Proof Rules for intuitionistic First-Order Logic in Refinement Style

Robert L. Constable

September 15, 2014

1 Introduction

These notes present complete proof rules for intuitionistic First-Order Logic (iFOL) in the style of refinement rules pioneered in the LambdaPRL [3] and Nuprl [6] proof assistants and based on the style adopted by Bates [2] in his PhD thesis and the style used by Edinburgh LCF tactics [10] and in Thompson's book on type theory [16].[1]

Our presentation of the syntax of proofs will simultaneously provide enough of the syntax of types and extracts (programs and data) that it would be redundant to explain them separately. This is not the orthodox way of going about things, but in the context of other course material or books, it is efficient.

2 Proof Formats and Proof Expressions

2.1 Proof expressions and evidence

We assign meaning to proofs, according to the "proofs as terms" idea adopted in Automath [8]; however, for us it is not the proofs themselves that have

[1]The LCF proof rules are in the Gentzen style and generate trees grown in the standard way with the root at the bottom; these are the same rules that were proved to be complete with respect to *uniform validity* in the article "Intuitionistic Completeness of First-Order Logic" [5].The Thompson book, *Type Theory and Functional Programming*, is now available free on the web.

direct mathematical meaning. The proofs provide a systematic way to construct data and functions which have direct mathematical meaning. These mathematical objects are *extracted* from the proof object as we will explain. This extraction of meaning is especially natural in *refinement style logics* studied by Bates [2] and Griffin [11], motivated in part by Wirth's notion of step-wise refinement for developing programs [17] and used in Nuprl [6] and *tableaux systems* [4, 15, 9].[2] [3]

For each rule we provide a rule name that will also be the name of a constructor for data or for a functional program or it will be the name of an operation to select a component of the data or to apply the function to an argument. The full item of data or the completely defined function is constructed top down with the proof. This is an efficient way of defining evidence and showing how it is typed and constructed. The rule names will have slots to be filled in as the refinement style proof is developed. These slots will be filled in by further pieces of data or further subexpressions of the functional program. Thus the slots are replaced as the proof is constructed, so we do not give them meaning independent of this context. The object with rule names and slots is called the *partial justification* for the proof step at which they are used. If there are no slots in the expression, then it is complete.

The partial proofs are organized as a tree generated *top down*. The top down creation is driven by a user (person or machine) creating justifications with *slots* in them. To recapitulate, the slots are pieces of the justification yet to be filled in as the *proof attempt* progresses.

Once the proof is finished, i.e. the proof attempt succeeds, all of the slots can be filled in on a bottom up pass. The result is that the top level justification is an *element* for the type that we are proving to be *inhabited* or *evidence* for proposition we are attempting to *justify* or a *solution* to the problem we are tying to *solve*.

The presence of slots indicates that the proof (justification, element, evidence, solution) is incomplete, containing bits yet to be filled in. Filling in the slots creates a complete proof term (justification, element, evidence, solution). The proof process is also providing types for all the subterms of

[2]For imperative programs with assertions integrated into the code, *natural deduction* [13] is quite "'natural." In that context, programs look a lot like proofs as is evident from work on programming logics [7].

[3]The Nuprl book is also freely available on line at www.nuprl.org.

the complete justification. The full justification carries the *computational content* of the proof.

It is easy to keep track of the partial proof term as it is being assembled step by step. That is done by carrying out the replacements of slots by the evidence term built up along any branch in the proof tree being constructed. But that is not an easy process to carry out by hand because we are repeating the bits of syntax over and over. When working "by hand" it is easier to draw arrows from the new partial justifications to their parents in the tree, thus we don't repeat the bits already constructed.

The completed evidence term is also referred to as the *extract* of the proof. At each step of constructing the proof top down and along any branch, we always have a *partial extract*. The complete extract is accessible as data or as a program. The extract is referenced as $ext(thm_i)$ where thm_i is the name assigned to the proposition (type, problem) being proved (inhabited, solved). It is good practice to realize that when we are building a proof, we are also assembling its extract. So the construction process is building a pair consisting of a *a partial proof tree* and its associated *extract*, say $< pt, ext >$. When all branches of the proof tree are completed, we have a *complete proof tree* from which we can build a completed extract.

Atomic propositions All of the examples we will examine are about what a computer scientist would call *polymorphic* propositions or types. Instead of examining specific concrete atomic types such as the unit type, $Unit$, or the empty type, $Void$ (which is also the atomic proposition $False$ in our type theory), or the type of Boolean truth values, $\mathbb{B}$, or the alphabetical characters of English, $Symbols$ or the natural numbers, $\mathbb{N}$, we will use capital letters such as $A, B, C, ...$ to stand for any (all, poly) types or propositions. These will include structured atomic propositions such as $0 = 0 \ in \ \mathbb{N}$. We could even imagine that our examples would apply to propositions used in everyday conversation, e.g. "my laptop crashed," or "the internet is down for an hour." The natural language case is beyond our normal scope, however the book by Ranta, *Type Theoretical Grammar* [14] deals with natural language and type theory and covers such propositions.

The key assumption we make is that the letters $A, B, C, ...$ stand for any propositions or types, but we are not looking further into their structure in these examples. That is, when we examine $A \Rightarrow B$, we do not consider the

internal structure of A and B because we want our analysis to apply to any meaningful proposition we can assign to these letters. We say that the type $A \Rightarrow B$ is *polymorphic* because A and B can be any propositions.

We will also consider families of propositional functions $P(x)$ indexed by a type D, that is, x ranges over the type D called the domain of the family. We discuss this kind of family more when we start to examine the quantifier rules. The quantified expressions will be polymorphic in the domain D as well as in the propositions indexed by D.

In the approach taken here and in many textbooks on logic or programming, the account starts with atomic propositions whose internal structure we must ignore because it could be anything. We assume only that these atomic propositions can refer to any specific propositions which we know to be meaningful. Later on we will look at rules for understanding when an expression is a meaningful proposition.

An example Here is the development of evidence for an intuitionistic proposition **thm-k**: $A \Rightarrow (B \Rightarrow A)$ using the method of *top down refinement.*

$$\vdash A \Rightarrow (B \Rightarrow A) \textit{ by } \lambda(x.slot_1(x))$$

$$x : A \vdash (B \Rightarrow A) \textit{ by } slot_1(x)$$

The justification (or rule name) $\lambda(x.slot_1(x))$ is a partial extract. If we can find evidence for $slot_1(x)$, that achieves the subgoal specification, then we will have created adequate evidence for the goal $A \Rightarrow (B \Rightarrow A)$. We see that in this case, the step we have taken is essential and cannot be a mistake. That is because the required object to achieve this goal must be a function, and we are completely free to pick the name of the input variable, so we can't have done anything wrong yet. We have not constrained the solution beyond the constraints given by the structure of the goal. The name we pick for the variable provides a name for the hypothesis generated for the subgoal, namely $x : A$. There is so far no way to pick this name incorrectly. So the structure of the rule name provides all the information we need to generate a well formed subgoal.

In the next proof step, we need to elaborate the term $slot_1(x)$. We can think of this as a well formed meta-object *at the leaf of the tree.* It needs to be elaborated into a legitimate concrete object that can be a component of

an evidence term. That must be our next move in trying to find complete evidence.

We also see from the new subgoal that we need to make a move similar to the first one, but now we could make a mistake. The new subgoal is $x : A \vdash (B \Rightarrow A)$ *by* $slot_1(x)$. This is a different kind of goal. It has a context, the hypothesis list $x : A$. However, this hypothesis has no structure, so we can't do anything with it. On the other hand, it uses the variable x in a particular way, and we need to take that into account.

We need to focus on the subgoal, to know $(B \Rightarrow A)$. This is almost the same problem as the one we just solved, so we make a similar move. We propose building another function in this context where $x : A$ is available as an unspecified element of the type A. So we try the function introduction rule again, use $\lambda(x.slot_2(x))$. But this would be a mistake if we went on to generate another hypothesis $x : B$ because now we could not use the variable x to reference a specific hypothesis. So we agree that we can take almost the same step as before, but use another name for the bound variable. We use y instead of x. In addition, we need to recognize that whatever new slot we use for the body of the internal function, it might need to also reference the variable x. So the next proof step looks like this.

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

We could imagine writing $slot_1(x) := \lambda(y.slot_2(x, y))$ after the justification. This makes sense in the context because the variable name x appears among the hypotheses as $x : A$, so we are not introducing a free variable with no type associated with it. Also we see that the suggested justification has a new *bound variable* y, and that name provides an identifier to use in the new hypothesis generated by the proposed justification.

These considerations lead to the following new subgoal where the name y comes from the rule name:

$$x : A, y : B \vdash A \text{ by } slot_2(x, y).$$

Now life is simpler. The goal has no structure, so we can't decompose it further. The hypotheses have no structure, so we can't decompose them either. The best we can do is cite an hypothesis as sufficient reason to know the new subgoal. Indeed, knowing the first hypothesis, $x : A$, is precisely all we need; and if we did not have it, there could be no proof, and there

would be no evidence. So we see clearly that if our logic problem had been $A \Rightarrow (B \Rightarrow C)$, it would be unsolvable without more information about C. Notice that this is also a programming problem, and we would know that it is unsolvable.

Returning to the problem at hand, the subgoal arising from the proposed step of using $\lambda(y.slot_2(x,y))$ for $slot_1(x)$ leads to this subgoal, $x : A, y : B \vdash A \ by \ slot_2(x,y)$, and we see that we want $slot_2(x,y)$ to cite the right hypothesis by name. It should essentially just say, use x.

$$x : A, y : B \vdash A \ by \ slot_2(x,y)$$

We could imagine writing $slot_2(x,y) := x$ after the justification, and the check that this is sensible in that the slot mentions both variables x and y.

When the proof is finished, we see the slots filled in at each inference step, e.g.

$$\begin{aligned} &\vdash A \Rightarrow (B \Rightarrow A) \ by \ \lambda(x.\lambda(y.x)) \\ &\quad x : A \vdash (B \Rightarrow A) \ by \ \lambda(y.x) \\ &\quad x : A, y : B \vdash A \ by \ x \end{aligned}$$

The variable x fills $slot_2(x,y)$, and the lambda term $\lambda(y.x)$ fills $slot_1$. The complete extract is thus the lambda term $\lambda(x.\lambda(y.x))$. So the complete derivation object is the pair which can be presented this way, using **qed** as a separator between the derivation and the extract.

$$\begin{aligned} &\vdash A \Rightarrow (B \Rightarrow A) \ by \ \lambda(x.\lambda(y.x)) \\ &\quad x : A \vdash (B \Rightarrow A) \ by \ \lambda(y.x) \\ &\quad x : A, y : B \vdash A \ by \ x \ \textbf{qed} \ \ \lambda(x.(\lambda(y.x))). \end{aligned}$$

It is easy to see intuitively that the meaning of the extract term is precisely the evidence needed to show why we believe this proposition or *know an element for the type* or have a *solution* to the (programming) problem – which is also a mathematics problem. If we used *typed lambda terms*, the proof term would be $\lambda(x : A.(\lambda(y : B.x)))$. (We could make the proof expression more standard if we used a name such as *impin* (standing for implication introduction instead of λ; we prefer to use notation that makes the semantic ideas clearer than the traditional rule names in logic textbooks do.

We have access to this extract, using ext(**thm-k**), which in this case is a computable function.

Rule formats The rules are given in a top down style showing the *construction rules* first, often called the *introduction rules* because they introduce the canonical proof terms. They are also called the *right hand side* rules since they apply to terms on the right hand side of the turnstile, the goal side. Typical names from the literature are these: for & say *AndIntro* or *AndR*; for $\Rightarrow$ say *ImpIntro* or *ImpR*; for $\vee$ say *OrIn-r* or *OrIn-l*, and for $\forall x$ say *AllIntro* or *AllR*, and for $\exists$ say *ExistsIntro* or *ExistsR*.

For construction rules, the constructor requires subterms which provide the component pieces of evidence. Thus for *AndR* the complete 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 evolves. 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.* Thus we see rule names such as $ImpR(x.slot(x))$ or $AllR(x.slot(x))$. The rule for $\exists$, is subtle in that the rule name provides two slots, but the second depends on the object built for the first, so we use names such as $ExistsR(a; slot(a))$ indicating the dependence of the second slot on the value provided for the first one.

For each connective and operator there are also 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 determine how to use evidence that was built with the corresponding construction rules. 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.* Typical rule names are: for & the name $AndElim(x)$ or $AndL(x)$. There must be more to this rule name because typically the rule application adds new formulas to the hypothesis list, one for each of the conjuncts, so we need to provide labels for these formulas. Thus the form of elimination rule 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 like $AndR$, $AndL$, OrR_l, OrR_r, OrL, and so forth suggest features of the proof system, but they are not suggestive of the structure of the evidence being created. We prefer rule names that suggest the computational forms of evidence. The *evaluation rules* for these proof terms are given

as in constructive type theory, for instance in the book *Implementing Mathematics* [6] that defines Constructive Type Theory 84 (also see the Nuprl Reference Manual [12]).

So instead of the rule name $AndR(a;b)$ where a and b are the subterms built by a completed proof obtained 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 subtle to motivate and use intuitively. Since the evidence for $A \Rightarrow B$ is a function $\lambda(x.b(x))$, a user 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 subgoals 1 and 2 with conclusions A and G respectively are refined, say into proof terms $g(f, v)$ and a respectively. Therefore 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. The rule name *apseq* is a *sequencing operator* as well as an application, and after the subterms are created, we can evaluate the term further as we show below. Thus the rule is presented 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* $slot_a$

The term $apseq(f; a; v.g(v))$ evaluates to $g(ap(f;a))$ or more succinctly to $g(f(a))$. This simplification can only be done on the final bottom up pass

that creates a closed proof expression with no slots.

Semantic consistency The following rules are to be understood in light of the above explanation. They define what are called *pure proof expressions*. We can prove by induction on the structure of proofs that there is computational evidence for every provable formula. Moreover, the evidence is polymorphic (uniform). This provides a simple semantic consistency proof for iFOL and easy demonstrations that specific formulas such as $P \vee \sim P$ are not provable.

3 First-order refinement-style inference 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**

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

- **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* $slot_b(x)$

- **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[4]
  $H, f : A \Rightarrow B, H' \vdash A$ *by* $slot_a$

  $H, f : A \Rightarrow B, H', v : B \vdash G$ *by* $slot_g(v)$

[4]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.

- **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* $slot_g(v)$[5]

- **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, "ex falso quodlibet"**

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

  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.

[5] 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.

Classical Rules

- **Non-empty Domain of Discourse**

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

- **Double Negation Elimination (DNE)**

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

  $H \vdash (\sim\sim A \Rightarrow A)$ *by* **dne**(A)

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

3.1 Computation rules

Each of the rule forms when completely filled in becomes a term in an applied lambda calculus [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 is in the book *Implementing Mathematics* [6, 12].

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. We give this definition in the next section.

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.[6]. 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.

Fact 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, as is
$\lambda(x.decide(inr(x); l.x; r.x))$.[7].

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

[7]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] Henk P. Barendregt. Lambda calculi with types. In *Handbook of Logic in Computer Science*, volume 2, pages 118–310. Oxford University Press, 1992.

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

[3] J. L. Bates and Robert L. Constable. Proofs as programs. *ACM Transactions of Programming Language Systems*, 7(1):53–71, 1985.

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

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

[6] 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.

[7] Robert L. Constable and Michael J. O'Donnell. *A Programming Logic.* Winthrop, Mass., 1978.

[8] N. G. de Bruijn. The mathematical language Automath: its usage and some of its extensions. In J. P. Seldin and J. R. Hindley, editors, *Symposium on Automatic Demonstration*, volume 125 of *Lecture Notes in Mathematics*, pages 29–61. Springer-Verlag, 1970.

[9] M. Fitting. *Intuitionistic model theory and forcing.* North-Holland, Amsterdam, 1969.

[10] Michael Gordon, Robin Milner, and Christopher Wadsworth. *Edinburgh LCF: a mechanized logic of computation*, volume 78 of *Lecture Notes in Computer Science.* Springer-Verlag, NY, 1979.

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

[12] Christoph Kreitz. The Nuprl Proof Development System, version 5, Reference Manual and User's Guide. Cornell University, Ithaca, NY, 2002. `http://www.nuprl.org/html/02cucs-NuprlManual.pdf`.

[13] D. Prawitz. *Natural Deduction*. Dover Publications, New York, 1965.

[14] Aarne Ranta. *Type-theoretical grammar*. Oxford Science Publications. Clarendon Press, Oxford, England, 1994.

[15] R. M. Smullyan. *First–Order Logic*. Springer-Verlag, New York, 1968.

[16] Simon Thompson. *Type Theory and Functional Programming*. Addison-Wesley, 1991.

[17] Niklaus Wirth. Program development by stepwise refinement. *Communications of the ACM*, 14(4):221–227, 1971.