Formalizing natural deduction with assumption bases

Konstantine Arkoudas

MIT Computer Science and AI Lab
arkoudas@csail.mit.edu

Abstract. We introduce $\mathcal{NDL}$, a new formalization of Fitch-style natural deduction based on the abstraction of *assumption bases*. $\mathcal{NDL}$ proofs tend to be readable, writable, concise, and can be checked very efficiently. We present the abstract syntax of $\mathcal{NDL}$, followed by two different kinds of formal semantics: evaluation and denotational. We then relate the two semantics and show how proof checking becomes tantamount to evaluation. We present sample $\mathcal{NDL}$ proofs of some well-known theorems of propositional logic, and we also develop the proof theory of the language, formulating and studying a rigorous notion of observational equivalence for proofs. We compare our approach to formalizations of natural deduction based on the Curry-Howard isomorphism and to other systems.

1 Introduction

Proof representation and checking is a central problem in the effort to mechanize mathematical knowledge. Recent Computer Science applications such as proof-carrying code (PCC [23]) have sparked renewed interest in the subject. Much work has been done in this field. A lot of it has derived inspiration from higher-order type theory, as exemplified by logical frameworks such as Automath [7], the Calculus of Constructions [11, 10], LF [17], Nuprl [8], etc., but there have been systems based on other approaches as well, e.g., Mizar [30], Ontic [22], and others. In this paper we present an alternative approach to proof representation and checking, not based on the λ-calculus and type theory, but rather on a novel variable-free syntax and on the semantic notion of *assumption bases*.

Specifically, we introduce $\mathcal{NDL}$, a language that formalizes Fitch-style[1] natural deduction [25]. Fitch-style systems are the most popular pedagogical choice for teaching symbolic logic, used by numerous influential logic textbooks [20, 27, 13, 9, 5, 21, 6]. They are considered to be the most "natural" of the three main families of proof systems that claim to capture the way in which mathematicians present proofs in practice, the other two being the natural-deduction trees deriving from Gentzen's N calculus [31] and sequent-based systems originating in

[1] This style of deduction was pioneered by the Polish logician Jáskowski circa 1930, not by Fitch. But Fitch streamlined the approach, and it is now standard practice in the literature to speak of "Fitch systems."

Gentzen's L calculus (used in books such as "Mathematical Logic" by Ebbinghaus et al. [12] and in theorem-proving systems such as HOL [16] and Isabelle [16]). Proof readability and writability are seriously compromised both in tree-based and in sequent-based systems, and it is questionable to what extent such systems may be said to reflect ordinary mathematical reasoning.

Our formalization leverages several important advances in programming language theory. In particular, $\mathcal{NDL}$ proofs are succinctly specified by a precise abstract grammar [28], while a formal evaluation semantics in the style of Kahn [19] and Plotkin [26] attaches a rigorous meaning to every proof that is syntactically well-formed. Assumption scope is captured by context-free block structure. Semantically, the formal meaning of a $\mathcal{NDL}$ proof is specified relative to a given *assumption base*, which is a set of premises, i.e., a set of propositions that we take for granted for the purposes of the proof. The key idea is that if a $\mathcal{NDL}$ proof is sound, then its meaning (denotation) is the conclusion established by the proof; if the proof is unsound, then its meaning is an error token. To obtain the meaning, we *evaluate* the proof in accordance with the formal semantics of the language. Evaluation will either produce the advertised conclusion, which will verify that the proof is sound, or else it will generate an error, which will indicate that the proof is unsound. Therefore, proof checking becomes tantamount to evaluation.

Apart from clarity and ease of presentation, defining Fitch-style natural deduction in this manner has two additional advantages. First, standard programming language implementation techniques become available for the purpose of mechanizing proofs. With a formal syntax, parsing tools can be used to read input proofs; and with a formal semantics, one can readily build an interpreter that evaluates proofs. As discussed above, such an interpreter would be a proof checker. In the case of $\mathcal{NDL}$, an interpreter has been written in one page of SML code, resulting in a very small trusted computing base [3]. Second, a formal semantics allows us to develop a rigorous theory of observational equivalence for proofs, providing precise answers to questions such as: What does it mean for two proofs to be equivalent? When can one proof be substituted for another, i.e., under what conditions can one proof be "plugged in" inside another proof without changing the latter's meaning? When can one proof be considered more efficient than another? What kinds of optimizations can be performed on proofs? When is it safe to carry out such optimizations? And so on.

We present the formal syntax and semantics of $\mathcal{NDL}$, as well as several examples demonstrating our thesis that proofs in this style are readable, writable, and succinct. We prove that $\mathcal{NDL}$ deductions are efficiently checkable, and show that they are amenable to rigorous analysis by developing a theory of observational proof equivalence. For purposes of exposition we confine attention to propositional (zero-order) natural deduction. This is not a real restriction, as most of the interesting issues already surface in this setting. In any event, it is already known that the approach scales; a conservative extension of $\mathcal{NDL}$ to full first-order logic—including equality—has been given elsewhere in detail (e.g. see

Chapter 4 of [1]). Other extensions to modal logics, higher-order logic, etc., have also been formulated.

$\mathcal{NDL}$ is representative of a class of languages called *denotational proof languages* (DPLs) [1] whose semantics are based on the abstraction of assumption bases. There are two types of DPLs, type-α and type-ω. $\mathcal{NDL}$ is an example of a type-α DPL. Type-α DPLs allow for lucid proof presentation and efficient proof checking, but not for proof search. Type-ω DPLs allow for search as well as simple presentation and checking, but termination is no longer guaranteed and proof checking may diverge. An example of a type-ω DPL is Athena [4], a DPL for first-order logic with sorts and polymorphism. It has been used in several projects, e.g., to implement credible compilers [29], to integrate model checking and theorem proving for relational reasoning [2], and others. Ongoing Athena projects include efforts in mathematical knowledge representation (e.g., encoding parts of ZF set theory) as well as verification efforts (e.g., verifying implementations of Unix file systems). Like $\mathcal{NDL}$, Athena uses a Fitch-style block-structured natural deduction system for representing proofs. In fact $\mathcal{NDL}$ can be viewed as the deductive kernel of Athena; every type-α Athena deduction can be desugared into a unique isomorphic $\mathcal{NDL}$ deduction and vice versa.

2 $\mathcal{NDL}$

2.1 Abstract syntax

We will use the letters P, Q, $R, \ldots$, to designate arbitrary *propositions*. Propositions are built in accordance with the following abstract grammar:

$$P ::= A \mid \mathbf{true} \mid \mathbf{false} \mid \neg P \mid P \wedge Q \mid P \vee Q \mid P \Rightarrow Q \mid P \Leftrightarrow Q$$

where A ranges over a countable set of atomic propositions ("atoms") which we need not specify in detail. The letters A, B, and C will be used as typical atoms. Parentheses and brackets will be used as necessary to disambiguate parsing. We write **Prop** for the set of all propositions generated by the above grammar.

The proofs (or "deductions") of $\mathcal{NDL}$ have the following abstract syntax:

$$D ::= \textit{Prim-Rule}\ P_1, \ldots, P_n \mid D_1; D_2 \mid \mathbf{assume}\ P\ \mathbf{in}\ D \qquad (1)$$

where *Prim-Rule* ranges over the collection of primitive inference rules shown in Figure 1, which are mostly introduction and elimination rules for the various logical connectives. The reader will notice one omission from Figure 1: an introduction rule for $\Rightarrow$. Conditional introduction has traditionally been the greatest challenge for natural deduction systems. As we will see shortly, in $\mathcal{NDL}$ conditionals are introduced via the language construct **assume**.

A deduction of the form *Prim-Rule* $P_1, \ldots, P_n$, $n \geq 0$, is called a *primitive rule application*, or simply a "rule application". We say that $P_1, \ldots, P_n$ are the *arguments* supplied to *Prim-Rule*. A deduction of the form $D_1; D_2$ is called a *composition*. Finally, deductions of the form **assume** P **in** D will be called

Prim-Rule ::=	**claim**	(reiteration)
|	**modus-ponens**	($\Rightarrow$-introduction)
|	**modus-tollens**	($\neg$-introduction)
|	**double-negation**	($\neg$-elimination)
|	**both**	($\wedge$-introduction)
|	**left-and**	($\wedge$-elimination)
|	**right-and**	($\wedge$-elimination)
|	**left-either**	($\vee$ -introduction)
|	**right-either**	($\vee$ -introduction)
|	**constructive-dilemma**	($\vee$ -elimination)
|	**equivalence**	($\Leftrightarrow$ -introduction)
|	**left-iff**	($\Leftrightarrow$ -elimination)
|	**right-iff**	($\Leftrightarrow$ -elimination)
|	**true-intro**	(**true**-introduction)
|	**absurd**	(**false**-introduction)
|	**false-elim**	(**false**-elimimation)

Fig. 1. Primitive inference rules

hypothetical or *conditional*. In a hypothetical deduction of the above form, P and D are called the *hypothesis* and the *body* of the deduction, respectively. The body represents the *scope* of the corresponding hypothesis. Compositions and hypothetical proofs are called *complex*, because they are recursively composed from smaller component proofs, while rule applications are *atomic* proofs—they have no internal structure. This distinction is reflected in the definition of $SZ(D)$, the size of a given D:

$$\begin{aligned} SZ(\textit{Prim-Rule}\ P_1, \ldots, P_n) &= 1 \\ SZ(D_1; D_2) &= SZ(D_1) + SZ(D_2) \\ SZ(\textbf{assume}\ P\ \textbf{in}\ D) &= SZ(D) + 1 \end{aligned}$$

Grammar (1) above specifies the abstract syntax of $\mathcal{NDL}$ proofs. It does not prescribe a concrete syntax for them because it is ambiguous. For instance, it is not clear whether **assume** $P \wedge Q$ **in true**; **left-and** $P \wedge Q$ is a hypothetical deduction with the composition **true**; **left-and** $P \wedge Q$ as its body, or a composition consisting of a hypothetical deduction followed by an application of **left-and**. We will use **begin-end** pairs or parentheses to resolve such ambiguities. Finally, we stipulate that the composition operator associates to the right, so that $D_1; D_2; D_3$ stands for $D_1; (D_2; D_3)$.

2.2 Evaluation semantics

The formal semantics of $\mathcal{NDL}$ are given in the style of Kahn [19] by a collection of evaluation rules that establish judgments of the form $\beta \vdash D \rightsquigarrow P$, where β is an *assumption base*. An assumption base is a finite set of propositions, say $\{\textbf{true}, A \vee B\}$. Intuitively, the elements of an assumption base are *premises*—propositions that are taken to hold for the purposes of a proof. In particular, a judgment of the form $\beta \vdash D \rightsquigarrow P$ states that "With respect to the assumption

$[R_1]$ $\beta \cup \{P\} \vdash \textbf{claim}\ P \rightsquigarrow P$
$[R_2]$ $\beta \cup \{P \Rightarrow Q, P\} \vdash \textbf{modus-ponens}\ P \Rightarrow Q, P \rightsquigarrow Q$
$[R_3]$ $\beta \cup \{P \Rightarrow Q, \neg Q\} \vdash \textbf{modus-tollens}\ P \Rightarrow Q, \neg Q \rightsquigarrow \neg P$
$[R_4]$ $\beta \cup \{\neg\neg P\} \vdash \textbf{double-negation}\ \neg\neg P \rightsquigarrow P$
$[R_5]$ $\beta \cup \{P_1, P_2\} \vdash \textbf{both}\ P_1, P_2 \rightsquigarrow P_1 \wedge P_2$
$[R_6]$ $\beta \cup \{P_1 \wedge P_2\} \vdash \textbf{left-and}\ P_1 \wedge P_2 \rightsquigarrow P_1$
$[R_7]$ $\beta \cup \{P_1 \wedge P_2\} \vdash \textbf{right-and}\ P_1 \wedge P_2 \rightsquigarrow P_2$
$[R_8]$ $\beta \cup \{P_1\} \vdash \textbf{left-either}\ P_1, P_2 \rightsquigarrow P_1 \vee P_2$
$[R_9]$ $\beta \cup \{P_2\} \vdash \textbf{right-either}\ P_1, P_2 \rightsquigarrow P_1 \vee P_2$
$[R_{10}]$ $\beta \cup \{P_1 \vee P_2, P_1 \Rightarrow Q, P_2 \Rightarrow Q\} \vdash \textbf{constructive-dilemma}\ P_1 \vee P_2, P_1 \Rightarrow Q, P_2 \Rightarrow Q \rightsquigarrow Q$
$[R_{11}]$ $\beta \cup \{P_1 \Rightarrow P_2, P_2 \Rightarrow P_1\} \vdash \textbf{equivalence}\ P_1 \Rightarrow P_2, P_2 \Rightarrow P_1 \rightsquigarrow P_1 \Leftrightarrow P_2$
$[R_{12}]$ $\beta \cup \{P_1 \Leftrightarrow P_2\} \vdash \textbf{left-iff}\ P_1 \Leftrightarrow P_2 \rightsquigarrow P_1 \Rightarrow P_2$
$[R_{13}]$ $\beta \cup \{P_1 \Leftrightarrow P_2\} \vdash \textbf{right-iff}\ P_1 \Leftrightarrow P_2 \rightsquigarrow P_2 \Rightarrow P_1$
$[R_{14}]$ $\beta \cup \{P, \neg P\} \vdash \textbf{absurd}\ P, \neg P \rightsquigarrow \textbf{false}$
$[R_{15}]$ $\beta \vdash \textbf{true-intro} \rightsquigarrow \textbf{true}$
$[R_{16}]$ $\beta \vdash \textbf{false-elim} \rightsquigarrow \neg\textbf{false}$

Fig. 2. Axioms for primitive deductions.

base β, proof D derives the proposition P"; or "D evaluates to P in the context of β"; etc. We write **AB** for the set of all assumption bases of $\mathcal{NDL}$.

The evaluation rules are partitioned into two groups: those dealing with atomic proofs, i.e., with rule applications, shown in Figure 2; and those dealing with complex proofs, presented below. We will first discuss rule applications. Consider, for example, the **double-negation** rule, which eliminates pairs of negation signs. It says that if we know that $\neg\neg P$ holds, we may derive P. In our framework, "knowing that $\neg\neg P$ holds" amounts to $\neg\neg P$ being in the assumption base. Thus we arrive at the semantics of rule $[R_4]$:

$$\beta \cup \{\neg\neg P\} \vdash \textbf{double-negation}\ \neg\neg P \rightsquigarrow P$$

saying that an application of **double-negation** to a proposition of the form $\neg\neg P$ results in the conclusion P, *provided that the premise* $\neg\neg P$ *is in the assumption base*. Hence, operationally, the algorithm for *evaluating* a rule application **double-negation** $\neg\neg P$ in a given assumption base β is simple: we check to see if $\neg\neg P$ is in the assumption base; if the look-up succeeds, we output P, otherwise we report an error message (e.g., "Invalid application of **double-negation**; the premise $\neg\neg P$ is not in the assumption base.") This will become more clear in the denotational version of the semantics. The rest of the rule applications have similar interpretations.

We continue with complex deductions. The semantics of hypothetical proofs are given by the following rule:

$$\frac{\beta \cup \{P\} \vdash D \rightsquigarrow Q}{\beta \vdash \textbf{assume}\ P\ \textbf{in}\ D \rightsquigarrow P \Rightarrow Q} \quad [Assume]$$

This rule is better read backwards, with an operational interpretation: to evaluate a deduction of the form **assume** P **in** D in a given assumption base β, we

add the hypothesis P to β and proceed recursively to evaluate the body D in the augmented assumption base $\beta \cup \{P\}$. If and when that produces a conclusion Q, we return the conditional $P \Rightarrow Q$ as the result.

Finally, the semantics of compositions are given by the following rule:

$$\frac{\beta \vdash D_1 \rightsquigarrow P_1 \qquad \beta \cup \{P_1\} \vdash D_2 \rightsquigarrow P_2}{\beta \vdash D_1; D_2 \rightsquigarrow P_2} \quad [Comp]$$

This rule is also better read backwards: to evaluate $D_1; D_2$ in some assumption base β, we first evaluate D_1 in β, getting some conclusion P_1 from it, and then we evaluate D_2 in $\beta \cup \{P_1\}$, i.e., in β *augmented with* P_1. Thus the result of D_1 becomes logically available, as a lemma, within D_2. The result of D_2 is the result of the entire composition.

As an example of these formal semantics in action, let

$$D = \begin{array}{l} \textbf{assume } A \wedge B \textbf{ in} \\ \quad \textbf{begin} \\ \quad\quad \textbf{left-and } A \wedge B; \\ \quad\quad \textbf{right-and } A \wedge B; \\ \quad\quad \textbf{both } B, A \\ \quad \textbf{end} \end{array}$$

The following derivation establishes the judgment $\emptyset \vdash D \rightsquigarrow A \wedge B \Rightarrow B \wedge A$:

1. $\{A \wedge B\} \vdash \textbf{left-and } A \wedge B \rightsquigarrow A$ — $[R_6]$
2. $\{A \wedge B, A\} \vdash \textbf{right-and } A \wedge B \rightsquigarrow B$ — $[R_7]$
3. $\{A \wedge B, A, B\} \vdash \textbf{both } B, A \rightsquigarrow B \wedge A$ — $[R_5]$
4. $\{A \wedge B, A\} \vdash \textbf{right-and } A \wedge B; \textbf{both } B, A \rightsquigarrow B \wedge A$ — 2, 3, $[Comp]$
5. $\{A \wedge B\} \vdash \textbf{left-and } A \wedge B; \textbf{right-and } A \wedge B; \textbf{both } B, A \rightsquigarrow B \wedge A$ — 1, 4, $[Comp]$
6. $\emptyset \vdash D \rightsquigarrow A \wedge B \Rightarrow B \wedge A$ — 5, $[Assume]$

Our first result about this semantics guarantees conclusion uniqueness: a deduction derives at most one proposition. It can be proved directly, or as a corollary of Theorem 5 or Theorem 6 below.

Theorem 1. *If* $\beta_1 \vdash D \rightsquigarrow P_1$ *and* $\beta_2 \vdash D \rightsquigarrow P_2$ *then* $P_1 = P_2$.

We end this section by introducing some useful syntax sugar: deductions of the form

$$\textbf{suppose-absurd } P \textbf{ in } D \tag{2}$$

that perform reasoning by contradiction. A proof of the form (2) seeks to establish the negation of P by deriving a contradiction. So, operationally, the evaluation of (2) in an assumption base β proceeds as follows: we add the hypothesis P to β and evaluate the body D in the augmented assumption base. If and when D produces a contradiction (the proposition **false**), we output the negation $\neg P$. This is justified because P led us to a contradiction—hence we must have $\neg P$. More formally, the semantics of (2) are given by the following rule:

$$\frac{\beta \cup \{P\} \vdash D \rightsquigarrow \textbf{false}}{\beta \vdash \textbf{suppose-absurd } P \textbf{ in } D \rightsquigarrow \neg P} \tag{3}$$

However, we do not take proofs of the form (2) as primitive because their intended behavior is expressible in terms of **assume**, primitive inference rules, and composition. In particular, (2) is defined as an abbreviation for the deduction

assume P **in** D;
false-elim;
modus-tollens $P \Rightarrow$ **false**, $\neg$**false**

It is readily shown that this desugaring results in the intended semantics (3):

Theorem 2. *If* $\beta \cup \{P\} \vdash D \rightsquigarrow$ **false** *then* $\beta \vdash$ **suppose-absurd** P **in** $D \rightsquigarrow \neg P$.

The machinery we have so far already results in a sound and complete system:

Theorem 3. *If* $\beta \vdash D \rightsquigarrow P$ *then* $\beta \models P$. *And if* $\beta \models P$, *then* $\exists\, D\,.\, \beta \vdash D \rightsquigarrow P$.

2.3 Denotational semantics: an interpreter for $\mathcal{NDL}$

For a denotational semantics for $\mathcal{NDL}$, let *error* be an object that is distinct from all propositions. We define a meaning function

$$\mathcal{M} : \mathbf{Ded} \to \mathbf{AB} \to \mathbf{Prop} \cup \{error\}$$

that takes a deduction D followed by an assumption base β and produces an element of $\mathbf{Prop} \cup \{error\}$ that is regarded as "the meaning" of D with respect to β. The definition of $\mathcal{M}[\![D]\!]$ is driven by the structure of D. We use the notation $e_1\,? \to e_2, e_3$ to mean "if e_1 then e_2 else e_3". For hypothetical deductions and compositions we set:

$$\mathcal{M}[\![\textbf{assume } P \textbf{ in } D]\!]\,\beta = (\mathcal{M}[\![D]\!]\,\beta \cup \{P\}) = error\,? \to error,\ P \Rightarrow (\mathcal{M}[\![D]\!]\,\beta \cup \{P\})$$
$$\mathcal{M}[\![D_1; D_2]\!]\,\beta = (\mathcal{M}[\![D_1]\!]\,\beta) = error\,? \to error,\ \mathcal{M}[\![D_2]\!]\,\beta \cup (\mathcal{M}[\![D_1]\!]\,\beta).$$

The equations for rule applications are given by a case analysis of the rule. We illustrate with the equations for **modus-ponens**, **left-and**, and **both**; the rest can be obtained in a similar manner by consulting Figure 2.

$$\mathcal{M}[\![\textbf{modus-ponens } P \Rightarrow Q, P]\!]\,\beta = \{P \Rightarrow Q, P\} \subseteq \beta\,? \to Q,\ error$$
$$\mathcal{M}[\![\textbf{left-and } P \wedge Q]\!]\,\beta = P \wedge Q \in \beta\,? \to P,\ error$$
$$\mathcal{M}[\![\textbf{both } P, Q]\!]\,\beta = \{P, Q\} \subseteq \beta\,? \to P \wedge Q,\ error$$

Equivalently, we can choose to emphasize that the formal meaning (denotation) of a proof is *a function over assumption bases*, a key characteristic of DPLs, by recasting the above equations in the form shown in Figure 3.

The above equations can be readily transcribed into an algorithm that takes a deduction D and an assumption base β and either produces a proposition P or else generates an error. In fact we can look at $\mathcal{M}$ itself as an interpreter for $\mathcal{NDL}$. In that light, it is easy to see that $\mathcal{M}$ always terminates. Furthermore, assuming that looking up a proposition in an assumption base takes constant time on average (e.g., by implementing assumption bases as hash tables), $\mathcal{M}$ terminates in $O(SZ(D))$ time in the average case. The worst-case complexity is $O(n \cdot \log n)$, where $n = SZ(D)$, achieved by implementing assumption bases as balanced trees, which guarantees logarithmic look-up time in the worst case. We refer to the process of obtaining $\mathcal{M}[\![D]\!]\,\beta$ as *evaluating* D *in* β.

$$
\begin{aligned}
\mathcal{M}[\![D_1; D_2]\!] &= \lambda\beta\,.\,(\mathcal{M}[\![D_1]\!]\,\beta) = \mathit{error}\,?\,\rightarrow \mathit{error},\ \mathcal{M}[\![D_2]\!]\,\beta \cup (\mathcal{M}[\![D_1]\!]\,\beta)\\
\mathcal{M}[\![\textbf{assume } P \textbf{ in } D]\!] &= \lambda\beta\,.\,(\mathcal{M}[\![D]\!]\,\beta \cup \{P\}) = \mathit{error}\,?\,\rightarrow \mathit{error},\ P \Rightarrow (\mathcal{M}[\![D]\!]\,\beta \cup \{P\})\\
\mathcal{M}[\![\textbf{claim } P]\!] &= \lambda\beta\,.\,P \in \beta\,?\,\rightarrow P,\ \mathit{error}\\
\mathcal{M}[\![\textbf{both } P, Q]\!] &= \lambda\beta\,.\,\{P, Q\} \subseteq \beta\,?\,\rightarrow P \wedge Q,\ \mathit{error}\\
\mathcal{M}[\![\textbf{left-and } P \wedge Q]\!] &= \lambda\beta\,.\,P \wedge Q \in \beta\,?\,\rightarrow P,\ \mathit{error}\\
&\vdots
\end{aligned}
$$

Fig. 3. Alternative denotational semantics for $\mathcal{NDL}$.

Theorem 4. *Evaluating a deduction D in a given assumption base takes $O(n)$ time on average, and $O(n \cdot \log n)$ time in the worst case, where $n = SZ(D)$.*

The following result relates the denotational semantics of $\mathcal{NDL}$ to its evaluation semantics. If we view $\mathcal{M}$ as an interpreter for $\mathcal{NDL}$, the theorem below becomes a correctness result for the interpreter: it tells us that $\mathcal{M}$ will produce a result P for a given D and β iff the judgment $\beta \vdash D \rightsquigarrow P$ holds according to the evaluation semantics.

Theorem 5. *$\beta \vdash D \rightsquigarrow P$ iff $\mathcal{M}[\![D]\!]\,\beta = P$. Hence, by termination, $\mathcal{M}[\![D]\!]\,\beta = \mathit{error}$ iff there is no P such that $\beta \vdash D \rightsquigarrow P$.*

2.4 Examples

In this section we present $\mathcal{NDL}$ proofs of some well-known theorems of propositional logic.

$$\boxed{P \Rightarrow \neg\neg P}$$

Proof:

assume P **in**
  suppose-absurd $\neg P$ **in**
    absurd $P, \neg P$

$$\boxed{(P \Rightarrow Q) \Rightarrow [(Q \Rightarrow R) \Rightarrow (P \Rightarrow R)]}$$

Proof:

assume $P \Rightarrow Q$ **in**
  assume $Q \Rightarrow R$ **in**
    assume P **in**
      begin
        modus-ponens $P \Rightarrow Q, P$;
        modus-ponens $Q \Rightarrow R, Q$
      end

$$\boxed{[P \Rightarrow (Q \Rightarrow R)] \Rightarrow [(P \wedge Q) \Rightarrow R]}$$

Proof:
assume $P \Rightarrow (Q \Rightarrow R)$ **in**
assume $P \wedge Q$ **in**
begin
left-and $P \wedge Q$;
modus-ponens $P \Rightarrow (Q \Rightarrow R), P$;
right-and $P \wedge Q$;
modus-ponens $Q \Rightarrow R, Q$
end

$$\boxed{\neg(P \vee Q) \Rightarrow (\neg P \wedge \neg Q)}$$

Proof:
assume $\neg(P \vee Q)$ **in**
begin
suppose-absurd P **in**
begin
left-either P, Q;
absurd $P \vee Q, \neg(P \vee Q)$
end;
suppose-absurd Q **in**
begin
right-either P, Q;
absurd $P \vee Q, \neg(P \vee Q)$
end;
both $\neg P, \neg Q$
end

2.5 Basic proof theory

We will say that a proof is *well-formed* iff every rule application in it has one of the forms shown in Figure 2. That is, a deduction is well-formed iff the right number and kind of arguments are supplied to every application of a primitive rule, so that, for instance, there are no applications such as **modus-ponens** $A \wedge B$. Checking a deduction to make sure that it is well-formed is trivial. Hereafter we will only be concerned with well-formed proofs. The *conclusion* of a deduction D, $\mathcal{C}(D)$, is defined by structural recursion. For complex D,

$$\mathcal{C}(D_1; D_2) = \mathcal{C}(D_2)$$
$$\mathcal{C}(\textbf{assume } P \textbf{ in } D) = P \Rightarrow \mathcal{C}(D)$$

while for atomic rule applications we have:

$$FA(D_1; D_2) = FA(D_1) \cup (FA(D_2) - \{\mathcal{C}(D_1)\}) \quad (4)$$
$$FA(\textbf{assume } P \textbf{ in } D) = FA(D) - \{P\} \quad (5)$$
$$FA(\textbf{left-either } P_1, P_2) = \{P_1\} \quad (6)$$
$$FA(\textbf{right-either } P_1, P_2) = \{P_2\} \quad (7)$$
$$FA(\textit{Prim-Rule } P_1, \ldots, P_n) = \{P_1, \ldots, P_n\} \quad (8)$$

Fig. 4. The definition of $FA(D)$ ($\textit{Prim-Rule} \notin \{\textbf{left-either}, \textbf{right-either}\}$ in 8).

$$\mathcal{C}(\textbf{claim } P) = P$$
$$\mathcal{C}(\textbf{modus-ponens } P \Rightarrow Q, P) = Q$$
$$\mathcal{C}(\textbf{modus-tollens } P \Rightarrow Q, \neg Q) = \neg P$$
$$\mathcal{C}(\textbf{double-negation } \neg\neg P) = P$$
$$\mathcal{C}(\textbf{both } P, Q) = P \wedge Q$$
$$\mathcal{C}(\textbf{left-and } P \wedge Q) = P$$
$$\mathcal{C}(\textbf{right-and } P \wedge Q) = Q$$
$$\mathcal{C}(\textbf{left-either } P, Q) = P \vee Q$$
$$\mathcal{C}(\textbf{right-either } P, Q) = P \vee Q$$
$$\mathcal{C}(\textbf{cd } P_1 \vee P_2, P_1 \Rightarrow Q, P_2 \Rightarrow Q) = Q$$
$$\mathcal{C}(\textbf{equivalence } P \Rightarrow Q, Q \Rightarrow P) = P \Leftrightarrow Q$$
$$\mathcal{C}(\textbf{left-iff } P \Leftrightarrow Q) = P \Rightarrow Q$$
$$\mathcal{C}(\textbf{right-iff } P \Leftrightarrow Q) = Q \Rightarrow P$$
$$\mathcal{C}(\textbf{true-intro}) = \textbf{true}$$
$$\mathcal{C}(\textbf{absurd } P, \neg P) = \textbf{false}$$
$$\mathcal{C}(\textbf{false-elim}) = \neg\textbf{false}$$

(writing **cd** as an abbreviation for **constructive-dilemma**). This definition can be used as a recursive algorithm for computing $\mathcal{C}(D)$. Computing $\mathcal{C}(D)$ is quite different—much easier—than evaluating D in a given β. For example, if D is of the form $D_1; D_2; \ldots; D_{99}; D_{100}$, we can ignore the first 99 deductions and simply compute $\mathcal{C}(D_{100})$, since, by definition, $\mathcal{C}(D) = \mathcal{C}(D_{100})$. The catch, of course, is that D might fail to establish its conclusion (in a given assumption base); evaluation is the only way to find out. However, it is easy to show that *if* $\beta \vdash D \rightsquigarrow P$ *then* $P = \mathcal{C}(D)$. Hence, in terms of the interpreter $\mathcal{M}$, either $\mathcal{M}[\![D]\!]\,\beta = \textit{error}$ or $\mathcal{M}[\![D]\!]\,\beta = \mathcal{C}(D)$, for any D and β. Loosely paraphrased: a proof succeeds iff it produces its purported conclusion.

Theorem 6. *If* $\beta \vdash D \rightsquigarrow P$ *then* $P = \mathcal{C}(D)$. *Therefore, by Theorem 5, either* $\mathcal{M}[\![D]\!]\,\beta = \mathcal{C}(D)$ *or else* $\mathcal{M}[\![D]\!]\,\beta = \textit{error}$.

Next, we define a function $FA : \textbf{Ded} \to \mathcal{P}_\infty(\textbf{Prop})$, where $\mathcal{P}_\infty(\textbf{Prop})$ denotes the set of all finite subsets of **Prop**. We call the elements of $FA(D)$ the *free assumptions* of D. Intuitively, a free assumption of D is a proposition that D uses without proof. It is a "premise" of D—a necessary assumption that D takes for granted. The main result of this section is that D can be successfully evaluated only in an assumption base that contains its free assumptions. The definition of $FA(D)$ is given in Figure 4. Note in particular the case of hypothetical deductions **assume** P **in** D: the free assumptions here are those of the body D *minus* the hypothesis P.

The computation of $FA(D)$ is not trivial, meaning that it cannot proceed in a local manner down the abstract syntax tree of D using only a fixed amount of memory: a variable amount of state must be maintained in order to deal

with clauses 5 and 4. The latter clause, in particular, is especially computation-intensive because it also calls for the computation of $\mathcal{C}(D_1)$. The reader should reflect on what would be involved in computing, for instance, $FA(D)$ for a D of the form $D_1; \cdots ; D_{100}$. In fact the next result shows that the task of evaluating D in a given β can be reduced to the computation of $FA(D)$.

Theorem 7. *$\beta \vdash D \rightsquigarrow P$ iff $FA(D) \subseteq \beta$. Accordingly, $\mathcal{M}[\![D]\!]\,\beta = error$ iff there is a P such that $P \in FA(D)$ and $P \notin \beta$.*

The above result captures the sense in which evaluation in $\mathcal{NDL}$ can be reduced to the computation of FA: to compute $\mathcal{M}[\![D]\!]\,\beta$, for any given D and β, simply compute $FA(D)$ and $\mathcal{C}(D)$: if $FA(D) \subseteq \beta$, output $\mathcal{C}(D)$; otherwise output *error*. Intuitively, this reduction is the reason why the computation of $FA(D)$ cannot be much easier than the evaluation of D in a given assumption base.

Next, we will say that two proofs D_1 and D_2 are *observationally equivalent with respect to an assumption base* β, written $D_1 \approx_\beta D_2$, whenever

$$\beta \vdash D_1 \rightsquigarrow P \quad \text{iff} \quad \beta \vdash D_2 \rightsquigarrow P \tag{9}$$

for all P. For instance,

$$D_1 = \textbf{left-either}\ A, B \quad \text{and} \quad D_2 = \textbf{right-either}\ A, B$$

are observationally equivalent with respect to any assumption base that either contains both A and B or neither of them.

Theorem 8. *$D_1 \approx_\beta D_2$ iff $\mathcal{M}[\![D_1]\!]\,\beta = \mathcal{M}[\![D_2]\!]\,\beta$.*

Thus D_1 and D_2 are observationally equivalent in a given β iff when we evaluate them in β we obtain the same result: either the same proposition, or an error in both cases. It follows that $\approx_\beta$ is decidable.

If we have $D_1 \approx_\beta D_2$ for *all* β then we write $D_1 \approx D_2$ and say that D_1 and D_2 are *observationally equivalent*. For a simple example, we have

$$\textbf{left-iff}\ A \Leftrightarrow A \ \approx\ \textbf{right-iff}\ A \Leftrightarrow A$$

as well as

$$\begin{array}{lcl} \textbf{left-and}\ A \wedge B; & & \textbf{right-and}\ A \wedge B; \\ \textbf{right-and}\ A \wedge B; & \approx & \textbf{left-and}\ A \wedge B; \\ \textbf{both}\ B, A & & \textbf{both}\ B, A \end{array}$$

The reader will verify that $\approx$ is an equivalence relation. The next result shows that composition is not associative over observational equivalence:

Theorem 9. *$D_1; (D_2; D_3) \not\approx (D_1; D_2); D_3$.*

Proof. Take $D_1 = \textbf{double-negation}\ \neg\neg A$, $D_2 = \textbf{true}$, $D_3 = A$, and consider any β that contains $\neg\neg A$ but not A. □

We note, however, that composition is idempotent (i.e., $D \approx D; D$) and that associativity does hold in the case of claims:

$$\textbf{claim}\ P_1; (\textbf{claim}\ P_2; \textbf{claim}\ P_3) \approx (\textbf{claim}\ P_1; \textbf{claim}\ P_2); \textbf{claim}\ P_3.$$

Commutativity fails in all cases. A certain kind of distributivity holds between the constructor **assume** and the composition operator (as well as between **suppose-absurd** and composition), in the following sense:

Theorem 10. **assume** P **in** $(D_1; D_2) \approx D_1$ **assume** P **in** D_2 *if* $P \notin FA(D_1)$.

This result forms the basis for a "hoisting" transformation in the theory of $\mathcal{NDL}$ optimization. Further, we note that $\approx$ is compatible with the abstract syntax constructors of $\mathcal{NDL}$.

Theorem 11. *If* $D \approx D'$ *then* **assume** P **in** $D \approx$ **assume** P **in** D'. *In addition, if* $D_1 \approx D_1'$ *and* $D_2 \approx D_2'$ *then* $D_1; D_2 \approx D_1'; D_2'$.

At first glance it might appear that $\approx$ is undecidable, as it is defined by quantifying over all assumption bases. However, the following result shows that two proofs are observationally equivalent iff they have the same conclusion and the same free assumptions. Since both of these are computable, it follows that $\approx$ is decidable:

Theorem 12. $D_1 \approx D_2$ *iff* $\mathcal{C}(D_1) = \mathcal{C}(D_2)$ *and* $FA(D_1) = FA(D_2)$.

3 Related work

In this section we compare $\mathcal{NDL}$ to other formalizations of natural deduction, particularly in type-based frameworks.

The reader will notice a similarity between [*Assume*], our evaluation rule for hypothetical deductions, and the typing rule for abstractions in the simply typed λ-calculus:

$$\frac{\beta[x \mapsto \sigma] \vdash E : \tau}{\beta \vdash \lambda\, x : \sigma\, .\, E : \sigma \to \tau}$$

where β in this context is a type environment, i.e., a finite function from variables to types (and $\beta[x \mapsto \sigma]$ is the extension of β with the binding $x \mapsto \sigma$). Likewise, the reader will notice a similarity between [*Comp*], our rule for compositions, and the customary typing rule for **let** expressions:

$$\frac{\beta \vdash E_1 : \tau_1 \qquad \beta[x \mapsto \tau_1] \vdash E_2 : \tau_2}{\beta \vdash \textbf{let}\ x = E_1\ \textbf{in}\ E_2 : \tau_2}$$

Indeed, it is straightforward to pinpoint the correspondence between $\mathcal{NDL}$ and the simply typed λ-calculus. Deductions correspond to terms, in accordance with the "proofs-as-programs" motto [18, 14, 31]. In particular, **assume**s correspond to λs and compositions correspond to **let**s. Propositions correspond to types, e.g., $A \Rightarrow B$ corresponds to $A \to B$, $A \wedge B$ corresponds to $A \times B$, etc.

Assumption bases correspond to type environments, and evaluation in $\mathcal{NDL}$ corresponds to type checking in the λ-calculus. Finally, observational equivalence corresponds to type cohabitation in the λ-calculus, and optimization via proof transformation in $\mathcal{NDL}$ corresponds to normalization in the λ-calculus. The correspondence extends to results: conclusion uniqueness in DPLs corresponds to type uniqueness theorems in type systems, and so on.

However, in the case of $\mathcal{NDL}$ this correspondence is only a similarity, not an isomorphism. The reason is that $\mathcal{NDL}$ has no variables, and hence infinitely many distinct terms of the typed λ-calculus collapse to the same DPL proof, destroying the bijection. For instance, consider the two terms $t_1 = \lambda\, x : A\,.\,\lambda\, y : A\,.\,x$ and $t_2 = \lambda\, x : A\,.\,\lambda\, y : A\,.\,y$. These are two distinct terms (they are not alphabetically equivalent), and under the Curry-Howard isomorphism they represent two distinct proofs of $A \Rightarrow A \Rightarrow A$. To see why, one should keep in mind Heyting's constructive interpretation of conditionals [14]. In t_1, we are given a proof x of A, then we are given another proof y of it, and we finally decide to *use the first proof*, which might well have significant computational differences from the second one (e.g., one might be much more efficient than the other). By contrast, in t_2 we are given a proof of A, then another proof of it, and finally we decide to *use the second proof*, disregarding the first. But in mathematical practice these two proofs would normally be conflated, because in most cases one postulates propositions, not proofs of propositions. The customary mode of hypothetical reasoning is "Assume that A holds; then $\cdots A \cdots$", *not* "Assume we have a proof Π of A; then $\cdots \Pi \cdots$". Clearly, to assume that something is true is not the same as to assume that we have a proof of it. Constructivists might argue against such a distinction, but regardless of whether or not the distinction is philosophically justified, it is certainly common in practice. DPLs reflect mathematical practice more closely, in that both t_1 and t_2 get conflated to the single deduction

assume A **in**
  assume A **in**
    claim A

This difference reflects a divergence between DPLs and Curry-Howard systems on the foundational issue of constructivism. In a DPL such as $\mathcal{NDL}$ or Athena you only care about truth, or more narrowly, about what is in the assumption base. On the other hand, logical frameworks [17, 7] based on the Curry-Howard isomorphism put the focus on *proofs* of propositions rather than on propositions *per se*. This is due to the constructive bent of these systems, which dictates that a proposition should not be supposed to hold unless we have a proof for it. There are types, but more importantly, there are variables that have those types, and those variables range over proofs. In a DPL such as $\mathcal{NDL}$ we stop at types—there are no variables. Another way to see this is to consider the difference between assumption bases and type environments: an assumption base is simply a set of propositions (types), whereas a type environment is a set of *pairs* of variables and types.

We believe that the approach of DPLs is more parsimonious, because there is no explicit talk about proofs: an inference rule R is applied directly to propositions $P_1, \ldots, P_n$, not to proofs of propositions $P_1, \ldots, P_n$; a hypothetical deduction assumes simply that we are given a hypothesis P, not that we are given a proof of P; and so on. Shaving away all the talk about proofs results in shorter and cleaner deductions.

Another important difference lies in the representation of syntactic categories such as propositions, proofs, etc. Curry-Howard systems typically rely on higher-order abstract syntax for representing such objects. While this has some advantages (most notably, alphabetically identical expressions at the object level are represented by alphabetically identical expressions at the meta level, and object-level syntactic operations such as substitution become reduced to β-reduction at the meta level), it also has drawbacks: higher-order expressions are less readable and writable, they are considerably larger on account of the extra abstractions and the explicit presence of type information, and complicate issues such as matching, unification, and reasoning by structural induction. A more detailed discussion of these issues, along with relevant examples, can be found elsewhere [1].

Other related systems include descendants of LCF [15] such as HOL [16] and Isabelle [24]. These have been widely used for digital system verification, as well as for mechanical representation of mathematical knowledge. They are particularly important in that they represent the tactic-based approach to theorem proving, pioneered by Milner, whereby users can extend the reasoning power of the underlying framework in a sound fashion by writing arbitrary inference methods expressed in terms of primitives. However, deduction in such systems is based on sequent calculi, which are not accurate models of ordinary mathematical reasoning. Isar [34] alleviates this to a certain extent for Isabelle by allowing for more readable proofs, but it appears to have complicated semantics, dividing the proof state into a static and a dynamic context and moving information from one to the other during the course of the proof in a way that often results in uninformative error messages.

Mizar [30] is another system that is much larger and more established than $\mathcal{NDL}$ or Athena, with a substantial body of abstract mathematics successfully encoded in it. As far as proof representation is concerned, Mizar also uses a Fitch-style framework, similar to that of Isar [35], but we are not aware of any formal semantics for it (say, in denotational or operational style). The lack of a formal semantics makes it harder to subject proofs to rigorous analysis, e.g., to ask and answer questions such as "Under what conditions can this subproof be replaced by another without changing the meaning of the surrounding proof?" It is also not clear exactly what the trusted base of Mizar is. For instance, proofs in Mizar articles make use of the `by` construct, which performs a degree of automated reasoning designed to dispense with tedious steps, but it appears that the implementation of this feature is part of the trusted base of the system (i.e., it is not expressed in terms of other, simpler primitives), which may increase the likelihood of bugs.

4 Conclusions

This paper introduced $\mathcal{NDL}$, a new formalization of Fitch-style natural deduction. We demonstrated that $\mathcal{NDL}$ proofs are readable, writable, compact, and can be checked very efficiently—in time linear in the size of the proof. We introduced special syntactic forms for proofs, such as compositions and hypothetical deductions, and showed how the abstraction of assumption bases allows us to give intuitive evaluation semantics to these constructs in such a way that evaluation becomes tantamount to proof checking. Assumption-base semantics were also seen to facilitate the rigorous analysis of $\mathcal{NDL}$ proofs, leading to a tractable and elegant proof theory.

We have contrasted $\mathcal{NDL}$ with typed logical frameworks, and we have seen that the Curry-Howard isomorphism does not obtain in $\mathcal{NDL}$: there is no isomorphism between it and a typed λ-calculus, because $\mathcal{NDL}$ does not have variables. This disparity can be traced to a foundational difference on the issue of constructivism. By doing away with the layer of explicit proof abstraction, DPLs such as $\mathcal{NDL}$ allow for shorter, cleaner, and more efficient deductions.

$\mathcal{NDL}$ does not have any mechanisms for automated reasoning, which means that every proof in it must be given exclusively in terms of introduction and elimination rules for the logical connectives (as well as **assume**). For large examples this would be impractical, and in that respect $\mathcal{NDL}$ is not a realistic tool for proof engineering. An extension of $\mathcal{NDL}$ that overcomes this is Athena, a DPL that integrates computation and deduction; includes a higher-order functional language in the style of Scheme and ML; allows for trusted tactics; and is seamlessly integrated with cutting-edge automatic theorem provers such as Vampire [32] and Spass [33]. These features allow the user to skip tedious steps, concentrating instead at the interesting parts of the proof. Nevertheless, $\mathcal{NDL}$ is the deductive kernel of Athena, and therefore formulating and studying $\mathcal{NDL}$ in isolation lends insight to the fundamental ideas behind DPLs and assumption bases in general.

References

1. K. Arkoudas. Denotational Proof Languages. PhD dissertation, MIT, 2000.
2. K. Arkoudas, S. Khurshid, D. Marinov, and M. Rinard. Integrating model checking and theorem proving for relational reasoning. In *Proceedings of the 7th International Seminar on Relational Methods in Computer Science (RelMiCS 7)*, Malente, Germany, May 2003.
3. K. Arkoudas and M. Rinard. Deductive runtime certification. In *Proceedings of the 2004 Workshop on Runtime Verification*, Barcelona, Spain, April 2004.
4. T. Arvizo. A virtual machine for a type-ω denotational proof language. Masters thesis, MIT, June 2002.
5. M. Bergmann, J. Moor, and J. Nelson. *The Logic Book*. Random House, New York, 1980.
6. D. Bonevac. *Deduction*. Blackwell Publishing, 2003.
7. N. G. De Brujin. The Automath checking project. In P. Braffort, editor, *Proceedings of Symposium on APL*, Paris, France, December 1973.

8. R. L. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, EngleWood Cliffs, New Jersey, 1986.
9. I. M. Copi. *Symbolic Logic*. Macmillan Publishing Co., New York, 5th edition, 1979.
10. T. Coquand. Metamathetical investigations of a Calculus of Constructions. In P. Odifreddi, editor, *Logic and Computer Science*, pages 91–122. Academic Press, London, 1990.
11. T. Coquand and G. Huet. The Calculus of Constructions. *Information and Computation*, 76:95–120, 1988.
12. H.-D. Ebbinghaus, J. Flum, and W. Thomas. *Mathematical Logic*. Springer-Verlag, 2nd edition, 1994.
13. F. B. Fitch. *Symbolic Logic: an Introduction*. The Ronald Press Co., New York, 1952.
14. J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*, volume 7 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1989.
15. M. J. Gordon, A. J. Miller, and C. P. Wadsworth. *Edinburgh LCF: A Mechanized Logic of Computation*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, 1979.
16. M. J. C. Gordon and T. F. Melham. *Introduction to HOL, a theorem proving environment for higher-order logic*. Cambridge University Press, Cambridge, England, 1993.
17. R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, January 1993.
18. W. A. Howard. The formulae-as-types notion of construction. In J. Hindley and J. R. Seldin, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalisms*, pages 479–490. Academic Press, 1980.
19. G. Kahn. Natural semantics. In *Proceedings of Theoretical Aspects of Computer Science*, Passau, Germany, February 1987.
20. D. Kalish and R. Montague. *Logic: Techniques of Formal Reasoning*. Harcourt Brace Jovanovich, Inc., New York, 1964. Second edition in 1980, with G. Mar.
21. E. J. Lemmon. *Beginning Logic*. Hackett Publishing Company, 1978.
22. D. McAllester. *Ontic*. MIT Press, 1989.
23. G. Necula and P. Lee. Safe kernel extensions without run-time checking. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation*, Seattle, Washington, October 1996.
24. L. Paulson. *Isabelle, A Generic Theorem Prover*. Lecture Notes in Computer Science. Springer-Verlag, 1994.
25. F. J. Pelletier. A Brief History of Natural Deduction. *History and Philosophy of Logic*, 20:1–31, 1999.
26. G. D. Plotkin. A structural approach to operational semantics. Research Report DAIMI FN-19, Computer Science Department, Aarhus University, Aarhus, Denmark, September 1981.
27. N. Rescher. *Introduction to Logic*. St. Martin's Press, 1964.
28. J. C. Reynolds. *Theories of Programming Languages*. Cambridge University Press, 1998.
29. M. Rinard and D. Marinov. Credible compilation with pointers. In *Proceedings of the 1999 Workshop on Run-Time Result Verification*, Trento, Italy, July 1999.
30. P. Rudnicki. An overview of the Mizar Project. In *Proceedings of the 1992 Workshop on Types for Proofs and Programs*, Chalmers University of Technology, Bastad, 1992.

31. A. S. Troelstra and H. Schwichtenberg. *Basic Proof Theory*. Cambridge University Press, Cambridge, England, 1996.
32. A. Voronkov. The anatomy of Vampire: implementing bottom-up procedures with code trees. *Journal of Automated Reasoning*, 15(2), 1995.
33. C. Weidenbach. Combining superposition, sorts, and splitting. volume 2 of *Handbook of Automated Reasoning*. North-Holland, 2001.
34. M. Wenzel. Isar — a generic interpretative approach to readable formal proof documents. In *Proceedings of the 1999 conference on theorem proving in higher-order logic*, pages 167–183, 1999.
35. M. Wenzel and F. Wiedijk. A comparison of Mizar and Isar. *Journal of Automated Reasoning*, 2002.