Merging procedural and declarative proof

Cezary Kaliszyk and Freek Wiedijk

{cek,freek}@cs.ru.nl
Institute for Computing and Information Sciences,
Radboud University Nijmegen, the Netherlands

Abstract. There are two different styles for writing natural deduction proofs: the 'Gentzen' style in which a proof is a tree with the conclusion at the root and the assumptions at the leaves, and the 'Fitch' style (also called 'flag' style) in which a proof consists of lines that are grouped together in nested boxes.
In the world of proof assistants these two kinds of natural deduction correspond to procedural proofs (tactic scripts that work on one or more subgoals, like those of the Coq, HOL and PVS systems), and declarative proofs (like those of the Mizar and Isabelle/Isar languages).
In this paper we give an algorithm for converting tree style proofs to flag style proofs. We then present a rewrite system that simplifies the results. This algorithm can be used to convert arbitrary procedural proofs to declarative proofs. It does not work on the level of the proof terms (the basic inferences of the system), but on the level of the statements that the user sees in the goals when constructing the proof.
The algorithm from this paper has been implemented in the ProofWeb interface to Coq. In ProofWeb a proof that is given as a Coq proof script (even with arbitrary Coq tactics) can be displayed both as a tree style and as a flag style proof.

1 Introduction

Proof assistants are computer programs for constructing and checking proofs. In these systems one can distinguish between two quite different kind of entities that both might be considered the 'proofs' that are being checked:

- First there are the low level proofs of the logic of the system. In type theoretical systems these are the *proof terms*. In other systems they are built from tiny proof steps called *basic inferences*. Generally such proof objects are huge and constructed from a small number of basic elements.
- Then there also are the high level proof texts that the user of the system works with. Often these texts are *scripts* of commands from the user to the proof assistant. These texts are of a size comparable to traditional mathematical texts, and contain a much larger variety of proof steps. For instance both the Coq and HOL systems have dozens of *tactics* that can occur in this kind of proof.

The proof assistant does two things for the user. First it translates high level proofs into low level proofs, and secondly it checks the low level proofs obtained in this way with respect to the rules of the logic of the system.

As an example, the following 'high level' Coq proof script:

```
Lemma example : forall n : nat, n <= n.
intros.
omega.
Qed.
```

is translated to the following 'low level' proof term:

```
fun n : nat => Decidable.dec_not_not (n <= n) (dec_le n n) (fun H : ~ n <= n =>
  ex_ind (fun (Zvar1 : Z) (Omega5 : Z_of_nat n = Zvar1 / (0 <= Zvar1 * 1 + 0)%Z) =>
    and_ind (fun (Omega3 : Z_of_nat n = Zvar1) (_ : (0 <= Zvar1 * 1 + 0)%Z) =>
     let H0 := eq_ind_r (fun x : Z => (0 <= x + -1 + - Z_of_nat n)%Z -> False)
      (eq_ind_r (fun x : Z => (0 <= Zvar1 + -1 + - x)%Z -> False)
       (fast_Zopp_eq_mult_neg_1 Zvar1 (fun x : Z => (0 <= Zvar1 + -1 + x)%Z -> False)
        (fast_Zplus_comm (Zvar1 + -1) (Zvar1 * -1) (fun x : Z => (0 <= x)%Z -> False)
         (fast_Zplus_assoc (Zvar1 * -1) Zvar1 (-1) (fun x : Z => (0 <= x)%Z -> False)
          (fast_Zred_factor3 Zvar1 (-1) (fun x : Z => (0 <= x + -1)%Z -> False)
           (fast_Zred_factor5 Zvar1 (-1) (fun x : Z => (0 <= x)%Z -> False)
            (fun Omega4 : (0 <= -1)%Z => Omega4 (refl_equal Gt)))))))) Omega3) Omega3 in
     H0 (Zgt_left (Z_of_nat n) (Z_of_nat n) (inj_gt n n (not_le n n H))))
    Omega5) (intro_Z n))
```

which then is type checked and found to be correct.

A good proof assistant should hide low level proofs from the user of the system as much as possible. Just like a user of a high level programming language should not need to be aware that the program internally is translated into machine code or bytecode, the user of a proof assistant should not have to be aware that internally a low level proof is being constructed.

It depends much on the specific proof assistant what the high level proofs look like. There are two basic groups of systems, as first introduced in [7]:

The procedural systems such as Coq, HOL and PVS. These systems generally are descendants of the LCF system. The proofs of a procedural system consist of tactics operating on goals. This leads to proofs that can naturally be represented as *tree shaped* derivations in the style of Gentzen. For instance, the example Coq proof then looks like:

```
                  ──────────────── omega
                  n:nat ⊢ n <= n
                ──────────────────── intros
                ⊢ ∀n:nat, n <= n
```

The above is a screenshot from the display of our ProofWeb system. In practice it is more useful to have ProofWeb display the tree without contexts:

```
                    ─────── omega
                    n <= n
                  ──────────── intros
                  ∀n:nat, n <= n
```

The declarative systems. The main two systems of this kind are Mizar and Isabelle (when used with its declarative proof language Isar), but also automated theorem provers like ACL2 and Theorema can be considered to be declarative. There are experimental declarative proof languages, like the ones by Pierre Corbineau for Coq and by John Harrison for HOL Light.
The proofs of a declarative system are *block structured*. They basically consist of a *list* of statements, where each statement follows from the previous ones, with the system being responsible for automatically constructing the low level proof that shows this to be the case. Apart from these basic steps declarative proofs have other steps, like the `assume` step which introduces an assumption.
In declarative systems these proof steps are grouped into a hierarchical structure of *blocks*, just like in block structured programming languages. In declarative proofs these blocks are delimited by keywords like `proof` and `qed`.

Some systems might be considered not to be *fully* declarative in the sense that they still require the user to indicate *how* a statement follows from earlier statements. For instance this holds for Isabelle, where the user can (and sometimes must) give explicit inference rules. Indeed, it is common among the users of Isabelle to refer to the Isar proofs not as 'declarative' but 'structured'. However, for the purposes of this paper this distinction does not matter. In fact, the declarative proofs that we generate with our ProofWeb system also have the property that they contain an explicit tactic at each step in the proof.

Mathematicians generally think of their proofs in a declarative way. Declarative proofs are similar (although more precise and, with current technology, much more fine-grained) to the language that one finds in mathematical articles and textbooks.

The contribution of this paper is a generic method for converting a procedural proof to a declarative proof. For Coq this method has been implemented in the ProofWeb system. ProofWeb can display a high level Coq proof as a block structured list of statements. Here is how it will display the example proof:

```
1 n: nat              assumption
2   n <= n            omega
3   ∀n:nat, n <= n    intros 1-2
```

The rest of the paper details the algorithms used for this.

In 2006–2008 we ran a project called *Web deduction for education in formal thinking*, in which we built a system for logic education. Our system allows students to practice natural deduction proofs. It has the following design choices:

– Our system runs on a web server. This means that students do not need to install anything, can access their work from anywhere (as long as they have Internet access), and that teachers can easily keep track of the progress of their students. Our ProofWeb server is at `http://proofweb.cs.ru.nl/`. It can be tried using the guest login, with no registration.

- The system uses the Coq proof assistant, and the Coq proof language is not hidden from the user. Students are typing actual Coq proof scripts that use a restricted set of custom tactics to make their proofs correspond exactly to the proofs from their textbook. The use of Coq makes ProofWeb especially attractive for teachers who want their students to work on non-trivial examples.
- The system allows the students to both work in Gentzen style as well as in Fitch style. Proofs are displayed in (almost) exactly the same way that they are shown in the textbook. We decided to have our system be compatible with a popular logic textbook by Michael Huth and Mark Ryan [8].

The ProofWeb system can present the tree shaped proof that corresponds to the Coq proof script as a Fitch style proof. This means that it converts a procedural proof (the Coq script) to a declarative proof (the Fitch display). The method that it uses to do this is *generic*. It will work for converting *any* procedural proof to *any* declarative proof text, independent of the specific proof assistants involved or their logical foundations.[1]

We decided against presenting the conversion method that we used generically. In this paper we present just the method for the very specific situation of natural deduction proofs for first order predicate logic with equality. However, the method *is* perfectly generic. Also, our implementation already is not restricted to the small set of tactics that the users of ProofWeb are supposed to use. It will work with *any* Coq proof, providing a block structured Fitch style display of that proof.

The specifics of the first order logics that ProofWeb uses can be found in the ProofWeb manual [9]. We here just show an example for both logics in Figure 1. In ProofWeb flags are rendered as boxes (like in Huth and Ryan), with the right hand border of the boxes omitted to conserve space.

Declarative proofs are much more robust than procedural proofs, and for this reason can be expected to have a longer useful lifetime than procedural proofs. For this reason, development of the technology presented here might mean current formalizations get a longer useful lifetime. A current version of the procedural system can be used to export a formalisation declaratively. Keeping the declarative proof instead of the procedural one gives a much higher chance of the proof being accepted by future versions of the proof assistant.

The conversion algorithm presented here also works on proofs that have not been completed yet. In that case one gets a declarative proof with *gaps*. For instance in ProofWeb, the Fitch style display of the proof *before* the `omega` tactic is executed will be:

[1] The proof might contain some statements that have no good equivalent in the target system, and the automation of the target system might not always be able to bridge the gaps between the steps, but apart from those issues, a good starting point for a formalization in the target system can always be generated.

$$\dfrac{\dfrac{[\exists x(P(x) \vee \neg Q(a))]^{\mathtt{H1}} \qquad \dfrac{\dfrac{[P(b) \vee \neg Q(a)]^{\mathtt{H3}} \qquad \dfrac{[P(b)]^{\mathtt{H4}}}{\exists x\, P(x)}\exists\mathrm{i} \qquad \dfrac{\dfrac{[\neg Q(a)]^{\mathtt{H5}} \quad [Q(a)]^{\mathtt{H2}}}{\bot}\neg\mathrm{e}}{\exists x\, P(x)}\bot\mathrm{e}}{\exists x\, P(x)}\vee\mathrm{e}\,[\mathtt{H4},\mathtt{H5}]}{\exists x\, P(x)}\exists\mathrm{e}\,[\mathtt{H3}]}{\dfrac{\exists x\, P(x)}{Q(a) \to \exists x\, P(x)}\to\mathrm{i}\,[\mathtt{H2}]}}{\exists x(P(x) \vee \neg Q(a)) \to Q(a) \to \exists x\, P(x)}\to\mathrm{i}\,[\mathtt{H1}]$$

1	$\exists x(P(x) \vee \neg Q(a))$	
2	$\quad Q(a)$	
3	$\quad b \mid P(b) \vee \neg Q(a)$	
4	$\quad\quad P(b)$	
5	$\quad\quad \exists x\, P(x)$	$\exists$i 4
6	$\quad\quad \neg Q(a)$	
7	$\quad\quad \bot$	$\neg$e 6,2
8	$\quad\quad \exists x\, P(x)$	$\bot$e 7
9	$\quad \exists x\, P(x)$	$\vee$e 3,4–5,6–8
10	$\quad \exists x\, P(x)$	$\exists$e 1,3–9
11	$Q(a) \to \exists x\, P(x)$	$\to$i 2–10
12	$\exists x(P(x) \vee \neg Q(a)) \to Q(a) \to \exists x\, P(x)$	$\to$i 1–11

Fig. 1. Example derivation in Gentzen's and Fitch's systems.

```
1 n: nat                    assumption
    ...
2   n <= n
3   ∀n:nat, n <= n  intros 1-2
```

ProofWeb users often use the system through this feature. They do not look at the Coq proof state (which is also available to them), but just think in terms of the incomplete Fitch style proof.

This leads us to propose a new kind of prover interface. We call it a *luxury* declarative proof assistant (after a suggestion by Henk Barendregt). In a luxury system, the user does not see goals, but works on an incomplete declarative proof. This proof then can be modified in two ways:

– Either the user just edits the text, the common way to work in a declarative proof assistant. This is flexible but gives the user no help in writing the proof.

– Alternatively the user executes a *tactic* at a step in the proof that has not been sufficiently justified yet, i.e., for which the system has not yet generated a low level proof. The 'goal' that this tactic sees has the statement of this step as the conclusion, and all the statements before it that are in scope as the assumptions. The tactic then will generate subgoals, which will be added to the proof text as new steps in, if needed, new sub-blocks.
  Modifying a proof in this style (by executing a tactic at a not yet justified step), needs exactly the same algorithms that the conversion from a procedural proof to a declarative proof needs.
  If one 'grows' a declarative proof in such a way, it basically will consist of a merged version of all the subgoals that the proof would have gone through in the procedural system.

It is important that in a luxury system *both* ways of working are available simultaneously. It should not be *required* to use tactics to modify a proof.

A simple version of this luxury concept is the following. In a declarative prover the user has to formulate the appropriate `assume` steps himself, while in a procedural prover he just can type `intros`. However in a luxury prover, the `intros` command will be available, which then will generate all the needed `assume` steps automatically. Similarly, appropriate statements in the case of an induction or application of a lemma can be generated automatically by the system.

The conversion from a tree style proof to a block structured proof is straightforward. It consists of two phases:

– First the tree is converted to a series of nested blocks in a naive way. This is trivial. However, it does not lead to a proof that a user will want to see, as there are many duplicate lines and boxes that are not necessary.
– The second phase is to *reduce* the proof. We use a rewrite system for this that eliminates various unwanted structures from the proof:
  • If a subproof has no new assumptions nor new variables, the block for it is not needed and can be flattened into the main proof.
  • Lines that are copies of earlier lines can generally be removed, as references to those lines can be replaced by references to the earlier lines.
  • 'Cuts' also can be removed from the proofs, as the declarative proofs really have a cut (in the Gentzen sense) at every line.

Below we will give the details of this rewrite system for proofs for the specific case of first order logic. We prove it to be terminating and confluent.

Our method is designed to convert proofs preserving the level of detail present in the original proof. When building a proof using automated tactics (decision procedures), the user might be curious after the proof that those tactics constructed internally. This is analogous to the rare occasion that a compiler user wants to see the machine code that was generated by the compiler. Our method does not work well for obtaining information on this level. However, Coq allows decomposition of tactics into smaller tactics using the `info` prefix, which means that getting such information is possible even when using our approach.

There have been various projects for translating proofs from a procedural proof assistant into a declarative presentation, most notably the HELM system by Asperti et al., which was further developed in the MoWGLI project [1, 2]. However, those systems almost always work on the level of proof terms and not on the level of tactics. For this reason the declarative proofs that these systems produced tend to be too convoluted for human consumption.

An exception is the system by Guilhot, Naciri and Pottier where Coq proofs are considered on the level of the tactics, by converting Coq proof trees just like we do [6]. However in this work the generated text is only considered a *presentation* – they call it an *explanation* – and not a proof in a formal system like Fitch-style natural deduction in its own right.

Geuvers and Nederpelt [4] define a translation of natural deductions in Fitch style to simply typed λ-terms (i.e., their translation goes the opposite way from ours). They present reduction relations for Fitch-style deductions that allow simpler λ-terms to be obtained. These reductions remove unnecessary subproofs, remove repeats and unshare shared subproofs. They prove that Fitch deductions are mapped to the same λ-term if and only if they are equal under these relations; which shows that there is an isomorphism between these classes.

Proof nets [5] allow representing proofs in a geometrical way where the order of the application of rules as well as irrelevant features of regular natural deduction proofs can be eliminated. Geuvers and Loeb [3] show the correspondence between deduction graphs and proof nets and give translations from minimal propositional logic to proof nets via context nets. They also shows how an operation of cut elimination in deduction graphs can be performed after the translation to a context net.

2 Translating minimal logic tree style proofs to flag style proofs

We first will restrict ourselves to minimal propositional logic. We introduce a translation operation ($\longmapsto$) that translates a tree style proof $\mathbf{G}$ of a proposition A to a flag style proof $\mathbf{F}$. An example of such a translation is:

$$\emptyset : \dfrac{\dfrac{[A]^x}{B \to A}\,{\to}\mathrm{i}[y]}{A \to B \to A}\,{\to}\mathrm{i}[x] \quad \longmapsto \quad \begin{array}{lll} 1 & \vert\ \ \vert\ A & \\ 2 & \vert\ \ \vert\ \vert\ B & \\ 3 & \vert\ \ \vert\ \vert\ A & \text{copy } 1 \\ 4 & \vert\ \ \vert\ B \to A & \to\text{i } 2\text{–}3 \\ 5 & \vert\ A \to B \to A & \to\text{i } 1\text{–}4 \end{array}$$

This operation always preserves the conclusion, and the conclusion will be most often the part of the proof that we match, so we write it explicitly:

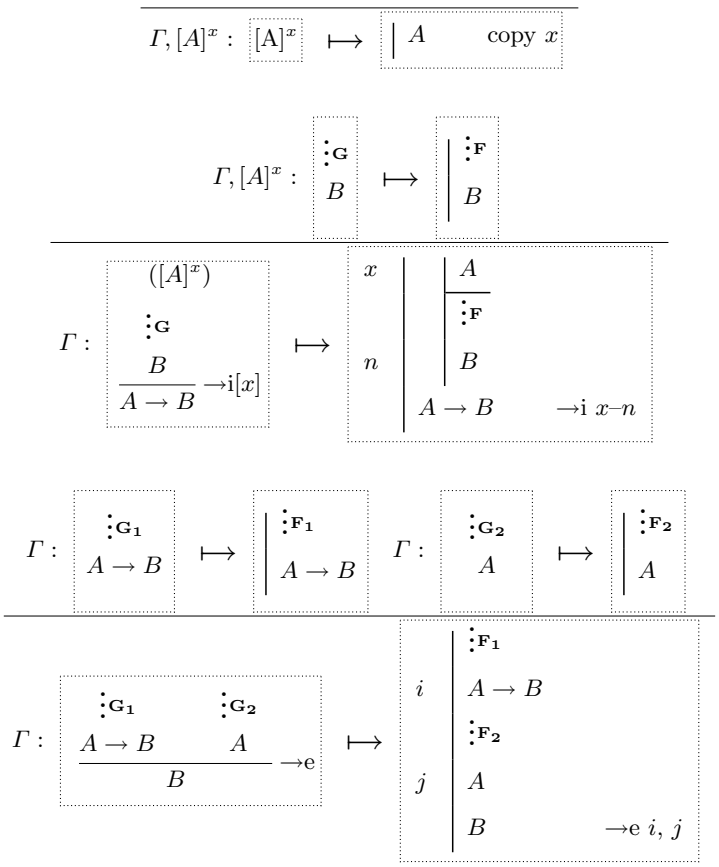
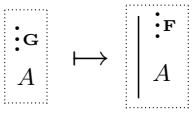
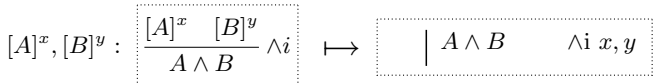


Fig. 2. The translation rules for minimal propositional logic.



The translation operates in a context Γ. This context is a list of assumptions accompanied by labels that can be used in the proofs **G** and **F**. The assumptions that are discharged in the proof are no longer in the context. Sometimes for clarity we will mark assumptions available in particular branches of proofs and discharged after by additional brackets. Below we give an example of a translation of proof styles in a non-empty context:



We define the translation operation inductively via the translation rules in Figure 2. The translation rules match the conclusion and the rule used and give a rule to build the flag style proof. All new labels introduced by the translation operation are fresh identifiers. The usual presentation of flag style proofs is with line numbers and rules that reference those numbers, but in our translation we will use identifiers. An implementation may render such proofs with lines numbered in the customary way, and we do indeed provide this in our ProofWeb implementation as described in Section 6.

The first rule translates the use of an assumption. We replace the use of an assumption with a `copy` line, and label this line with the name of the assumption variable.

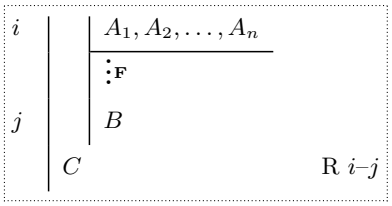
If the derivation ends with implication introduction we translate it to the implication introduction rule in the flag style. We use the name of the introduced assumption in the tree style as the label of assumption line in the flag style. The assumption $[A]$ is not in the context since it is discharged, but for readability we mark it in brackets in the tree style proof. This means that the proof $\mathbf{G}$ can use this assumption. We provide fresh identifiers for new lines.

Implication elimination is analogous. We do not need to introduce a flag for the subtree of the tree style proof. This is what makes the depth of flag proofs much lower then the depth of tree style proofs.

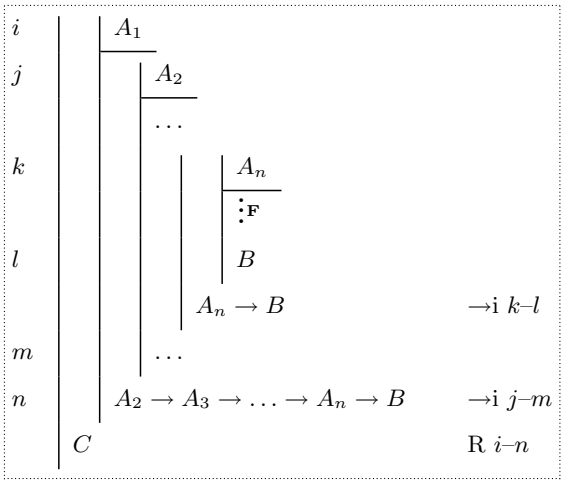
3 Translating proofs in more complicated logical systems

To translate a proof in tree style of an arbitrary deduction system we will first translate it to a non-optimized proof.

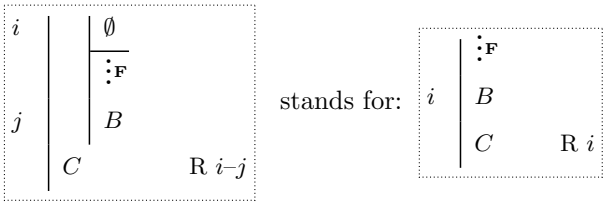
We often need to open a number of flags depending on a list of assumptions. This is why we introduce a shorthand notation. We will write flags with a *list* above the assumption line to denote opening a number of flags. The last flag is opened with the rule provided in the shorthand notation, while all other flags are introduced one by one using implication introduction:



This stands for:



For a list with just one assumption this is equivalent to opening one flag with just the given rule. For a flag with an empty list of assumptions no flags need to be opened:



We show the translation of a given tree style proof in terms of a general schema. This schema will be instantiated for every rule of the logic. Given a rule R that proves the formula B from the tree style proofs $G_1, G_2, \ldots, G_n$ that have conclusions $A_1, A_2, \ldots, A_n$, which discharge assumption lists (possibly empty) $S_1, S_2, \ldots, S_n$ we recursively translate all subproofs to generate the final flag style proof (Figure 3). The subproofs $A_1, \ldots, A_n$ can use the assumptions from their appropriate lists and this is marked in the schema by brackets. An example of instantiation of the schema for a rule for is given in Figure 4.

4 Simplification of obtained proofs

We can remove many of the copy lines by 'path compression', i.e., if a copy line is not the last line under a flag, the copy line can be removed and all further references should be renumbered to refer to the line that was copied:

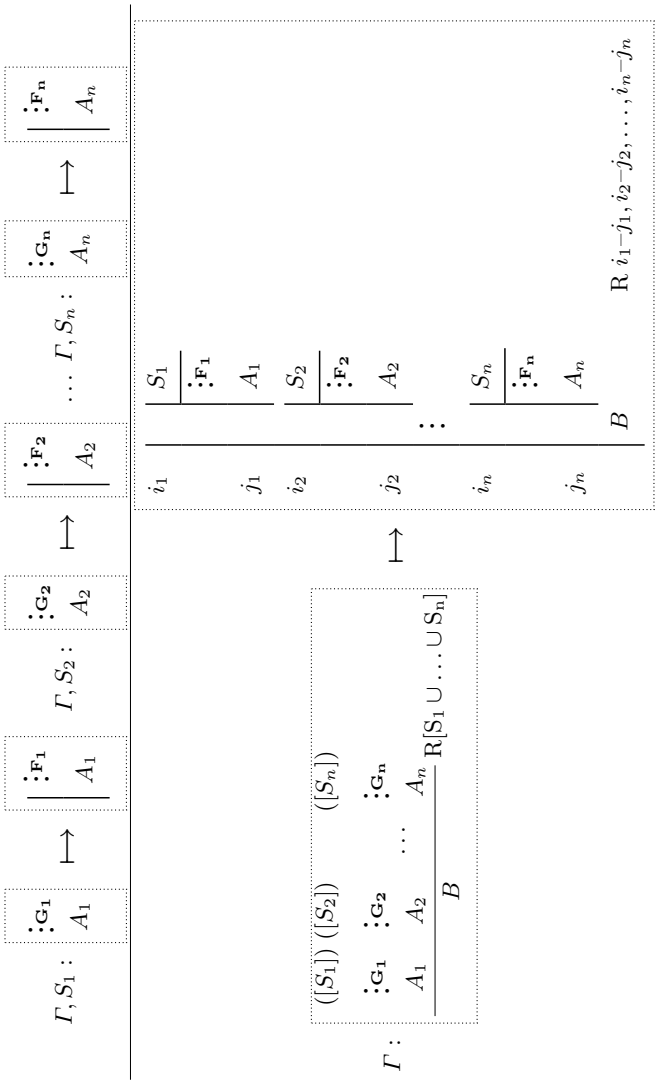


Fig. 3. The general schema for translating a rule of the logic.

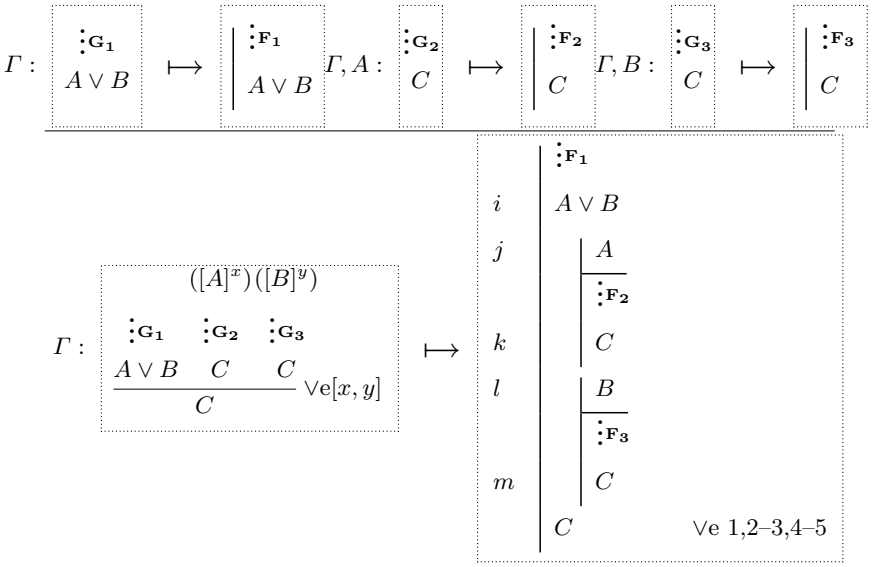
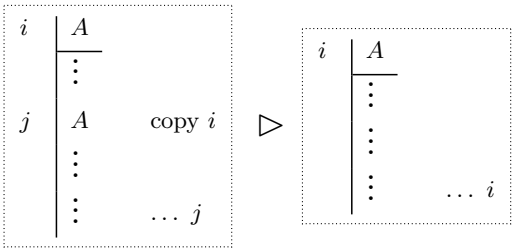
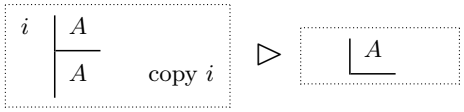


Fig. 4. Example of the general schema instantiated for ∨-elimination.



If the copy line is the only line under a flag and is the copy of the assumption introduced under this flag, the copy line can be removed. This creates proofs that resemble customary Fitch deduction drawing style:



Theorem 1. *The use of the translation followed by performing the above simplifications on a correct Gentzen style natural deduction proof results in a flag style proof that is a correct Fitch style natural deduction proof with the same conclusion and the same rules.*

Proof (Sketch). The conclusion and the rules are preserved by all steps of translation and simplification. The simplifications do not change any of the rules or lines they operate on. The translation of any correct Gentzen rule is a correct Fitch rule. The proof proceeds by verifying the correctness of the translation of all natural deduction rules from [9]. □

5 Simplification of forward proofs

One of the main advantages of flag style proofs over tree style proofs, is that the flag proof is typically almost linear, with very little nesting and therefore much easier to present on paper. For completed natural deduction derivation the proof that we obtain by translation is mostly flat, with nesting introduced only for assumptions. Our translation is also able to work with incomplete proofs. For incomplete proofs done in a backwards manner (starting from the conclusion) the tree style proof corresponds naturally to the flag style proof. This is not the case for forward proofs. For example in tree style:

$$
\begin{array}{lll}
i & A & \\
\hline
j & \quad B & \\
\hline
k & \quad A \wedge B & \wedge\mathrm{i}\ i, j \\
 & \quad \vdots & \\
 & \quad C &
\end{array}
$$

The line labeled k is obtained by $\wedge$-introduction from lines i and j. To represent this proof in Gentzen style natural deduction we need a cut with a branch where $A \wedge B$ is an assumption:

$$
\dfrac{\dfrac{[A]\ \ [B]}{A \wedge B}\wedge i \qquad \dfrac{\begin{array}{c}([A \wedge B]^x) \\ \vdots \\ C\end{array}}{A \wedge B \to C} \to i[x]}{C} \to e
$$

The cut in the above proof cannot be eliminated until the proof is completed. However, this is not the case for flag style proofs, where this kind of cut can be eliminated without influence on the rest of the proof (assuming the rest of the proof is translated as well).

We want to give a mechanism that allows translating the above tree style proof with a cut to a flag style proof without a cut. The use of cut is a general technique; it is often used for inserting a subgoal that can be used further in the proof. This is why we will eliminate all the implication cuts that could have been obtained in this way. To do this we present the rewrite rule in Figure 5, which can be applied only if line l is not used further in the proof.

Theorem 2. *The rewrite system including the above rewrite rule terminates.*

Proof (Sketch). By induction on the number of flags. □

Theorem 3. *The rewrite system including the above rewrite rule is confluent.*

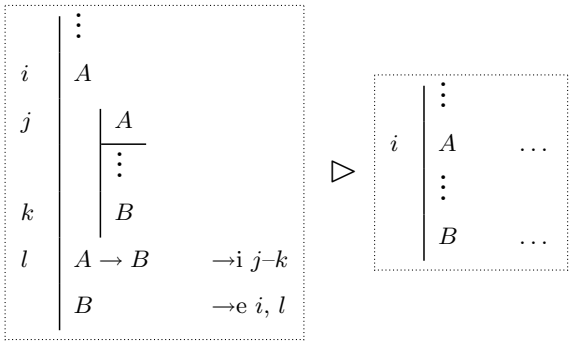


Fig. 5. Rewrite rule for eliminating explicit cuts from a Fitch deduction.

```
                  ...
                 ──────
                 ∃x, P x                                  [∀x, ∃y, (P x ∨ P y)]F
──────────────────────────────────── →i[F1] ──────────────────────────────────── ∀e
(∃y, (P x0 ∨ P y)) → ∃x, P x                  ∃y, (P x0 ∨ P y)
──────────────────────────────────────────────────────────────────────────────── →e
                                 ∃x, P x
──────────────────────────────────────────────────────────────────────────────── →i[F]
                  ∀x, ∃y, (P x ∨ P y) → ∃x, P x
```

```
1 ┌ F:  ∀x, ∃y, (P x ∨ P y)                assumption
2 │ F1: ∃y, (P x0 ∨ P y)                   ∀e 1
  │     ...
3 │     ∃x, P x                                        ┐
4       ∀x, ∃y, (P x ∨ P y) → ∃x, P x  →i 1-3          ┘
```

Fig. 6. An incomplete proof in Gentzen natural deduction and its translation to a Fitch deduction, as rendered by the implementation.

Proof (Sketch). If it is possible to apply a rule at two places in a proof, the two places are associated with two flags. Either one of the flags is under the other or they are in separate parts of the proof and thus independent. If one of the flags is under the other, it has to be inside the incomplete proof part of the rewrite rule. In that case the rewrite only moves the whole incomplete proof and thus the rewrites also are independent. □

We see in Figure 6, how the application of this rewrite rule makes the translation of a Coq proof from a Gentzen tree style proof into a flag style proof with a small number of nested flags.

6 Implementation for Coq proofs

The implementation of Coq keeps a proof tree. This is a recursive OCaml structure, that holds a goal, a rule to obtain this goal from the subgoals, and the subgoals themselves. It is not just a tree structure, since a rule can be a compound rule that contains other proof states. Tactics and tacticals modify the proof state. Coq includes commands for inspecting the proof state. `Show Tree` shows the succession of conclusions, hypotheses and tactics used to obtain the current goal and `Show Proof` displays the CIC term (possibly with holes). The output of these commands was not sufficient to transform the proof state in other formats. We added a new command `Dump Tree` to Coq that allows exporting the whole proof state in an XML format. An example of the output of the `Dump Tree` command for the Coq example from Section 1 is:

```
<tree><goal><concl type="forall n : nat, n <= n"/></goal>
  <cmpdrule><tactic cmd="intros"/>
    <tree><goal><concl type="forall n : nat, n <= n"/></goal>
      <cmpdrule><tactic cmd="intros"/>
        <tree><goal><concl type="forall n : nat, n <= n"/></goal>
          <rule text="intro n"/>
          <tree><goal><concl type="n <= n"/><hyp id="n" type="nat"/>
        </goal></tree></tree></cmpdrule>
      <tree><goal><concl type="n <= n"/><hyp id="n" type="nat"/>
    </goal></tree></tree>
  </cmpdrule><tree><goal><concl type="n <= n"/><hyp id="n" type="nat"/>
</goal></tree></tree>
```

This is the proof tree that corresponds to the incomplete Fitch proof on page 4.

The communication between ProofWeb and Coq is very narrow. The `Dump Tree` command is all that had to be added to Coq to allow our system to convert proofs, and its implementation only took a small amount of OCaml code. This code has now been integrated into the Coq code base, which means that the `Dump Tree` command will be standardly available in Coq from version 8.2.

Our system is intended to be used with simple tactics that correspond to the inference rules of first order logic, so currently we forget the information generated by automated tactics (the content of compound rules). We first transform the tree to a non-optimized flag proof. For every node of the Coq tree we create a new flag. This flag first contains all the assumptions. The notation presented in the previous sections where a flag is allowed to have an arbitrary number of assumptions is also used in the implementation; at a later step this gets translated according to the meaning of the notation. Then if a tree has subgoals, the transformed subgoals are attached. Otherwise, if the goal is not proved ellipses are attached. Finally the flag contains a line for the conclusion of the Coq node.

When rendering a flag style proof that was translated from a tree style proof done with the Coq tactics, the tactic names are printed in a special way. For tactics that match natural deduction rules, the names are changed to their natural deduction names. Furthermore we add the consecutive line numbers on the left

of assumption lines and conclusion lines. We then replace references to labels with the appropriate numbers.

As an example of a flag style version of a serious Coq proof, consider the following proof from the Coq standard library:

```
Lemma leb_complete : forall m n:nat, leb m n = true -> m <= n.
Proof.
  induction m. trivial with arith.
  destruct n. intro H. discriminate H.
  auto with arith.
Qed.
```

This proof is rendered by ProofWeb as:

```
1        ∀n:nat, leb 0 n = true → 0 <= n                trivial[with,arith]
2  m:    nat                                            assumption
3  IHm:  ∀n:nat, leb m n = true → m <= n                assumption
4  H:    leb (S m) 0 = true                             assumption
5        S m <= 0                                       discriminate[H]
6        leb (S m) 0 = true → S m <= 0                  intro[H] 4-5
7  n:    nat                                            assumption
8        leb (S m) (S n) = true → S m <= S n            auto[with,arith]
9        ∀n:nat, leb (S m) n = true → S m <= n          destruct[n] 6,7-8
10       ∀m:nat, ∀n:nat, leb m n = true → m <= n        induction[m] 1,2-9
```

7 Conclusion

The future work of this paper is to develop a *luxury* proof interface, as described in Section 1, for a serious proof assistant. The main difference with the ProofWeb system will then be that the system can also *input* a declarative proof. The declarative proofs then becomes the text that the user works on.

We implemented a rough prototype of a luxury proof language for the HOL Light system, and the approach seems to work quite well there. Currently we are redoing this system in a more systematic and structured manner. This experiment shows that our approach for converting procedural proofs into declarative proofs is not tied to any Coq specifics. It works just as well, and in exactly the same way, in a HOL environment.

A difference with ProofWeb will be to have one further rewrite rule for proofs. In the declarative language of the Mizar system there exists the `consider` statement that is used for existential elimination. If one knows that there exists an x that satisfies $P[x]$, one can write:

```
proof
  ...1
  consider x being A such that P[x] by ...2;
  ...3
  thus Q by ...4;
end;
```

This can be seen as a condensed version of

```
proof
  ...1
  proof
    let x be A;
    assume P[x];
    ...3
    thus Q by ...4;
  end;
  thus Q by ...2;
end;
```

In the case of the ProofWeb system we did *not* want the system to rewrite the latter to get the structure the former, as it would not leave Fitch-style proofs the way that student users would expect them to be. However, in a system for significant formalizations, an optimization like this will be essential.

We claim that a *luxury* proof interface – that is, an interface in which the user edits a declarative proof, but also can ask the system to extend that proof by executing tactics – combines the best of the procedural and declarative worlds. We expect that it will be straight-forward to implement such an interface using the methodology presented in this paper.

References

1. A. Asperti, L. Padovani, C. Sacerdoti Coen, F. Guidi, and I. Schena. Mathematical Knowledge Management in HELM. *Annals of Mathematics and Artificial Intelligence, Special Issue on Mathematical Knowledge Management*, 38:1–3, 2003.
2. A. Asperti and B. Wegner. MOWGLI – A New Approach for the Content Description in Digital Documents. In *Proceedings of the Nineth International Conference on Electronic Resources and the Social Role of Libraries in the Future*, volume 1, 2002.
3. H. Geuvers and I. Loeb. From Deduction Graphs to Proof Nets: Boxes and Sharing in the Graphical Presentation of Deductions. In R. Kralovic and P. Urzyczyn, editors, *MFCS*, volume 4162 of *LNCS*, pages 39–57. Springer, 2006.
4. H. Geuvers and R. Nederpelt. Rewriting for Fitch Style Natural Deductions. In V. van Oostrom, editor, *RTA*, volume 3091 of *LNCS*, pages 134–154. Springer, 2004.
5. J.Y. Girard. Linear Logic. *Theor. Comput. Sci.*, 50:1–102, 1987.
6. F. Guilhot, H. Naciri, and L. Pottier. Proof explanations: using natural language and graph view, 2003. Slides for a talk at a MoWGLI presentation.
7. J.R. Harrison. Proof Style. In E. Giménez and C. Paulin-Möhring, editors, *Types for Proofs and Programs: International Workshop TYPES'96*, volume 1512 of *LNCS*, pages 154–172, Aussois, France, 1996. Springer-Verlag.
8. M. Huth and M. Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge University Press, 2nd edition, 2004.
9. C. Kaliszyk, F. van Raamsdonk, F. Wiedijk, H. Wupper, M. Hendriks, and R. de Vrijer. Deduction using the ProofWeb system. Technical Report ICIS–R08016, Radboud University Nijmegen, September 2008.