Linear Natural Deduction (Sketch)

Clarence Protin

April 2020

Abstract

Introduction

As Voevodsky pointed out in a public lecture in Princeton, it is highly desirable to obtain a foundations of mathematics that will allow automatic verification of proofs. This task is usually carried out in the higher-order context of lambda calculus and type theory with the bonus of the constructive computational information furnished by the Curry-Howard isomorphism: contemporary examples are the promising formal mathematics projects employing Coq, Mizar, Isabelle and specially various approaches to implementing Voevodsky's Homotopy Type Theory such as Cubicaltt. PyLog is a computationally and philosophically alternative approach which aims at fulfilling a number of desiderata:

1. The formal environment must be easy, simple, intuitive and attractive to use for the average logician or mathematician.
2. The process of writing proofs (and the checking algorithm) should resemble as closely as possible the structure of actual mathematical proofs and their process of elaboration.
3. It should be first-order with all its type-free simplicity and versatility and yet not be necessarily based on ZFC.
4. It should be easy to combine different formalised theories.
5. It should be easy to write formalised proofs such that they can be easily checked either by a human or a machine; we should obtain a "perfect bridge" between programming and mathematics.
6. The difficulty of formalising and checking a given theory should not exceed mathematical difficulty of the theory involved.
7. Classical logic is to be seen as an extension of intuitionistic logic and the user is free to use the classical negation rule or not.

The key ingredients that I propose are:

- A linearised natural deduction for the predicate calculus extended with a Kelley-Morse style extension operator.
- A first-order approach to a "reverse mathematics" style formalisation of interesting fragments of mathematics.

Linear Natural Deduction

We consider the language of first-order predicate calculus, distinguishing between variables and parameters as in Prawitz. We shall also add an extension operator and polymorphic (propositional) variables.

A *proof-entry* is a triple (n, l, p) where n is a rule name-tag, l is a finite sequence (ordered set) of positive integers and p a finite sequence of formulas,terms or positive integers.

A *proof-element* is 5-tupe $p = (n, dep, par, f, dis)$ where n is the name-tag, dep is a finite set of positive integers called the *dependencies*, par the set of parameters, finite sequences of formulas,terms or (finite sequences of positive integers), f a formula and dis is a finite set of positive integers called the *discharges* of the proof-element..

Given a proof-element $p = (Hyp, dep, par, f, dis)$ then

$$D_k(p) = (Hyp, dep, par, f, dis \cup \{k\})$$

.

A *proof* is a finite sequence $p_1, ..., p_n$ of proof-elements such that the dependencies of each element p_k are strickly less than k and the discharges larger than k and at most n.

Given a proof and and a proof-element p_i we defined recursively the *dependencies*f $dep(p_i)$ in the obvious manner: we back-track and collapse the tree based on dep to obtain a proof-element list D. Then we eliminate from D all elements whose discharge components contain an element in D.

A proof is called *final* if the dependencies of the last element p_n is empty.

A *proof-system* is a set of proof-entries R, a finite ordered set of axioms Ax, Th of theorems, Def of definitions and a partial rule that takes as input a proof $p_1, ..., p_n$ and proof-entry (n, l, p) of R (where the elements of l are not greater than n) and returns a new proof of length $n+1$, $p'_1, ..., p'_n, p'_{n+1}$. Ax and Th consists of sentences and Def consists of pairs of terms or pairs of formulas. Given a proof-element p_i we denote by f_i its formula component and n_i its name-tag.

In the following [] denotes the empty (ordered) set and $left, right, up, leftc, rightc, dis, abs$ denote positive integers, f a formula, t a term, x a variable and $places$ a finite sequence of (distinct) positive integers.

The set R we shall consider in this paper contains proof-entries of the form:

Basic:

(Hyp, [], [f]),
(AndInt, (l, r), []), (AndElimL, (up), []), (AndElimR, (up), []) ,
(OrIntL, (up), (f)) , (OrIntR, (up), (f)), (OrElim, (up, lc, rc, d_l, d_r), []),
(ImpElim, $(left, right)$, []), (ImpInt, (up, dis), []),
(AbsI, (up), (f)), (AbsC, (abs, up), []),
(ForallElim, (up), (t)), (ForallInt, (up), (t, x)), .
(ExistsInt, (up), $(t, x, places)$), (ExistElim$E$, $(left, right, up)$, (t)),

Equality:

(Repl, (up, eq), $(places)$), (Refl, (), (t)),

Also we will consider the derivative Sym rule.
Axioms, Theorems and Definitions:

(AxInt, (), (n)), (ThInt, (), (n)), (DefInt, (), (n)),

Parameters and Polymorphism:

$(\text{Sub}, (up), (pred, f)), (\text{PolySub}, (up), (polyvar, f)),$

We now describe the partial operation on proofs. Whenever we write $D_{n+1}(p_k)$ is it to be understood that we are incorporating the condition $n_k = Hyp$.

Basic:

$$(\text{Hyp}, [], [f])((p_1, ..., p_n)) = (p_1, ..., p_n, (\text{Hyp}, [], [f], f, []))$$
$$(\text{AndInt}, (l, r), [])(p_1, ..., p_n) = (p_1, ..., p_n, (\text{AndInt}, (l, r), [], f_l \& f_r, []))$$
$$(\text{AndElimL}, (up), [])(p_1, ..., p_n) = (p_1, ..., p_n, (\text{AndElimL}, (up), [], f_l, []))$$

when $f_{up} = f_l \& f_r$

$$(\text{AndElimR}, (up), [])(p_1, ..., p_n) = (p_1, ..., p_n, (\text{AndElimR}, (up), [], f_r, []))$$

when $f_{up} = f_l \& f_r$.

$$(\text{OrIntL}, (up), [f])(p_1, ..., p_n) = (p_1, ..., p_n, (\text{OrIntL}, (up), [f], f \vee f_{up}, []))$$
$$(\text{OrIntR}, (up), f])(p_1, ..., p_n) = (p_1, ..., p_n, (\text{OrIntR}, (up), [f], f_{up} \vee f, []))$$

$$(\text{OrElim}, (up, lc, rc, d_l, d_r), [])(p_1, ..., p_n) =$$
$$(p_1, ..., D_{n+1}(p_{d_l}), ..., D_{n+1}(p_{d_r}), ..., p_n, (\text{OrElim}, (up, lc, rc, d_l, d_r), [], f_{lc}, []))$$

where the formula f_{lc} of p_{up} must be the conjunction of the formulas in p_{d_l} and p_{d_r}, l be in $dep(p_{d_l})$ and r in $dep(p_{d_r})$.

$$(\text{ImpElim}, (l, r), [])(p_1, ..., p_n) = (p_1, ..., p_n, (\text{ImpElim}, (l, r), [], f, []))$$

when $f_r = f_l \to f$.

$$(\text{ImpInt}, (up, dis), [])(p_1, ..., p_n) = (p_1, ..., D_{n+1}(p_{dis}), ..., p_n, (\text{ImpInt}, (up, dis), [], f_{dis} \to f_{up}, []))$$

when $dis \in dep(p_{up})$.

$$(\text{AbsI}, [up], [f])((p_1, ..., p_n)) = (p_1, ..., p_n, (\text{AbsI}, [up], [f], f, []))$$

if $f_{up} = \bot$.

$$(\text{AbsC}, [abs, up], [])((p_1, ..., p_n)) = (p_1, ..D_{n+1}(p_{up}), .., p_n, (\text{AbsC}, [abs, up], [], f, []))$$

when $f_{abs} = \bot$ and $f_{up} = f \to \bot$.

$$(\text{ForallElim}, (up), [t])(p_1, ..., p_n) = (p_1, ..., p_n, (\text{ForallElim}, (up), [t], f[t/x], []))$$

when $f_{up} = \forall x.f$.

$$(\text{ForallInt}, (up), [t, x])(p_1, ..., p_n) = (p_1, ..., p_n, (\text{ForallInt}, (up), [t, x], \forall x.f[x/t], []))$$

when t does not occur in f_d for $d \in dep(p_{up})$.

$$(\text{ExistsInt}, (up), [t, x, places])(p_1, ..., p_n) = (p_1, ..., p_n, (\text{ExistsInt}, (up), [t, x, places], \exists x.f', []))$$

where f' results from f_{up} be replacing t with x in the positions of $places$.

$$(\text{ExistsElim}, (l, r, up), [t])(p_1, ..., p_n) = (p_1, ..., D_{n+1}(p_{up}), ..., p_n, (\text{ExistsElim}, (l, r, up), [t], f', []))$$

when f_l is of the form $\exists x.A$, $up \in dep(p_r)$ and $f_{up} = A[t/x]$

Equality:

$$(\text{Repl}, (up, eq), [places])(p_1, ..., p_n) = (p_1, ..., p_n, (\text{Repl}, (up, eq), [places], f', []))$$

where f_{eq} is of the form $t = t'$ and f' is the result of substituting t' for t in f_{up} in the positions in $places$.

$$(\text{Refl}, (), [t])(p_1, ..., p_n) = (p_1, ..., p_n, (\text{Refl}, (), [t], t = t, []))$$

Axioms, Theorems and Definitions:

$$(\text{AxInt}, [], [n])((p_1, ..., p_n)) = (p_1, ..., p_n, (\text{AxInt}, [], [n], f, []))$$

where f is the nth formula of Ax.

$$(\text{ThInt}, [], [n])((p_1, ..., p_n)) = (p_1, ..., p_n, (\text{ThInt}, [], [n], f, []))$$

where f is the nth formula of Th.

$$(\text{DefInt}, [], [n])((p_1, ..., p_n)) = (p_1, ..., p_n, (\text{DefInt}, [], [n], f, []))$$

where f is the nth formula in Def.
Parameters and Polymorphism:

$$(\text{Sub}, (up), [pred, f])((p_1, ..., p_n)) = (p_1, ..., p_n, (\text{Sub}, (up), [pref, f], f', []))$$

where $pred$ and $f(x_1, ..., x_n)$ have the same number of parameters (free variables) and f' is obtained from f_{up} by taking all predicate parameters of the form $pred(t_1, ..., t_n)$ and replacing them with $f(x_1, ...x_n)[t_1, ..., t_n/x_1, ..., x_n]$.

$$(\text{PolySub}, (up), [polyvar, f])((p_1, ..., p_n)) = (p_1, ..., p_n, (\text{PolySub}, (up), [polyvar, f], f', []))$$

where $polyvar$ is a polymorphic variable and f' results from f_{up} by replacing each occurrence of $polyvar$ with f.

We denote the resulting system by $\mathcal{L}P_c$ and write $\vdash_{\mathcal{L}P_c} f$ if there is a a sequence of proof-entries that applied in order to the empty proof yields a final proof having the last proof-entry p_m with $f_m = f$. Let N_c denote the standard natural deduction calculus. And likewise we use the same notation for the intuitionstic and minimal fragments i and m. Clearly Sub and $PolySub$ do not add to the proof-theoretic power. Consider also equality.

Lemma 0.1 *For any formula f we have $\vdash_{\mathcal{L}P_x} f$ iff $\vdash_{N_x} f$ for x equal to c,i or m.*

Proof. Given a proof in $\mathcal{L}P_c$ we construct the dependence grahp in the obvious way. It need not be a tree because a leaf u can have more than one connections. We split such leaves into clones $u', u'', ...$ for each connection. Note that each clone cannot have been discharged more than once, as each dischardged involves a connection. Thus we obtain a natural deduction tree. On the other hand, given a natural deduction tree, we coalesce all leaves which are the same formula and apply a numbering and transformation in the expected way.

The upper-bounds for the linearisation is the size of the natural deduction tree. On the other hand, given a proof of length n the tree will have at most 2^n nodes.

Examples

As usual we write $\sim A$ for $A \to \bot$ and $A \leftrightarrow B$ for $(A \to B)\&(B \to A)$. Our notation is as follows. We omit all commas in the the proof-entries and omit parameters which can be deduced immediately (this applies to Hyp, OrIntL, OrIntR). The (optional) numbers after Hyp are the the proof-entries which discharge it. These are added as we go along. The symbol QED means that the proof-entry has no dependencies.

1. $\sim\sim X$ Hyp
2. $\sim X$ Hyp
3. $\bot$ ImpElim 2 1
4. X AbsC 3 2
5. $\sim\sim X \supset X$ ImpInt 4 1 QED
6. X Hyp
7. $\sim X$ Hyp
8. $\bot$ ImpElim 6 7
9. $\sim\sim X$ ImpInt 8 7
10. $X \supset \sim\sim X$ ImpInt 9 6 QED
11. $A \supset B$ Hyp
12. $\sim B$ Hyp
13. A Hyp
14. B ImpElim 13 11
15. $\bot$ ImpElim 14 12
16. $\sim A$ ImpInt 15 13
17. $\sim B \supset \sim A$ ImpInt 16 12
18. $(A \supset B) \supset (\sim B \supset \sim A)$ ImpInt 17 11 QED
19. $\sim A\&A$ Hyp
20. A AndElimR 19
21. $\sim A$ AndElimL 19
22. $\bot$ IntElim 20 21
23. $\sim(\sim A\&A)$ ImpInt 22 19 QED
24. $\sim(A \vee \sim A)$ Hyp
25. A Hyp
26. $A \vee \sim A$ OrIntL 25
27. $\bot$ ImpElim 26 24
28. $\sim A$ ImpInt 27 25
29. $\sim A$ Hyp
30. $A \vee \sim A$ OrIntR 29
31. $\bot$ ImpElim 30 24
32. $\sim\sim A$ ImpInt 31 29

33. $\sim\sim A \supset A$ PolySub X A
34. A ImpElim 32 33
35. $A\&\sim A$ AndInt 24 28
36. $\sim(A \vee \sim A) \supset (A\&\sim A)$ ImpInt 35 24 QED
37. $(\sim(A \vee \sim A) \supset B) \supset (\sim B \supset \sim\sim(A \vee \sim A))$ PolySub 18 A $\sim(A \vee \sim A)$
38. $(\sim(A \vee \sim A) \supset (A\&\sim A)) \supset (\sim(A\&\sim A) \supset \sim\sim(A \vee \sim A))$ PolySub 18 B $(A\&\sim A)$
39. $\sim(A\&\sim A) \supset \sim\sim(A \vee \sim A)$ ImpElim 36 38
40. $\sim\sim(A \vee \sim A)$ ImpElim 23 39
41. $\sim\sim(A \vee \sim A) \supset (A \vee \sim A)$ Polysub 5 X $A \vee \sim A$
42. $A \vee \sim A$ ImpElim 40 41 QED

1. $\exists x.\sim P(x)$ Hyp
2. $\sim P(a)$ Hyp
3. $\forall x.P(x)$ Hyp
4. $P(a)$ ForallElim 3 a
5. $\perp$ ImpElim 4 2
6. $\sim\forall x.P(x)$ ImpInt 5 3
7. $\sim\forall x.P(x)$ ExistsElim 1 6 2 a
8. $\exists x.\sim P(x) \supset \sim\forall x.P(x)$ ImpInt 7 1 QED

Kelley-Morse Set Theory

In order to formalise mathematics we propose that we use *Kelley-Morse Set Theory* or Tarski-Grothendieck Set Theory (as in the Mizar system). One reason is that convenience of the extension operator both for introducing new definitions and for the associated rule. A major goal is to formalise category theory; once this achieved we can formalise easily those parts of mathematics that can be given a category-theoretic formulation. Using the intuitionistic system of natural deduction wherein the classical rule is added on will be of great advantage and theoretical interest once category theory is formalised, for category theory seems to use strictly minimal logic (intuitionistic logic without the absurdity rule). But for set theoretic foundations it is not so convenient.

We now extend our language with an extension operator yielding terms $\{x : \phi(x, u)\}$ for each formula ϕ (here u represents the other free variables in ϕ) in the usual way and a distinguished binary predicate ϵ. We define

$$Set(x) \equiv \exists y.x \in y$$

We extend $\mathcal{L}P_c$ with two new rules $(ClassInt, (up, set), (x, places))$ and $(ClassElim, (up), [])$. This two rules act as follows:

$$(\text{ClassElim}, (up), [])((p_1, ..., p_n)) = (p_1, ..., p_n, (\text{ClassElim}, (up), [], f, []))$$

when f_{up} is of the form $x \in \{y : \phi(y, u)\}$ and $f = \phi(y, u)[x/y]$.

$$(\text{ClassInt}, (up, set), [])((p_1, ..., p_n)) = (p_1, ..., p_n, (\text{ClassInt}, (up, set), [], f, []))$$

where f is $Set(y)\&y \in \{x : f'\}$, where f' results from f_{up} by substituting x for y in the positions *places*, when f_{set} is $Set(y)$. Consider also the conjunction version.

We refer the reader to the PyLog Manual for numerous examples of theorems in Kelley-Morse Set Theory.

PyLog

https://github.com/owl77/NaturalDeductionProofAssistant