# Clausification of First-Order Formulae, Representation & Correctness in Type Theory

Dimitri Hendriks

Summer 1998

# Preamble

Ever since my first undergraduate days (autumn 1993) I have been interested in automated and computer-aided reasoning. After I was provided with the necessary tools for guiding my thoughts within the scientific frame of Cognition & Artificial Intelligence, I 'specialized' in the subject of Logic and Automated Deduction. Let me spend some words on these topics and their interrelationship, without the intention to be complete.

Artificial Intelligence (AI) can roughly be defined as the study of computations required for action, perception and reasoning. On one side, AI is to be thought of as the project of building machines to perform tasks that would 'usually' be taken to require human intelligence. On the other side, intelligent programs can serve as a model for human intelligence and cognition. AI encourages the idea of intelligence and cognition as being matters of computational information processing.

One of the key theoretical pillars of Cognitive Science is Logic. Logic is a formalism to represent, analyse, verify and reason about knowledge and involves the manipulation of abstract symbols. Different logics have been designed, evolving from different philosophical traditions. The fundamental dichotomy in Logic is between *truth* and *proofs*, or, by Frege: between *denotation* and *sense*.

Classical logic[1] has been concentrating upon denotation only and corresponds to the platonic perspective of an external, stable and infinite mathematical universe. Statements about this universe are simply true or false and this leaves no room for proofs.

Intuitionistic logic, the basis for constructive mathematics and type theory [5, 1], employs the notions of proof and computation as more primitive than those of truth and denotation. Proofs correspond to programs and actions, whereas denotation is only the *result* of an action. To a constructivist, logical disjunction is *effective*; we can only state $\phi \vee \psi$ if we exhibit a proof of $\phi$ or of $\psi$. Hence, he cannot accept the classical principle of Excluded Middle (EM, the universal $\phi \vee \neg\phi$), for that would mean that we have a method to solve (i.e. to prove or refute) all existing, as well as future mathematical problems. Judgement, e.g. a theorem, is always the conclusion of a proof.

Still, one has to admit, classical logic is *the* mathematical practice. A mathematician often makes use of the refutation strategy called *reductio ad absurdum*; if assuming the negation of a certain statement leads to absurdity, then that statement must be true (a direct consequence of EM).

An important inference scheme is *resolution* [6]; basically, it comes down to: $\phi_1 \vee \phi_2, \neg\phi_1 \vee \phi_3 \vdash \phi_2 \vee \phi_3$. A general precondition for its application to a first-order formula is that the formula is in *clausal form*. The transformation of formulae into their classical equivalent clausal forms is called *clausification* and will be the focus of the present study.

Automated Deduction (AD) is concerned with the mechanization of deductive processes; as such, it clearly is a subarea of AI. Its ultimate (but still distant) goal

[1] See [8] for a good introduction.

can said to be the full automatization of the proof abilities of a mathematician. Program synthesis, proof discovery and formal verification of correctness of programs are major AD-topics.

A succesful first-order theorem prover, in which an autonomous mode is available, is Otter [7]. It features inference rules (variations/generalizations of the above resolution scheme) that take a small set of clauses and infer a (possibly) new clause that can be used for subsequent inferences. Otter is implemented in the imperative language C and is therefore—I state without motivating—prone to errors. Proofs are not always found, and if they are, it is hard to access the way they are built up.

Another AD-system is Coq [2]. Coq is a powerful proof development tool, based on the *formulae as types & proofs as terms* paradigm. In this paradigm, a formula is a type in a typed lambda calculus (here: the Calculus of Inductive Constructions) and proofs of this formula are lambda terms of the corresponding type. The familiar analogue is:

- $\pi$ is a proof of $A$;
- $\pi$ is a program satisfying specification $A$.

Coq is an interactive assistant in the sense that the user constructs a proof step by step, while the system checks the correctness of each step. The constructed proof is fully explicit as lambda term and is guaranteed to be error-free. Proofs can even be printed in natural language! In contrast to Otter, Coq does not feature a complete mechanization of classical first-order logic.

To combine the best of both systems, Marc Bezem proposed to 'use Otter in Coq' (spring 1997). This project can roughly be divided into two stages:

- Build a clausification construction and formally verify its correctness;
- Make a syntax-based translation program, that exports clauses (obtained from the above construction) to Otter and converts resolution proofs to lambda terms.

The gains will be the automation of the resolution method in Coq and therewith the verifiability of proofs obtained by that method. The price to be paid for the proposed theorem proving procedure: it may invoke unnecessary applications of classical logic.

The first part of the project has been carried out successfully; the present paper reports its results.

The complete clausification program and the proof of its correctness can be found on the following webpage[2]:

`http://www.phil.uu.nl/~bezem/Coq/clausification.v`

## Acknowledgements

Foremost, I thank Marc Bezem for his instructive supervision and for giving me the oppurtunity to get an impression of scientific life. Our collaboration has led to [3]. Thanks also to Erik Barendsen for criticizing earlier versions of this paper. Thanks both to Marc and Erik as well as Peter Lucas for their willingness to referee this thesis.

Dimitri Hendriks
Utrecht, August 1998

[2]The compiled version (extension `vo`) is available too.

# Contents

# Chapter 1

# Introduction

The proof generation capabilities of proof construction systems such as Coq, based on type theory, could still be improved. Resolution based theorem provers such as Otter are more fully automated, but have the drawback that the constructed proof objects are not explicitly available.

Whilst there always is a reason to doubt about the correctness of the outcome of an Otter proof session[1], Coq is a reliable system. In Coq, proofs are automatically verified and are, in effect, well-typed programs. On the other hand, proof generation in Coq suffers from the small granularity of the inference steps and the corresponding astronomic size of the search space. The combination of both systems ('using Otter in Coq') would enhance the readability of resolution proofs and increase Coq's level of automation.

The ideal procedure would be as follows. Identify a non-trivial step in a Coq session that amounts to a first-order tautology. Export this tautology[2] to Otter, and delegate its (refutation) proof to the Otter inference engine with all its clever handles such as strategies, weights, the hot-list, and so on. Convert the resolution proof to type theoretic format and import the result back in Coq.

A general precondition for the application of resolution is that the input formula is of a certain format, called *clausal form*. In Otter we cannot expect to find a precise formal relation between a formula and its clausal form. Therefore, it is required that the clausification procedure is programmed in Coq and that its correctness is formally verified in Coq. The goals of this exam can be taken to be the actual making of such a program as well as the formal verification of its correctness.

The present results are contiguous to the project of programming resolution in Coq. At the same time, they constitute the formal verification of the correctness of clausification, which has never been done before.

Most of the necessary metatheory is already known. The prenex and conjunctive normal form transformations can be axiomatized by classical logic. Skolemization can be axiomatized by so-called Skolem axioms, which can be viewed as specific instances of the Axiom of Choice. Higher order logic is particularly suited for this axiomatization: we get logical equivalence modulo classical logic plus the Axiom of Choice, instead of weaker invariants as equiconsistency or equisatisfiability in the first-order case.

The automation of the clausification part of the project has been carried out and will be described in the sequel. Converting resolution proofs to lambda terms is a parsing and code generation problem of manageable difficulty. This is planned as the next step in the project. Furthermore, by adapting a result of Kleene, Skolem functions and -axioms can be eliminated from resolution proofs, which allows one

[1] In particular, how can we verify the correctness of Otter's clausification method?

[2] More precise: the clausal form of its negation.

to obtain directly a proof of the original formula. This will be the third and final step in the project.

Of course application has to be limited to mathematics that is compatible with classical logic (plus, as yet, the Axiom of Choice). This is the price to be paid for the automated theorem proving procedure that we propose: it may invoke unnecessary applications of classical logic. In particular it is possible that the automated proofs of intuitionistic tautologies are not optimal in the sense that they use classical logic.

To illustrate the differences between both Otter and Coq, we end this introduction with an example. The rest of the paper is organized as follows. In the next chapter it is first considered how to represent first-order logic in Coq. Thereafter, we discuss the developed uniform conversion procedure from first-order formulae to their clausal form; matters are presented in a modular way. In Section 2.4 we discuss the proof of preservation of logical equivalence by this procedure. Next, we present a function (and a corresponding lemma) that makes proving by resolution more convenient. The use of the built constructions is illustrated by the examples in 2.6. In the concluding chapter we discuss some ideas for future work on the project. In Section 3.1 we consider a subset of the inductively defined—as we call them—formal propositions and sketch some nice properties. The first steps towards resolution in Coq are described in 3.2. At last, we demonstrate how Skolem axioms can be eliminated.

## Aperitif: the Drinker's Principle

As running example we use a well-known classical tautology called the Drinker's Principle: *(in every non-empty group of people) there is somebody such that if (s)he is drunk, then everybody is drunk*:

$$\exists x(drunk(x) \Rightarrow \forall y drunk(y))$$

Otter refutes its negation almost instantaneously:

```
-(exists x (drunk(x) -> (all y (drunk(y)))))
```

after first clausifying this into:

```
0 [] drunk(x)
0 [] -drunk($f1(x))
```

where `$f1` is a Skolem function, by the following refutation, with `$F` for $\bot$:

```
1 [] drunk(x)
2 [] -drunk($f1(x))
3 [binary,2.1,1.1] $F
```

The example illustrates well that the clausal form is quite different from the original formulation of the problem. It is left to the reader to make the relation between the original formulation of the problem and its clausal form precise. In the next chapter, a uniform method that makes such relations explicit, is described.

In Coq we have the tactic program session below, to be entered by the user. Here `x:X` expresses the typing relation, to be interpreted as '`x` is an element of `X`' when `X:Set` ('`X` is a set'), and as '`x` is a proof of `X`' when `X:Prop` ('`X` is a proposition'). Furthermore, Coq uses `(M N)` for well typed application, `(x:X)` for universal quantification, `~` for negation and `EX` for existential quantification. The notation `->` stands both for non-dependent function spaces as well as for logical implication. This overloading witnesses the so-called Curry-Howard-De Bruijn isomorphism.

```
Lemma Drinker's_Principle:
((p:Prop)(p\/~p))->(S:Set)(drunk:S->Prop)(s:S)
 (EX x:S | (drunk x)->(y:S)(drunk y)).

Intros EM S drunk s.
Elim (EM (EX z:S | ~(drunk z))); Intro H.

Elim H. Intros z H0.
Exists z.
Intro H1.
Elim H0.
Exact H1.

Exists s.
Intros H0 y.
Elim (EM (drunk y)); Auto; Intro H1.
Elim H. Exists y. Exact H1.

Qed.
```

This tactic program generates a lambda term of about half a page. To improve readability we print the proof using the tool `Natural`, which generates—from the lambda term—the following proof in natural language (automatically!).

```
Theorem : Drinker's_Principle.
Statement : ((p:Prop)(p\/~p))->(S:Set)(drunk:S->Prop)(s:S)
             (EX x:S | (drunk x)->(y:S)(drunk y)).
Proof:
Assume (p:Prop)(p\/~p) (EM) such that a set S
and a term drunk of type S->Prop;
consider an element s of S.
From EM we obtain (EX z:S | ~(drunk z))\/~(EX z:S | ~(drunk z)).
To prove (EX x:S | (drunk x)->(y:S)(drunk y)) we distinguish
two cases.
Case 1.
  Consider an element z of S such that ~(drunk z) (H0).
  We will prove (EX x:S | (drunk x)->(y:S)(drunk y)).
  Suppose (drunk z) (H1).
    Using H0 with the last hypothesis we get False.
    We can assert (y:S)(drunk y).
  So we can assert (drunk z)->(y:S)(drunk y).
Case 2.
  We have ~(EX z:S | ~(drunk z)) (H).
  We will prove (EX x:S | (drunk x)->(y:S)(drunk y)).
  Assume (drunk s) (H0) such that an element y of S.
    From EM we obtain (drunk y)\/~(drunk y).
    To prove (drunk y) we distinguish two cases.
    Case 2.1. (trivial)
      We have (drunk y) (H1).
    Case 2.2. (vacuous)
      We have ~(drunk y) (H1).
      With the last hypothesis we have (EX z:S | ~(drunk z)).
      Using H with this result we get False.
  We have proved (drunk s)->(y:S)(drunk y).
Q.E.D.
```

The example illustrates the differences in style. Otter is real automated theorem proving. The Otter proof is a refutation of the clausal form of the negation of the Drinker's Principle. As it stands, the proof is incomplete with respect to the original statement. In fact, due to the Skolem function `$f1`, the signature of the language has been extended and a different statement has been proved.

On the other hand, Coq is interactive, the proof is detailed and fully explicit as lambda term. It clearly shows the use of classical logic in the assumption `(p:Prop)(p\/~p)` and the polymorphism in `(S:Set)` and `(drunk:S->Prop)`. Moreover, the assumption that the domain is not empty is made explicit by `(s:S)` in the formulation of the lemma.

# Chapter 2

# Clausification

A formula is in clausal form if and only if it is a conjunction of clauses. A clause is a disjunction of literals. Literals are atomic formulae or their negations.

In Otter universal quantifiers are omitted, i.e. variables are implicitly bound. This is not the case with the clausal form computed by our program. Universal quantifiers shall come right before the clauses. Variables that were originally bound by an existential quantifier, are replaced by new terms (Skolem functions applied to a number of universally bound variables). In order to keep logical equivalence (modulo the Axiom of Choice), a higher order existential quantifier that binds the Skolem functions is introduced. See Subsection 2.3.3 for a further discussion. Summarizing, the general format to which any formula is transformed is:

$$\exists f((\forall x_0 \cdots \forall x_n(l_0 \vee \cdots \vee l_k)) \wedge \cdots \wedge (\forall y_0 \cdots \forall y_m(l'_0 \vee \cdots \vee l'_l)))$$

## 2.1 Propositions as formal objects

In order to manipulate propositions as syntactic objects on one hand, and reflect upon them on the other hand, we adopt an ontology consisting of two levels. We distinguish between a higher (interpretative, meta-) level `Prop` and a lower (formal, object-) level `prop`. Such a distinction enables us to:

- Build constructions that account for the necessary transformations of objects on the formal level;
- Prove properties about these objects on the interpretation level.

One of the basic sorts in Coq is `Prop`, the type of logical propositions. An object $M$ in `Prop` denotes the class of terms representing proofs of $M$. `Prop` is the 'language' in which we reason about our formal objects. At the same time, of course, we want the clausification function to be part of an automated procedure that constructs proof terms of some `Prop`-objects (more precise: first-order tautologies).

We have defined `prop`, an inductive set in which first-order propositions can be represented formally. Objects in `prop` will be interpreted in `Prop`. As every inductive set in Coq, `prop` is equipped with higher order primitive recursion as powerful computational device.

Besides the above ontological reasons for a distinct formal level, the universe `Prop` includes higher order propositions, in fact full impredicative type theory, and is as such too large for our purposes. Moreover, Coq supplies only limited computational power on `Prop`. `Prop` alone would not suffice.

To begin with, we need a domain `D`, for predication and quantification:

```
Parameter D : Set.
```

The inductive set of formal propositions is defined by:

```
Inductive prop : Set :=
  f_atom : D -> prop
| f_not  : prop -> prop
| f_and  : prop -> prop -> prop
| f_or   : prop -> prop -> prop
| f_impl : prop -> prop -> prop
| f_ex   : (D -> prop) -> prop
| f_all  : (D -> prop) -> prop.
```

Formal connectives are used in prefix notation. Atomic propositions are built using the first constructor; `f_atom` is the formal counterpart of an arbitrary predicate (`A`). The other names of constructors are self-explanatory. Note that, by definition of the formal quantifiers, we get (bound) variables for 'free'. Variables do not have a distinct status, because any argument of `f_ex` (as well as of `f_all`) is in the arrow type `D->prop`. Although this is a satisfactory representation of quantifiers, there are some diffulties to this approach (see Section 3.1).

For the time being, we have assumed the existence of only one domain of individuals (`D`). In the future we can decide to declare `D` as a formal argument in the definition of `prop`; `prop` then gets type `Set->Set`. Its constructors get an extra type argument too. Note that, by doing so, we can do 'higher order' reasoning, e.g. take `(prop nat)` as the domain and construct formal statements like `(f_all ? [p:(prop nat)](f_atom ? p))` in type `(prop (prop nat))`.[1] Here, 'higher order' is between quotation marks, because it will not be possible to express impredicative statements.

In order to accomodate Skolemization the formal language is extended with a higher order existential quantifier. Therefore we have defined a second formal level (`prop'`) with one constructor `f_Ex`. `SKF` is the type of Skolem functions. A Skolem function has two arguments, an index and a list of `D`-elements and maps these to `D`. The reason for this specific representation of Skolem functions will be explained in Subsection 2.3.3. We now give the definition of polymorphic lists[2], `SKF` and `prop'`; `[x:X]` denotes lambda abstraction.

```
Inductive list [X:Set] : Type :=
   nil: (list X)
| cons: X->(list X)->(list X).

Definition SKF := nat->(list D)->D.

Inductive prop' : Set :=
  f_Ex : (SKF -> prop) -> prop'.
```

We are now able to define a function whose domain is `prop`, by means of a combination of case analysis over the possible constructors and primitive recursion. In Coq, one defines a primitive recursive function by means of the fixpoint construction. This construction is explained by an example at the end of this section.

We proceed by defining `E` and `E'`, canonical mappings that interpret `prop`- resp. `prop'`-objects in `Prop`, by primitive recursion. We declare a unary predicate symbol `A` to form atoms. Again, it should be said, later it can be decided to abstract

[1] The question marks are implicit type arguments; the system automatically infers the correct type (`(prop nat)`) from other arguments.

[2] This construction is reused several times.

from this parameter so as to improve expressivity.[3] For now, let us remark that the limitation to unary predicates only, can be circumvented by an easy tric (cf. Figure 2.2).

```
Parameter A : D->Prop.

Fixpoint E [phi:prop] : Prop :=
Cases phi of
  (f_atom d)   => (A d)
| (f_not p)    => ~(E p)
| (f_and p q)  => (E p)/\(E q)
| (f_or p q)   => (E p)\/(E q)
| (f_impl p q) => (E p)->(E q)
| (f_ex dp)    => (EX x:D | (E (dp x)))
| (f_all dp)   => (x:D)(E (dp x))
end.

Definition E' : prop'->Prop :=
[p:prop'] Cases p of
  (f_Ex skfp) => (EX f:SKF | (E (skfp f)))
end.
```

The function `E`—as well as `E'`—can be viewed as a truth predicate. The classical complications entailed by such a predicate (diagonalization, paradoxes and worse) are properly avoided by Coq. For example, we cannot take `prop` for `D` in the inductive definition of `prop` above, as `D` occurs negatively in the argument types of the constructors `f_ex` and `f_all`. As we already have pointed out, after abstraction from `D` as parameter of the inductive definition, we get `prop` in type `Set->Set` and we cannot apply `prop` to itself for simple typing reasons.

Consider the application of `E` to the Drinker's Principle, which is formally phrased as:

```
Definition f_DP :=
   (f_ex [x:D](f_impl (f_atom x)(f_all [y:D](f_atom y)))).
```

Indeed, applying the interpretation function yields the desired result (read `drunk` for `A`):

```
Eval Compute in (E f_DP).
 = (EX x:D | (A x)->(y:D)(A y)) : Prop
```

In the following, objects in `Prop` as well as those in `prop` will be called 'propositions', and it should be clear from the context which level applies.

## The fixpoint construction

In an untyped functional language, the factorial of a given integer would be defined somehow like:

$$\mathsf{letrec}\ fact = \lambda n.\ \mathsf{if}\ n = 0\ \mathsf{then}\ 1\ \mathsf{else}\ n \cdot fact(n-1)$$

With $\beta$-expansion, this can be rewritten as:

$$\mathsf{letrec}\ fact = (\lambda f.(\lambda n.\ \mathsf{if}\ n = 0\ \mathsf{then}\ 1\ \mathsf{else}\ n \cdot f(n-1)))fact$$

[3] Perhaps this will also involve changing the `f_atom`-constructor and more.

Thus, *fact* is the fixpoint of:

$$\lambda f.(\lambda n.\ \mathsf{if}\ n = 0\ \mathsf{then}\ 1\ \mathsf{else}\ n \cdot f(n-1))$$

If there is a function $Y$ that computes the fixpoint of any function, we can declare that *fact is* the function:

$$Y(\lambda f.(\lambda n.\ \mathsf{if}\ n = 0\ \mathsf{then}\ 1\ \mathsf{else}\ n \cdot f(n-1)))$$

Such a function $Y$ is a powerful computational device in the sense that it allows us to build recursive functions. Unfortunately, $Y$ is not even weakly normalizing[4] and cannot be allowed in a strongly normalizing system, such as the Calculus of Inductive Constructions, the underlying formal language of Coq. However, limited to the case of primitive recursion, Coq provides a construction for computation of fixpoints. This construction has to satisfy constraints as: guardedness, the recursion variables belonging to an inductive type, the function starting with case analysis and recursive calls being done on pattern variables. In Coq our example program is specified as follows; `nat`urals and `mult`iplication are predefined, `S` is the successor function:

```
Fixpoint fact [n:nat] : nat :=
Cases n of O => (S O) | (S m) => (mult n (fact m)) end.
```

## 2.2 Schematic overview of the whole procedure

A given first-order formula $\phi$ in `Prop` will be translated to its corresponding formal counterpart `f_`$\phi$ by a syntax-based translation `Quote` outside Coq (as yet: by hand). We abbreviate `Quote(`$\phi$`)` by `f_`$\phi$. Since objects in `prop` represent certain objects in `Prop`, we have defined above the canonical interpretation function `E` from `prop` to `Prop`. `E` and `Quote` should be such that $\phi$ and the normal form of `(E f_`$\phi$`)` are identical, whenever `Quote` applies. Also, `Quote(E f_`$\phi$`)` has to be identical to `f_`$\phi$. The function `CLAUS`, applied to `f_`$\phi$`:prop`, computes the clausal form of `f_`$\phi$, which is subsequently mapped into `Prop` by `E'`. See Figure 2.1 for the scheme of the clausification process.

Suppose that, during a Coq proof session, a first-order tautology $\phi$ (in `Prop`) is to be proved. First, $\phi$ has to be translated to its corresponding formal counterpart, `f_`$\phi$. Second, the clausification function `CLAUS` is applied to the negation `(f_not f_`$\phi$`)`. (Recall that resolution is a refutation procedure.) Preservation of derivability modulo the principle of Excluded Middle (`EM`), the Axiom of Choice (`AC`) and the condition that `D` is non-empty, is ensured by the following theorem:

```
Theorem CLAUSeq :
   EM->AC->D->(p:prop)(E p)<->(E' (CLAUS p)).
```

The last step is the extraction of clauses from `(CLAUS (f_not f_`$\phi$`))`, which is done by the function `MIMPL`.[5] Applying the function `MIMPL` to `(CLAUS (f_not f_`$\phi$`)))` yields an implication of the form $C_0 \Rightarrow \cdots \Rightarrow C_n \Rightarrow \bot$, where the $C_i$ are the clauses from the clausal form of $\neg\phi$. These clauses can conveniently be introduced in the context. Note that `MIMPL` swaps the polarity of the proposition, so that we are back to the polarity of $\phi$.

If $\phi$ is indeed a tautology, then the clauses obtained in this way are inconsistent. By the completeness of resolution, see for example [6], there exists a resolution refutation of these clauses.

[4] E.g. all reduction paths from $Y(\lambda x.x)$ are infinite.
[5] See Section 2.5.

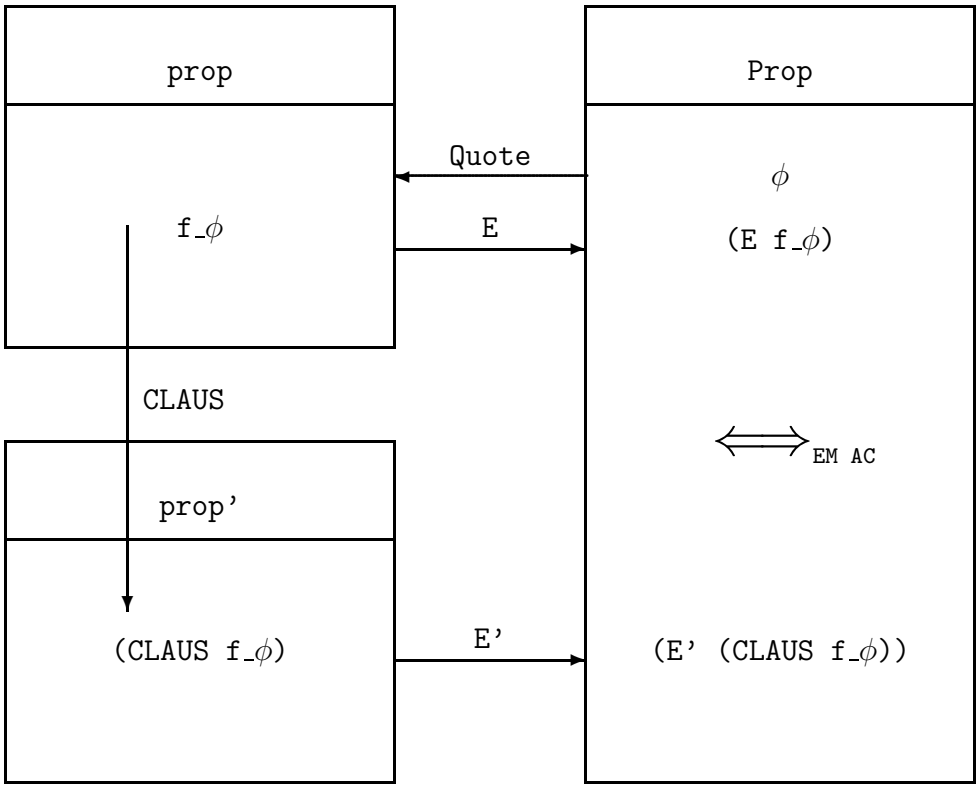


Figure 2.1: Schema of the clausification procedure; the proof of the equivalence in the right box can be generated uniformly in `f_`$\phi$.

## 2.3 The program CLAUS

In the following subsections, we introduce the four subsequent steps in the clausification process. By way of example, Figure 2.2 displays the application of these transformations to a ('meaningless') proposition. We have declared parameters `P` and `Q`, typed as `D->D` and `D->D->D` respectively. They can be seen as (unary, resp. binary) predicates. To be at the reader's beck and call, the propositions are given in traditional notation too. Throughout the paper, we use the abbreviating `ml` to form one-element `D`-lists:

```
Definition ml : D->(list D) := [d:D](cons D d (nil D)).
```

`CLAUS` is defined as the concatenation of the four presented modular functions:

```
Definition CLAUS : prop->prop' :=
   [p:prop](DUC (SKLM (CNF (PNNF p)))).
```

I.e. `CLAUS` applied to the proposition at the top of Figure 2.2 $\beta\delta\iota$-reduces to the bottom one.

### 2.3.1 Prenex and Negation Normal Form

The first function that is applied to the input proposition of the clausification program, is the function `PNNF`. It puts formal propositions into *Prenex* and *Negation Normal Form* in one pass, i.e. the output proposition starts with a quantifier prefix

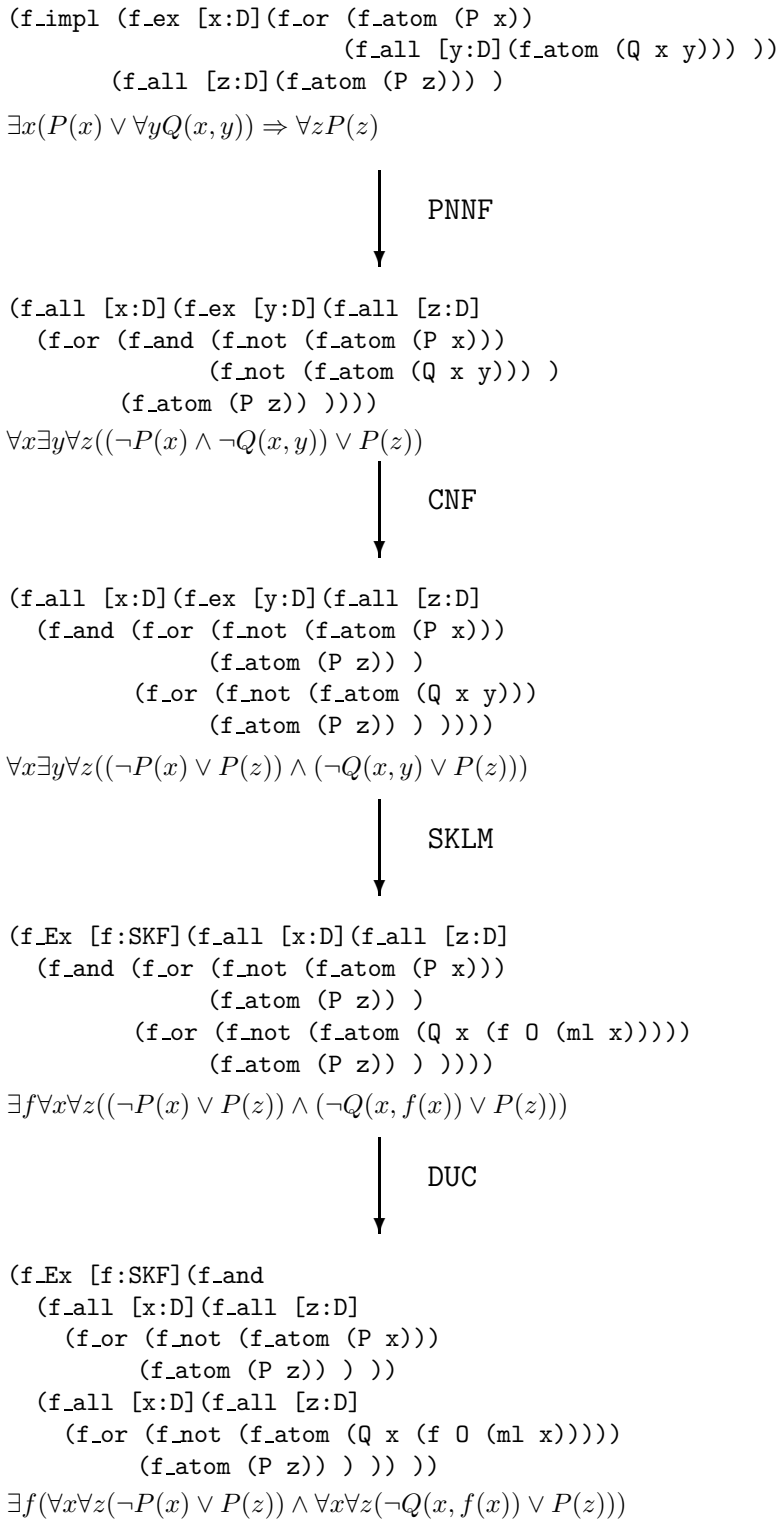


Figure 2.2: Subsequent clausification steps applied to an example.

followed by a quantifier-free matrix containing no implications, and with negations occurring only at the atomic level.

We have chosen to combine the computation of both *NNF* and *PNF*, because this reduces the complexity of the program, without obscuring it. Matters are presented in a top-down manner.

We have defined an inductive set `POL` of polarities consisting of constants `pos` and `neg`:

```
Inductive POL : Set := pos : POL | neg : POL.
```

Given a formula and its polarity, the function `pnnf` returns its prenex negation normal form at the formal level. The main call will be: `(pnnf f pos)`, abbreviated by `(PNNF f)`.

```
Fixpoint pnnf [f:prop] : POL->prop :=
[p:POL] Cases f p of
 (f_atom x)     pos => f
|(f_atom x)     neg => (f_not f)
|(f_not f0)     pos => (pnnf f0 neg)
|(f_not f0)     neg => (pnnf f0 pos)
|(f_and f0 f1)  pos => (pnf_cj (pnnf f0 p)(pnnf f1 p))
|(f_and f0 f1)  neg => (pnf_dj (pnnf f0 p)(pnnf f1 p))
|(f_or f0 f1)   pos => (pnf_dj (pnnf f0 p)(pnnf f1 p))
|(f_or f0 f1)   neg => (pnf_cj (pnnf f0 p)(pnnf f1 p))
|(f_impl f0 f1) pos => (pnf_dj (pnnf f0 neg)(pnnf f1 p))
|(f_impl f0 f1) neg => (pnf_cj (pnnf f0 pos)(pnnf f1 p))
|(f_ex df)      pos => (f_ex [x:D](pnnf (df x) p))
|(f_ex df)      neg => (f_all [x:D](pnnf (df x) p))
|(f_all df)     pos => (f_all [x:D](pnnf (df x) p))
|(f_all df)     neg => (f_ex [x:D](pnnf (df x) p))
end.

Definition PNNF := [p:prop](pnnf p pos).
```

The idea of a proposition being accompanied by its polarity is to bookhold the number of negative 'signs' (an even number results in `pos`; an odd number results in `neg`) during the recursive destruction of that proposition. Efficiency is gained by this tric. A function that moves negations inwards without analysing polarity, would be redundant. Consider e.g. the case of a proposition that starts with two negations: `(f_not (f_not p))`. The recursive call would be on the subformula `(f_not p)`; i.e. the second formal negation is then moved inwards by destructing `p`. Thereafter the first `f_not` would be moved inwards. Compare this to the more economical reduction sequence: `(pnnf (f_not (f_not p)) pos)` $\longrightarrow^*_{\beta\delta\iota}$ `(pnnf (f_not p) neg)` $\longrightarrow^*_{\beta\delta\iota}$ `(pnnf p pos)`.

The function `pnnf` makes use of `pnf_cj` and `pnf_dj`. According to common logical rules, `pnf_cj` moves quantifiers outwards of conjunctions, and `pnf_dj` does the same for disjunctions. Keeping this in mind, let us inspect the above definition of `pnnf` more closely. A given formula `f` is decomposed until the level of literals is reached. It is at this level that the information carried by polarities results into a positive or negative literal. In case the input formula starts with a negation, `pnnf` continues recursively with the subformula and with the polarity swapped. Apart from the `f_impl`-case, the cases with positive polarity are straightforward. The elimination of implications appeals to the following classical equivalences:

- $(F_0 \Rightarrow F_1) \Leftrightarrow \neg F_0 \vee F_1$

- $\neg(F_0 \Rightarrow F_1) \Leftrightarrow F_0 \wedge \neg F_1$

The remaining transformations appeal to the classical De Morgan laws:

- $\neg(F_0 \wedge F_1) \Leftrightarrow \neg F_0 \vee \neg F_1$
- $\neg(F_0 \vee F_1) \Leftrightarrow \neg F_0 \wedge \neg F_1$
- $\neg\exists x F(x) \Leftrightarrow \forall x \neg F(x)$
- $\neg\forall x F(x) \Leftrightarrow \exists x \neg F(x)$

We now continue to discuss `pnf_cj`.[6] Given two propositions `l` and `r` in *PNNF* (in *PNF* as well as in *NNF*)[7], `pnf_cj` produces a conjunction of their matrices prefixed by the pulled-out quantifiers of `l` and `r`:

```
Fixpoint pnf_cj [l:prop] : prop->prop :=
[r:prop] Cases l r of
 (f_ex dl)   _         => (f_ex [x:D](pnf_cj (dl x) r))
|(f_all dl) (f_all dr) => (f_all [x:D](pnf_cj (dl x)(dr x)))
|(f_all dl)  _         => (f_all [x:D](pnf_cj (dl x) r))
| _          _         => (pnf_cj_r l r)
end.
```

The most interesting case is where both arguments start with a universal quantifier. In that case, the quantified variables are unified. This is justified by:

- $\forall x L(x) \wedge \forall x R(x) \Leftrightarrow \forall x(L(x) \wedge R(x))$

In the dual story for disjunctions this is:

- $\exists x L(x) \vee \exists x R(x) \Leftrightarrow \exists x(L(x) \vee R(x))$

The notation `_` is used to exhaust all remaining cases. Those where `l`'s head constructor is a quantifier, `pnf_cj` pulls it outwards and continues recursively. In case `l` does not have a quantifier prefix, the accompanying function `pnf_cj_r` is called; `pnf_cj_r` is recursive in its second argument[8] and moves the quantifiers of `r`'s prefix, if any, to the fore.

```
Fixpoint pnf_cj_r [l,r:prop] : prop :=
Cases r of
  (f_ex dr)  => (f_ex [x:D](pnf_cj_r l (dr x)))
| (f_all dr) => (f_all [x:D](pnf_cj_r l (dr x)))
|  _         => (f_and l r)
end.
```

Thus, if both arguments of `pnf_cj` have empty prefixes, their conjunction is returned.

As yet, we did not minimize the number of arguments of the Skolem functions that will be introduced later on (Subsection 2.3.3). The algorithm can lead to Skolem functions which are more complicated than necessary; it can be optimized in such a way that the Skolem functions have as few arguments as possible. This

[6] A dual story can be told for `pnf_dj`.

[7] Functional languages are intelligible for humans; we can reason top-down: `pnnf` calls `pnf_cj` with two arguments in *PNNF*.

[8] It is a Coq convention that the recursion variable is always the last one listed in the parameter 'list' (here: `[l,r:prop]`).

should be done by moving existential quantifiers out of the scope of universal ones as much as possible. Consider the following conversion:

$$\forall x P(x) \wedge \exists y Q(y) \longrightarrow_{\mathrm{PNNF}} \forall x \exists y (P(x) \wedge Q(y))$$

Logically speaking, the quantifier that binds variable $y$ existentially, does not have to be in the scope of the universal quantifier. [6] may serve as a guide in finding the optimal order for pulling out quantifiers.

For explanatory reasons we presented the different program parts in reversed (i.e. top-down) order. The system should, of course, read/compile `pnf_cj_r` first, then `pnf_cj` (as well as `pnf_dj_r` and `pnf_dj`) and finally `pnnf` and `PNNF`.

### 2.3.2 Conjunctive Normal Form

A formula is in *Conjunctive Normal Form* if and only if its matrix is a conjunction of disjunctions of literals. Again, we work in a top-down way and discuss first the main function: `CNF`. The point of departure is that any input proposition `p` of `CNF` is already in *PNNF*.[9]

Take, for example, the case that `p`'s head constructor is `f_not`. Then we know that we deal with a negative literal, since `PNNF` pushed all negations inwards. In that case `CNF` just returns the unchanged `p`, because literals are in *CNF* already. The cases of literals and implication are represented by `_`.

```
Fixpoint CNF [p:prop] : prop :=
Cases p of
 (f_and p0 p1) => (f_and (CNF p0)(CNF p1))
|(f_or p0 p1)  => (distr (CNF p0)(CNF p1))
|(f_ex dp)     => (f_ex [x:D](CNF (dp x)))
|(f_all dp)    => (f_all [x:D](CNF (dp x)))
| _            => p
end.
```

In case the input formula is prefixed by a quantifier, `CNF` just continues recursively with its subformula. If the head connective of the input proposition is the constructor `f_and`, the result will be a conjunction with its conjuncts transformed into *CNF*. This is easily done with two recursive calls.

The most interesting case is the one where the formula to be put in *CNF* is a disjunction. If one (or both) of the disjuncts contains a (nested) conjunction, the `f_or` is distributed over the conjunctions. This appeals to the following implications:

- $A \vee (B \wedge C) \Rightarrow (A \vee B) \wedge (A \vee C)$
- $(B \wedge C) \vee A \Rightarrow (B \vee A) \wedge (C \vee A)$

We leave this task to the accompanying function `distr`.

Here `max` computes the maximum of two integers; `leqbo` is the usual boolean predicate 'less/equal'. We have assumed an explicit element `ee` in `D` in order to analyze the structure of propositions. We specify `distr` as follows:

```
Fixpoint distr [l:prop] : prop->prop :=
[r:prop] Cases l r of
 (f_and l0 l1) (f_and r0 r1) =>
   (f_and (f_and (distr l0 r0)(distr l0 r1))
          (f_and (distr l1 r0)(distr l1 r1)) )
```

[9] Although we regard the output of `PNNF` to be the input of `CNF`, we will show logical equivalence for all objects of `prop`. See Subsection 2.4.2.

```
|(f_and l0 l1)  _            => (f_and (distr l0 r)(distr l1 r))
| _            _            => (distr_r l r)
end.
```

The arguments of `distr` are disjuncts coming from `CNF` (which performed recursion on them; so they are in *CNF*). The function will never be called with a formula the constructor of which is `f_impl`, `f_ex` or `f_all`, since implications left the scene a while ago and quantifiers are dealt with by `CNF`.

The first mentioned case where both arguments are conjunctions is to be understood as the following conversion:

$$(A \wedge B) \vee (C \wedge D) \longrightarrow_{\texttt{distr}} ((A \vee C) \wedge (A \vee D)) \wedge ((B \vee C) \wedge (B \vee D))$$

The function `distr_r` is recursive in its second argument.

```
Fixpoint distr_r [l,r:prop] : prop :=
Cases r of
 (f_and r0 r1) => (f_and (distr_r l r0)(distr_r l r1))
| _            => (f_or l r)
end.
```

### 2.3.3 Skolem Normal Form

*Skolemization* of a given formula is done as follows. All existential quantifiers are removed and the variables bound by them are replaced by *Skolem functions*. The arguments of these Skolem functions are all universally quantified variables whose quantifier had the (removed) existential quantifier in its scope. To preserve logical equivalence, Skolem functions are quantified by higher order existential quantifiers. For instance:

$$\forall x \exists y \forall z \exists u \, P(x, y, z, u)$$

Skolemizes into:

$$\exists f \exists g \forall x \forall z \, P(x, f(x), y, g(x, z))$$

Note also the possibility of a Skolem constant: a 0-ary Skolem function.

The most natural way to handle the problem of defining a function according the above specification, would be to introduce an index type. At the end of this subsection we explain why the then encountered difficulties could not be overcome.

We chose for a simple solution. Skolemized formulae will (always) be prefixed by just one higher order existential quantifier, stating: *there exists a family of Skolem functions such that....* Within the formula the required information can be found; family-members get a distinguishing integer and a list of variables they depend on. Consider for instance the second clause of the negated Drinker's Principle: `(x:D)~(drunk (f O (ml x)))`, as will be derived in Section 2.6. Here we have a Skolem function indexed by `O` that depends on the list with one element, the universally quantified `x`.

Since input propositions of `SKLM` are already in prenex form, the recursive definition is relatively simple; only the quantifier cases have to be considered:[10]

```
Fixpoint sklm [f:SKF;n:nat;L:(list D);p:prop] : prop :=
Cases p of
 (f_ex dp)  => (* substitute (f n L) for x and increment index n *)
               (sklm f (S n) L (dp (f n L)))
|(f_all dp) => (* add x to the end of argument list *)
               (f_all [x:D](sklm f n (snoc D L x)(dp x)))
```

[10] Coq treats text enclosed between `(*` and `*)` as comment.

```
| _          => (* do nothing *) p
end.

Definition SKLM : prop->prop' :=
[p:prop](f_Ex [f:SKF](sklm f O (nil D) p)).
```

In the first case, the index is incremented and the variable that was bound by the eliminated existential quantifier is substituted by the term `(f n L)`. In the `f_all`-case, the universally quantified `x` is put at the end of the so far constructed list by `(snoc D L x)`. The function `snoc` is defined by:

```
Fixpoint snoc [X:Set;L:(list X)] : X->(list X) :=
[x:X] Cases L of
 (cons head tail) => (cons X head (snoc X tail x))
| nil             => (cons X x (nil X))
end.
```

**Problems with recursively moving out existential quantifiers**

Suppose we had defined an index type for Skolem functions:

```
Fixpoint SKtype [n:nat] : Set :=
Cases n of
   O    => D
| (S m) => D->(SKtype m)
end.
```

Then the idea would be to iterate[11] a function `sklm1`, that moves an existential quantifier in a prefix one step to the left. Corresponding to such a step, the type of the new existential quantifier gets one degree higher, e.g.:

$$\forall x \forall y_1 \cdots \forall y_n \exists z \, P(x, y_1, \ldots, y_n, z)$$

$$\underbrace{\longrightarrow_{sklm1} \cdots \longrightarrow_{sklm1}}_{n \text{ times}}$$

$$\forall x \exists f : \texttt{SKtype}(n) \, \forall y_1 \cdots \forall y_n \, P(x, y_1, \cdots, y_n, f(y_1, \ldots, y_n))$$

$$\longrightarrow_{sklm1}$$

$$\exists g : \texttt{SKtype}(n+1) \, \forall x \forall y_1 \cdots \forall y_n \, P(x, y_1, \cdots, y_n, g(x, y_1, \ldots, y_n))$$

This would have to be based on a new formal level:

```
Inductive SKprop : Set :=
  SK_map : prop -> SKprop
| SK_Ex  : (n:nat)((SKtype n) -> SKprop) -> SKprop
| SK_all : (D -> SKprop) -> SKprop.
```

After definition of a `transl`ation function, the main function `SKLM` would produce some iteration of `sklm1` (specified by `it_nr`) to a given input proposition.

```
Fixpoint transl [p:prop] : SKprop :=
Cases p of
  (f_ex dp)  => (SK_Ex O [f:(SKtype O)](transl (dp f)))
| (f_all dp) => (SK_all [x:D](transl (dp x)))
| q          => (SK_map q)
end.
```

[11] A necessary and sufficient number of times; cf. the iterator as specified in the next subsection.

```
Definition SKLM := [p:prop]
 (it_sklm1 (transl p) (it_nr p)).
```

The troublemaker would be `sklm1`. Remember that `ee` is just a postulated element in `D`:

```
Fixpoint sklm1 [p:SKprop] : SKprop :=
Cases p of
  (SK_all dp)    => Cases (dp ee) of
     (SK_Ex n skfp) =>
       (SK_Ex (S n)[f:(SKtype (S n))](SK_all [x:D]
         Cases (dp x) of
           (SK_Ex m skfp0) => (skfp0 (f x))
         |  _              => p end ))
   | (SK_all dp0)   => (SK_all [x:D](sklm1 (dp x)))
   | _              => p end
| (SK_Ex n skfp) => (SK_Ex n [f:(SKtype n)](sklm1 (skfp f)))
| _              => p end.
```

Let us inspect the reason for failure. Consider the first case (the other cases are trivial recursive continuations). Here it is first checked whether the second constructor of the input formula is `SK_Ex` (by: `Cases (dp ee) of`). If so, the order of quantifiers is swapped and the index type becomes one degree higher. In order to get the binding of `x` right, case-analysis on `(dp x)` is done (!). Of course, `(dp x)` starts—just like `(dp ee)`—with an existential quantifier (i.e. the remaining `_` will never be the case).

The point is: *we* know that `m` equals `n`, but the system does not. Therefore, `(f x)` in type `(SKtype n)` should be the right argument for `skfp0:(SKtype m)->SKprop`. Unfortunately, the system does not accept this, since the type of `skfp` might, in some way, depend on `ee` and could therefore be different from the type of `skfp0`. It remains an intriguing problem.

### 2.3.4 Distribution of Universal Quantifiers

Skolemization resulted in propositions of the form:[12]

$$\exists f \, \forall x_0 \cdots \forall x_k (C_0 \wedge \cdots \wedge C_n)$$

The next step is to distribute universal quantifiers over clauses. In general, the above format shall be transformed to:

$$\exists f ((\forall x_0 \cdots \forall x_k C_0) \wedge \cdots \wedge (\forall x_0 \cdots \forall x_k C_n))$$

Note that this distribution sometimes yields non-dependent products, which, at the `prop`-level, means 'dummy' abstractions. This is because not all $x_i$ occur in all clauses.

The following function distributes the innermost $\forall$ of the $\forall$-prefix over the outermost conjunction of a given proposition. It takes as its arguments two integers `n`, `m` and a proposition `p`, and is recursive in the latter.

```
Fixpoint duc1 [n,m:nat;p:prop] : prop :=
Cases n m p of
  (S O) (S _) (f_all dp) =>
```

[12] Again, we use these 'metasymbols' for the corresponding `prop`-constructors.

```
   (f_and (f_all [x:D] Cases (dp x) of
                       (f_and p0 p1) => p0 | _ => (dp x) end)
          (f_all [x:D] Cases (dp x) of
                       (f_and p0 p1) => p1 | _ => (dp x) end) )
| (S(S k)) (S z) (f_all dp) =>
   (f_all [x:D](duc1 (S k) (S z) (dp x)))
| O (S _) (f_and p0 p1) => (f_and (duc1 (pfx_lh p0)(cdepth p0) p0)
                                  (duc1 (pfx_lh p1)(cdepth p1) p1) )
|  _ _ _ => p
end.
```

Due to the call by the main function `DUC`: the integer `n` in `duc1` represents the length of the ∀-prefix (the outcome of `(pfx_lh p)`) and `m` stands for the depth of conjunctions (which we shall refer to as *c-depth*).

```
Fixpoint cdepth [p:prop] : nat :=
Cases p of
 (f_and p0 p1) => (S (max (cdepth p0) (cdepth p1)))
|(f_all dp)    => (cdepth (dp ee))
| _            => O
end.
Fixpoint pfx_lh [p:prop] : nat :=
Cases p of
  (f_all dp) => (S (pfx_lh (dp ee)))
| _          => O
end.
```

We continue to discuss `duc1`. Of course, the given proposition remains unchanged in case it contains just one clause, i.e. no conjunctions. This is expressed by the condition that `m` is of the form `(S _)`.

Consider the first case. Here we deal with a proposition prefixed by just one universal quantifier. Its second constructor has to be `f_and`, since its matrix is in *CNF*. The result is a conjunction with the quantifier distributed over the conjuncts.

The second case is about a prefix longer than one. The function moves recursively towards the first case, i.e. until it encounters the combination `(f_all [x:D](f_and ...))`. Iteration of `duc1` will be effective due to the third case (!).

The number of times that `duc1` has to be iterated on a certain proposition is exactly the product of its *c-depth* and the length of its prefix.

```
Definition It_nr := [p:prop](mult (pfx_nr p)(cdepth p)).

Fixpoint It_duc1 [p:prop;n:nat] : prop :=
Cases n of
   O    => p
| (S n) => (duc1 (pfx_lh (It_duc1 p n))
                 (cdepth (It_duc1 p n))
                 (It_duc1 p n) )
end.

Definition DUC : prop'->prop' :=
[p:prop'] Cases p of
 (f_Ex skfp) => (f_Ex [f:SKF](It_duc1 (skfp f) (It_nr (skfp f))))
end.
```

Distribution of universal quantifiers over clauses is computationally expensive. The following computation takes Coq more than three minutes:

```
Eval Compute in
(DUC (f_Ex [f:SKF]
      (f_all [x:D](f_all [y:D](f_all [z:D]
       (f_and (f_atom (f O (ml x)))
              (f_and (f_and (f_atom x)(f_atom y))
                     (f_atom z) )))))).
```

Anyway, the answer is correct:

```
(f_Ex [f:SKF](f_and
 (f_all [x:D](f_all [_:D](f_all [_:D](f_atom (f O (ml x))))))
 (f_and (f_and (f_all [x:D](f_all [_:D](f_all [_:D](f_atom x))))
               (f_all [_:D](f_all [x:D](f_all [_:D](f_atom x)))) )
        (f_all [_:D](f_all [_:D](f_all [x:D](f_atom x)))) )))
: prop'
```

## 2.4 Preservation of Logical Equivalence, CLAUSeq

In this section we discuss the preservation of logical equivalence modulo the Excluded Middle (EM), the Axiom of Choice (AC) and the condition that D is non-empty, as expressed by the following theorem:

```
Theorem CLAUSeq :
   EM->AC->D->(p:prop)(E p)<->(E' (CLAUS p)).
```

Let us give the definitions of EM and AC just now:

```
Definition EM := (p:Prop)p\/~p.

Definition AC :=
(S,S':Set)(P:S->S'->Prop)
 ((x:S)(EX y:S' | (P x y)))->
  (EX f:S->S' | (x:S)(P x (f x))).
```

Analogous to the modular set-up of the clausification program, CLAUSeq is supported by four lemmas. In the following subsections their proof constructions are treated. Of course, we will not consider the proof in full detail; redemption of the proof obligation only, already converts one into a monk.[13] The proof of the above theorem is rather complicated, and the exposition may be difficult to follow. The better way to follow the proof is to process the vernacular file clausification.v mentioned in the preamble. The reader who is not primarily interested in the details of this proof may skip this section and rely on the fact that the proof has been type checked by Coq.

Proofs are constructed by a backward chaining method. Coq permits to develop a proof by decomposing the current goal into subgoals. In a given situation, several subgoals may remain to be proved, to each of which a local context is associated.

When proving a logical property for all p in prop, one enumerates all the cases where p starts with a different constructor and one makes use of the Induction Principle, which Coq automatically generated as a $\lambda$-term when prop was defined. We can check its type:

```
prop_ind :
(P:prop->Prop)
 ((d:D)(P (f_atom d)))
```

[13] A new English saying.

```
->((p:prop)(P p)->(P (f_not p)))
 ->((p:prop)(P p)->(q:prop)(P q)->(P (f_and p q)))
  ->((p:prop)(P p)->(q:prop)(P q)->(P (f_or p q)))
   ->((p:prop)(P p)->(q:prop)(P q)->(P (f_impl p q)))
    ->((dp:D->prop)((d:D)(P (dp d)))->(P (f_ex dp)))
     ->((dp:D->prop)((d:D)(P (dp d)))->(P (f_all dp)))
      ->(p:prop)(P p)
```

Note that, `prop_ind`, `CLAUSeq` and the like abbreviate $\lambda$-terms.

### 2.4.1 PNNFeq

To show that the module `PNNF` preserves logical equivalence, we have proved the following lemma:

```
Lemma PNNFeq : EM->D->(p:prop)(E p)<->(E (PNNF p))
```

Thus, `PNNFeq` states that if there is a proof object of `EM` and an element in `D`, then the interpretation of any formal object `p` is logically equivalent to the interpretation of `(PNNF p)`. The proof is supported by three lemmas, whose proofs are lenghty but straightforward.

```
Lemma pnf_cj_eq :
   D->(l,r:prop)(E l)/\(E r)<->(E (pnf_cj l r)).

Lemma pnf_dj_eq :
   EM->D->(l,r:prop)(E l)\/(E r)<->(E (pnf_dj l r)).

Lemma POLeq :
   EM->D->(p:prop)~(E (pnnf p pos))<->(E (pnnf p neg)).
```

It is a well-known fact that first-order models have non-empty domains. We briefly illustrate the need of the hypothesis of `D` being non-empty. In order to show validity of the prenex operation of working universal quantifiers out of a conjunction, we need classical facts like:

- $\forall x(P \wedge Q(x)) \Leftrightarrow P \wedge \forall x Q(x)$[14]

The domain over which is quantified has to contain at least one element, for, in case it does not, distribution of $\forall$ over conjuncts is not valid. This requirement holds for `pnf_dj_eq` too: $(P \vee \exists x Q(x)) \Rightarrow \exists x(P \vee Q(x))$[14] is not valid in case `D` is empty.

The lemma `pnf_dj_eq` makes use of the Excluded Middle. Its application is met at the moment we need the classical $(P \vee \forall x Q(x)) \Leftrightarrow \forall x(P \vee Q(x))$[14].

`POLeq` is of classical nature, since it essentially uses *reductio ad absurdum* to derive (intuitionistically) strong conclusions from weak premisses. Take, for instance, De Morgan's law:

- $\neg(P \wedge Q) \Leftrightarrow \neg P \vee \neg Q$

Constructively speaking, it states that we can make an effective decision based on something which has no proof ($\Rightarrow$). To give an idea of the actual proof development, we check the point at which this law is applied.

The lemma is proved by structural induction on `p`. Consider the situation, where `p` is of the form `(f_and p0 p1)` and we have to prove the equivalence of `~(E (pnnf p pos))` and `(E (pnnf p neg))`, which $\beta\delta\iota$-reduces to the following goal:

[14] $x \notin FV(P)$.

```
 em : EM
  d : D
 p0 : prop
IH0 : ~(E (pnnf p0 pos))<->(E (pnnf p0 neg))
 p1 : prop
IH1 : ~(E (pnnf p1 pos))<->(E (pnnf p1 neg))
============================
 ~(E (pnf_cj (pnnf p0 pos) (pnnf p1 pos)))
 <->(E (pnf_dj (pnnf p0 neg) (pnnf p1 neg)))
```

Now, by using `pnf_cj_eq` and `pnf_dj_eq` (this is why `POLeq` requires `D` being non-empty), this goal can be rewritten as:

```
   ~((E (pnnf p0 pos))/\(E (pnnf p1 pos)))
   <-> ((E (pnnf p0 neg))\/(E (pnnf p1 neg)))
```

The left-hand side of this equivalence can, by applying the above De Morgan's law, be rewritten, obtaining:

```
   (~(E (pnnf p0 pos))\/~(E (pnnf p1 pos)))
   <-> ((E (pnnf p0 neg))\/(E (pnnf p1 neg)))
```

And this is trivially deduced from both induction hypotheses (`IH0` and `IH1`).

The proof follows closely the function definitions: anytime `PNNF` calls a subfunction, we construct an equivalence proof corresponding to that subfunction. Structural induction is performed whenever we deal with a recursion variable; e.g. in the proof of `pnf_cj_eq` the term `prop_ind` was applied to the first argument `l`, since the function `pnf_cj` is recursive in that argument. Analogously, the proof corresponding to the function `pnf_cj_r` is done by induction on the second argument.

### 2.4.2 CNFeq

The following lemma is intuitionistically valid:

```
Lemma CNFeq : (p:prop)(E p)<->(E (CNF p)).
```

Conversion steps for the preservation proof of which we needed classical logic, are already dealt with previously. Again, to prove the equivalence for all `p` in `prop`, structural induction is performed on `p`. Except for the `f_or`-case, the other cases are proved quite easily, using Coq's friendly facilities. The function `distr` preserves derivability:

```
Lemma distr_eq : (l,r:prop)(E l)\/(E r)<->(E (distr l r)).
```

As the proof discussed in the next subsection will be more interesting, we no longer bother the reader with the monk's work.

### 2.4.3 SKLMeq

As we have already outlined in Subsection 2.3.3, the Skolemization algorithm always results in an existential statement:

$$\exists f : \texttt{SKF}\ \forall x_0 \cdots \forall x_k\, M$$

Recall that `SKF` abbreviates `nat->(list D)->D`. Here we find another reason for non-emptiness of the domain: due to our way of encoding Skolem functions using lists, `D` cannot be empty, otherwise Skolem functions (mappings to `D`) would not exist. More precise: if `D` is empty, there are no objects in the arrow type `(list`

D)->D, since (list D) is never empty ((nil D)). This problem does not arise in the usual representation $D^n \to D$, since, if $D$ is empty, $D^n$ is empty too, and there exists a function in $\emptyset \to \emptyset$.

The Axiom of Choice guarantees the existence of such 'non-constructively' defined functions:

```
Definition AC :=
(S,S':Set)(P:S->S'->Prop)
 ((x:S)(EX y:S' | (P x y)))->
  (EX f:S->S' | (x:S)(P x (f x))).
```

As is stated by the following lemma, SKLM preserves derivability modulo the Axiom of Choice and the existence of an element in D.

```
Lemma SKLMeq : AC->D->(p:prop)(E p)<->(E' (SKLM p))
```

Let us inspect some crucial points in the proof construction of SKLMeq. Consider the following subgoal arising in the inductive proof of SKLMeq. (Case f_all, left-to-right half of <->.)[15]

```
ac : AC
 d : D
dp : D->prop
IH : (d:D)(E (dp d))<->(EX f:SKF | (E (sklm f 0 (nil D)(dp d))))
 H : (x:D)(E (dp x))
============================
(EX g:SKF | (x:D)(E (sklm g 0 (ml x)(dp x))))
```

From the context—using H and IH—we can infer a proof object H0 having type:

```
H0 : (x:D)(EX f:SKF | (E (sklm f 0 (nil D) (dp x))))
```

Now, to be able to construct the witnessing Skolem function g for the goal, the existential quantifier in H0 has to be moved outwards. By application of ac to H0[16] we can construct a proof term H1 having type:

```
H1 : (EX F:D->SKF |(x:D)([y:D][f:SKF]
      (E (sklm f 0 (nil D)(dp y))) x (F x) ))
```

which reduces to:

```
H1 : (EX F:D->SKF |(x:D)(E (sklm (F x) 0 (nil D)(dp x))))
```

By performing elimination on H1, we get a function F in type D->SKF and a proof H2 of:

```
H2 : (x:D)(E (sklm (F x) 0 (nil D)(dp x)))
```

The Skolem function g we are looking for has to be such that:

```
(x:D)(E (sklm g 0 (ml x) (dp x)))
```

[15] We think Coq can be improved on two points:

- Subgoals should be indexed according to the hierarchical structure of the proof at construction;
- It should be possible to print a proof that is under construction, with 'holes' for the remaining subgoals to be proved.

[16] Where D is substituted for S, SKF for S' and [y:D][f:SKF](E (sklm f 0 (nil D)(dp y))) for P in AC.

is a logical consequence of H2. This means that for any list L:(list D) and with head x, g must behave just like (F x) does on the tail of L. The witnessing g for the goal is constructed as:

```
Exists [n:nat][L:(list D)]Cases L of
     nil       => d
   |(cons h t) => (F h n t)
end.
```

We continue to name this function g. As stated informally above, the key property of g is:

```
(g n (cons D x L)) = ((F x) n L)
```

for all n,x,L, which can be shown to imply:

```
(E (sklm g O (ml x) (dp x))) <-> (E (sklm (F x) O (nil D) (dp x)))
```

We observe that all we need, is the fact that (F x) and

```
[n:nat][L:(list D)](g n (cons D x L))
```

have the same extensional behaviour. Therefore we have proved the following general lemma:

```
Lemma EEQ :
   (f,g:SKF)(p:prop)(i,j:nat)(Lf,Lg:(list D))
    ((k:nat)(L:(list D))
     (f (plus i k)(concat D Lf L))=(g (plus j k)(concat D Lg L)) )
     ->(E (sklm f i Lf p))->(E (sklm g j Lg p)).
```

This lemma is applied as follows:

```
Apply (EEQ (F x) g (dp x) O O (nil D) (ml x)).
```

Two subgoals are generated; the first of which is trivial:

```
(k:nat)(L:(list D))
 (F x (plus O k) (concat D (nil D) L))
 =(Cases (concat D (ml x) L) of
     nil        => d
   | (cons h t) => (F h (plus O k) t)
       end)
```

after $\beta\delta\iota$-reduction. The second subgoal:

```
(E (sklm (F x) O (nil D) (dp x)))
```

is proved by (H2 x).

Let us inspect another point where EEQ is applied. The next subgoal corresponds to the f_ex-case generated by application of the Induction Principle, right-to-left half of <-> in the statement of SKLMeq above:

```
ac : AC
 d : D
 p : prop
dp : D->prop
IH : (d:D)(E (dp d))<->(EX f:SKF | (E (sklm f O (nil D) (dp d))))
 H : (EX f:SKF | (E (sklm f (S O) (nil D) (dp (f O (nil D))))))
============================
 (EX x:D | (E (dp x)))
```

Given the Skolem function `f` in `H` (which is obtained by elimination on `H`), we have to give a witnessing `x` for the goal: `Exists (f O (nil D))`. Using the `I`nduction `H`ypothesis applied to this `(f O (nil D))`, what remains to be proved is:

```
(EX g:SKF | (E (sklm g O (nil D) (dp (f O (nil D))))))
```

For any index `n`, the Skolem function `g` we have to construct, behaves just like `f` (in `H`) does on the successor of `n`:

```
Exists [n:nat][L:(list D)](f (S n) L).
```

We continue to name this function `g`. The remaining goal:

```
(E (sklm g O (nil D) (dp (f O (nil D)))))
```

follows from `H` by application of `EEQ`:

```
Apply (EEQ f g (dp (f O (nil D))) (S O) O (nil D) (nil D)).
```

### 2.4.4 DUCeq

The one-step distribution function `duc1` preserves derivability:

```
Lemma duc1_eq : (p:prop)(E p)<->(E (duc1 (pfx_lh p) (cdepth p) p)).
```

It does not matter how many times it is applied:

```
Lemma It_duc1_eq : (n:nat)(p:prop)(E p)<->(E (It_duc1 p n)).
```

Finally, the lemma corresponding to the main function:

```
Lemma DUCeq : (p':prop')(E' p')<->(E' (DUC p')).
```

## 2.5 Refutation

The input of the clausification program is going to be the negation of the proposition of which a resolution proof is to be made. The idea of the present step is to put the so obtained clauses in implicative form such that they entail false. Thus, if we have: $\exists f : \texttt{SKF}\,(C_0 \wedge \cdots \wedge C_n)$, we now get: $\forall f : \texttt{SKF}\,(C_0 \Rightarrow \cdots \Rightarrow C_n \Rightarrow \bot)$.[17] Although this transformation is not part of the clausification algorithm, it is described in Chapter 2, because it is about manipulation of propositions as formal objects much in the same way as the previous modules. As will be shown in the next section, the lemma `Refute` embodying this transformation is convenient for use when doing resolution proofs. It will be easier to introduce the clauses as separate hypotheses into the context. Applying `Refute` to a proposition `p` is to be interpreted as: *refute the clausal form of the negation of* `p`.

```
Lemma Refute : EM->AC->D->
   (p:prop)(E'' (MIMPL (CLAUS (f_not p))))->(E p).
```

We introduce a third level of object language (accompanied by an interpreter `E''`), mainly motivated by pragmatic reasons. For now, let us note that in the future we can decide to unify `prop'` and `prop''`.

[17] Yet, the $C_i$ should be understood as disjunctions of literals *with* $\forall$-prefixes; here we call them clauses.

```
Inductive prop'' : Set :=
  bottom  : prop''
| map     : prop -> prop''
| f_impl2 : prop'' -> prop'' -> prop''
| f_All   : (SKF ->prop'') -> prop''.
```

The first constructor, `bottom`, is the formal counterpart of $\bot$; `map` just translates from the first to the third level; `f_impl2` represents implication (on the third level) and `f_All` is the dual of `f_Ex`.

The corresponding interpretation function is defined as:

```
Fixpoint E'' [p:prop''] : Prop :=
Cases p of
  bottom        => False
| (map p)       => (E p)
| (f_impl2 p q) => (E'' p) -> (E'' q)
| (f_All skfp)  => (x:SKF)(E'' (skfp x))
end.
```

In the following we work with lists of formal propositions; one of the used functions is the polymorphic `concat` concatenating two lists of a given type:

```
Fixpoint concat [X:Set;L1:(list X)] : (list X)->(list X) :=
[L2:(list X)] Cases L1 of
 (cons head tail) => (cons X head (concat X tail L2))
| _               => L2
end.
```

The recursive `mLoC` puts all clauses (the conjuncts of the input proposition) in a list:

```
Fixpoint mLoC [p:prop] : (list prop) :=
Cases p of
  (f_and p0 p1) => (concat prop (mLoC p0)(mLoC p1))
| clause        => (cons prop clause (nil prop))
end.
```

In case the input proposition is not a conjunction, we know that we are dealing with a clause (as we consider the output of `CLAUS` to be the input of `MIMPL`). In this way, we can deal with nested conjunctions: clauses are separated as elements of a list.

Given a list of clauses, the following function builds an implication (on the `prop''`-level):

```
Fixpoint mimpl [L:(list prop)] : prop'' :=
Cases L of
  nil                => bottom
| (cons clause tail) => (f_impl2 (map clause)(mimpl tail))
end.
```

The function relies upon right-associativeness of implication. When the list is recursively emptied, `mimpl` produces the consequence `bottom` of the constructed implication.

Finally, the main function of this module is defined. It simply turns `f_Ex` into `f_All` (as the output is classically equivalent to negation of the input).

```
Definition MIMPL : prop'->prop'' :=
[p:prop'] Cases p of
(f_Ex skfp) => (f_All [f:SKF](mimpl (mLoC (skfp f))))
end.
```

We show its working by applying `MIMPL` to the outcome of the pipeline in Figure 2.2 (the output of `CLAUS`); resulting in:

```
(f_All [f:SKF]
 (f_impl2
  (map (f_all [x:D](f_all [z:D](f_or (f_not (f_atom (P x)))
                                     (f_atom (P z)) ))))
  (f_impl2 (map (f_all [x:D](f_all [z:D]
             (f_or (f_not (f_atom (Q x (f 0 (ml x)))))
                   (f_atom (P z)) ) )))
           bottom ) ) )
```

In traditional notation:

$$\forall f\ (\forall x \forall z(\neg P(x) \vee P(z)) \Rightarrow \forall x \forall z(\neg Q(x, f(x)) \vee P(z)) \Rightarrow \bot)$$

We have proved the following lemma:

```
Lemma MIMPLeq : EM->(p:prop')~(E' p)<->(E'' (MIMPL p)).
```

An important lemma that supports `MIMPLeq` is:

```
Lemma splitLeq :
 EM->(L0,L1:(list prop))(E'' (mimpl (concat prop L0 L1)))
  <->((E'' (mimpl L0))\/(E'' (mimpl L1))).
```

It is to be understood as follows. Suppose we have two lists of clauses:

$$L_0 = [C_0, \cdots, C_n],\ L_1 = [C'_0, \cdots, C'_m]$$

Then `splitLeq` states the equivalence of:

1. $C_0 \Rightarrow \cdots \Rightarrow C_n \Rightarrow C'_0 \Rightarrow \cdots \Rightarrow C'_m \Rightarrow \bot$ and
2. $(C_0 \Rightarrow \cdots \Rightarrow C_n \Rightarrow \bot) \vee (C'_0 \Rightarrow \cdots \Rightarrow C'_m \Rightarrow \bot)$.

The implication from 1 to 2 is classically valid only(!).

Finally, we present the lemma stating the preservation of derivability (modulo EM and AC) of the successive application of `CLAUS` and `MIMPL`, of which `Refute` (given above) is a direct consequence:

```
Lemma CMeq :
   EM->AC->D->(p:prop)(E p)<->(E'' (MIMPL (CLAUS (f_not p)))).
```

Actually, it states the classical equivalence (modulo AC) of any first-order proposition $\phi$ and the negation of the clausal form of $\neg\phi$.

## 2.6 Sample applications

In this section we give three examples to illustrate the use of the lemma `Refute` when proving first-order tautologies. The automated clausification procedure facilitates the proofs already. The obtained clauses are resolved by hand.

## Back to the Drinker

The Drinker's Principle is formally represented by:

```
Definition f_DP :=
   (f_ex [x:D](f_impl (f_atom x)(f_all [y:D](f_atom y)))).
```

The principle is then entered as:

```
Lemma Drinker's_Principle : EM->AC->D->(E f_DP).
```

There are some differences between this statement and the one we gave in the introduction. This time the Axiom of Choice is added as hypothesis; another difference: now we have assumed a domain `D` and a predicate `A`, whereas `S` and `drunk` were polymorphic in the earlier statement.

First the hypotheses are introduced into the context:

```
Intros em ac d.
```

Then `Refute` is applied:

```
Apply (Refute em ac d).
```

In the local context of `em:EM`, `ac:AC` and `d:D`, the new goal is:

```
(E'' (MIMPL (CLAUS (f_not f_DP))))
```

This goal is $\beta\delta\iota$-reduced. We consider this to be the actual application of the clausification and refutation program; `Simpl`:

```
(f:SKF)((x:D)(A x))->((x:D)~(A (f O (ml x))))->False
```

Then, the Skolem function and the clauses are introduced:

```
Intros f C1 C2.
```

Thus, we have arrived at the following situation:

```
 em : EM
 ac : AC
  d : D
  f : SKF
 C1 : (x:D)(A x)
 C2 : (x:D)~(A (f O (ml x)))
============================
  False
```

Until here the proof procedure has been fully automated. Due to the simplicity of the example, the proof can now be completed easily by hand:

```
Apply (C2 d).
Apply C1.
Qed.
```

Figure 2.3 represents the above proof by a natural deduction tree.[18]

The lambda term generated by the tactic program from Chapter 1, is even larger than the lambda term constructed above, due to the power of `Refute`. The normal form of the latter is, of course, considerably larger. Interestingly, this cut-free proof does not contain any occurrence of `prop`, see Section 3.1.

It is instructive to compare the clauses `C1` and `C2` with the ones Otter came up with:[19]

[18] Wouldn't it be great if such figures were produced by just clicking the mouse?

[19] At the moment that we had our aperitif.

```
1 [] drunk(x)
2 [] -drunk($f1(x))
```

They are essentially the same. The Skolem function `f` in `C2` may look more complicated, but the term `(f 0 (ml x))` is, after appropriate renaming, the same as `$f1(x)`.

$$\dfrac{\dfrac{\dfrac{\dfrac{[\forall x D(x)]^2}{D(f(d))}\,\forall E \quad \dfrac{[\forall x \neg D(f(x))]^1}{\neg D(f(d))}\,\forall E}{\bot}\Rightarrow E}{\forall x \neg D(f(x)) \Rightarrow \bot}\Rightarrow I_1}{\forall x D(x) \Rightarrow \forall x \neg D(f(x)) \Rightarrow \bot}\Rightarrow I_2 \qquad \begin{array}{c}\texttt{(Refute em ac d f\_DP)}\\ \vdots \\ (\forall x D(x) \Rightarrow \forall x \neg D(f(x)) \Rightarrow \bot) \Rightarrow \exists x (D(x) \Rightarrow \forall y D(y))\end{array}}{\exists x (D(x) \Rightarrow \forall y D(y))}\Rightarrow E$$

Figure 2.3: Natural deduction tree representing the resolution proof of the Drinker's Principle.

## Swap: $\forall x \exists y (P(x) \vee Q(y)) \Rightarrow \exists y \forall x (P(x) \vee Q(y))$

The formal counterpart of this tautology is phrased as:

```
Definition f_swap :=
 (f_impl
   (f_all [x:D](f_ex [y:D](f_or (f_atom (P x))
                                (f_atom (Q y)) )))
   (f_ex [y:D](f_all [x:D](f_or (f_atom (P x))
                                (f_atom (Q y)) ))) ).
```

Here `P` and `Q` are variables in type `D->D`. We enter:

```
Lemma swap : EM->AC->D->(E f_swap).
```

Followed by:

```
Intros em ac d.
Apply (Refute em ac d).
Simpl.
Intros f C1 C2 C3.
```

Resulting in:

```
  em : EM
  ac : AC
   d : D
   f : SKF
  C1 : (x:D)(A (P x))\/(A (Q (f 0 (ml x))))
  C2 : (x:D)~(A (P (f (S 0) (ml x))))
  C3 : (x:D)~(A (Q x))
  ============================
   False
```

Resolution:

```
Elim (C1 (f (S 0) (ml d))).
Exact (C2 d).
Exact (C3 (f 0 (ml (f (S 0) (ml d))))).
Qed.
```

## One for the road: the Principle of Pubs

If we name an explicit domain (the pub), the Drinker's Principle can be reformulated as:

$$\exists x((pub(x) \wedge drunk(x)) \Rightarrow \forall y(pub(y) \Rightarrow drunk(y)))$$

After we have declared parameters `drunk` and `pub` in type `D->D`, we can phrase its formal counterpart as follows:

```
Definition f_DP2 :=
 (f_ex [x:D](f_impl
  (f_and (f_atom (pub x))
         (f_atom (drunk x)) )
  (f_all [y:D](f_impl (f_atom (pub y))
                      (f_atom (drunk y)) )) )).
```

Now we introduce the Principle of Pubs, stating: *if the Drinker's Principle does not hold, then there is nobody in the pub.* Formally represented by:

```
Definition f_PoP :=
 (f_impl (f_not f_DP2)
         (f_not (f_ex [x:D](f_atom (pub x)))) ).
```

We show the proof of:

```
Lemma PoP : EM ->AC->D->(E f_PoP).
```

Again, the tactics are:

```
Intros em ac d.
Apply (Refute em ac d).
Simpl.
Intros f C1 C2 C3 C4 C5.
```

Computation of the $\beta\delta\iota$-contractum (`Simpl`) takes Coq more than two minutes (on our machine). We have arrived at:

```
 em : EM
 ac : AC
  d : D
  f : SKF
 C1 : (x:D)(A (pub x))
 C2 : (x:D)(A (drunk x))
 C3 : (x:D)(A (pub (f O (ml x))))
 C4 : (x:D)~(A (drunk (f O (ml x))))
 C5 : (x:D)(A (pub (f (S O) (ml x))))
 ============================
  False
```

We deduce `False` by:

```
Apply (C4 d).
Apply C2.
Qed.
```

# Chapter 3

# Future Research & Discussion

The first step of the 'using Otter in Coq' project has been carried out succesfully, as we have fully outlined in the previous chapter. After a discussion on an important dichotomy in the class of formal propositions, we anticipate on the next steps of the project. In Section 3.2 we consider how the proof information from Otter can be used to build lambda terms in Coq. The ideas about elimination of Skolem axioms are presented in the concluding section.

## 3.1 Good and bad formulae

Besides the theorem of Section 2.4, correctness of the clausification algorithm involves showing that this algorithm always produces the desired result. I.e. we have to prove that any input proposition will actually be transformed to its clausal form. This requires to give an inductive definition of clausal forms.

Another nice property we would like to show, can be phrased as:

> *For every* `p:prop` *the* $\beta\delta\iota$*-normal form of* `(Refute em ac d p)` *is typable in the initial environment of Coq with a context consisting of* `em:EM`*,* `ac:AC`*,* `D:Set`*,* `d:D` *and* `A:D->Prop` *only.*

Or, put differently: after reduction, no instance of `Refute` refers to any of the definitions, fixpoints, inductive sets, lemmas etcetera, we have constructed; i.e. all these instances are cut-free.

Recall that the constructors `f_all`, `f_ex` both have type `(D->prop)->prop`. Due to the power of Coq, there is much freedom in the construction of formal propositions. In particular, every definable function of type `D->prop` can be quantified. This can lead to objects of type `prop` that do not represent first-order propositions, so that the clausification program cannot be expected to give the desired result.

Consider the following example, where we take `nat` for `D`. First we define a function that, given an integer `n`, iterates `f_not` $n$ times.

```
Fixpoint It_f_not [n:nat]: prop := Cases n of
 O => (f_atom O) | (S m) => (f_not (It_f_not m)) end.
```

Now observe that `It_f_not` has type `nat->prop`, so that `(f_all It_f_not)`[1] is typed `prop`, but it is impossible to make sense of the clausal form of this term. Such an object can be qualified as a *bad* formula.

[1] This anomalous object can be thought of as: $\forall n \neg^n A(0)$.

It obviously makes sense to single out objects of type `prop` that are not bad formulae. Therefore we define inductively a subset *FORM* of `prop`.

DEFINITION 1 (*FORM* $\subset$ `prop`)

- *If* `x:D` *is a variable*[2]*, then* `(f_atom x)` $\in$ *FORM;*
- *FORM is closed under* `f_not`*,* `f_and`*,* `f_or` *and* `f_impl`*;*
- *If* `t` $\in$ *FORM, then* `(f_ex [x:D]t)`*,*`(f_all [x:D]t)` $\in$ *FORM.*

*FORM* allows dependencies at term positions only.[3] Objects of type `prop` that are in the subset *FORM* will be qualified as *good* formulae. They can safely be viewed as representing first-order propositions in a language with one unary predicate symbol. For example, the formal counterpart `f_DP` of the Drinker's Principle in Section 2.6 is a good formula. There are certainly more formulae that can be allowed as good. For example, adding a binary predicate symbol to the language can be represented by allowing also atoms `(f_atom (P x y))`$\in$ *FORM*, with `x,y:D` and `P:D->D->D` all variables.

Good formulae have a number of interesting properties. For example, we can prove that the output of `CLAUS` applied to a good formula is indeed a clausal form. Interestingly, normal forms of terms `(CLAUSeq em ac d p)` with `p` a good formula do not contain any reference to `prop`, and can hence be typed in a much weaker context. For spatial reasons we do not illustrate this property here.

## Application of `PNNF` to good formulae

Restricted to good formulae, we can indeed prove that the output of `CLAUS` is always of the desired format. Throughout this section we use the following convention; here $\longrightarrow^*_{\beta\delta\iota}$ is the reflexive, transitive closure of the one-steps $\longrightarrow_{\beta\delta\iota}$.

CONVENTION 1 `t`$\varepsilon S$ *if and only if* $\exists$`t'` *such that* `t`$\longrightarrow^*_{\beta\delta\iota}$`t'` *and* `t'`$\in S$.

By way of example, we give a proof of (Theorem 1):
$\forall$`t` $\in$ *FORM*. `(PNNF t)`$\varepsilon$*PNNF*, where *PNNF* is defined below.

DEFINITION 2 (*Qfree* $\subset$ *FORM*)

- *If* `x:D` *is a variable, then* `(f_atom x)` $\in$ *Qfree;*
- *Qfree is closed under* `f_not`*,* `f_and`*,* `f_or` *and* `f_impl`*.*

DEFINITION 3 (*PNF* $\subset$ *FORM*)

- *If* `t` $\in$ *Qfree, then* `t` $\in$ *PNF;*
- *If* `t` $\in$ *PNF, then* `(f_ex [x:D]t)`*,* `(f_all [x:D]t)` $\in$ *PNF.*

DEFINITION 4 (*NNF* $\subset$ *FORM*)

- *If* `x:D` *is a variable, then* `(f_atom x)`*,*`(f_not (f_atom x))` $\in$ *NNF;*
- *If* `t1`*,*`t2` $\in$ *NNF, then* `(f_and t1 t2)`*,*`(f_or t1 t2)` $\in$ *NNF;*
- *If* `t` $\in$ *NNF, then* `(f_ex [x:D]t)`*,*`(f_all [x:D]t)` $\in$ *NNF.*

[2] Remark that variables don't have a distinct status at the formal level; thus, we define *FORM* on the metalevel of this LaTeX-document.

[3] I.e. only in the argument of the `f_atom`-constructor.

DEFINITION 5 $PNNF = PNF \cap NNF$

Theorem 1 is supported by the following lemmas; $\equiv$ denotes syntactical equality.

LEMMA 1 $\forall$`t1,t2` $\varepsilon PNNF$. `(pnf_cj t1 t2)` $\varepsilon PNNF$

**Proof** Let `t1,t2` $\varepsilon PNNF$. Observe the fact that for all `t` $\varepsilon PNNF$:
*either* `t` $\varepsilon Qfree$, *or* `t` $\equiv$ `(f_ex [x:D]u)` with `u` $\varepsilon PNNF$,
*or* `t` $\equiv$ `(f_all [x:D]u)` with `u` $\varepsilon PNNF$.
We distinguish the following cases.

`t1` $\varepsilon Qfree$

`t2` $\varepsilon Qfree$ We have: `(pnf_cj t1 t2)` $\longrightarrow^*_{\beta\delta\iota}$ `(f_and t1 t2)`.
Therefore: `(pnf_cj t1 t2)` $\varepsilon PNNF$.

`t2` $\equiv$ `(f_ex [x:D]u2)` Induction Hypothesis:
`u2` $\varepsilon PNNF \Rightarrow$ `(pnf_cj t1 u2)` $\varepsilon PNNF$.
From definitions 3 and 4 it follows that `u2` $\varepsilon PNNF$. Therefore by IH and the reduction `(pnf_cj t1 u2)` $\longrightarrow^*_{\beta\delta\iota}$ `(pnf_cj_r t1 u2)`,
we have: `(pnf_cj_r t1 u2)` $\varepsilon PNNF$.
Since `(pnf_cj t1 t2)` $\longrightarrow^*_{\beta\delta\iota}$ `(f_ex [x:D](pnf_cj_r t1 u2))`,
it follows: `(pnf_cj t1 t2)` $\varepsilon PNNF$.

`t2` $\equiv$ `(f_all [x:D]u2)` By an argument analogous to the one above, we have:
`(pnf_cj_r t1 u2)` $\varepsilon PNNF$.
Moreover: `(pnf_cj t1 t2)` $\longrightarrow^*_{\beta\delta\iota}$ `(f_all [x:D](pnf_cj_r t1 u2))`,
and so: `(pnf_cj t1 t2)` $\varepsilon PNNF$.

`t1` $\equiv$ `(f_ex [x:D]u1)` We have:
`(pnf_cj t1 t2)` $\longrightarrow^*_{\beta\delta\iota}$ `(f_ex [x:D](pnf_cj u1 t2))`.
As we infer that `u1` $\varepsilon PNNF$, we have by IH: `(pnf_cj t1 t2)` $\varepsilon PNNF$.

`t1` $\equiv$ `(f_all [x:D]u1)`

`t2` $\varepsilon Qfree$ We have:
`(pnf_cj t1 t2)` $\longrightarrow^*_{\beta\delta\iota}$ `(f_all [x:D](pnf_cj u1 t2))`.
Thus: `(pnf_cj t1 t2)` $\varepsilon PNNF$.

`t2` $\equiv$ `(f_ex [x:D]u2)` We have:
`(pnf_cj t1 t2)` $\longrightarrow^*_{\beta\delta\iota}$ `(f_all [x:D](pnf_cj u1 t2))`.
Thus: `(pnf_cj t1 t2)` $\varepsilon PNNF$.

`t2` $\equiv$ `(f_all [x:D]u2)` We have:
`(pnf_cj t1 t2)` $\longrightarrow^*_{\beta\delta\iota}$ `(f_all [x:D](pnf_cj u1 u2))`.
Thus: `(pnf_cj t1 t2)` $\varepsilon PNNF$.

□

LEMMA 2 $\forall$`t1,t2` $\varepsilon PNNF$. `(pnf_dj t1 t2)` $\varepsilon PNNF$

**Proof** analogous to the proof of Lemma 1.

□

LEMMA 3 $\forall$`t` $\in FORM$. `(pnnf t pos),(pnnf t neg)` $\varepsilon PNNF$

**Proof** by structural induction on `t`:

`t` ≡ `(f_atom x)`

- `(pnnf t pos)` $\longrightarrow^*_{\beta\delta\iota}$ `t` $\varepsilon PNNF$.
- `(pnnf t neg)` $\longrightarrow^*_{\beta\delta\iota}$ `(f_not t)` $\varepsilon PNNF$.

`t` ≡ `(f_not t0)`

- `(pnnf t pos)` $\longrightarrow^*_{\beta\delta\iota}$ `(pnnf t0 neg)` $\varepsilon PNNF$ by IH.
- `(pnnf t neg)` $\longrightarrow^*_{\beta\delta\iota}$ `(pnnf t0 pos)` $\varepsilon PNNF$ by IH.

`t` ≡ `(f_and t1 t2)`

- `(pnnf t pos)` $\longrightarrow^*_{\beta\delta\iota}$ `(pnf_cj (pnnf t1 pos)(pnnf t2 pos))`
  Thus: `(pnnf t pos)` $\varepsilon PNNF$ by IH and Lemma 1.
- `(pnnf t neg)` $\longrightarrow^*_{\beta\delta\iota}$ `(pnf_dj (pnnf t1 neg)(pnnf t2 neg))`
  Thus: `(pnnf t neg)` $\varepsilon PNNF$ by IH and Lemma 2.

`t` ≡ `(f_or t1 t2)`

- `(pnnf t pos)` $\longrightarrow^*_{\beta\delta\iota}$ `(pnf_dj (pnnf t1 pos)(pnnf t2 pos))`
  Thus: `(pnnf t pos)` $\varepsilon PNNF$ by IH and Lemma 2.
- `(pnnf t neg)` $\longrightarrow^*_{\beta\delta\iota}$ `(pnf_cj (pnnf t1 neg)(pnnf t2 neg))`
  Thus: `(pnnf t neg)` $\varepsilon PNNF$ by IH and Lemma 1.

`t` ≡ `(f_impl t1 t2)`

- `(pnnf t pos)` $\longrightarrow^*_{\beta\delta\iota}$ `(pnf_dj (pnnf t1 neg)(pnnf t2 pos))`
  Thus: `(pnnf t pos)` $\varepsilon PNNF$ by IH and Lemma 2.
- `(pnnf t neg)` $\longrightarrow^*_{\beta\delta\iota}$ `(pnf_cj (pnnf t1 pos)(pnnf t2 neg))`
  Thus: `(pnnf t neg)` $\varepsilon PNNF$ by IH and Lemma 1.

`t` ≡ `(f_ex [x:D]t0)`

- `(pnnf t pos)` $\longrightarrow^*_{\beta\delta\iota}$ `(f_ex [x:D](pnnf t0 pos))` $\varepsilon PNNF$ by IH.
- `(pnnf t neg)` $\longrightarrow^*_{\beta\delta\iota}$ `(f_all [x:D](pnnf t0 neg)` $\varepsilon PNNF$ by IH.

`t` ≡ `(f_all [x:D]t0)`

- `(pnnf t pos)` $\longrightarrow^*_{\beta\delta\iota}$ `(f_all [x:D](pnnf t0 pos))` $\varepsilon PNNF$ by IH.
- `(pnnf t neg)` $\longrightarrow^*_{\beta\delta\iota}$ `(f_ex [x:D](pnnf t0 neg)` $\varepsilon PNNF$ by IH.

□

THEOREM 1 $\forall$`t`$\in$ *FORM*. `(PNNF t)`$\varepsilon PNNF$

**Proof** Let `t` $\in$ *FORM*. `(PNNF t)` $\longrightarrow^*_{\beta\delta\iota}$ `(pnnf t pos)`
Thus: `(PNNF t)` $\varepsilon PNNF$ by Lemma 3.

□

## 3.2 Towards Resolution in Coq

Suppose we are in the middle of a Coq proof session and we have applied the clausification algorithm via the lemma `Refute`. We have a number of clauses in the context and `False` is to be deduced.

By way of example, let us zoom in on two particular clauses having a complementary literal:

$$(p \vee q) \vee r \text{ and } s \vee (\neg r \vee t)$$

Suppose these clauses are given to Otter.[4] Moreover, suppose we find the following lines within the output produced by Otter:

```
1 [] p|q|r.
2 [] s| -r|t.
3 [binary,1.3,2.2] p|q|s|t.
```

Thus, the third literal in the first clause is resolved with the second literal in the second clause, yielding the resolvent listed in line 3. Now, how can we mirror Otter in Coq; more precise: given the information above, how to build a lambda term corresponding to this single resolution step?

In general, we might choose to encode single resolution steps by a term `resolve1`:

```
Lemma resolve1 : (A,B,C:Prop)(A\/B)->(~A\/C)->(B\/C).
```

Observe that, in order to apply `resolve1` to the example above, we need to pull the resolution literals $r$ and $\neg r$ forewards, such that they become the head of their respective disjunctions. That is: reorder $(p \vee q) \vee r$ into $r \vee (p \vee q)$ and $s \vee (\neg r \vee t)$ into $\neg r \vee (s \vee t)$.

We have defined functions `select` and `delete`. Given an integer `k`, the former selects the $k$-th disjunct; the latter deletes the $k$-th disjunct and returns the remaining disjunction. They are supported by a function that counts literals among clauses:[5]

```
Fixpoint lit_nr [p:prop] : nat :=
Cases p of
 (f_or p0 p1) => (plus (lit_nr p0)(lit_nr p1))
|  _           => (S O)
end.
```

The function `select` is defined as follows:

```
Fixpoint select [k:nat;p:prop] : prop :=
Cases k (leqbo k (lit_nr p)) of
  (S _) true (*1*) => Cases p of
     (f_or p0 p1) => Cases (leqbo k (lit_nr p0)) of
        true  (*2*) => (select k p0)
      | false (*3*) => (select (minus k (lit_nr p0)) p1) end
   | _      (*4*) => p end
| _     _          => p
end.
```

We give a short description of its working. Counting starts from 1 (`(S O)`), just like Otter does. Propositions that are not constructed by `f_or` are treated as literals. Let $k$ denote the $k$-th literal to be selected and let the input proposition `p` be of the general format:

$$(l_1 \vee \cdots \vee l_i) \vee (l'_{i+1} \vee \cdots \vee l'_n)$$

[4] Where the necessary syntactical transformations have been performed.

[5] Here, by clauses we mean unprefixed disjunctions.

`(*1*)` $0 < k \leq n$.

`(*2*)` $0 < k \leq i$.

`(*3*)` $k > i$ and so the $k$-th literal is to be found in the second disjunct: `(select` $(k-i)$ $(l'_1 \vee \cdots \vee l'_{n-i})$`)`.

`(*4*)` This is the literal to be selected, since the number of literals in `p` equals 1 and therewith $k = 1$.

Now we give the definition of `delete`. Again, let `p` $\equiv (l_1 \vee \cdots \vee l_i) \vee (l'_{i+1} \vee \cdots \vee l'_n)$.

```
Fixpoint delete [k:nat;p:prop] : prop :=
Cases k (leqbo k (lit_nr p)) of
 (S _) true (*1*) => Cases p of
  (f_or p0 p1) => Cases (leqbo k (lit_nr p0)) of
     true  (*2*) => Cases (lit_nr p0) of
     (S O) (*5*) => p1
    | _    (*6*) => (f_or (delete k p0) p1) end
   | false (*7*) => Cases (lit_nr p1) of
     (S O) (*8*) => p0
    | _    (*9*) =>
           (f_or p0 (delete (minus k (lit_nr p0)) p1)) end end
 | _     (*0*) => p end
| _     _         => p
end.
```

`(*5*)` The first disjunct is the literal to be deleted, as $i = 1$ (and thus $k = 1$); the second disjunct (`p1`) is returned.

`(*6*)` Remove the $k$-th literal from the first disjunct.

`(*7*)` $k > i$.

`(*8*)` The second disjunct (`p1`) contains one literal. This must be the literal to be deleted, as we infer $k = i + 1$.

`(*9*)` Remove the $(k-i)$-th literal from the second disjunct.

`(*0*)` As we have to give an object in `prop`, `(delete (S O) p)` with `p` a literal results in `p`.

We have proved the following lemma.[6]

```
Lemma Pull :
 (k:nat)(p:prop)(E p)->((E (select k p))\/(E (delete k p))).
```

Let us show its use by going back to the example above. First, the context is extended as follows:

```
Parameter pd,qd,rd,sd,td: D.

Definition p := (f_atom pd).
Definition q := (f_atom qd).
Definition r := (f_atom rd).
Definition s := (f_atom sd).
Definition t := (f_atom td).
```

[6] Of course, we have proved the reverse implication too.

Second, we formally phrase the clauses $(p \vee q) \vee r$ and $s \vee (\neg r \vee t)$ as:

```
Definition f_C1 := (f_or (f_or p q) r).
Definition f_C2 := (f_or s (f_or (f_not r) t)).
```

Their interpretations are put as hypotheses into the context:

```
Hypothesis H1 : (E f_C1).
Hypothesis H2 : (E f_C2).
```

Now we prove their resolvent $(p \vee q) \vee (s \vee t)$.

```
Lemma resolvent : (E (f_or (f_or p q)(f_or s t))).
```

First, we apply the proof term `resolve1`:

```
Apply (resolve1 (E r) (E (f_or p q)) (E (f_or s t)) ).
```

Now two subgoals are generated. The first:

```
(*sg1*)  (E r)\/(E (f_or p q))
```

is proved by pulling the third literal from the first clause to the fore:

```
Exact (Pull (S(S(S O))) f_C1 H1).
```

The second subgoal:

```
(*sg2*)  ~(E r)\/(E (f_or s t))
```

is proved by:

```
Exact (Pull (S(S O)) f_C2 H2).
```

## 3.3 Elimination of Skolem axioms

By adapting a result of Kleene, Skolem functions and -axioms can be eliminated from resolution refutations, which allows one to obtain proofs independent of the Axiom of Choice. We refer to [4] for a modern exposition of Kleene's result. Figure 3.1 is just a variation of the example in [4]; it shows how the elimination procedure works in the case of the Drinker's Principle.

The assumptions $\forall x(D(x) \wedge \neg D(f(x)))$ of the upper deduction tree are the clausal form of the negated Drinker's Principle, and the existence of the function $f$ relies on the Axiom of Choice. In the middle and lower deduction trees, we have replaced every Skolem term $t$ by a fresh free variable $v_t$. It is to be understood that $v_{f(d)}$ does not contain an occurence of $d$. In the lower deduction tree, the assumptions $\forall x \exists y(D(x) \wedge \neg D(y))$ are the prenex negation normal form of the negated Drinker's Principle. The order in which the $\exists$-eliminations take place is of crucial importance to satisfy the eigenvariable condition. We plan to elaborate the elimination procedure for resolution refutations more generally.

$$\dfrac{\dfrac{\dfrac{\forall x(D(x) \wedge \neg D(f(x)))}{D(f(d)) \wedge \neg D(f(f(d)))}\,\forall E}{D(f(d))}\,\wedge E \qquad \dfrac{\dfrac{\forall x(D(x) \wedge \neg D(f(x)))}{D(d) \wedge \neg D(f(d))}\,\forall E}{\neg D(f(d))}\,\wedge E}{\bot}\,\Rightarrow E$$

$\Delta$, *natural deduction format of the resolution refutation.*

$$\dfrac{\dfrac{[D(v_{f(d)}) \wedge \neg D(v_{f(f(d))})]^1}{D(v_{f(d)})}\,\wedge E \qquad \dfrac{[D(v_d) \wedge \neg D(v_{f(d)})]^2}{\neg D(v_{f(d)})}\,\wedge E}{\bot}\,\Rightarrow E$$

$\Delta'$, *canonical translation of the propositional frame of* $\Delta$.

$$\dfrac{\dfrac{\forall x \exists y(D(x) \wedge \neg D(y))}{\exists y(D(v_d) \wedge \neg D(y))}\,\forall E \qquad \dfrac{\dfrac{\forall x \exists y(D(x) \wedge \neg D(y))}{\exists y(D(v_{f(d)}) \wedge \neg D(y))}\,\forall E \qquad \Delta'}{\bot}\,\exists E_1}{\bot}\,\exists E_2$$

*Natural deduction format of the proof without Skolem functions.*

Figure 3.1: Elimation of Skolem functions from a resolution refutation. The indices connect the discharging of an assumption with the corresponding $\exists$-elimination.

# Bibliography


[1] H.P. Barendregt *The lambda calculus: its syntax and semantics*. Studies in Logic 103, North-Holland, Amsterdam, 1984.

[2] B. Barras et al. *The Coq Proof Assistent Reference Manual, version 6.1*. INRIA, 1996.

[3] M. Bezem and D. Hendriks. *Clausification in Coq*. Submitted to Proceedings TYPES '98.

[4] G. Dowek. *Automated theorem proving in type theory*. Course notes for the 2nd International Summer School in Logic for Computer Science, University of Chambery, France, 1994.

[5] J.-Y. Girard, Y. Lafont and P. Taylor. *Proofs and Types*. Cambridge Tracts in Theoretical Computer Science 7. Cambridge University Press, 1989.

[6] D.W. Loveland. *Automated theorem proving: a logical basis*. Fundamental studies in computer science. North-Holland, Amsterdam, 1978.

[7] W. McCune. *Otter 3.0 Reference Manual and Guide*. Tech. Report ANL-94/6, Argonne National Laboratory, Argonne, IL, 1994.

[8] J.R. Shoenfield. *Mathematical Logic*. Addison-Wesley series in logic. Addison-Wesley, 1967.