Clausification in Coq

Marc Bezem
Utrecht University *

Dimitri Hendriks
Utrecht University †

Abstract

Clausification is an essential step in the so-called resolution method, one of the most successful procedures for automated theorem proving. Anticipating the use of resolution in proof construction systems based on type theory, we elaborate the clausification procedure in `Coq` and illustrate its usefulness. The results presented in this paper also constitute the formal verification of the correctness of clausification. The complete clausification program and the correctness proof can be found on the following Internet address:

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

1 Introduction

The proof generation capabilities of proof construction systems such as `Coq` [1], based on type theory, could still be improved. Resolution based theorem provers such as `Otter` [5] are more powerful in this respect, but have the drawback that they work with normal forms of formulas, so-called clausal forms. Clauses are (universally closed) disjunctions of literals, and a literal is either an atom or the negation of an atom. The clausal form of a formula is essentially its prenex-Skolem-conjunctive normal form, which need not be exactly logically equivalent to the original formula. This makes resolution proofs hard to read and understand, and makes the navigation of the theorem prover through the search space very difficult. Type theory, with its highly expressive language, is much better in this respect, but the proof generation capabilities suffer from the small granularity of the inference steps and the corresponding astronomic size of the search space. Typically, one hyperresolution step requires a few dozens of inference steps in type theory. The idea of the present project is to combine the best of both worlds.

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 to `Otter`, and delegate its 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`.

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 awkward invariants as equiconsistency or equisatisfiability in the first-order case.

*Department of Philosophy, P.O. Box 80126, 3508 TC Utrecht, The Netherlands, e-mail `bezem@phil.uu.nl`.

†e-mail `hendriks@phil.uu.nl`.

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 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.

The paper is organised as follows. In the next section we introduce the Drinker's Principle as running example. In Section 3 we set up a formal representation of first-order formulae. In Section 4 we give a schematic overview of the clausification procedure, which is applied to prove the Drinker's Principle in Section 5. In Sections 6 and 7 we give some more details of the clausification procedure and of its correctness proof, respectively. In the last Section 8 we describe future research.

2 Example: 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.

`Otter` [5] refutes almost instantaneously the negation of the Drinker's Principle:

```
-(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 false:

```
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. We leave it as an exercise to the reader to make the relation between the original formulation of the problem and the clausal form precise.

In type theory, and in particular in `Coq`, the typing relation is expressed by `t:T`, to be interpreted as '`t` belongs to `T`' when `T:Set` ('`T` is a set'), and as '`t` is a proof of `T`' when `T:Prop` ('`T` is a proposition'). Furthermore, `(x:T)` denotes universal quantification, `[x:T]` lambda abstraction, and `(M N)` well typed application. `Coq` uses `->` for implication between propositions as well as for function spaces, `~` for negation and `EX` for existential quantification. Now the Drinker's Principle can be formulated as follows.

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

In order to prove the Drinker's Principle in Coq a dozen of tactic commands have to be entered by the user, generating a proof term of about half a page.

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.

3 Propositions as formal objects

The universe Prop of all propositions in Coq 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. In order to manipulate first-order propositions as syntactic objects on one hand, and reflect upon them on the other hand, we adopt a two-level approach, with higher (meta-) level Prop and lower (object-, formal) level prop. On both levels objects will be called 'propositions', and it should be clear from the context which level applies.

We found it necessary to define prop as an inductive set in which first-order propositions can be represented formally. Objects in prop will be interpreted in Prop. As every inductive set, prop is equipped with higher order primitive recursion as powerful computational device. To begin with, we need a domain of discourse D and define the inductive set prop depending on D.

```
Parameter D: Set.

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.
```

In order to accomodate Skolemization the formal language is extended with a higher order existential quantifier. Therefore we define prop' with one constructor f_Ex. Here 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 way of representing Skolem functions will be explained later.

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

We proceed by defining E and E', canonical homomorphisms[1] that interpret

[1] 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. Furthermore, after abstraction from D as parameter of the inductive definition, we get prop:Set->Set and we cannot apply prop to itself for simple typing reasons.

prop- resp. prop'-objects in Prop, by primitive recursion. We declare a unary predicate symbol drunk to form atoms.

```
Parameter drunk : D->Prop.

Fixpoint E [phi:prop] : Prop :=
Cases phi of
  (f_atom d)   => (drunk 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.
```

4 Schematic overview of the whole procedure

A given first-order proposition ϕ in Prop will be translated to its corresponding formal counterpart in prop by a syntax-based translation Quote outside Coq (as yet: by hand). We abbreviate Quote(ϕ) by f_ϕ. 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 ϕ and the normal form of (E f_ϕ) are identical, whenever Quote applies. Also, Quote(E f_ϕ) has to be identical to f_ϕ. The function CLAUS, applied to f_ϕ:prop, computes the clausal form of f_ϕ, which is subsequently mapped into Prop by E'. See Figure 1 for the schema of the clausification.

Suppose that, during a Coq proof session, a first-order tautology ϕ (in Prop) is to be proved. First, ϕ has to be translated to its corresponding formal counterpart, f_ϕ. Second, the clausification function CLAUS is applied to the negation (f_not f_ϕ). (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_ϕ)), which is done by the function MIMPL. Applying the function MIMPL to (CLAUS (f_not f_ϕ))) 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 ϕ.

If ϕ is indeed a tautology, then the clauses obtained in this way are inconsistent. By the completeness of resolution, see for example [4], there exists a resolution refutation of these clauses.

In the next section we illustrate the outlined procedure by an example.

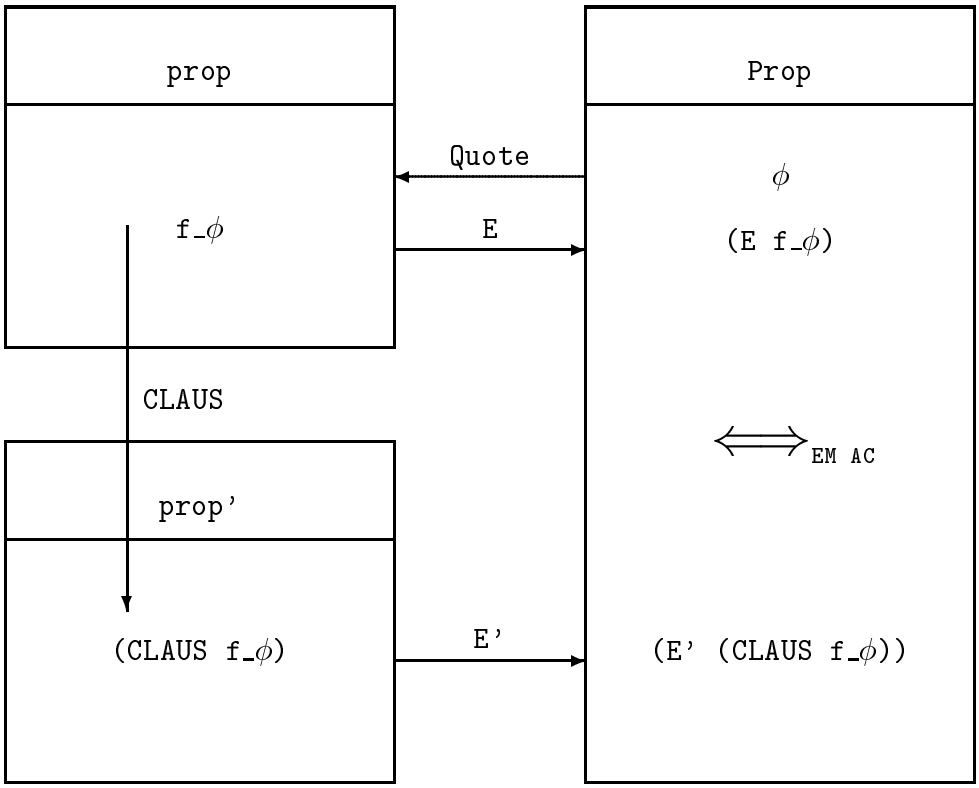

Figure 1: Schema of the clausification procedure; the proof of the equivalence in the right box can be generated uniformly in `f_`ϕ.

5 Back to the Drinker's Principle

We phrase the formal counterpart of the discussed Drinker's Principle as follows:

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

Indeed, applying `E` and normalizing yields the desired result:

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

Now we prove the Drinker's Principle, this time not from scratch but by clausification (automated) and resolution (as yet: by hand). We make use of the following general lemma:

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

Here comes the `Coq` session:

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

First we introduce the hypotheses into the context.

```
Intros em ac d.
```

Then `Refute` is applied. The resulting goal is $\beta\delta\iota$-reduced by `Simpl`, that is, β-redexes, definitions and recursion schemata are maximally unfolded. Thereafter the Skolem function and the clauses are introduced.

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

We are now in the following proof state:

```
 em : EM
 ac : AC
  d : D
  f : SKF
 C1 : (x:D)(drunk x)
 C2 : (x:D)~(drunk (f 0 (ml x)))
==================================
  False
```

Here and below `(ml x)` is the list with single element `x:D`. 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.
```

The handcrafted lambda term from Section 2, 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 normal form does not contain any occurrence of `prop`, see Section 8.

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

```
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)`.

In the next section we discuss some of the programmed conversion steps in the clausification procedure.

6 The program CLAUS

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

First, we have defined an inductive set of polarities `POL` consisting of two constants `pos` and `neg`:

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

Second, the function `pnnf` is recursively defined, which, given a proposition and its polarity, returns its prenex negation normal form. It makes use of `pnf_cj` and `pnf_dj`, functions that move quantifiers outwards of conjunctions and disjunctions, respectively, according to common logical rules. For instance,

```
(pnf_cj (f_and (f_all [x:D](dl x))(f_all [y:D](dr y))))
```

with `dl` and `dr` of type `D->prop`, yields

```
(f_all [x:D](f_and (dl x)(dr x)))
```

relying on

- $(\forall x F_0(x) \wedge \forall y F_1(y)) \Leftrightarrow \forall x(F_0(x) \wedge F_1(x))$

The main call will be: `(pnnf f pos)`, invoked 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).
```

Thus a given proposition `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 proposition starts with a negation, `pnnf` continues recursively with the subproposition and with swapped polarity. Apart from the `f_impl`-case, the cases with positive polarity are straightforward. The elimination of implications appeals to the 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:

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

After application of `PNNF`, formal propositions are put into *Conjunctive Normal Form*. We don't present this module here, but continue to discuss the next conversion step, *Skolemization*, performed by the function `SKLM`.

Skolemization of a given prenex 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 the 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. Consider the following example: $\forall x \exists y \forall z \exists u\ P(x, y, z, u)$ Skolemizes into $\exists f \exists g \forall x \forall y\ P(x, f(x), y, g(x, y))$. Note the possibility of 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. Then, everytime an existential quantifier jumps over a universal quantifier, the type of the corresponding Skolem function gets a higher index. It is possible to compute this index on beforehand, but, unfortunately, the typing system will not recognize on beforehand that the type of the Skolem function is correctly indexed in this way.

We chose for a simple and elegant solution. Skolemized formulae will now 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: different 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)))`. Here we have a Skolem function indexed by `O` that depends on the list with one element, the universally quantified variable `x`.

Since input propositions of `SKLM` are already in prenex form, the recursive definition is relatively simple.

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

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

7 Preservation of logical equivalence

The theorem `CLAUSeq` states that `CLAUS` preserves equivalence modulo the Excluded Middle (`EM`), the Axiom of Choice (`AC`) and non-emptiness of `D`. In this section we give a sample of the proof of `CLAUSeq`. We chose to explain the most interesting lemma `SKLMeq`, stating the correctness of Skolemization. The proof of this lemma 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 abstract. 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`.

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

The Axiom of Choice is defined as follows:

```
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))).
```

Let us inspect the crucial point at which the Axiom of Choice is applied. Consider the following subgoal arising in the inductive proof of `SKLMeq`. (Case `f_all`, left-to-right half of `<->`.)

```
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. Now comes the crucial point where the Axiom of Choice is used. By application of `ac` to `H0`[2] 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

```
(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 term `H2` having type

```
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)))
```

is a logical consequence of `H2`. This means that for any list `L:(list D)` 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 that

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

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

```
(E (sklm g 0 (ml x) (dp x))) <-> (E (sklm (F x) 0 (nil D) (dp x)))
```

The above equivalence requires considerable effort, but thereafter the proof (of one half of this inductive case ...) can be completed by a simple application of `H2`.

[2] 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`.

8 Future research

8.1 Good and bad formulas

Recall that the constructors `f_all`, `f_ex` have both 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 the application of `f_not` on `(f_atom 0)` as many as `n` times.

```
Fixpoint It_f_not [n:nat]: prop := Cases n of
 0 => (f_atom 0) | (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)` is typed `prop`, but it is impossible to make sense of the clausal form of this term. Such an object of type `prop` can be qualified as a *bad* formula.

It obviously makes sense to single out objects of type `prop` that are not bad formulas. Therefore we define inductively a subset *form* of `prop` as follows:

- If `x:D` is a variable, 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)`$\in$ *form* and `(f_all [x:D]t)`$\in$ *form*.

Objects of type `prop` that are in the subset *form* will be qualified as *good* formulas. 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 5 is a good formula. There are certainly more formulas 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 formulas 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 the correctness proof of the clausification of a good formula (to be precise, 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. Some of these facts have already been elaborated in [3].

8.2 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 [2] for a modern exposition of Kleene's result. In Figure 2 we show as an example 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.

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

Δ, natural deduction format of the resolution refutation

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

Δ', canonical translation of the propositional frame of Δ

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

natural deduction format of the proof without Skolem functions

Figure 2: Elimation of Skolem functions from a resolution refutation. The indices connect the discharging of an assumption with the corresponding $\exists$-elimination.

Acknowledgements

We thank Mark van der Zwaag for his contribution to the first version of the `CLAUS`-program, Erik Barendsen for helpful comments and Gilles Dowek for pointing out the possibility of the elimination of Skolem functions.

References

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

[2] 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.

[3] D. Hendriks. *Formal Representation & Correctness of Clausification of First-Order Formulae in Type Theory.* Master Thesis, Utrecht University, 1998.

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

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