Extraction of Proofs from the Clausal Normal Form Transformation

Hans de Nivelle

Max Planck Institut für Informatik
Stuhlsatzenhausweg 85
66123 Saarbrücken, Germany
nivelle@mpi-sb.mpg.de

Abstract. This paper discusses the problem of how to transform a first-order formula into clausal normal form, and to simultaneously construct a proof that the clausal normal form is correct. This is relevant for applications of automated theorem proving where people want to be able to use theorem prover without having to trust it.

1 Introduction

Modern theorem provers are complicated pieces of software containing up to 100,000 lines of code. In order to make the prover sufficiently efficient, complicated datastructures are implemented for efficient maintenance of large sets of formulas ([16]) In addition, they are written in programming languages that do not directly support logical formulas, like C or C++. Because of this, theorem provers are subject to errors.

One of the main applications of automated reasoning is in verification, both of software and of hardware. Because of this, users must be able to trust proofs from theorem provers completely. There are two approaches to obtain this goal: The first is to formally verify the theorem prover (the *internalization* approcah), the second is to make sure that the proofs of the theorem prover can be formally verified. We call this the *external approach.*

The first approach has been applied on simple versions of the CNF-transformation with success. In [10], a CNF-transformer has been implemented and verified in ACL2. In [5], a similar verification has been done in COQ.

The advantage of this approach is that once the check of the CNF-transformer is complete, there is no additional cost in using the CNF-transformer. It seems however difficult to implement and verify more sophisticated CNF-transformations, as those in [12], [1], or [8]. As a consequence, users have to accept that certain decision procedures are lost, or that less proofs will be found.

A principal problem seems to be the fact that in general, program verification can be done on only on small (inductive) types. For example in [5], it was necessary to inductively define a type **prop** mimicking the behaviour of Prop in COQ. In [10], it was necessary to limit the correctness proof to finite models. Because of this limitation, the internalization approach seems to be restricted to problems that are strictly first-order.

Another disadvantage of the internalization approach is the fact that proofs cannot be communicated. Suppose some party proved some theorem and wants to convince another party, who is skeptical. The other party is probably not willing to recheck correctness of the theorem prover and rerun it, because this might be very costly. It is much more likely that the other party is willing to recheck a proof.

In this paper, we explore the external approach. The main disadvantage of the external approach is the additional cost of proof checking. If one does the proof generation naively, the resulting proofs can have unacceptible size [6]. We present methods that bring down this cost considerably.

In this paper, we discuss the three main technical problems that appear when one wants to generate explicit type theory proofs from the CNF-transformation. The problems are the following: **(1)** Some of the transformations in the CNF-transformation are not equivalence preserving, but only satisfiability preserving. Because of this, it is in general not possible to prove $F \leftrightarrow \mathrm{CNF}(F)$. The problematic conversions are Skolemization, and subformula replacement. In order to simplify the handling of such transformations, we will define an intermediate proof representation language that has instructions that allow signature extension, and that make it possible to specify the condition that the new symbol must satisfy. When it is completed, the proof script can be tranformed into a proof term.

(2) The second problem is that naive proof construction results in proofs of unacceptible size. This problem is caused by the fact that one has to build up the context of a replacement, which constructs proofs of quadratic size. Since for most transformations (for example the Negation Normal Form transformation), the total number of replacements is likely to be at least linear in the size of the formula, the resulting proof can easily have a size cubic in the size of the formula. Such a complexity would make the external approach impossible, because it is not uncommon for a formula to have 1000 or more symbols. We discuss this problem in Section 3. For many transformations, the complexity can be brought down to a linear complexity.

(3) The last technical problem that we discuss is caused by improved Skolemization methods, see [11], [13]. Soundness of Skolemization can be proven through choice axioms. There are many types of Skolemization around, and some of them are parametrized. We do not want have a choice axiom for each type of Skolemization, for each possible value of the parameter. That would result in far too many choice axioms. In Section 4 we show that all improved Skolemization methods (that the author knows of) can be reduced to standard Skolemization.

In the sequel, we will assume familiarity with type theory. (See [15], [3]) We make use only of standard polymorphic type theory. In particular, we don't make use of inductive types.

2 Proof Scripts

We assume that the goal is find a proof term for $F \rightarrow \bot$, for some given formula F. If instead one wants to have a proof instead of rejection, for some G, then one has to first construct a proof of $\neg\neg G \rightarrow \bot$, and then transform this into a proof of G.

It is convenient not to construct this proof term directly, but first to construct a sequence of intermediate formulas that follow the derivation steps of the theorem prover. We call such sequence of formulas a *proof script.*

The structure of the proof script will be as follows: First $\Gamma \vdash A_1$, is proven. Next, $\Gamma, A_1 \vdash A_2$, is proven, etc. until $\Gamma, A_1, A_2, \ldots, A_{n-1} \vdash A_n = \bot$ is reached.

The advantage of proof scripts is that they can closely resemble the derivation process of the theorem prover. In particular, no stack is necessary to translate the steps of the theorem prover into a proof script. It will turn out, (Definition 2) that in order to translate a proof script into one proof term, the proof script has to be read backwards. If one would want to construct the proof term at once from the output of the theorem prover, one would have to maintain a stack inside the translation program, containing the complete proof. This should be avoided, because the translation of some of the proof steps alone may already require much memory. (See Section 3) When generating proof scripts, the intermediate proofs can be forgotten once they have been output.

Another advantage is that a sequence of intermediate formulas is more likely to be human readable than a big λ-term. This makes it easier to present the proof or to debug the programs involved in the proof generation.

Once the proof script has been constructed, one can translate the proof script into one proof term of the original formula. Alternatively, one can simply check the proof script itself.

We now define what a proof script is and when it is correct in some context. There are instructions for handling all types of intermediate steps that can occur in resolution proofs. The lemma-instruction proves an intermediate step, and gives a name to the result. The witness-instruction handles signature extension, as is needed for Skolemization. The split-instruction handles reasoning by cases. Some resolution provers have this rule implemented, most notably Spass, [17], see also [18].

Definition 1. *A proof script is a list of commands* $(c_1, \ldots, c_p)$ *with* $p > 0$. *We recursively define when a proof script is correct in some context. We write* $\Gamma \vdash (c_1, \ldots, c_p)$ *if* $(c_1, \ldots, c_p)$ *is correct in context* Γ.

- *If* $\Gamma \vdash x{:}\bot$, *then* $\Gamma \vdash (\text{false}(x))$.
- *If* $\Gamma, a_1{:}X_1, \ldots, a_m{:}X_m \vdash (c_1, \ldots, c_p)$, *and* c *has form*

$$\text{lemma}(a_1, x_1, X_1;\ \ldots\ ; a_m, x_m, X_m),\ \textit{with}\ m \geq 1,$$

the $a_1, \ldots, a_m$ *are distinct atoms, not occurring in* Γ, *and there are* $X'_1, \ldots, X'_m$, *such that for each* k, $(1 \leq k \leq m)$, $\Gamma \vdash x_k{:}X'_k$, *and*

$$\Gamma,\quad a_1 := x_1{:}X_1,\quad \ldots,\quad a_{k-1} := x_{k-1}{:}X_{k-1} \vdash X_k \equiv_{\alpha,\beta,\delta,\eta} X'_k,$$

then $\Gamma \vdash (c, c_1, \ldots, c_p)$.

- *Assume that* $\Gamma, a{:}A,\ h{:}(P\ a) \vdash (c_1, \ldots, c_p)$, *the atoms* a, h *are distinct and do not occur in* Γ. *If* $\Gamma \vdash x{:}(\forall a{:}A\ (P\ a) \to \bot) \to \bot$, *and* c *has form* $\mathrm{witness}(a, A, h, x, (P\ a))$, *then* $\Gamma \vdash (c, c_1, \ldots, c_p)$.
- *Assume that* $\Gamma, a_1{:}A_1 \vdash (c_1, \ldots, c_p)$ *and* $\Gamma, a_2{:}A_2 \vdash (d_1, \ldots, d_q)$. *If atoms* a_1, a_2 *do not occur in* Γ,

$$\Gamma \vdash x{:}(A_1 \to \bot) \to (A_2 \to \bot) \to \bot,$$

and c *has form* $\mathrm{split}(a_1, A_1, a_2, A_2, x)$, *then*

$$\Gamma \vdash (c, c_1, \ldots, c_p, d_1, \ldots, d_q).$$

When the lemma-instruction is used for proving a lemma, one has $m = 1$. Using the Curry-Howard isomorphism, the lemma-instruction can be also used for introducing definitions. The case $m > 1$ is needed in a situation where wants to define some object, prove some of its properties while still remembering its definition, and then forget the definition. Defining the object and proving the property in separate lemma-instructions would not be possible, because the definition of the object is immediately forgotten after the first lemma-instruction.

The witness-instruction is needed for proof steps in which one can prove that an object with a certain property exists, without being able to define it explicitly. This is the case for Skolem-functions obtained with the axiom of choice.

The split-instruction and the witness-instruction are more complicated than intuitively necessary, because we try to avoid using classical principles as much as possible. The formula $(\forall a{:}A\ (P\ a) \to \bot) \to \bot$ is equivalent to $\exists a{:}A\ (P\ a)$ in classical logic. Similarly $(A_1 \to \bot) \to (A_2 \to \bot) \to \bot$ is equivalent to $A_1 \vee A_2$ in classical logic. Sometimes the first versions are provable in intuitionistic logic, while the second versions are not.

Checking correctness of proof scripts is straightforward, and we omit the algorithm. We now give a translation schema that translates a proof script into a proof term. The proof term will provide a proof of $\bot$.

The translation algorithm constructs a translation of a proof script $(c_1, \ldots, c_p)$ by recursion. It breaks down the proof script into smaller proof scripts and calls itself with these smaller proof scripts. There is no need to pass complete proof scripts as argument. It is enough to maintain one copy of the proof script, and to pass indices into this proof script.

Definition 2. *We define a translation function* T. *For correct proof scripts,* $T(c_1, \ldots, c_p)$ *returns a proof of* $\bot$. *The algorithm* $T(c_1, \ldots, c_p)$ *proceeds by analyzing* c_1 *and by making recursive calls.*

- *If* c_1 *equals* $\mathrm{false}(x)$, *then* $T(c_1) = x$.
- *If* c_1 *has form* $\mathrm{lemma}(a_1, x_1, X_1, \ldots, a_m, x_m, X_m)$, *then first construct* $t := T(c_2, \ldots, c_p)$. *After that,* $T(c_1, \ldots, c_p)$ *equals*

$$(\lambda a_1{:}X_1 \cdots\ a_m{:}X_m\ t) \cdot x_1 \cdot \ldots \cdot x_m.$$

- *If* c_1 *has form* witness$(a, A,\ h, x, (P\ a))$, *first compute* $t := T(c_2, \ldots, c_p)$. *Then* $T(c_1, \ldots, c_p)$ *equals*

$$(x\ (\lambda a{:}A\ \ \lambda h{:}(P\ a)\ \ t)\ ).$$

- *If* c_1 *has form* split(a_1, A_1, a_2, A_2, x), *then there are two* false *statements in* $(c_2, \ldots, c_p)$, *corresponding to the left and to the right branch of the case split. Let* k *be the position of the* false-*statement belonging to the first branch. It can be easily found by walking through the proof script from left to right, and keeping track of the* split *and* false-*statements.*
  Then compute $t_1 = T(c_2, \ldots, c_k)$, *and* $t_2 = T(c_{k+1}, \ldots, c_p)$.
  The translation $T(c_1, \ldots, c_p)$ *equals*

$$(x\ (\lambda a_1{:}A_1\ \ t_1)\ (\lambda a_2{:}A_2\ \ t_2)\ ).$$

The following theorem is easily proven by induction on the length of the proof script.

Theorem 1. *Let the size of a proof script* $(c_1, \ldots, c_p)$ *be defined as* $|c_1| + \cdots + |c_p|$, *where for each instruction* c_i, *the size* $|c_i|$ *is defined as the sum of the sizes of the terms that occur in it.*
Then $|T(c_1, \ldots, c_p)|$ *is linear in* $|(c_1, \ldots, c_p)|$.

Proof. It can be easily checked that in $T(c_1, \ldots, c_p)$ no component of $(c_1, \ldots, c_p)$ is used more than once.

Theorem 2. *Let* $(c_1, \ldots, c_p)$ *be a proof script. If* $\Gamma \vdash (c_1, \ldots, c_p)$, *then* $\Gamma \vdash t{:}\bot$.

3 Replacement of Equals with Proof Generation

We want to apply the CNF-transformation on some formula F. Let the result be G. We want to construct a proof that G is a correct CNF of F. In the previous section we have seen that it is possible to generate proof script commands that generate a context Γ in which F and G can be proven logically equivalent. (See Definition 1) In this section we discuss the problem of how to prove equivalence of F and G.
Formula G is obtained from F by making a sequence of replacements on subformulas. The replacements made are justified by some equivalance, which then have to lifted into a context by functional reflexivity axioms.

Example 1. Suppose that we want to transform $(A_1 \wedge A_2) \vee B_1 \vee \cdots \vee B_n$ into Clausal Normal Form. We assume that $\vee$ is left-associative and binary. First $(A_1 \wedge A_2) \vee B_1$ has to replaced by $(A_1 \vee B_1) \wedge (A_2 \vee B_1)$. The result is $((A_1 \vee B_1) \wedge (A_2 \vee B_1)) \vee B_2 \vee \cdots \vee B_n$. Then $((A_1 \vee B_1) \wedge (A_2 \vee B_1) \vee B_2)$ is replaced by $(A_1 \vee B_1 \vee B_2) \wedge (A_2 \vee B_1 \vee B_2)$. n such replacements result in the CNF $(A_1 \vee B_1 \vee \cdots \vee B_n) \wedge (A_2 \vee B_1 \vee \cdots \vee B_n)$.
The i-th replacement can be justified by lifting the proper instantiation of the axiom $(P \wedge Q) \vee R \leftrightarrow (P \vee R) \wedge (Q \vee R)$ into the context $(\#) \wedge B_i \wedge \cdots \wedge B_n$. This can be done by taking the right instantiation of the axiom $(P_1 \leftrightarrow Q_1) \rightarrow (P_2 \leftrightarrow Q_2) \rightarrow (P_1 \wedge P_2 \leftrightarrow Q_1 \wedge Q_2)$.

The previous example gives the general principle with which proofs are to be generated. In nearly all cases the replacement can be justified by direct instantiation of an axiom. In most cases the transformations can be specified by a rewrite system combined with a strategy, usually outermost replacement.

In order to make proof generation feasible, two problems need to be solved: The first is the problem that in type theory, it takes quadratic complexity to build up a context. This is easily seen from Example 1. For the first step, the functional reflexivity axiom needs to be applied $n-1$-times. Each time, it needs to be applied on the formula constructed so far. This causes quadratic complexity.

The second problem is the fact that the same context will be built up many times. In Example 1, the first two replacements both take place in context $(\#) \vee B_3 \vee \cdots \vee B_n$. All replacements, except the last take place in context $(\#) \vee B_n$. It is easily seen that in Example 1, the total proof has size $O(n^3)$. The size of the result is only $2n$.

Our solution to the problem is based on two principles: Reducing the redundancy in proof representation, and combination of contexts.

Type theory is extremely redundant. If one applies a proof rule, one has to mention the formulas on which the rule is applied, even though this information can be easily derived. In [4], it has been proposed to obtain proof compression by leaving out redundant information. However, even if one does not store the formulas, they are still generated and compared during proof checking, so the order of proof checking is not reduced. (If one uses type theory. It can be different in other calculi) We solve the redundancy problem by introducing abbreviations for repeated formulas. This has the advantage that the complexity of checking the proof is also reduced, not only of storing.

The problem of repeatedly building up the same context can be solved by first combining proof steps, before building up the context. One could obtain this by tuning the strategy that makes the replacements, but that could be hard for some strategies. Therefore we take another approach. We define a calculus in which repeated constructions of the same context can be normalized away. We call this calculus the *replacement calculus.* Every proof has a unique normal form. When a proof is in normal form, there is no repeated build up of contexts. Therefore, it corresponds to a minimal proof in type theory. The replacement calculus is somewhat related to the rewriting calculus of [7], but it is not restricted to rewrite proofs, although it can be used for rewrite proofs. Another difference is that our calculus is not intended for doing computations, only for concisely representing replacement proofs.

Definition 3. *We recursively define what is a valid replacement proof π in a context Γ. At the same time, we associate an equivalence $\Delta(\pi)$ of form $A \equiv B$ to each valid replacement proof, called the* conclusion *of π.*

- *If formula A is well-typed in context Γ, then* $\text{refl}(A)$ *is a valid proof in the replacement calculus. Its conclusion is $A \equiv A$.*
- *If π_1, π_2 are valid replacemet proofs in context Γ, and there exist formulas A, B, C, s.t. $\Delta(\pi_1)$ equals $(A \equiv B)$, $\Delta(\pi_2)$ equals $(B \equiv C)$, then* $\text{trans}(\pi_1, \pi_2)$ *is a valid replacement proof with conclusion $(A \equiv C)$ in Γ.*

- *If* $\pi_1, \ldots, \pi_n$ *are valid replacement proofs in* Γ, *for which* $\Delta(\pi_1) = (A_1 \equiv B_1), \ldots, \Delta(\pi_n) = (A_n \equiv B_n)$, *both* $f(A_1, \ldots, A_n)$ *and* $f(B_1, \ldots, B_n)$ *are well-typed in* Γ, *then* $\mathrm{func}(f, \pi_1, \ldots, \pi_n)$ *is a valid replacement proof with conclusion* $f(A_1, \ldots, A_n) \equiv f(B_1, \ldots, B_n)$ *in* Γ.
- *If* π *is a valid replacement proof in a context of form* $\Gamma, x{:}X$, *with* $\Delta(\pi) = (A \equiv B)$, *the formulas* A, B *are well-typed in context* $\Gamma, x{:}X$, *then* $\mathrm{abstr}(x, X, \pi)$ *is a valid replacement proof, with conclusion* $(\lambda x{:}X\ A) \equiv (\lambda x{:}X\ B)$.
- *If* $\Gamma \vdash t{:}A \equiv B$, *then* $\mathrm{axiom}(t)$ *is a valid replacement proof in* Γ, *with conclusion* $A \equiv B$

In a concrete implementation, there probably will be additional constraints. For example use of the refl-, trans-rules will be restricted to certain types. Similarly, use of the func-rule will probably be restricted.

The $\equiv$-relation is intended as an abstraction from the concrete equivalence relation being used. In our situation, $\equiv$ should be read as $\leftrightarrow$ on Prop, and it could be equality on domain elements. In addition, one could have other equivalence relations, for which functional reflexivity axioms exist. (Actually not a full equivalence relation is needed. Any relation that is reflexive, transitive, and that satisfies at least one axiom of form $A \succ B \Rightarrow s(A) \succ s(B)$ could be used)

The abstr-rule is intended for handling quantifiers. A formula of form $\forall x{:}X\ P$ is represented in typetheory by (forall $\lambda x{:}X\ P$). If one wants to make a replacement inside P, one first has to apply the abstr-rule, and then to apply the refl-rule on forall. In order to be able to make such replacements, one needs an additional equivalence relation equivProp, such that (equivProp $P\ Q$) $\rightarrow$ (forall P) $\leftrightarrow$ (forall Q). This can be easily obtained by defining equivProp as $\lambda X{:}\mathrm{Set}\ \lambda P, Q{:}X \rightarrow \mathrm{Prop}\ \forall x{:}X\ (P\ x) \leftrightarrow (Q\ x)$.

We now define two translation functions that translate replacement proofs into type theory proofs. The first function is fairly simple. It uses the method that was used in Example 1. The disadvantage of this method is that the size of the constructed proof term can be quadratic in the size of the replacement proof. On the other hand it is simple, and for some applications it may be good enough. The translation assumes that we have for each type of discourse terms of type refl_X, and trans_X available. In addition, we assume availability of terms of type func_f with obvious types.

Definition 4. *The following axioms are needed for translating proofs of the rewrite calculus into type theory.*

- refl_X *is a proof of* $\Pi x{:}X\ X \equiv X$.
- trans_X *is a proof of* $\Pi x_1, x_2, x_3{:}X\quad x_1 \equiv x_2 \rightarrow x_2 \equiv x_3 \rightarrow x_1 \equiv x_3$.
- func_f *is a proof of* $\Pi x_1, y_1{:}X_1\ \cdots \Pi x_n, y_n{:}X_n\quad x_1 \equiv y_1 \rightarrow \cdots \rightarrow x_n \equiv y_n \rightarrow (f\ x_1 \cdots x_n) \equiv (f\ y_1 \cdots y_n)$. *Here* $X_1, \ldots, X_n$ *are the types of the arguments of* f.

Definition 5. *Let* π *be a valid replacement proof in context* Γ. *We define translation function* $T(\pi)$ *by recursion on* π.

- $T(\mathrm{refl}(A)\ )$ *equals* $(\mathrm{refl}_X\ A)$, *where* X *is the type of* A.

- $T(\text{trans}(\pi_1, \pi_2)\ )$ *equals as* $(\text{trans}_X\ A\ B\ C\ T(\pi_1)\ T(\pi_2)\ )$, *where* A, B, C *are defined from* $\Delta(\pi_1) = (A \equiv B)$ *and* $\Delta(\pi_2) = (B \equiv C)$.
- $T(\text{func}(f, \pi_1, \ldots, \pi_n)\ )$ *is defined as* $(\text{func}_f\ A_1\ B_1 \cdots\ A_n\ B_n\ T(\pi_1) \cdots T(\pi_n)\ )$, *where* A_i, B_i *are defined from* $\Delta(\pi_i) = (A_i \equiv B_i)$, *for* $1 \leq i \leq n$.
- $T(\text{abstr}(x, X, \pi)\ )$ *is defined as* $(\text{abstr}_X\ (\lambda x{:}X\ A)\ (\lambda x{:}X\ B)\ (\lambda x{:}X\ T(\pi)\ )\ )$, *where* A, B *are defined from* $\Delta(\pi) = (A \equiv B)$.
- $T(\text{axiom}(t))$ *is defined simply as* t.

Theorem 3. *Let* π *be a valid replacement proof in context* Γ. *Then* $|T(\pi)| = O(|\pi|^2)$.

Proof. The quadratic upperbound can be shown by induction. That this upperbound is also a lowerbound was demonstrated in Example 1.

Next we define an improved translation function that constructs a proof of size linear in the size of the replacement proof. The main idea is to introduce definitions for all subformulas. In this way, the iterated built-ups of subformulas can be avoided. In order to introduce the definitions, proof scripts with lemma-instructions are constructed simultaneously with the translations.

Definition 6. *Let* π *be a valid replacement proof in context* Γ. *The improved translation function* $T(\pi)$ *returns a quadruple* (Σ, t, A, B), *where* Σ *is a proof script and* t *is a term such that* $\Gamma, \Sigma \vdash t{:}A \equiv B$. *(The notation* Γ, Σ *means:* Γ *extended with the definitions induced by* Σ*)*

- $T(\text{refl}(A)\ )$ *equals* $(\emptyset, (\text{refl}_X\ A), A, A)$, *where* X *is the type of* A.
- $T(\text{trans}(\pi_1, \pi_2)\ )$ *is defined as* $(\Sigma_1 \cup \Sigma_2, (\text{trans}_X\ A\ B\ C\ t_1\ t_2), A, C)$, *where* $\Sigma_1, \Sigma_2, t_1, t_2, A, C$ *are defined from*

$$T(\pi_1) = (\Sigma_1, t_1, A, B), \quad T(\pi_2) = (\Sigma_2, t_2, B, C).$$

- $T(\text{func}(f, \pi_1, \ldots, \pi_n)\ )$ *is defined as*

$$(\Sigma_1 \cup \cdots \cup \Sigma_n \cup \Sigma, (\text{func}_f\ A_1\ B_1 \cdots\ A_n\ B_n\ t_1 \cdots\ t_n), x_1, x_2),$$

where, for i *with* $1 \leq i \leq n$, *the* Σ_i, A_i, B_i, t_i *are defined from*

$$T(\pi_i) = (\Sigma_i, t_i, A_i, B_i).$$

Both x_1, x_2 *are new atoms, and* Σ *is defined from*

$$\Sigma = \{\text{lemma}(x_1, (f\ A_1\ \cdots\ A_n), X),\ \text{lemma}(x_2, (f\ B_1\ \cdots\ B_n), X)\},$$

where X *is the common type of* $(f\ A_1 \cdots A_n)$ *and* $(f\ B_1 \cdots B_n)$.
- $T(\text{abstr}(x, X, \pi)\ )$ *is defined as*

$$(\Sigma \cup \Theta, (\text{abstr}_X\ (\lambda x{:}X\ A)\ (\lambda x{:}X\ B)\ (\lambda x{:}X\ t), x_1, x_2),$$

where Σ, t, A, B *are defined from* $T(\pi) = (\Sigma, t, A, B)$. *The* x_1, x_2 *are new atoms, and*

$$\Theta = \{\text{lemma}(x_1, (\lambda x{:}X\ A), X \to Y),\ \text{lemma}(x_2, (\lambda x{:}X\ B), X \to Y)\}.$$

– $T(\text{axiom } t)$ *is defined as* $(\emptyset, t, A, B)$, *where* A, B *are defined from* $\Gamma \vdash t{:}A \equiv B$.

Definition 7. *We define the following reduction rules on replacement proofs. Applying* trans *on a* refl-*proof does not change the equivalence being proven:*

– $\text{trans}(\pi, \text{refl}(A)) \Rightarrow \pi$,
– $\text{trans}(\text{refl}(A), \pi) \Rightarrow \pi$.

The trans-*rule is associative. The following reduction groups* trans *to the left:*

– $\text{trans}(\pi, \text{trans}(\rho, \sigma)) \Rightarrow \text{trans}(\text{trans}(\pi, \rho), \sigma)$.

If the func-*rule, or the* abstr-*rule is applied only on* refl-*rules, then it proves an identity. Because of this, it can be replaced by one* refl-*application.*

– $\text{func}(f, \text{refl}(A_1), \dots, \text{refl}(A_n)) \Rightarrow \text{refl}(f(A_1, \dots, A_n))$.
– $\text{abstr}(x, X, \text{refl}(A)) \Rightarrow \text{refl}(\lambda x{:}X\ A)$.

The following two reduction rules are the main ones. If a trans-*rule, or an* abstr-*rule is applied on two proofs that build up the same context, then the context building can be shared:*

– $\text{trans}(\text{func}(f, \pi_1, \dots, \pi_n), \text{func}(f, \rho_1, \dots, \rho_n)) \Rightarrow$
$\text{func}(f, \text{trans}(\pi_1, \rho_1), \dots, \text{trans}(\pi_n, \rho_n))$.
– $\text{trans}(\text{abstr}(x, X, \pi), \text{abstr}(x, X, \rho)) \Rightarrow \text{abstr}(x, X, \text{trans}(\pi, \rho)\ )$.

Theorem 4. *The rewrite rules of Definition 7 are terminating. Moreover, they are confluent. For every proof* π, *the normal form* π' *corresonds to a type-theory proof of minimal complexity.*

Now a proof can be generated naively in the replacment calculus, after that it can be normalized, and from that, a type theory proof can be generated.

4 Skolemization Issues

We discuss the problem of generating proofs from Skolemization steps. Witness-instructions can be used to introduce the Skolem functions into the proof scripts, see Definition 1. The wittness-instructions can be justified by either a choice axiom or by the ϵ-function.

It would be possible to completely eliminate the Skolem-functions from the proof, but we prefer not to do that for efficiency reasons. Elimination of Skolem-functions may cause hyperexponential increase of the size of the proof, see [2]. This would make proof generation not feasible. However, we are aware of the fact that for some applications, it may be necessary to perform the elimination of Skolem functions. Methods for doing this have been studied in [9] and [14]

It is straightforward to handle standard Skolemization using of a witness-instruction. However, several improved Skolemization methods have been proposed, in particular *optimized Skolemization* [13] and *strong Skolemization.* (see [11] or [12]) Experiments show that such improved Skolemization methods do improve the chance of finding a proof. Therefore, we need to be able to handle these methods. In order to obtain this, we will show that both strong and optimized Skolemization can be reduced to standard Skolemization. Formally this means the following: For every first-order formula F, there is a first-order formula F', which is first-order equivalent to F, such that the standard Skolemization of F' equals the strong/optimized Skolemization of F. Because of this, no additional choice axioms are needed to generate proofs from optimized or strong Skolemization steps. An additional consequence of our reduction is that the Skolem-elimination techniques of [9] and [14] can be applied to strong and optimized Skolemization as well, without much difficulty.

The reductions proceed through a new type of Skolemization that we call *stratified* Skolemization. Both strong and improved Skolemization can be reduced to stratified Skolemization (in the way that we defined a few lines above). Stratified Skolemization in its turn can be reduced to standard Skolemization. This solves the question that was asked in the last line of [11] whether or not it is possible to unify strong and optimized Skolemization.
We now repeat the definitions of inner and outer Skolemization, which are standard. (Terminology from [12]) After that we give the definitions of strong and optimized Skolemization.

Definition 8. *Let F be a formula in* NNF. *Skolemization replaces an outermost existential quantifier by a new function symbol. We define four types of Skolemization. In order to avoid problems with variables, we assume that F is standardized apart. Write $F = F[\ \exists y{:}Y\ A,]$, where $\exists y{:}Y\ A$ is not in the scope of another existential quantifier. We first define outer Skolemization, after that we define the three other type of Skolemization.*

Outer Skolemization *Let $x_1, \ldots, x_p$ be the variables belonging to the universal quantifiers which have $\exists y{:}Y\ A$ in their scope. Let $X_1, \ldots, X_p$ be the corresponding types. Let f be a new function symbol of type $X_1 \rightarrow \cdots \rightarrow X_p \rightarrow Y$. Then replace $F[\exists y{:}Y\ A]$ by $F[A\ [y := (f\ x_1 \cdots x_p)]\ ]$.*

With the other three types of Skolemization, the Skolem functions depend only on the universally quantified variables that actually occur in A. Let $x_1, \ldots, x_p$ be the variables that belong to the universal quantifiers which have A in their scope, and that are free in A. The $X_1, \ldots, X_p$ are the corresponding types.

Inner Skolemization *Inner Skolemization is defined in the same way as outer Skolemization, but it uses the improved $x_1, \ldots, x_p$.*

Strong Skolemization *Strong Skolemization can be applied only if formula A has form $A_1 \wedge \cdots \wedge A_q$ with $q \geq 2$. For each k, with $1 \leq k \leq q$, we first define the sequence of variables $\overline{\alpha}_k$ as those variables from $(x_1, \ldots, x_p)$ that do not occur in $A_k \wedge \cdots \wedge A_q$. It can be easily checked that for $1 \leq k < q$, sequence $\overline{\alpha}_k$ is a subsequence of $\overline{\alpha}_{k+1}$.*
For each k with $1 \leq k \leq q$, write $\overline{\alpha}_k$ as $(v_{k,1}, \ldots, v_{k,l_k})$. Write $(V_{k,1}, \ldots, V_{k,l_k})$ for the corresponding types. Define the functions $\overline{Q}_k$ from

$$\overline{Q}_k(Z) = \forall v_{k,1}{:}V_{k,1} \cdots\ \forall v_{k,l_k}{:}V_{k,l_k}\ (Z),$$

It is intended that the quantifiers $\forall v_{k,j}{:}V_{k,j}$ will capture the free atoms of Z. Let f be new function symbol of type $X_1 \rightarrow \cdots \rightarrow X_p \rightarrow Y$. For each k, with $1 \leq k \leq q$, define $B_k = A_k[y := (f\ x_1 \cdots x_p)]$. Finally replace

$$F[\exists y{:}Y\ (A_1 \wedge A_2 \wedge \cdots \wedge A_q)] \text{ by } F[\overline{Q}_1(B_1) \wedge \overline{Q}_2(B_2) \wedge \cdots \wedge \overline{Q}_q(B_q)].$$

Optimized Skolemization *Formula A must have form $A_1 \wedge A_2$, and F must have form $F_1 \wedge \cdots \wedge F_q$, where one of the F_k, $1 \leq k \leq q$ has form*

$$F_k = \forall x_1{:}X_1\ \forall x_2{:}X_2 \cdots \forall x_p{:}X_p\ \exists y{:}Y\ A_1.$$

If this is the case, then $F[\ \exists y{:}Y\ (A_1 \wedge A_2)]$ can be replaced by the formula

$$F_k[A_2[y := (f\ x_1 \cdots x_p)]\ ],$$

and F_k can be simultaneously replaced by the formula

$$\forall x_1{:}X_1\ \forall x_2{:}X_2\ \cdots\ \forall x_p{:}X_p\ A_1[y := (f\ x_1 \cdots x_p)].$$

If F is not a conjunction or does not contain an F_k of the required form, but it does imply such a formula, then optimized Skolemization can still be used. First replace F by $F \wedge \forall x_1{:}X_1\ \forall x_2{:}X_2 \cdots \forall x_p{:}X_p\ \exists y{:}Y\ A_1$, and then apply optimized Skolemization.

As said before, choice axioms or ϵ-functions can be used in order to justify the wittness-instructions that introduce the Skolem-funcions. This is straightforward, and we omit the details here.

In the rest of this section, we study the problem of generating proofs for optimized and strong Skolemization. We want to avoid introducing additional axioms, because strong Skolemization has too many parameters. (The number of conjuncts, and the distribution of the $x_1, \ldots, x_p$ through the conjuncts). We will obtain this by reducing strong and optimized Skolemization to inner Skolemization. The reduction proceeds through a new type of Skolemization, which we call *Stratified Skolemization.* We show that Stratified Skolemization can be obtained from inner Skolemization in first-order logic. In the process, we answer a question asked in [11], whether or not there a common basis in strong and optimized Skolemization.

Definition 9. *We define* stratified Skolemization. *Let F be some first-order formula in negation normal form. Assume that F contains a conjunction of the form $F_1 \wedge \cdots \wedge F_q$ with $2 \leq q$, where each F_k has form*

$$\forall x_1{:}X_1 \cdots\ x_p{:}X_p\ (C_k \rightarrow \exists y{:}Y\ A_1 \wedge \cdots \wedge A_k).$$

The C_k and A_k are arbitrary formulas. It is assumed that the F_k have no free variables. Furthermore assume that for each k, $1 \leq k < q$, the following formula is provable:

$$\forall x_1{:}X_1 \cdots\ x_p{:}X_p\ (C_{k+1} \rightarrow C_k).$$

Then $F[\ F_1 \wedge \cdots \wedge F_q]$ can be Skolemized into $F[F_1' \wedge \cdots \wedge F_q']$, where each F_k', $1 \leq k \leq q$ has form

$$\forall x_1{:}X_1 \cdots\ x_p{:}X_p\ (C_k \rightarrow A_k[\ y := (f\ x_1 \cdots\ x_p)\ ]\ ).$$

As with optimized and strong Skolemization, it is possible to Skolemize more than one existential quantifier at the same time. Stratified Skolemization improves over standard Skolemization by the fact that it allows to use the same Skolem-function for existential quantifiers, which is an obvious improvement. In addition, it is allowed to drop all but the last members from the conjunctions on the righthandsides. It is not obvious that this is an improvement.

The $C_1, \dots, C_q$ could be replaced by any context through a subformula replacement.

We now show that stratified Skolemization can be reduced to inner Skolemization. This makes it possible to use a standard choice axiom for proving the correctness of a stratified Skolemization step.

Theorem 5. *Stratified Skolemization can be reduced to inner Skolemization in first-order logic. More precisely, there exists a formula G, such that F is logically equivalent to G in first-order logic, and the stratified Skolemization of F equals the inner Skolemization of G.*

Proof. Let $F_1, \dots, F_q$ be defined as in Definition 9. Without loss of generality, we assume that F is equal to $F_1 \wedge \cdots \wedge F_q$. The situation where F contains $F_1 \wedge \cdots \wedge F_q$ as a subformula can be easily obtained from this.

For G, we take

$$\forall x_1{:}X_1 \cdots \forall x_p{:}X_p\ \exists y{:}Y\ (C_1 \rightarrow A_1) \wedge \cdots \wedge (C_q \rightarrow A_q).$$

It is easily checked that the inner Skolemization of G equals the stratified Skolemization of F, because y does not occur in the C_k.
We will show that for all $x_1, \dots, x_p$, the instantiated formulas are equivalent, so we need to prove for abitrary $x_1, \dots, x_p$,

$$\bigwedge_{k=1}^{q} C_k \rightarrow \exists y{:}Y\ (A_1 \wedge \cdots \wedge A_k) \Leftrightarrow \exists y{:}Y\ \bigwedge_{k=1}^{q} (C_k \rightarrow A_k).$$

We will use the abbreviation LHS for the left hand side, and RHS for the right hand side.
Define $D_0 = \neg C_1 \wedge \cdots \wedge \neg C_q$.
For $1 < k < q$, define

$$D_k = C_1 \wedge \cdots \wedge C_k \wedge \neg C_{k+1} \wedge \cdots \wedge \neg C_q.$$

Finally, define $D_q = C_1 \wedge \cdots \wedge C_q$.
It is easily checked that $(C_2 \rightarrow C_1) \wedge \cdots \wedge (C_q \rightarrow C_{q-1})$ implies $D_0 \vee \cdots \vee D_q$.

Assume that the LHS holds. We proceed by case analysis on $D_0 \vee \cdots \vee D_q$. If D_0 holds, then RHS can be easily shown for an arbitrary y. If a D_k with $k > 0$ holds, then C_k holds. It follows from the k-th member of the LHS, that there is a y such that the $A_1, \dots, A_k$ hold. Since $k' > k$ implies $\neg C_{k'}$, the RHS can be proven by chosing the same y.

Now assume that the RHS holds. We do another case analysis on $D_0 \vee \cdots \vee D_q$. Assume that D_k holds, with $0 \leq k \leq q$.

For $k' > k$, we then have $\neg C_{k'}$. There is a $y{:}Y$, such that for all $k' \leq k$, $A_{k'}$ holds. Then the LHS can be easily proven by choosing the same y in each of the existential quantifiers.

Theorem 6. *Optimized Skolemization can be trivially obtained from stratified Skolemization.*

Proof. Take $q = 2$ and take for C_1 the universally true predicate.

Theorem 7. *Strong Skolemization can be obtained from stratified Skolemization in first-order logic.*

Proof. We want to apply strong Skolemization on the following formula

$$\forall x_1{:}X_1 \cdots \forall x_p{:}X_p \ (C \ x_1 \ \cdots \ x_p) \rightarrow \exists y{:}Y \ \ A_1 \wedge \cdots \wedge A_q.$$

For sake of clarity, we write the variables in C explicitly. First reverse the conjunction into

$$\forall x_1{:}X_1 \cdots \forall x_p{:}X_p \ (C \ x_1 \cdots \ x_p) \rightarrow \exists y{:}Y \ \ A_q \wedge \cdots \wedge A_1.$$

Let $\overline{\alpha}_1, \ldots, \overline{\alpha}_q$ be defined as in Definition 8. The fact that A_k does not contain the variables in $\overline{\alpha}_k$ can be used for weakening the assumptions $(C \ x_1 \cdots \ x_p)$ as follows:

$$\bigwedge_{k=q}^{1} \forall x_1{:}X_1 \cdots \forall x_p{:}X_p \ \ [\ \exists \overline{\alpha}_k \ (C \ x_1 \cdots x_p)\ ] \rightarrow \exists y{:}Y \ A_q \wedge \cdots \wedge A_k.$$

Note that k runs backwards from q to 1. Because $\overline{\alpha}_k \subseteq \overline{\alpha}_{k+1}$, we have $\exists \overline{\alpha}_k \ (C \ x_1 \cdots x_p)$ implies $\exists \overline{\alpha}_{k+1} \ (C \ x_1 \cdots x_p)$. As a consequence, stratified Skolemization can be applied. The result is:

$$\bigwedge_{k=q}^{1} \forall x_1{:}X_1 \cdots \forall x_p{:}X_p \ [\ \exists \overline{\alpha}_k \ (C \ x_1 \cdots x_p)\ ] \rightarrow A_k[y := (f \ x_1 \cdots x_p)\ ].$$

For each k with $1 \leq k \leq$, let $\overline{\beta}_k$ be the variables of $(x_1, \ldots, x_p)$ that are not in $\overline{\alpha}_k$. Then the formula can be replaced by

$$\bigwedge_{k=q}^{1} \forall \overline{\alpha}_k \ \forall \overline{\beta}_k \ [\ \exists \overline{\alpha}_k \ (C \ x_1 \cdots x_p)\ ] \rightarrow A_k[y := (f \ x_1 \cdots x_p)\ ].$$

This can be replaced by

$$\bigwedge_{k=q}^{1} \forall \overline{\beta}_k \ [\ \exists \overline{\alpha}_k \ (C \ x_1 \cdots x_p)\ ] \rightarrow \forall \overline{\alpha}_k \ A_k[y := (f \ x_1 \cdots x_p)\ ],$$

which can in turn be replaced by

$$\bigwedge_{k=q}^{1} \forall \overline{\beta}_k \ \forall \overline{\alpha}_k \ (C \ x_1 \cdots x_p) \rightarrow \forall \overline{\alpha}_k \ \ A_k[y := (f \ x_1 \cdots x_p)\ ],$$

The result follows immediately.

It can be concluded that strong and optimized Skolemization can be reduced to Stratified Skolemization, which in its turn can be reduced to inner Skolemization. It is an interesting question whether or not Stratified Skolemization has useful applications on its own. We intend to look into this.

5 Conclusions

We have solved the main problems of proof generation from the clausal normal form transformation. Moreover, we think that our techniques are wider in scope: They can be used everywhere, where explicit proofs in type theory are constructed by means of rewriting, automated theorem proving, or modelling of computation.

We also reduced optimized and strong Skolemization to standard Skolemization. In this way, only standard choice axioms are needed for translating proofs involving these forms of Skolemization. Alternatively, it has become possible to remove applications of strong and optimized Skolemization commpletely from a proof.

We do intend to implement a clausal normal tranformer, based on the results in this paper. The input is a first-order formula. The output will be the clausal normal form of the formula, together with a proof of its correctness.

References

1. Matthias Baaz, Uwe Egly, and Alexander Leitsch. Normal form transformations. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 5, pages 275–333. Elsevier Science B.V., 2001.
2. Matthias Baaz and Alexander Leitsch. On skolemization and proof complexity. *Fundamenta Informatika*, 4(20):353–379, 1994.
3. Henk Barendregt and Herman Geuvers. Proof-assistents using dependent type systems. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, volume II, chapter 18, pages 1151–1238. Elsevier Science B.V., 2001.
4. Stefan Berghofer and Tobias Nipkow. Proof terms for simply typed higher order logic. In Mark Aagaard and John Harrison, editors, *Theorem Proving in Higher-Order Logics, TPHOLS 2000*, volume 1869 of *LNCS*, pages 38–52. Springer Verlag, 2000.
5. Marc Bezem, Dimitri Hendriks, and Hans de Nivelle. Automated proof construction in type theory using resolution. In David McAllester, editor, *Automated Deduction - CADE-17*, number 1831 in LNAI, pages 148–163. Springer Verlag, 2000.
6. Samuel Boutin. Using reflection to build efficient and certified decision procedures. In Martín Abadi and Takayasu Ito, editors, *Theoretical Aspects of Computer Software (TACS)*, volume 1281 of *LNCS*, pages 515–529, 1997.
7. Horatiu Cirstea and Claude Kirchner. The rewriting calculus, part 1 + 2. *Journal of the Interest Group in Pure and Applied Logics*, 9(3):339–410, 2001.
8. Hans de Nivelle. A resolution decision procedure for the guarded fragment. In Claude Kirchner and Hélène Kirchner, editors, *Automated Deduction- CADE-15*, volume 1421 of *LNCS*, pages 191–204. Springer, 1998.

9. Xiaorong Huang. Translating machine-generated resolution proofs into ND-proofs at the assertion level. In Norman Y. Foo and Randy Goebel, editors, *Topics in Artificial Intelligence, 4th Pacific Rim International Conference on Artificial Intelligence*, volume 1114 of *LNCS*, pages 399–410. Springer Verlag, 1996.
10. William McCune and Olga Shumsky. Ivy: A preprocessor and proof checker for first-order logic. In Matt Kaufmann, Pete Manolios, and J. Moore, editors, *Using the ACL2 Theorem Prover: A tutorial Introduction and Case Studies.* Kluwer Academic Publishers, 2002? preprint: ANL/MCS-P775-0899, Argonne National Labaratory, Argonne.
11. Andreas Nonnengart. Strong skolemization. Technical Report MPI-I-96-2-010, Max Planck Institut für Informatik Saarbrücken, 1996.
12. Andreas Nonnengart and Christoph Weidenbach. Computing small clause normal forms. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 6, pages 335–367. Elsevier Science B.V., 2001.
13. Hans Jürgen Ohlbach and Christoph Weidenbach. A note on assumptions about skolem functions. *Journal of Automated Reasoning*, 15:267–275, 1995.
14. Frank Pfenning. Analytic and non-analytic proofs. In Robert E. Shostak, editor, *7th International Conference on Automated Deduction CADE 7*, volume 170 of *LNCS*, pages 394–413. Springer Verlag, 1984.
15. Frank Pfenning. Logical frameworks. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, volume II, chapter 17, pages 1065–1148. Elsevier Science B.V., 2001.
16. R. Sekar, I.V. Ramakrishnan, and Andrei Voronkov. Term indexing. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, volume 2, chapter 26, pages 1853–1964. Elsevier Science B.V., 2001.
17. Christoph Weidenbach. The spass homepage. `http://spass.mpi-sb.mpg.de/` .
18. Christoph Weidenbach. Combining superposition, sorts and splitting. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, volume II, chapter 27, pages 1965–2013. Elsevier Science B.V., 2001.