Human-Readable Machine-Verifiable Proofs for Teaching Constructive Logic

Andreas Abel[1⋆], Bor-Yuh Evan Chang[2], and Frank Pfenning[2]

[1] Dept. of Computer Science, Ludwig Maximilian University, Munich, Germany
abel@informatik.uni-muenchen.de
[2] Dept. of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA
bechang@andrew.cmu.edu, fp@cs.cmu.edu

Abstract. A linear syntax for natural deduction proofs in first-order intuitionistic logic is presented, which has been an effective tool for teaching logic. The proof checking algorithm is also given, which is the core of the tutorial proof checker Tutch. This syntax is then extended to proofs on the assertion level which resemble single inferences one would make in a rigorous proof. The resulting language has only *four* constructs. Checking of these proofs is decidable, and an efficient algorithm is given.

1 Introduction

Teaching formal reasoning usually starts at the foundations that logic lays for it. Since the seminal work of Gentzen [Gen35] his natural deduction calculus has proven an adequate formalization of human reasoning at the *logical level*. To teach the formal discipline of logic to students, a computer seems to be an ideal tool to check proofs done by students.

Several proof tutor systems are available for education purposes, like ETPS [GRB93], ProofEasy [Bur98], the CMU Proof Tutor [SS94] or the HTML-based alfie [vS00]. Most of these systems require the student to develop a proof by interacting with the proof tutor. This rigid framework allows the proof tutor system to provide detailed and helpful guidance to the student working out a proof. However, we noticed the following drawbacks of the interactive style. First, the student must learn numerous commands to steer the interactive interface. Secondly, to remedy a mistake made early in the proof often requires backtracking to the point of the error, losing all work since then. Finally, the user is very restricted in his proof development by the direction the system takes him. In practice, proofs are developed on paper where one can work on goals in any order, mix forward and backward reasoning, erase wrong parts, and replace them by correct steps.

⋆ This work was carried out during a visit to the CMU in Pittsburgh (email: abel@cs.cmu.edu). It was supported by the Graduiertenkolleg Logik in der Informatik (GKLI), Munich, and the Office of Technology in Education, Carnegie Mellon University.

In natural deduction, proofs can be developed either forward or backward. Forward reasoning starts at the hypotheses and works down to the conclusion, while backward reasoning begins at the conclusion and proceeds up to the hypotheses. These issues led us to design our *Tut*orial Proof *Ch*ecker, Tutch[1], in the following way. Proofs are written up in an ASCII-file, which gives the user the freedom to edit the proof with his favorite editor and in the order he chooses. This freedom enables the user to fill gaps by combining forward and backward steps. The proof checker is a compiler-like tool that provides feedback to the user by pointing to remaining gaps or accepting the proof. We believe that our approach comes closer to teaching students how to write a formal proof rather than how to complete one in a specific tutor system.

The first prototype of Tutch was well accepted by the students and used successfully for intuitionistic propositional logic. However, larger proofs, properties of functional programs for example, were tedious since each step had to be a single natural deduction inference. For practical formal reasoning this restriction is highly unsatisfactory, so we set out to find a new notion of "single inference" which more closely corresponds to a step one would make in a rigorous proof.

Decades ago a formal connection between proofs and programs was stated that is today known as the Curry-Howard Isomorphism. It seems sensible to extend this connection to the method of formal proof development. While numerous high-level programming languages have been designed since the 1960s, concerning the codification of proofs, "we seem to be stuck at the assembly language level" [Wen99, p. 2].

In this article, we will motivate and formally define a proof syntax for first-order logic which allows bigger steps and could be the basis for a "high-level proving language". We analyze the principal steps found in rigorous proofs as a model for designing a proof language. Surprisingly, *four* constructs will be sufficient. To our knowledge, this simplicity is novel as well as the explicit formulation of a case construct which chains together an arbitrary number of eliminations of disjunctions and existential quantifiers.

The remainder of this paper is organized as follows: in Sect. 2 we will present a linear syntax for natural deduction proofs together with an algorithm for proof checking. Then we will discuss how to enlarge the proof steps by incorporating proof search. In Sect. 3 we will motivate and present a syntax that resembles human proof steps. In Sect. 4 an algorithm to verify these steps will be given. Sect. 5 contains experiences from using Tutch in an university course, and Sect. 6 summarizes this article, points out directions of further development and discusses some related work.

Preliminaries and Notation. We assume that the reader is familiar with the concepts of propositions, judgments and hypothetical judgments à la Martin-Löf and with the intuitionistic natural deduction calculus for first-order logic. We furthermore assume familiarity with intuitionistic sequent calculi as proof search

[1] Tutch is pronounced like "touch".

tools for natural deduction. A gentle introduction can be found in [Pfe99], and the first presentation was given by Gentzen [Gen35].

We will denote propositions with A, B, C and L, first-order terms with r, s and t and types with ρ, σ, τ and ι. Bold typeface indicates a vector; for example, $\boldsymbol{A}$ denotes a vector of propositions.

2 A Linear Syntax for Natural Deduction Proofs

The most direct and intuitive way of writing natural deduction proofs is in the form of trees. However, this is not common practice since the trees rapidly explode in size as the propositions become less trivial. A simple solution is to put the steps in linear form, which is common to various systems. Each proof step is just a proposition A. The step is valid if A follows from already proven propositions by application of a single rule.

Step	S	$::= A$	*Assertion*	*Hypothesis*	H	$::= A$		*Assertion* $(\supset\mathcal{I}, \vee\mathcal{E})$
		$\mid$ `[`H`;`S^*`]`	*Frame*			$\mid$ $x{:}\tau$		*Parameter* $(\forall\mathcal{I})$
						$\mid$ $x{:}\tau, A(x)$		*Constraint* $(\exists\mathcal{E})$
Proof	S^*	$::= A$	*Final step*					
		$\mid$ $S;S^*$	*Step sequence*					

Table 1. Linear Proof Syntax.

Another problem with proof trees is the slight ambiguity involving hypotheses. For example, a hypothesis discharged by an implication introduction is cancelled only at the leaves of the current subtree—not at all leaves. This is a common source of errors for students; it is easy to construct a proof tree that looks “right” but suffers from such subtle flaws. Thus, we decided to scope assumptions explicitly by using brackets `[...]`, which form a *frame*. For instance, in `[`A`;`$\ldots C$`]` $\ldots$, the hypothesis A may occur anywhere in the proof starting at the semicolon `;` and ending at the right bracket `]`, but not in the parts following the closing bracket. These considerations motivate the grammar for proofs given in Table 1. The final step of a proof must always be an assertion A; this is the proposition that is proven.

2.1 Judgments and Proof Checking

A proof step containing an assertion A establishes the *judgment* “A is true”, written $\vdash A$. The evidence of this judgment is given by a rule of the natural deduction calculus. In most cases, this will involve truth of other propositions, like the truth of A and B in the rule $\wedge\mathcal{I}$.

$$\frac{\vdash A \qquad \vdash B}{\vdash A \wedge B}\ \wedge\mathcal{I} \qquad\qquad \frac{\vdash \exists x{:}\tau.A(x) \qquad x{:}\tau, A(x) \vdash C}{\vdash C}\ \exists\mathcal{E}$$

The following four rules involve *hypothetical judgments*: $\supset\mathcal{I}$, $\forall\mathcal{I}$, $\vee\mathcal{E}$ and $\exists\mathcal{E}$. For instance, the second premise of rule $\exists\mathcal{E}$ (see above) is the hypothetical judgment: "under the assumption of the truth of $A(x)$ with some witness x, the assertion C is true". In Tutch, hypothetical judgments are proven by frames; this one in particular is represented by the frame `[`$x : \tau$`,`A`;`$\ldots C$`]`.

We will refer to the list of already proven non-hypothetical and hypothetical judgments by Δ. Then we can write all one-step inferences of the natural deduction calculus as axioms. For example, the rule $\wedge\mathcal{I}$ becomes $\Delta, \vdash A, \vdash B \Rightarrow A \wedge B$. The complete list of axioms is given in Table 2.

Judgments

$\Delta \Rightarrow A$	A follows from Δ by application of one rule.
$\Delta \blacktriangleright\!\blacktriangleright S^* : C$	S^* is a valid proof of C.

One-step inferences

$$\frac{}{\Delta, (A \vdash B) \Rightarrow A \supset B}\,\supset\!\mathcal{I} \qquad \frac{}{\Delta, \vdash A \supset B, \vdash A \Rightarrow B}\,\supset\!\mathcal{E}$$

$$\frac{}{\Delta, \vdash A, \vdash B \Rightarrow A \wedge B}\,\wedge\mathcal{I} \qquad \frac{}{\Delta, \vdash A \wedge B \Rightarrow A}\,\wedge\mathcal{E}_1 \qquad \frac{}{\Delta, \vdash A \wedge B \Rightarrow B}\,\wedge\mathcal{E}_2$$

$$\frac{}{\Delta, \vdash A \Rightarrow A \vee B}\,\vee\mathcal{I}_1 \qquad \frac{}{\Delta, \vdash B \Rightarrow A \vee B}\,\vee\mathcal{I}_2 \qquad \frac{}{\Delta, \vdash A \vee B, A \vdash C, B \vdash C \Rightarrow C}\,\vee\mathcal{E}$$

$$\frac{}{\Delta \Rightarrow \top}\,\top\mathcal{I} \qquad \frac{}{\Delta, x{:}\tau \vdash A \Rightarrow \forall x{:}\tau.A}\,\forall\mathcal{I} \qquad \frac{}{\Delta, \vdash \forall x{:}\tau.A \Rightarrow [t/x]A}\,\forall\mathcal{E}$$

$$\frac{}{\Delta, \vdash \bot \Rightarrow C}\,\bot\mathcal{E} \qquad \frac{}{\Delta, \vdash [t/x]A \Rightarrow \exists x{:}\tau.A}\,\exists\mathcal{I} \qquad \frac{}{\Delta, \vdash \exists x{:}\tau.A, (x{:}\tau, A \vdash C) \Rightarrow C}\,\exists\mathcal{E}$$

Proof checking

$$\frac{\Delta \Rightarrow C}{\Delta \blacktriangleright\!\blacktriangleright C : C}\,\mathit{final} \qquad \frac{\Delta \Rightarrow A \quad \Delta, \vdash A \blacktriangleright\!\blacktriangleright S^* : C}{\Delta \blacktriangleright\!\blacktriangleright A; S^* : C}\,\mathit{line} \qquad \frac{\Delta, \vdash H \blacktriangleright\!\blacktriangleright S_1^* : A \quad \Delta, (H \vdash A) \blacktriangleright\!\blacktriangleright S_2^* : C}{\Delta \blacktriangleright\!\blacktriangleright \texttt{[}H\texttt{;}S_1^*\texttt{]}\texttt{;}S_2^* : C}\,\mathit{frame}$$

Table 2. Tutch Proof Checking.

Validity of a proof can be checked by the following procedure: a proof step is valid when the assertion A it states follows by a one-step inference. Before we check the next step, we add $\vdash A$ to the list of established judgments Δ. In case the step is a frame, we temporarily add its hypotheses to Δ, prove the frame, and then add its conclusion A to Δ to check the remainder of the proof. A formal description is given in Table 2.

2.2 Enlarging the proof steps

In essence, Tutch is a *guided forward reasoner*; it is a forward reasoner since it accumulates proven judgments in Δ and works forward towards the goal, and it is guided by the proof plan S^*. The proof "plan" is very precise in our case since every step must be given. We can weaken this requirement and allow the omission of steps; then the judgment $\Delta \Rightarrow A$ must be replaced by another one which implements proof search. The primary candidate for this is the intuitionistic sequent calculus $\Delta \Longrightarrow A$.

In its most general formulation the sequent calculus is very non-deterministic which makes proof search rather ineffective. A complete strategy that cuts down the search space considerably is to alternate phases of invertible steps and non-invertible steps. This strategy of *focusing proofs* has been formulated by Andreoli [And92] for linear logic and adapted to intuitionistic linear logic [Pfe99]. We distinguish four classes of rules.

1. Invertible right rules: $\supset R$, $\forall R$ and $\wedge R$. These introduce new hypotheses or parameters, or split the conclusion (the *goal*), respectively. The axiom $\top R$ can be added to this group since truth is isomorphic to an empty conjunction.
2. Invertible left rules: $\vee L$, $\exists L$ and $\wedge L$, perform case distinction, witness extraction, and hypothesis splitting. Since absurdity can be seen as the empty disjunction, the axiom $\bot L$ fits into this class as well.
3. Non-invertible right rules: $\vee R_1$, $\vee R_2$ and $\exists R$. These rules involve decisions and can be chained together with $\wedge R$, $\top R$ and $\bot L$ to complete a goal. We call this strategy *finishing*.
4. Non-invertible left rules: $\supset L$, $\forall L$, $\wedge L_1$ and $\wedge L_2$. They are chained together in the *focusing* strategy. To solve the current goal we pick one hypothesis which has the form of a lemma, i.e., it has leading universal quantifiers and conditions, and we instantiate them to solve the goal. If we succeed, we continue by proving the conditions.

Note that rules for conjunction appear in all four groups and can be treated as invertible or non-invertible.

Even under application of strategies the general provability problem $\Delta \Longrightarrow A$ is undecidable. Since we desire efficient proof checking we have to limit to size of the steps in the proof plan. In the next section we will investigate what is considered to be a "human" proof step and how to formalize it to obtain a new proof grammar, which gives a reasonable size limit on steps.

3 Towards Human Proof Steps

Once the student has mastered the logic and proceeds to proofs of mathematical theorems or properties of programs, it is no longer feasible to formalize these proofs as a sequence of one-step inferences in the natural deduction calculus. In mathematical practice, the proof steps are larger in most cases. Following the analysis of Huang [Hua94], justifications are provided at three levels: the *logical* level, the *assertion* level, and the *proof* level. The aforementioned system operates at the logical level; however, humans apply logical rules implicitly "in the background". An explicit reference to them interrupts rather than supports the flow of reasoning. In mathematical proofs, humans provide justifications on the assertion level—by citing axioms, definitions, or theorems—or on the proof level. Justifications on the proof level such as "by analogy" are hard to verify by machine and their formalization is still an open problem.

The contribution of this paper is to give a simple proof language that allows steps at the assertion level. However, there are two important differences. First, our work is carried out in the context of intuitionistic logic. Second, we also combine the invertible rules into bigger steps, introducing multiple hypotheses or cases at once. Our proof language will be an extension of our linear syntax, and we will be careful not to lose the clear structure we have developed. In particular, we will continue to be explicit about hypotheses; all assumptions and parameters that are used in the proof must be declared prior to their usage.

```
axiom indNat : P(0) ⊃ (∀x:nat. P(x) ⊃ P(s(x))) ⊃ ∀n:nat.P(n);
axiom eq0 :    0 = 0;
axiom eqS :    ∀x:nat.∀y:nat. x = y ⊃ s(x) = s(y);

assertion proof splitNat : ∀x:nat. 0 = x ∨ ∃y:nat.s(y) = x ≡
assume x:nat in
    % Induction on x:nat
    % Base case: x = 0
    0 = 0 by axiom eq0;
    % Step case: x = s(x')
    assume x':nat, 0 = x' ∨ ∃y:nat.s(y) = x' in
        0 = s(x') ∨ ∃y:nat.s(y) = s(x') by
        case 0 = x' ∨ ∃y:nat.s(y) = x' of
              0 = x'                     ⟶  s(0) = s(x') by axiom eqS
           || y:nat where s(y) = x'  ⟶  s(s(y)) = s(x') by axiom eqS
        end
    end;
    0 = x ∨ ∃y:nat.s(y) = x by axiom indNat
end;
```

Fig. 1. Example: $\forall x{:}nat.\, 0 = x \vee \exists y{:}nat.s(y) = x$

The key question that guided our design is: "What is considered a single proof step in mathematical practice?" We identified the following steps:

1. Introduction of new hypotheses ("assume", "let") and parameters ("fix").
2. Application of an axiom, a definition, a lemma or a theorem.
3. Application of a local lemma.
4. Use of a special inference rule for a special area of mathematics.
5. Distinguishing cases.
6. Initiating mathematical induction.
7. Reference to the induction hypothesis.

Since we restrict ourselves to first-order logic for now, we do not treat induction explicitly; step 6 is the application of an induction axiom and subsumed under step 2; step 7 is a special case of step 3. Special inference rules (step 4), often incorporated into a theorem prover by *tactics*, are not treated in our simple approach but in more advanced systems like Isabelle [Pau90] and *Theorema* [BJK+97].

For each of the remaining steps we introduce a syntax. The resulting grammar for assertion level proofs is the following:

$$
\begin{array}{lll}
\textit{Step} & S & ::= \texttt{assume}\, H_1, \ldots, H_n \;\texttt{in}\; S^* \;\texttt{end} \\
 & & \mid C \;\texttt{by case}\; \boldsymbol{A} \;\texttt{of}\; \boldsymbol{K}^1 \longrightarrow S^{*1} \mid\mid \ldots \mid\mid \boldsymbol{K}^n \longrightarrow S^{*n} \;\texttt{end} \\
 & & \mid A \;\texttt{by lemma}\; l \\
 & & \mid \texttt{triv}\; A \\
\textit{Proof} & S^* & ::= S \mid S; S^* \\
\textit{Hypothesis} & H & ::= A \mid x{:}\tau \\
\textit{Constraint} & K & ::= \langle x_1{:}\tau_1, \ldots, x_m{:}\tau_m \rangle A
\end{array}
$$

The `triv` construct justifies a proposition that follows "trivially" by application of some logical rules or a local lemma. The `case` syntax formalizes case distinction and existential elimination. As in mathematical practice we allow parallel case splitting of a list of formulas $\boldsymbol{A}$. The cases are given by equally long lists of constraints $\boldsymbol{K}^1, \ldots, \boldsymbol{K}^n$ where n is the number of cases. For disjunction elimination, these constraints will simply be propositions and the list of parameters $\boldsymbol{x} : \boldsymbol{\tau}$ will be empty. In case of existential elimination, $\boldsymbol{x} : \boldsymbol{\tau}$ will contain new parameters which stand for the witnesses of the existential assertion. Disjunction and existential elimination can go hand in hand as demonstrated in the example below. For just existential eliminations, we introduce the syntactic sugar

$$
\begin{bmatrix} \texttt{obtain} \\ \quad x_1{:}\tau_1, \ldots, x_m{:}\tau_m \;\texttt{where}\; A \;\texttt{in}\; S^* \\ \texttt{end} \end{bmatrix} \text{ for } \begin{bmatrix} \texttt{case}\, \exists x_1{:}\tau_1 \ldots \exists x_m{:}\tau_m . A \;\texttt{of} \\ \quad x_1{:}\tau_1, \ldots, x_m{:}\tau_m \;\texttt{where}\; A \longrightarrow S^* \\ \texttt{end} \end{bmatrix}
$$

Note that $\boldsymbol{x{:}\tau}$ `where` A is the external syntax for the constraint $\langle \boldsymbol{x{:}\tau} \rangle A$. Furthermore we drop the keyword `triv` and use "%" to introduce a line of comment in the external syntax. Fig. 1 exemplifies our proof language by formally proving in Heyting Arithmetic that each natural number x is either 0 or $y + 1$ for some number y.

4 Proof Checking

In this section, we give a formal system for verifying "human step" proofs. We present the system in the form of seven judgments listed in Table 3. These can be interpreted operationally and yield an effective decision procedure for the validity of proofs. More precisely, a proof S^* can be checked in $O(|S^*|^2)$ worst-case time modulo the time spent on unification and lookup of lemmata and already proven propositions.

The validity of an assertion level proof is verified by checking each step in two contexts, Λ (available lemmata) and Δ (already proven assertions). A step proving assertion A is verified by using the *strategies* given in Table 3. Before proceeding to the next step, we add A to Δ. A formal description of the algorithm is given in the first section of of Table 4.

Contexts	
Λ	Collection of available lemmata L.
Δ	Already proven assertions A and introduced parameters $x{:}\tau$.
Proof Checking	
$\Lambda \vert \Delta \rhd\!\!\rhd S^* : C$	The step sequence S^* proves C.
$\Lambda \vert \Delta \rhd S : C$	The step S proves C.
Strategies	
$\Delta \Longrightarrow_F C$	Finishing: Proving C by right rules.
$\Delta; A \gg C$	Focusing: Proving C as a consequence of A.
$\Delta; L \Longrightarrow_L C$	Lemma: Proving C by lemma L.
$K' \twoheadrightarrow_C \boldsymbol{K}$	Cases: Determining case splitting of the formulas in $\boldsymbol{K}$ with constraints K'.
$K' \succ_C K$	Cases: Proving case covering on constraint K'.

Table 3. Judgments.

4.1 Strategies

As discussed in the previous section, the steps we consider are: introducing new hypotheses, applying a lemma, distinguishing cases, or a series of "trivial" logical inferences, which motivates the four constructs (`assume`, `case`, `lemma`, `triv`). One logical rule is always available: hypotheses in form of conjunctions can be split at any time ($\wedge L$). In other words, on the left, "$A \wedge B$" is isomorphic to "A, B". All the other rules are invoked by specific *strategies*. Each kind of step determines a strategy as described below and treated formally in Table 4.

`assume` This strategy takes a goal of the form $A \supset C$, adds the hypothesis A to Δ, and simplifies the goal to C. This corresponds to $\supset R$ in the sequent calculus. Analogously, in case of $\forall x{:}\tau.A$, it introduces $x{:}\tau$ as a new parameter ($\forall R$). Several introductions may be chained together. Note that each hypothesis and parameter must be explicitly named and in the order it is to be introduced.

`triv` A step is considered trivial if the assertion C it states is an immediate consequence of already established assertions in Δ. That is, C follows by contradiction ($\bot L$) or application of right rules like $\vee R_1$, $\vee R_2$, $\wedge R$ and $\exists R$. This strategy is called goal-directed. Another "trivial" step—in the sense that no special justification is provided—is the application of a "local lemma" (see below).

`lemma` As a proof step, two kinds of lemmata L can be applied: either a globally available lemma from Λ, which must be referenced by its name l; or one established inside the current proof in an earlier step (called a *local lemma*), which is applied anonymously and chosen non-deterministically from Δ. In both cases, L is passed to the $\Longrightarrow_L$ strategy.

`case` To check a proof of C by case distinction over A, we first determine that the subproof S^{*i} for each provided clause concludes in a proposition C^i from which the goal C immediately follows. Finally, we check that the given constraints K^i for the clauses adequately cover the possible cases induced

Sequences and Splitting Conjunctions

$$\frac{\Lambda|\Delta \rhd S : C}{\Lambda|\Delta \rhd\!\!\rhd S : C}\; final \qquad \frac{\Lambda|\Delta \rhd S : A \quad \Lambda|\Delta, A \rhd\!\!\rhd S^* : C}{\Lambda|\Delta \rhd\!\!\rhd S;\, S^* : C}\; step \qquad \frac{\Lambda|\Delta, A, B \rhd S : C}{\Lambda|\Delta, A \wedge B \rhd S : C}\; \wedge L$$

Introducing Hypotheses

$$\frac{\Lambda|\Delta, A \rhd \texttt{assume}\ \boldsymbol{H}\ \texttt{in}\ S^*\ \texttt{end} : C}{\Lambda|\Delta \rhd \texttt{assume}\ A, \boldsymbol{H}\ \texttt{in}\ S^*\ \texttt{end} : A \supset C}\; assume_A$$

$$\frac{\Lambda|\Delta \rhd\!\!\rhd S^* : C}{\Lambda|\Delta \rhd \texttt{assume in}\ S^*\ \texttt{end} : C}\; assume_- \qquad \frac{\Lambda|\Delta, x{:}\tau \rhd \texttt{assume}\ \boldsymbol{H}\ \texttt{in}\ S^*\ \texttt{end} : A}{\Lambda|\Delta \rhd \texttt{assume}\ x{:}\tau, \boldsymbol{H}\ \texttt{in}\ S^*\ \texttt{end} : \forall x{:}\tau. A}\; assume_x$$

Proof by Filling

$$\frac{\Delta \Longrightarrow_F C}{\Lambda|\Delta \rhd \texttt{triv}\ C : C}\; fill$$

Lemma Application

$$\frac{\Delta; L \Longrightarrow_L C}{\Lambda, l{:}L|\Delta \rhd C\ \texttt{by lemma}\ l : C}\; lemma \qquad \frac{\Delta; L \Longrightarrow_L C}{\Lambda|\Delta, L \rhd \texttt{triv}\ C : C}\; local$$

Proof by Cases

$$\frac{\Lambda|\Delta, A, \ulcorner K^i \urcorner \rhd\!\!\rhd S^{*i} : C^i \quad \Delta, A, \ulcorner K^i \urcorner, C^i \Longrightarrow_F C \quad (\text{for all } i.\ 1 \le i \le n) \qquad \langle\rangle A \twoheadrightarrow_C \boldsymbol{K}}{\Lambda|\Delta, A \rhd C\ \texttt{by case}\ A\ \texttt{of}\ K^1 \longrightarrow S^{*1} \mid\mid \ldots \mid\mid K^n \longrightarrow S^{*n}\ \texttt{end} \ : \ C}\; cases$$

where $\ulcorner K \urcorner$ is defined as follows:

$$\ulcorner \langle\rangle A \urcorner = A$$
$$\ulcorner \langle a{:}\tau, \boldsymbol{b{:}\sigma} \rangle A \urcorner = a{:}\tau, \ulcorner \langle \boldsymbol{b{:}\sigma} \rangle A \urcorner$$

Table 4. Proof Checking.

by A. The case construct guides forward reasoning along the left rules $\vee L$, $\perp L$, and $\exists L$. This can be extended to parallel case distinction over a list of propositions $\boldsymbol{A}$ in a straightforward manner.

4.2 Finishing

Finishing ($\Longrightarrow_F$) is used to verify a series of "trivial" logical steps, as well as an auxiliary judgment to complete the verification of other kind of steps. As shown in Table 5, finishing consists of primarily right rules (with $\perp L$ being the special case). This corresponds to seeing whether we can break up the goal into pieces that we have proven already. Also, notice that no new hypotheses are added to Δ while finishing; the $\supset R_F$ rule does not introduce the premise A, and $\forall R_F$ does not introduce $x{:}\tau$ as new parameter so that x cannot become a witness for an existential in C.

The rule $\exists R_F$ involves "guessing" a witness t, which can be implemented by introducing a unification variable X for t and solving for X at the initial sequents. This requires first-order unification for which efficient algorithms are known.

$$\frac{}{\Delta, C \Longrightarrow_F C}\ init_F \qquad \frac{}{\Delta \Longrightarrow_F \top}\ \top R \qquad \frac{}{\Delta, \bot \Longrightarrow_F C}\ \bot L_F$$

$$\frac{\Delta \Longrightarrow_F A}{\Delta \Longrightarrow_F A \vee B}\ \vee R_F^1 \qquad \frac{\Delta \Longrightarrow_F B}{\Delta \Longrightarrow_F A \vee B}\ \vee R_F^2 \qquad \frac{\Delta \Longrightarrow_F [t/x]A}{\Delta \Longrightarrow_F \exists x{:}\tau.A}\ \exists R_F$$

$$\frac{\Delta \Longrightarrow_F A \quad \Delta \Longrightarrow_F B}{\Delta \Longrightarrow_F A \wedge B}\ \wedge R \quad \frac{\Delta \Longrightarrow_F B}{\Delta \Longrightarrow_F A \supset B}\ \supset R_F \quad \frac{\Delta \Longrightarrow_F C}{\Delta \Longrightarrow_F \forall x{:}\tau.C}\ \forall R_F \quad \frac{\Delta, A, B \Longrightarrow_F C}{\Delta, A \wedge B \Longrightarrow_F C}\ \wedge L_F$$

Table 5. Finishing.

4.3 Focusing and Lemma Application

The focusing strategy $\gg$ is designed to capture the idea that in mathematical practice, one instantiates all universal quantifiers and all preconditions at once when applying a lemma. We contrast this with individual applications of $\supset L$ or $\forall L$ in the sequent calculus. For instance, consider Δ to be:

$$\forall x{:}\tau.\forall y{:}\sigma.P(x,y) \supset Q(x,y) \supset R(x,y)$$

Suppose we choose the first assertion and choose a parameter t, now Δ is:

$$\forall x{:}\tau.\forall y{:}\sigma.P(x,y) \supset Q(x,y) \supset R(x,y),\ \forall y{:}\sigma.P(t,y) \supset Q(t,y) \supset R(t,y)$$

Clearly, the number of possible choices grows quickly. Focusing allows the instantiation of all parameters and all preconditions of a lemma to occur at once as detailed in Table 6. This formulation is adapted from the discussion presented in [Pfe99]. Notice that the postcondition of an implication and the kernel of a universal quantification continues to be focused upon corresponding to repeated application of $\supset L$ or $\forall L$ rules. The precondition of an implication may be verified by finishing.

The $\Longrightarrow_L$ strategy sets up the focusing procedure to apply a lemma L. For this strategy, we allow the introduction of hypotheses ($\supset R_L$) and parameters ($\forall R_L$) into Δ. Note that due to the anatomy of the focusing procedure on L, these new hypotheses can only be used to verify preconditions of the lemma by finishing. This design allows us, for example, to prove $A \wedge B \supset C$ in one step by applying lemma $A \supset (B \supset C)$, but we cannot derive $A \supset A$ without introducing hypothesis A.

4.4 Proving by Cases

For case distinctions, the verification is divided into two phases: determining the split tree of the formula A the case distinction is over ($\twoheadrightarrow_C$), called the *split formula*, and checking that the stated cases cover the leaves of split tree ($\succ_C$).

The left hand side of both judgments is of the form $\langle \boldsymbol{x{:}\tau} \rangle A$. The list $\boldsymbol{x{:}\tau}$ contains the free *existential* variables of A. During the split phase, a disjunction in the split formula A creates two branches in the split tree ($\vee L$), and leading

Focusing

$$\frac{}{\Delta; C \gg C}\ init_{\gg}$$

$$\frac{\Delta; B \gg C \quad \Delta \Longrightarrow_F A}{\Delta; A \supset B \gg C}\ \supset L \qquad \frac{\Delta; [t/x]A \gg C}{\Delta; \forall x{:}\tau.A \gg C}\ \forall L \qquad \frac{\Delta; A \gg C}{\Delta; A \wedge B \gg C}\ \wedge L^1_{\gg} \qquad \frac{\Delta; B \gg C}{\Delta; A \wedge B \gg C}\ \wedge L^2_{\gg}$$

Lemma Application

$$\frac{\Delta, A; L \Longrightarrow_L B}{\Delta; L \Longrightarrow_L A \supset B}\ \supset R_L \qquad \frac{\Delta, x{:}\tau; L \Longrightarrow_L A}{\Delta; L \Longrightarrow_L \forall x{:}\tau.A}\ \forall R_L \qquad \frac{\Delta, A, B; L \Longrightarrow_L C}{\Delta, A \wedge B; L \Longrightarrow_L C}\ \wedge L_L$$

$$\frac{\Delta; L \gg C}{\Delta; L \Longrightarrow_L C}\ focus$$

Table 6. Focusing and Lemma Application.

existential quantifications in A are removed ($\exists L$). Splitting absurdity succeeds immediately ($\bot L_C$).

After completing the splitting, every leaf of the split tree has to be covered by one of the given cases which are provided as constraints $K^1, \ldots, K^n$ (rule *cover*). A constraint $K^i \equiv \langle \boldsymbol{a{:}\sigma} \rangle C$ consists of a kernel C and a list of parameters $\boldsymbol{a{:}\sigma}$ which appear in the ith clause of the `case` construct and are explicitly declared in the `where` statement. Basically, a constraint $\langle \boldsymbol{a{:}\sigma} \rangle C$ covers a leaf $\langle \boldsymbol{x{:}\tau} \rangle A$ if there is a parameter a for each existential variable x that is actually free in A (rules $init_{\succ_C}$, *strengthen*, and *name*). The $\wedge L$ rules allow the constraint to be weaker than the leaf, that is, to "forget" preconditions that are unnecessary for proof of the case.

Case Splitting

$$\frac{}{\langle \boldsymbol{x{:}\tau} \rangle \bot \twoheadrightarrow_C \boldsymbol{K}}\ \bot L_C \qquad \frac{\langle \boldsymbol{x{:}\tau} \rangle A \twoheadrightarrow_C \boldsymbol{K} \quad \langle \boldsymbol{x{:}\tau} \rangle B \twoheadrightarrow_C \boldsymbol{K}}{\langle \boldsymbol{x{:}\tau} \rangle A \vee B \twoheadrightarrow_C \boldsymbol{K}}\ \vee L$$

$$\frac{\langle \boldsymbol{x{:}\tau}, y{:}\sigma \rangle A \twoheadrightarrow_C \boldsymbol{K}}{\langle \boldsymbol{x{:}\tau} \rangle \exists y{:}\sigma.A \twoheadrightarrow_C \boldsymbol{K}}\ \exists L \qquad \frac{K' \succ_C K^i \quad (for\ some\ i\ .\ 1 \leq i \leq n)}{K' \twoheadrightarrow_C K^1, \ldots K^i, \ldots K^n}\ cover$$

Case Covering

$$\frac{}{\langle\rangle C \succ_C \langle\rangle C}\ init_{\succ_C} \qquad \frac{\langle \boldsymbol{x{:}\tau}, \boldsymbol{z{:}\rho} \rangle A \succ_C \langle \boldsymbol{a{:}\sigma} \rangle C \quad (y \notin \mathsf{FV}(A))}{\langle \boldsymbol{x{:}\tau}, y{:}\iota, \boldsymbol{z{:}\rho} \rangle A \succ_C \langle \boldsymbol{a{:}\sigma} \rangle C}\ strengthen$$

$$\frac{\langle \boldsymbol{x{:}\tau} \rangle [b/y]A \succ_C \langle \boldsymbol{a{:}\sigma} \rangle C}{\langle \boldsymbol{x{:}\tau}, y{:}\rho \rangle A \succ_C \langle \boldsymbol{a{:}\sigma}, b{:}\rho \rangle C}\ name \qquad \frac{\langle \boldsymbol{x{:}\tau} \rangle A \succ_C \langle \boldsymbol{a{:}\sigma} \rangle C}{\langle \boldsymbol{x{:}\tau} \rangle A \wedge B \succ_C \langle \boldsymbol{a{:}\sigma} \rangle C}\ \wedge L^1_C \qquad \frac{\langle \boldsymbol{x{:}\tau} \rangle B \succ_C \langle \boldsymbol{a{:}\sigma} \rangle C}{\langle \boldsymbol{x{:}\tau} \rangle A \wedge B \succ_C \langle \boldsymbol{a{:}\sigma} \rangle C}\ \wedge L^2_C$$

Table 7. Case Splitting and Covering.

5 Evaluation

After the technical discussion, we present some practical experiences with the proof checker Tutch. So far, only the version presented in Sect. 2 which allows one-step logical inferences has been tested by students. It was used in the undergraduate course *Constructive Logic* in Fall 2000 at Carnegie Mellon University. In the midterm evaluation, Tutch got positive feedbacks: 15 out of 26 found Tutch to be very *helpful* for doing the assignments (5 out of 5 points), only one student found it not helpful at all (1 point). The average score was 4.28. *Usability* of Tutch was rated 3.96 in the average. This rating is surprisingly high, given that Tutch had a simple Unix command line interface and had to be invoked every time a proof was to be verified. We credit the good feedback to our concept to treat proving like programming and proof development like program development. In particular, our proof language is designed after the example of structured programming languages like PASCAL or SML.

In the introduction, we listed some drawbacks of interactive proof tutors:

1. The student has to learn how to steer the interactive interface.
2. Mistakes can only be corrected by backtracking.
3. The system directs the proof development.

Our tutor Tutch avoids these drawbacks as argued in the following.

1. The interface of Tutch is almost trivial; in our case, what the student has to learn is the *proof language*. Because of its simplicity, that was unproblematic for the students so far.
2. A main advantage of writing proofs like programs is that the student does not edit a *proof script* containing commands which steer the interactive prover, but the *proof itself*. Thus, mistakes in proofs can be corrected easily. For example, forgotten steps can be inserted at the required place directly. In the case of proof scripts, insertion of a command usually causes the subsequent commands to fail, since the proof state they are supposed to manipulate has changed.
3. Proving in a proof language gives the students the freedom a programmer has in developing software: He can edit his program at any point, work bottom-up or top-down. In contrast, interactive provers often restrict development to the current proof state or force the user to input his proof via a structure editor. It can be very frustrating for a student that the proof step he wants to take is blocked by the system even if it is the wrong step in the current situation. We think it is better to give the student the freedom to try his solution and then learn what his mistake was rather than blocking wrong attempts from the beginning without giving the student a chance to see why.

An area where Tutch definitely has to improve is *error messages*. In the course evaluation the students gave Tutch an average of 3.68 points (out of 5) for its feedback on mistakes. But how to give good error messages is a topic of research itself.

6 Conclusion

We have presented a syntax for proof that resembles human steps and still is efficiently verifiable. We presume this system as presented here will be applicable in the educational context immediately and will be a nice addition to Tutch. Yet it is work in progress, and requires some extensions to become a more convenient and natural proof language. First, a syntax for the prominent proof method of induction has to be added. Secondly, equational reasoning must be supported adequately; in particular, it should be possible to write down a chain of equations without citing transitivity at each step.

The guiding principles for our design are proof-theoretic properties of the underlying logic, particular the distinctions between invertible and non-invertible rules, the distinctions between left and right rules, and focusing. This means our design is likely to be applicable to a rich variety of systems for which focusing system can be designed (including classical logic [Hua94], classical linear logic [And92], intuitionistic linear logic, modal logics, temporal logics, and others).

Related Work. Lamport gives general guidelines on how to structure rigorous proofs in [Lam93]. Comparable to our linear proof syntax with frames given in Sect. 2 is the *Block Calculus* by Dahn and Wolf [DW94] based on *classical* first-order logic.

Pioneering work in the area of human-readable machine-verifiable proofs has been done in the *Mizar* project [Rud92]. The Mizar language has been derived from common language used in mathematical proofs and therefore is very rich. We tried to find a minimum of concepts. Closest to our work is Wenzel's formal proof language for the tactic-based theorem prover Isabelle, called *Isar* [Wen99]. His approach is very flexible: he provides a simple meta-logical framework that lays the foundation for reasoning in any logic and adds special constructs for forward and backward reasoning in natural deduction. He provides statements analogous to our `assume` that handles invertible right rules, but has no counterpart of our `case` construct that chains invertible left rules.

A related area is the research on proof verbalization: how to generate a human-readable output of a computer-generated proof. Huang [Hua94] investigated what humans would regard as a single step at the *assertion level.* His results flowed into our design of the justification by `lemma`; the PROVERB system [HF97] implements his ideas.

Acknowledgments. We thank the students of the course 15-399 (Fall 2000) at Carnegie Mellon University for enduring the shortcomings of Tutch.

References

[And92] Jean-Marc Andreoli. Logic programming with focusing proofs in linear logic. *Journal of Logic and Computation*, 2(3):297–347, 1992.

[BJK+97] Bruno Buchberger, Tudor Jebelean, Franz Kriftner, Mircea Marin, Elena Tomuta, and Daniela Vasaru. A survey of the theorema project. In Wolfgang Küchlin, editor, *Proceedings of the 1997 International Symposium on Symbolic and Algebraic Computation (ISSAC)*, pages 384–391. ACM Press, 1997.

[Bur98] Rod Burstall. Teaching people to write proofs. In *CafeOBJ Symposium, Numazu, Japan*, April 1998.

[DW94] Bernd Ingo Dahn and Andreas Wolf. A calculus supporting structured proofs. *Journal of Information Processing and Cybernetics (EIK)*, 30(5–6):261–276, 1994. Akademie Verlag Berlin.

[Gen35] Gerhard Gentzen. Untersuchungen über das logische Schließen. *Mathematische Zeitschrift*, 39:176–210, 405–431, 1935. English translation in M. E. Szabo, editor, *The Collected Papers of Gerhard Gentzen*, pages 68–131, North-Holland, 1969.

[GRB93] Doug Goldson, Steve Reeves, and Richard Bornet. A review of several programs for the teaching of logic. *The Computer Journal*, 36:373–386, 1993.

[HF97] Xiaorong Huang and Armin Fiedler. Proof verbalization in PROVERB. In *Proceedings of the First International Workshop on Proof Transformation and Presentation (PTP)*, pages 35–36, 1997. Extended abstract.

[Hua94] Xiaorong Huang. Reconstructing proofs at the assertion level. In *12th International Conference on Automated Deduction*, volume 814 of *Lecture Notes in Artificial Intelligence*. Springer, 1994.

[Lam93] Leslie Lamport. How to write a proof. In *Global Analysis in Modern Mathematics*, pages 311–321. Publish or Perish, Houston, Texas, U.S.A., February 1993. Also appeared as SRC Research Report 94.

[Pau90] Lawrence Paulson. Isabelle: The next 700 theorem provers. In P. Odifreddi, editor, *Logic and Computer Science*, pages 361–386. Academic Press, 1990.

[Pfe99] Frank Pfenning. Automated theorem proving. Course Notes on the WWW, 1999. URL: http://www.cs.cmu.edu/~fp/courses/atp/atp.ps.

[Rud92] Piotr Rudnicki. An overview of the Mizar project. In *1992 Workshop on Types for Proofs and Programs, Bøastad, Sweden*. Chalmers University of Technology, 1992. URL: http://www.lfcs.informatics.ed.ac.uk/research/types-bra/proc/.

[SS94] Richard Scheines and Wilfried Sieg. Computer environments for proof construction. *Interactive Learning Environments*, 4:159–169, 1994.

[vS00] Björn von Sydow. alfie, a proof editor for propositional logic. WWW, 2000.

[Wen99] Markus Wenzel. Isar – a generic interpretative approach to readable formal proof documents. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Thery, editors, *Theorem Proving in Higher Order Logics, 12th International Conference, TPHOLs'99*, volume 1690 of *Lecture Notes in Computer Science*. Springer-Verlag, 1999.