Intuitionistic First-Order Logic Refinement Rules

Robert Constable

May 18, 2018

Abstract

This is a list of the proof rules used in the completeness proof for intuitionistic First-Order Logic (iFOL) with respect to uniform validity referenced in the brief introduction. References are kept to a minimum in this list of rules.

1 Introduction

These iFOL proof rules are not entirely novel, but the refinement style proof system implemented in proof assistants might not be familiar to all readers. We provide published references for them. A widely cited source is the book on the Nuprl proof assistant [2]. The rules used here are from [1].

The semantic tradition is grounded in precise knowledge of the underlying computation system and its efficient implementation made rigorous by researchers in programming languages. Our operational semantics of evidence terms follows the method of *structured operational semantics*.

2 Proof Rules and Proof Expressions

We can assign meaning to proofs, 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* as used in Constructive Type Theory (CTT) [2].

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 then on the algorithmic bottom up pass once the downward pass is complete. Here is a simple proof of the intuitionistic tautology $A \Rightarrow (B \Rightarrow A)$.

$$\begin{array}{l} \vdash A \Rightarrow (B \Rightarrow A) \ by\ \lambda(x.slot_1(x)) \\ \quad x : A \vdash (B \Rightarrow A) \ by\ slot_1(x) \end{array}$$

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

$$\begin{array}{l} \vdash A \Rightarrow (B \Rightarrow A) \ by\ \lambda(x.slot_1(x)) \\ \quad\quad x : A \vdash (B \Rightarrow A) \ by\ \lambda(y.slot_2(x,y))\ for\ slot_1(x) \\ \quad\quad x : A, y : B \vdash A \ by\ slot_2(x,y) \end{array}$$

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

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 *ImpIntro* or *ImpR*; for $\vee$ we say *OrIn-r* or *OrIn-l*, and for $\forall x$ we say *AllIntro* 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 supplies 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* [2].

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, a 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.

$$H, f : A \Rightarrow B, H' \vdash G \text{ by } apseq(f;slot_a;v.slot_g(v))$$
$$H, f : A \Rightarrow B, v : B, H' \vdash G \text{ by } slot_g(v)$$
$$H, f : A \Rightarrow B, H' \vdash A \text{ by } slot_a$$

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.

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

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

Minimal Logic

Construction rules

- **And Construction**
  $H \vdash A\&B \text{ by } pair(slot_a;slot_b)$
  $H \vdash A \text{ by } slot_a$
  $H \vdash B \text{ by } slot_b$

- **Exists Construction**[a]
  $H \vdash \exists x.B(x) \text{ by } pair(d;slot_b(d))$
  $H \vdash d \in D \text{ by } obj(d)$
  $H \vdash B(d) \text{ by } slot_b(d)$

[a] Also see Alternative Rules below.

- **Implication Construction**
  $H \vdash A \Rightarrow B \text{ by } \lambda(x.slot_b(x)) \text{ new } x$
  $H,\ x : A \vdash B \text{ by } slot_b(x)$

- **All Construction**
  $H \vdash \forall x.B(x) \text{ by } \lambda(x.slot_b(x)) \text{ new } x$
  $H,\ x : D \vdash B(x) \text{ by } slot_b(x)$

- **Or Construction - left**
  $H \vdash A \vee B \text{ by } inl(slot_l)$
  $H \vdash A \text{ by } slot_l$

- **Or Construction - right**
  $H \vdash A \vee B \text{ by } inr(slot_r)$
  $H \vdash B \text{ by } slot_r$

Decomposition rules

- **And Decomposition**

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

- **Exists Decomposition**

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

- **Implication Decomposition**

  $H, f : A \Rightarrow B, H' \vdash G \text{ by } apseq(f;slot_a;v.slot_g[ap(f;slot_a)/v]) \text{ new } v$[1]

[1] 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.

$H, f : A \Rightarrow B, H' \vdash A$ *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* $slot_g(v)$[2]

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

- **Hypothesis - formula (A)**
  $H, x : A, H' \vdash \ A$ *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, called "ex falso quodlibet". 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 is especially important to be clear about the difference, so we use both notations.

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.

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.

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

Alterative Rule

- **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*.

2.2 Computation rules

Each of the rule forms when completely filled in becomes a term in an applied lambda calculus, 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* [2]

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.

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 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.[3]. 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.

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, as is
$\lambda(x.decide(inr(x); l.x; r.x))$[4].

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

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

References

[1] Robert Constable and Mark Bickford. Intuitionistic Completeness of First-Order Logic. *Annals of Pure and Applied Logic*, 165(1):164–198, January 2014.

[2] 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.