Evidence Semantics and Refinement Rules for First-Order Logics: Minimal, Intuitionistic, and Classical

Robert Constable

September 27, 2012

Abstract

This document provides a *computational semantics* for three versions of First-Order Logic (FOL): minimal, intuitionistic, and classical. Evidence semantics is a computational explanation of the meaning of first-order statements. We take this meaning to be the idea or concept expressed by first-order formulas relative to a particular type D as the domain of discourse and to propositional functions over D as the predicates, i.e. relative to a *first-order model*. A formula is *computationally valid* with respect to this semantics provided there is evidence for it in every computationally meaningful first-order model. The evidence gives a computational reason that the formula is true. This explanation provides a framework for thinking of validity with respect to *oracle-computations*, and that is perhaps sensible as a *relative computability* semantics for classical first-order logic.

The document also presents *refinement style* rules for various first-order logics including minimal logic, intuitionistic logic, and classical logic. Refinement style is a top down version of Gentzen's sequent calculus and is closely related to tableaux style proofs. It is noted that these rules are sound for evidence semantics as it applies to all three variants of FOL.

1 Evidence Semantics

This section reviews the semantics of evidence along the lines of [6]. Given any first-order signature $\mathcal{L}$ of predicates (relations) $P_i^{n_i}$ over a domain D of individuals of a model $\mathcal{M}$ for $\mathcal{L}$, we assign to every formula A over this signature a type of objects denoted $[A]_{\mathcal{M}}$ called the *evidence* for A with respect to $\mathcal{M}$. We normally leave off the subscript $\mathcal{M}$ when there is only one model involved. Here is how evidence is defined for the various kinds of first-order propositional functions. This definition will also implicitly provide a syntax of first-order formulas.

1.1 First-order formulas and their meaning

1. **atomic propositional functions** The domain of discourse, D can be any constructive type in the sense of data types such as $\mathbb{N}$ the natural numbers or $\mathbb{R}$ the computable real numbers or graphs or lists or trees and so forth. We do not analyze its structure further and do not examine the equality relation on the type when dealing with the pure first-order theory, as is standard practice. $P_i^{n_i}$ are interpreted as functions from D^{n_i} into $\mathbb{P}$ the **atomic propositions**, and for the atomic proposition $P_i^{n_i}(a_1, ..., a_{n_i})$, the basic evidence must be supplied, say objects p_i if there is evidence, otherwise the evidence type is empty. We include a *designated atomic proposition* (for minimal logic), $\perp$, called *bottom*, whose evidence type can be $\{\star\}$, or the empty type $\{\}$.

2. **conjunction** $[A\&B] \quad = \quad [A] \times [B]$, the **Cartesian product** of the evidence types.

3. **existential** $[\exists x.B(x)] \quad = \quad x : [D]_{\mathcal{M}} \times [B(x)]$, the **dependent product**.
4. **implication** $[A \Rightarrow B] \quad = \quad [A] \rightarrow [B]$, the **function space**[1]
5. **universal** $[\forall x.B(x)] \quad = \quad x : [D]_{\mathcal{M}} \rightarrow [B(x)]$, **dependent function space**.
6. **disjunction** $[A \vee B] \quad = \quad [A] + [B]$, **disjoint union**.
7. **false** $[False] \quad = \quad \{\}$ **empty set, void type**.

Intuitively, a formula A is satisfied in a model $\mathcal{M}$ if and only if there is evidence in $[A]_{\mathcal{M}}$. It is easy to prove a semantic equivalence theorem as done in the semantics of evidence articles [6].

Reference Theorem : For any model $\mathcal{M}$ of signature $\mathcal{L}$ and any first-order formula A,

$$\models_{\mathcal{M}} A \texttt{ iff } \exists \mathtt{a} \in [\mathtt{A}]$$

.

1.2 Proof expressions

Next we observe that we can assign meaning to proofs. They can be considered either as terms in the *meta logic* or as terms in the *object logic*, according to the "proofs as terms" principle (PAT). Here we view them as terms in the meta logic in order to keep the object logic standard. The rules include constraints on the subexpressions of a proof. This is especially natural in *refinement style logics* and *tableaux systems* [3, 14] as used in CTT and studied by Bates [2] and Griffin [10]. For each rule we provide a name that is the outer operator of a proof expression with slots to be filled in as the proof is developed. The partial proofs are organized as a tree generated in two passes. The first pass is top down, driven by the user creating terms with slots to be filled in on the algorithmic bottom up pass once the downward pass is complete. Here is a simple proof of the (constructive) tautology $A \Rightarrow (B \Rightarrow A)$.

$\vdash A \Rightarrow (B \Rightarrow A)$ *by* $\lambda(x.slot_1(x))$
$x : A \vdash (B \Rightarrow A)$ *by* $slot_1(x)$

In the next step, $slot_1(x)$ is replaced at the leaf of the tree by $\lambda(y.slot_2(x, y))$ to give:

$\vdash A \Rightarrow (B \Rightarrow A)$ *by* $\lambda(x.slot_1(x))$
$x : A \vdash (B \Rightarrow A)$ *by* $\lambda(y.slot_2(x, y))$ *for* $slot_1(x)$
$x : A, y : B \vdash A$ *by* $slot_2(x, y)$

When the proof is complete, we see the slots filled in at each inference step as in:

$\vdash A \Rightarrow (B \Rightarrow A)$ *by* $\lambda(x.\lambda(y.x))$
$x : A \vdash (B \Rightarrow A)$ *by* $\lambda(y.x)$
$x : A, y : B \vdash A$ *by* x

The variable x fills $slot_2$ and the lambda term $\lambda(y.x)$ fills $slot_1$. The complete proof expression is thus the lambda term $\lambda(x.\lambda(y.x))$. It is easy to see intuitively that the meaning of this term is precisely the evidence needed to show that the formula is true. If we used *typed lambda terms*, the proof term would be $\lambda(a : A.(\lambda(y : B.x)))$. (We could make the proof expression more standard if we used a name such as *impin* instead of λ; we prefer to use notation that makes the semantic ideas clearer than the traditional rule names.

Construction of the proof term requires that we say how the slots are to be filled as the proof progresses. In the above example there are no choices. We provide annotations in the rules that spell out the substitutions. For example, we say *by* $\lambda(y.slot2)$ into $slot1$.

[1] This function space is interpreted type theoretically and is assumed to consist of *effectively computable deterministic functions.*

We present the rules in a top down style showing the *construction rules* first, often called the *introduction rules* because they introduce the canonical proof terms or called the *right hand side* rules because they apply to terms on the right hand side of the turnstile. So typical names seen in the literature are these: for & we say $Andintro$ or $AndR$; for $\Rightarrow$ we say $Imp-intro$ or $ImpR$; for $\vee$ we say $Or-intro_r$ or OrR_r, and $Or-intro_l$ or OrR_l, for $\forall x$ we say $All-intro$ or $AllR$, and for $\exists$ we say $Existsintro$ or $ExistsR$.

For each of these construction rules, the constructor needs subterms which build the component pieces of evidence. Thus for $AndR$ the full term will have slots for the two pieces of evidence needed, the form will be $AndR(slot1, slot2)$ where the slots are filled in as the proof tree is expanded. When the object to be filled in depends on a new hypothesis to be added to the left hand side of the turnstile, the rule name must supply a unique label for the new hypothesis, so we see a rule name like $ImpR(x.slot(x))$ or $AllR(x.slot(x))$. In the case of the rule for $\exists$, there is a subtlety. The rule name provides two slots, but the second depends on the object built for the first, so we see rule names such as $ExistsR(a; slot(a))$.

For each connective and operator we also have rules for their occurrence on the left of the turnstile. These are the *rules for decomposing* or using or *eliminating* a connective or operator. They tell us how to use the evidence that was built with the corresponding construction rules, and the formula being decomposed is always named by a label in the list of hypotheses, so there is a variable associated with each rule application. Here are typical names: for & we say $Andelim(x)$ or $AndL(x)$. However, there must be more to this rule name because typically new formulas are added to the hypothesis list, one for each of the conjuncts, so we need to provide labels for these formulas. Thus the form of elimination for & is actually $AndL(x; l, r.slot(l, r))$ where l stands for the left conjunct and r for the right one.

Rule names such as $AndR$, $AndL$, OrR_l, OrR_r, OrL, and so forth are suggestive in terms of the details of the proof system, but they are not suggestive of the structure of the evidence, the semantics. We will use rule names that define the computational forms of evidence. The *evaluation rules* for these proof terms are given as in ITT or CTT, for instance in the book *Implementing Mathematics* [7] or in the Nuprl Reference Manual [12].
So instead of $AndR(a; b)$ where a and b are the subterms built by a completed proof by progressively filling in open slots, we use $pair(a; b)$ or even more succinctly $< a, b >$, and for the corresponding decomposition rule we use $spread(x; l, r.t(l, r))$ where the binding variables l, r have a scope that is the subterm $t(l, r)$. This term is a compromise between using more familiar operators for decomposing a pair p such as $first(p)$ and $second(p)$ or $p.1$ and $p.2$ with the usual meanings, e.g., $first(< a, b >) =< a, b > .1 = a$. The reason to use *spread* is that we need to indicate how the subformulas of $A\&B$ will be named in the hypothesis list.

The decomposition rules for $A \Rightarrow B$ and $\forall x.B(x)$ are the most difficult to motivate and use intuitively. Since the evidence for $A \Rightarrow B$ is a function $\lambda(x.b(x))$, a reader might expect to see a decomposition rule name such as $apply(f; a)$ or abbreviated to $ap(f; a)$. However, the standard Gentzen sequent-style proof rule for decomposing an implication has this form:
$H, f : A \Rightarrow B, H' \vdash G$ *by* $ImpL$ *on* f
1. $H, f : A \Rightarrow B, H' \vdash A$
2. $H, f : A \Rightarrow B, v : B, H' \vdash G$

As the proof proceeds, the two subgoals 1 and 2 with conclusions A and G respectively will be refined, say with proof terms $g(f, v)$ and a respectively. We need to indicate that the value v is

$ap(f;a)$, but at the point where the rule is applied, we only have slots for these subterms and a name v for the new hypothesis B. So the rule form is $apseq(f;slot_a;v.slot_g(v))$ where we know that v will be assigned the value $ap(f;slot_a)$ to "sequence" the two subgoals properly. So *apseq* is a sequencing operator as well as an application, and when the subterms are created, we can evaluate the term further as we show below. We thus express the rule as follows.
$H, f : A \Rightarrow B, H' \vdash G$ *by* $apseq(f;slot_a;v.slot_g(v))$
$H, f : A \Rightarrow B, v : B, H' \vdash G$ *by* $slot_g(v)$
$H, f : A \Rightarrow B, H' \vdash A$ *by* $slot_a$
We can evaluate the term $apseq(f;a;v.g(v))$ to $g(ap(f;a))$ or more succinctly to $g(f(a))$. This simplification can only be done on the final bottom up pass of creating a closed proof expression, one with no slots.
It would make sense to allow a rule of this form
$H, f : A \Rightarrow B, H' \vdash G$ *by* $apseq(f;slot_a;v.slot_g(v))$
$H, f : A \Rightarrow B, v : B, H' \vdash G$ *by* $slot_g(ap(f;slot_a))$
$H, f : A \Rightarrow B, H' \vdash A$ *by* $slot_a$
In this form, the variable v is not used in the extract and is present only to indicate where the application of the function named by f is to be applied. In the case of the All Decomposition rule, we can make this idea more explicit because we already have the argument to f specified in the rule, the term a.
Another variant we use in the main theorem records constraints on v as follows:

$H, f : A \Rightarrow B, H' \vdash G$ *by* $apseq(f;slot_a;v.slot_g(v))$
$H, f : A \Rightarrow B, H' \vdash A$ *by* $slot_a$
$H, f : A \Rightarrow B, v = ap(f;slot_a), v : B, H' \vdash G$ *by* $slot_g(ap(f;slot_a))$

With this introduction, we hope that the following rules will make good sense. These rules define what we will call the *pure proof expressions.*

1.3 First-order refinement style proof rules over domain of discourse D

Minimal Logic
Construction rules

- **And Construction**

  $H \vdash A\&B$ *by* $pair(slot_a;slot_b)$
  $H \vdash A$ *by* $slot_a$
  $H \vdash B$ *by* $slot_b$

- **Exists Construction**[2]

  $H \vdash \exists x.B(x)$ *by* $pair(d;slot_b(d))$
  $H \vdash d \in D$ *by* $obj(d)$
  $H \vdash B(d)$ *by* $slot_b(d)$

- **Implication Construction**

  $H \vdash A \Rightarrow B$ *by* $\lambda(x.slot_b(x))$ *new* x
  $H,\ x : A \vdash B$ *by* $slot_b(x)$

- **All Construction**

  $H \vdash \forall x.B(x)$ *by* $\lambda(x.slot_b(x))$ *new* x
  $H,\ x : D \vdash B(x)$ *by* $slot_b(x)$

[2]Also see Alternative Rules below.

- **Or Construction**

  $H \vdash A \vee B$ *by* $inl(slot_l)$
  $H \vdash A$ *by* $slot_l$

  $H \vdash A \vee B$ *by* $inr(slot_r)$
  $H \vdash B$ *by* $slot_r$

Decomposition rules

- **And Decomposition**

  $H, x : A\&B, H' \vdash G$ *by* $spread(x; l, r.slot_g(l, r))$ *new* l, r
  $H, l : A, r : B, H' \vdash G$ *by* $slot_g(l, r)$

- **Exists Decomposition**

  $H, x : \exists y.B(y), H' \vdash G$ *by* $spread(x; d, r.slot_g(d, r))$ *new* d, r
  $H, d : D, r : B(d), H' \vdash G$ *by* $slot_g(d, r)$

- **Implication Decomposition**

  $H, f : A \Rightarrow B, H' \vdash G$ *by* $apseq(f; slot_a; v.slot_g[ap(f; slot_a)/v])$ *new* v[3]
  $H, f : A \Rightarrow B, H' \vdash A$ *by* $slot_a$
  $H, f : A \Rightarrow B, H', v : B \vdash G$ *by* $slot_g(v)$

- **All Decomposition**

  $H, f : \forall x.B(x), H' \vdash G$ *by* $apseq(f; d; v.slot_g[ap(f; d)/v])$
  $H, f : \forall x.B(x), H' \vdash d \in D$ *by* $obj(d)$
  $H, f : \forall x.B(x), H', v : B(d) \vdash G$ *by* $slot_g(v)$[4]

- **Or Decomposition**

  $H, y : A \vee B, H' \vdash G$ *by* $decide(y; l.leftslot(l); r.rightslot(r))$
  1. $H, l : A, H' \vdash G$ *by* $leftslot(l)$
  2. $H, r : B, H' \vdash G$ *by* $rightslot(r)$

- **Hypothesis**

  $H, d : D, H' \vdash d \in D$ *by* $obj(d)$

  $H, x : A, H' \vdash A$ *by* $hyp(x)$
  We usually abbreviate the justifications to *by* d and *by* x respectively.

Intuitionistic Rules

- **False Decomposition**

  $H, f : False, H' \vdash G$ *by* $any(f)$

  This is the rule that distinguishes intuitionistic from minimal logic. We use the constant $False$ for intuitionistic formulas and $\bot$ for minimal ones to distinguish the logics. In practice, we would use only one constant, say $\bot$, and simply add the above rule with $\bot$ for $False$ to axiomatize iFOL. However, for our results it's especially important to be clear about the difference, so we use both notations.

[3] This notation shows that $ap(f; slot_a)$ is substituted for v in $g(v)$. In the CTT logic we stipulate in the rule that $v = ap(f; slot_a)$ *in* B.

[4] In the CTT logic, we use equality to stipulate that $v = ap(f; d)$ *in* $B(v)$ just before the hypothesis $v : B(d)$.

Note that we use the term d to denote objects in the domain of discourse D. In the classical evidence semantics, we assume that D is non-empty by postulating the existence of some d_0 in it. Also note that in the rule for $False$ Decomposition, it is important to use the $any(f)$ term which allows us to thread the explanation for how $False$ was derived into the justification for G. This will be important in proof construction from evidence.

Structural Rules

- **Cut rule**

  $H \vdash G$ *by* $CutC(x.slot_g(slot_c))$ *new* x
  1. $H, x : C \vdash G$ *by* $slot_g(x)$
  2. $H \vdash C$ *by* $slot_c$.

Classical Rules

- **Non-empty Domain of Discourse**

  $H \vdash d_0 \in D$ *by* $obj(d_0)$

- **Law of Excluded Middle (LEM)**

  Define $\sim A$ as $(A \Rightarrow False)$

  $H \vdash (A \vee \sim A)$ *by* **magic**(A)

  Note that this is the only rule that mentions a formula in the rule name.

Alterative Rule

The following rule for existential introduction is more in the style of *logic programming* rather than *functional programming* and was used in Edinburgh Nuprl (called Oyster [4]).

- **Exists Construction**

  $H \vdash \exists x.B(x)$ *by* $pair(slot_d/X; slot_b[slot_d/X])$
  $H \vdash D$ *by* $slot_d$
  $H \vdash B(X)$ *by* $slot_b(X)$
  Note, the substitution of $slot_d$ propagates to $B(X)$ as soon as the first subgoal determines the value of the slot for the goal rule. The term X acts as a *logic variable*.

1.4 Computation Rules

Each of the rule forms when completely filled in becomes a term in an applied lambda calculus [8, 5, 1], and there are *computation rules* that define how to reduce these terms in one step. These rules are given in detail in several papers about Computational Type Theory and Intuitionistic Type Theory, so we do not repeat them here. One of the most detailed accounts in the book *Implementing Mathematics* [7, 12] and in ITT82 [13].

Some parts of the computation theory are needed here, such as the notion that all the terms used in the rules can be reduced to *head normal form*. Defining that reduction requires identifying the *principal argument places* in each term. Here is how they are defined. The principle argument places are the ones identified mnemonically in the rules, thus in $ap(f; a)$ it is f for *function*, in $spread(p; x, y.g)$ it is p for *pair*, in $decide(d; l.left; r.right)$ it is d for *decision*.

The reduction rules are simple. For $ap(f; a)$, first reduce f, if it becomes a function term, $\lambda(x.b)$, then reduce the function term to $b[a/x]$, that is, substitute the argument a for the variable x in

the body of the function b and continue computing. If it does not reduce to a function, then no further reductions are possible and the computation aborts. It is possible that such a reduction will abort or continue indefinitely. But the terms arising from proofs will always reduce to head normal form. This fact is discussed in the references.
To reduce $spread(p; x, y.g)$, reduce the principal argument p. If it does not reduce to $pair(a; b)$, then there are no further reductions, otherwise, reduce $g[a/x, b/y]$.

To reduce $decide(d; l.left; r.right)$, reduce the principal argument d until it becomes either $inl(a)$ or $inr(b)$ or aborts or fails to terminate.[5]. In the first case, continue by reducing $left[a/l]$ and in the other case, continue by reducing $right[b/r]$.

It is important to see that none of the first-order proof terms is recursive, and it is not possible to hypothesize such terms without adding new computation forms. It is thus easy to see that all evidence terms terminate on all inputs from all models. We state this below as a theorem about valid evidence structures.

Theorem 1 Every uniform evidence term for minimal, intuitionistic, and classical logic denotes canonical evidence, and the functional terms terminate on any inputs from any model.

Additional notations It is useful to generalize the semantic operators to n-ary versions. For example, we will write λ terms of the form $\lambda(x_1, ..., x_n.b)$ and a corresponding n-ary application, $f(x_1, ..., x_n)$. We allow n-ary conjunctions and n-tuples which we decompose using $spread_n(p; x_1, ..., x_n.b)$. More rarely we use n-ary disjunction and the decider, $decide_n(d; case_1.b_1; ...; case_n.b_n)$. It is clear how to extend the computation rules and how to define these operators in terms of the primitive ones.
It is also useful to define $True$ to be the type $\bot \Rightarrow \bot$ with element $id = \lambda(x.x)$. Note that $\lambda(x.spread(pair(x; x); x_1, x_2.x_1))$ is computationally equivalent to id [11], as is $\lambda(x.decide(inr(x); l.x; r.x))$.[6].

1.5 Consistency of First-Order Refinement Logic

We can prove a simple *consistency theorem* for our refinement style axiomatization of the first-order logics iFOL and mFOL. We show that a logical function provable in iFOL (hence in mFOL) is *uniformly valid*.[7] Moreover, the evidence term is *purely logical* in the sense that the only terms used are the logical operators from the rules.
We do not discuss this method here, but the reader can imagine that we have access to the syntax of the proof expression and can analyze it to say that only logical operators are used.[8] For any type T, the type T^0 is the type of all closed terms that belong to T.

Theorem 2 - Soundness and Consistency: If
$\vdash_i^1 \ \forall\ D : Type, m : \mathbb{N}^+, r : \mathbb{N}_m^+, F : \mathcal{L}_{m,r}(D) \rightarrow \mathbb{P}, \bar{R} : \mathcal{L}_{m,r}(D).F(\bar{R})$, then
$\exists\ evd : (\bar{R} \bigcap [F(\bar{R})])^0.Pure(evd)$.

[5]The computation systems of CTT and ITT include diverging terms such as $fix(\lambda(x.x))$. We sometimes let $\uparrow$ denote such terms

[6]We could also use the term $\lambda(x.decide(inr(x); l.div; r.r))$ and normalization would reduce it to $\lambda(x.x)$

[7]We write $\vdash_i$ to indicate intuitionistic provability and $\vdash_m$ *in* to indicate mFOL provability.

[8]The way we say that the evidence is purely logical is to define an operator $Pure$ which operates on the syntactic structure of elements in the type $Base$, the type of all closed terms with Howe's squiggle equality [11].

Corollary 1: There is no first-order proof of $False$.
Corollary 2: There is no uniform proof of $P \vee \sim P$ in classical evidence semantics.

1.6 Properties of proof terms

We have noted that proof terms pf are evidence for the formulas they prove. It is also interesting to note that for mFOL, all the subterms of pf can be typed by subformulas of F. We call such terms *fully typed*. Moreover, for the propositional evidence terms, if we provide fully normalized inputs to functional proof terms, the value can be reduced to fully canonical form, and we can reduce the output term in any order. This is related to the strong normalization property of the typed lambda calculus with products and sums [9].

References

[1] Henk P. Barendregt. *Handbook of Logic in Computer Science*, volume 2, chapter Lambda Calculi with Types, pages 118–310. Oxford University Press, 1992.

[2] J. L. Bates. *A Logic for Correct Program Development*. PhD thesis, Cornell University, 1979.

[3] Evert W. Beth. *The Foundations of Mathematics*. North-Holland, Amsterdam, 1959.

[4] Alan Bundy, Frank van Harmelen, Christian Horn, and Alan Smaill. The oyster-clam system. In *Proceedings of the 10th International Conference on Automated Deduction*, pages 647–648, London, UK, 1990. Springer-Verlag.

[5] Alonzo Church. *The Calculi of Lambda-Conversion*, volume 6 of *Annals of Mathematical Studies*. Princeton University Press, Princeton, 1941.

[6] Robert L. Constable. The semantics of evidence (also appeared as Assigning Meaning to Proofs). *Constructive Methods of Computing Science*, F55:63–91, 1989.

[7] Robert L. Constable, Stuart F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, Douglas J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, James T. Sasaki, and Scott F. Smith. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, NJ, 1986.

[8] Haskell B. Curry. Functionality in combinatory logic. *Proc National Acad of Science*, 20:584 – 590, 1934.

[9] Daniel J. Dougherty. Some lambda calculi with categorical sums and products. In *Proc. 5th International Conference on Rewriting Techniques and Applications (RTA)*, volume 690 of *Lecture Notes in Computer Science*, pages 137–151, Berlin, 1993. Springer-Verlag.

[10] T. G. Griffin. *Notational Definition and Top-Down Refinement for Interactive Proof Development Systems*. PhD thesis, Cornell University, 1988.

[11] Douglas J. Howe. Equality in lazy computation systems. In *Proceedings of the 4^{th} IEEE Symposium on Logic in Computer Science*, pages 198–203. IEEE Computer Society Press, June 1989.

[12] Christoph Kreitz. The Nuprl Proof Development System, version 5, Reference Manual and User's Guide. Cornell University, Ithaca, NY, 2002.

[13] Per Martin-Löf. Constructive mathematics and computer programming. In *Proceedings of the Sixth International Congress for Logic, Methodology, and Philosophy of Science*, pages 153–175, Amsterdam, 1982. North Holland.

[14] R. M. Smullyan. *First–Order Logic*. Springer-Verlag, also Dover, New York, 1995, New York, 1968.