2)`). All the primitive Athena rules are also non-commutative in that sense; e.g.

```
>(assert ?x (if ?x ?y))
The propositions ?x, (if ?x ?y) have been added to the assumption base.

>($mp (if ?x ?y) ?x)
Theorem:?y

>($mp ?x (if ?x ?y))
Error: incorrect application of mp.
```

But usually it is an easy task to write a commutative version of a given method, primitive or not, if we wish to do so. E.g. for Modus Tollens we could write

```
>(define commutative-modus-tollens
   (method (P1 P2)
     (dmatch P1
       ((if x y) (dmatch P2
                   ((not (val-of y)) ($modus-tollens P1 P2))))
       ((not y) (dmatch p2
                  ((if x (val-of y)) ($modus-tollens P2 P1)))))))
commutative-modus-tollens:prop * prop ->> prop

>($commutative-modus-tollens (if ?A ?B) (not ?B))
Theorem:(not ?A)

>($commutative-modus-tollens (not ?B) (if ?A ?B))
Theorem:(not ?A)
```

4.3 Syntax sugar

We will now extend the language with a number of new syntactic forms that simplify the task of writing deductions. Each new form will be introduced as a "macro", with a definition of the form $C_{\text{new}} \Longrightarrow F_{\text{kernel}}$, where C_{new} is the new construct and F_{kernel} is the corresponding kernel phrase which C_{new} will abbreviate. Some of these new constructs have already been introduced and used in the previous section. We stress that all of the following are simple syntactic abbreviations that do not interfere with the type discipline of the language.[23] Users cannot introduce their own abbreviations—the language does *not* offer a macro facility.

- `?I` $\Longrightarrow$ `(var I)`

  A convenient shortcut for writing variables.

- `($`$E\ F_1 \cdots F_n$`)` $\Longrightarrow$ `(apply-method` $E\ F_1 \cdots F_n$`)`

  An abbreviation for method application. Thus function and method applications are of the form `(`$E\ F_1 \cdots F_n$`)` and `($`$E\ F_1 \cdots F_n$`)`, respectively.

- F`!` $\Longrightarrow$ `($claim` F`)`

  An abbreviation for claims.

[23]For the purposes of a type-inference algorithm, each of the few constructs we introduce as sugar should be typed with kernel status, to avoid perplexing error messages.

- `_` $\Longrightarrow$ `any`

  An abbreviation for the "match-all" pattern `any`.

- `(assume*` $(F_1 \cdots F_n)$ D`)` $\Longrightarrow$ `(assume` F_1 `(assume` F_2 $\cdots$ `(assume` F_n D`)` $\cdots$ `))`

  This makes it easier to write deductions that use multiple assumptions and introduce nested conditionals. E.g.

```
(assume* ((and ?x ?y)
          (if ?y ?z))
  ($mp (if ?y ?z)
       ($right-and (and ?x ?y))))
```

  yields the proposition

```
(if (and ?x ?y)
    (if (if ?y ?z) ?z))
```

- `(assume-let ((`I_1 F_1`)` $\cdots$ `(`I_n F_n`))` D`)` $\Longrightarrow$
  `(dlet ((`I_1 F_1`)` $\cdots$ `(`I_n F_n`)) (assume*` $(I_1 \cdots I_n)$ D`))`

  This provides a convenient way of referring to the propositions denoted by F_i within their logical scope D. Asides from saving typing, this is useful in giving assumptions appropriate names that reflect their structure and role. An example:

```
(assume-let ((A-gives-B (if ?A ?B))
             (B-gives-C (if ?B ?C))
             (A-holds ?A))
  ($mp B-gives-C
       ($mp A-gives-B A-holds)))
```

  produces

```
(if (if ?A ?B)
    (if (if ?B ?C)
        (if ?A ?C)))
```

- `(suppose-absurd-let (`I F`)` D`)` $\Longrightarrow$ `(dlet ((`I F`)) (suppose-absurd` I D`))`

  Like `assume-let`, this form allows us to name the hypothesis which we want to prove contradictory.

- `(cbegin` F_1 $\cdots$ F_n E`)` $\Longrightarrow$ `(clet ((`I F_1`)` $\cdots$ `(`I F_n`))` E`)`
  where I is any identifier that does not occur in any F_i or in E.

  This also provides for sequential evaluation, but in situations in which we do not care to name the intermediate results. The last phrase must be an expression, whose value becomes the value of the entire `cbegin` expression.

- `(dbegin` F_1 $\cdots$ F_n D`)` $\Longrightarrow$ `(dlet ((`I F_1`)` $\cdots$ `(`I F_n`))` D`)`
  where I is any identifier that does not occur in any F_i or in E.

  As above, only now the last component is a deduction, the value of which is the value of the entire `dbegin` form.

- `(`P `by` D`)` $\Longrightarrow$ `(dmatch` D `(`P D`))` This evaluates D and checks to see that the resulting proposition is P; if so, P is returned. An error occurs if either the evaluation of D produces an error, or if the result of evaluating D is not P. This construct enhances readability for those who are used to working with Kalish-Montague-like systems. Here is a deduction of $Q \wedge R$ from $P \wedge R$ and $P \Rightarrow Q$ that makes heavy use of this construct:

```
(assume-let ((P-gives-Q (if ?P ?Q))
             (P-and-R (and ?P ?R)))
  (dbegin (?P by ($left-and P-and-R))
          (?Q by ($mp P-gives-Q ?P))
          (?R by ($right-and P-and-R))
          ((and ?Q ?R) by ($both ?Q ?R))))
```

  An alternative desugaring of this construct is

  `(`P `by` D`)` $\Longrightarrow$ `(dlet ((`I D`)) (dmatch` I `(`P I`!)))`

  for any I that does not occur in D. This desugaring is longer, but it is slightly more efficient since D is only evaluated once.

4.4 Additional examples

In this section we will show how some well-known "derived inference rules" can be straightforwardly expressed as Athena methods. We begin by writing a method `dm1` which takes a theorem of the form $\neg(p_1 \vee p_2)$ and logically derives the conclusion $\neg p_1 \wedge \neg p_2$. Thus we can view `dm1` as implementing one of *DeMorgan's laws*.

```
>(define dm1
   (method (P)
     (dmatch P
       ((not (or p1 p2))
          (dlet ((not-p1 (suppose-absurd p1
                          ($absurd ($both ($either p1 p2) P))))
                 (not-p2 (suppose-absurd p2
                          ($absurd ($both ($either p1 p2) P)))))
            ($both not-p1 not-p2))))))
dm1:prop ->> prop

>(assert (not (or ?x ?y)))
The proposition (not (or ?x ?y)) has been added to the assumption base.

>($dm1 (not (or ?x ?y)))
Theorem:(and (not ?x) (not ?y))
```

Next we define a method that can apply this rule in the converse direction, i.e. one that can derive $\neg(p_1 \vee p_2)$ from $\neg p_1 \wedge \neg p_2$.

```
>(define dm1-dual
   (method (P)
```

```
    (dmatch P
      ((and (not p1) (not p2))
         (suppose-absurd-let (p1-or-p2 (or p1 p2))
           (dlet ((case1 (assume p1
                           ($absurd ($both p1 ($left-and P)))))
                               ;; Thus case1 is that p1 implies false
                  (case2 (assume p2
                           ($absurd ($both p2 ($right-and P))))))
                               ;; while case2 is that p2 implies false
             ($cd p1-or-p2 case1 case2)))))))
dm1-dual:prop ->> prop

>($dm1-dual (and (not ?x) (not ?y)))
Theorem:(not (or ?x ?y))
```

Notice that the Athena interpreter remembers that `(and (not ?x) (not ?y))` was derived as a theorem, so we do not have to assert it. That is, just as the evaluation of a definition causes all subsequent inputs to be evaluated in an environment that contains the relevant binding, the evaluation of a deduction causes all subsequent inputs to be evaluated in an assumption base that contains the deduced proposition.

Now we can write a method that takes a theorem of the form $p_1 \Rightarrow p_2$ and derives from it the conclusion $\neg p_1 \vee p_2$:

```
>(define imp1
   (method (P)
     (dmatch P
       ((if p1 p2)
         ($dn (suppose-absurd-let (x (not (or (not p1) p2)))
                (dlet ((nnp1-and-np2 ($dm1 x))
                       (p1-holds ($dn ($left-and nnp1-and-np2)))
                       (p2-holds ($mp P p1-holds))
                       (not-p2 ($right-and nnp1-and-np2)))
                  ($absurd ($both p2-holds not-p2)))))))))
>imp1:prop ->> prop

>(define true-gives-true (assume true true!))
Theorem:(if true true)

>($imp1 true-gives-true)
Theorem (or (not true) true)
```

Next we will write the dual of `imp1`, i.e. a method that derives $P_1 \Rightarrow P_2$ from $\neg P_1 \vee P_2$. To do this, we first write a method that derives $\neg\neg P$ from P (this can be seen as the dual of the primitive `dn`):

```
>(define negate-twice
   (method (P)
     (suppose-absurd (not P)
       ($absurd ($both P (not P))))))
```

```
negate-twice: prop ->> prop

>($negate-twice true)
Theorem:(not (not true))

>(define imp1-dual
   (method (P)
     (dmatch P
       ((or (not p1) p2)
          (assume p1
            (dlet ((nnp2 (suppose-absurd (not p2)
                           (dlet ((not-P ($dm1-dual ($both ($negate-twice p1)
                                                           (not p2)))))
                             ($absurd ($both P not-P))))))
              ($dn nnp2)))))))
imp1-dual: prop ->> prop

>($imp1-dual (or (not true) true))
Theorem:(if true true)
```

Now we can write a more versatile method `imp` that works in both directions depending on the form of the input proposition. If the latter is a conditional $p_1 \Rightarrow p_2$, `imp` will derive $\neg p_1 \vee p_2$; if it is a disjunction of the form $\neg p_1 \vee p_2$, `imp` will derive $p_1 \Rightarrow p_2$:

```
>(define imp
   (method (P)
     (dmatch P
       ((if p1 p2) ($imp1 P))
       ((or (not p1) p2) ($imp1-dual P)))))
imp:prop ->> prop

>($imp (if true true))
Theorem:(or (not true) true)

>($imp (or (not true) true))
Theorem:(if true true)
```

We continue with DeMorgan's laws. The next method derives $\neg p_1 \vee \neg p_2$ from $\neg(p_1 \wedge p_2)$:

```
>(define dm2
   (method (P)
     (dmatch P
       ((not (and p1 p2))
          (dlet ((p1-gives-not-p2 (assume p1
                                    (suppose-absurd p2
                                      ($absurd ($both ($both p1 p2) P))))))
            ($imp p1-gives-not-p2))))))
>dm2:prop ->> prop
```

```
>(assert (not (and ?A ?B)))
The proposition (not (and ?A ?B)) has been added to the assumption base.

>($dm2 (not (and ?A ?B)))
Theorem:(or (not ?A) (not ?B))
```

And the dual:

```
>(define dm2-dual
    (method (P)
      (dmatch P
        ((or (not p1) (not p2))
           (suppose-absurd (and p1 p2)
             (dlet ((p1-gives-not-p2 ($imp P))
                    (p1-holds ($left-and (and p1 p2)))
                    (p2-holds ($right-and (and p1 p2))))
                    (not-p2 ($mp p1-gives-not-p2 p1-holds))
               ($absurd ($both p2-holds not-p2))))))))
>dm2dual:prop ->> prop

>($dm2-dual (or (not ?A) (not ?B)))
Theorem:(not (and ?A ?B))
```

Finally we can write a flexible method that applies the appropriate DeMorgan's law depending on the form of the input:

```
>(define dm
   (method (P)
     (dmatch P
       ((not (or _ _)) ($dm1 P)
       ((and (not _) (not _))  ($dm1-dual P))
       ((not (and _ _))        ($dm2 P))
       ((or (not _) (not _))   ($dm2-dual P)))))
dm:prop ->> prop

>($dm (or (not ?A) (not ?B)))
Theorem:(not (and ?A ?B))

>($dm (not (and ?A ?B)))
Theorem:(or (not ?A) (not ?B))

>($dm (not (or ?x ?y)))
Theorem:(and (not ?x) (not ?y))

>($dm (and (not ?x) (not y)))
Theorem:(or (not ?x) (not ?y))
```

Next we implement the *exportation* law, which entitles us to infer either one of $(p_1 \wedge p_2) \Rightarrow p_3$, $p_1 \Rightarrow (p_2 \Rightarrow p_3)$ from the other.

```
>(define export
   (method (P)
     (dmatch P
       ((if p1 (if p2 p3))
          (assume-let ((p1-and-p2 (and p1 p2)))
            ($mp ($mp P ($left-and p1-and-p2))
                 ($right-and p1-and-p2))))
       ((if (and p1 p2) p3)
          (assume* (p1 p2)
            ($mp P ($both p1 p2)))))))
>export:prop ->> prop

>($assert (if ?x1 (if ?x2 ?x3)))
The proposition (if ?x1 (if ?x2 ?x3)) has been added to the assumption base.

>($export (if ?x1 (if ?x2 ?x3)))
Theorem:(if (and ?x1 ?x2) ?x3)

>($export (if (and ?x1 ?x2) ?x3))
Theorem:(if ?x1 (if ?x2 ?x3))
```

The following is an example of a recursive method. It takes a nested conditional of the form $p_1 \Rightarrow p_2 \Rightarrow \cdots \Rightarrow p_n \Rightarrow p$, for arbitrary $n \geq 2$, and keeps applying the exportation law, finally resulting in the proposition $(p_1 \wedge \cdots \wedge p_n) \Rightarrow p$:

```
>(define export1-rec
   (cletrec ((export-rec
     (method (P)
       (dmatch P
         ((if p1 (if p2 p3))
            (dlet ((res ($export P)))
              (dmatch p3
                ((if _  _) ($export-rec res))
                (_ res!))))))))
      export-rec))
export-rec:prop ->> prop

>(assert (if ?a1 (if ?a2 (if ?a3 ?b))))
The proposition (if ?a1 (if ?a2 (if ?a3 ?b))) has been added to the assumption base.

>($export1-rec (if ?a1 (if ?a2 (if ?a3 ?b))))
Theorem:(if (and (and ?a1 ?a2) ?a3) ?b)
```

The combination of recursion and higher-order values makes it possible to write powerful methods that can perform sophisticated proofs by induction on their input propositions. As an example, we show how to write a method `replace-all` that takes a theorem P and two propositions P_1, P_2 and returns the result of replacing every occurrence of P_1 in P by P_2. Of course this can only be done if P_1 and P_2 are logically equivalent,[24] so we specify that `replace-all` must take as an additional argument a method which may be applied to either one of P_1, P_2 to derive the other. As a simple example of what we will be able to do with `replace-all`, suppose P is the theorem

```
(and (not (if ?x ?y)) (or ?z1 ?z2))
```

and we want to dig inside P and replace `(if ?x ?y)` by the equivalent `(or (not ?x) ?y)`. One call to `replace-all` will do this: `($replace-all` P `(if ?x ?y) (or (not ?x) ?y) imp)` will replace all occurrences of `(if ?x ?y)` inside P by `(or (not ?x) ?y)`, using the fact that the `imp` method can prove that the replacement is equivalent to the part being replaced.

To implement `replace-all` we will use an auxiliary method `prove-equiv`, which takes two propositions P_1 and P_2 and a method m which can derive P_1 from P_2 and vice-versa, and derives the biconditional `(iff` P_1 P_2`)`:

```
>(define prove-equiv
   (method (P1 P2 deriv-method)
     (dlet ((P1-gives-P2 (assume P1
                           ($deriv-method P1)))
            (P2-gives-P1 (assume P2
                           ($deriv-method P2))))
       ($equiv P1-gives-P2 P2-gives-P1))))
prove-equiv:(prop * prop * (prop ->> prop)) ->> prop

>($prove-equiv (not (and ?x ?y)) (or (not ?x) (not ?y)) dm)
Theorem:(iff (not (and ?x ?y))
             (or (not ?x) (not ?y)))

>($prove-equiv ?x ?x claim)
Theorem:(iff ?x ?x)
```

We can now write the replacement method:

```
(define replace-all
  (method (P old new proof)
    (dletrec
        ((get-equivalence
           (method (P)
             (dmatch P
               ((val-of old) ($prove-equiv old new proof))
               ((not P1) (dlet ((P1-iff-newP1 ($get-equivalence P1)))
                           ($not-congruence P1-iff-newP1)))
               ((and P1 P2) (dlet ((P1-iff-newP1 ($get-equivalence P1))
                                   (P2-iff-newP2 ($get-equivalence P2)))
                              ($and-congruence P1-iff-newP1 P2-iff-newP2)))
               ((or P1 P2) (dlet ((P1-iff-newP1 ($get-equivalence P1))
                                  (P2-iff-newP2 ($get-equivalence P2)))
```

[24] In particular, it is not sufficient for P_2 to follow logically from P_1. For let $P =$ `(if ?x1 ?y)`, $P_1 =$ `?x1`, and $P_2 =$ `(or ?x1 ?x2)`. Here P_2 follows from P_1, yet the result of substituting P_2 for P_1 in P, namely `(if (or ?x1 ?x2) ?y)`, does *not* follow from P, a counter-example interpretation being `?x1` $\mapsto \mathbf{f}$, `?y` $\mapsto \mathbf{f}$, `?x2` $\mapsto \mathbf{t}$.

```
                        ($or-congruence P1-iff-newP1 P2-iff-newP2)))
             ((if P1 P2) (dlet ((P1-iff-newP1 ($get-equivalence P1))
                                (P2-iff-newP2 ($get-equivalence P2)))
                           ($if-congruence P1-iff-newP1 P2-iff-newP2)))
             ((iff P1 P2) (dlet ((P1-iff-newP1 ($get-equivalence P1))
                                 (P2-iff-newP2 ($get-equivalence P2)))
                            ($iff-congruence P1-iff-newP1 P2-iff-newP2)))
             (_ ($prove-equiv P P claim))))))
     (dlet ((P-iff-Pnew ($get-equivalence P))
            (P-gives-Pnew ($left-iff P-iff-Pnew)))
       ($mp P-gives-Pnew P)))))
```

Note that `replace-all` uses a local method `get-equivalence`, which inductively derives the biconditional `(iff P` P'`)`, where P' is the proposition obtained from `P` by replacing all occurrences of `old` by `new`. E.g. if `P` = `(not (if ?x ?y))`, `old` = `(if ?x ?y)`, `new` = `(or (not ?x) ?y)`, and `proof` = `imp`, then `($get-equivalence P)` will derive the biconditional

```
(iff (not (if ?x ?y))
     (not (or (not ?x ?y)))).
```

The main base case arises when `P` is `old`, and then `get-equivalence` deduces the desired biconditional simply by calling `($prove-equiv old new proof)` (recall that `proof` is supposed to be a method that can derive `new` from `old` and vice versa). The other base case is when `P` is an atom, and different than `old`. In that case the proposition P' obtained from `P` by replacing all occurrences of `old` by `new` is simply `P`, so the desired biconditional is `(iff P P)`, which is easily obtained by `($prove-equiv P P claim)`. In the rest of the cases, when `P` is compound and different than `old`, we proceed by induction. When `P` is a negation `(not` P_1`)`, we recursively call `($get-equivalence` P_1`)` to get the biconditional `(iff` P_1 P_1'`)`, where P_1' results from P_1 by replacing all occurrences of `old` by `new`. Then we simply pass this biconditional to `not-congruence`, which is a method that takes a theorem of the form `(iff` P_1 P_2`)` and returns the theorem `(iff (not` P_1`) (not` P_2`))`, which in our case will be the biconditional `(iff P` P'`)`. Similarly, `and-congruence` is a method that takes two theorems of the form `(iff` P_1 P_2`)` and `(iff` P_3 P_4`)` and derives the theorem

`(iff (and` P_1 P_3`) (and` P_2 P_4`))`.

Likewise for `or-congruence`, etc. These methods are easy to implement and we omit the definitions for simplicity. Finally, once the biconditional `(iff P` P'`)` has been derived, we simply use `left-iff` to get the conditional `(if P` P'`)`, and then `mp` on this and `P` to obtain P'.[25]

The examples in this section were meant to demonstrate the expressiveness of Athena even when nothing else besides primitive rules are used. In practice, most of the methods we derived in this section (`dm`, `imp`, `export`, `replace-all`, etc.) are so commonly used that, in the interest of convenience and efficiency, every Athena implementation should offer them as primitives.

4.5 Formal Semantics

We will present the evaluation semantics of Athena in the "natural" style of Kahn [18], a sort of big-step structural operational semantics [24] formulated as an abstract deduction system. This type

[25] A slightly more efficient implementation could result if we hoisted the base-case call `($prove-equiv old new proof)` outside of the recursive method `get-equivalence`, but that would lead to an improvement only when the input proposition `P` contained more than one occurrence of `old`.

$$\frac{}{\rho_1 \cdot [I \mapsto V] \cdot \rho_2, \beta \vdash I \rightsquigarrow V} \quad \text{[Ide]}$$

provided $I \notin \mathrm{Dom}(\rho_2)$

$$\frac{}{\rho, \beta \vdash X \rightsquigarrow \mathit{prop}(X)} \quad \text{[Var]}$$

$$\frac{}{\rho, \beta \vdash \texttt{(function (}I_1 \cdots I_n\texttt{)}\ E\texttt{)} \rightsquigarrow \mathit{fclos}(\texttt{(function (}I_1 \cdots I_n\texttt{)}\ E\texttt{)}, \rho, \beta)} \quad \text{[Function]}$$

$$\frac{}{\rho, \beta \vdash \texttt{(method (}I_1 \cdots I_n\texttt{)}\ D\texttt{)} \rightsquigarrow \mathit{mclos}(\texttt{(method (}I_1 \cdots I_n\texttt{)}\ D\texttt{)}, \rho)} \quad \text{[Method]}$$

$$\frac{\begin{array}{c}\rho, \beta \vdash E \rightsquigarrow \mathit{fclos}(\texttt{(function (}I_1 \cdots I_n\texttt{)}\ E_b\texttt{)}, \rho', \beta') \\ \rho, \beta \vdash F_1 \rightsquigarrow V_1 \quad \cdots \quad \rho, \beta \vdash F_n \rightsquigarrow V_n \quad \rho' \cdot [I_1 \mapsto V_1, \ldots, I_n \mapsto V_n], \beta' \vdash E_b \rightsquigarrow V\end{array}}{\rho, \beta \vdash \texttt{(}E\ \ F_1 \cdots F_n\texttt{)} \rightsquigarrow V} \quad \text{[FApp]}$$

$$\frac{\begin{array}{c}\rho, \beta \vdash F_1 \rightsquigarrow V_1 \quad \rho \cdot [I_1 \mapsto V_1], \beta \vdash F_2 \rightsquigarrow V_2 \quad \cdots \quad \rho \cdot [I_1 \mapsto V_1, \ldots, I_{n-1} \mapsto V_{n-1}], \beta \vdash F_n \rightsquigarrow V_n \\ \rho \cdot [I_1 \mapsto V_1, \ldots, I_n \mapsto V_n], \beta \vdash E \rightsquigarrow V\end{array}}{\rho, \beta \vdash \texttt{(clet ((}I_1\ \ F_1\texttt{)}\ \cdots\ \texttt{(}I_n\ \ F_n\texttt{))}\ E\texttt{)} \rightsquigarrow V} \quad \text{[CLet]}$$

$$\frac{\begin{array}{c}\rho \cdot [I_1 \mapsto E_1^*, \ldots, I_n \mapsto E_n^*], \beta \vdash E_1 \rightsquigarrow V_1 \quad \cdots \quad \rho \cdot [I_1 \mapsto E_1^*, \ldots, I_n \mapsto E_n^*], \beta \vdash E_n \rightsquigarrow V_n \\ \rho \cdot [I_1 \mapsto V_1, \ldots, I_n \mapsto V_n], \beta \vdash E \rightsquigarrow V\end{array}}{\rho, \beta \vdash \texttt{(cletrec ((}I_1\ \ E_1\texttt{)} \cdots \texttt{(}I_n\ \ E_n\texttt{))}\ E\texttt{)} \rightsquigarrow V} \quad \text{[CLetRec]}$$

$$\frac{\rho_1 \cdot [I_1 \mapsto E_1^*, \ldots, I_n \mapsto E_n^*], \beta \vdash E_j \rightsquigarrow V_j}{\rho_1 \cdot [I_1 \mapsto E_1^*, \ldots, I_n \mapsto E_n^*] \cdot \rho_2, \beta \vdash I_j \rightsquigarrow V_j} \quad \text{[Rec]}$$

for $j \in \{1, \ldots, n\}$, provided $I_j \notin \mathrm{Dom}(\rho_2)$

$$\frac{\rho, \beta \vdash F \rightsquigarrow \mathit{prop}(P) \quad \rho, \theta \vdash P \preceq Q_i \quad \rho \cdot \overrightarrow{\theta}, \beta \vdash E_i \rightsquigarrow V}{\rho, \beta \vdash \texttt{(cmatch}\ F\ \ \texttt{(}Q_1\ \ E_1\texttt{)} \cdots \texttt{(}Q_n\ \ E_n\texttt{))} \rightsquigarrow V} \quad \text{[CMatch]}$$

for $i \in \{1, \ldots, n\}$, provided $\mathrm{Dom}(\overrightarrow{\theta}) = \mathit{PtVar}(Q_i)$ and $\rho, \theta' \not\vdash P \preceq Q_j$ for all θ' and $j \in \{1, \ldots, i-1\}$.

Figure 8: Evaluation semantics, expressions.

of semantic specification is particularly appropriate for the deductive part of Athena, as it serves to expose the functional similarities between assumption bases and sequent contexts.

The semantics are syntax-based. We begin by defining an **environment** ρ as a finite list of ordered pairs (I, B) consisting of an identifier I and a *binding* B, where a binding is either a string of the form E^* (a "starred" expression E), or a **value** V. The domain of values **Val** defined as

$$V \in \mathbf{Val} = \mathbf{PropVal} \cup \mathbf{FVal} \cup \mathbf{MVal} \cup \mathbf{TopVal}$$

where

$$\frac{\rho, \beta \vdash F \rightsquigarrow prop(P_1) \qquad \rho, \beta \cup \{P_1\} \vdash D \rightsquigarrow prop(P_2)}{\rho, \beta \vdash \texttt{(assume } F \ D\texttt{)} \rightsquigarrow prop(\texttt{(if } P_1 \ P_2\texttt{))}} \text{ [Assume]}$$

$$\frac{\rho, \beta \vdash F \rightsquigarrow prop(P) \qquad \rho, \beta \cup \{P\} \vdash D \rightsquigarrow prop(\texttt{false})}{\rho, \beta \vdash \texttt{(suppose-absurd } F \ D\texttt{)} \rightsquigarrow prop(\texttt{(not } P\texttt{))}} \text{ [SupAbsurd]}$$

$$\frac{\rho, \beta \vdash E \rightsquigarrow V \qquad \rho \cdot [I \mapsto V], \beta \vdash D \rightsquigarrow prop(P)}{\rho, \beta \vdash \texttt{(dlet ((} I \ E\texttt{))} \ D\texttt{)} \rightsquigarrow prop(P)} \text{ [DLet-E]}$$

$$\frac{\rho, \beta \vdash D_1 \rightsquigarrow prop(P_1) \qquad \rho \cdot [I \mapsto prop(P_1)], \beta \cup \{P_1\} \vdash D_2 \rightsquigarrow prop(P_2)}{\rho, \beta \vdash \texttt{(dlet ((} I \ D_1\texttt{))} \ D_2\texttt{)} \rightsquigarrow prop(P_2)} \text{ [DLet-D]}$$

$$\frac{\rho, \beta \vdash \texttt{(dlet ((} I_1 \ F_1\texttt{)) (dlet ((} I_2 \ F_2\texttt{)} \ \cdots \ \texttt{(} I_n \ F_n\texttt{))} \ D\texttt{))} \rightsquigarrow prop(P)}{\rho, \beta \vdash \texttt{(dlet ((} I_1 \ F_1\texttt{)} \ \cdots \ \texttt{(} I_n \ F_n\texttt{))} \ D\texttt{)} \rightsquigarrow prop(P)} \text{ [DLet-Mult]}$$

$$\frac{\begin{array}{c}\rho \cdot [I_1 \mapsto E_1^*, \ldots, I_n \mapsto E_n^*], \beta \vdash E_1 \rightsquigarrow V_1 \quad \cdots \quad \rho \cdot [I_1 \mapsto E_1^*, \ldots, I_n \mapsto E_n^*], \beta \vdash E_n \rightsquigarrow V_n \\ \rho \cdot [I_1 \mapsto V_1, \ldots, I_n \mapsto V_n], \beta \vdash D \rightsquigarrow prop(P)\end{array}}{\rho, \beta \vdash \texttt{(dletrec ((} I_1 \ E_1\texttt{)} \cdots \texttt{(} I_n \ E_n\texttt{))} \ D\texttt{)} \rightsquigarrow prop(P)} \text{ [DLetRec]}$$

$$\frac{\begin{array}{c}\rho, \beta \vdash E \rightsquigarrow mclos(\texttt{(method (} I_1 \cdots I_n\texttt{)} \ D\texttt{)}, \rho') \\ \rho, \beta \vdash F_1 \rightsquigarrow V_1 \quad \cdots \quad \rho, \beta \vdash F_n \rightsquigarrow V_n \\ \rho' \cdot [I_1 \mapsto V_1, \ldots, I_n \mapsto V_n], \beta \cup ValidArgs \vdash D \rightsquigarrow prop(P)\end{array}}{\rho, \beta \vdash \texttt{(apply-method } E \ F_1 \cdots F_n\texttt{)} \rightsquigarrow prop(P)} \text{ [MApp]}$$

where $ValidArgs = \{P \mid V_i = prop(P) \text{ and } F_i \in \mathbf{Ded}, 1 \leq i \leq n\}$.

$$\frac{\rho, \beta \vdash F \rightsquigarrow prop(P_d) \qquad \theta \vdash prop(P_d) \preceq Q_i \qquad \rho \cdot \overrightarrow{\theta}, \beta \vdash D_i \rightsquigarrow prop(P)}{\rho, \beta \vdash \texttt{(dmatch } F \ \texttt{(} Q_1 \ D_1\texttt{)} \cdots \texttt{(} Q_n \ D_n\texttt{))} \rightsquigarrow prop(P)} \text{ [DMatch]}$$

for $i \in \{1, \ldots, n\}$, provided $\mathrm{Dom}(\theta) = PtVar(Q_i)$ and $\rho, \theta' \not\vdash P_d \preceq Q_j$ for all θ' and $j \in \{1, \ldots, i-1\}$.

Figure 9: Evaluation semantics, deductions.

$$\begin{array}{rcl}
\mathbf{PropVal} & = & \{prop(P)\} \\
\mathbf{FVal} & = & \{fclos(\texttt{(function (} I_1 \cdots I_n\texttt{)} \ E\texttt{)}, \rho, \beta)\} \\
\mathbf{MVal} & = & \{mclos(\texttt{(method (} I_1 \cdots I_n\texttt{)} \ D\texttt{)}, \rho)\} \\
\mathbf{TopVal} & = & \{op(f) \mid f \in \{\texttt{not}, \texttt{and}, \texttt{or}, \texttt{if}, \texttt{iff}\}\} \cup \{rule(r) \mid r \in PrimRule\}
\end{array}$$

and β stands for an **assumption base**, namely, a recursive subset of **Prop** that contains `true`.

Elements of **FVal** (**MVal**) are called *function* (*method*) *closures*. The "constructors" *prop*, *fclos*, *mclos*, *op*, and *rule* encode rudimentary type information and can be regarded simply as pieces of (meta-)syntax. We use the customary square bracket notation $[a_1, \ldots, a_n]$ to represent lists, an infix dot '$\cdot$' to denote list concatenation, and we write $I \mapsto B$ for the ordered pair (I, B). Finally, the

domain of an environment $\rho = [I \mapsto B_1, \ldots, I_n \mapsto B_n]$, Dom($\rho$), will be the set $\{I_1, \ldots, I_n\}$.

An **evaluation judgement** is defined as a quadruple (ρ, β, F, V) consisting of an environment ρ, an assumption base β, a phrase F, and a value V. We will symbolize such a judgement by the more suggestive notation $\rho, \beta \vdash F \rightsquigarrow V$, to be read as "Under environment ρ and assumption base β, phrase F evaluates to V", or, less operationally, "the value of F under ρ and β is V". The inference rules shown in figures 8 and 9 and in the four figures included in the appendix comprise an abstract deduction system $\mathcal{AS}$ on the set of all judgements, $\mathbf{Env} \times \mathcal{AB} \times \mathbf{Phr} \times \mathbf{Val}$, where **Env** is the set of all environments and $\mathcal{AB}$ the set of all assumption bases. Accordingly, an $\mathcal{AS}$-deduction of a judgement $\rho, \beta \vdash F \rightsquigarrow V$ from a set of judgements J is a sequence of judgements $j_1, \ldots, j_n$ such that each j_i is either a member of J, or an instance of an axiom in $\mathcal{AS}$, or follows from previous judgements through an inference rule in $\mathcal{AS}$. As usual, we write $J \vdash_{\mathcal{AS}} \rho, \beta \vdash F \rightsquigarrow V$ to indicate that such a deduction exists. When $J = \emptyset$ we will call this deduction an $\mathcal{AS}$**-derivation**, and will write $\vdash_{\mathcal{AS}} \rho, \beta \vdash F \rightsquigarrow V$ or simply $\rho, \beta \vdash F \rightsquigarrow V$ rather than $\emptyset \vdash_{\mathcal{AS}} \rho, \beta \vdash F \rightsquigarrow V$. If such a derivation exists we say that the judgement $\rho, \beta \vdash F \rightsquigarrow V$ is *derivable* in the natural semantics of Athena. We *define* the value of a phrase F in a given environment ρ and assumption base β to be the unique[26] value V such that the judgement $\rho, \beta \vdash F \rightsquigarrow V$ is derivable, *if* such a V exists; if not, the value of F in ρ and β is undefined. Operationally, this will be the case if the evaluation of F in ρ and β causes an error or fails to terminate.

4.5.1 Pattern matching

By a **propositional substitution** we will mean a finite set of pairs

$$\theta = \{(I_1, \mathit{prop}(P_1)), \ldots, (I_n, \mathit{prop}(P_n))\}$$

that is a function from **Ide** to **PropVal**; that is, $I_j \neq I_k$ whenever $P_j \neq P_k, 1 \leq j, k \leq n$. The notation $\overrightarrow{\theta}$ will be used in reference to an arbitrary listing (enumeration) of the pairs in a given θ.

Figure 10 shows a deduction system defining judgements of the form $\rho, \theta \vdash P \preceq Q$. We read such a judgement as "P is an **instance** of Q under environment ρ and substitution θ", or that "P **matches** Q under ρ and θ"; we also say that ρ and θ *prove* that P matches Q. As before, we write $I \mapsto P$ for the pair (I, P).

We find this deductive formulation of pattern matching cleaner than the usual algorithmic presentations. An an example, we give a proof that

$$\rho, \theta \vdash \texttt{(and ?foo (or (not ?a) ?foo))} \preceq \texttt{(and x (or (val-of y) x))}$$

where $\rho = [y \mapsto \mathit{prop}(\texttt{(not ?a)})]$ and $\theta = \{\texttt{x} \mapsto \mathit{prop}(\texttt{?foo})\}$:

1. $\rho, \theta \vdash \texttt{(not ?a)} \preceq \texttt{(val-of y)}$ [*BVarPat*]
2. $\rho, \theta \vdash \texttt{?foo} \preceq \texttt{x}$ [*IdPat*]
3. $\rho, \theta \vdash \texttt{(or (not ?a) ?foo)} \preceq \texttt{(or (val-of y) x)}$ 1, 2, [*CompPat*]
4. $\rho, \theta \vdash \texttt{(and ?foo (or (not ?a) ?foo))} \preceq \texttt{(and x (or (val-of y) x))}$ 2, 3, [*CompPat*]

$$\frac{}{\rho, \theta \vdash \texttt{true} \preceq \texttt{true}} \quad [\mathit{TruePat}]$$

$$\frac{}{\rho, \theta \vdash \texttt{false} \preceq \texttt{false}} \quad [\mathit{FalsePat}]$$

$$\frac{}{\rho, \theta \vdash X \preceq X} \quad [\mathit{VarPat}]$$

$$\frac{}{\rho, \theta \vdash P \preceq \texttt{any}} \quad [\mathit{AnyPat}]$$

$$\frac{}{\rho_1 \cdot [I \mapsto \mathit{prop}(P)] \cdot \rho_2, \theta \vdash P \preceq \texttt{(val-of } I\texttt{)}} \quad [\mathit{BVarPat}]$$

provided $I \notin \mathrm{Dom}(\rho_2)$.

$$\frac{}{\rho, \theta \cup \{I \mapsto \mathit{prop}(P)\} \vdash P \preceq I} \quad [\mathit{IdPat}]$$

$$\frac{\rho, \theta_1 \vdash P_1 \preceq Q_1 \cdots \; \rho, \theta_n \vdash P_n \preceq Q_n}{\rho, \theta_1 \cup \cdots \cup \theta_n \vdash \texttt{(}f \;\; P_1 \cdots P_n\texttt{)} \preceq \texttt{(}f \;\; Q_1 \cdots Q_n\texttt{)}} \quad [\mathit{CompPat}]$$

for $f \in \{\texttt{not}, \texttt{and}, \texttt{or}, \texttt{if}, \texttt{iff}\}, 1 \leq n \leq 2$.

Figure 10: Deductive definition of pattern matching.

4.6 A polymorphic type system

The **types** of the language can be described by the grammar

$$\tau ::= \alpha \mid \textbf{prop} \mid (\tau_1 \times \cdots \times \tau_n \to \tau) \mid (\tau_1 \times \cdots \times \tau_n \twoheadrightarrow \textbf{prop})$$

where α ranges over a set $\{S, T, S_1, T_1, S_2, \ldots\}$ of **type variables**. In addition to α, the letters β and γ will also be used as metavariables ranging over this set. The types $\tau_1 \times \cdots \times \tau_n \to \tau$ and $\tau_1 \times \cdots \times \tau_n \twoheadrightarrow \textbf{prop}$ are the types of functions and methods, respectively.

A **type schema** is either a type or an expression of the form $(\forall\alpha)\sigma$, where α is a type variable and σ a type schema. We will use the letters τ and σ to denote types and type schemas, respectively.

An occurrence of a type variable α in a type schema σ is **free** iff it is not within an expression of the form $(\forall\alpha)\sigma'$. The **free variables** of a type schema σ are those type variables that have free occurrences in σ. They can be inductively defined as follows:

$$\begin{aligned}
FV(\alpha) &= \{\alpha\} \\
FV(\textbf{prop}) &= \emptyset \\
FV((\tau_1 \times \cdots \times \tau_n \;\; op \;\; \tau)) &= FV(\tau_1) \cup \cdots \cup FV(\tau_n) \cup FV(\tau) \\
FV((\forall\alpha)\sigma) &= FV(\sigma) - \{\alpha\}
\end{aligned}$$

for $op \in \{\to, \twoheadrightarrow\}$. We say that α is "free in σ" iff $\alpha \in FV(\sigma)$. We will write $\sigma[\tau/\alpha]$ to signify the type

[26] A simple strucural induction will show that for given ρ, β, and F, $\hookrightarrow$ is a partial function, i.e. there is at most one V such that $\rho, \beta \vdash F \rightsquigarrow V$. The restriction that no element of *LogOp* or *PrimRule* may be bound in a phrase is essential in this respect.

$$\frac{}{\Gamma_1 \cdot [I : \sigma] \cdot \Gamma_2 \vdash I : \sigma} \quad [Ide]$$
provided $I \notin \mathrm{Dom}(\Gamma_2)$.

$$\frac{}{\Gamma \vdash X : \mathbf{prop}} \quad [Var]$$

$$\frac{\Gamma \cdot [I_1 : \tau_1, \ldots, I_n : \tau_n] \vdash E : \tau}{\Gamma \vdash \texttt{(function (}I_1 \cdots I_n\texttt{)}\ E\texttt{)} : (\tau_1 \times \cdots \times \tau_n \to \tau)} \quad [Fun]$$

$$\frac{\Gamma \vdash E : \tau_1 \times \cdots \times \tau_n \to \tau \quad \Gamma \vdash F_1 : \tau_1 \ \cdots \ \Gamma \vdash F_n : \tau_n}{\Gamma \vdash \texttt{(}E\ F_1\ \cdots\ F_n\texttt{)} : \tau} \quad [FunApp]$$

$$\frac{\Gamma \cdot [I_1 : \tau_1, \ldots, I_n : \tau_n] \vdash D : \mathbf{prop}}{\Gamma \vdash \texttt{(method (}I_1 \cdots I_n\texttt{)}\ D\texttt{)} : (\tau_1 \times \cdots \times \tau_n \twoheadrightarrow \mathbf{prop})} \quad [Meth]$$

$$\frac{\Gamma \vdash F_1 : \sigma_1 \quad \Gamma \cdot [I_1 : \sigma_1] \vdash F_2 : \sigma_2 \ \cdots \ \Gamma \cdot [I_1 : \sigma_1, \ldots, I_n : \sigma_n] \vdash E : \tau}{\Gamma \vdash \texttt{(clet ((}I_1\ F_1\texttt{)}\ \cdots\ \texttt{(}I_n\ F_n\texttt{))}\ E\texttt{)} : \tau} \quad [CLet]$$

$$\frac{\Gamma \cdot [I_1 : \tau_1, \ldots, I_n : \tau_n] \vdash E_1 : \tau_1 \quad \Gamma \cdot [I_1 : \tau_1, \ldots, I_n : \tau_n] \vdash E_n : \tau_n \quad \Gamma \cdot \Gamma' \vdash E : \tau}{\Gamma \vdash \texttt{(cletrec ((}I_1\ E_1\texttt{)}\ \cdots\ \texttt{(}I_n\ E_n\texttt{))}\ E\texttt{)} : \tau} \quad [CLetRec]$$
where $\Gamma' = [I_1 : (\forall \alpha_1^1) \cdots (\forall \alpha_{k_1}^1)\tau_1, \ldots, I_n : (\forall \alpha_1^n) \cdots (\forall \alpha_{k_n}^n)\tau_n]$
and, for $i = 1, \ldots, n$, $\{\alpha_1^i, \ldots, \alpha_{k_i}^i\} = FV(\tau_i) - FV(\Gamma)$.

$$\frac{\begin{array}{c}\Gamma \vdash F : \mathbf{prop} \quad \Gamma \cdot \Gamma_1' \vdash Q_1 : \mathbf{prop} \ \cdots \ \Gamma \cdot \Gamma_n' \vdash Q_n : \mathbf{prop} \\ \Gamma \cdot \Gamma_1 \vdash E_1 : \tau \ \cdots \ \Gamma \cdot \Gamma_n \vdash E_n : \tau\end{array}}{\Gamma \vdash \texttt{(cmatch}\ F\ \texttt{(}Q_1\ E_1\texttt{)}\ \cdots\ \texttt{(}Q_n\ E_n\texttt{))} : \tau} \quad [CMatch]$$
where for $i = 1, \ldots, n$, $\Gamma_i = [I_1^i : \mathbf{prop}, \ldots, I_{k_i}^i : \mathbf{prop}]$
$\{I_1^i, \ldots, I_{k_i}^i\} = PtVar(Q_i)$, and $\Gamma_i' = \Gamma_i \cdot [\texttt{any} : \mathbf{prop}, \texttt{val-of} : \mathbf{prop} \to \mathbf{prop}]$.

$$\frac{\Gamma \vdash E : \sigma}{\Gamma \vdash E : (\forall \alpha)\sigma} \quad [Gen]$$
provided $\alpha \notin FV(\Gamma)$.

$$\frac{\Gamma \vdash E : (\forall \alpha)\sigma}{\Gamma \vdash E : \sigma[\tau/\alpha]} \quad [Spec]$$

Figure 11: Type system, expressions.

schema obtained from σ by replacing every free occurrence of α by the *type* τ.[27]

A **type environment** Γ is a finite list of ordered pairs (I, σ), called **typings**, consisting of an

[27] We cannot replace a free type variable by an arbitrary type schema, as this would result in "non-shallow" type schemas such as $(\forall S)(\mathbf{prop} \times S \to (\forall T)(T \to T))$.

$$\frac{\Gamma \vdash F : \mathbf{prop} \quad \Gamma \vdash D : \mathbf{prop}}{\Gamma \vdash (\texttt{assume}\ F\ \ D) : \mathbf{prop}} \quad [\mathit{Assume}]$$

$$\frac{\Gamma \vdash F : \mathbf{prop} \quad \Gamma \vdash D : \mathbf{prop}}{\Gamma \vdash (\texttt{suppose-absurd}\ F\ \ D) : \mathbf{prop}} \quad [\mathit{Suppose\text{-}Absurd}]$$

$$\frac{\Gamma \vdash E : \tau_1 \times \cdots \times \tau_n \twoheadrightarrow \mathbf{prop} \quad \Gamma \vdash F_1 : \tau_1 \ \cdots \ \Gamma \vdash F_n : \tau_n}{\Gamma \vdash (\texttt{apply-method}\ E\ \ F_1\ \cdots\ F_n) : \mathbf{prop}} \quad [\mathit{MethodApp}]$$

$$\frac{\Gamma \vdash F_1 : \sigma_1 \quad \Gamma \cdot [I_1 : \sigma_1] \vdash F_2 : \sigma_2 \ \cdots \ \Gamma \cdot [I_1 : \sigma_1, \ldots, I_n : \sigma_n] \vdash D : \mathbf{prop}}{\Gamma \vdash (\texttt{dlet}\ ((I_1\ \ F_1)\ \cdots\ (I_n\ \ F_n))\ \ D) : \mathbf{prop}} \quad [\mathit{DLet}]$$

$$\frac{\begin{array}{c}\Gamma \cdot [I_1 : \tau_1, \ldots, I_n : \tau_n] \vdash E_1 : \tau_1 \ \cdots \ \Gamma \cdot [I_1 : \tau_1, \ldots, I_n : \tau_n] \vdash E_n : \tau_n \\ \Gamma \cdot \Gamma' \vdash E : \mathbf{prop}\end{array}}{\Gamma \vdash (\texttt{dletrec}\ ((I_1\ \ E_1)\ \cdots\ (I_n\ \ E_n))\ \ E) : \mathbf{prop}} \quad [\mathit{DLetRec}]$$

where $\Gamma' = [I_1 : (\forall \alpha_1^1) \cdots (\forall \alpha_{k_1}^1)\tau_1, \ldots, I_n : (\forall \alpha_1^n) \cdots (\forall \alpha_{k_n}^n)\tau_n]$
and, for $i = 1, \ldots, n, \{\alpha_1^i, \ldots, \alpha_{k_i}^i\} = FV(\tau_i) - FV(\Gamma)$.

$$\frac{\begin{array}{c}\Gamma \vdash F : \mathbf{prop} \quad \Gamma \cdot \Gamma_1' \vdash Q_1 : \mathbf{prop} \cdots \Gamma \cdot \Gamma_n' \vdash Q_n : \mathbf{prop} \\ \Gamma \cdot \Gamma_1 \vdash D_1 : \mathbf{prop} \ \cdots \ \Gamma \cdot \Gamma_n \vdash D_n : \mathbf{prop}\end{array}}{\Gamma \vdash (\texttt{dmatch}\ F\ \ (Q_1\ \ D_1)\ \cdots\ (Q_n\ \ D_n)) : \mathbf{prop}} \quad [\mathit{DMatch}]$$

where for $i = 1, \ldots, n$, $\Gamma_i = [I_1^i : \mathbf{prop}, \ldots, I_{k_i}^i : \mathbf{prop}]$
$\{I_1^i, \ldots, I_{k_i}^i\} = \mathit{PtVar}(Q_i)$, and $\Gamma_i' = \Gamma_i \cdot [\texttt{any} : \mathbf{prop}, \texttt{val-of} : \mathbf{prop} \to \mathbf{prop}]$.

Figure 12: Type system, deductions.

identifier I and a type schema σ. The more suggestive notation $I : \sigma$ will be used as an alternative to (I, σ). We will use the same list notation for type environments that we did for semantic environments (e.g. $\cdot$ for list concatenation, etc.). And likewise, we define

$$\begin{array}{rcl}\mathrm{Dom}([]) & = & \emptyset \\ \mathrm{Dom}([I : \sigma] \cdot \Gamma) & = & \{I\} \cup \mathrm{Dom}(\Gamma).\end{array}$$

Further, if Γ is of the form $\Gamma_1 \cdot [I : \sigma] \cdot \Gamma_2$ and $I \notin \mathrm{Dom}(\Gamma_2)$ then the free variables of σ are also said to be *free in* Γ. We write $FV(\Gamma)$ for the set of all type variables that are free in Γ; these are "the free variables of Γ".

The deductive system shown in the three figures of this section define Athena's *typing judgements*, which are triples (Γ, F, σ) consisting of a type environment Γ, a phrase F, and a type schema σ. As is customary, such judgements are symbolized as $\Gamma \vdash F : \sigma$, and read as "In the type environment Γ, the type of F is σ". A type inference algorithm virtually identical to the well-known $\mathcal{W}$ algorithm of Milner [22] can be easily devised to discover the *most general* type schema of a given phrase F in a given type environment Γ, i.e. the most general σ such that $\Gamma \vdash F : \sigma$.

The type system of Athena will be much richer in the setting of predicate logic: it will include subtyping, and the user will be able to introduce arbitrary new datatypes.

$$\frac{}{\Gamma \vdash \texttt{true} : \textbf{prop}} \quad [\mathit{True}]$$

$$\frac{}{\Gamma \vdash \texttt{false} : \textbf{prop}} \quad [\mathit{True}]$$

$$\frac{}{\Gamma \vdash \texttt{not} : \textbf{prop} \to \textbf{prop}} \quad [\mathit{Not}]$$

$$\frac{}{\Gamma \vdash f : \textbf{prop} \times \textbf{prop} \to \textbf{prop}} \quad [\mathit{BLogOp}]$$

for $f \in \{\texttt{and}, \texttt{or}, \texttt{if}, \texttt{iff}\}$.

$$\frac{}{\Gamma \vdash g : \textbf{prop} \twoheadrightarrow \textbf{prop}} \quad [\mathit{UnRule}]$$

for $g \in \{\texttt{claim}, \texttt{dn}, \texttt{absurd}, \texttt{left-and}, \texttt{right-and}, \texttt{left-iff}, \texttt{right-iff}\}$

$$\frac{}{\Gamma \vdash h : \textbf{prop} \times \textbf{prop} \twoheadrightarrow \textbf{prop}} \quad [\mathit{BinRule}]$$

for $h \in \{\texttt{mp}, \texttt{both}, \texttt{either}, \texttt{equiv}\}$

$$\frac{}{\Gamma \vdash \texttt{cd} : \textbf{prop} \times \textbf{prop} \times \textbf{prop} \twoheadrightarrow \textbf{prop}} \quad [\mathit{CD}]$$

Figure 13: Type system, top names.

4.7 Basic metatheory

We begin by recasting the usual semantics of propositional logic in the context of Athena. First, recall that **Var** is the set of all Athena *propositional variables*, that is, expressions of the form `(var I)`. If we let Σ be the (first-order, single-sorted) function signature $\{\texttt{true}, \texttt{false}, \texttt{not}, \texttt{and}, \texttt{or}, \texttt{if}, \texttt{iff}\}$, where the arities are as expected, then we may identify **Prop** with the (unique) term algebra over Σ and **Var**. Let $B = (\{\mathbf{f}, \mathbf{t}\}, \mathbf{f}, \mathbf{t}, \neg, \wedge, \vee, \Rightarrow, \Leftrightarrow\})$ denote the two-element Boolean algebra $\{\mathbf{f}, \mathbf{t}\}$ with the usual operations defined on it. B can be seen as a Σ-algebra under the expected correspondence (`true` to $\mathbf{t}$, `false` to $\mathbf{f}$, `and` to $\wedge$, etc.). Now by an **interpretation** $\mathcal{I}$ we will mean a function from **Var** to $\{\mathbf{f}, \mathbf{t}\}$. We will write $\hat{\mathcal{I}}$ for the unique homomorphic extension of $\mathcal{I}$ to **Prop**. Since $\hat{\mathcal{I}}$ is completely determined by $\mathcal{I}$, we may, for most purposes, conflate the two.

If $\mathcal{I}(P) = \mathbf{t}$ ($\mathcal{I}(P) = \mathbf{f}$) we say that $\mathcal{I}$ **satisfies** (resp. **falsifies**) P, or that it is a **model** of it. This is written as $\mathcal{I} \models P$. We call P **satisfiable** if it satisfied by some interpretation; a **tautology** (or **valid**) if it is satisfied by all interpretations; and **unsatisfiable** (or a **contradiction**) if it is not satisfied by any interpration. If $\mathcal{I} \models P$ for every $P \in \Gamma \subseteq \textbf{Prop}$ then we write $\mathcal{I} \models \Gamma$ and say that $\mathcal{I}$ satisfies (or is a model of) Γ. We regard the empty set of propositions as (vacuously) satisfied by every interpretation. We write $\Gamma_1 \models \Gamma_2$ to indicate that every model of Γ_1 is also a model of Γ_2. If this is the case we say that the elements of Γ_2 are **logical consequences** of (or "are logically implied by") the propositions in Γ_1. A single proposition P may appear in place of either Γ_1 or Γ_2 as an abbreviation for the singleton $\{P\}$.

The lemma below is a semantic version of the various deduction theorems and follows directly from the foregoing definitions:

Lemma 4.1 *For all $\beta \in \mathcal{AB}$ and $P_1, P_2 \in \mathbf{Prop}$, if $\beta \cup \{P_1\} \models P_2$ then $\beta \models$* `(if` P_1 P_2`)`.

We are now ready to prove the soundness of Athena:

Theorem 4.1 (Soundness) *For all $D \in \mathbf{Ded}, \rho \in \mathbf{Env}, \beta \in \mathcal{AB}$, and $P \in \mathbf{Prop}$,*

$$\textit{if } \rho, \beta \vdash D \rightsquigarrow \textit{prop}(P) \textit{ then } \beta \models P.$$

Proof. Using strong induction on the length of the $\mathcal{AS}$-derivation $j_1, \ldots, j_n$ of the judgement $j_n = \rho, \beta \vdash D \rightsquigarrow \textit{prop}(P)$, we proceed with a case analysis of the rule R by which j_n is derived from earlier judgements $j_{i_1}, \ldots, j_{i_k}$, $1 \leq i_1, \ldots, i_k < n$.

- Case 1: R is [Assume]. In that case P is of the form `(if` P_1 P_2`)`, and the inductive hypothesis implies that $\beta \cup \{P_1\} \models P_2$. Hence, by the preceding lemma, $\beta \models$ `(if` P_1 P_2`)`.

- Case 2: R is [SupAbsurd]. In that case P is of the form `(not` P'`)`, and the inductive hypothesis implies that $\beta \cup \{P'\} \models$ `false`. Therefore, by Lem. 4.1, $\beta \models$ `(if` P' `false)`, hence $\beta \models$ `(if (not false) (not` P'`))`, and since $\beta \models$ `(not false)`, $\beta \models$ `(not` P'`)` $= P$.

- Case 3: R is [DLet-E]. Here the result follows immediately from the inductive hypothesis.

- Case 4: R is [DLet-D]. In that case D is of the form `(dlet ((`I D_1`))` D_2`)`, and we have $\rho, \beta \vdash D_1 \rightsquigarrow \textit{prop}(P_1)$, $\rho, \beta \vdash D_2 \rightsquigarrow \textit{prop}(P)$. By the inductive hypothesis, $\beta \models P_1$, and $\beta \cup \{P_1\} \models P$. Thus $\beta \models \beta \cup \{P_1\}$, $\beta \cup \{P_1\} \models P$, hence, by the transitivity of the satisfaction relation, $\beta \models P$.

- Case 5: R is [DLet-Mult]. Here again the result follows immediately from the inductive hypothesis.

- Case 6: R is [DLetRec]. Immediately from the inductive hypothesis.

- Case 7: R is [MApp]. In that case D is of the form `(apply-method` E $F_1 \cdots F_n$`)`, and we have $\rho, \beta \vdash E \rightsquigarrow \textit{mclos}($`(method (`$I_1 \cdots I_n$`)` D'`)`$, \rho')$, $\rho, \beta \vdash F_i \rightsquigarrow V_i$, and $\rho', \beta \cup \textit{ValidArgs} \vdash D' \rightsquigarrow \textit{prop}(P)$. By the inductive hypothesis, $\beta \models \textit{ValidArgs}$, and also $\beta \cup \textit{ValidArgs} \models P$, hence $\beta \models P$.

- Case 8: R is [DMatch]—immediately from the inductive hypothesis.

- Case 9: R is a primitive method rule (one of the rules shown in the figures in the appendix). Here our task reduces to proving the soundness of the primitive rules, which is an easy exercise left for the reader (note that the inductive hypothesis must be used in all rules of the form [RuleName-$\cdots D \cdots$]). ■

Next we consider completeness. The logical completeness of the Athena kernel in tandem with the primitive inference rules should be apparent to anyone who has worked with natural deduction systems of the Kalish-Montague variety. An algorithm for translating deductions of the latter kind into Athena deductions practically writes itself: a three-column line numbered k such as "k. P Modus Ponens, i, j" where $i, j < k$ are numbers of earlier lines, translates into `(`P `by (mp` P_i P_j`))`, where P_i and P_j are the propositions in lines i and j, respectively; while nested boxes translate into `assume` or `suppose-absurd` expressions. In this way we can easily obtain a `dlet` or `dbegin` phrase that faithfully mirrors the original deduction. Such a translation method would prove that Athena is complete, provided that the system from which we are translating is known to be complete.

To demonstrate the expressive versatility of Athena, however, we will instead prove completeness by translating from a first-order axiom-based system, the system $\mathcal{K}$ shown in Fig. 1 (which is well-known to be complete). The following is the essence of our proof:

Theorem 4.2 *There is an algorithm which, given any $\mathcal{K}$-deduction $\mathcal{D}$ of a proposition P from a set of propositions $\{P_1, \ldots, P_n\}$, produces an Athena deduction $D_{\mathcal{D}}$ such that*

$$[], \{P_1, \ldots, P_n\} \vdash D_{\mathcal{D}} \rightsquigarrow prop(P).$$

Proof. First, for each axiom [A1]—[A14] of $\mathcal{K}$ we write an Athena method $\mathtt{A}i, i = \mathtt{1}, \ldots, \mathtt{14}$, which given propositions $P_1, P_2, \ldots$ to be used as instances of the metavariables $A, B, \ldots$ that occur in the formulation of the axioms in Fig. 1, produces a proof of the corresponding instance of the axiom. For example, the method for axiom [A3] is

```
(method (A B)
  (assume A
    (assume B
      ($claim A))))
```

or, more succinctly,

```
(method (A B) (assume* (A B) A!))).
```

This method produces a proof of *any* instance of [A3], depending on the actual arguments we pass to it. E.g. to prove the instance `(if (not true) (if ?foo (not true)))` in an environment in which the above method is named `A3`, we would invoke

```
($A3 (not true) ?foo).
```

Here is the list of all fourteen methods:

```
(define A1 (method () true!))

(define A2
  (method ()
    (suppose-absurd false
      false!)))

(define A3
  (method (A B)
    (assume* (A B)
      A!)))

(define A4
  (method (A B C)
    (assume-let ((A->B->C (if A (if B C)))
                 (A->B (if A B))
                 (A-holds A))
      ($mp ($mp A->B->C A-holds)
           ($mp A->B A-holds)))))

(define A5
  (method (A B)
    (assume-let ((x (if (not A) B))
                 (y (if (not A) (not B))))
```

```
    ($dn (suppose-absurd (not A)
           ($absurd ($both ($mp x (not A))
                           ($mp y (not A)))))))))

(define A6
  (method (A B)
    (assume* (A B)
      ($both A B))))

(define A7
  (method (A B)
    (assume (and A B)
      ($left-and (and A B)))))

(define A8
  (method (A B)
    (assume (and A B)
      ($right-and (and A B)))))

(define A9
  (method (A B C)
    (assume-let ((case1 (if A C))
                 (case2 (if B C))
                 (either-way (or A B)))
      ($cd either-way case1 case2))))

(define A10
  (method (A B)
    (assume A
      ($either A B))))

(define A11
  (method (A B)
    (assume B
      ($either A B))))

(define A12
  (method (A B)
    (assume-let ((part1 (if A B))
                 (part2 (if B A)))
      ($equiv part1 part2))))

(define A13
  (method (A B)
    (assume-let ((bicond (iff A B)))
      ($left-iff bicond))))
```

```
(define A14
  (method (A B)
    (assume-let ((bicond (iff A B)))
      ($right-iff bicond))))
```

Let $\rho_{\mathcal{K}}$ be the environment obtained by evaluating these definitions in the empty environment and assumption base. Now, given $\mathcal{D}$, we define $D_{\mathcal{D}}$ to be the Athena deduction

(dlet ((prop1 D_1)
$\vdots$
(prop$n$ D_n))
prop$n$!)

where $n \geq 1$ is the length of $\mathcal{D}$ and for $i \in \{1, \ldots, n\}$, D_i is defined as follows:

- if the i^{th} element of $\mathcal{D}$ is a premise P, then D_i is the claim P!;
- if the i^{th} element of $\mathcal{D}$ is an instance of an axiom [Aj], then D_i is the application (Aj$ $P_1 \cdots P_k$), where $P_1, \ldots, P_k$ are the appropriate instances of the metavariables of [Aj]; and
- if the i^{th} element of $\mathcal{D}$ is obtained from two earlier propositions P_{j_1} and P_{j_2} through Modus Ponens then E_i is ($mp prop$j_1$ propj_2).

To illustrate, the translation of the following $\mathcal{K}$-deduction of $p \Rightarrow r$ from $\{p \Rightarrow q, q \Rightarrow r\}$:

1.	$(q \Rightarrow r) \Rightarrow [p \Rightarrow (q \Rightarrow r)]$	[A3]
2.	$q \Rightarrow r$	Premise
3.	$p \Rightarrow (q \Rightarrow r)$	1, 2, MP
4.	$[p \Rightarrow (q \Rightarrow r)] \Rightarrow [(p \Rightarrow q) \Rightarrow (p \Rightarrow r)]$	[A4]
5.	$(p \Rightarrow q) \Rightarrow (p \Rightarrow r)$	4, 3, MP
6.	$p \Rightarrow q$	Premise
7.	$p \Rightarrow r$	5, 6, MP

is

```
(dlet ((prop1 ($A3 (if ?q ?r) ?p))
       (prop2 (if ?q ?r)!)
       (prop3 ($mp prop1 prop2))
       (prop4 ($A4 ?p ?q ?r))
       (prop5 ($mp prop3 prop4))
       (prop6 (if ?p ?q)!)
       (prop7 ($mp prop5 prop6)))
   prop7!)
```

Returning to the general case, it is now an easy exercise to prove by induction on n that

$$\rho_{\mathcal{K}}, \{P_1, \ldots, P_n\} \vdash D_{\mathcal{D}} \rightsquigarrow prop(P).$$

If we package the definitions of A1—A14 in a single dlet whose body is $D_{\mathcal{D}}$ then the reference to $\rho_{\mathcal{K}}$ may be omitted and the result may be phrased in terms of the empty environment, as in the statement of the theorem. ■
We conclude:

Theorem 4.3 (Completeness) *If* $\{P_1, \ldots, P_n\} \models P$ *then there is an Athena deduction* D *such that* $[], \{P_1, \ldots, P_n\} \vdash D \leadsto prop(P)$.

We conclude this section with a few remarks on the extensional power of Athena's functions and methods. Let f be any (partial) function from $\mathbf{Prop}^n$ to $\mathbf{Prop}$. We call f (partially) *Athena-computable* iff there is an Athena expression E_f of the form `(function` $(I_1 \cdots I_n)$ E`)` such that for all $P_1, \ldots, P_n, P$,

$$[], \emptyset \vdash (E_f \;\; P_1 \cdots P_n) \leadsto prop(P) \quad \text{iff} \quad f(P_1, \ldots, P_n) = P.$$

Church's thesis in the present context states that f is Athena-computable iff there is an algorithm which can produce the value $f(P_1, \ldots, P_n)$ for any given $P_1, \ldots, P_n$. If one accepts Church's thesis for the lambda calculus then one must also accept this formulation of it, since the computational part of Athena is a superset of the lambda calculus.

Things get a little more interesting when we come to methods. Let us call a (partial) function $f : \mathbf{Prop}^n \to \mathbf{Prop}$ (partially) *Athena-deducible* iff there is an Athena method M_f of the form `(method` $(I_1 \cdots I_n)$ D`)` such that for all $P_1, \ldots, P_n, P$,

$$[], \emptyset \vdash (\texttt{apply-method}\ M_f \;\; P_1 \;\cdots\; P_n) \leadsto prop(P) \quad \text{iff} \quad f(P_1, \ldots, P_n) = P.$$

It is obvious from the soundness theorem that, in the context of Athena, deducibility is a much stronger notion than computability, i.e. the class of Athena-deducible functions does not include (although it is included in, as we discuss below) the class of Athena-computable functions; or, more plainly put, "that which can be deduced can be computed, but that which can be computed cannot necessarily be deduced". The function $f(P) = \texttt{false}$, for instance, is trivially Athena-computable but not Athena-deducible. A necessary condition for f to be Athena-deducible is that $f(P_1, \ldots, P_n)$ be a tautology for all $P_1, \ldots, P_n$. But this is clearly not a sufficient condition. Consider for instance an effective enumeration of $\mathbf{Prop}$, $P^1, P^2, \ldots, P^n, \ldots$ and define

$$f(P^i) = \begin{cases} \texttt{true} & \text{if Turing machine \#i halts on } i \\ \texttt{(not false)} & \text{otherwise} \end{cases}$$

for some standard Gödel numbering # of Turing machines. Clearly, $f(P)$ is a tautology for all P, yet f is not Athena-deducible.

The answer lies at the intersection of the two domains in question:

Empirical proposition 4.1 (Church's thesis for Athena-deducibility) *A function* f *from* $\mathbf{Prop}^n$ *to* $\mathbf{Prop}$ *is Athena-deducible iff for all* $P_1, \ldots, P_n$*, the value* $f(P_1, \ldots, P_n)$ *is a tautology that can be obtained mechanically from* $P_1, \ldots, P_n$.

This is a rendition of Church's thesis specific to the deductive part of Athena. A simple way to establish it in one direction is to observe that all Athena-deducible functions are also Athena-computable. For if there is a method `(method` $(I_1 \cdots I_n)$ D`)` that deduces $f(P_1, \ldots, P_n)$ for all $P_1, \ldots, P_n$, then the function

$$\texttt{(function}\ (I_1 \cdots I_n)\ \texttt{((function (x) x) (\$(method}\ (I_1 \cdots I_n)\ D\texttt{)}\ I_1 \cdots I_n\texttt{)))}$$

computes f. Therefore, the soundness theorem in combination with Church's thesis for Athena functions entails the only-if part of Prop. 4.1. The other direction is deeper and does not admit of irrefutable argument.

4.8 Efficiency

Athena allows us to write and evaluate deductions at a very high level of abstraction, but from the standpoint of efficiency there seems to be one respect in which there is good room for improvement. Consider a very lengthy and complicated deduction that draws a very large number of intermediate inferences, making thousands of method calls. It is conceivable that during the evaluation of such a deduction a method like `dm` (which applies DeMorgan's laws) might be called a great number of times. Each time it is invoked, the body of `dm` will execute the exact same deduction; the arguments will likely be different each time, but the same basic steps will be taken over and over again. In effect, this amounts to constructing an "on-the-fly" argument that the result of the method call follows logically from the arguments.

That is not the way humans work, however. Once a person has convinced himself that DeMorgan's laws are sound, he does not bother to *logically derive* the result from the argument every time he applies one of these laws. Instead, when given any proposition of the form $\neg(P_1 \wedge P_2)$ (for instance) as input, a person will go ahead and immediately produce $\neg P_1 \vee \neg P_2$ as output, without a second thought, knowing that it is sound to do so.

The Athena interpeter would clearly be much more intelligent if it could duplicate this behavior, provided we had some guarantee that soundness would be preserved every time corners would be cut in this manner. Roughly, what we would like in the ideal scenario is this: the interpreter somehow "checks" a method *once*, convinces itself that it is sound, and then simply "memorizes" the input/output behavior of the method. Later, when the method is applied, the interpeter immediately produces the right answer without taking the time to execute the body of the method; by "the right answer" we mean the answer that would be produced if the interpreter went ahead and executed the call. That is, after the first time, the method essentially becomes a "primitive" such as `mp` or `both`. More precisely, the method turns into a rewrite system.

Although there is no general way to accomplish this (it is not difficult to show that the problem is undecidable as formulated), we can go a long way by exploiting the nice properties that logical validity has with respect to propositional substitutions. The main theorem on which our optimization will be based is a well-known result of propositional logic:

Theorem 4.4 *If a proposition P is a tautology then so is every instance of P, i.e. so is every proposition obtainable from P by consistently replacing the latter's variables by arbitrary propositions.*

in tandem with

Theorem 4.5 *A proposition P is logically implied by a finite number of propositions $P_1, \ldots, P_n$ iff $P_1 \Rightarrow P_2 \Rightarrow \cdots \Rightarrow P_n \Rightarrow P$ is a tautology.*

The first of these theorems is also the tool that humans use, often subconsciously, to make the kind of shortcut we discussed above.

We now sketch the way in which these two theorems are used to let one write optimized methods. Athena allows the user to write a different kind of method, introduced by keyword `opt-method` rather than `method`. The body of such a method is more syntactically constrained than the body of an arbitrary method. Specifically, let ρ be the environment describing the bindings that are active at the top level at some point. An `opt-method` can then be introduced at the top level (and that is the only way to introduce such a method) in the following form:

(`opt-method` ($P_1 \;\cdots\; P_n$)
  ($a_1^1 : Q_1^1 \;\cdots\; a_{k_1}^1 : Q_{k_1}^1 \;\; D_1$)

$$\vdots$$

$(a^m_1 : Q^m_1 \ \cdots \ a^m_{k_m} : Q^m_{k_m} \ \ D_m)$)

where for every $i = 1, \ldots, m$,

- $\{a^i_1, \ldots, a^i_{k_i}\} \subseteq \{P_1, \ldots, P_n\}$ and $a^i_1 \neq \cdots \neq a^i_{k_i}$, i.e. $a^i_1, \ldots, a^i_{k_i}$ are distinct parameters.
- for every $j = 1, \ldots, k_i$, if Q^i_j contains a sub-pattern of the form `(val-of` I`)` then I is either one of the parameters $P_1, \ldots, P_n$ or a pattern variable introduced in a previous pattern, i.e. an element of $PtVar(Q^i_1) \cup \cdots \cup PtVar(Q^i_{j-1})$; and
- D_i is a deduction whose free variables are amongst the parameters $P_1, \ldots, P_n$, the pattern variables of $Q^i_1, \ldots, Q^i_{k_i}$, and $\mathrm{Dom}(\rho)$.

Note that an `opt-method` cannot take higher-order values as results.

As an example, here is how we might write the contrapositive law, Modus Tollens, and one of DeMorgan's laws as optimized methods:

```
>(define contra-opt
   (opt-method (P)
     (P:(if p1 p2) (assume (not p2)
                     (suppose-absurd p1
                       ($absurd ($both ($mp P p1)
                                       (not p2))))))))
opt-contra:prop ->> prop

>(define mt-opt
   (opt-method (P1 P2)
     (P1:(if x y) P2:(not (val-of y)) ($mp ($contra-opt P1) P2))))
opt-mt:prop ->>prop

>(define dm1-opt
   (opt-method (P)
     (P:(not (or p1 p2))
          (dlet ((not-p1 (suppose-absurd p1
                           ($absurd ($both ($either p1 p2) P))))
                 (not-p2 (suppose-absurd p2
                           ($absurd ($both ($either p1 p2) P)))))
            ($both not-p1 not-p2)))))
dm1:prop ->> prop
```

It should be clear from these examples that, despite the syntactic constraints, optimized methods look almost identical to regular methods and are equally easy to write.

Without going into all the details, let us explain what happens when the user defines an `opt-method`. The idea is to symbolically evaluate each deduction $D_i, i = 1, \ldots, m$ in the body using fresh propositional variables in place of the parameters and the pattern variables (i.e. propositional variables that do not occur in D_i). The evaluation of each D_i takes place in the environment ρ and the empty assumption base augmented with the propositions obtained from the parameters and the patterns in the i^{th} arm of the body by substituting in the said propositional variables. If the deduction goes through resulting in a proposition P, then the reverse substitution occurs in P, i.e. the fresh propositional variables that originally replaced the parameters and the pattern variables in D_i are now replaced by the latter, and the result becomes the right-hand side of a rewrite rule (the left-hand side is obtained directly from the patterns). By the soundness theorem of Athena, we know that P is logically implied by the propositions we obtained from the patterns and the parameters by putting in the fresh variables. Hence by Th. 4.5, we know that the result of joining the latter propositions in a chain of implications and attaching P as the last consequent is a tautology. Hence, by Th. 4.4, we know that all instances of this tautology are also tautologies. The rewrite rule we formed captures precisely these instances, hence applying the rule will always be sound if the arguments are in the assumption base.

If this process fails for some D_i then the definition is rejected with a message stating that the method is unsound. Otherwise the system internalizes the constructed rewrite system, and every time the method is applied thereafter the interpreter simply checks to ensure that the arguments are in the current assumption base and then immediately produces the correct answer in accordance with the rewrite rules (if no match is found, an error is raised as usual). The deductions D_i are never again evaluated after the first time. In fact checking whether the arguments are in the assumption base can also be averted. Specifically, when each D_i is symbolically evaluated, the interpreter checks to see whether the result is a tautology. If it is, then whenever the arguments match the left-hand side of the i^{th} rule it is unnecessary to look them up in the assumption base, and the interpreter omits it. With a conservative static "logical strictness" analysis, the interpreter can avoid the argument look-up at run time *even* if the result of a D_i is not a tautology, provided that the analysis tells us that the argument is never logically used. In all of these cases the result is an extremely efficient method, with soundness intact.

In fact in many cases the Athena interpreter will figure out on its own how to construct an optimized version of a given method, and will do so tacitly. In such cases the users do not have to write `opt-methods` themselves. Examples of methods in this paper for which Athena can automatically derive optimized versions are `contra-positive`, `modus-tollens`, all of DeMorgan's laws, `imp`, and `export`. It is possible to query the system to find out whether it has managed to derive an optimized version of a certain method.

A Evaluation semantics of primitive inference rules and logical connectives

$$\frac{\rho,\beta \vdash E_m \rightsquigarrow \mathit{rule}(\texttt{mp}) \quad \rho,\beta \vdash E_1 \rightsquigarrow \mathit{prop}(\texttt{(if}\ P_1\ \ P_2\texttt{)}) \quad \rho,\beta \vdash E_2 \rightsquigarrow \mathit{prop}(P_1)}{\rho,\beta \cup \{\texttt{(if}\ P_1\ \ P_2\texttt{)}, P_1\} \vdash \texttt{(apply-method}\ E_m\ \ E_1\ \ E_2\texttt{)} \rightsquigarrow \mathit{prop}(P_2)} \text{[MP-EE]}$$

$$\frac{\rho,\beta \vdash E_m \rightsquigarrow \mathit{rule}(\texttt{mp}) \quad \rho,\beta \vdash E \rightsquigarrow \mathit{prop}(\texttt{(if}\ P_1\ \ P_2\texttt{)}) \quad \rho,\beta \vdash D \rightsquigarrow \mathit{prop}(P_1)}{\rho,\beta \cup \{\texttt{(if}\ P_1\ \ P_2\texttt{)}\} \vdash \texttt{(apply-method}\ E_m\ \ E\ \ D\texttt{)} \rightsquigarrow \mathit{prop}(P_2)} \text{[MP-ED]}$$

$$\frac{\rho,\beta \vdash E_m \rightsquigarrow \mathit{rule}(\texttt{mp}) \quad \rho,\beta \vdash D \rightsquigarrow \mathit{prop}(\texttt{(if}\ P_1\ \ P_2\texttt{)}) \quad \rho,\beta \vdash E \rightsquigarrow \mathit{prop}(P_1)}{\rho,\beta \cup \{P_1\} \vdash \texttt{(apply-method}\ E_m\ \ D\ \ E\texttt{)} \rightsquigarrow \mathit{prop}(P_2)} \text{[MP-DE]}$$

$$\frac{\rho,\beta \vdash E_m \rightsquigarrow \mathit{rule}(\texttt{mp}) \quad \rho,\beta \vdash D_1 \rightsquigarrow \mathit{prop}(\texttt{(if}\ P_1\ \ P_2\texttt{)}) \quad \rho,\beta \vdash D_2 \rightsquigarrow \mathit{prop}(P_1)}{\rho,\beta \vdash \texttt{(apply-method}\ E_m\ \ D_1\ \ D_2\texttt{)} \rightsquigarrow \mathit{prop}(P_2)} \text{[MP-DD]}$$

$$\frac{\rho,\beta \vdash E_m \rightsquigarrow \mathit{rule}(\texttt{dn}) \quad \rho,\beta \vdash E \rightsquigarrow \mathit{prop}(\texttt{(not (not}\ P\texttt{))})}{\rho,\beta \cup \{\texttt{(not (not}\ P\texttt{))}\} \vdash \texttt{(apply-method}\ E_m\ \ E\texttt{)} \rightsquigarrow \mathit{prop}(P)} \text{[DN-E]}$$

$$\frac{\rho,\beta \vdash E_m \rightsquigarrow \mathit{rule}(\texttt{dn}) \quad \rho,\beta \vdash D \rightsquigarrow \mathit{prop}(\texttt{(not (not}\ P\texttt{))})}{\rho,\beta \vdash \texttt{(apply-method}\ E_m\ \ D\texttt{)} \rightsquigarrow \mathit{prop}(P)} \text{[DN-D]}$$

$$\frac{\rho,\beta \vdash E_m \rightsquigarrow \mathit{rule}(\texttt{both}) \quad \rho,\beta \vdash E_1 \rightsquigarrow \mathit{prop}(P_1) \quad \rho,\beta \vdash E_2 \rightsquigarrow \mathit{prop}(P_2)}{\rho,\beta \cup \{P_1, P_2\} \vdash \texttt{(apply-method}\ E_m\ \ E_1\ \ E_2\texttt{)} \rightsquigarrow \mathit{prop}(\texttt{(and}\ P_1\ \ P_2\texttt{)})} \text{[Both-EE]}$$

$$\frac{\rho,\beta \vdash E_m \rightsquigarrow \mathit{rule}(\texttt{both}) \quad \rho,\beta \vdash E \rightsquigarrow \mathit{prop}(P_1) \quad \rho,\beta \vdash D \rightsquigarrow \mathit{prop}(P_2)}{\rho,\beta \cup \{P_1\} \vdash \texttt{(apply-method}\ E_m\ \ E\ \ D\texttt{)} \rightsquigarrow \mathit{prop}(\texttt{(and}\ P_1\ \ P_2\texttt{)})} \text{[Both-ED]}$$

$$\frac{\rho,\beta \vdash E_m \rightsquigarrow \mathit{rule}(\texttt{both}) \quad \rho,\beta \vdash D \rightsquigarrow \mathit{prop}(P_1) \quad \rho,\beta \vdash E \rightsquigarrow \mathit{prop}(P_2)}{\rho,\beta \cup \{P_2\} \vdash \texttt{(apply-method}\ E_m\ \ D\ \ E\texttt{)} \rightsquigarrow \mathit{prop}(\texttt{(and}\ P_1\ \ P_2\texttt{)})} \text{[Both-DE]}$$

$$\frac{\rho,\beta \vdash E_m \rightsquigarrow \mathit{rule}(\texttt{both}) \quad \rho,\beta \vdash D_1 \rightsquigarrow \mathit{prop}(P_1) \quad \rho,\beta \vdash D_2 \rightsquigarrow \mathit{prop}(P_2)}{\rho,\beta \vdash \texttt{(apply-method}\ E_m\ \ D_1\ \ D_2\texttt{)} \rightsquigarrow \mathit{prop}(\texttt{(and}\ P_1\ \ P_2\texttt{)})} \text{[Both-DD]}$$

$$\frac{\rho,\beta \vdash E_m \rightsquigarrow \mathit{rule}(\texttt{absurd}) \quad \rho,\beta \vdash E \rightsquigarrow \mathit{prop}(\texttt{(and}\ P\ \texttt{(not}\ P\texttt{))})}{\rho,\beta \cup \{\texttt{(and}\ P\ \texttt{(not}\ P\texttt{))}\} \vdash \texttt{(apply-method}\ E_m\ \ E\texttt{)} \rightsquigarrow \mathit{prop}(\texttt{false})} \text{[Absurd-E]}$$

$$\frac{\rho,\beta \vdash E_m \rightsquigarrow \mathit{rule}(\texttt{absurd}) \quad \rho,\beta \vdash D \rightsquigarrow \mathit{prop}(\texttt{(and}\ P\ \texttt{(not}\ P\texttt{))})}{\rho,\beta \vdash \texttt{(apply-method}\ E_m\ \ D\texttt{)} \rightsquigarrow \mathit{prop}(\texttt{false})} \text{[Absurd-D]}$$

Figure 14: Evaluation semantics, primitive inference rules, part 1.

$$\frac{\rho, \beta \vdash E \leadsto prop(P)}{\rho, \beta \cup \{P\} \vdash \texttt{(apply-method claim } E\texttt{)} \leadsto propP} \quad \text{[Claim-E]}$$

$$\frac{\rho, \beta \vdash D \leadsto prop(P)}{\rho, \beta \vdash \texttt{(apply-method claim } D\texttt{)} \leadsto propP} \quad \text{[Claim-E]}$$

$$\frac{\begin{array}{c}\rho, \beta \vdash E_m \leadsto rule(\texttt{cd}) \quad \rho, \beta \vdash E_1 \leadsto prop(\texttt{(or } P_1 \ P_2\texttt{)}) \\ \rho, \beta \vdash E_2 \leadsto prop(\texttt{(if } P_1 \ P_3\texttt{)}) \quad \rho, \beta \vdash E_3 \leadsto prop(\texttt{(if } P_2 \ P_3\texttt{)})\end{array}}{\begin{array}{c}\rho, \beta \cup \{\texttt{(or } P_1 \ P_2\texttt{)}, \texttt{(if } P_1 \ P_3\texttt{)}, \texttt{(if } P_2 \ P_3\texttt{)}\} \vdash \\ \texttt{(apply-method } E_m \ E_1 \ E_2 \ E_3\texttt{)} \leadsto prop(P_3)\end{array}} \quad \text{[CD-EEE]}$$

$$\vdots$$

$$\frac{\begin{array}{c}\rho, \beta \vdash E_m \leadsto rule(\texttt{cd}) \quad \rho, \beta \vdash E_1 \leadsto prop(\texttt{(or } P_1 \ P_2\texttt{)}) \\ \rho, \beta \vdash D \leadsto prop(\texttt{(if } P_1 \ P_3\texttt{)}) \quad \rho, \beta \vdash E_2 \leadsto prop(\texttt{(if } P_2 \ P_3\texttt{)})\end{array}}{\rho, \beta \cup \{\texttt{(or } P_1 \ P_2\texttt{)}, \texttt{(if } P_2 \ P_3\texttt{)}\} \vdash \texttt{(apply-method } E_m \ E_1 \ D \ E_2\texttt{)} \leadsto prop(P_3)} \quad \text{[CD-EDE]}$$

$$\vdots$$

$$\frac{\begin{array}{c}\rho, \beta \vdash E_m \leadsto rule(\texttt{cd}) \quad \rho, \beta \vdash D_1 \leadsto prop(\texttt{(or } P_1 \ P_2\texttt{)}) \\ \rho, \beta \vdash D_2 \leadsto prop(\texttt{(if } P_1 \ P_3\texttt{)}) \quad \rho, \beta \vdash D_3 \leadsto prop(\texttt{(if } P_2 \ P_3\texttt{)})\end{array}}{\rho, \beta \vdash \texttt{(apply-method } E_m \ D_1 \ D_2 \ D_3\texttt{)} \leadsto prop(P_3)} \quad \text{[CD-DDD]}$$

$$\frac{\rho, \beta \vdash E_m \leadsto rule(\texttt{equiv}) \quad \rho, \beta \vdash E_1 \leadsto prop(\texttt{(if } P_1 \ P_2\texttt{)}) \quad \rho, \beta \vdash E_2 \leadsto prop(\texttt{(if } P_2 \ P_1\texttt{)})}{\rho, \beta \cup \{\texttt{(if } P_1 \ P_2\texttt{)}, \texttt{(if } P_2 \ P_1\texttt{)}\} \vdash \texttt{(apply-method } E_m \ E_1 \ E_2\texttt{)} \leadsto prop(\texttt{(iff } P_1 \ P_2\texttt{)})} \quad \text{[EQ-EE]}$$

$$\frac{\rho, \beta \vdash E_m \leadsto rule(\texttt{equiv}) \quad \rho, \beta \vdash E \leadsto prop(\texttt{(if } P_1 \ P_2\texttt{)}) \quad \rho, \beta \vdash D \leadsto prop(\texttt{(if } P_2 \ P_1\texttt{)})}{\rho, \beta \cup \{\texttt{(if } P_1 \ P_2\texttt{)}\} \vdash \texttt{(apply-method } E_m \ E \ D\texttt{)} \leadsto prop(\texttt{(iff } P_1 \ P_2\texttt{)})} \quad \text{[EQ-ED]}$$

$$\frac{\rho, \beta \vdash E_m \leadsto rule(\texttt{equiv}) \quad \rho, \beta \vdash D \leadsto prop(\texttt{(if } P_1 \ P_2\texttt{)}) \quad \rho, \beta \vdash E \leadsto prop(\texttt{(if } P_2 \ P_1\texttt{)})}{\rho, \beta \cup \{\texttt{(if } P_2 \ P_1\texttt{)}\} \vdash \texttt{(apply-method } E_m \ D \ E\texttt{)} \leadsto prop(\texttt{(iff } P_1 \ P_2\texttt{)})} \quad \text{[EQ-DE]}$$

$$\frac{\rho, \beta \vdash E_m \leadsto rule(\texttt{equiv}) \quad \rho, \beta \vdash D_1 \leadsto prop(\texttt{(if } P_1 \ P_2\texttt{)}) \quad \rho, \beta \vdash D_2 \leadsto prop(\texttt{(if } P_2 \ P_1\texttt{)})}{\rho, \beta \vdash \texttt{(apply-method } E_m \ D \ E\texttt{)} \leadsto prop(\texttt{(iff } P_1 \ P_2\texttt{)})} \quad \text{[EQ-DD]}$$

Figure 15: Evaluation semantics, primitive inference rules, part 2.

$$\frac{\rho,\beta \vdash E_m \leadsto \mathit{rule}(\texttt{either}) \quad \rho,\beta \vdash E \leadsto \mathit{prop}(P_1) \quad \rho,\beta \vdash F \leadsto \mathit{prop}(P_2)}{\rho,\beta \cup \{P_1\} \vdash \texttt{(apply-method } E_m \; E \; F\texttt{)} \leadsto \mathit{prop}(\texttt{(or } P_1 \; P_2\texttt{)})} \text{ [Either-EF]}$$

$$\frac{\rho,\beta \vdash E_m \leadsto \mathit{rule}(\texttt{either}) \quad \rho,\beta \vdash F \leadsto \mathit{prop}(P_1) \quad \rho,\beta \vdash E \leadsto \mathit{prop}(P_2)}{\rho,\beta \cup \{P_2\} \vdash \texttt{(apply-method } E_m \; F \; E\texttt{)} \leadsto \mathit{prop}(\texttt{(or } P_1 \; P_2\texttt{)})} \text{ [Either-FE]}$$

$$\frac{\rho,\beta \vdash E_m \leadsto \mathit{rule}(\texttt{either}) \quad \rho,\beta \vdash D \leadsto \mathit{prop}(P_1) \quad \rho,\beta \vdash F \leadsto \mathit{prop}(P_2)}{\rho,\beta \vdash \texttt{(apply-method } E_m \; D \; F\texttt{)} \leadsto \mathit{prop}(\texttt{(or } P_1 \; P_2\texttt{)})} \text{ [Either-DF]}$$

$$\frac{\rho,\beta \vdash E_m \leadsto \mathit{rule}(\texttt{either}) \quad \rho,\beta \vdash F \leadsto \mathit{prop}(P_1) \quad \rho,\beta \vdash D \leadsto \mathit{prop}(P_2)}{\rho,\beta \vdash \texttt{(apply-method } E_m \; F \; D\texttt{)} \leadsto \mathit{prop}(\texttt{(or } P_1 \; P_2\texttt{)})} \text{ [Either-FD]}$$

$$\frac{\rho,\beta \vdash E_m \leadsto \mathit{rule}(\texttt{left-and}) \quad \rho,\beta \vdash E \leadsto \mathit{prop}(\texttt{(and } P_1 \; P_2\texttt{)})}{\rho,\beta \cup \{\texttt{(and } P_1 \; P_2\texttt{)}\} \vdash \texttt{(apply-method } E_m \; E\texttt{)} \leadsto \mathit{prop}(P_1)} \text{ [LeftAnd-E]}$$

$$\frac{\rho,\beta \vdash E_m \leadsto \mathit{rule}(\texttt{left-and}) \quad \rho,\beta \vdash D \leadsto \mathit{prop}(\texttt{(and } P_1 \; P_2\texttt{)})}{\rho,\beta \vdash \texttt{(apply-method } E_m \; D\texttt{)} \leadsto \mathit{prop}(P_1)} \text{ [LeftAnd-D]}$$

$$\frac{\rho,\beta \vdash E_m \leadsto \mathit{rule}(\texttt{right-and}) \quad \rho,\beta \vdash E \leadsto \mathit{prop}(\texttt{(and } P_1 \; P_2\texttt{)})}{\rho,\beta \cup \{\texttt{(and } P_1 \; P_2\texttt{)}\} \vdash \texttt{(apply-method } E_m \; E\texttt{)} \leadsto \mathit{prop}(P_2)} \text{ [RightAnd-E]}$$

$$\frac{\rho,\beta \vdash E_m \leadsto \mathit{rule}(\texttt{right-and}) \quad \rho,\beta \vdash D \leadsto \mathit{prop}(\texttt{(and } P_1 \; P_2\texttt{)})}{\rho,\beta \vdash \texttt{(apply-method } E_m \; D\texttt{)} \leadsto \mathit{prop}(P_2)} \text{ [RightAnd-D]}$$

$$\frac{\rho,\beta \vdash E_m \leadsto \mathit{rule}(\texttt{left-iff}) \quad \rho,\beta \vdash E \leadsto \mathit{prop}(\texttt{(iff } P_1 \; P_2\texttt{)})}{\rho,\beta \cup \{\texttt{(iff } P_1 \; P_2\texttt{)}\} \vdash \texttt{(apply-method } E_m \; E\texttt{)} \leadsto \mathit{prop}(P_1)} \text{ [LeftIff-E]}$$

$$\frac{\rho,\beta \vdash E_m \leadsto \mathit{rule}(\texttt{left-iff}) \quad \rho,\beta \vdash D \leadsto \mathit{prop}(\texttt{(iff } P_1 \; P_2\texttt{)})}{\rho,\beta \vdash \texttt{(apply-method } E_m \; E\texttt{)} \leadsto \mathit{prop}(P_1)} \text{ [LeftIff-D]}$$

$$\frac{\rho,\beta \vdash E_m \leadsto \mathit{rule}(\texttt{right-iff}) \quad \rho,\beta \vdash E \leadsto \mathit{prop}(\texttt{(iff } P_1 \; P_2\texttt{)})}{\rho,\beta \cup \{\texttt{(iff } P_1 \; P_2\texttt{)}\} \vdash \texttt{(apply-method } E_m \; E\texttt{)} \leadsto \mathit{prop}(P_2)} \text{ [RightIff-E]}$$

$$\frac{\rho,\beta \vdash E_m \leadsto \mathit{rule}(\texttt{right-iff}) \quad \rho,\beta \vdash D \leadsto \mathit{prop}(\texttt{(iff } P_1 \; P_2\texttt{)})}{\rho,\beta \vdash \texttt{(apply-method } E_m \; D\texttt{)} \leadsto \mathit{prop}(P_2)} \text{ [RightIff-D]}$$

Figure 16: Evaluation semantics, primitive inference rules, part 3.

$$\frac{}{\rho, \beta \vdash \texttt{true} \leadsto \mathit{prop}(\texttt{true})} \quad \text{[TrueEval]}$$

$$\frac{}{\rho, \beta \vdash \texttt{false} \leadsto \mathit{prop}(\texttt{false})} \quad \text{[FalseEval]}$$

$$\frac{}{\begin{array}{c}\rho, \beta \vdash f \leadsto \mathit{op}(f) \\ \text{for } f \in \{\texttt{not}, \texttt{and}, \texttt{or}, \texttt{if}, \texttt{iff}\}\end{array}} \quad \text{[OpEval]}$$

$$\frac{\rho, \beta \vdash E \leadsto \mathit{op}(\texttt{not}) \qquad \rho, \beta \vdash F \leadsto \mathit{prop}(P)}{\rho, \beta \vdash \texttt{(}E\ \ F\texttt{)} \leadsto \mathit{prop}(\texttt{(not}\ \ P\texttt{)})} \quad \text{[NotOp]}$$

$$\frac{\rho, \beta \vdash E \leadsto \mathit{op}(f) \qquad \rho, \beta \vdash F_1 \leadsto \mathit{prop}(P_1) \qquad \rho, \beta \vdash F_2 \leadsto \mathit{prop}(P_2)}{\begin{array}{c}\rho, \beta \vdash \texttt{(}E\ \ F_1\ \ F_2\texttt{)} \leadsto \mathit{prop}(\texttt{(}f\ \ P_1\ \ P_2\texttt{)}) \\ \text{for } f \in \{\texttt{and}, \texttt{or}, \texttt{if}, \texttt{iff}\}\end{array}} \quad \text{[BinOp]}$$

$$\frac{}{\begin{array}{c}\rho, \beta \vdash f \leadsto \mathit{rule}(f) \\ \text{for } f \in \mathit{PrimRule}\end{array}} \quad \text{[RuleEval]}$$

Figure 17: Evaluation semantics, logical connectives and primitive rules.

References

[1] P. B. Andrews, *An introduction to mathematical logic and type theory: to truth through proof*, Computer Science and Applied Mathematics, Academic Press, 1986.

[2] M. Bergmann, J. Moor, J. Nelson, *The Logic Book*, Random House, New York, 1980.

[3] E. W. Beth, *The foundations of mathematics*, North-Holland, 1959.

[4] E. W. Beth, *Semantic entailment and formal derivability*, Mededlingen der Koninklijke Nederlandse Akademie van Wetenschappen, 18(13):309–342, 1955.

[5] D. Clément, J. Despeyroux, T. Desperyoux, G. Kahn, *Natural Semantics on the Computer*, INRIA Research Report RR 416, INRIA-Sophia-Antipolis, June 1985.

[6] I. M. Copi, *Symbolic Logic*, 5th edition, Macmillan Publishing Co., New York, 1979,

[7] M. Davis, H. Putnam, *A computing procedure for quantification theory*, Journal of the ACM 7 (1960), 201–215.

[8] H. -D. Ebbinghaus, J. Flum, W. Thomas, *Mathematical Logic*, Springer-Verlag (2nd edition), New York, 1994.

[9] M. Fitting, *First-Order Logic and Automated Theorem Proving*, 2nd edition, Springer-Verlag, New York, 1996.

[10] D. M. Gabbay, *Classical vs Non-classical Logics*, in Handbook of Logic in Artificial Intelligence and Logic Programming, Vol. 2, Gabbay, Hogger, and Robinson (editors), Oxford Science Publications, Oxford, 1994.

[11] D. M. Gabbay, *Elements of Algorithmic Proof*, in Handbook of Logic in Computer Science, Vol. 2, Abramsky, Gabbay, and Maibaum (editors), Oxford Science Publications, Oxford, 1992.

[12] J. H. Gallier, *Logic for Computer Science*, Harper & Row Publishers, 1986.

[13] M. J. C. Gordon, T. F. Melham, *Introduction to HOL, a theorem proving environment for higher order logic*, Cambridge University Press, Cambridge, 1993.

[14] S. Haack, *Philosophy of Logics*, Cambridge University Press, 1978.

[15] I. Hacking, *What is Logic?*, in What is a Logical System?, Studies in Logic and Computation Series, Gabbay (editor), Oxford Science Publications, Oxford, 1994.

[16] W. Hodges, *Elementary Predicate Logic*, in Handbook of Philosophical Logic, Vol.1, D. Gabbay and F. Guenthner (Eds.)

[17] D. J. Israel, *The role(s) of Logic in Artifical Intelligence*, in Handbook of Logic in Artificial Intelligence and Logic Programming, Vol. 1, Gabbay, Hogger, and Robinson (editors), Oxford Science Publications, Oxford, 1993.

[18] G. Kahn, *Natural Semantics*, Proc. of Symposium on Theoretical Aspects of Computer Science, Passau, Germany, February 1987.

[19] D. Kalish, R. Montague, *Logic: Techniques of Formal Reasoning*, Harcourt, Brace and World, New York, 1964 (2nd edition, Harcourt Brace Jovanovich, 1980).

[20] S. C. Kleene, *Mathematical Logic*, John Wiley & Sons, 1967.

[21] Z. Manna, *Mathematical Theory of Computation*, McGraw-Hill Computer Science Series, 1974.

[22] R. Milner, *A theory of type polymorphism in programming.*, Journal of Computer and System Science, **17**, 348-75

[23] D. Prawitz, *Natural Deduction, A Proof-Theoretical Study*, Almqvist and Wiskell, Stockholm, 1965.

[24] G. D. Plotkin, *A Structural Approach to Operational Semantics*, DAIMI FN-19, Computer Science Department, Aarhus University, Aarhus, Denmark, September 1981.

[25] W. V. Quine, *Methods of Logic*, Holt, New York, 1950.

[26] M. Ryan, M. Sadler, *Valuation Systems and Consequence Relations*, in Handbook of Logic in Computer Science, Vol. 1, Abramsky, Gabbay, and Maibaum (editors), Oxford Science Publications, Oxford, 1992.

[27] R. M. Smullyan, *First-Order Logic*, Springer-Verlag, New York, 1968 (reprinted as a Dover edition in 1995).

[28] P. Suppes, *Introduction to Logic*, Van Nostrand, Princeton, NJ, 1957.

[29] M. E. Szabo, *The collected papers of Gerhard Gentzen*, North-Holland publishing company, Amsterdam, 1969.

[30] G. Takeuti, *Proof Theory*, Studies in Logic and the Foundations of Mathematics, Vol. 81, Barwise, Kaplan, Keisler, Suppes, Troelstra (editors), North-Holland, Amsterdam, 1975 (2nd edition, 1987).

[31] A. Tarski, *Fundamental concepts of the methodology of the deductive sciences*, in Logic, Semantics, Metamathematics, J. H. Woodger (ed.), Oxford University Press (1956), London, 1930.

[32] A. Tarski, *On the concept of logical consequence*, in Logic, Semantics, Metamathematics, J. H. Woodger (ed.), Oxford University Press (1956), London, 1936.

[33] A. S. Troelstra, H. Schwichtenberg, *Basic Proof Theory*, Cambridge Tracts in Theoretical Computer Science, Cambridge University Press, Cambridge, 1996.

[34] D. Van Dalen, *Logic and Structure*, Springer, Berlin, 1980.

[35] W. Wechler, *Universal Algebra for Computer Scientists*, Springer-Verlag, 1992.

[36] S. Wolfram, *Philosophical Logic*, Routledge, 1989.

[37] L. Zhongwan, *Mathematical Logic for Computer Science*, World Scientific Publishing, Singapore, 1989.