Sonstige wissenschafliche Arbeiten

zu der Habilitationsschrift

Using Resolution as Decision Procedure

Dr. Ir. Jean Marie Guillaume Gérard de Nivelle
Scheidterstrasse 62
66125 Dudweiler
Deutschland

Juni 2005

Sonstige Wissenschaftliche Arbeiten

Dieser Band enthält die sonstigen wissenschaftliche Arbeiten, die ich zusammen mit meiner Habilitationsschrift einreiche. Die ersten vier Arbeiten beschäftigen sich mit Beweiserzeugung durch Theorembeweiser, die fünfte mit Implementierung von Resolution.

Automated Proof Construction in Type Theory using Resolution[†]

Marc Bezem (bezem@ii.uib.no)
University of Bergen, Department of Informatics

Dimitri Hendriks (hendriks@phil.uu.nl)
Utrecht University, Department of Philosophy

Hans de Nivelle (nivelle@mpi-sb.mpg.de)
Max Planck Institut für Informatik, Saarbrücken

Abstract. We provide techniques to integrate resolution logic with equality in type theory. The results may be rendered as follows.

- A clausification procedure in type theory, equipped with a correctness proof, all encoded using higher-order primitive recursion.
- A novel representation of clauses in minimal logic such that the λ-representation of resolution steps is linear in the size of the premisses.
- A translation of resolution proofs into lambda terms, yielding a verification procedure for those proofs.
- The power of resolution theorem provers becomes available in interactive proof construction systems based on type theory.

1. Introduction

Type theory (= typed lambda calculus, with dependent products as most relevant feature) offers a powerful formalism for formalizing mathematics. Strong points are: the logical foundation, the fact that proofs are first-class citizens and the generality which naturally facilitates extensions, such as inductive types. Type theory captures definitions, reasoning and computation at various levels in an integrated way. In a type-theoretical system, formalized mathematical statements are represented by types, and their proofs are represented by λ-terms. The problem whether π is a proof of statement A reduces to checking whether the term π has type A. Computation is based on a simple notion of rewriting. The level of detail is such that the well-formedness of definitions and the correctness of derivations can automatically be verified.

However, there are also weak points. It is exactly the appraised expressivity and the level of detail that makes automation at the same

[†] Extended and modified version of [2].

time necessary and difficult. Automated deduction appears to be mostly successful in weak systems, such as propositional logic and predicate logic, systems that fall short to formalize a larger body of mathematics. Apart from the problem of the expressivity of these systems, only a minor part of the theorems that can be expressed can actually be proved automatically. Therefore it is necessary to combine automated theorem proving with interactive theorem proving. Recently a number of proposals in this direction have been made. In [4] a two-level approach (called *reflection*) is used to develop in Coq a certified decision procedure for equations in abelian rings. In the same vein, [15] certifies ELAN traces in Coq. In [13] Otter is combined with the Boyer-Moore theorem prover. (A verified program rechecks proofs generated by Otter.) In [12] Gandalf is linked to HOL. (The translation generates scripts to be run by the HOL-system.) In [20], proofs are translated into Martin-Löf's type theory, for the Horn clause fragment of first-order logic. In the Omega system [10, 16] various theorem provers have been linked to a natural deduction proof checker. The purpose there is to automatically generate proofs from so called *proof plans*. Our approach is different in that we generate complete proof objects for both the clausification and the refutation part.

Resolution theorem provers, such as Bliksem [3], are powerful, but have the drawback that they work with normal forms of formulae, so-called clausal forms. Clauses are (universally closed) disjunctions of literals, and a literal is either an atom or a negated atom. The clausal form of a formula is essentially its 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 interactive navigation of the theorem prover through the search space very difficult. Moreover, optimized implementations of proof procedures are error-prone. It has occurred that systems that took part in the yearly theorem prover competition CASC had to withdraw afterwards, due to the fact that the system turned out unsound. In one year (1999) the system that otherwise would have won the MIX category was withdrawn, see [19].

In type theory, 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. In order to make the formalisation of a large body of mathematics feasible, the level of automation of interactive proof construction systems such as Coq [1], based on type theory, has to be improved.

We propose the following proof procedure. Identify a non-trivial step in a Coq session that amounts to a first-order tautology. Export this tautology to Bliksem, and delegate the proof search to the Bliksem inference engine. Convert the resolution proof to type theoretic format and import the result back in Coq. We stress the fact that the above procedure is as secure as Coq. Hypothetical errors (e.g. the clausification procedure not producing clauses, possible errors in the resolution theorem prover or the erroneous formulation of the lambda terms corresponding to its proofs) are intercepted because the resulting proofs are type-checked by Coq. The security could be made independent of Coq by using another type-checker.

Most of the necessary meta-theory is already known. The negation normal form transformation can be axiomatized by classical logic. The prenex and conjunctive normal form transformations require that the domain is non-empty. 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.

Following the proof of the conservativity of the Axiom of Choice over first-order logic, see e.g. [18] (elaborated in [6]) and [17], Skolem functions and –axioms could be eliminated from resolution proofs, which would allow us to obtain directly a proof of the original formula, but currently we still make use of the Axiom of Choice.

The paper is organized as follows. In Section 2 we set out a two-level approach and define a deep embedding to represent first-order logic. Section 3 describes a uniform clausification procedure. We explain how resolution proofs are translated into λ-terms in Sections 4 and 5. Finally, the outlined constructions are demonstrated in Section 6.

2. A two-level approach

The basic sorts in Coq are $*^p$ and $*^s$. An object M of type $*^p$ denotes a logical proposition and objects of type M denote proofs of M. Objects of type $*^s$ are usual sets such as the set of natural numbers, lists etc. The typing relation is expressed by $t : T$, to be interpreted as 't belongs to set T' when $T : *^s$, and as 't is a proof of proposition T' when $T : *^p$. The primitive type constructor is the product type $\Pi x : T.\, U$ and is called *dependent* if x occurs in U; if not, we write $T \rightarrow U$. The product type is used for logical quantification (implication) as well as for function spaces. Scopes of Π's, λ's and other binders extend to the right as far as brackets allow ($\rightarrow$ associates to the right). Furthermore, well-typed application is denoted by $(M\ N)$ and associates to the left.

We choose for a deep embedding in adopting a two-level approach for the treatment of arbitrary first-order languages. The idea is to represent first-order formulae as objects in an inductive set $o : *^s$, accompanied by an interpretation function $[\![\_]\!]$ that maps these objects into $*^p$.[1] The next paragraphs explain why we distinguish a higher (*meta-*, *logical*) level $*^p$ and a lower (*object-*, *computational*) level o.

The universe $*^p$ includes higher-order propositions; in fact it encompasses full impredicative type theory. As such, it is too large for our purposes. Given a suitable signature, any first-order formula $\varphi : *^p$ will have a formal counterpart $p : o$ such that φ equals $[\![p]\!]$, the interpretation of p. Thus the first-order fragment of $*^p$ can be identified as a collection of interpretations of objects in o.

Secondly, Coq supplies only limited computational power on $*^p$, whereas o, as every inductive set, is equipped with the powerful computational device of higher-order primitive recursion. This enables the syntactical manipulation of object-level propositions.

Reflection is used for the proof construction of first-order formulae in $*^p$ in the following way. Let $\varphi : *^p$ be a first-order formula. Then there is some $\dot{\varphi} : o$ such that $[\![\dot{\varphi}]\!]$ is convertible with φ.[2] Moreover, suppose we have proved

$$T_{sound} : \Pi p \colon o.\, [\![(T\ p)]\!] \rightarrow [\![p]\!]$$

for some function $T : o \rightarrow o$, typically a transformation to clausal form. Then, to prove φ it suffices to prove $[\![(T\ \dot{\varphi})]\!]$. Matters are presented schematically in Figure 1. In Section 3 we discuss a concrete function T, for which we have proved the above. For this T, proofs of $[\![(T\ \dot{\varphi})]\!]$ will be generated automatically, as will be described in Sections 4 and 5.

Object-level propositions and the reflection operation

In Coq, we have constructed a general framework to represent first-order languages with multiple sorts. Bliksem is one-sorted, so we describe the setup for one-sorted signatures only.

The set o (formulas) defined in the present section depends on the signature—constituted by an arbitrary but fixed list of natural numbers, representing relation arities. This dependence remains implicit in the sequel. We start by giving some preliminary definitions.

DEFINITION 2.1. *Given a set A,* lists *of type* (*list* A) *are defined by* $[\ ]$ *and* $[a|l]$*, where* $a : A$ *and* $l :$ (*list* A)*. Given a list* $L :$ (*list* A)*, its*

[1] Both o as well as $[\![\_]\!]$ depend on a fixed but arbitrary signature.
[2] The mapping $\dot{\_}$ is a syntax-based translation outside Coq.

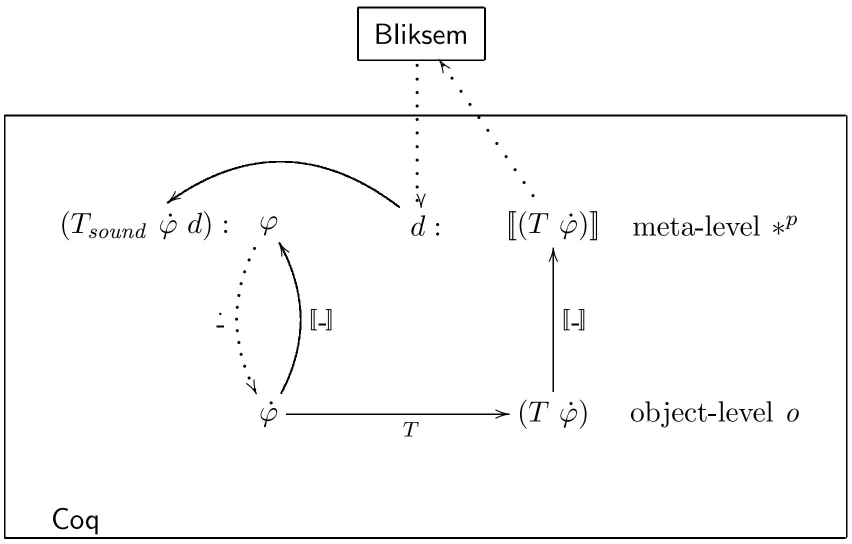


Figure 1. Schematic overview of the general procedure. Arrows correspond to application in Coq, dotted arrows are not performed by Coq. The term $[\![(T\ \dot{\varphi})]\!]$ is computed by Coq and exported to Bliksem. Bliksem is to return a proof term d, which is imported back in Coq. Then $(T_{sound}\ \dot{\varphi}\ d)$ is a proof of $[\![\dot{\varphi}]\!]$, and hence of φ.

index set I_L *is defined by the equations:*

$$I_{[]} = \emptyset \qquad I_{[a|L']} = \mathbf{1} + I_{L'}$$

where $\emptyset$ *is the empty set (i.e. without contructors),* $\mathbf{1}$ *the unit set (with one sole inhabitant* $\bullet$*) and* $A + B$ *is the disjoint sum of sets* A *and* B*. We write* $|L|$ *to denote the length of* L*. For the sake of readability we set* $I_L = \{0, \ldots, |L| - 1\}$*. Furthermore, we write* $L(i)$ *for the element indexed by* $i : I_L$*. The cartesian product of* n *copies of a set* A *is defined by:*

$$A^0 = \mathbf{1} \qquad A^{n+1} = A \times A^n$$

DEFINITION 2.2. *Assume a domain of discourse* $A : *^s$ *and let* l_{rel} *be a list of natural numbers representing arities. The set* o *of objects representing propositions is inductively defined as follows, where* $p, q : o,\ p' : A \to o,\ t_1, \ldots, t_k : A,\quad i : I_{l_{rel}}$ *and* $l_{rel}(i) = k$*.*

$$o := R_i(t_1, \ldots, t_k) \mid \dot{\neg} p \mid p \dot{\to} q \mid p \mathrel{\dot{\wedge}} q \mid p \mathrel{\dot{\vee}} q \mid (\dot{\forall}\ p') \mid (\dot{\exists}\ p')$$

Note that $R : \Pi i : I_{l_{rel}}. A^{l_{rel}(i)} \to o$, we write R_i instead of $(R\ i)$. We use the dot-notation $\dot{}$ to distinguish the object-level constructors from

Coq's predefined connectives. The constructors $\dot{\forall}$, $\dot{\exists}$ are typed $(A \to o) \to o$; they map propositional functions of type $A \to o$ to propositions of type o. This representation has the advantage that binding and predication are handled by λ-abstraction and λ-application. On the object-level, existential quantification of x in p (of type o, possibly containing occurrences of x) is written as $(\dot{\exists}\ (\lambda x : A.\, p))$. Although this representation suffices for our purposes, it causes some well-known difficulties. See [14, Sections 8.3, 9.2] for a further discussion.

For our purposes, a shallow embedding of function symbols is sufficient. We have not defined an inductive set *term* representing the first-order terms in A like we have defined o representing the first-order fragment of $*^p$. Instead, 'meta-level' terms of type A are taken as arguments of object-level predicates. Due to this shallow embedding, we cannot check whether variables have occurrences in a given term. Because of that, e.g., distributing universal quantifiers over conjuncts can yield dummy abstractions. These problems could be overcome by using De Bruijn-indices (see [5]) for a deep embedding of terms in Coq, cf. [8].

DEFINITION 2.3. *The* interpretation *function* $[\![\_]\!]$ *is a canonical homomorphism recursively defined as follows. Assume a family of relations indexed over* $I_{l_{rel}}$.

$$\mathcal{R} : \Pi i : I_{l_{rel}}.\, A^{l_{rel}(i)} \to *^p$$

We write $\mathcal{R}_i$ *for* $(\mathcal{R}\ i)$.

$$\begin{array}{rcl}
[\![R_i(t_1, \ldots, t_k)]\!] & = & \mathcal{R}_i(t_1, \ldots, t_k) \\
[\![\dot{\neg} p]\!] & = & \neg [\![p]\!] \\
[\![p \dot{\to} q]\!] & = & [\![p]\!] \to [\![q]\!] \\
[\![p \mathbin{\dot{\wedge}} q]\!] & = & [\![p]\!] \wedge [\![q]\!] \\
[\![p \mathbin{\dot{\vee}} q]\!] & = & [\![p]\!] \vee [\![q]\!] \\
[\![(\dot{\forall}\ p')]\!] & = & \Pi x : A.\, [\![(p'\ x)]\!] \\
[\![(\dot{\exists}\ p')]\!] & = & \exists x : A.\, [\![(p'\ x)]\!]
\end{array}$$

We use $\wedge, \vee, \exists$ for Coq's predefined logical connectives. Note that '$\to$' (and 'Π') is used for both (dependent) function space as well as for logical implication (quantification); this overloading witnesses the Curry-Howard isomorphism.

We don't have to worry about name conflicts when introducing a new $x : A$ for interpretation of formulas whose head constructor is a quantifier. Coq's binding mechanisms are internally based on De Bruijn indices (with a user-friendly tool showing named variables on top of it).

In the above definitions of o, its constructors and of $[\![\_]\!]$, the dependency on the signature (constituted by A, l_{rel} and $\mathcal{R}$) has been suppressed.

3. Clausification and correctness

We describe the transformation to *clausal form* (see Section 4), which is realized on both levels. On the object-level, we define an algorithm $mcf : o \to o$ that converts object-level propositions into their clausal form. On the meta-level, clausification is realized by a term mcf_{sound}, which (given the axiom of excluded middle and the axiom of choice) transforms a proof of $[\![(mcf\ p)]\!]$ into a proof of $[\![p]\!]$.

The algorithm *mcf* consists of the subsequent application of the following functions: *nnf*, *pnf*, *cnf*, *sklm*, *duqc*, *impf* standing for transformations to negation, prenex and conjunctive normal form, Skolemization, distribution of universal quantifiers over conjuncts and transformation to implicational form, respectively. As an illustration, we describe the functions *nnf* and *sklm*.

3.1. Negation normal form

Concerning negation normal form, a recursive call like

$$(nnf\ \dot{\neg}(p \mathrel{\dot{\wedge}} q)) = (nnf\ \dot{\neg}p) \mathrel{\dot{\vee}} (nnf\ \dot{\neg}q)$$

is not primitive recursive, since $\dot{\neg}p$ and $\dot{\neg}q$ are not subformulae of $\dot{\neg}(p \mathrel{\dot{\wedge}} q)$. Such a call requires general recursion. Coq's computational mechanism is higher-order primitive recursion, which is weaker than general recursion but ensures universal termination.

DEFINITION 3.1. *The function nnf : o → pol → o makes use of the so-called polarity (⊕ or ⊖) of an input formula.*

$$\begin{array}{rcl}
(nnf\ R_i(t_1,\ldots,t_k)\ \oplus) &=& R_i(t_1,\ldots,t_k) \\
(nnf\ R_i(t_1,\ldots,t_k)\ \ominus) &=& \dot{\neg}R_i(t_1,\ldots,t_k) \\
(nnf\ \dot{\neg}p\ \oplus) &=& (nnf\ p\ \ominus) \\
(nnf\ \dot{\neg}p\ \ominus) &=& (nnf\ p\ \oplus) \\
(nnf\ p_1 \mathrel{\dot{\to}} p_2\ \oplus) &=& (nnf\ p_1\ \ominus) \mathrel{\dot{\vee}} (nnf\ p_2\ \oplus) \\
(nnf\ p_1 \mathrel{\dot{\to}} p_2\ \ominus) &=& (nnf\ p_1\ \oplus) \mathrel{\dot{\wedge}} (nnf\ p_2\ \ominus) \\
(nnf\ p_1 \mathrel{\dot{\wedge}} p_2\ \oplus) &=& (nnf\ p_1\ \oplus) \mathrel{\dot{\wedge}} (nnf\ p_2\ \oplus) \\
(nnf\ p_1 \mathrel{\dot{\wedge}} p_2\ \ominus) &=& (nnf\ p_1\ \ominus) \mathrel{\dot{\vee}} (nnf\ p_2\ \ominus) \\
(nnf\ p_1 \mathrel{\dot{\vee}} p_2\ \oplus) &=& (nnf\ p_1\ \oplus) \mathrel{\dot{\vee}} (nnf\ p_2\ \oplus)
\end{array}$$

$$\begin{aligned}
(\mathit{nnf}\ p_1 \dot{\vee}\, p_2\ \ominus) &= (\mathit{nnf}\ p_1\ \ominus)\ \dot{\wedge}\ (\mathit{nnf}\ p_2\ \ominus)\\
(\mathit{nnf}\ (\dot{\forall}\ p')\ \oplus) &= (\dot{\forall}\ (\lambda x{:}A.\,(\mathit{nnf}\ (p'\ x)\ \oplus)))\\
(\mathit{nnf}\ (\dot{\forall}\ p')\ \ominus) &= (\dot{\exists}\ (\lambda x{:}A.\,(\mathit{nnf}\ (p'\ x)\ \ominus)))\\
(\mathit{nnf}\ (\dot{\exists}\ p')\ \oplus) &= (\dot{\exists}\ (\lambda x{:}A.\,(\mathit{nnf}\ (p'\ x)\ \oplus)))\\
(\mathit{nnf}\ (\dot{\exists}\ p')\ \ominus) &= (\dot{\forall}\ (\lambda x{:}A.\,(\mathit{nnf}\ (p'\ x)\ \ominus)))
\end{aligned}$$

In order to prove soundness of *nnf* we need the principle of excluded middle *EM*, which we define in such a way that it affects the first-order fragment only (like o, *EM* depends on the signature).

DEFINITION 3.2.

$$\mathit{EM} := \Pi p{:}o.\,[\![p\ \dot{\vee}\ \dot{\neg} p]\!]$$

LEMMA 3.1. *Assume EM is inhabited, then we have for all* $p : o$*:*

$$\begin{aligned}
[\![p]\!] &\leftrightarrow [\![(\mathit{nnf}\ p\ \oplus)]\!]\\
\neg[\![p]\!] &\leftrightarrow [\![(\mathit{nnf}\ p\ \ominus)]\!]
\end{aligned}$$

3.2. Skolemization

Skolemization of a formula means the removal of all existential quantifiers and the replacement of the variables that were bound by the removed existential quantifiers by new terms, that is, Skolem functions applied to the universally quantified variables whose quantifier had the existential quantifier in its scope. Instead of quantifying each of the Skolem functions, we introduce an index type $\mathcal{S}$, which may be viewed as a type for families of Skolem functions.

$$\mathcal{S} := \mathbb{N} \to \mathbb{N} \to \Pi n{:}\mathbb{N}.\, A^n \to A$$

A Skolem function, then, is a term $(f\ i\ j\ n) : A^n \to A$ with $f : \mathcal{S}$ and $i, j, n : \mathbb{N}$. Here, i and j are indices that distinguish the family members. If the output of *nnf* yields a conjunction, the remaining clausification steps are performed separately on the conjuncts. (This yields a significant speed-up in performance.) Index i denotes the position of the conjunct, j denotes the number of the replaced existentially quantified variable in that conjunct.

DEFINITION 3.3. *The function sklm is defined as follows.*

$$\begin{aligned}
(\mathit{sklm}\ f\ i\ j\ n\ t\ (\dot{\forall}\ p')) &= (\dot{\forall}\ (\lambda x{:}A.\,(\mathit{sklm}\ f\ i\ j\ n+1\ (t,x)\ (p'\ x))))\\
(\mathit{sklm}\ f\ i\ j\ n\ t\ (\dot{\exists}\ p')) &= (\mathit{sklm}\ f\ i\ j+1\ n\ t\ (p'\ (f\ i\ j\ n\ t)))\\
(\mathit{sklm}\ f\ i\ j\ n\ t\ p) &= p \text{ *if* } p \text{ *is neither* } (\dot{\forall}\ p') \text{ *nor* } (\dot{\exists}\ p')
\end{aligned}$$

Here and below (t, x) denotes the tuple typed A^{n+1} obtained by appending x to t. If the input formula is of the form $(\dot{\forall}\ p')$, then the quantified variable is added at the end of the so far constructed tuple t of universally quantified variables. In case the input formula matches $(\dot{\exists}\ p')$ with $p' : A \to o$ the term $(f\ i\ j\ n\ t)$ is substituted for the existentially quantified variable (the 'hole' in p') and index j is incremented. This substitution comes for free and is performed on the meta-level by β-reducing $(p'\ (f\ i\ j\ n\ t))$. The third case exhausts the five remaining cases. As we enforce input formulae of *sklm* to be in prenex normal form (via the definition of *mcf*), nothing remains to be done.

LEMMA 3.2. *For all $i : \mathbb{N}$ and $p : o$ we have:*

$$A \to AC_{\mathcal{S}} \to [\![p]\!] \to \exists f{:}\mathcal{S}.\, [\![(\mathit{sklm}\ f\ i\ 0\ 0\ \bullet\ p)]\!]$$

In the above lemma, $A \to \cdots$ expresses the condition that A is non-empty, and below $a : A$ denotes a canonical inhabitant. $AC_{\mathcal{S}}$ is a specific formulation of the Axiom of Choice, which allows us to form Skolem functions. Like *EM*, $AC_{\mathcal{S}}$ implicitly depends on the signature, that is, on A, l_{rel} and $\mathcal{R}$.

$$\begin{aligned} AC_{\mathcal{S}} :=\ & \Pi\alpha{:}A \to \mathcal{S} \to o. \\ & (\Pi x{:}A.\, \exists f{:}\mathcal{S}.\, [\![(\alpha\ x\ f)]\!]) \\ & \quad \to \exists F{:}A \to \mathcal{S}.\, \Pi x{:}A.\, [\![(\alpha\ x\ (F\ x))]\!] \end{aligned}$$

which indeed follows from the more general:

$$\begin{aligned} AC :=\ & \Pi A, B{:}*^{s}. \\ & \Pi P{:}A \to B \to *^{p}. \\ & \quad (\Pi x{:}A.\, \exists y{:}B.\, (P\ x\ y)) \\ & \qquad \to \exists f{:}A \to B.\, \Pi x{:}A.\, (P\ x\ (f\ x)) \end{aligned}$$

Let us inspect a crucial step in the proof of this lemma, which proceeds by induction on $p : o$. Consider the case that p is of the form $(\dot{\forall}\ p')$. Our induction hypothesis is:

$$\Pi x{:}A.\, [\![(p'\ x)]\!] \to \exists f{:}\mathcal{S}.\, [\![(\mathit{sklm}\ f\ i\ 0\ 0\ \bullet\ (p'\ x))]\!]$$

Assume $\Pi x{:}A.\, [\![(p'\ x)]\!]$. Then we have:

$$\Pi x{:}A.\, \exists f{:}\mathcal{S}.\, [\![(\mathit{sklm}\ f\ i\ 0\ 0\ \bullet\ (p'\ x))]\!]$$

By application of $AC_{\mathcal{S}}$ we get:

$$\Pi x{:}A.\, [\![(\mathit{sklm}\ (F\ x)\ i\ 0\ 0\ \bullet (p'\ x))]\!]$$

for some function $F : A \to \mathcal{S}$. Our goal is:

$$\exists g{:}\mathcal{S}.\, \Pi x{:}A.\, [\![(\mathit{sklm}\ g\ i\ 0\ 1\ (x, \bullet)\ (p'\ x))]\!]$$

(Note that $(x, \bullet) : A^1$ denotes the parameter list with only x, which is the result of appending x to the empty parameter list, $(\bullet, x)$. In examples we simply write x.) The witnessing g is given by:

$$\begin{aligned}(g\ i\ j\ 0\ \bullet) &= a \\ (g\ i\ j\ n+1\ (x,t)) &= (F\ x\ i\ j\ n\ t)\end{aligned}$$

Now

$$[\![(\mathit{sklm}\ g\ i\ 0\ 1\ (x,\bullet)\ (p'\ x))]\!]$$

follows from

$$[\![(\mathit{sklm}\ (F\ x)\ i\ 0\ 0\ \ \bullet (p'\ x))]\!]$$

via Lemma 3.3, as for any $n : \mathbb{N}$, g behaves like $(F\ x)$ on any tail $t :\ A^n$.

LEMMA 3.3. *For all $i, j_f, j_g, n_f, n_g : \mathbb{N}$, $t_f : A^{n_f}$, $t_g : A^{n_g}$, $p : o$, we have: if for all $m, n : \mathbb{N}, t : A^n$*

$$(f\ \ i\ \ j_f + m\ \ n_f + n\ \ (t_f, t)) = (g\ \ i\ \ j_g + m\ \ n_g + n\ \ (t_g, t))$$

then

$$[\![(\mathit{sklm}\ \ f\ \ i\ \ j_f\ \ n_f\ \ t_f\ \ p)]\!] \rightarrow [\![(\mathit{sklm}\ \ g\ \ i\ \ j_g\ \ n_g\ \ t_g\ \ p)]\!]$$

Here tuples $(t_f, t) : A^{n_f+n}$ and $(t_g, t) : A^{n_g+n}$ are the result of appending t to t_f and t_g, respectively.

3.3. Composing the modules

Reconsider Figure 1 and substitute *mcf* for T. Given a suitable signature, from any first-order formula $\varphi : *^p$, we can compute the clausal form $[\![(\mathit{mcf}\ \ \dot{\varphi})]\!]$.

LEMMA 3.4. *There exists a proof term $\mathit{mcf}_{\mathit{sound}}$ which validates clausification on the meta-level. More precisely:*

$$\mathit{mcf}_{\mathit{sound}} : \mathit{EM} \rightarrow \mathit{AC}_{\mathcal{S}} \rightarrow A \rightarrow \Pi p\!:\!o.\ [\![(\mathit{mcf}\ \ p)]\!] \rightarrow [\![p]\!]$$

The term $[\![(\mathit{mcf}\ \ \dot{\varphi})]\!]$ computes a format $C_1 \rightarrow \cdots \rightarrow C_n \rightarrow \bot$. Here $C_1, \ldots, C_n : *^p$ are universally closed clauses that will be exported to Bliksem, which constructs the proof term d representing a resolution refutation of these clauses (see Sections 4 and 5). Finally, d is type-checked in Coq. Section 6 demonstrates the outlined constructions.

The complete Coq-script generating the correctness proof of the clausification algorithm comprises ± 65 pages. It is available at [9].

4. Minimal resolution logic

There exist many representations of clauses and corresponding formulations of resolution rules. The traditional form of a clause is a disjunction of literals, that is, of atoms and negated atoms. Another form which is often used is that of a sequent, that is, the implication of a disjunction of atoms by a conjunction of atoms.

Here we propose yet another representation of clauses, as far as we know not used before. There are three main considerations.

- A structural requirement is that the representation of clauses is closed under the operations involved, such as instantiation and resolution.
- The Curry-Howard correspondence is most direct between minimal logic ($\to$,$\forall$) and a typed lambda calculus with product types (with $\to$ as a special, non-dependent, case of Π). Conjunction and disjunction in the logic require either extra type-forming primitives and extra terms to inhabit these, or impredicative encodings.
- The λ-representation of resolution steps should preferably be linear in the size of the premisses.

These considerations have led us to represent a clause like:

$$L_1 \vee \cdots \vee L_p$$

by the following classically equivalent implication in minimal logic:

$$\overline{L}_1 \to \cdots \to \overline{L}_p \to \bot$$

Here $\overline{L}_i$ is the complement of L_i in the classical sense (i.e. double negations are removed). If C is the disjunctive form of a clause, then we denote its implicational form by $[C]$. As usual, these expressions are implicitly or explicitly universally closed.

A resolution refutation of given clauses $C_1, \ldots, C_n$ proves their inconsistency, and can be taken as a proof of the following implication in minimal logic:

$$C_1 \to \cdots \to C_n \to \bot$$

Here and below, 'minimal' refers to minimal logic, as we use no particular properties of $\bot$. In particular, 'minimal clause' refers to the representation in minimal logic, and not to any other kind of minimality. We are now ready for the definition of the syntax of minimal resolution logic.

DEFINITION 4.1. *Let $\forall\vec{x}.\,\phi$ denote the universal closure of ϕ. Let Atom be the set of atomic propositions. We define the sets Literal, Clause and MCF of, respectively, literals, clauses and minimal clausal forms by the following abstract syntax.*

$$\begin{array}{rcl} \mathit{Literal} & ::= & \mathit{Atom} \mid \mathit{Atom} \to \bot \\ \mathit{Clause} & ::= & \bot \mid \mathit{Literal} \to \mathit{Clause} \\ \mathit{MCF} & ::= & \bot \mid (\forall\vec{x}.\,\mathit{Clause}) \to \mathit{MCF} \end{array}$$

Next we elaborate the familiar inference rules for factoring, permuting and weakening clauses, as well as the binary resolution rule.

Factoring, Permutation, Weakening

Let C and D be clauses, such that C subsumes D propositionally, that is, any literal in C also occurs in D. Let $A_1, \ldots, A_p, B_1, \ldots, B_q$ be literals ($p, q \geq 0$) and write

$$[C] = A_1 \to \cdots \to A_p \to \bot$$

and

$$[D] = B_1 \to \cdots \to B_q \to \bot$$

assuming that for every $1 \leq i \leq p$ there is $1 \leq j \leq q$ such that $A_i = B_j$.

A proof of $[C] \to [D]$ is the following λ-term:

$$\lambda c\!:\![C].\,\lambda b_1\!:\!B_1 \ldots \lambda b_q\!:\!B_q.\,(c\ \pi_1\ \ldots\ \pi_p)$$

with $\pi_i = b_j$, where j is such that $B_j = A_i$.

Binary Resolution

In the traditional form of the binary resolution rule for disjunctive clauses we have premisses C_1 and C_2, containing one or more occurrences of a literal L and of $\overline{L}$, respectively. The conclusion of the rule, the resolvent, is then a clause D consisting of all literals of C_1 different from L joined with all literals of C_2 different from $\overline{L}$. This rule is completely symmetric with respect to C_1 and C_2.

For clauses in implicational form there is a slight asymmetry in the formulation of binary resolution. Let $A_1, \ldots, A_p, B_1 \ldots, B_q$ be literals ($p, q \geq 0$) and write

$$[C_1] = A_1 \to \cdots \to A_p \to \bot,$$

with one or more occurrences of the negated atom $A \to \bot$ among the A_i and

$$[C_2] = B_1 \to \cdots \to B_q \to \bot,$$

with one or more occurrences of the atom A among the B_j. Write the resolvent D as

$$[D] = D_1 \to \cdots \to D_r \to \bot$$

consisting of all literals of C_1 different from $A \to \bot$ joined with all literals of C_2 different from A. A proof of $[C_1] \to [C_2] \to [D]$ is the following λ-term:

$$\lambda c_1\!:\![C_1].\,\lambda c_2\!:\![C_2].\,\lambda d_1\!:\!D_1 \ldots \lambda d_r\!:\!D_r.\,(c_1\ \pi_1\ \ldots\ \pi_p)$$

For $1 \leq i \leq p$, π_i is defined as follows. If $A_i \neq (A \to \bot)$, then $\pi_i = d_k$, where k is such that $D_k = A_i$. If $A_i = A \to \bot$, then we put

$$\pi_i = \lambda a\!:\!A.\,(c_2\ \pi'_1\ \ldots\ \pi'_q),$$

with π'_j $(1 \leq j \leq q)$ defined as follows. If $B_j \neq A$, then $\pi'_j = d_k$, where k is such that $D_k = B_j$. If $B_j = A$, then $\pi'_j = a$. It is easily verified that $\pi_i : (A \to \bot)$ in this case.

If $(A \to \bot)$ occurs more than once among the A_i, then $(c_1\ \pi_1\ \ldots\ \pi_p)$ need not be linear. This can be avoided by factoring timely. Even without factoring, a linear proof term is possible: by taking the following β-expansion of $(c_1\ \pi_1\ \ldots\ \pi_p)$ (with a' replacing copies of proofs of $(A \to \bot)$):

$$(\lambda a'\!:\!A \to \bot.\,(c_1\ \pi_1\ \ldots\ a'\ \ldots\ a'\ \ldots\ \pi_p))(\lambda a\!:\!A.\,(c_2\ \pi'_1\ \ldots\ \pi'_q))$$

This remark applies to the rules in the next subsections as well.

Paramodulation

Paramodulation combines equational reasoning with resolution. For equational reasoning we use the inductive equality of Coq. In order to simplify matters, we assume a fixed domain of discourse A, and denote equality of $s_1, s_2 \in A$ by $s_1 \approx s_2$.

Coq supplies us with the following terms:

$$\begin{aligned} \mathit{eqrefl} &: \forall s\!:\!A.\ (s \approx s) \\ \mathit{eqsubst} &: \forall s\!:\!A.\forall P\!:\!A \to *^p.\ (P\ s) \to \forall t\!:\!A.\ (s \approx t) \to (P\ t) \\ \mathit{eqsym} &: \forall s_1, s_2\!:\!A.\ (s_1 \approx s_2) \to (s_2 \approx s_1) \end{aligned}$$

As an example we define *eqsym* from *eqsubst*, *eqrefl*:

$$\lambda s_1, s_2\!:\!A.\,\lambda h\!:\!(s_1 \approx s_2).\,(\mathit{eqsubst}\ s_1\ (\lambda s\!:\!A.\,(s \approx s_1))\ (\mathit{eqrefl}\ s_1)\ s_2\ h)$$

Paramodulation for disjunctive clauses is the rule with premiss C_1 containing the equality literal $t_1 \approx t_2$ and premiss C_2 containing literal

$L[t_1]$. The conclusion is then a clause D containing all literals of C_1 different from $t_1 \approx t_2$, joined with C_2 with $L[t_2]$ instead of $L[t_1]$.

Let $A_1, \ldots, A_p, B_1 \ldots, B_q$ be literals ($p, q \geq 0$) and write

$$[C_1] = A_1 \to \cdots \to A_p \to \bot,$$

with one or more occurrences of the equality atom $t_1 \approx t_2 \to \bot$ among the A_i, and

$$[C_2] = B_1 \to \cdots \to B_q \to \bot,$$

with one or more occurrences of the literal $L[t_1]$ among the B_j. Write the conclusion D as

$$[D] = D_1 \to \cdots \to D_r \to \bot$$

and let l be such that $D_l = L[t_2]$. A proof of $[C_1] \to [C_2] \to [D]$ can be obtained as follows:

$$\lambda c_1 : [C_1].\, \lambda c_2 : [C_2].\, \lambda d_1 : D_1 \ldots \lambda d_r : D_r.\, (c_1\ \pi_1\ \ldots\ \pi_p)$$

If $A_i \neq (t_1 \approx t_2 \to \bot)$, then $\pi_i = d_k$, where k is such that $D_k = A_i$. If $A_i = (t_1 \approx t_2 \to \bot)$, then we want again that $\pi_i : A_i$ and therefore put

$$\pi_i = \lambda e : (t_1 \approx t_2).\, (c_2\ \pi'_1\ \ldots\ \pi'_q).$$

If $B_j \neq L[t_1]$, then $\pi'_j = d_k$, where k is such that $D_k = B_j$. If $B_j = L[t_1]$, then we also want that $\pi'_j : B_j$ and put (with $d_l : D_l = L[t_2]$)

$$\pi'_j = (\mathit{eqsubst}\ t_2\ (\lambda s : A.\, L[s])\ d_l\ t_1\ (\mathit{eqsym}\ t_1\ t_2\ e))$$

The term π'_j has type $L[t_1]$ in the context $e : (t_1 \approx t_2)$. The term π'_j contains an occurrence of *eqsym* because of the fact that the equality $t_1 \approx t_2$ comes in the wrong direction for proving $L[t_1]$ from $L[t_2]$. With this definition of π'_j, the term π_i has indeed type $A_i = (t_1 \approx t_2 \to \bot)$.

As an alternative, it is possible to expand the proof of *eqsym* in the proof of the paramodulation step.

Equality Factoring

Equality factoring for disjunctive clauses is the rule with premiss C containing equality literals $t_1 \approx t_2$ and $t_1 \approx t_3$, and conclusion D which is identical to C but for the replacement of $t_1 \approx t_3$ by $t_2 \not\approx t_3$. The soundness of this rule relies on $t_2 \approx t_3 \vee t_2 \not\approx t_3$.

Let $A_1, \ldots, A_p, B_1 \ldots, B_q$ be literals ($p, q \geq 0$) and write

$$[C] = A_1 \to \cdots \to A_p \to \bot,$$

with equality literals $t_1 \approx t_2 \to \bot$ and $t_1 \approx t_3 \to \bot$ among the A_i. Write the conclusion D as

$$[D] = B_1 \to \cdots \to B_q \to \bot$$

with $B_{j'} = (t_1 \approx t_2 \to \bot)$ and $B_{j''} = (t_2 \approx t_3)$. We get a proof of $[C] \to [D]$ from

$$\lambda c\colon[C].\, \lambda b_1\colon B_1 \ldots \lambda b_q\colon B_q.\, (c\ \pi_1\ \ldots\ \pi_p).$$

If $A_i \neq (t_1 \approx t_3 \to \bot)$, then $\pi_i = b_j$, where j is such that $B_j = A_i$. For $A_i = (t_1 \approx t_3 \to \bot)$, we put

$$\pi_i = (\mathit{eqsubst}\ t_2\ (\lambda s\colon A.\, (t_1 \approx s \to \bot))\ b_{j'}\ t_3\ b_{j''}).$$

The type of π_i is indeed $t_1 \approx t_3 \to \bot$.

Note that the equality factoring rule is constructive in the implicational translation, whereas its disjunctive counterpart relies on the decidability of $\approx$. This phenomenon is well-known from the double negation translation.

Positive and Negative Equality Swapping

The positive equality swapping rule for disjunctive clauses simply swaps an atom $t_1 \approx t_2$ into $t_2 \approx t_1$, whereas the negative rule swaps the negated atom. Both versions are obviously sound, given the symmetry of $\approx$.

We give the translation for the positive case first and will then sketch the simpler negative case. Let C be the premiss and D the conclusion and write

$$[C] = A_1 \to \cdots \to A_p \to \bot,$$

with some of the A_i equal to $t_1 \approx t_2 \to \bot$, and

$$[D] = B_1 \to \cdots \to B_q \to \bot.$$

Let j' be such that $B_{j'} = (t_2 \approx t_1 \to \bot)$. The following term is a proof of $[C] \to [D]$.

$$\lambda c\colon[C].\, \lambda b_1\colon B_1 \ldots \lambda b_q\colon B_q.\, (c\ \pi_1\ \ldots\ \pi_p)$$

If $A_i \neq (t_1 \approx t_2 \to \bot)$, then $\pi_i = b_j$, where j is such that $B_j = A_i$. Otherwise

$$\pi_i = \lambda e\colon(t_1 \approx t_2).\, (b_{j'}\ (\mathit{eqsym}\ t_1\ t_2\ e))$$

such that also $\pi_i : (t_1 \approx t_2 \to \bot) = A_i$.

In the negative case the literals $t_1 \approx t_2$ in question are not negated, and we change the above definition of π_i into

$$\pi_i = (\mathit{eqsym}\ t_2\ t_1\ b_{j'}).$$

In this case we have $b_{j'} : (t_2 \approx t_1)$ so that $\pi_i : (t_1 \approx t_2) = A_i$ also in the negative case.

Equality Reflexivity Rule

The equality reflexivity rule simply cancels a negative equality literal of the form $t \not\approx t$ in a disjunctive clause. We write once more the premiss

$$[C] = A_1 \to \cdots \to A_p \to \bot,$$

with some of the A_i equal to $t \approx t$, and the conclusion

$$[D] = B_1 \to \cdots \to B_q \to \bot.$$

The following term is a proof of $[C] \to [D]$:

$$\lambda c\!:\![C].\,\lambda b_1\!:\!B_1 \ldots \lambda b_q\!:\!B_q.\,(c\ \pi_1\ \ldots\ \pi_p).$$

If $A_i \neq (t \approx t)$, then $\pi_i = b_j$, where j is such that $B_j = A_i$. Otherwise $\pi_i = (\mathit{eqrefl}\ t)$.

5. Lifting to Predicate Logic

Until now we have only considered inference rules without quantifications. In this section we explain how to lift the resolution rule to predicate logic. Lifting the other rules is very similar.

Recall that we must assume that the domain is not empty. Proof terms below may contain a variable $a : A$ as free variable. By abstraction $\lambda a : A$ we will close all proof terms. This extra step is necessary since $\forall a\!:\!A.\,\bot$ does not imply $\bot$ when the domain A is empty. This is to be compared to $\Box\bot$ being true in a blind world in modal logic.

Consider the following clauses

$$C_1 = \forall x_1, \ldots, x_p\!:\!A.\,[A_1 \vee R_1]$$

and

$$C_2 = \forall y_1, \ldots, y_q\!:\!A.\,[\neg A_2 \vee R_2]$$

and their resolvent

$$R = \forall z_1, \ldots, z_r\!:\!A.\,[R_1\theta_1 \vee R_2\theta_2]$$

Here θ_1 and θ_2 are substitutions such that $A_1\theta_1 = A_2\theta_2$ and $z_1, \ldots, z_r$ are all variables that actually occur in the resolvent, that is, in $R_1\theta_1 \vee R_2\theta_2$ after application of θ_1, θ_2. It may be the case that $x_i\theta_1$ and/or $y_j\theta_2$ contain other variables than $z_1, \ldots, z_r$; these are understood to be replaced by the variable $a : A$ (see above). It that case θ_1, θ_2 may not represent a *most general* unifier. For soundness this is no problem at all, but even completeness is not at stake since the resolvent is not affected. The reason for this subtlety is that the proof terms involved must not contain undeclared variables.

Using the methods of the previous sections we can produce a proof π that has the type

$$[A_1 \vee R_1]\theta_1 \to [\neg A_2 \vee R_2]\theta_2 \to [R_1\theta_1 \vee R_2\theta_2].$$

A proof of $C_1 \to C_2 \to R$ is obtained as follows:

$$\begin{array}{l} \lambda c_1 : C_1.\, \lambda c_2 : C_2.\, \lambda z_1 \ldots z_r : A. \\ (\pi\ (c_1\ (x_1\theta_1) \ldots (x_p\theta_1))\ (c_2\ (y_1\theta_2) \ldots (y_q\theta_2))) \end{array}$$

We finish this section by showing how to assemble a λ-term for an entire resolution refutation from the proof terms justifying the individual steps. Consider a Hilbert-style resolution derivation

$$C_1, \ldots, C_m, C_{m+1}, \ldots, C_n$$

with premisses $c_1 : C_1, \ldots, c_m : C_m$. Starting from n and going downward, we will define by recursion for every $m \leq k \leq n$ a term π_k such that

$$\pi_k[c_{m+1}, \ldots, c_k] : C_n$$

in the context extended with $c_{m+1} : C_{m+1}, \ldots, c_k : C_k$. For $k = n$ we can simply take $\pi_n = c_n$. Now assume π_{k+1} has been constructed for some $k \geq m$. The proof π_k is more difficult than π_{k+1} since π_k cannot use the assumption $c_{k+1} : C_{k+1}$. However, C_{k+1} is a resolvent, say of C_i and C_j for some $i, j \leq k$. Let d be the proof of $C_i \to C_j \to C_{k+1}$. Now define

$$\pi_k[c_{m+1}, \ldots, c_k] = (\lambda x : C_{k+1}.\pi_{k+1}[c_{m+1}, \ldots, c_k, x])(d\ c_i\ c_j) : C_n$$

The downward recursion yields a proof $\pi_m : C_n$ which is linear in the size of the original Hilbert-style resolution derivation. Observe that a forward recursion from m to n would yield the normal form of π_m, which could be exponential.

6. Examples

6.1. A small example

Let P be a property of natural numbers such that P holds for n if and only if P does not hold for any number greater than n. Does this sound paradoxical? It is contradictory. We have $P(n)$ if and only if $\neg P(n+1), \neg P(n+2), \neg P(n+3), \ldots$, which implies $\neg P(n+2), \neg P(n+3), \ldots$, so $P(n+1)$. It follows that $\neg P(n)$ for all n. However, $\neg P(0)$ implies $P(n)$ for some n, contradiction.

A closer analysis of this argument shows that the essence is not arithmetical, but relies on the fact that $<$ is transitive and serial. The argument is also valid in a finite cyclic structure, say $0 < 1 < 2 < 2$. This qualifies for a small refutation problem, which we formalize in Coq.

Let us adopt $\mathbb{N}$ as the domain of discourse. We declare a unary relation P and a binary relation $<$.

$$\begin{array}{rcl} P & : & \mathbb{N} \to *^p \\ < & : & \mathbb{N} \times \mathbb{N} \to *^p \end{array}$$

Let $l_{rel} = [1,2]$ be the corresponding list of arities. The relations are packaged by $\mathcal{R}$ of type $\Pi i : [0,1].\, \mathbb{N}^{l_{rel}(i)} \to *^p$. We write $\mathcal{R}_i$ for $(\mathcal{R}\ i)$; note $I_{l_{rel}} = [0,1]$.

$$\mathcal{R}_0 = P \qquad \mathcal{R}_1 = <$$

We write $\dot{P}$ for R_0 and infix $\dot{<}$ for R_1 respectively.

Let us construct the formal propositions *trans* and *serial*, stating that $\dot{<}$ is serial and transitive. $\dot{\forall} x.\, \phi$ is syntactic sugar for $(\dot{\forall}\ (\lambda x{:}\mathbb{N}.\, \phi))$, likewise for $\dot{\exists}$.

$$\begin{array}{rcl} \mathit{trans} & = & \dot{\forall} x, y, z.\, (x \mathrel{\dot{<}} y \mathbin{\dot{\wedge}} y \mathrel{\dot{<}} z) \mathrel{\dot{\to}} x \mathrel{\dot{<}} z \\ \mathit{serial} & = & \dot{\forall} x.\, \dot{\exists} y.\, x \mathrel{\dot{<}} y \end{array}$$

We define *foo*.

$$\mathit{foo} = \dot{\forall} x.\, (\dot{P}\ x) \mathrel{\dot{\leftrightarrow}} (\dot{\forall} y.\, x \mathrel{\dot{<}} y \mathrel{\dot{\to}} \dot{\neg}(\dot{P}\ y))$$

Furthermore, we define *taut* on the object-level, representing the example informally stated at the beginning of this section. (If the latter is denoted by φ, then $\mathit{taut} = \dot{\varphi}$.)

$$\mathit{taut} = (\mathit{trans} \mathbin{\dot{\wedge}} \mathit{serial}) \mathrel{\dot{\to}} \dot{\neg}\mathit{foo}$$

Interpreting *taut*, that is $\beta\delta\iota$-normalizing $[\![\mathit{taut}]\!]$, results in '*taut* without dots'.

We declare $em : EM$, $ac : AC_{\mathcal{S}}$ and use 0 to witness the non-emptiness of $\mathbb{N}$. We reduce the goal $[\![taut]\!]$ using the result of Section 3, to the goal $[\![(mcf\ \ taut)]\!]$. If we prove this latter goal, say by a term d, then

$$(mcf_{sound}\ em\ ac\ 0\ taut\ d) : [\![taut]\!]$$

We compute the minimal clausal form (Definition 4.1) of $taut$ by $\beta\delta\iota$-normalizing the goal $[\![(mcf\ \ taut)]\!]$.

$$\begin{array}{l}
[\![(mcf\ \ taut)]\!] =_{\beta\delta\iota} \\
\quad (\Pi x, y, z{:}\mathbb{N}.\, x < y \to y < z \to (x < z \to \bot) \to \bot) \\
\qquad \to (\Pi x{:}\mathbb{N}.\, (x < (f\ 1\ 0\ 1\ x) \to \bot) \to \bot) \\
\qquad \to (\Pi x{:}\mathbb{N}.\, (x < (f\ 2\ 0\ 1\ x) \to \bot) \to ((P\ x) \to \bot) \to \bot) \\
\qquad \to (\Pi x{:}\mathbb{N}.\, ((P\ (f\ 2\ 0\ 1\ x)) \to \bot) \to ((P\ x) \to \bot) \to \bot) \\
\qquad \to (\Pi x, y{:}\mathbb{N}.\, (P\ x) \to x < y \to (P\ y) \to \bot) \\
\qquad \to \bot
\end{array}$$

This is the minimal clausal form of the original goal. We refrained from exhibiting its proof d. All files can be found in [9].

6.2. A MEDIUM SCALE EXAMPLE: NEWMAN'S LEMMA

A medium scale example is provided by the automation of Huet's [11] proof of Newman's Lemma (NL), a well known result in rewriting theory stating that a rewriting relation is confluent whenever it is both locally confluent and terminating. For a precise analysis we have to introduce some notions from rewriting theory.

DEFINITION 6.1. *Let $\to$ be a binary relation on a set S and let $\twoheadrightarrow$ be the reflexive-transitive closure of $\to$.*

1. *We say that x is* confluent *if, for all $x_1, x_2 \in S$, $x \twoheadrightarrow x_1$ and $x \twoheadrightarrow x_2$ implies that $x_1 \twoheadrightarrow y$ and $x_2 \twoheadrightarrow y$ for some $y \in S$. In other words, any two diverging reductions starting from x can always be brought together. We say that $\to$ is* confluent *if every $x \in S$ is confluent.*

2. *We say that x is* locally confluent *if, for all x_1, x_2, $x \to x_1$ and $x \to x_2$ implies that $x_1 \twoheadrightarrow y$ and $x_2 \twoheadrightarrow y$ for some $y \in S$. Here the 'locality' lies in the fact that only diverging* one-step *reductions can be brought together. We say that $\to$ is* locally confluent *if every $x \in S$ is locally confluent.*

3. *We say that $\to$ is* terminating *if there is no infinite sequence $x_0 \to x_1 \to x_2 \to \cdots$ in S.*

NL provides an interesting test case for several reasons. First, it consists of a mix of first-order and higher-order aspects. The higher-order aspects are the transitive closure and the termination. This makes the identification of the first-order combinatorial core of the proof non-trivial. Second, the proof of Newman's Lemma is not completely trivial, as experienced by everybody seeing it for the first time. It will turn out to be a reasoning step that is just on the edge of what can be achieved by current theorem provers. As such the successful automation is very sensitive to the exact formalization of the problem, the settings of the theorem prover and the machine on which one runs the proof. We admit that this is in some sense a disadvantage for an example. However, the aim of this example is to explore the borders of what is possible, and not to show-off how great the method is. It is to be expected that, with faster machines and better strategies for proof search, the automatic solution of problems of the size of NL will soon become routine. Moreover, the inductive approach to termination and the speed-up obtained by removing superfluous symmetries have a generality that goes beyond NL.

The classical proof of NL is by contradiction. Assume there is an x which is not confluent, that is, there exist $x_1, x_2 \in S$ such that $x \twoheadrightarrow x_1$ and $x \twoheadrightarrow x_2$ and no $y \in S$ exists such that $x_1 \twoheadrightarrow y$ and $x_2 \twoheadrightarrow y$. Since $\rightarrow$ is terminating, we may assume without loss of generality that x is an $\rightarrow$-maximal[3] non-confluent element. If not, there would be a non-confluent x' with $x \rightarrow x'$, and if that x' is not $\rightarrow$-maximal, then there would be a non-confluent x'' with $x' \rightarrow x''$ and so on, leading to a sequence contradicting the termination of $\rightarrow$. This part is difficult to explain, it actually uses the Axiom of Dependent Choice (DC). From the fact that x_1 and x_2 have no common reduct, it follows that we do not have $x = x_1$ or $x = x_2$, so there must exist intermediate points i_1, i_2 such that $x \rightarrow i_1 \twoheadrightarrow x_1$ and $x \rightarrow i_2 \twoheadrightarrow x_2$. To x and these intermediate points we can apply local confluence to obtain a common reduct of the intermediate points. By the maximality of x we can then complete the diagram in Figure 2 below. This is a contradiction and hence NL has been proved.

The formalization of the classical argument requires higher-order logic (to express transitive closure) and three-sorted first-order logic: one sort for the set S, one for the natural numbers and one for infinite sequences of elements of S. An important improvement is obtained by taking the constructive reformulation of NL as point of departure. In this formulation the infinite sequences such as used in the definition

[3] If the transitive closure of $\rightarrow$ is viewed as a *greater than* ordering, then it would be natural to speak of $\rightarrow$-*minimal* instead.

$$\begin{tikzcd}
x \arrow[r] \arrow[d] \arrow[dr, phantom, "\text{local confluence}"] & i_1 \arrow[r, two heads] \arrow[d, two heads] & x_1 \arrow[d, two heads] \\
i_2 \arrow[r, two heads] \arrow[d, two heads] & \cdot \arrow[r, two heads] & \cdot \arrow[d, two heads] \\
x_2 \arrow[rr, two heads] & & \cdot
\end{tikzcd}$$

Figure 2. Diagram chase for confluence

of termination and in DC are avoided by using an inductively defined predicate called accessibility.

DEFINITION 6.2. *Let $\rightarrow$ be a binary relation on a set S. The unary predicate $Acc_{\rightarrow}$ is inductively defined as follows: if $Acc_{\rightarrow}(y)$ for all $y \in S$ such that $x \rightarrow y$, then $Acc_{\rightarrow}(x)$. By $Acc_{\rightarrow}(S)$ we express that $Acc_{\rightarrow}(x)$ for all $x \in S$.*

In other words, all $\rightarrow$-maximal elements are accessible, as well as all elements whose successors are all $\rightarrow$-maximal, and so on. An infinite sequence $x_0 \rightarrow x_1 \rightarrow x_2 \rightarrow \cdots$ consists of elements that are not accessible. The reason is that they can be left out without violating the defining rule for *Acc*. In fact one can prove by classical logic and DC that $\rightarrow$ is terminating if and only if all elements of S are accessible, that is, if $Acc_{\rightarrow}(S)$.

The advantages of using $Acc_{\rightarrow}(S)$ instead of the traditional formulation of termination are three-fold.

- DC is not needed anymore in the proof of NL.
- The sorts for the natural numbers and for infinite sequences become obsolete.
- We can reason by induction on $Acc_{\rightarrow}(x)$, the induction step being first-order.

These reasons above should motivate the following reformulation of NL: if $Acc_{\rightarrow}(S)$, then confluence of $\rightarrow$ follows from local confluence.

We could have added a fourth advantage to the three advantages above, namely that the proof of NL in the formulation with the accessibility predicate can be done constructively. This would require resolution to be used bottom-up, in a forward reasoning style. We have not been able to generate a proof in this way. Instead, we had to appeal to classical logic by using resolution as a refutation procedure. The constructive proof is not more complicated than the classical one, it is actually shorter, but the relevant point here is that the search space for finding the proof in a bottom-up way appears to be larger than that for finding a proof in a more top-down, goal-oriented, way. We consider the situation in which there is a constructive proof, but for ill-understood reasons of efficiency only a classical proof can be found, as unsatisfactory.

We will sketch the constructive argument. By induction one proves that every accessible x is confluent. By $Acc_{\rightarrow}(S)$ we then obtain confluence. The induction step we have to prove is that confluence is preserved under the inductive definition of $Acc_{\rightarrow}$. In other words, we have to prove that x is confluent if the induction hypothesis (IH) holds, that is, every y such that $x \rightarrow y$ is confluent. Assume IH and let $x_1, x_2 \in S$ such that $x \twoheadrightarrow x_1$ and $x \twoheadrightarrow x_2$. If $x = x_1$ or $x = x_2$ then x_2 or x_1 is a common reduct of x_1, x_2. Otherwise, actually appealing to the inductive definition of the reflexive–transitive closure, there exist intermediate points as in the classical proof above. Now a common reduct can be obtained in exactly the same way as in the classical proof, with IH replacing the $\rightarrow$-maximality of x. This proves the induction step.

The above proof of the induction step is completely first-order, provided that we replace the appeal to the inductive definition of $\twoheadrightarrow$ by some first-order sentences that trivially follow from the inductive definition of $\twoheadrightarrow$ and are sufficient for the proof.

$$\left.\begin{array}{l} = \text{ is reflexive and symmetric} \\ \twoheadrightarrow \text{ includes } = \text{ and } \rightarrow \text{ and is transitive} \\ \twoheadrightarrow \text{ is included in the union of } = \text{ and } \rightarrow\cdot\twoheadrightarrow \\ \rightarrow \text{ is locally confluent} \end{array}\right\} \Rightarrow \begin{array}{l} \text{confluence} \\ \text{is } Acc_{\rightarrow}\text{-inductive} \end{array}$$

Here the conclusion that confluence is $Acc_{\rightarrow}$-inductive means that for all $x \in S$ confluence of x follows from confluence of all y such that $x \rightarrow y$. Note that we do not need transitivity of $=$. Moreover, $\rightarrow\cdot\twoheadrightarrow$ is the composition of $\rightarrow$ and $\twoheadrightarrow$.

We have formalized in Coq the proof of NL based on the above first-order tautology, with the intention to delegate the proof of the latter to a resolution theorem prover in the style of Section 6. The automatic clausification in Coq was a matter of seconds and resulted in 14 clauses. Both Otter and Bliksem were slow to refute the 14 clauses (without any tuning at least half an hour). The best results have been obtained with ordered hyperresolution in combination with unit-resulting resolution. The proof found by Otter is quite close to a ‘human’ proof by contradiction and the diagram chase in Figure 3. Bliksem managed to refute the corresponding set of clauses and to generate a proof object in the form of a lambda term. Although this lambda term has a considerable size (100 KByte), it could be type checked by Coq without any problem and included in a complete proof of NL in Coq. All files can be found in [9].

An obvious difficulty for proof search is the symmetry of the formulation of NL. Inspection of the proof shows that it is possible to distinguish between 'horizontal' and 'vertical' steps in the formulation of both confluence and local confluence. This leads to an asymmetrical version of Newman's Lemma (aNL), which can be proved by the same proof with all the steps properly labelled as either 'horizontal' or 'vertical'. NL can easily be recovered from aNL by removing the distinction. The advantage of the asymmetrical over the symmetrical formulation is that the search space for the proof is considerably reduced. For example, in the symmetrical case any step $x \to y$ leads to useless common reducts of y and y, which are avoided in the asymmetrical case. The asymmetrical analogues of confluence and local confluence are known in the literature as commutativity and weak commutativity, respectively.

DEFINITION 6.3. *Let $\to_h$ and $\to_v$ be binary relations on a set S, with reflexive-transitive closures $\twoheadrightarrow_h$ and $\twoheadrightarrow_v$, respectively.*

1. *We say that x is* commutative *if, for all $x_1, x_2 \in S$, $x \twoheadrightarrow_h x_1$ and $x \twoheadrightarrow_v x_2$ implies that $x_1 \twoheadrightarrow_v y$ and $x_2 \twoheadrightarrow_h y$ for some $y \in S$. We say that $\to_h$ and $\to_v$* commute *if every $x \in S$ is commutative.*

2. *We say that x is* weakly commutative *if, for all $x_1, x_2 \in S$, $x \to_h x_1$ and $x \to_v x_2$ implies that $x_1 \twoheadrightarrow_v y$ and $x_2 \twoheadrightarrow_h y$ for some $y \in S$. We say that $\to_h$ and $\to_v$ are* commute weakly *if every $x \in S$ is weakly commutative.*

The precise statement of aNL is that $\to_h$ and $\to_v$ commute if they commute weakly, provided $Acc_{\to_{hv}}(S)$. Here $\to_{hv}$ is the union of $\to_h$ and $\to_v$. A glance at Figure 3 tells us that we need the induction hypothesis both for i_1 with $x \to_h i_1$ and for i_2 with $x \to_v i_2$.

The proof of aNL follows the pattern of the proof of NL, but is based on the following first-order tautology:

$$\left.\begin{array}{l}
= \text{ is reflexive and symmetric} \\
\twoheadrightarrow_h \text{ includes } = \text{ and } \to_h \text{ and is transitive} \\
\twoheadrightarrow_v \text{ includes } = \text{ and } \to_v \text{ and is transitive} \\
\twoheadrightarrow_h \text{ is included in the union of } = \text{ and } \to_h \cdot \twoheadrightarrow_h \\
\twoheadrightarrow_v \text{ is included in the union of } = \text{ and } \to_v \cdot \twoheadrightarrow_v \\
\to_h \text{ and } \to_v \text{ are weakly commutative}
\end{array}\right\} \Rightarrow \begin{array}{l}\text{commutativity is} \\ Acc_{\to_{hv}}\text{-inductive}\end{array}$$

$$\begin{tikzcd}
x \arrow[r, "h"] \arrow[d, "v"'] \arrow[dr, phantom, "\text{weak commutativity}"] & i_1 \arrow[r, two heads, "h"] \arrow[d, two heads, "v"] & x_1 \arrow[d, two heads, "v"] \\
i_2 \arrow[r, two heads, "h"] \arrow[d, two heads, "v"'] & \cdot \arrow[r, two heads, "h"] & \cdot \arrow[d, two heads, "v"] \\
x_2 \arrow[rr, two heads, "h"] & & \cdot
\end{tikzcd}$$

Figure 3. Diagram chase for commutativity

Here the conclusion means that for all $x \in S$ commutativity of x follows from commutativity of all y such that $x \to_h y$ or $x \to_v y$.

We formalized in Coq the proof of aNL based on the above first-order tautology. Proof search in the asymmetrical case is about two orders of magnitude faster than in the symmetrical case. Again all files can be found in [9].

Summarizing, the method can be put to work on medium scale examples. However, it is obvious that some human intelligence has been spent on stylizing the proof before it could be automated. The techniques for proof search should be improved before the method can be scaled up any further.

Acknowledgements

The authors are indebted to Freek Wiedijk for acting as a wizard in Ocaml and to an anonymous referee for some detailed technical corrections.

References

1. B. Barras, S. Boutin, C. Cornes, J. Courant, J.-C. Filliâtre, E. Giménez, H. Herbelin, G. Huet, C. Muñez, C. Murthy, C. Parent, C. Paulin-Mohring, A. Saïbi, B. Werner. *The Coq Proof Assistant Reference Manual, version 6.2.4.* INRIA, 1998. Available at: `ftp.inria.fr/INRIA/coq/V6.2.4/doc/Reference-Manual.ps`
2. M. Bezem, D. Hendriks and H. de Nivelle. Automated Proof Construction in Type Theory using Resolution. In D. McAllester (ed.) *Proceedings CADE 17*, Lecture Notes in Computer Science 1831, pages 148–163. Springer-Verlag, 2000.
3. `www.mpi-sb.mpg.de/~bliksem`

4. S. Boutin. Using Reflection to Build Efficient and Certified Decision Procedures. In M. Abadi and T. Ito (eds.) *Theoretical Aspects of Computer Software (TACS)*, Lecture Notes in Computer Science 1281, pages 515–529. Springer Verlag, 1997.
5. N.G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation. *Indagationes Mathematicae* **34**, pages 381–392, 1972.
6. B. Günzel. *Logik und das Auswahlaxiom.* Diplomarbeit, Fakultät für Mathematik und Informatik der Ludwig-Maximilians-Universität München, 2000.
7. D. Hendriks. *Clausification of First-Order Formulae, Representation & Correctness in Type Theory.* Master Thesis, Utrecht University, 1998.
8. D. Hendriks. *Proof reflection in Coq.* Number 28 in Artificial Intelligence Preprint Series, Dept. of Philosophy, Utrecht University, 2001.
9. `www.phil.uu.nl/~hendriks/coq/blinc`
10. X. Huang. Translating machine-generated resolution proofs into ND-proofs at the assertion level. In *Proceedings of PRICAI-96*, pages 399–410, 1996.
11. G. Huet. Confluent reductions: Abstract properties and applications to term rewriting systems. *Journal of the ACM* **27**(4), pages 797–821, 1980.
12. J. Hurd. Integrating Gandalf and HOL. In *Proceedings TPHOL's 99*, Lecture Notes in Computer Science 1690, pages 311–321. Springer-Verlag, 1999.
13. W. McCune and O. Shumsky. *IVY: A preprocessor and proof checker for first-order logic.* Preprint ANL/MCS-P775-0899, Argonne National Laboratory, Argonne IL, 1999.
14. G. Nadathur and D. Miller. Higher-order logic programming. In D. Gabbay e.a. (eds.) *Handbook of Logic in Artificial Intelligence*, Vol. 5, pages. 499–590. Clarendon Press, Oxford, 1998.
15. Q.-H. Nguyen. Certifying term rewriting proofs in ELAN. *Electronic Notes in Theoretical Computer Science*, Vol. 59.4, 21 pages. Elsevier, 2001.
16. `www.ags.uni-sb.de/~omega/`
17. F. Pfenning. Analytic and non-analytic proofs. In *Proceedings CADE 7*, Lecture Notes in Computer Science 170, pages 394–413. Springer-Verlag, 1984.
18. H. Schwichtenberg. Logic and the Axiom of Choice. In *Logic Colloquium 78*, pages 351-356.
19. G. Sutcliffe. The CADE-16 ATP System Competition. *Journal of Automated Reasoning* **24**, pages 371–396, 2000.
20. J. Smith and T. Tammet. Optimized encodings of fragments of type theory in first-order logic. In *Proceedings Types 95*, Lecture Notes in Computer Science 1158, pages 265–287. Springer-Verlag, 1995.

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

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

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* $\mathrm{refl}(A)$ *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* $\mathrm{trans}(\pi_1, \pi_2)$ *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* $\Pi x_1, x_2, x_3{:}X\ \ x_1 \equiv x_2 \rightarrow x_2 \equiv x_3 \rightarrow x_1 \equiv x_3$.
– func_f *is a proof of* $\Pi x_1, y_1{:}X_1\ \cdots \Pi x_n, y_n{:}X_n\ \ x_1 \equiv y_1 \rightarrow \cdots \rightarrow x_n \equiv y_n \rightarrow (f\ x_1 \cdots x_n) \equiv (f\ y_1 \cdots y_n)$. *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* $1 \le i \le n$.
– $T(\text{abstr}(x, X, \pi)\ )$ *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* $1 \le i \le n$, *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.*

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

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:*

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

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 $C_1, \ldots, C_q$ 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 $F_1, \ldots, F_q$ 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 $x_1, \ldots, x_p$, the instantiated formulas are equivalent, so we need to prove for abitrary $x_1, \ldots, x_p$,

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

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 $A_1, \ldots, A_k$ 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.

Translation of Resolution Proofs into Short First-Order Proofs without Choice Axioms

Hans de Nivelle

Max-Planck Institut für Informatik
Stuhlsatzenhausweg 85
66123 Saarbrücken, Germany

Abstract

We present a way of transforming a resolution-style proof containing Skolemization into a natural deduction proof without Skolemization. The size of the proof increases only moderately (polynomially). This makes it possible to translate the output of a resolution theorem prover into a purely first-order proof that is moderate in size.

Key words: Theorem Proving, Proof Theory, Skolemization.
1991 MSC: 68T15,
1991 MSC: 03F20,
1991 MSC: 03F05

If one wants a resolution based theorem prover to generate explicit proofs, one has to decide what to do with Skolemization. One possibility is to allow Skolemization, (or equivalently the axiom of choice) as a proof principle. In that case, the resolution proof can be translated more or less one-to-one into a natural deduction proof. In [10] it is described how to do this efficiently for the clausal normal form (CNF) transformation. In [6] and [7], a hybrid method was developed. For resolution on the clause level, explicit proofs were generated. For the CNF-transformation, an algorithm was developed inside COQ and proven correct. Using this approach, explicit generation of proofs for the CNF-transformation could be avoided. (Although strictly seen, inside COQ, the term defining the algorithm also defines a proof principle) A related approach was taken in [14], using the Boyer-Moore theorem prover instead of COQ. Both approaches use the axiom of choice. In [6], the axiom of choice was used for proving the clausification algorithm correct. In [14], it is assumed that domains are finite, which implies the axiom of choice.

Email address: `nivelle@mpi-sb.mpg.de` (Hans de Nivelle).
URL: `www.mpi-sb.mpg.de/~nivelle` (Hans de Nivelle).

Another possibility is to completely eliminate the Skolemization steps from the proof. If one is interested in correctness only, the axiom of choice is certainly acceptable, but it is much more elegant to avoid using the axiom of choice at all in proofs of first-order formulas. Until recently, the only known way of eliminating applications of Skolemization from a proof made use of cut elimination. Because of this, these methods can cause a hyperexponential increase in proof size in the worst case, see [21] or [18], or also [4]. In [19], such an algorithm is described in detail. In [13], an improved method is given, which is optimized towards readability of the resulting proof. This method has been implemented in Ωmega by Andreas Meijer. (see [20])

In [1], a method for eliminating Skolem functions from first-order proofs was presented, which results in proofs of polynomial size. The method works only in the context of a theory that is strong enough to encode finite functions. This is a weak requirement, because for example axiomatizations of common datastructures, like lists or arrays would suffice. The finite functions are used to approximate the Skolem functions through an internalized forcing argument. We think that the method could be implemented, but it would not work in the general first-order case.

The general problem whether Skolem functions can be efficiently eliminated from every first-order logic proof seems to be open, see the table in [8], Page 9.

In this paper, we give a general method for eliminating Skolem functions from *resolution proofs*, which can be implemented and expected to be efficient. In addition, it is *structure preserving*, by which we mean that it does almost not change the structure of the proof. The main idea is the following: Assume that f is a Skolem function in the clausal formula $\forall x \; p(x) \vee q(f(x))$. [1] Then f can be replaced by a binary relation F as follows: $\forall x \alpha \; F(x, \alpha) \rightarrow p(x) \vee q(\alpha)$. It turns out that if one replaces Skolem functions by relations in a resolution proof, and for each relation one can show *seriality*, then the result will still be a valid first-order proof. The surprising fact is that resolution does not make use of the functionality of F, only of its seriality. Because f is a Skolem function, it originates from a formula of form $\forall x \exists y \; F(x, y)$. Hence, F can be taken as serial relation. Proofs containing paramodulation steps can also be handled. There is only one restriction on the use of paramodulation, namely that it has to be *simultaneous in the Skolem functions*. Simultaneous in the Skolem functions means that whenever an equality $t_1 \approx t_2$ is applied inside a Skolem term, *all* instances of t_1 that are inside some Skolem term have to be replaced by t_2. The completeness of this restriction follows from the fact that one does not have to paramodulate at all into Skolem terms for completeness. This was proven in [5], and generally accepted as an efficient restriction of

[1] When writing a first-order formula, we assume that the scope of a quantifier extends as far to the right as possible.

Fig. 1. An unsatisfiable set of first-order formulas

$$\forall x \exists y \; p(x,y),$$
$$\forall xyz \; p(x,y) \wedge p(y,z) \rightarrow x \approx z,$$
$$\forall xy \; \neg p(x,y) \vee \neg p(y,x).$$

Fig. 2. Resolution Refutation with Skolem functions

1	$\forall x \; p(x, f(x))$	initial clause
2	$\forall xyz \; \neg p(x,y) \vee \neg p(y,z) \vee x \approx z$	initial clause
3	$\forall xy \; \neg p(x,y) \vee \neg p(y,x)$	initial clause
4	$\forall xz \; \neg p(f(x), z) \vee x \approx z$	resolvent of 1 with 2
5	$\forall x \; x \approx f(f(x))$	resolvent of 1 with 4
6	$\forall x \; p(f(x), x)$	paramodulant of 5 into 1
7	$\forall x \; \neg p(f(x), x)$	resolvent of 1 with 3
8	$\perp$	resolvent of 6 with 7

resolution.

We will give an example of a complete transformation. Consider the set of first-order formulas, given in Figure 1. The set is unsatisfiable, because the first formula requires that there exists a chain of p's which has no end. The second formula ensures that this chain actually is a cycle of length two. The third formula insists that cycles of length two do not exist. Using resolution and paramodulation, one can construct the refutation of Figure 2. The function symbol f is a Skolem symbol. If one replaces f by a binary predicate symbol F, one obtains the set of formulas $1', 2', 3'$ of Figure 3. The refutation of $1, 2, 3$ can be stepwise translated into the refutation of $1', 2', 3'$ of Figure 3. The result is a proof of $\perp$ from $1', 2', 3'$, which is still first-order. The proof does not use any special properties of F. The only condition on F is that it has to be serial, i.e. $\forall x \exists y \; F(x,y)$ must be provable. We can obtain a completely first-order refutation (= proof of the negation) of the formulas in Figure 1, if we can find an F, for which $1', 2', 3'$, together with seriality are provable from the original formulas in Figure 1. This can be easily obtained by putting $F(x,y) := p(x,y)$. In that case, seriality immediately follows from the first formula of Figure 1, and clause $1'$ becomes a tautology.

In the rest of the paper we will show that the method, suggested by the

Fig. 3. Resolution Refutation with Replaced Skolem Functions

1’	$\forall x\alpha\ F(x,\alpha) \rightarrow p(x,\alpha)$	Initial clause.
2’	$\forall xyz\ \neg p(x,y) \lor \neg p(y,z) \lor x \approx z$	Initial clause.
3’	$\forall xy\ \neg p(x,y) \lor \neg p(y,x)$	Initial clause.
4’	$\forall xz\alpha\ F(x,\alpha) \rightarrow \neg p(\alpha,z) \lor x \approx z$	Instantiate 2’ with $y := \alpha$, then resolve with 1’ on $p(x,\alpha)$.
5’	$\forall x\alpha\beta\ F(x,\alpha) \land F(\alpha,\beta) \rightarrow x \approx \beta$	Instantiate 1’ with $x := \alpha,\quad \alpha := \beta$, instantiate 4’ with $z := \beta$, then resolve results on $p(\alpha,\beta)$.
6’	$\forall x\alpha\ F(x,\alpha) \rightarrow p(\alpha,x)$	Instantiate 1’ with $x := \alpha,\quad \alpha := \beta$, then paramodulate from 5’ into $p(\alpha,\beta)$. In the result, remove $F(\alpha,\beta)$ with $\forall x \exists y\ F(x,y)$.
7’	$\forall x\alpha\ F(x,\alpha) \rightarrow \neg p(\alpha,x)$	Instantiate 3’ with $y := \alpha$, Resolve result with 1’ on $p(x,\alpha)$.
8’	$\bot$	resolve 6’ with 7’. After that, remove $F(x,\alpha)$ with $\forall x \exists y\ F(x,\alpha)$.

example, works in general. In Section 3, we show that resolution proofs remain correct first-order proofs, when certain functions are replaced by serial relations in the clauses. We show that the resolution proof can be stepwise translated into a first-order proof. It is probably no surprise that paramodulation steps are the most difficult to translate. Paramodulation is the rule that looks into the term structure, and the replacement of functions by relations modifies the term structure. In order for the translation to be possible, the paramodulation rule has to be slightly restricted. The restriction is very natural, and the resolution provers we are aware of, implement a much stronger restriction, because it improves the efficiency of proof search. In Section 2, we do some introductory work for the translation of paramodulation.

In Section 4, we show that it is in general possible to prove the initial clauses (with replaced Skolem functions) from the initial formulas. We show that this is possible for most of the standard CNF-transformations that are currently in use. It is in general not as easy as in the example, because there can be nesting of existential quantifiers (as in $\forall x_1 \exists y_1 \forall x_2 \exists y_2 \; p(x_1, y_1, x_2, y_2)$), and existential quantifiers may occur in a conditional context (as in $\forall x \; p(x) \rightarrow \exists y \; q(x, y)$). The problems will be discussed in Section 4.

1 Preliminaries

Definition 1 *We assume a fixed set of predicate symbols $\mathcal{P}$ and a fixed set of function symbols $\mathcal{F}$. The sets $\mathcal{P}$ and $\mathcal{F}$ are assumed disjoint. We assume a fixed function* ar, *that attaches to each $f \in \mathcal{F}$ a natural number* $\mathrm{ar}(f) \geq 0$. *In addition,* ar *attaches to each $p \in \mathcal{P}$ a natural number* $\mathrm{ar}(p) \geq 0$. *We assume that for each $n \geq 0$ there are countably infinitely many elements $f \in \mathcal{F}$ with* $\mathrm{ar}(f) = n$.

Similarly, we assume that for each $n \geq 0$, there are countably infinitely many elements $p \in \mathcal{P}$ with $\mathrm{ar}(p) = n$.

We assume that there is no syntactic distinction between variables and constants. We call the elements $c \in \mathcal{F}$, for which $\mathrm{ar}(c) = 0$, either *constants* or *variables* depending on how they are used.

Definition 2 *We recursively define the set of* terms. *If $n \geq 0$, $t_1, \ldots, t_n$ are terms, $f \in \mathcal{F}$ and* $\mathrm{ar}(f) = n$, *then $f(t_1, \ldots, t_n)$ is also a term.*

Next we define the set of atoms. *If $n \geq 0$, $t_1, \ldots, t_n$ are terms, $p \in \mathcal{P}$ and* $\mathrm{ar}(p) = n$, *then $p(t_1, \ldots, t_n)$ is an atom. If t_1, t_2 are terms, then $t_1 \approx t_2$ is an atom. Formulas are recursively defined as follows:*

- *If A is an atom, then A is also a formula,*
- *$\bot$ and $\top$ are formulas,*
- *if A is a formula, then $\neg A$ is also a formula,*
- *if A, B are formulas, then $A \wedge B$, $A \vee B$, $A \rightarrow B$, $A \leftrightarrow B$ are also formulas,*
- *if $x \in \mathcal{F}$ has* $\mathrm{ar}(x) = 0$, *and A is a formula, then $\forall x \; P$ and $\exists x \; P$ are also formulas.*

For our purpose, it is convenient to define clauses as a subset of formulas:

Definition 3 *If A is an atom, then the formulas A and $\neg A$ are* literals. *A literal of form A is called* positive. *A literal of form $\neg A$ is called* negative.

If $F_1, \ldots, F_n$, are formulas with $n > 0$, then $F_1 \vee \cdots \vee F_n$ simply denotes the disjunction of $F_1, \ldots, F_n$. In case that $n = 0$, $F_1 \vee \cdots \vee F_n$ denotes $\bot$.

A clause *is a formula of form $\forall x_1 \cdots x_k \;\; L_1 \vee \cdots \vee L_n$ in which $L_1, \ldots, L_n$ are* *literals, and $k \geq 0$. We assume that the x_i are distinct. The clause is* empty *if $n = 0$. It is ground if $k = 0$.*

Definition 4 *Let $\mathcal{S} \subseteq \mathcal{P} \cup \mathcal{F}$. For each of the objects defined before (formula, term, atom, literal, clause), we call it an object* over $\mathcal{S}$ *if it contains only predicate and free function symbols from $\mathcal{S}$.*

Since we are going to replace function symbols by relations, we need to formally define what a relation is.

Definition 5 *If F is a formula and $x_1, \ldots, x_k \in \mathcal{F}$ have $\text{ar}(x_i) = 0$, then the expression*

$$R = \lambda x_1 \cdots \lambda x_k \ F$$

is a k-ary relation. We also write $\text{ar}(R) = k$.

If $t_1, \ldots, t_n$ are terms, then $R(t_1, \ldots, t_n)$ denotes the formula

$$R[x_1 := t_1] \ [x_2 := t_2] \cdots [x_k := t_k].$$

The notation $[x_i := t_i]$ denotes capture avoiding substitution.

The λ-symbol will not occur in the proofs that we construct, because relations will be always instantiated in proofs.

We now define function replacements. In order to define a function replacement, one needs to specify the function symbols that will be replaced. Terms that have such a function symbol on top will be replaced by fresh variables. As a consequence, one also needs to specify a set of fresh variables that will be big enough.

Definition 6 *We write $\mathcal{F}_{\text{Prob}}$ for the set of function symbols that occur in the orginal problem and its resolution proof, including the Skolem function symbols. Obviously $\mathcal{F}_{\text{Prob}} \subseteq \mathcal{F}$.*

We assume a subset $\mathcal{F}_{\text{Repl}}$ of $\mathcal{F}_{\text{Prob}}$, specifying the function symbols that will be replaced. (These would normally be the Skolem functions)

Let $\mathcal{F}_{\text{Def}}$ with $\mathcal{F}_{\text{Prob}} \cap \mathcal{F}_{\text{Def}} = \emptyset$ be the set of variables that will be used as definitions. We use greek letters $\alpha, \beta, \gamma, \delta$ to denote elements of $\mathcal{F}_{\text{Def}}$. It is assumed that $\mathcal{F}_{\text{Def}}$ is countably infinite.

The function replacement *is a function $[\ ]$ that*

- *assigns to each $f \in \mathcal{F}_{\text{Repl}}$ a relation R_f, s.t. $\text{ar}(R_f) = \text{ar}(f) + 1$.*

- *assigns to each term* $f(t_1, \ldots, t_n)$ *with* $f \in \mathcal{F}_{\text{Repl}}$, *and* $t_1, \ldots, t_n$ *over* $\mathcal{F}_{\text{Prob}}$ *a unique element* $\alpha \in \mathcal{F}_{\text{Def}}$.

Definition 7 *Let* $\mathcal{F}_{\text{Repl}}$, $\mathcal{F}_{\text{Def}}$ *and* $[\ ]$ *be defined as in Definition 6. The function* $[\ ]$ *is extended to terms over* $\mathcal{F}_{\text{Prob}}$ *as follows: The range of extended* $[\ ]$ *is the set of terms over* $(\mathcal{F}_{\text{Prob}} \backslash \mathcal{F}_{\text{Repl}}) \cup \mathcal{F}_{\text{Def}}$.

- *For a term* $f(t_1, \ldots, t_n)$ *with* $f \in \mathcal{F}_{\text{Repl}}$, *the replacement* $[f(t_1, \ldots, t_n)]$ *is as defined by Definition 6.*
- *For a term* $f(t_1, \ldots, t_n)$ *with* $f \in \mathcal{F}_{\text{Prob}} \backslash \mathcal{F}_{\text{Repl}}$, *the replacement* $[f(t_1, \ldots, t_n)]$ *is defined as* $f(\ [t_1], \ldots, [t_n]\ )$.

For a quantifier-free formula F, *we define* $[F]$ *as the result of replacing each term* t *in* F *by its corresponding* $[t]$.

For a quantifier-free formula F, *we define*

- *the set* $\text{Var}(F)$ *as*

$$\{\alpha \in \mathcal{F}_{\text{Def}} \mid \exists t' \textit{ in } F, \textit{ s.t. } \alpha = [t']\ \}.$$

  These are all the variables of $\mathcal{F}_{\text{Def}}$ *that were introduced for defining a subterm of F.*
- *the* definition set $\text{Def}(F)$ *as the set*

$$\{\ [f](\ [t_1], \ldots, [t_n],\ \alpha) \mid \alpha \in \text{Var}(F), \quad \alpha = [\ f(t_1, \ldots, t_n)\ ]\ \}.$$

For a term t, *the notions* $\text{Var}(t)$ *and* $\text{Def}(t)$ *are defined correspondingly. For a sequence of quantifier-free formulas and terms* $U_1, \ldots, U_n$ *(possibly mixed), we define* $\text{Var}(U_1, \ldots, U_n) = \text{Var}(U_1) \cup \cdots \cup \text{Var}(U_n)$, *and* $\text{Def}(U_1, \ldots, U_n) = \text{Def}(U_1) \cup \cdots \cup \text{Def}(U_n)$.

Lemma 8 *For each term* t' *over* $(\mathcal{F}_{\text{Prob}} \backslash \mathcal{F}_{\text{Repl}}) \cup \mathcal{F}_{\text{Def}}$, *there is at most one term* t *over* $\mathcal{F}_{\text{Prob}}$, *s.t.* $[t] = t'$.

For a variable free formula F, $\text{Var}(F)$ and $\text{Def}(F)$ depend only on the terms in F, and not on the formula structure of F. Therefore it is possible to write $\text{Var}(F, G)$ instead of $\text{Var}(F \wedge G)$ or $\text{Def}(t_1, t_2, F)$ instead of $\text{Def}(t_1 \approx t_2 \vee F)$, etc.

Example 9 *Let* A *be the atomic formula* $p(\ s(f(s(f(0))))\ )$. *Assume that* $\mathcal{F}_{\text{Repl}} = \{f\}$ *and* $[f] = F$. *Further assume that* $[f(0)] = \alpha$, $[f(s(f(0)))] = \beta$. *Then*

$[s(f(s(f(0))))] = s(\beta)$, $\qquad [f(s(f(0)))] = \beta$,

$[s(f(0))] = s(\alpha)$, $\qquad [f(0)] = \alpha$,

$[0] = 0$.

$\mathrm{Var}(A) = \{\alpha, \beta\}$. $\mathrm{Def}(A) = \{F(0, \alpha),\ F(s(\alpha), \beta)\}$.

We will use the previous definitions to replace a clause

$$C = \forall x_1 \cdots x_k \ \ L_1 \vee \cdots \vee L_n$$

by

$$\forall x_1 \cdots x_k \ \ \forall\ \mathrm{Var}(L_1, \ldots, L_n) \ \bigwedge \mathrm{Def}(L_1, \ldots, L_n) \rightarrow [L_1] \vee \cdots \vee [L_n].$$

We will usually omit the $\bigwedge$-symbol.

Example 10 *Let $C = \forall x\ p(x, f(x)) \vee q(f(f(x)), x)$ be a clause. Assume that $\mathcal{F}_{\mathrm{Repl}} = \{f\}$, $\ [f] = F$, $\ [f(x)] = \alpha$, and $[f(f(x))] = \beta$. Then the translation of C equals*

$$\forall x\ \forall \alpha\beta\ F(x, \alpha) \wedge F(\alpha, \beta) \rightarrow p(x, \alpha) \vee q(\beta, x).$$

It may appear strange that the value of [ ] depends on the syntactic appearance of a term. For example, one has $[\ f(x)\ ] = \alpha$, $[\ f(y)\ ] = \beta$, while at the same time, the clauses $\forall x\ p(f(x))$ and $\forall y\ p(f(y))$ are α-equivalent. The explanation for this fact is that we introduced a global translation function [ ] for convenience only. It would suffice to define a distinct replacement function $[\ ]_C$ for each clause C. However, this would only complicate the presentation of the translations in the next section, without introducing more generality. The clauses $\forall x\ p(f(x))$ and $\forall y\ p(f(y))$ will be translated as $\forall x\alpha\ F(x, \alpha) \rightarrow p(\alpha)$ and $\forall y\beta\ F(y, \beta) \rightarrow p(\beta)$, which are again α-equivalent. In practice, if one implements the translation method, it may be inefficient to construct a global replacement function, because it needs to store all terms that occur in the proof.

2 Term Replacement

In this section we explain how paramodulation behaves in combination with function replacements. The results in this section are the essence of the translation method. We define three related concepts, and show that they have related properties. The concepts are *substitutions, generalized substitutions* *and systems of equations.* A substitution is defined as usual. It assigns terms to variables. In the context of a function replacement, it has to be extended to the variables in $\mathcal{F}_{\text{Def}}$, which is unproblematic.

A generalized substitution is a set of replacement rules of form $t := u$, where t and u are arbitrary terms. When it is applied, *every occurrence* of t has to be replaced by u. Using generalized substitutions, it is possible to define *simultaneous paramodulation.* In [12], it was shown that Skolem functions can be eliminated from resolution proofs in which all paramodulation steps are simultaneous.

In this paper, we show that it is possible to use a more general form of paramodulation, which we call *non-separating paramodulation.* In non-separating paramodulation, replacement is controlled by *extensions* of *systems of equations.* Roughly speaking, non-separating paramodulation means that it is not allowed to introduce a distinction between two Skolem terms by equality replacement.

Example 11 *Consider the equality* $0 \approx 1$, *and the clause* $p(f(0), 0)$. *Assume that* $\mathcal{F}_{\text{Repl}} = \{f\}$, $[f] = F$, *and that* $[f(0)] = \alpha$, $[f(1)] = \beta$. *The translation of* $p(f(0), 0)$ *(as a clause) equals*

$$\forall\alpha\ F(0, \alpha) \rightarrow p(\alpha, 0).$$

Using paramodulation from the equality $0 \approx 1$, *one can obtain each of the following clauses*

clause	*translation*
$p(f(1), 0)$,	$\forall\beta\ F(1, \beta) \rightarrow p(\beta, 0)$,
$p(f(0), 1)$,	$\forall\alpha\ F(0, \alpha) \rightarrow p(\alpha, 1)$,
$p(f(1), 1)$,	$\forall\beta\ F(1, \beta) \rightarrow p(\beta, 1)$.

Example 12 *Now consider the equality* $f(0) \approx f(1)$, *and the clause* $p(f(0), f(0))$. *Let* $\mathcal{F}_{\text{Repl}}$ *and* $[\ ]$ *be as in the previous example. The translation of* $f(0) \approx f(1)$ *equals* $\forall\alpha\beta\ F(0, \alpha) \wedge F(1, \beta) \rightarrow \alpha \approx \beta$. *The translation of* $p(f(0), f(0))$ *equals* $\forall\alpha\ F(0, \alpha) \rightarrow p(\alpha, \alpha)$. *The following clauses can be obtained by paramodulation:*

clause	*translation*
$p(f(0), f(1))$,	$\forall\alpha\beta\ F(0,\alpha) \land F(1,\beta) \rightarrow p(\alpha,\beta)$,
$p(f(1), f(0))$,	$\forall\alpha\beta\ F(0,\alpha) \land F(1,\beta) \rightarrow p(\beta,\alpha)$,
$p(f(1), f(1))$,	$\forall\beta\ F(0,\beta) \rightarrow p(\beta,\beta)$.

The last clause is derived through $\forall\alpha\beta\ F(0,\alpha) \land F(1,\beta) \rightarrow p(\beta,\beta)$, *from which* $F(0,\alpha)$ *can be removed through the seriality axiom for* F.

The examples show the principle of how paramodulation steps can be reconstructed after translation by a function replacement. If some term t_1 that needs to be replaced, occurs inside some literal R, then $[t_1]$ occurs either in $[R]$ or in $\mathrm{Def}(R)$, and the replacement can be made there.

Example 13 *Consider the equality* $0 \approx 1$, *and the clause* $p(f(0), f(0))$. *Let* $[\ ]$ *and* $\mathcal{F}_{\mathrm{Repl}}$ *be defined as in the previous examples. From* $\forall\alpha\ F(0,\alpha) \rightarrow p(\alpha,\alpha)$ *and* $0 \approx 1$, *one can prove* $\forall\beta\ F(1,\beta) \rightarrow p(\beta,\beta)$, *but not* $\forall\alpha\beta\ F(0,\alpha) \land F(1,\beta) \rightarrow p(\alpha,\beta)$.

The last example shows the main problem when translating arbitrary paramodulation steps. If one wants to replace t_1 by t_2 inside some atom R, and $[t_1]$ occurs in $\mathrm{Def}(R)$, then all subterms that depend on the occurrence $[t_1]$ will be automatically modified. Because of this reason, one does not have unlimited freedom in choosing which occurrences of t_1 are to be replaced, and which ones remain unchanged. For this reason, only simultaneous paramodulation was considered in [12]. In this paper, we show that a weaker restriction of paramodulation, which we will call *non-separating*, works as well.

We now define both substitutions and generalized substitutions, and how they are translated by a function replacement $[\ ]$.

Definition 14 *A* substitution *is a set of form* $\Theta = \{x_1 := t_1, \ldots, x_k := t_k\}$, *s.t.* $(x_{i_1} = x_{i_2}) \Rightarrow (t_{i_1} = t_{i_2})$. *Each* x_i *is a variable, and each* t_i *is a term.*

The application of Θ on a term t, *notation* $t \cdot \Theta$, *is recursively defined as follows (in the standard way):*

- *if* t *equals one of the* x_i, *then* $t \cdot \Sigma = t_i$.
- *Otherwise, write* $t = f(w_1, \ldots, w_n)$. *The application* $f(w_1, \ldots, w_n) \cdot \Sigma$ *equals* $f(w_1 \cdot \Sigma, \ldots, w_n \cdot \Sigma)$.

The application of Θ *on a quantifier-free formula* F *is defined term-wise.*

Definition 15 *A* generalized substitution *is a set of form* $\Sigma = \{t_1 := u_1, \ldots, t_k := u_k\}$, *s.t. there exist no distinct* i_1, i_2 *with* $1 \leq i_1, i_2 \leq k$, *and* t_{i_1} *is a subterm of* t_{i_2}. *The application of* Σ *on a term* t, *notation* $t \cdot \Sigma$, *is recursively defined as follows:*

- *If t equals one of the t_i, then $t \cdot \Sigma = u_i$.*
- *Otherwise, write $t = f(w_1, \ldots, w_n)$. The application $f(w_1, \ldots, w_n) \cdot \Sigma$ equals $f(w_1 \cdot \Sigma, \ldots, w_n \cdot \Sigma)$.*

The application of Σ on a quantifier-free formula F is defined term-wise.

Substitutions and generalized substitutions are closely related. One could say that substitutions are 'a subclass' of generalized substitutions. We now define how a function replacement [ ] translates a generalized substitution. By 'inheritance', the translation also applies to simple substitutions.

Definition 16 *Let $\Sigma = \{t_1 := u_1, \ldots, t_k := u_k\}$ be a generalized substitution on terms over $\mathcal{F}_{\text{Prob}}$. Let [ ] be a function replacement, replacing functions from $\mathcal{F}_{\text{Repl}} \subseteq \mathcal{F}_{\text{Prob}}$ and introducing variables from $\mathcal{F}_{\text{Def}}$.*

We define the replacement of Σ, *for which we write $[\Sigma]$, as the union of a substitution and a generalized substitution. The first one, $[\Sigma]_{\text{Prob}}$ contains the straightforward translation of Σ by [ ]. The second one, $[\Sigma]_{\text{Def}}$, defines the translation of the application operator on $\mathcal{F}_{\text{Def}}$.*

$$[\Sigma]_{\text{Prob}} = \{\ [t_1] := [u_1], \ldots, [t_k] := [u_k]\ \}$$

$$[\Sigma]_{\text{Def}} = \{\ \alpha := [t \cdot \Sigma] \mid \alpha \in \mathcal{F}_{\text{Def}},\quad \alpha = [t] \text{ and } t \neq t \cdot \Sigma\ \}.$$

$$[\Sigma] = [\Sigma]_{\text{Prob}} \cup [\Sigma]_{\text{Def}}.$$

Note that the notation $[\Sigma]_{\text{Prob}}$ is slightly misleading, because the $[t_i]$ and $[u_j]$ can contain variables from $\mathcal{F}_{\text{Def}}$ as well. It is easily checked that $[\Sigma]$ is always a well-formed, generalized substitution.

Theorem 17 *Let $\Sigma = \{t_1 := u_1, \ldots, t_k := u_k\}$ be a generalized substitution on terms over $\mathcal{F}_{\text{Prob}}$. Let [ ] be a function replacement, replacing functions from $\mathcal{F}_{\text{Repl}} \subseteq \mathcal{F}_{\text{Prob}}$, and introducing variables from $\mathcal{F}_{\text{Def}}$. For every term term t over $\mathcal{F}_{\text{Prob}}$,*

$$[t \cdot \Sigma] = [t] \cdot [\Sigma].$$

PROOF. We use induction on the term structure of t.

- If t equals one of the t_i, then $[t_i \cdot \Sigma] = [u_i]$, by construction of $[\Sigma] \supseteq [\Sigma]_{\text{Prob}}$.
- If t does not equal any of the t_i, and $[t] \in \mathcal{F}_{\text{Def}}$, then $[t] \cdot [\Sigma] = [t \cdot \Sigma]$, by construction of $[\Sigma] \supseteq [\Sigma]_{\text{Def}}$.

- If t does not equal any of the t_i, and $[t] \notin \mathcal{F}_{\mathrm{Def}}$, then write $t = g(w_1, \ldots, w_n)$. We have $[g(w_1, \ldots, w_n) \cdot \Sigma] = [g(w_1 \cdot \Sigma, \ldots, w_n \cdot \Sigma)] = g([w_1 \cdot \Sigma], \ldots, [w_n \cdot \Sigma])$. By induction, this equals $g([w_1] \cdot [\Sigma], \ldots, [w_n] \cdot [\Sigma])$. But this is equal to $[g(w_1, \ldots, w_n)] \cdot [\Sigma]$, because $[t] \neq [t_i]$ and $[t] \notin \mathcal{F}_{\mathrm{Def}}$.

Theorem 18 *Let $\Sigma = \{t_1 := u_1, \ldots, t_k := u_k\}$ be a generalized substitution on terms over $\mathcal{F}_{\mathrm{Prob}}$. Let $[\ ]$ be a function replacement, replacing functions from $\mathcal{F}_{\mathrm{Repl}} \subseteq \mathcal{F}_{\mathrm{Prob}}$, and introducing variables from $\mathcal{F}_{\mathrm{Def}}$.*

For every term t over $\mathcal{F}_{\mathrm{Prob}}$,

$$[t_1] \approx [u_1], \ldots, [t_k] \approx [u_k] \vdash ([t] \cdot [\Sigma]) \approx ([t] \cdot [\Sigma]_{\mathrm{Def}}).$$

PROOF. The missing replacements can be made up by equality replacement.

In the rest of this paper, we will only use substitutions, not generalized substitutions. Theorem 18 will not be used. We have included it here for sake of completeness, because it was used in [12]. Instead we prove a more general statement about which equality replacements can be translated. Before we state it, we give an example:

Example 19 *Consider the equality $0 \approx 1$, and the clause $p(f(0,0), f(0,0))$. Assume that $\mathcal{F}_{\mathrm{Repl}} = \{f\}$, $\quad [f] = F$, and that*

$$[f(0,0)] = \alpha, \quad [f(0,1)] = \beta, \quad [f(1,0)] = \gamma, \quad [f(1,1)] = \delta.$$

Using $[\ ]$, the clause $p(f(0,0), f(0,0))$ translates into

$$\forall\alpha\ F(0,0,\alpha) \rightarrow p(\alpha,\alpha).$$

Using paramodulation from $0 \approx 1$, the following 3 clauses can be obtained:

$$\forall\beta\ F(0,1,\beta) \rightarrow p(\beta,\beta),$$
$$\forall\gamma\ F(1,0,\gamma) \rightarrow p(\gamma,\gamma),$$
$$\forall\delta\ F(1,1,\delta) \rightarrow p(\delta,\delta).$$

The following clauses are examples of clauses that cannot be obtained:

$$\forall\alpha\beta\ F(0,0,\alpha) \land F(0,1,\beta) \rightarrow p(\alpha,\beta),$$
$$\forall\beta\gamma\ F(0,1,\beta) \land F(1,0,\gamma) \rightarrow p(\beta,\gamma).$$

Example 19 shows that one does not always have to replace all occurrences. On the other side, one also does not have a full freedom when deciding which occurrences are to be replaced. If one wants to paramodulate from an equation $t_1 \approx t_2$ into a literal A, then the possibilities are determined by the occurrences of $[t_1]$ in $[A]$. In the clause of Example 19, the first and second occurrence of 0 are represented by distinct arguments of F. However the (first and third), and the (second and fourth) occurrence are represented by the same argument of F. Therefore these cannot be separated.

Whenever some term, constructed by a function symbol in $\mathcal{F}_{\text{Repl}}$, has more than one occurrence, all occurrences are represented by the same variable in the [ ]-translation. Therefore, paramodulation must be carried out in such a way that it does not introduce a distinction between identical terms with a symbol from $\mathcal{F}_{\text{Repl}}$ on top. In the previous example, $f(0,0)$ was such a term. We call the resulting restriction of paramodulation *non-separating.*

We now define the notion of a *system of equations.* Only systems with $k = 1$ will be used in this paper, but the results that we prove in this section also hold for $k > 1$.

Definition 20 *A* system of equations $\mathcal{E}$ *is a set of form* $\mathcal{E} = \{u_1 \approx t_1, \ldots, u_k \approx t_k\}$, *where* $u_1, t_1, \ldots, u_k, t_k$ *are terms.*

Replacement of equals is controlled by *extensions.* An extension determines how replacements are made inside identical $\mathcal{F}_{\text{Repl}}$-terms.

Definition 21 *Let* $\mathcal{E} = \{u_1 \approx t_1, \ldots, u_k \approx t_k\}$ *be a system of equations with terms over* $\mathcal{F}_{\text{Prob}}$. *Let* t *and* u *be two terms over* $\mathcal{F}_{\text{Prob}}$. *We write* $\mathcal{E}(t, u)$ *if* u *can be obtained from* t *by finitely often replacing a* t_i *by its* u_i *(or a* u_i *by its* t_i*), at arbitrary positions, but never in the scope of a function* $f \in \mathcal{F}_{\text{Repl}}$.

An extension Σ *of* $\mathcal{E}$ *is a function from the set of terms over* $\mathcal{F}_{\text{Prob}}$ *to itself. For every term* $f(w_1, \ldots, w_n)$ *over* $\mathcal{F}_{\text{Prob}}$, *the following recursive condition must hold:*

- *If* $f \in \mathcal{F}_{\text{Repl}}$, *then*

  $$f(w_1, \ldots, w_n) \cdot \Sigma \text{ has form } f(v_1, \ldots, v_n),$$

  and for each i *with* $1 \leq i \leq n$, *it must be the case that*

  $$\mathcal{E}(\ v_i,\ w_i \cdot \Sigma\ ).$$

- *If* $f \notin \mathcal{F}_{\text{Repl}}$, *then*

  $$f(w_1, \ldots, w_n) \cdot \Sigma = f(w_1 \cdot \Sigma, \ldots, w_n \cdot \Sigma).$$

The application of Σ on a quantifier-free formula F is obtained by applying Σ on each top level term in F.

We write $t \cdot \Sigma$ instead of $\Sigma(t)$, because of the close relation with the extension of a generalized substitution. The $\mathcal{E}$-relation allows arbitrary replacement of equals by equals, but not in the scope of a function symbol from $\mathcal{F}_{\text{Repl}}$. Replacements inside the scope of a function symbol from $\mathcal{F}_{\text{Repl}}$ are controlled by the extension, which ensures that the same term is always rewritten in the same way. The non-separating paramodulation rule is defined in Definition 32, using systems of equations and their extensions.

Example 22 *In the first paramodulant of Example 19,* $\mathcal{E} = \{0 \approx 1\}$, *and* $f(0,0) \cdot \Sigma = f(0,1)$. *For the atom* $p(f(0,0), f(0,0))$, *the only atom A with* $\mathcal{E}(\, p(f(0,0), f(0,0)), A)$ *equals* $p(f(0,1), f(0,1))$. *For atom* $q(f(0,0), 0, f(0,0))$, *there would be two possibilities,* $q(f(0,1), 0, f(0,1))$ *and* $q(f(0,1), 1, f(0,1))$.

Definition 23 *Let* $\mathcal{E} = \{t_1 \approx u_1, \ldots, t_k \approx u_k\}$ *be a system of equations with terms over* $\mathcal{F}_{\text{Prob}}$. *The* replacement of $\mathcal{E}$, *written as* $[\mathcal{E}]$, *is defined as the system of equations*

$$\{\ [t_1] \approx [u_1], \ldots, [t_k] \approx [u_k]\ \}.$$

Let Σ be an extension of $\mathcal{E}$. The replacement $[\Sigma]$ *of Σ is defined as the substitution*

$$[\Sigma] = \{\ \alpha := [t \cdot \Sigma] \mid \alpha \in \mathcal{F}_{\text{Def}},\ \textit{and}\ \alpha = [t]\ \}.$$

Theorem 24 *For every term t over* $\mathcal{F}_{\text{Prob}}$,

$$[t \cdot \Sigma] = [t] \cdot [\Sigma].$$

For every quantifier-free formula F with terms over $\mathcal{F}_{\text{Prob}}$,

$$[F \cdot \Sigma] = [F] \cdot [\Sigma].$$

Theorem 25 *For each pair w_1, w_2 of terms over* $\mathcal{F}_{\text{Prob}}$,

$$\mathcal{E}(w_1, w_2)\ \textit{implies}\ [\mathcal{E}] \vdash [w_1] \approx [w_2].$$

PROOF. It is enough to show the lemma under the assumption that w_2 can be obtained from w_1 by a single replacement. Suppose that there is an equation $(t \approx u) \in \mathcal{E}$, s.t. there is a position π in w_1 and w_2, s.t. w_1 and w_2 differ only at position π, w_1 contains t on π, and w_2 contains u on π., Then $([t] \approx [u]) \in [\mathcal{E}]$, $[w_1]$ and $[w_2]$ differ only at position π, $[w_1]$ contains $[t]$ on π and $[w_2]$ contains $[u]$ on π.

Theorem 26 *Let $\mathcal{E} = \{t_1 \approx u_1, \ldots, t_k \approx u_k\}$ be a system of equations with terms over $\mathcal{F}_{\mathrm{Prob}}$. Let Σ be an extension of $\mathcal{E}$. Let z_1 be a term over $\mathcal{F}_{\mathrm{Prob}}$, which is constructed by a function symbol $f \in \mathcal{F}_{\mathrm{Repl}}$. Let z_2 be some term over $\mathcal{F}_{\mathrm{Prob}}$ that contains z_1.*

Then either $z_1 \cdot \Sigma$ is a subterm of $z_2 \cdot \Sigma$, or $z_1 \cdot \Sigma$ is a subterm of one of the terms $t_1, u_1, \ldots, t_k, u_k$.

PROOF. Suppose that $z_1 \cdot \Sigma$ is not a subterm of $z_2 \cdot \Sigma$. Let z' be a smallest subterm of z_2, s.t.

- z_1 is a strict subterm of z' and z' is a subterm of z_2,
- z' has form $f(w_1, \ldots, w_n)$ with $f \in \mathcal{F}_{\mathrm{Repl}}$.
- $z_1 \cdot \Sigma$ is not contained in $z' \cdot \Sigma$.

We show that such z' exists. First, let z'' be a smallest subterm of z_2 which contains z_1 and for which $z_1 \cdot \Sigma$ is not a subterm of $z'' \cdot \Sigma$. Then, if all subterms between z_1 and z'' would have a function symbol $f \notin \mathcal{F}_{\mathrm{Repl}}$ on top, then $z_1 \cdot \Sigma$ would be a subterm of $z'' \cdot \Sigma$, because $f(w_1, \ldots, w_n) \cdot \Sigma = f(w_1 \cdot \Sigma, \ldots, w_n \cdot \Sigma)$ in case that $f \notin \mathcal{F}_{\mathrm{Repl}}$,

The application $f(w_1, \ldots, w_n) \cdot \Sigma$ has form $f(v_1, \ldots, v_n)$. Term z_1 is a subterm of one of the w_i. By definition of extension, $\mathcal{E}(\ w_i \cdot \Sigma,\ v_i\ )$. By minimality of z', it must be the case that $z_1 \cdot \Sigma$ is a subterm of $w_i \cdot \Sigma$. From the construction of z', it follows that $z_1 \cdot \Sigma$ is not a subterm of v_i. Because $\mathcal{E}(w_i \cdot \Sigma,\ v_i)$, it is possible to rewrite $w_i \cdot \Sigma$ into v_i, using the equalities in $\mathcal{E}$. In the rewrite sequence, there is a last term that still contains $z_1 \cdot \Sigma$. Because $z_i \cdot \Sigma$ is constructed by a term in $\mathcal{F}_{\mathrm{Repl}}$, rewriting inside $z_1 \cdot \Sigma$ is not allowed. Therefore the equality $t_j \approx u_j$ (with $1 \leq j \leq k$) that removes $z_1 \cdot \Sigma$ must contain $z_1 \cdot \Sigma$.

3 Translation of Resolution on the Clause Level

In this section we will show the following: Let [ ] be some function replacement replacing functions from $\mathcal{F}_{\mathrm{Repl}}$ and introducing variables from $\mathcal{F}_{\mathrm{Def}}$. Let S be some set of clauses. If S has a resolution refutation in which all paramodulation steps are non-separating, (relative to [ ]) and $[S]$ is obtained from S by replacing each clause $\forall x_1 \cdots x_k\ R$ by its translation $\forall x_1 \cdots x_k\ \forall\ \mathrm{Var}(R)\ \mathrm{Def}(R) \rightarrow [R]$, then $[S]$ has a natural deduction refutation with size bounded by a polynomial in the size of the refutation of S. For each replaced function f, the translation $[f]$ must be serial, which means the following:

Definition 27 *Let R be an $(n+1)$-ary relation. The* seriality axiom *for R is the formula $\forall x_1 \cdots x_n\ \exists y\ R(x_1, \ldots, x_n, y)$.*

The refutation of $[S]$ can be obtained by step-by-step translation of the proof steps. We will sum up the standard resolution (+-paramodulation) rules, as they can be found for example in [15], and show that for each rule the translation of the conclusion is provable from the translations of the premises. We will not explicitly determine the complexity bound, because it will be clear from the proof constructions that the complexity bound is of low polynomial degree. It is not useful to determine the degree more explicitly, because its exact value would depend on details of the calculus used.

In order for the translation to work, paramodulation needs to be *non-separating*, which intuitively means that equality replacement cannot introduce a distinction between two Skolem terms in a clause.

In resolution, instantiation is controlled by unification of terms or literals that need to be made equal before the rule can be applied. The use of unification is important for efficiency, but not important for soundness of the rules. Therefore we can define a separate instantiation rule, and assume that the other rules do not instantiate, which simplifies the presentation.

We now define instantiation. The definition is more complicated than usual, because we cannot make use of implicit quantification, but apart from that, it is completely standard.

Definition 28 *A* generalization Λ *is a set of form* $\{x_1, \ldots, x_m\}$, *s.t. each* x_i *is a variable.*

Definition 29 *Let* $C = \forall x_1 \cdots x_k \; R$ *and* $D = \forall y_1 \cdots y_m \; S$ *be clauses. We call* D an instance *of* C *if there exists a substitution* Θ, *which assigns only to variables from* $x_1, \ldots, x_k$, *s.t.* $R \cdot \Theta = S$, *and for the generalization* $\{y_1, \ldots, y_m\}$, *none of the variables* $y_1, \ldots, y_m$ *is free in* $\forall x_1 \cdots x_k \; R$.

It is easily checked that $C \models D$, if D is an instance of C. We treat permutation separately:

Definition 30 *Clauses* $\forall x_1 \cdots x_k \;\; L_1 \vee \cdots \vee L_m$ *and* $\forall x_1 \cdots x_k \;\; M_1 \vee \cdots \vee M_n$ *are permutations of each other if*

$$\{L_1, \ldots, L_m\} = \{M_1, \ldots, M_n\}.$$

Definition 31 *We define the unary rules:*

equality swapping $$\frac{\forall x_1 \cdots x_k \;\; t_1 \approx t_2 \vee R}{\forall x_1 \cdots x_k \;\; t_2 \approx t_1 \vee R} \qquad \frac{\forall x_1 \cdots x_k \;\; t_1 \not\approx t_2 \vee R}{\forall x_1 \cdots x_k \;\; t_2 \not\approx t_1 \vee R}$$

equality reflexivity $$\frac{\forall x_1 \cdots x_k \;\; t \not\approx t \vee R}{\forall x_1 \cdots x_k \;\; R}$$

equality factoring $\dfrac{\forall x_1 \cdots x_k \;\; t_1 \approx t_2 \lor t_1 \approx t_3 \lor R}{\forall x_1 \cdots x_k \;\; t_1 \approx t_2 \lor t_2 \not\approx t_3 \lor R}$

Definition 32 *We define the binary rules:*

resolution $\dfrac{\forall x_1 \cdots x_k \;\; A \lor R_1 \qquad \forall x_1 \cdots x_k \;\; \neg A \lor R_2}{\forall x_1 \cdots x_k \;\; R_1 \lor R_2}$

non-separating paramodulation
Let $\mathcal{E} = \{t_1 \approx t_2\}$. *Let* Σ *be an extension of* $\mathcal{E}$. *Assume that* $\mathcal{E}(\; R_2',\; R_2 \cdot \Sigma \;)$, *Then*

$$\frac{\forall x_1 \cdots x_k \;\; t_1 \approx t_2 \lor R_1 \qquad \forall x_1 \cdots x_k \;\; R_2}{\forall x_1 \cdots x_k \;\; R_1 \lor R_2'}$$

The intuitive meaning of the non-separating paramodulation rule is as follows: If some term $f(w_1, \ldots, w_n)$ with $f \in \mathcal{F}_{\mathrm{Repl}}$ contains an occurrence of t_1, and there are multiple occurrences of $f(w_1, \ldots, w_n)$ in R_2, then t_1 has to be replaced by t_2 either in all of them or in neither of them.

Example 33 *Assume that* $\mathcal{F}_{\mathrm{Repl}} = \{f, g\}$. *Using equality* $0 \approx 1$, *we have*

$p(0,0,0)$	$\Rightarrow$	$p(0,1,1)$	*possible,*
$q(s(0,0), s(0,0))$	$\Rightarrow$	$q(s(0,1), s(1,0))$	*possible because* $s \notin \mathcal{F}_{\mathrm{Repl}}$,
$q(f(0,0), f(0,0))$	$\Rightarrow$	$q(f(1,1), f(0,0))$	*not possible,*
$p(f(0,0), f(0,0), 0)$	$\Rightarrow$	$p(f(0,1), f(0,1), 0)$	*possible,*
$p(f(0,0), f(0,0), 0)$	$\Rightarrow$	$p(f(1,0), f(1,0), 1)$	*possible,*
$q(f(0,0), g(0,0))$	$\Rightarrow$	$q(f(0,1), g(1,0))$	*possible.*

In case one does not paramodulate into Skolem terms, which is known to be complete, and the standard strategy in all theorem provers that we are aware of, then all paramodulation steps will be automatically non-separating.

We provide the translations for the derivation rules:

Assume that the clause $\forall y_1 \cdots y_m\ S$ is an instance of the clause $\forall x_1 \cdots x_k\ R$ through substitution Θ and generalization Λ. We need to construct a proof that

$$\forall x_1 \cdots x_k\ \forall\ \mathrm{Var}(R)\ \ \mathrm{Def}(R) \rightarrow [R]$$

implies

$$\forall y_1 \cdots y_m\ \forall\ \mathrm{Var}(S)\ \ \mathrm{Def}(S) \rightarrow [S].$$

Write $\Theta = \{x_1 := t_1, \ldots, x_k := t_k\}$. We have $R \cdot \Theta = S$. The generalization Λ equals $\{y_1, \ldots, y_m\}$, and none of the y_j is free in $\forall x_1 \cdots x_k\ R$.

Let $[\Theta]$ be constructed from Θ as in Definition 16. It is easily checked that $(x := t) \in [\Theta]$ implies $x \in \mathcal{F}_{\mathrm{Def}}$ or x is among the $x_1, \ldots, x_k$. As a consequence $[\Theta]$ is a substitution and it is possible to construct the proof given in Figure 4. We justify the proof steps:

S1 Because $y_1, \ldots, y_m$ are not free in $\forall x_1 \cdots x_k\ R$, they are also not free in $\forall x_1 \cdots x_k\ \forall\ \mathrm{Var}(R)\ \ \mathrm{Def}(R) \rightarrow [R]$. Therefore, the $y_1, \ldots, y_m$ are fresh.
S2 If a variable $\alpha \in \mathrm{Var}(S)$ occurs in $[R]$, then it also occurs in $\mathrm{Var}(R)$. Hence it is still fresh.
S3 An assumption.
S4 $[\Theta]$ is a well-formed substitution.
S5 It is easily seen that $(\mathrm{Def}(R) \rightarrow [R]) \cdot [\Theta] = (\mathrm{Def}(R) \cdot [\Theta]) \rightarrow ([R] \cdot [\Theta])$, but we also need to check that all atoms in $\mathrm{Def}(R) \cdot [\Theta]$ are provable. Let $A \in \mathrm{Def}(R)$. Then A has form $R_f(\ [w_1], \ldots, [w_n], [f(w_1, \ldots, w_n)]\ )$, where $f \in \mathcal{F}_{\mathrm{Repl}}$ and $f(w_1, \ldots, w_n)$ occurs in R. Because f is not in the domain of Θ, $\ f(w_1, \ldots, w_n) \cdot \Theta$ occurs in S and equals $f(w_1 \cdot \Theta, \ldots, w_n \cdot \Theta)$. As a consequence, we have the atom $R_f([w_1 \cdot \Theta], \ldots, [w_n \cdot \Theta], [f(w_1, \ldots, w_n) \cdot \Theta]) \in \mathrm{Def}(S)$. From Theorem 17, it follows that this equals

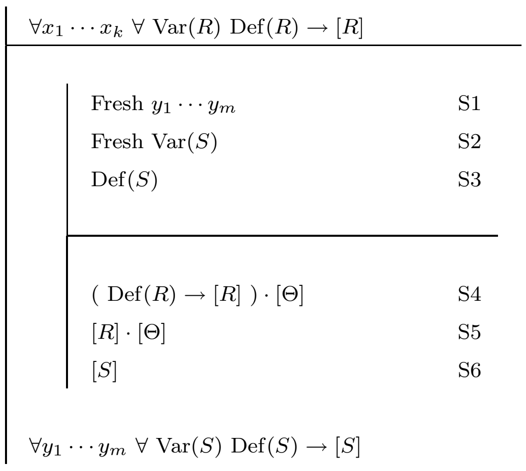
$$R_f(\ [w_1] \cdot [\Theta], \ldots, [w_n] \cdot [\Theta],\ [f(w_1, \ldots, w_n)] \cdot [\Theta]\ ),$$

which in turn equals

$$R_f(\ [w_1], \ldots, [w_n],\ [f(w_1, \ldots, w_n)]\ ) \cdot [\Theta] = A \cdot [\Theta].$$

S6 By Theorem 17, $\ [R] \cdot [\Theta] = [R \cdot \Theta] = [S]$.

Fig. 4. Natural Deduction Proof for the Instantiation Rule


$\forall x_1 \cdots x_k \ \forall \ \mathrm{Var}(R) \ \mathrm{Def}(R) \rightarrow [R]$
Fresh $y_1 \cdots y_m$ S1
Fresh $\mathrm{Var}(S)$ S2
$\mathrm{Def}(S)$ S3
$(\ \mathrm{Def}(R) \rightarrow [R]\ ) \cdot [\Theta]$ S4
$[R] \cdot [\Theta]$ S5
$[S]$ S6
$\forall y_1 \cdots y_m \ \forall \ \mathrm{Var}(S) \ \mathrm{Def}(S) \rightarrow [S]$

3.2 Equality Reflexivity

Assume that the clause $\forall x_1 \cdots x_k \ R$ is obtained from $\forall x_1 \cdots x_k \ t \not\approx t \vee R$ by equality reflexivity. We need to construct a proof of the fact that

$$\forall x_1 \cdots x_k \ \forall \ \mathrm{Var}(t, R) \ \mathrm{Def}(t, R) \rightarrow \ [t \not\approx t \vee R]$$

implies

$$\forall x_1 \cdots x_k \ \forall \ \mathrm{Var}(R) \ \mathrm{Def}(R) \rightarrow [R].$$

There is no difficulty in showing that $[t] \not\approx [t] \vee [R]$ implies $[R]$. The difficulty of the proof is the fact that there may be variables in $\mathrm{Var}(t, R)$, with corresponding definitions in $\mathrm{Def}(t, R)$, that do not occur in $\mathrm{Var}(R)$ (and $\mathrm{Def}(R)$) For these variables, proper instantiations need to be found. In order to find these, the seriality axioms are needed.

Write

$$\mathrm{Var}(t, R)\backslash\mathrm{Var}(R) = \{\alpha_1, \ldots, \alpha_n\}, \text{ with } \ n \geq 0.$$

Assume that the α_i are ordered in such a way that if $\alpha_i = [s_1], \ \ \alpha_j = [s_2]$, and s_1 is a subterm of s_2, then $i \leq j$. Write $A_1(\overline{w}_1, \alpha_1), A_2(\overline{w}_2, \alpha_2), \ldots, A_n(\overline{w}_n, \alpha_n)$ for $\mathrm{Def}(t, R)\backslash\mathrm{Def}(R)$. Due to the way the $\alpha_1, \ldots, \alpha_n$ are ordered, α_j does not occur in $\overline{w}_i$, if $i \leq j$. Using this, we can construct the proof given in Figure 5, in which the $\alpha_1, \ldots, \alpha_n$ are 'resolved away' with the seriality axioms.

Fig. 5. Proof for Equality Reflexivity

> $\forall x_1 \cdots x_k\ \forall\ \mathrm{Var}(t,R)\ \mathrm{Def}(t,R) \rightarrow [t \not\approx t \vee R]$ (assumption)
>
> ---
>
> $\forall x_1 \cdots x_k$
> $\quad \forall \alpha_1\ A_1(\overline{w}_1,\alpha_1)\ \ \forall \alpha_2\ A_2(\overline{w}_2,\alpha_2)\ \ \cdots\ \ \forall \alpha_n\ A_n(\overline{w}_n,\alpha_n)$
> $\quad\quad \forall\ \mathrm{Var}(R)\ \mathrm{Def}(R) \rightarrow [t \not\approx t \vee R]$ (rearranging quantifiers)
>
> > Fresh $x_1 \cdots x_k$
> > Fresh $\mathrm{Var}(R)$
> > $\mathrm{Def}(R)$
> >
> > ---
> >
> > $\forall \alpha_1\ A_1(\overline{w}_1,\alpha_1)\ \ \forall \alpha_2\ A_2(\overline{w}_2,\alpha_2)\ \ \cdots\ \ \forall \alpha_n\ A_n(\overline{w}_n,\alpha_n)$
> > $\quad \forall\ \mathrm{Var}(R)\ \mathrm{Def}(R) \rightarrow [t \not\approx t \vee R]$ (instantiation)
> >
> > $\exists \alpha_1\ A_1(\overline{w}_1,\alpha_1)$ (instantiation of seriality axiom for A_1)
> >
> > > $A_1(\overline{w}_1,\alpha_1)$
> > >
> > > ---
> > >
> > > $\forall \alpha_2\ A_2(\overline{w}_2,\alpha_2)\ \ \cdots\ \ \forall \alpha_n\ A_n(\overline{w}_n,\alpha_n)$
> > > $\quad \forall\ \mathrm{Var}(R)\ \mathrm{Def}(R) \rightarrow [t \not\approx t \vee R]$ (instantiation)
> > >
> > > $\exists \alpha_2\ A_2(\overline{w}_2,\alpha_2)$ (instantiation of seriality axiom for A_2)
> > >
> > > > $A_2(\overline{w}_2,\alpha_2)$
> > > >
> > > > ---
> > > >
> > > > ....
> > > >
> > > > > $\exists \alpha_n\ A_n(\overline{w}_n,\alpha_n)$ (instantiation of seriality axiom for A_n)
> > > > >
> > > > > > $A_n(\overline{w}_n,\alpha_n)$
> > > > > >
> > > > > > ---
> > > > > >
> > > > > > $\forall\ \mathrm{Var}(R)\ \mathrm{Def}(R) \rightarrow [t \not\approx t \vee R]$
> > > > > > $[t \not\approx t \vee R]$
> > > > > > $[R]$
> > > > >
> > > > > $[R]$ ($\exists$-elimination)
> > > >
> > > > ....
> > > >
> > > > $[R]$ ($\exists$-elimination)
> > >
> > > $[R]$ ($\exists$-elimination)
> >
> > $[R]$ ($\exists$-elimination)
>
> $\forall x_1 \cdots x_k\ \ \forall\ \mathrm{Var}(R)\ \ \mathrm{Def}(R) \rightarrow [R]$ ($\forall$-introduction, $\rightarrow$-introduction)

For the other rules, with the exception of paramodulation, it is fairly easy to show that they can be reconstructed.

In the resolution rule, it is possible that a term with an $f \in \mathcal{F}_{\text{Repl}}$ as top symbol occurs in one of the premises, but not in the result. In that case, the definitions for the terms that do not occur in the result need to be resolved away, in the same way as with the equality reflexivity rule. One can either do this directly, or alternatively reformulate the resolution rule as follows:

resolution 2 $$\frac{\forall x_1 \cdots x_k \;\; A \vee R_1 \qquad \forall x_1 \cdots x_k \;\; \neg A \vee R_2}{\forall x_1 \cdots x_k \;\; u_1 \not\approx u_1 \vee \cdots \vee u_n \not\approx u_n \vee R_1 \vee R_2},$$

Here $u_1, \ldots, u_n$ are the subterms that occur in A but not in $R_1 \vee R_2$. After this, $\forall x_1 \cdots x_k \;\; R_1 \vee R_2$ can be obtained through n applications of equality reflexivity.

3.4 Non-Separating Paramodulation

The non-separating paramodulation rule is the rule that is the most complicated to translate:

non-separating paramodulation

$$\frac{\forall x_1 \cdots x_k \;\; t_1 \approx t_2 \vee R_1 \qquad \forall x_1 \cdots x_k \; R_2}{\forall x_1 \cdots x_k \;\; R_1 \vee R_2'}$$

on the condition that $\mathcal{E}(\, R_2',\; R_2 \cdot \Sigma \,)$, with $\mathcal{E} = \{t_1 \approx t_2\}$ and Σ an extension of $\mathcal{E}$.

As is the case with the resolution rule, there can be terms occurring in one of the premises that do not occur in the conclusion. One can proceed in the same way as with the resolution, by keeping the removed terms in negated equations in the conclusion. However, there is no need to keep the negative equations since their removal is trivial. It is sufficient to keep the definitions of the terms that disappeared. The result is the following rule:

$$\forall x_1 \cdots x_k \;\; \forall \operatorname{Var}(t_1, t_2, R_1) \;\; \operatorname{Def}(t_1, t_2, R_1) \rightarrow [t_1 \approx t_2 \vee R_1]$$

and

$$\forall x_1 \cdots x_k \;\; \forall \operatorname{Var}(R_2) \;\; \operatorname{Def}(R_2) \rightarrow [R_2]$$

Fig. 6. Proof for Non-Separating Paramodulation

$\forall x_1 \cdots x_k\ \forall\ \mathrm{Var}(t_1, t_2, R_1)\ \mathrm{Def}(t_1, t_2, R_1) \rightarrow [t_1 \approx t_2 \vee R_1]$ (premise)

$\forall x_1 \cdots x_k\ \forall\ \mathrm{Var}(R_2)\ \mathrm{Def}(R_2) \rightarrow [R_2]$ (premise)

> Fresh $x_1, \ldots, x_k$
>
> Fresh $\mathrm{Var}(t_1, t_2, R_1, R_2, R_2')$
>
> $\mathrm{Def}(t_1, t_2, R_1, R_2, R_2')$
>
> ---
>
> $[t_1] \approx [t_2]\ \vee\ [R_1]$ S1
>
> > $[t_1] \approx [t_2]$
> >
> > ---
> >
> > $\forall\ \mathrm{Var}(R_2)\ \mathrm{Def}(R_2) \rightarrow [R_2]$ S2
> >
> > $(\ \mathrm{Def}(R_2) \rightarrow [R_2]\ ) \cdot [\Sigma]$ S3
> >
> > $[R_2] \cdot [\Sigma]$ S4
> >
> > $[R_2 \cdot \Sigma]$ S5
> >
> > $[R_2']$ S6
> >
> > $[R_1] \vee [R_2']$ S7
>
> > $[R_1]$
> >
> > ---
> >
> > $[R_1] \vee [R_2']$ S8
>
> $[R_1] \vee [R_2']$ S9

$\forall x_1 \cdots x_k\ \forall\ \mathrm{Var}(t_1, t_2, R_1, R_2, R_2')\ \mathrm{Def}(t_1, t_2, R_1, R_2, R_2') \rightarrow [R_1 \vee R_2']$

imply

$$\forall x_1 \cdots x_k\ \ \forall\ \mathrm{Var}(t_1, t_2, R_1, R_2, R_2')\ \mathrm{Def}(t_1, t_2, R_1, R_2, R_2') \rightarrow [R_1 \vee R_2'\ ].$$

Given Σ, one can define $[\Sigma]$ as in Definition 23. The proof is given in Figure 6.

The proof steps can be justified as follows:

S1 Instantiation of the first premise.
S2 Instantiation of the second premise.
S3 Instantiation of S2, using $[\Sigma]$.
S4 We show that the atoms in $\mathrm{Def}(R_2) \cdot [\Sigma]$ are provable. Let $A \in \mathrm{Def}(R_2)$. One can write $A = R_f(\ [w_1], \ldots, [w_n],\ [\ f(w_1, \ldots, w_n)\ ]\ )$, where $f \in \mathcal{F}_{\mathrm{Repl}}$ and $f(w_1, \ldots, w_n)$ occurs in R_2.
By definition of extension, $f(w_1, \ldots, w_n) \cdot \Sigma$ has form $f(v_1, \ldots, v_n)$, with $\mathcal{E}(w_i \cdot \Sigma,\ v_i)$, for $1 \leq i \leq n$. From Theorem 26, it follows that $f(w_1, \ldots, w_n) \cdot \Sigma$

occurs in either $R_2 \cdot \Sigma,\ \ t_1$ or t_2.

If $f(w_1, \ldots, w_n) \cdot \Sigma$ occurs in $R_2 \cdot \Sigma$, but not in R'_2, then one can apply an argument, similar to the last part of the proof of Theorem 26.

((Because $\mathcal{E}(R_2 \cdot \Sigma,\ R'_2)$, it is possible to rewrite $R_2 \cdot \Sigma$ into R'_2, using the equality $t_1 \approx t_2$. In the rewrite sequence, there is a last term that still contains $f(v_1, \ldots, v_n)$. Because replacing in or at a v_i is not allowed, and rewriting by $t_1 \approx t_2$ removes $f(v_1, \ldots, v_n)$, either t_1 or t_2 must contain $f(v_1, \ldots, v_n) = f(w_1, \ldots, w_n) \cdot \Sigma$))

Because $f(v_1, \ldots, v_n)$ occurs in t_1, t_2 or R'_2, it must be the case that

$$R_f(\ [v_1], \ldots, [v_n],\ [\ f(v_1, \ldots, v_n)\ ]\ ) \in \mathrm{Def}(t_1, t_2, R_1, R_2, R'_2). \tag{1}$$

Since $[\Sigma]$ is a substitution, $R_f(\ [w_1], \ldots, [w_n],\ [\ f(w_1, \ldots, w_n)\ ]\ ) \cdot [\Sigma]$ equals

$$R_f(\ [w_1] \cdot [\Sigma], \ldots, [w_n] \cdot [\Sigma],\ [f(w_1, \ldots, w_n)] \cdot [\Sigma]\ ),$$

which, by Theorem 24, equals

$$R_f(\ [w_1 \cdot \Sigma], \ldots, [w_n \cdot \Sigma],\ [\ f(w_1, \ldots, w_n) \cdot \Sigma\ ]\ ). \tag{2}$$

Since for each i, (with$1 \leq i \leq n$), $\mathcal{E}(w_i \cdot \Sigma,\ v_i)$, it follows from Theorem 25, that $[t_1] \approx [t_2] \vdash [w_i \cdot \Sigma] \approx [v_i]$. Then it follows from (1) that (2) is provable.

S5 Follows from Theorem 24.
S6 Follows from Theorem 25, because $\mathcal{E}(R'_2,\ R_2 \cdot \Sigma)$.
S7 $\vee$-introduction.
S8 $\vee$-introduction.
S9 $\vee$-elimination.

4 Translation of the CNF-transformation

In the previous section we have shown that it is possible to replace function symbols by arbitrary, serial relations in resolution proofs on the clause level. This is possible on the condition that the paramodulation steps in the proof are non-separating.

In the present section, we show that the CNF-transformation can be modified in such a way that it is completely first-order, and constructs clauses in which the Skolem functions are replaced by serial relations. The modified clause transformation is very general. It supports the standard optimizations mentioned in [3,2,17,9].

In this way, it becomes possible to run a resolution and paramodulation theorem prover as usual, and construct a proof from its output, which is completely first-order, and polymial. (Under reasonable assumptions on what the theorem prover should output)

We will consider CNF transformations with the following general pattern: First, new names are introduced for subformulas that would cause blow-up. After that, the resulting formula is transformed into negation normal form. Then antiprenexing is applied. The resulting formula is Skolemized, and factored into clauses.

The modified translation will proceed as the standard transformation until Skolemization. Then, instead of Skolem functions, serial relations will be introduced. The resulting formula can be factored into clauses as usual. Although the intuition behind the relation-introduction is straightforward, the actual transformation is technically involved, due to technical difficulties that we will explain shortly. We first explain the basic idea of relation introduction, and after that the source of the technical difficulties.

Consider the formula $\forall x\, p(x) \rightarrow \exists y\, q(x,y)$. Its Skolemization equals $\forall x\, p(x) \rightarrow q(x, f(x))$, which results in the clause $\forall x\, \neg p(x) \vee q(x, f(x))$. Instead of Skolemizing, one can introduce the relation
$F(x,y) := (\exists z\, q(x,z)) \rightarrow q(x,y)$ and obtain the formula $\forall x\, p(x) \rightarrow \forall y\, F(x,y) \rightarrow q(x,y)$. This formula can be written as $\forall x \forall \alpha\; F(x,\alpha) \rightarrow \neg p(x) \vee q(x,\alpha)$. Relation F can be easily proven serial, because $\forall x \exists y\; (\exists z\; q(x,z)) \rightarrow q(x,y)$ is a tautology. In addition, the formula $\forall x\; p(x) \rightarrow \forall y\; ((\exists z\; q(x,z)) \rightarrow q(x,y)) \rightarrow q(x,y)$ is easily provable from $\forall x\; p(x) \rightarrow \exists y\; q(x,y)$.

Once the Skolemized formula has been replaced by its relational counterpart, the CNF-transformation can proceed in the same way as on the Skolemized formulas. However, there is a technical difficulty which is caused by the fact that standard outermost Skolemization cannot be iterated, when relations are introduced. In order to obtain Skolem terms that are as small as possible, Skolemization is nearly always done from outside to inside, because otherwise one would obtain nested Skolem terms.

In a standard Skolemization step, an existentially quantified variable is replaced by a functional term, and it disappears from the formula. If instead we introduce a relation, then the existentially quantified variable does not disappear, but is replaced by a universal quantifier. Later Skolemization steps will depend on these universal quantifiers, and introduce unwanted dependencies. The following example shows the problem:

Example 34 *Outermost Skolemization of* $\forall x\; \exists y_1\; \exists y_2\; p(x, y_1, y_2)$ *results in* $\forall x\; \exists y_2\;\; p(x, f_1(x), y_2)$. *Skolemizing one more time results in* $\forall x\; p(x, f_1(x), f_2(x))$. *It appears that there exist no binary relations* F_1, F_2 *for which the formulas*

$\forall x\ \forall y_1\ \forall y_2\ F_1(x, y_1) \rightarrow F_2(x, y_2) \rightarrow p(x, y_1, y_2)$,

$\forall x\ \exists y_1\ F_1(x, y_1)$,

$\forall x\ \exists y_2\ F_2(x, y_2)$,

are provable. The problem is due to the fact that in the original formula, the y_2 *can only be chosen with knowledge of* y_1*. The same problem appears in the formula* $\forall x_1\ \exists y_1\ \forall x_2\ \exists y_2\ p(x_1, y_1, x_2, y_2)$*. Outermost Skolemization results in* $\forall x_1\ \forall x_2\ p(x_1, f_1(x_1), x_2, f_2(x_1, x_2)\ )$.

Again, there seems to be no way of finding relations that are serial and for which $\forall x_1\ \forall y_1\ \forall x_2\ \forall y_2\ F_1(x_1, y_1) \rightarrow F_2(x_1, x_2, y_2) \rightarrow p(x_1, y_1, x_2, y_2)$ *is provable.*

We solve the problem by using innermost Skolemization instead of outermost Skolemization. Innermost Skolemization was considered in [16] and proven sound there. Innermost Skolemization works in the same way as standard Skolemization, but it starts with an innermost existential quantifier, instead of an outermost existential quantifier.

Example 35 *On the first formula of the previous example, one step of innermost Skolemization results in* $\forall x\ \exists y_1\ p(x, y_1, f_2(x, y_1))$*. One more step produces* $\forall x\ p(x, f_1(x), f_2(x, f_1(x_1))\ )$.

Similarly, innermost Skolemization iterated on the second formula produces the formula $\forall x_1\ \forall x_2\ p(x_1, f_1(x_1), x_2, f_2(x_1, f_1(x_1), x_2)\ )$.

Innermost Skolemization is not suitable for proof search because it results in bigger Skolem terms. However, the length of a non-separating resolution proof does not increase if one uses innermost Skolemization instead of outermost Skolemization. This is due to the fact that, although Skolem terms obtained from innermost Skolemization are bigger, they do not depend on more variables. Whenever in a proof two (outermost) Skolem terms are equal, their innermost counterparts are also equal. As a consequence, all proof steps will remain valid when outermost Skolem terms are replaced by innermost Skolem terms.

Using this, the proof reconstruction strategy will be as follows: Pass the clauses obtained by outermost Skolemization to the theorem prover. If the prover finds a refutation, then one can convert it into a refutation of the clauses obtained by innermost Skolemization, which does not increase the number of proof steps. The clauses obtained by innermost Skolemization and with relations instead of functions, have a first-order proof from the original formula. Therefore, the result will be a first-order refutation of the original first-order formula.

There is one remaining problem, which is caused by the fact that non-separating paramodulation steps on Skolem terms obtained from outermost Skolemization are not necessarily non-separating paramodulation steps on the corresponding Skolem terms that one obtains from innermost Skolemization. We will discuss this in Section 4.1.

Definition 36 *A formula F is in* negation normal form *if it does not contain $\leftrightarrow$ and $\rightarrow$, and every occurrence of $\neg$ is applied to an atom.*

In the rest of this paper, we assume that all first-order formulas F are *standardized apart*, i.e. no variable is bound twice in F. This can be easily obtained by renaming.

Definition 37 *Let F be a formula in negation normal form. If F contains an existentially quantified subformula, then F can be written as $F[\ \exists y\ G\ ]$. Let $x_1, \ldots, x_k$ be the free variables of $\exists y\ G$ that are bound by a quantifier in F. Let f be a new function symbol, s.t.* $\mathrm{ar}(f) = k$*. Then $F[\ G \cdot \{y := f(x_1, \ldots, x_k)\}\ ]$ is a* one-step Skolemization of $F[\ \exists y\ G\ ]$.

Let $F = F_1, \ldots, F_m$ be a Skolemization sequence, i.e.

- *each F_{i+1} is a one-step Skolemization of F_i, which Skolemizes some subformula $\exists y_i\ G_i$.*
- *F_m has no remaining existential quantifiers.*

F_m is an outermost Skolemization *of F, if each $\exists y_i\ G_i$ is not in the scope of another existential quantifier in F_i.*

F_m is an innnermost Skolemization *of F, if no G_i contains another existential quantifier.*

We prove the claim that instead of Skolem functions, serial relations can be obtained:

Theorem 38 *Let F be a formula in negation normal form containing an existential quantifier. Write F as $F[\ \exists y\ A(x_1, \ldots, x_k, y)\ ]$, where $x_1, \ldots, x_k$ are the quantified variables that are bound in F and free in $\exists y\ A(x_1, \ldots, x_k)$. Let $F[\ A(x_1, \ldots, x_k, g(x_1, \ldots, x_k))\ ]$ be obtained by one-step Skolemization*

There is a $(k+1)$-place relation G, for which the following formulas are provable:

SER $\forall x_1 \cdots x_k\ \exists y\ G(x_1, \ldots, x_k, y)$,
SKOL $F[\ \forall y\ G(x_1, \ldots, x_k, y) \rightarrow A(x_1, \ldots, x_k, y)\ ]$.

PROOF. Take $G(x_1, \ldots, x_k, y) := (\ \exists z\ A(x_1, \ldots, x_k, z)\ ) \rightarrow A(x_1, \ldots, x_k, y)$.

Then SER becomes

$$\forall x_1 \cdots x_k \; \exists y \; (\; \exists z \; A(x_1, \ldots, x_k, z) \;) \rightarrow A(x_1, \ldots, x_k, y),$$

which is a simple tautology.

We show that SKOL is logically equivalent to F. Expanding G in SKOL yields:

$$F[\; \forall y \; (\; (\exists z \; A(x_1, \ldots, x_k, z) \;) \rightarrow A(x_1, \ldots, x_k, y) \;) \rightarrow A(x_1, \ldots, x_k, y) \;].$$

This is logically equivalent to

$$F[\; \forall y \; (\; \exists z \; A(x_1, \ldots, x_k, z) \;) \wedge \neg A(x_1, \ldots, x_k, y) \vee A(x_1, \ldots, x_k, y) \;],$$

which in turn is equivalent to

$$F[\; \forall y \; (\; \exists z \; A(x_1, \ldots, x_k, z) \;) \vee A(x_1, \ldots, x_k, y) \;].$$

This final formula is equivalent to

$$F[\; \exists z \; A(x_1, \ldots, x_k, z) \;].$$

If one replaces an innermost Skolemization sequence by an innermost relation-introduction sequence, then one obtains a formula that has the same structure as the Skolemized formula, but with the Skolem terms replaced by variables based on some function replacement [ ]. In addition, it contains definitions of form $G(x_1, \ldots, x_k, y) \rightarrow A$ that introduce the variables that are used in the translations of the Skolem functions. The fact that in the orginal formula y was in the scope of an existential quantifier $\exists y$, ensures that in the relational translation, y is in the scope of a definition $\forall y \; G(x_1, \ldots, x_k, y) \rightarrow A$. The following definition specifies more precisely the relation between the innermost Skolemization and the innermost relation-introduction.

Definition 39 *Let [ ] be a function replacement, which replaces functions from $\mathcal{F}_{\mathrm{Repl}}$ and introduces variables from $\mathcal{F}_{\mathrm{Def}}$. Let F be a first-order formula that is standardized apart and which is in negation normal form. We define the one-step replacement as follows: Find a maximal (not contained in any other such terms) term $f(t_1, \ldots, t_n)$ that occurs in F and which has $f \in \mathcal{F}_{\mathrm{Repl}}$. Select a subformula G of F that contains all occurrences of $f(t_1, \ldots, t_n)$, but which is still in the scope of all quantifiers that bind the variables in $t_1, \ldots, t_n$. Let $\alpha = [f(t_1, \ldots, t_n)]$. Let G' be obtained by replacing $f(t_1, \ldots, t_n)$ by α everywhere in G. Finally replace $F[G]$ by $F[\; \forall \alpha \; R_f(\; t_1, \ldots, t_n, \alpha \;) \rightarrow G' \;]$.*

For a first-order formula F, the replacement $[F]$ is defined as the formula that one obtains by making one-step replacements, until no further replacements are possible.

Strictly seen, the function $[F]$ is not a function, because there is some freedom in where the definitions are inserted. However, it is easily checked that different choices result in equivalent formulas.

Example 40 *Assume that $[f(x)] = \alpha$, $[f(f(x))] = \beta$, $[f] = F$. Then $\forall x\ p(f(f(x))$ is in one step replaced by $\forall x \forall \beta\ F(f(x), \beta) \rightarrow p(\beta)$. One more step results in $\forall x\ \forall \beta \alpha\ F(x, \alpha) \wedge F(\alpha, \beta) \rightarrow p(\beta)$.*

Theorem 41 *Let F_1 be obtained from F by innermost Skolemization. Let F_2 be obtained from F by making the corresponding relation replacements. Then there is a function replacement $[\ ]$, s.t. F_2 is a $[\ ]$-translation of F_1.*

Example 42 *We demonstrate relation introduction on the formula $\forall x_1\ \exists y_1\ \forall x_2\ \exists y_2\ \ p(x_1, y_1, x_2, y_2)$. One step of innermost Skolemization results in $\forall x_1\ \exists y_1\ \forall x_2\ \ p(x_1, y_1, x_2, f_2(x_1, y_1, x_2))$, and one more step of innermost Skolemization yields $\forall x_1\ \forall x_2\ p(x_1, f_1(x_1), x_2, f_2(x_1, f_1(x_1), x_2))$. First put $R_{f_2}(x_1, y_1, x_2, y_2) := (\exists z\ \ p(x_1, y_1, x_2, z)\ ) \rightarrow p(x_1, y_1, x_2, y_2)$. Then one can prove*

$$\forall x_1\ \exists y_1\ \forall x_2\ \forall y_2\ R_{f_2}(x_1, y_1, x_2, y_2) \rightarrow p(x_1, y_1, x_2, y_2).$$

After that, put

$$R_{f_1}(x_1, y_1) := (\ \exists z\ (\forall x_2\ \forall y_2\ R_{f_2}(x_1, z, x_2, y_2) \rightarrow p(x_1, z, x_2, y_2)\ ))\rightarrow$$

$$(\ \forall x_2\ \forall y_2\ R_{f_2}(x_1, y_1, x_2, y_2) \rightarrow p(x_1, y_1, x_2, y_2)\ ).$$

Then $\forall x_1\ \forall y_1\ R_{f_1}(x_1, y_1) \rightarrow \forall x_2\ \forall y_2\ R_{f_2}(x_1, y_1, x_2, y_2) \rightarrow p(x_1, y_1, x_2, y_2)$ is provable, together with the seriality axioms for R_{f_1} and R_{f_2}.

4.1 Replacing Innermost Skolemization by Outermost Skolemization

We now justify the claim made in the previous section that innermost Skolemization does not increase the proof length. Before we do this, we give an example that makes clear that one has to use a form of paramodulation that is more restricted than non-separating paramodulation.

Example 43 *Consider the formula $\forall x\ \exists y_1\ y_2\ p(x, y_1, y_2)$ and the two clauses*

$C_1 \qquad p(0, f_1(0), f_2(0)),$

$C_2 \qquad p(0, f_1(0), f_2(0, f_1(0))).$

The clause C_1 is obtained by instantiating the outermost Skolemization of F by 0. The clause C_2 is obtained by making the same instantiation in the innermost Skolemization. From C_1, it is possible to derive $p(\ 0, f_1(1), f_2(0)\ )$ by non-separating paramodulation from equation $0 \approx 1$. However, it is not possible to derive $p(\ 0, f_1(1), f_2(0, f_1(0)\ )$ from C_2 by non-separating paramodulation.

The example shows that non-separating paramodulation steps on clauses with innermost Skolem terms can in general not be translated into non-separating paramodulation steps on the corresponding clauses with outermost Skolem terms. Therefore, the following more restricted version of paramodulation is needed.

Definition 44 *Let [ ] be a function replacement replacing functions from $\mathcal{F}_{\mathrm{Repl}}$. The $\mathcal{F}_{\mathrm{Repl}}$-*simultaneous paramodulation rule *is the following rule:*

$$\frac{\forall x_1 \cdots x_k \quad t_1 \approx t_2 \vee R_1 \qquad \forall x_1 \cdots x_k \quad R_2}{\forall x_1 \cdots x_k \quad R_1 \vee R_2'}$$

where R_2' is obtained from R_2 by replacing arbitrary occurrences of t_1 by t_2. If at least one occurrence of t_1 in the scope of a function symbol from $\mathcal{F}_{\mathrm{Repl}}$ is replaced, then all occurrences of t_1 that are in the scope of a function symbol from $\mathcal{F}_{\mathrm{Repl}}$ have to be replaced.

Example 45 *The paramodulation step from C_1 in Example 43 was not $\mathcal{F}_{\mathrm{Repl}}$-simultaneous, because $\mathcal{F}_{\mathrm{Repl}} = \{f_1, f_2\}$. The clauses that can be obtained by $\mathcal{F}_{\mathrm{Repl}}$-simultaneous paramodulation from $p(\ 0, f_1(0), f_2(0)\ )$ are $p(\ 1, f_1(0), f_2(0)\ )$, $p(\ 0, f_1(1), f_2(1)\ )$, $p(\ 1, f_1(1), f_2(1)\ )$.*

Using $\mathcal{F}$-simultaneous paramodulation, we can state the main result of this paper:

Theorem 46 *Given a first-order formula F, and a resolution refutation for which*

- *the CNF-transformation uses subformula replacement, antiprenexing and outermost Skolemization,*
- *the resolution refutation on the clause level uses the standard resolution rules but $\mathcal{F}_{\mathrm{Repl}}$-simultaneous paramodulation, where $\mathcal{F}_{\mathrm{Repl}}$ are the Skolem functions introduced in the CNF-transformation,*

then one can effectively obtain a purely first-order refutation of F that is polynomial in the size of the CNF-transformation plus the size of the resolution *refutation.*

The proof will follow in the rest of this section. Observe that, in case one does not make any replacements at all inside Skolem functions, which is known to be complete because of the results in [5], one automatically has $\mathcal{F}_{\text{Skol}}$-simultaneous paramodulation.

In that case, one also gets *the splitting rule* for free. The splitting rule is the following rule, used on the clause level by resolution theorem provers: If the prover derives a clause $\forall x_1 \cdots x_k \ (R_1 \vee R_2)$, s.t. no x_i occurs in both R_1 and R_2, then $\forall x_1 \cdots x_k \ (R_1 \vee R_2)$ implies $(\forall x_1 \cdots x_k \ R_1) \vee (\forall x_1 \cdots x_k \ R_2)$. These two clauses can be refuted seperately by backtracking.

In general, if some clause $\forall x_1 \cdots x_k \ R_1 \vee R_2$ is splittable, its translation $\forall x_1 \cdots x_k \ \forall \ \text{Var}(R_1, R_2) \ \text{Def}(R_1, R_2) \rightarrow [R_1] \vee [R_2]$ need not be splittable, consider for example the ground clause $p(f(0)) \vee q(f(0))$ with translation $\forall \alpha \ F(0, \alpha) \rightarrow p(\alpha) \vee q(\alpha)$. However, if one knows that one never paramodulates inside Skolem functions, one can 'glue' $q(f(0))$ to the clauses in the refutation of $p(f(0))$, without ever being forced to modify $p(f(0))$. (and we have to admit that the remark about general splitting in the conclusion of [12] was incorrect)

We now continue by showing that the difference between innermost and outermost Skolemization is smaller than it appears to be. Skolem terms obtained from innermost Skolemization have exactly the same variables as Skolem terms obtained from outermost Skolemization.

Theorem 47 *Let F be a formula. Let F_1 be its outermost Skolemization. Let F_2 be its innermost Skolemization. Then F_1 and F_2 have the same logical structure (this means that they only differ inside some atoms), and each Skolem term in F_1 depends on exactly the same variables as its counterpart in F_2.*

PROOF. It is easy to see that F_1 and F_2 have the same logical structure, because Skolemization does not change the logical structure, except for the elimination of existential quantifiers. We will show that the Skolem terms in F_1 and F_2 depend on the same set of variables in the following sequence of definitions and lemmas.

Definition 48 *Let F be a first-order formula. Let x and y be two variables that are quantified inside F. Let A be the subformula of F that is quantified by x. We say that y* eventually depends on *x if there exists a sequence $\exists y_1 \ B_1, \ldots, \exists y_n \ B_n$ of subformulas of F, s.t.*

(1) $\exists y_1 \ B_1$, is inside A and x is free in $\exists y_1 \ B_1$,
(2) each $\exists y_{i+1} \ B_{i+1}$ is inside B_i and y_i is free in $\exists y_{i+1} \ B_{i+1}$, and

(3) $y_n = y$.

Similarly, we say that a term t eventually depends on *x in F if there exists a sequence $\exists y_1\ B_1, \ldots, \exists y_n\ B_n$ of subformulas of F, s.t.*

(1) $\exists y_1\ B_1$, *is inside A and x is free in* $\exists y_1\ B_1$,
(2) *each* $\exists y_{i+1}\ B_{i+1}$ *is inside B_i and y_i is free in* $\exists y_{i+1}\ B_{i+1}$, *and*
(3) y_n *occurs in t.*

Lemma 49 *Let $F_1, \ldots, F_m$ be a Skolemization sequence in which each F_{i+1} is obtained from F_i by Skolemization at an arbitrary position.*

Let x be a universally quantified variable, occurring in F_1. Let y be a universally quantified variable, occurring in F_1. Let t be its Skolem term in F_m. Then x occurs in t iff y eventually depends on x in F_1.

PROOF. It follows, by n times applying Lemma 50, that y eventually depends on x in F_1 iff its Skolem term t eventually depends on x in F_2. Because F_2 does not contain existential quantifiers, this is case iff x occurs in t.

Lemma 50 *Let F' be obtained from F by one-step Skolemization. Let x be a quantified variable, which occurs in both F and F'. Let A be the subformula quantified by x in F. Let A' be the subformula quantified by x in F'. Then:*

(1) *Some term t in F eventually depends on x iff its counterpart t' in F' eventually depends on x.*
(2) *Let y be an existentially quantified variable in F, which is not Skolemized in F'. Then y eventually depends on x in F iff y eventually depends on x in F'.*
(3) *Let y be the existentially quantified variable in F which is Skolemized in F'. Then y eventually depends on x in F iff the Skolem term for y eventually depends on x in F'.*

PROOF. In order to prove the 3 properties from left to right, it is sufficient to observe the following two points:

(1) Let z_1 be a quantified variable occurring in F. Let G_1 be the subformula of F that is quantified by z_1. Let $\exists z_2\ G_2$ and $\exists z_3\ G_3$ be subformulas of F, s.t.
- $\exists z_2\ G_2$ is a subformula of G_1 and z_1 is free in G_2,
- $\exists z_3\ G_3$ is a subformula of G_2 and z_2 is free in G_3.

If F' is obtained from F' by Skolemizing z_2, then $\exists z_3\ G'_3$ is a subformula of G'_1, and z_1 is free in $\exists z_3\ G'_3$.

(2) Let z_1 be a quantified variable occurring in F. Let G_1 be the subformula of F that is quantified by z_1. Let $\exists z_2\ G_2$ be a subformula of F, let t be a term occurring in G_2, s.t.
   - $\exists z_2\ G_2$ is a subformula of G_1 and z_1 is free in $\exists z_2\ G_2$,
   - the variable z_2 occurs in t.

   If F' is obtained from F' by Skolemizing z_2, then z_1 occurs in t'.

Using this, the 3 properties can be easily proven from left to right.

Next we prove the 3 properties from right to left. In order to prove these, we need the following 2 properties, which are essentially the converses of the properties above.

(1) Let z_1 be a quantified variable occurring in F. Let G_1 be the subformula of F that is quantified by z_1. Let $\exists z_3\ G_3$ be a subformula of G_1. Assume that both z_1 and z_3 are not Skolemized in F'. Then G_1' is a subformula of F' and still quantified by z_1. Also $\exists z_3\ G_3'$ remains a subformula of G_1'. Assume that z_1 is free in $\exists z_3\ G_3'$. Then either
   - z_1 is free in $\exists z_3\ G_3$, or
   - there is an existentially quantified subformula $\exists z_2\ G_2$ of F, s.t.
     · $\exists z_2\ G_2$ is a subformula of G_1 and z_1 is free in $\exists z_2\ G_2$,
     · $\exists z_3\ G_3$ is a subformula of G_2 and z_2 is free in G_3.

   Assume that the first case does not hold. Then z_1 occurs in the Skolem term introduced in F'. Let $\exists z_2\ G_2$ be the corresponding subformula of F. Because z_1 occurs in the Skolem term, it must be the case that z_1 is free in $\exists z_2\ G_2$. As a consequence, $\exists z_2\ G_2$ is a subformula of G_1. Because the Skolem term occurs in G_3', it must be the case that G_2 overlaps with G_3. Then either $\exists z_2\ G_2$ occurs in G_3, or $\exists z_3\ G_3$ occurs in G_2. The first possibility cannot happen, because in that case z_1 would also occur in G_3.

(2) Let z_1 be a quantified variable occurring in F. Let G_1 be the subformula of F that is quantified by z_1. Let t be a term occurring in G_1. Assume that z_1 is not Skolemized in F'. Assume that z_1 occurs in t'. Then either
   - z_1 occurs in t, or
   - there exists an existentially quantified subformula $\exists z_2\ G_2$ of F, s.t.
     · $\exists z_2\ G_2$ is a subformula of G_1 and z_1 is free in $\exists z_2\ G_2$,
     · z_2 is free in t.

We now have shown that terms obtained from innermost Skolemization do not contain more variables than terms obtained from outermost Skolemization. In order to prove that resolution refutations from sets of clauses with innermost Skolem terms can be translated into resolution refutations from sets of clauses with outermost Skolem terms, we also need to look into their structure:

Definition 51 *We recursively define the set of* Skolem-type *terms.*

- *A variable x is also a Skolem-type term.*
- *If $t_1, \ldots, t_n$ are variables or Skolem-type terms, f is a Skolem function, then $f(t_1, \ldots, t_n)$ Skolem-type term.*

Given a Skolem-type term $f(t_1, \ldots, t_n)$, we call the positions of the t_i that contain other Skolem-type terms internal positions. *The positions that contain the variables are called* external. *An outer-inner transformation Θ is a function that assigns to Skolem terms of form $f(x_1, \ldots, x_n)$ Skolem-type terms. Θ must satisfy the following conditions:*

(1) $\Theta(\ f(x_1, \ldots, x_n)\ )$ is a Skolem-type term containg exactly variables $x_1, \ldots, x_n$ and some additional Skolem functions. In particular, there are no non-Skolem functions in $\Theta(\ f(x_1, \ldots, x_n)\ )$.
(2) If some Skolem function g occurs in $\Theta(\ f(x_1, \ldots, x_n)\ )$, and its i-th argument is an internal position, then its i-th argument is an internal position in every $\Theta(\ f'(x_1, \ldots, x_m)\ )$ in which g occurs.

Outer-inner transformation Θ is extended to terms, literals and clauses as expected, by recursion.

Theorem 52 *Let $\mathcal{F}_{\text{Repl}}$ be the set of Skolem functions. Let Θ be an outer-to-inner transformation. Let $C_1, \ldots, C_n$ be some set of clauses. If $C_1, \ldots, C_n$ have a resolution refutation using $\mathcal{F}$-non-separating paramodulation, then $\Theta(C_1), \ldots, \Theta(C_n)$ have a resolution refutation using $\mathcal{F}$-simultaneous paramodulation.*

PROOF. It is not hard to see that all for all (unrestricted) resolution steps holds: If D can be obtained from $D_1, \ldots, D_k$, (with $k = 1$ or $k = 2$), then $\Theta(D)$ can be obtained from $\Theta(D_1), \ldots, \Theta(D_k)$. In addition one has to show that if D is obtained from D_1 and D_2 by $\mathcal{F}_{\text{Repl}}$-simultaneous paramodulation, then $\Theta(D)$ can be obtained from $\Theta(D_1)$ and $\Theta(D_2)$ by $\mathcal{F}$-simultaneous paramodulation.

Write

$$D_1 = \forall x_1 \cdots x_k\ t_1 \approx t_2 \vee R_1, \qquad D_2 = \forall x_1 \cdots x_k\ R_2,$$

$$D = \forall x_1 \cdots x_k\ R_1 \vee R_2',$$

where R_2' is obtained from R_2 by $\mathcal{F}_{\text{Repl}}$-simultaneous paramodulation. Then

$$\Theta(D_1) = \forall x_1 \cdots x_k\ \Theta(t_1) \approx \Theta(t_2) \vee \Theta(R_1), \quad \Theta(D_2) = \forall x_1 \cdots x_k\ \Theta(R_2),$$

and

$$\Theta(D) = \forall x_1 \cdots x_k \ \Theta(R_1) \vee \Theta(R_2').$$

Let $\mathcal{E} = \{ \ \Sigma(t_1) \approx \Sigma(t_2) \ \}$. We need to show that there exists an extension Σ of $\mathcal{E}$, s.t. $\mathcal{E}(\ \Theta(R_2) \cdot \Sigma, \ \Theta(R_2') \ )$.

If t_1 is not replaced inside Skolem terms in R_2, then one can define Σ from $f(w_1, \ldots, w_n) \cdot \Sigma = f(w_1, \ldots, w_n)$, for all Skolem terms $f(w_1, \ldots, w_n)$.

Otherwise, Θ is defined as follows: For each $f \in \mathcal{F}_{\mathrm{Repl}}$, put $f(w_1, \ldots, w_n) \cdot \Sigma = f(v_1, \ldots, v_n)$, where $w_i = v_i$ if w_i is on an internal position of f. Otherwise v_i is obtained from w_i by replacing all occurrences of $\Sigma(t_1)$ that are not in the scope of a function from $\mathcal{F}_{\mathrm{Repl}}$ by $\Sigma(t_2)$.

4.2 The Complete Transformation

We can now describe the complete proof transformation. The CNF-transformation usually starts with *subformula replacement*, in order to avoid exponential blowup later in the transformation, see [17,2,9,11]. For example, the following formula results in 2^p clauses, when naively factored into clausal normal form: $(a_1 \wedge b_1) \vee \cdots \vee (a_p \wedge b_p)$.

Definition 53 *Let F be a first-order formula. A* formula definition (relative to F) *is a formula of form*

$$\forall x_1 \cdots x_k \ A(x_1, \ldots, x_k) \leftrightarrow R(x_1, \ldots, x_k),$$

in which A is a k-ary relational symbol which does not occur in F, and R is a k-ary relation. (in the sense of Definition 5)

The following is standard:

Theorem 54 *Suppose $\forall x_1 \cdots x_k \ A(x_1, \ldots, x_k) \leftrightarrow R(x_1, \ldots, x_k)$ is a formula definition, relative to some formula F, and that there exists a proof Π of*

$$F \wedge \ \ \forall x_1 \cdots x_k \ A(x_1, \ldots, x_k) \leftrightarrow R(x_1, \ldots, x_k) \vdash \bot,$$

then there exists a proof of $F \vdash \bot$.

PROOF. First substitute $\Pi' := \Pi[A := R]$. The result Π' is a proof of

$$F \wedge \ \ \forall x_1 \cdots x_k \ R(x_1, \ldots, x_k) \leftrightarrow R(x_1, \ldots, x_k) \vdash \bot.$$

Because $\forall x_1 \cdots x_k \; R(x_1, \ldots, x_k) \leftrightarrow R(x_1, \ldots, x_k)$ is a tautology, one can obtain a proof Π'' of $F \vdash \bot$.

In the example before Definition 53, one can replace the formula by $c_1 \leftrightarrow (a_1 \wedge b_1), \ldots, c_p \leftrightarrow (a_p \wedge b_p),\; c_1 \vee \cdots \vee c_p$, which can be factored into a clausal normal form of size $3p + 1$.

Theorem 54 makes it possible to remove the definitions which were introduced in the clause transformation. Whether or not it is a good idea to do this, has to be empirically determined. In our view, definitions are much less ugly than choice functions, and it is probably acceptable to keep them in most cases.

After replacement of subformulas, the formula is transformed into negation normal form. At this point, usually *antiprenexing* (also called *miniscoping*) is attempted, see [2]. Antiprenexing tries to reduce the scope of quantifiers using transformations of form $(\forall x \; P(x) \wedge Q) \Rightarrow (\forall x \; P(x)) \wedge Q$, in case x is not free in Q. Such transformations are sound, provably correct in first-order logic, and they sometimes reduce the dependencies between quantifiers, which may result in smaller Skolem terms. As an example, one can consider the formula $\forall x \; (\; P(x) \rightarrow \exists y \; Q(y) \;)$. Since antiprenexing takes place before Skolemization, it is not affected by our proof transformation, and it can be carried out as usual.

After Skolemization, the resulting formula can be factored into clauses. We first describe the usual procedure, after that we describe how it is modified in case relations are used instead of Skolem functions.

Definition 55 *Let F be a first-order formula in negation normal form, which does not contain existential quantifiers. We define* the set of clauses obtained from F *as the set of formulas that can be obtained by applying the following rules:*

(1) Replace a conjunction $A \wedge B$ by one of A or B.
(2) Move universal quantifiers forward, using rules

$$(\; \forall x \; A \;) \vee B \Rightarrow \forall x \; (A \vee B), \; \textit{and} \; A \vee (\; \forall x \; B \;) \Rightarrow \forall x \; (A \vee B).$$

(3) Replace universal quantifiers $\forall x \; A$, for which x is not free in A, by A.

The different clauses are obtained by making different choices in Step 1.

Definition 56 *Let F be a first-order formula in negation normal form, which does not contain existential quantifiers. Let [ ] be a function replacement. For relations R introduced by [ ], we add the following rules to Definition 55:*

2a $\quad (\; \forall \alpha \; R(t_1, \ldots, t_n, \alpha \;) \rightarrow A \;) \vee B \Rightarrow \forall \alpha \; R(t_1, \ldots, t_n, \alpha) \rightarrow (A \vee B),$

and

$$A \vee (\ \forall \alpha\ R(t_1, \ldots, t_n, \alpha) \rightarrow B\ ) \Rightarrow \forall \alpha\ R(t_1, \ldots, t_n, \alpha) \rightarrow (A \vee B).$$

3a *Replace quantifications $\forall \alpha\ R(t_1, \ldots, t_n, \alpha) \rightarrow A$, for which α is not free in A, by A. (In order to do this, one needs the seriality axiom for R)*

Theorem 57 *Let F be a first-order formula in negation normal form that is standardized apart, and which does not contain any existential quantifiers. Let $[\ ]$ be a function replacement, and let F' be a translation of F. (See Definition 39) Then, for every clause $\forall x_1 \cdots x_k\ S$ that can be obtained from F using the factoring procedure of Definition 55 there exists a clause of form $\forall x_1 \cdots x_k\ \forall\ Var(S) \rightarrow \mathrm{Def}(S) \rightarrow [S]$, which can be obtained from F' by the factoring procedure of Definition 56.*

4.3 Readability of Proofs

In our view, the proofs constructed are reasonably readable, as long as one replaces the Skolem functions by fresh relation symbols, and does not further expand them. One would have to add the definitions of the relation symbols, similar to the structural clause transformation. For example, the proof in Figure 3 is quite readable, as long as one keeps the F-symbol. In order to ensure that the proof remains correct, one needs to add the definition $\forall xy\ F(x, y) \leftrightarrow p(x, y)$.

If one would substitute $F(x, y)$ away, the resulting proof would already be less readable. In case the relations have more complicated definitions, reading the proof with expanded definitions is hopeless. For example, if the definition of $F(x, y)$ would equal the definition of R_{f_1} in Example 42, there would be no chance of reading the expanded proof.

5 Conclusions and Future Work

We gave a method for translating resolution proofs that include the CNF-transformation into purely first-order proofs. The method is efficient and structure preserving. On the clause level, the resolution prover can make use of all of the standard resolution rules, but paramodulation has to be slightly restricted. The CNF-transformer can make use of subformula replacement and standard Skolemization.

Paramodulation has to be carried out in such a way that all occurrences of the replaced term inside Skolem functions are treated consistently. Either all occurrrences are replaced, or none of them is replaced. Most theorem provers have a stronger restriction of paramodulation, in which no paramodulation at all is performed inside Skolem functions. In that case, one does not paramodulate into Skolem functions, and one can also keep the splitting rule.

We intend to implement the proof generation method (see [11]), and see how well the method can be used in practice. It needs to be seen how readable the resulting proofs are. In many cases, Skolem functions are meaningful (for example the Skolem function for $\exists y$ in the power set axiom $\forall x\ \exists y\ \forall \alpha\ (\ \alpha \subseteq x \leftrightarrow \alpha \in y)\ \ )$ and such Skolem functions are better not eliminated. One criterion could be that for such cases, functionality of the Skolem function is provable within the theory.

When translating resolution proofs on the clause level, there are quite some variations possible, which are not yet explored. As an example, if both f and g are Skolem functions, then one can obtain different variables for the two occurrences of $g(x)$ in the resolvent of $\forall x\ p(f(x)) \vee p(g(x))$ with
$\forall x\ \neg p(f(x)) \vee q(g(x))$. In some cases, it would be possible to obtain more liberal paramodulation in this way.

In addition, there are some variations possible when generating the serial relations during CNF-transformation. Currently, we use the weakest possible such relations.

It remains to be checked whether the reduction from improved Skolemization to standard Skolemization, described in [10], can be combined with the proof generation method of this paper.

Finally, it should be studied whether Skolemization with relations can be used as a theorem proving strategy. In this paper, we have assumed that the theorem prover is run with usual Skolem functions, and that afterwards the proof is reconstructed using relations. One could also enter the relational translation directly into the theorem prover. It follows from the results in this paper, that this would be a complete theorem proving strategy. Although it would probably not be efficient as a general-purpose strategy, it could perform well on certain fragments.

6 Acknowledgements

It was due to a question asked by Georg Moser during the Deduktionstreffen 2002 in Freiburg that I started thinking about the problem of eliminating Skolemization without eliminating cuts. I thank Marc Bezem for discussions on the subject of this paper and Dimitri Hendriks for the pleasant cooperation

in [6], which is one of the roots of the current paper.

References

[1] Jeremy Avigad. Eliminating definitions and skolem functions in first-order logic. In Harry Mairson, editor, *Proceedings of the 16-th Annual IEEE Symposion on Logic in Computer Science*, LICS, pages 139–146, Boston, Massachusetts, June 2001. IEEE Computer Society.

[2] 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.

[3] Matthias Baaz, Christian G. Fermüller, and Alexander Leitsch. A non-elementary speed-up in proof length by structural clause form transformation. In *IEEE Symposion on Logic in Computer Science 1994*, pages 213–219, 1994.

[4] Matthias Baaz and Alexander Leitsch. On skolemization and proof complexity. *Fundamenta Informatika*, 4(20):353–379, 1994.

[5] Leo Bachmair, Harald Ganzinger, Christopher Lynch, and Wayne Snyder. Basic paramodulation. *Information and Computation*, 121(2):172–192, 1995.

[6] Marc Bezem, Dimitri Hendriks, and Hans de Nivelle. Automated proof construction in type theory using resolution. In David McAllester, editor, *Automated Deduction - CADE-17*, volume 1831 of *LNAI*, pages 148–163. Springer Verlag, 2000.

[7] Marc Bezem, Dimitri Hendriks, and Hans de Nivelle. Automated proof construction in type theory using resolution. *Journal of Automated Reasoning*, 29(3-4):253–275, December 2002.

[8] Peter Clote and Jan Krajíček. *Arithmetic, Proof Theory and Computational Complexity*, volume 23 of *Oxford Logic Guides*. Oxford Science Publications, 1993.

[9] Thierry Boy de la Tour. An optimality result for clause form transformation. *Journal of Symbolic Computation*, 14:283–301, 1992.

[10] Hans de Nivelle. Extraction of proofs from the clausal normal form transformation. In Julian Bradfield, editor, *Proceedings of the 16 International Workshop on Computer Science Logic (CSL 2002)*, volume 2471 of *Lecture Notes in Artificial Intelligence*, pages 584–598, Edinburgh, Scotland, UK, September 2002. Springer.

[11] Hans de Nivelle. Implementing the clausal normal form transformation with proof generation. In Boris Konev and Renate Schmidt, editors, *Fourth Workshop on the Implementation of Logics*, volume ULCS-03-018, pages 69–83. University of Liverpool, Department of Computer Science, September 2003.

[12] Hans de Nivelle. Translation of resolution proofs into short first-order proofs without choice axioms. In Franz Baader, editor, *Automated deduction, CADE-19 : 19th International Conference on Automated Deduction*, volume 2741 of *Lecture Notes in Artificial Intelligence*, pages 365–379, Miami, USA, July 2003. Springer.

[13] 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.

[14] 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.

[15] Robert Nieuwenhuis and Albert Rubio. Paramodulation-based theorem proving. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 7, pages 371–443. Elsevier Science, B.V., 2001.

[16] Andreas Nonnengart. Strong Skolemization. Technical Report MPI-I-96-2-010, Max Planck Institut für Informatik Saarbrücken, 1996.

[17] 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.

[18] V.P. Orevkov. Lower bounds for increasing complexity of derivations after cut elimination. *Zapiski Nauchnykh Seminarov Leningradskogo Otdeleniya Matematicheskogo Instituta Imenyi V.A. Steklova AN SSSR*, 88:137–161, 1979. English translation in Journal of Soviet Mathematics 2337–2350, 1982.

[19] Frank Pfenning. Analytic and non-analytic proofs. In R.E. Shostak, editor, *7th International Conference on Automated Deduction*, volume 170 of *Lecture Notes in Artificial Intelligence*, pages 394–413, Napa, California, USA, May 1984. Springer Verlag.

[20] Jörg
Siekmann, Christoph Benzmüller, Vladimir Brezhnev, Lassaad Cheickhrouhou, Armin Fiedler, Andreas Franke, Helmut Horacek, Michael Kohlhase, Andreas Meier, Erica Melis, Markus Moschner, Immanuel Normann, Martin Pollet, Volker Sorge, Carsten Ullrich, Claus-Peter Wirth, and Jürgen Zimmer. Proof development with Ωmega. In Andrei Voronkov, editor, *Automated Deduction - CADE-18*, volume 2392 of *LNAI*, pages 144–149. Springer Verlag, 2002.

[21] R. Statman. Lower bounds on herbrand's theorem. In *Proceedings of the American Mathematical Society*, volume 75-1, pages 104–107. American Mathematical Society, June 1979.

Implementing the Clausal Normal Form Transformation with Proof Generation

Hans de Nivelle

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

Abstract. We explain how we intend to implement the clausal normal form transformation with proof generation. We present a convenient data structure for sequent calculus proofs, which will be used for representing the generated proofs. The data structure allows easy proof checking and generation of proofs. In addition, it allows convenient implementation of proof normalization, which is necessary in order to keep the size of the generated proofs acceptable.

1 Introduction

In [2], a method for generating explicit proofs from the clausal normal form transformation was presented, which does not make use of choice axioms. It is our intention to implement this method. In this paper we introduce the data structure for the representation of proofs that we intend to use, and we give a general algorithm scheme, with which one can translate formulas and obtain correctness proofs at the same time.

In [2], natural deduction was used for showing that it is in principle possible to generate explicit proofs. It is however in practice better to use sequent calculus, because sequent calculus allows proof reductions that reduce the size of generated proofs. In order to be able to keep the sizes of the resulting proofs acceptable, it is necessary to normalize proofs in such a way that repeated building up of contexts is avoided.

In the preceeding paper [1], which was still proposing to use choice axioms, it was explained how to do this in type theory. An intermediate calculus was introduced, called the *replacement calculus*, which allows for proof normalization. After normalization, the resulting proof could be translated into type theory through a simple replacement schema. If one uses sequent calculus instead of natural deduction, the standard reductions of sequent calculus can do the proof normalization. It turns out that proof normalization in the replacement calculus corresponds to a restricted form of cut elimination in sequent calculus. Therefore, if one uses sequent calculus instead of natural deduction, the replacement calculus can be omitted alltogether.

In the next section we introduce sequent calculus. After that, we introduce the data structure that we will use for representing sequent calculus proofs. Then we will give a general scheme for translating formulas and generating proofs at the same time. In the last section, we show that our sequent proof data structure is convenient for implementing the kind of proof reduction that we need.

2 Sequent Calculus

Definition 1. *A* sequent *is an object of form $\Gamma \vdash \Delta$, where both Γ and Δ are multisets.*

We give the rules of sequent calculus. We assume that α-equivalent formulas are not distinguished. We also give equality rules, although equality plays no rule in the CNF-transformation.

$$\text{(axiom)}\ \frac{}{\Gamma, A \vdash \Delta, A}$$

$$\text{(cut)}\ \frac{\Gamma, A \vdash \Delta \qquad \Gamma \vdash \Delta, A}{\Gamma \vdash \Delta}$$

Structural Rules:

$$\text{(weakening left)}\ \frac{\Gamma \vdash \Delta}{\Gamma, A \vdash \Delta} \qquad \text{(weakening right)}\ \frac{\Gamma \vdash \Delta}{\Gamma \vdash \Delta, A}$$

$$\text{(contraction left)}\ \frac{\Gamma, A, A \vdash \Delta}{\Gamma, A \vdash \Delta} \qquad \text{(contraction right)}\ \frac{\Gamma \vdash \Delta, A, A}{\Gamma \vdash \Delta, A}$$

Rules for the truth constants:

$$(\top\text{-left})\ \frac{\Gamma \vdash \Delta}{\Gamma, \top \vdash \Delta} \qquad (\top\text{-right})\ \frac{}{\Gamma \vdash \Delta, \top}$$

$$(\bot\text{-left})\ \frac{}{\Gamma, \bot \vdash \Delta} \qquad (\bot\text{-right})\ \frac{\Gamma \vdash \Delta}{\Gamma \vdash \Delta, \bot}$$

Rules for $\neg$:

$$(\neg\text{-left})\ \frac{\Gamma \vdash \Delta, A}{\Gamma, \neg A \vdash \Delta} \qquad (\neg\text{-right})\ \frac{\Gamma, A \vdash \Delta}{\Gamma \vdash \Delta, \neg A}$$

Rules for $\wedge, \vee, \leftarrow, \leftrightarrow$:

$$(\wedge\text{-left})\ \frac{\Gamma, A, B \vdash \Delta}{\Gamma, A \wedge B \vdash \Delta} \qquad (\wedge\text{-right})\ \frac{\Gamma \vdash \Delta, A \qquad \Gamma \vdash \Delta, B}{\Gamma \vdash \Delta, A \wedge B}$$

$$(\vee\text{-left})\ \frac{\Gamma,\ A \vdash \Delta \qquad \Gamma, B \vdash \Delta}{\Gamma, A \vee B \vdash \Delta} \qquad (\vee\text{-right})\ \frac{\Gamma \vdash \Delta, A, B}{\Gamma \vdash \Delta, A \vee B}$$

$$(\rightarrow\text{-left})\ \frac{\Gamma \vdash \Delta, A \qquad \Gamma, B \vdash \Delta}{\Gamma, A \rightarrow B \vdash \Delta} \qquad (\rightarrow\text{-right})\ \frac{\Gamma, A \vdash \Delta, B}{\Gamma \vdash \Delta, A \rightarrow B}$$

$$(\leftrightarrow\text{-left})\ \frac{\Gamma, A \rightarrow B,\ B \rightarrow A \vdash \Delta}{\Gamma, A \leftrightarrow B \vdash \Delta}$$

$$(\leftrightarrow\text{-right})\ \frac{\Gamma \vdash \Delta, A \rightarrow B \qquad \Gamma \vdash \Delta, B \rightarrow A}{\Gamma \vdash \Delta, A \leftrightarrow B}$$

Rules for the quantifiers:

$$(\forall\text{-left})\ \frac{\Gamma, P[x := t] \vdash \Delta}{\Gamma, \forall x\ P \vdash \Delta} \qquad (\forall\text{-right})\ \frac{\Gamma \vdash \Delta, P[x := y]}{\Gamma \vdash \Delta, \forall x\ P}$$

$$(\exists\text{-left})\ \frac{\Gamma, P[x := y] \vdash \Delta}{\Gamma, \exists x\ P \vdash \Delta} \qquad (\exists\text{-right})\ \frac{\Gamma \vdash \Delta, P[x := t]}{\Gamma \vdash \Delta, \exists x\ P}$$

The t is an arbitrary term. The y is a variable which is not free in Γ, Δ, P

Rules for equality:

$$(\text{refl})\ \frac{}{\Gamma \vdash \Delta,\ t \approx t}$$

$$(\text{repl-left})\ \frac{t_1 \approx t_2,\ \Gamma[t_1] \vdash \Delta}{t_1 \approx t_2,\ \Gamma[t_2] \vdash \Delta} \qquad (\text{repl-right})\ \frac{t_1 \approx t_2,\ \Gamma \vdash \Delta[t_1]}{t_1 \approx t_2,\ \Gamma \vdash \Delta[t_2]}.$$

The last rules mean: If $t_1 \approx t_2$ appears among the premisses, then an arbitrary occurrence of t_1 can be replaced by t_2. The replacement can take place either on the left or on the right. Only one replacement at the same time is possible.

3 Proof Trees

We introduce a concise sequent calculus format, which allows for easy proof checking and implementation of proof reductions. It is closely related to the embedding of sequent calculus in LF, which is introduced in [5].

We first prove a simple lemma that shows that one should avoid explicitly mentioning the formulas occurring in the proof:

Lemma 1. *Consider the sequents* $(\neg\neg)^n A \vdash A$, *for* $n \geq 0$.

If one has a proof representation method that explicitly mentions the formulas in a sequent, then the proofs have size $O(n^2)$.

Proof. Because one will have to represent all subformulas
$A, \neg A, (\neg)^2 A, (\neg)^3 A, \ldots, (\neg\neg)^n A$.

Nevertheless, the proof has a length of only n steps. If one does not mention the formulas, one can obtain a representation of size n. In our representation, we avoid explicitly mentioning formulas by assigning labels to them. Whenever a new formula is constructed, it will be clear what the new formula is, from the way it is constructed, so that we will not have to mention it.

Definition 2. *We redefine a sequent as an object of form $\Gamma \vdash \Delta$, where both Γ and Δ are sets of labelled formulas. So we have $\Gamma = \{\alpha_1{:}A_1, \ldots, \alpha_p{:}A_p\}$ and $\Delta = \{\beta_1{:}B_1, \ldots, \beta_q{:}B_q\}$, where $\alpha_i = \alpha_j$ implies $i = j$ and $\beta_i = \beta_j$ implies $i = j$.*

In case there is no A', s.t. $\alpha{:}A' \in \Gamma$, the notation $\Gamma + \alpha{:}A$ denotes $\Gamma \cup \{\alpha{:}A\}$. Otherwise, $\Gamma + \alpha{:}A$ is undefined. (even when $A = A'$)

In case there is an A, s.t. $\alpha{:}A \in \Gamma$, the notation $\Gamma - \alpha$ denotes $\Gamma \backslash \{\alpha{:}A\}$. Otherwise $\Gamma - \alpha$ is not defined.

In case there is an A, s.t. $\alpha{:}A \in \Gamma$, the notation $\Gamma[\alpha]$ denotes A. Otherwise $\Gamma[\alpha]$ is not defined.

For Δ, we define $\Delta + \beta{:}B, \quad \Delta - \beta, \quad \Delta[\beta]$ in the same way as for Γ.

Proofs are checked *top-down*, i.e. from the goal sequent towards the axioms. For each node in the proof tree, the node states the label of the conslusion in the derived sequent, and what labels the premisses should receive in the child sequents. During checking, the conclusion is removed from the sequent (if it exists, and has the right form), and replaced by the children, after which proof checking continues.

Definition 3. *We recursively define* proof trees *and when a proof tree* accepts *a labelled sequent. In the following list, we implicitly assume that α, β are labels. We will omit the definedness conditions. So we will assume that $\Delta[\alpha] = \Delta[\beta]$ means: $F[\alpha]$ and $F[\beta]$ are both defined and $F[\alpha] = F[\beta]$.*

- ax(α, β) *is a proof tree. It is a proof of $\Gamma \vdash \Delta$, if $\Gamma[\alpha]$ is an α-variant of $\Delta[\beta]$.*
- *If π_1, π_2 are proof trees, and A is a formula, then* cut$(A, \pi_1, \alpha, \pi_2, \beta)$ *is also a proof tree. It is a proof of $\Gamma \vdash \Delta$ if π_1 is a proof of $\Gamma + \alpha{:}A \vdash \Delta$ and π_2 is a proof of $\Gamma \vdash \Delta + \beta{:}A$.*
- *If π is a proof tree, then* weakenleft(α, π) *is also a proof tree. It is a proof of $\Gamma \vdash \Delta$ if π is a proof of $\Gamma - \alpha \vdash \Delta$.*
- *If π is a proof tree, then* weakenright(β, π) *is also a proof tree. It is a proof of $\Gamma \vdash \Delta$ if π is a proof of $\Gamma \vdash \Delta - \beta$.*
- *If π is a proof tree, then* contrleft$(\alpha_1, \pi, \alpha_2)$ *is also a proof tree. It is a proof of $\Gamma \vdash \Delta$ if π is a proof of $\Gamma + \alpha_1{:}A[\alpha_2] \vdash \Delta$.*
- *If π is a proof tree, then* contrright(β_1, π, β_2) *is also a proof tree. It is a proof of $\Gamma \vdash \Delta$ if π is a proof of $\Gamma \vdash \Delta + \beta_1{:}\Delta[\beta_2]$.*
- *If π is a proof tree, then* trueleft(α, π) *is also a proof tree. It is a proof of $\Gamma \vdash \Delta$ if $\Gamma[\alpha] = \top$, and π is a proof of $\Gamma - \alpha \vdash \Delta$.*

- trueright(β) *is a proof tree. It is a proof of* $\Gamma \vdash \Delta$ *if* $\Delta[\beta] = \top$.
- falseleft(α) *is a proof tree. It is a proof of* $\Gamma \vdash \Delta$ *if* $\Gamma[\alpha] = \bot$.
- falseright(β, π) *is a proof tree. It is a proof of* $\Gamma \vdash \Delta$ *if* $\Delta[\beta] = \bot$ *and* π *is a proof of* $\Gamma \vdash \Delta - \beta$.
- *If* π *is a proof tree, then* negleft(α, π, β) *is also a proof tree. It is a proof of* $\Gamma \vdash \Delta$ *if* $\Gamma[\alpha]$ *has form* $\neg A$, *and* π *is a proof of* $\Gamma \vdash \Delta + \beta{:}A$.
- *If* π *is a proof tree, then* negright(β, π, α) *is also a proof tree. It is a proof of* $\Gamma \vdash \Delta$ *if* $\Delta[\beta]$ *has form* $\neg A$, *and* π *is a proof of* $\Gamma + \alpha{:}A \vdash \Delta$.
- *If* π *is a proof tree and* $\alpha_1 \neq \alpha_2$, *then* andleft($\alpha, \pi, \alpha_1, \alpha_2$) *is also a proof tree. It is a proof of* $\Gamma \vdash \Delta$ *if* $\Gamma[\alpha]$ *has form* $A \wedge B$, *and*
  π *is a proof of* $(\Gamma - \alpha) + \alpha_1{:}A + \alpha_2{:}B \vdash \Delta$.
- *If* π_1, π_2 *are proof trees, then* andright($\beta, \pi_1, \beta_1, \pi_2, \beta_2$) *is also a proof tree. It is a proof of* $\Gamma \vdash \Delta$ *if* $\Delta[\beta]$ *has form* $A \wedge B$,
  π_1 *is a proof of* $\Gamma \vdash (\Delta - \beta) + \beta_1{:}A$, *and*
  π_2 *is a proof of* $\Gamma \vdash (\Delta - \beta) + \beta_2{:}B$.
- *If* π_1, π_2 *are proof trees, then* orleft($\alpha, \pi_1, \alpha_1, \pi_2, \alpha_2$) *is also a proof tree. It is a proof of* $\Gamma \vdash \Delta$ *if* $\Delta[\alpha]$ *has form* $A \vee B$,
  π_1 *is a proof of* $(\Gamma - \alpha) + \alpha_1{:}A \vdash \Delta$, *and*
  π_2 *is a proof of* $(\Gamma - \alpha) + \alpha_2{:}B \vdash \Delta$.
- *If* π *is a proof tree and* $\beta_1 \neq \beta_2$, *then* orright($\beta, \pi, \beta_1, \beta_2$) *is also a proof tree. It is a proof of* $\Gamma \vdash \Delta$ *if* $\Delta[\beta]$ *has form* $A \vee B$, *and*
  π *is a proof of* $\Gamma \vdash (\Delta - \beta) + \beta_1{:}A + \beta_2{:}B$.
- *If* π *is a proof tree, then* impliesleft($\alpha, \pi_1, \beta_1, \pi_2, \alpha_2$) *is also a proof tree. It is a proof of* $\Gamma \vdash \Delta$ *if* $\Gamma[\alpha]$ *has form* $A \rightarrow B$, *and*
  π_1 *is a proof of* $(\Gamma - \alpha) \vdash \Delta + \beta_1{:}A$, *and*
  π_2 *is a proof of* $(\Gamma - \alpha) + \alpha_2{:}B \vdash \Delta$.
- *If* π *is a proof tree and* $\alpha_1 \neq \beta_2$, *then* impliesright($\beta, \pi_1, \alpha_1, \pi_2, \beta_2$) *is also a proof tree. It is a proof of* $\Gamma \vdash \Delta$ *if* $\Delta[\beta]$ *has form* $A \rightarrow B$,
  π_1 *is a proof of* $\Gamma + \alpha_1{:}A \vdash (\Delta - \beta) + \beta_2{:}B$.
- *If* π *is a proof tree and* $\alpha_1 \neq \alpha_2$, *then* equivleft($\alpha, \pi, \alpha_1, \alpha_2$) *is also a proof tree. It is a proof of* $\Gamma \vdash \Delta$ *if* $\Gamma[\alpha]$ *has form* $A \leftrightarrow B$, *and*
  π *is a proof of* $(\Gamma - \alpha) + \alpha_1{:}(A \rightarrow B) + \alpha_2{:}(B \rightarrow A) \vdash \Delta$.
- *If* π_1, π_2 *are proof trees, then* equivright($\beta, \pi_1, \beta_1, \pi_2, \beta_2$) *is also a proof tree. It is a proof of* $\Gamma \vdash \Delta$ *if* $\Delta[\beta]$ *has form* $A \leftrightarrow B$,
  π_1 *is a proof of* $\Gamma \vdash (\Delta - \beta) + \beta_1{:}(A \rightarrow B)$, *and*
  π_2 *is a proof of* $\Gamma \vdash (\Delta - \beta) + \beta_2{:}(B \rightarrow A)$.
- *If* π *is a proof tree and* t *is a term, then* forallleft(α, π, α_1, t) *is also a proof tree. It is a proof of* $\Gamma \vdash \Delta$ *if* $\Gamma[\alpha]$ *has form* $\forall x\ P$ *and* π *is a proof of* $(\Gamma - \alpha) + \alpha_1{:}P[x := t] \vdash \Delta$.
- *If* π *is a proof tree and* y *is a variable, then* forallright(β, π, β_1, y) *is also a proof tree. It is a proof of* $\Gamma \vdash \Delta$ *if* $\Delta[\beta]$ *has form* $\forall x\ P$,
  y *is not free in* Γ, Δ *or* P, *and*
  π *is a proof of* $\Gamma \vdash (\Delta - \beta) + \beta_1{:}P[x := y]$.
- *If* π *is a proof tree and* y *is a variable, then* existsleft(α, π, α_1, y) *is also a proof tree. It is a proof of* $\Gamma \vdash \Delta$ *if* $\Gamma[\alpha]$ *has form* $\exists x\ P$,
  y *is not free in* Γ, Δ *or* P, *and*
  π *is a proof of* $(\Gamma + \alpha) + \alpha_1{:}P[x := y] \vdash \Delta$

- *If π is a proof tree and t is a term, then* $\mathrm{existsright}(\beta, \pi, \beta_1, t)$ *is also a proof tree. It is a proof of $\Gamma \vdash \Delta$ if $\Delta[\beta]$ has form $\exists x\ P$ and*
  π is a proof of $\Gamma \vdash (\Delta - \beta) + \beta_1{:}P[x := t]$.
- *If t is a term, then* $\mathrm{eqrefl}(\beta, t)$ *is a proof tree. It is a proof of $\Gamma \vdash \Delta$ if*
  $\Delta[\beta] = (t \approx t)$.
- *If π is a proof tree and ρ is a position, then* $\mathrm{replleft}(\alpha_1, \alpha_2, \pi, \rho, \alpha_3)$ *is also a proof tree. It is a proof of $\Gamma \vdash \Delta$ if $\Gamma[\alpha_1]$ has form $t_1 \approx t_2$, if $\Gamma[\alpha_2]$ has form $A_\rho[t_2]$, and π is a proof of $(\Gamma - \alpha_2) + \alpha_3{:}A_\rho[t_1] \vdash \Delta$.*
- *If π is a proof tree and ρ is a position, then* $\mathrm{replright}(\alpha, \beta_1, \pi, \rho, \beta_2)$ *is also a proof tree. It is a proof of $\Gamma \vdash \Delta$ if $\Gamma[\alpha]$ has form $t_1 \approx t_2$, if $\Delta[\beta_1]$ has form $B_\rho[t_1]$, and π is a proof of $\Gamma \vdash (\Delta - \beta_1) + \beta_2{:}B_\rho[t_2]$.*

As an example, consider the following proof:

$$\dfrac{\dfrac{\alpha_1{:}A,\ \alpha_2{:}B \vdash \beta_1{:}B \qquad\qquad \alpha_1{:}A,\ \alpha_2{:}B \vdash \beta_2{:}A}{\alpha_1{:}A,\ \alpha_2{:}B \vdash \beta{:}B \wedge A}}{\alpha{:}A \wedge B \vdash \beta{:}B \wedge A}$$

It can be represented by the following proof term:

$$\mathrm{andleft}(\alpha, \mathrm{andright}(\beta, \mathrm{ax}(\alpha_2, \beta_1), \beta_1, \mathrm{ax}(\alpha_1, \beta_2), \beta_2), \alpha_1, \alpha_2).$$

Following [5], we consider a rule as a binder that binds the labels that it introduces in the subproofs where the label is introduced. For example, $\mathrm{andleft}(\alpha, \pi, \alpha_1, \alpha_2)$ introduces the labels α_1, α_2 in π. Therefore, it can be viewed as binding any occurrences of α_1, α_2 in π. Likewise, we consider $\mathrm{forallright}(\beta, \pi, \beta_1, y)$, as a binder that binds any occurrences of y in π.

Viewing the rules as binders makes it possible to define a notion of α-equivalence of proofs. This has the advantage that label conflicts can be resolved by renaming labels. Without α-equivalence, a rule introducing some formula with label α cannot be applied on a labelled sequent already containing α. However, if we use α-equivalence, we can rename α into a new label α' and continue proof checking. As an example, the proof tree given a few lines above would not be a proof of $\alpha{:}A \wedge B,\ \alpha_1{:}A \vdash \beta{:}B \wedge A$. Using α-equivalence, we can replace α_1 in the proof tree by some α_1' and the sequent will be accepted. The main advantage of this is that proof checking becomes monotone:

Lemma 2. *If π is a proof tree, which is a proof of some labelled sequent $\Gamma \vdash \Delta$, and $\Gamma \subseteq \Gamma'$, $\Delta \subseteq \Delta'$, then π is also a proof of $\Gamma' \vdash \Delta'$.*

The following property is important for proof reductions. It is assumed that substitution is capture avoiding:

Lemma 3. *Let π a proof of labelled sequent $\Gamma \vdash \Delta$ containing a free variable x. Let t be some term. Then $\pi[x := t]$ is a proof of $(\Gamma \vdash \Delta)[x := t]$.*

The following property is important, because it makes it possible to use proof terms as schemata, i.e. as objects that can be instantiated.

Theorem 1. *Let π be a proof of a labelled sequent $\Gamma \vdash \Delta$. Let $A(x_1, \ldots, x_n)$ be an n-ary atom occurring in $\Gamma \vdash \Delta$, s.t. $x_1, \ldots, x_n$ are its free variables. Let $F(x_1, \ldots, x_n, y_1, \ldots, y_m)$ be a formula having at least free variables $x_1, \ldots, x_n$, and with possible other free variables $y_1, \ldots, y_m$. Assume that no occurrence of $A(x_1, \ldots, x_n)$ in $\Gamma \vdash \Delta$ is in the scope of a quantifier that binds one of the y_j and that no occurrence of $A(x_1, \ldots, x_n)$ in a cut formula occurring in π is in the scope of a quantifier that binds one of the y_j. Let π' be obtained from π by substituting $A(x_1, \ldots, x_n) := F(x_1, \ldots, x_n, y_1, \ldots, y_m)$ in every cut formula in π. Then π' is a proof of $(\Gamma \vdash \Delta)[A(x_1, \ldots, x_n) := F(x_1, \ldots, x_n, y_1, \ldots, y_m)]$.*

Note that when π is cut free, then $\pi' = \pi$. The reason that Theorem 1 holds, is the fact that the cut rule is the only rule that explicitly mentions formulas.

In case variables from $y_1, \ldots, y_m$ are caught, it is always possible to obtain an α-variant of $\Gamma \vdash \Delta$ and π, s.t. no capture takes place. The same holds for any cut formula in π.

As an example, consider the sequent $\forall x(A(x) \wedge B) \vdash (\forall x A(x)) \wedge B$, which clearly has a cut free proof. Lemma 1 allows to substitute $P(y, z)$ for B, (because y and z are not caught), but it does not allow to substitute $P(x, y, z)$ for B. The sequent can be renamed into $\forall x_1(A(x_1) \wedge B) \vdash (\forall x_1 A(x_1)) \wedge B$.

4 The Negation Normal Form Transformation

We describe in detail how we intend to implement the negation normal form transformation with proof generation.

Definition 4. *Formula F is in* negation normal form *(NNF) if* **(1)** *F does not contain $\rightarrow$ or $\leftrightarrow$,* **(2)** *negation is applied only on atoms in F,* **(3)** *if F contains $\top$ (or $\bot$,) then $F = \top$, (or $\bot$).*

A formula can be easily transformed into NNF by two rewrite systems. The first rewrite system removes $\rightarrow$ and $\leftrightarrow$, and it pushes the negations inwards. The second rewrite system moves $\bot$ and $\top$ upwards until they either disappear, or reach the top of the formula. The rewrite systems could be combined into one rewrite system, but that would be inefficient, because the two rewrite systems are more efficient with different rewrite systems. The first rewrite system is given by the following table:

$$\begin{array}{ll}
A \rightarrow B & \Rightarrow \neg A \vee B \\
A \leftrightarrow B & \Rightarrow (\neg A \vee B) \wedge (A \vee \neg B) \\
 & \\
\neg\neg A & \Rightarrow A \\
\neg(A \vee B) & \Rightarrow \neg A \wedge \neg B \\
\neg(A \wedge B) & \Rightarrow \neg A \vee \neg B \\
\neg(\forall x\ P(x)) & \Rightarrow \exists x\ \neg P(x) \\
\neg(\exists x\ P(x)) & \Rightarrow \forall x\ \neg P(x)
\end{array}$$

The following algorithm normalizes a formula under the set of rules.

Algorithm 1

```
formula nnf12(formula$F$)
begin
    while there are a rule $A \Rightarrow B$ and a substitution $\Theta$, s.t.
    $A\Theta = F$ do
        $F := B\Theta$
    if $F$ is an atom, $A = \bot$ or $A = \top$, then return $F$
    if $F$ has form $\neg A$, with $A$ an atom, $A = \bot$, or $A = \top$, then return $\neg A$.
    if $F$ has form $A \wedge B$, then return $\mathrm{nnf12}(A) \wedge \mathrm{nnf12}(B)$
    if $F$ has form $A \vee B$, then return $\mathrm{nnf12}(A) \vee \mathrm{nnf12}(B)$
    if $F$ has form $\forall x\ P(x)$, then return $\forall x\ \mathrm{nnf12}(P(x))$
    if $F$ has form $\exists x\ P(x)$, then return $\exists x\ \mathrm{nnf12}(P(x))$
end
```

The algorithm implements a particular rewrite strategy, namely *outside-inside* normalization. It assumes that the rewrite front starts at the outside and then moves inward. When the formula has been normalized at one point, then this point does not need to be reconsidered anymore. There second part of the rewrite system needs exactly the opposite strategy, *inside-outside* normalization. If one would combine the systems, one would have to look for possible rewrites everywhere in the formula, which is less efficient.

Definition 5. *A* justified sequent *is a pair of form* $(\alpha{:}A \vdash \beta{:}B,\ \pi)$, *s.t.* $\alpha \neq \beta$ *and* π *is a proof of* $\alpha{:}A \vdash \beta{:}B$. *A* justified rewrite rule *is a justified sequent* $(\alpha{:}A \vdash \beta{:}B,\ \pi)$, *s.t.* $A \Rightarrow B$ *is a rewrite rule.*

There is no formal distinction between a justified sequent and a justified rewrite rule, but we give them different names because their roles are different.

We now modify the rewrite algorithm, so that it will output a proof at the same time with its result. It will do this by returning a justified sequent.

Algorithm 2 *Function* $\mathrm{nnf12}(F, \alpha)$ *returns a justified sequent* $(\alpha{:}F \vdash \beta{:}F',\ \pi)$, *s.t.* $F' = \mathrm{nnf12}(F)$, *and* β *is some new label.*

```
justifiedsequent nnf12( formula $F$, label $\alpha$)
begin
    array of justifiedsequent $\Pi$;
    Initialize $\Pi$ to the empty (zero length) array.
    while there are a justified rewrite rule $(\alpha'{:}A' \vdash \beta'{:}B',\ \pi')$ and
            a substitution $\Theta$, s.t. $A'\Theta = F$ do
    begin
        Let $\gamma$ be a new label, not occurring in $\Pi$, and distinct from $\alpha$.
        Assign $\pi' := \pi'[\alpha' := \alpha,\ \beta' := \gamma]$. (so that $\pi'$ now proves $\alpha{:}A' \vdash \gamma{:}B$)
        Append $(\alpha{:}A'\Theta \vdash \gamma{:}B'\Theta,\ \pi')$ to $\Pi$. (the length of $\Pi$ is increased by $1$,
                there is no need to modify $\pi'$ because of Theorem 1)
        Assign $F := B\Theta$.
        Assign $\alpha := \gamma$.
    end
    If $F$ is an atom, $F = \bot$ or $F = \top$, then
        return applycut($\Pi$).
    If $F$ has form $\neg A$, where $A$ is an atom, $A = \bot$ or $A = \top$, then
        return applycut($\Pi$).
    If $F$ has form $A_1 \wedge A_2$, then
    begin
        Let $\alpha_1, \alpha_2, \beta$ be new, distinct labels.
        Assign $B_1, \beta_1, \pi_1$ from $(\alpha_1{:}A_1 \vdash \beta_1{:}B_1,\ \pi_1) :=$ nnf12($A_1, \alpha_1$)
        Assign $B_2, \beta_2, \pi_2$ from $(\alpha_2{:}A_2 \vdash \beta_2{:}B_2,\ \pi_2) :=$ nnf12($A_2, \alpha_2$)
        Append $\alpha{:}A_1 \wedge A_2 \vdash \beta{:}B_1 \wedge B_2$,
            andleft($\alpha$,
                andright($\beta$,
                    weakenleft($\alpha_2, \pi_1$), $\beta_1$,
                    weakenleft($\alpha_1, \pi_2$), $\beta_2$),
                $\alpha_1, \alpha_2$) ) to $\Pi$.
        return applycut($\Pi$)
    end
    If $F$ has form $A_1 \vee A_2$, then
    begin
        Let $\alpha_1, \alpha_2, \beta$ be new, distinct labels.
        Assign $B_1, \beta_1, \pi_1$ from $(\alpha_1{:}A_1 \vdash \beta_1{:}B_1,\ \pi_1) :=$ nnf12($A_1, \alpha_1$)
        Assign $B_2, \beta_2, \pi_2$ from $(\alpha_2{:}A_2 \vdash \beta_2{:}B_2,\ \pi_2) :=$ nnf12($A_2, \alpha_2$)
        Append $(\alpha{:}A_1 \vee A_2 \vdash \beta{:}B_1 \vee B_2$,
            orright($\beta$,
                orleft($\alpha$,
                    weakenright($\beta_2, \pi_1$), $\alpha_1$,
                    weakenright($\beta_1, \pi_2$), $\alpha_2$),
                $\beta_1, \beta_2$) ) to $\Pi$.
        return applycut($\Pi$).
    end
    If $F$ has form $\forall x\ P(x)$, then
    begin
        Let $\alpha_1$ and $\beta$ be a new, distinct labels.
        Assign $Q(x), \beta_1, \pi_1$ from $(\alpha_1{:}P(x) \vdash \beta_1{:}Q(x),\ \pi_1) := \mathrm{nnf12}(P(x), \alpha_1)$
        Append $(\alpha{:}\forall x\ P(x) \vdash \beta{:}\forall x\ Q(x),$
                    $\mathrm{forallright}(\beta, \mathrm{forallleft}(\alpha, \pi_1, \alpha_1, x), \beta_1, x)\ )$ to $\Pi$.
        return $\mathrm{applycut}(\Pi)$
    end

    If $F$ has form $\exists x\ P(x)$, then
    begin
        Let $\alpha_1$ and $\beta$ be new, distinct labels.
        Assign $Q(x), \beta_1, \pi_1$ from $(\alpha_1{:}P(x) \vdash \beta_1{:}Q(x),\ \pi_1) := \mathrm{nnf12}(P(x), \alpha_1)$
        Append $(\alpha{:}\exists x\ P(x) \vdash \beta{:}\exists x\ Q(x),$
                    $\mathrm{existsleft}(\alpha, \mathrm{existsright}(\beta, \pi_1, \beta_1, x), \alpha_1, x)\ )$ to $\Pi$.
        return $\mathrm{applycut}(\Pi)$
    end
end
```

Function applycut(Π) *combines the proofs π_i of $\alpha_i{:}A_i \vdash \beta_i{:}B_i$ into one proof by using the cut rule. It must be the case that $\beta_{i+1} = \alpha_i$, and $B_{i+1} = A_i$, for $1 \leq i < |\Pi|$.*

```
(justifiedsequent) applycut( array of justifiedsequent $\Pi$)
begin
    $\Sigma$ is a variable of type labelled sequent.
    $\pi$ is a variable of type proof tree.
    Assign $(\Sigma, \pi) := \Pi_1$

    for $i := 2$ to $|\Pi|$ do
    begin
        Assign $(\alpha{:}A \vdash \beta{:}B) := \Sigma$
        Assign $(\beta{:}B \vdash \gamma{:}C,\ \rho) := \Pi_i$
        Assign $\Sigma := \alpha{:}A \vdash \gamma{:}C$
        Assign $\pi := \mathrm{cut}(B, \mathrm{weakenleft}(\alpha, \rho), \beta, \mathrm{weakenright}(\gamma, \pi), \beta)$
    end
    return $(\Sigma,\ \pi)$
end
```

We now come to the second part of the rewrite system that will ensure the third condition of Definition 4.

$A \vee \bot \Rightarrow A$	$A \wedge \bot \Rightarrow \bot$
$A \vee \top \Rightarrow \top$	$A \wedge \top \Rightarrow A$
$\bot \vee A \Rightarrow A$	$\bot \wedge A \Rightarrow \bot$
$\top \vee A \Rightarrow \top$	$\top \wedge A \Rightarrow A$
$\forall x \ \bot \ \Rightarrow \bot$	$\exists x \ \bot \ \Rightarrow \bot$
$\forall x \ \top \ \Rightarrow \top$	$\exists x \ \top \ \Rightarrow \top$

In order to obtain a normal form, Algorithm 1 cannot be used, because the outside-inside strategy does generally not result in a normal form. Instead, an inside-outside rewrite strategy has to be used:

Algorithm 3

```
formula nnf3 (formula $F$ )
begin
    if $F$ is an atom, $A = \bot$ or $A = \top$, then $G := F$
    if $F$ has form $\neg A$, with $A$ an atom, $A = \bot$, or $A = \top$, then $G := F$
    if $F$ has form $A \wedge B$, then $G := \mathrm{nnf3}(A) \wedge \mathrm{nnf3}(B)$
    if $F$ has form $A \vee B$, then $G := \mathrm{nnf3}(A) \vee \mathrm{nnf3}(B)$
    if $F$ has form $\forall x \ P(x)$, then $G := \forall x \ \mathrm{nnf3}(P(x))$
    if $F$ has form $\exists x \ P(x)$, then $G := \exists x \ \mathrm{nnf3}(P(x))$

    while there are a rule $A \Rightarrow B$ and a substitution $\Theta$, s.t. $A\Theta = G$ do
        $G := B\Theta$
end
```

Algorithm 3 differs from Algorithm 1 only in the fact that rewriting on the current level is attempted only after the subterms have been normalized.

Algorithm 2 can be easily modified correspondingly, by moving the while-loop in the beginning towards the end. It can be also easily adopted to situations where more complicated rewrite strategies are needed.

5 Subformula Replacement

Some steps in the clausal normal form transformation can cause exponential blowup of the formula. The problematic steps are the replacement of $A \leftrightarrow B$ by $(\neg A \vee B) \wedge (A \vee \neg B)$, and the factoring of conjunctions over disjunctions performed by the following rules: $(A \wedge B) \vee C \Rightarrow (A \vee C) \wedge (B \vee C)$, $A \vee (B \wedge C) \Rightarrow (A \vee B) \wedge (A \vee C)$.

Expansion of $\leftrightarrow$ would cause exponential blowup on the following sequence of formulas

$$(a_1 \leftrightarrow (a_2 \leftrightarrow \cdots (a_{n-1} \leftrightarrow a_n))), \ n > 0.$$

Factoring would cause exponential blowup on the following sequence of formulas

$$(a_1 \wedge b_1) \vee \cdots \vee (a_n \wedge b_n), \ n > 0.$$

In order to avoid this, it is possible to use subformula replacement. For example, in the last formula, one can introduce new symbols $x_1, \ldots, x_n$, and replace it by the equisatisfiable set of formulas

$$x_1 \vee \cdots \vee x_n,\ x_1 \leftrightarrow (a_1 \wedge b_1), \ldots, x_n \leftrightarrow (a_n \wedge b_n).$$

Subformula replacement as such is not first-order, but it can be easily dealt with within first-order logic, by observing that the new names are abbreviations of certain formulas. During the CNF-transformation, we allow to add premisses of the following form to the set of premisses:

$$\forall x_1 \cdots x_n\ X(x_1, \ldots, x_n) \leftrightarrow F(x_1, \ldots, x_n).$$

X is a new symbol that does not yet occur in the premisses and also not in $F(x_1, \ldots, x_n)$. When the resolution prover succeeds, one obtains a proof π of a sequent $\Gamma, D_1, \ldots, D_k \vdash \bot$, in which Γ is the set of original first-order formulas, and $D_1, \ldots, D_k$ are the introduced premisses, which are all of form

$$\forall x_1 \cdots x_{n_j}\ X_j(x_1, \ldots, x_{n_j}) \leftrightarrow F_j(x_1, \ldots, x_{n_j}),\ \text{for } 1 \leq j \leq k.$$

A new symbol X_j can occur in $F_{j'}$ only when $j' > j$, and it cannot occur in Γ. By substituting the X_j away and applying Theorem 1, the proof π can be transformed into a proof π' of $\Gamma, E_1, \ldots, E_k \vdash \bot$ in which each E_j has form

$$\forall x_1 \cdots x_{n_j}\ F(x_1, \ldots, x_{n_j}) \leftrightarrow F(x_1, \ldots, x_{n_j}).$$

These are simple tautologies which can be proven and cut away.

6 Antiprenexing

The purpose of anti-prenexing (also called miniscoping) is to obtain smaller Skolem terms. In many formulas, not everything that is in the scope of a quantifier, does also depend on this quantifier. If one systematically factors such subformulas out of the scope of the quantifier, one can often reduce dependencies between quantifiers. For details, we refer to [4], here we give only a few examples:

Example 1. Without anti-prenexing, $\forall x\ \exists y[\ p(x) \wedge q(y)]$ skolemizes into $\forall x\ [p(x) \wedge q(f(x))]$. Antiprenexing reduces the formula to $(\ \forall x\ p(x)\ ) \wedge (\ \exists y\ q(y)\ )$, which Skolemizes into $(\ \forall x\ p(x)\ ) \wedge q(c)$.

Without anti-prenexing, $\forall x\ \exists y_1 y_2\ [p(y_1) \wedge q(x, y_2)]$ skolemizes into $\forall x\ [\ p(f_1(x)) \wedge q(x, f_2(x))\ ]$. Antiprenexing reduces the formula to $\forall x\ [\exists y_1\ p(y_1) \wedge \exists y_2\ q(x, y_2)]$, which Skolemizes into $\forall x\ [p(c_1) \wedge q(f_2(x))]$.

Without anti-prenexing, $\forall x\ \exists y\ [p(x) \wedge q(y) \wedge r(x)]$ skolemizes into $\forall x[p(x) \wedge q(f(x)) \wedge r(x)]$. Antiprenexing can reduce the formula to $\forall x\ [p(x) \wedge r(x) \wedge \exists y\ q(y)]$, which can be Skolemized into $\forall x\ [p(x) \wedge r(x) \wedge q(c)]$.

As far as we can see, all replacements can be handled by the following 'rewrite system':

$$
\begin{array}{llll}
A \vee B & \Rightarrow B \vee A & A \wedge B & \Rightarrow B \wedge A \\
A \vee (B \vee C) & \Rightarrow A \vee B \vee C & A \wedge (B \wedge C) & \Rightarrow A \wedge B \wedge C \\
\\
\forall x\ (P(x) \wedge Q) & \Rightarrow (\forall x\ P(x)) \wedge Q & \exists x\ (P(x) \wedge Q) & \Rightarrow (\exists x\ P(x)) \wedge Q \\
\forall x\ (P \wedge Q(x)) & \Rightarrow P \wedge \forall x\ Q(x) & \exists x\ (P \wedge Q(x)) & \Rightarrow P \wedge \exists x\ Q(x) \\
\forall x\ (P(x) \vee Q) & \Rightarrow (\forall x\ P(x)) \vee Q & \exists x\ (P(x) \vee Q) & \Rightarrow (\exists x\ P(x)) \vee Q \\
\forall x\ (P \vee Q(x)) & \Rightarrow P \vee \forall x\ Q(x) & \exists x\ (P \vee Q(x)) & \Rightarrow P \vee \exists x\ Q(x) \\
\\
\forall x\ P & \Rightarrow P & \exists x\ P & \Rightarrow P \\
\forall x \forall y\ P(x,y) & \Rightarrow \forall y \forall x\ P(x,y) & \exists x \exists y\ P(x,y) & \Rightarrow \exists y \exists x\ P(x,y)
\end{array}
$$

The system is not a rewrite system in the usual sense, because an additional strategy is needed for deciding when a certain rule should be applied. Straightforward normalization would not terminate due to the presence of permutation rules. If one would remove the permutation rules, one would often not obtain the best possible result. For example, in the last formula of the example, $(p(x) \wedge q(y)) \wedge r(x)$ first has to be permuted into $(p(x) \wedge r(x)) \wedge q(y)$, before the rule $\exists x(P \wedge Q(x)) \Rightarrow P \wedge \exists x\ Q(x)$ can be applied.

Despite the fact that the decision making is more complicated than was the case for the NNF, Algorithm 2 can be still modified for anti-prenexing, because the decision making plays no role in the proof generation. For the proof generation, only correctness of the rules matters, and all rules can be easily proven correct.

7 Proof Reductions

Proof reductions are important, because they make it possible to obtain modularity and flexibility. For a detailed motivation, we refer to [1]. There, a special calculus called *replacement calculus* was introduced which allows for certain reductions that remove repeated building up of the same context in a proof. In sequent calculus, the standard reductions of cut elimination correspond to the reductions of the replacement calculus, so there is no need anymore for the replacement calculus. For the purpose of proof simplification, one should implement all standard reductions of cut elimination (see [3]), except for the permutation of a cut with a contraction, because this permutation is the cause of increasement in proof length.

The proof reductions are needed in order to combine the repeated building up of contexts. Suppose that one has a big formula of form $F[A_1]$, that A_1 is first rewritten into A_2, and after that into A_3. Algorithm 2 lifts a proof of $A_1 \vdash A_2$ to a proof of $F[A_1] \vdash F[A_2]$. After that, it lifts a proof of $A_2 \vdash A_3$ to a proof of $F[A_2] \vdash F[A_3]$, which is then combined, using cut, into a proof of $F[A_1] \vdash F[A_3]$.

However, it would be more efficient to first apply cut on $A_1 \vdash A_2$ and $A_2 \vdash A_3$, resulting in $A_1 \vdash A_3$, and lift this proof to $F[A_1] \vdash F[A_3]$.

Combination of context lifting can be done only if one knows in advance the order in which the replacements will be made, and when they are near to each other. This was the case for the NNF-transformation, and Algorithm 2 makes use of this fact, both for the outside-inside strategy, and for the inside-outside strategy.

If one does not know the order of replacements in advance, then Algorithm 2 will not avoid repeated lifting into the same context. This would be the case for anti-prenexing. In that case, one has to rely on proof reductions. Using the standard reductions of cut elimination, the cut on the top level can be permuted with the rules that build up the context, until it either disappears, or reaches a contraction.

Using proof terms, the reductions can be easily implemented by a rewrite system on proof terms. We give a few examples of the reductions involved, and give the corresponding rewrite rules:

$$\dfrac{\dfrac{\Gamma \vdash \Delta,\ \beta_1{:}A \qquad \Gamma \vdash \Delta,\ \beta_2{:}B}{\Gamma \vdash \Delta,\ \beta{:}A \wedge B} \qquad \dfrac{\Gamma,\ \alpha_1{:}A,\ \alpha_2{:}B \vdash \Delta}{\Gamma,\ \alpha{:}A \wedge B \vdash \Delta}}{\Gamma \vdash \Delta}$$

is replaced by

$$\dfrac{\Gamma \vdash \Delta,\ \beta_2{:}B \qquad \dfrac{\Gamma \vdash \Delta,\ \beta_1{:}A \qquad \Gamma,\ \alpha_1{:}A,\ \alpha_2{:}B \vdash \Delta}{\Gamma,\ \alpha_2{:}B \vdash \Delta}}{\Gamma \vdash \Delta.}$$

The corresponding rewrite rule is

$$\mathrm{cut}(A \wedge B, \mathrm{andleft}(\alpha, \pi, \alpha_1, \alpha_2), \alpha, \mathrm{andright}(\beta, \pi_1, \beta_1, \pi_2, \beta_2), \beta) \Rightarrow$$
$$\mathrm{cut}(B, \mathrm{cut}(A, \pi, \alpha_1, \pi_1, \beta_1), \alpha_2, \pi_2, \beta_2).$$

The following proof fragment

$$\dfrac{\dfrac{\Gamma \vdash \Delta,\ \beta_1{:}P[x := y]}{\Gamma \vdash \Delta,\ \beta{:}\forall x\ P(x)} \qquad \dfrac{\Gamma,\ \alpha_1{:}P[x := t] \vdash \Delta}{\Gamma,\ \alpha{:}\forall x\ P(x) \vdash \Delta}}{\Gamma \vdash \Delta}$$

reduces into

$$\frac{\Gamma \vdash \Delta,\ \beta_1{:}P[x := t] \qquad\qquad \Gamma,\ \alpha_1{:}P[x := t] \vdash \Delta}{\Gamma \vdash \Delta}$$

The corresponding rewrite rule is

$$\mathrm{cut}(\forall x\ P(x), \mathrm{foralllleft}(\alpha, \pi_2, \alpha_1, t), \alpha, \mathrm{forallright}(\beta, \pi_1, \beta_1, y), \beta) \Rightarrow$$

$$\mathrm{cut}(P[x := t], \pi_2, \alpha_1, \pi_1[y := t], \beta_1).$$

8 Conclusions

We have shown that implementing the CNF-transformation with proof generation is possible. We have given a data structure (inspired by [5]) for the represention of sequent calculus proofs, which is concise, and which allows for implementation of proof reductions. We have given a general translation algorithm, based on rewriting, that covers nearly all of the transformations involved.

Proof generation will not be feasible for formulas that are propositionally complex. Such formulas will have exponentially large proofs, (because probably $\mathcal{NP} \neq \text{co-}\mathcal{NP}$.)

References

1. Hans de Nivelle. Extraction of proofs from the clausal normal form transformation. In Julian Bradfield, editor, *Proceedings of the 16th International Workshop on Computer Science Logic (CSL 2002)*, volume 2471 of *Lecture Notes in Artificial Intelligence*, pages 584–598, Edinburgh, Scotland, UK, September 2002. Springer.
2. Hans de Nivelle. Translation of resolution proofs into short first-order proofs without choice axioms. In Franz Baader, editor, *Proceedings of the 19th International Conference on Computer Aided Deduction (CADE 19)*, volume 2741 of *Lecture Notes in Artificial Intelligence*, pages 365–379, Miami, USA, July 2003. Springer Verlag.
3. Jean-Yves Girard, Paul Taylor, and Yves Lafont. *Proofs and Types.* Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1989.
4. 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.
5. Frank Pfenning. Structural cut elimination. In Dexter Kozen, editor, *Proceedings 10th Annual IEEE Symposion on Logic in Computer Science*, pages 156–166. IEEE Computer Society Press, 1995.

An Algorithm for the Retrieval of Unifiers from Discrimination Trees

Hans de Nivelle
CWI, PO BOX 94079, 1090 GB Amsterdam, The Netherlands, nivelle@cwi.nl

Abstract. KEYWORDS: Unification, Discrimination trees, Implementation.

We present a modification of the unification algorithm which is adapted to the extraction of simultaneously unifiable literals from discrimination trees. The algorithm is useful for efficient implementation of binary resolution, hyperresolution, and paramodulation. The algorithm is able to traverse simultaneously more than one discrimination tree and to construct a unifier at the same time. In this way backtracking is reduced.

1. Introduction

Resolution is still one of the most successful techniques for automated theorem proving in classical logic. It was introduced in ([13]). Since its introduction improvement has come from two directions:

Refinements: There are restrictions on the resolution rule which do not destroy completeness. However, a tremendous increase in efficiency can be gained by using these restrictions. The main restrictions are *subsumption*, already present in ([13]), *ordering restrictions* ([6], [17], [5], [10]) and *hyperresolution* ([14]).

Implementation: There has also been a large improvement on the side of implementation. The main problem in the implementation is to efficiently extract unifiable literals from a large clause set. This is necessary for the construction of clauses that can be resolved, and for the retrieval of candidates for the subsumption test. In the literature there are two manners to optimize this process: The method of *discrimination trees* ([16], [8], [2]) and the method of *FPA-indexing* ([15], [8] [4]).

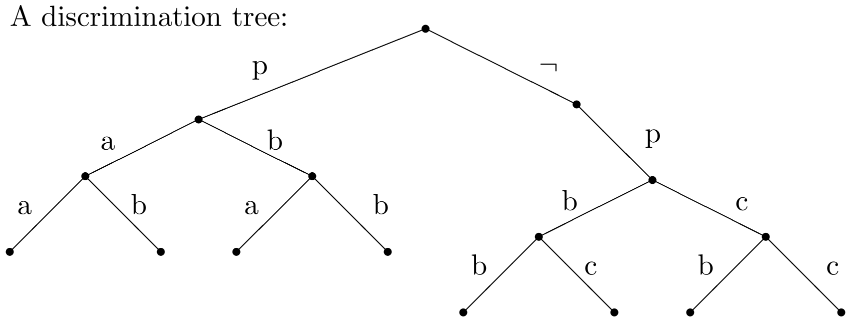
In this paper we will concentrate on the problem of retrieving unifiers for resolution. In order to do this it is necessary to simultaneously extract unifiable literals. For example, in order to construct a binary resolvent it is necessary to extract two literals A_1 and $\neg A_2$ that are unifiable. In the case of hyperresolution it may be necessary to extract m literals $B_1, \ldots, B_m$ that are simultaneously unifiable with literals $A_1, \ldots, A_m$ belonging to a nucleus $\neg A_1, \ldots, \neg A_m, C_1, \ldots, C_n$. The

cost of extracting these literals may be high. For example, for binary resolution, this process may have a worst case complexity of $\mid C \mid ^2$, where $\mid C \mid$ is the size of the clause set. The cost of finding n literals in parallel may be $\mid C \mid ^n$.
Since the clause set may be large, for example > 5000 clauses, this complexity is unacceptable. For this reason it has been necessary to find more efficient ways of extracting unifiable literals.
The two main methods for doing this are FPA-indexing and discrimination tree indexing. In OTTER ([9]) the method of FPA-indexing is used. (Discrimination trees are also used in OTTER, but for the retrieval of possibly subsuming clauses) We will first shortly describe FPA-indexing, then discrimination tree indexing. After that we will motivate our algorithm for the extraction of literals from discrimination trees.
In the FPA-method, literals are characterized by their paths. For example the literal $p(a, s(s(X)))$ has paths
$\{(p), (p, 1, a), (p, 2, s), (p, 2, s, 1, s), (p, 2, s, 1, s, 1, Var)\}$. (Usually variables are not distinguished in the FPA-method, an exception is ([4])). The path $(p, 2, s, 1, s)$ means: The head of the literal is p, the second argument of this p is s and the first argument of s is also s. Given a set of clauses C, it is possible to construct for every path the set of literals in which it occurs. When a unifier is extracted this is done by building an expression, which represents a necessary condition on the paths of the wanted literal. Using this expression potential unifiers are extracted. After they are extracted it is checked whether or not they are real unifiers.
In the method of discrimination trees the literals that have to be accessed are grouped into a tree structure as follows: Write the literals in prefix notation. Then build a tree structure by combining all initial segments of literals that are equal. When one is looking for a literal that unifies with a given literal, one can match the structure of this literal into the tree.
In Example 1.1 a discrimination tree is constructed that contains the literals $p(a, a), p(a, b), p(b, a), p(b, b), \neg\, p(b, b), \neg\, p(b, c), \neg\, p(c, b), \neg\, p(c, c)$. Suppose that we are looking for a literal that unifies with $p(b, a)$. Instead of making possibly 8 comparisons, the literal $p(b, a)$ can be extracted immediately by following the unique path in the discrimination tree that matches $p(b, a)$.
Unfortunately both the tree and the literal for which a unifier is looked for, may contain variables. In general when variables occur in the discrimination tree, or in the literal for which one wants to find a unifying literal, backtracking will be necessary. If for example we want to extract

a literal that unifies with $p(X, X)$ from the tree of Example 1.1, then backtracking will be necessary.
The common way to extract literals from discrimination trees in the presence of variables is by treating variables as dontcares. (See [8], [2]). This means that initially a variable is allowed to match every term. When the path is complete it is checked if the variables are consistently matched (i.e. there is no assignment of different terms to the same variable, and no violation of the occurs check, etc.)
Suppose that we have a tree T containing the terms $p(f(X), X)$ and $p(X, f(X))$. If we try to extract a unifier for $p(a, b)$ there will be no potential candidates. If we try to extract a potential unifier for $p(f(a), a)$, then $p(f(X), X)$ will be returned. If we try to extract a unifier for $p(f(a), b)$ also $p(f(X), X)$ will be returned. If we try to extract a unifier of $p(g(Y), Y)$ then $p(f(X), X)$ will fail because of the clash between f and g. However $p(X, f(X))$ will be extracted as a potential unifier because there is no direct clash between the two terms. Similarly $p(f(a), b)$ will return $p(f(X), X)$.
For this reason it is suggested in the literature ([8]) to unify at the same time as extracting the literal from the tree. This method is more restrictive than the primitive method.
We will present an algorithm for doing this. Moreover our algorithm has another feature, namely that it is well adapted to extracting unifiers simultaneously. We illustrate the advantage of this with an example: Suppose that we want to extract two complementary, unifiable literals from the tree of Example 1.1. Usually this is done by first selecting one literal, and then using the discrimination tree to retrieve a literal that is unifiable with the first literal. This means in the example below that 4 literals will be retrieved initially, of which only one results in a complementary pair.
It is however in principle possible to scan the left and the right subtree in parallel. If we look for two complementary literals, then we will enter the left tree for the positive literal, and skip the $\neg$-symbol on the right for the negative literal. After that there is only one choice at each moment. We will present a unification algorithm which is able to exploit this type of determinism. It will simultaneously backtrack in n-discrimination trees and construct the unifier at the same time. At every moment it will try to read from the tree which has the least number of alternatives. In this way it will obtain as much as possible determinism.
Our approach will be the following: We will try to see the discrimination trees as branching tapes, and the retrieval algorithm as a usual unification algorithm which runs on a non-deterministic Turing machine with branching tapes. It will non-deterministically read the input tapes and

construct a unifier. When it has read a complete literal from each tape it has successfully retrieved unifiable literals. In principle this algorithm can be encoded on a deterministic computer by careful programming. Unfortunately the standard unification algorithm has a feature which destroys the effect that we want: Because of the occurs check it is not always possible to read the input tapes in parallel. For the occurs check it may be necessary to read a complete term from the input and this enforces serial reading instead of the simultaneous reading that we want. We will solve this problem by postponing the moment at which the occurs check is made.
So the plan of the rest of the paper is as follows: First we give a usual unification algorithm, but adapted for prefix terms. Then we adapt this algorithm such that it will be able to make all its decisions by seeing only the next character on the input tapes. (This could be characterized as making the algorithm lookahead(1)). This algorithm has a complete freedom to choose from which of the tapes it will read its next input symbol. After that we show how this algorithm can be transformed into a backtracking algorithm on a deterministic Turing machine.

EXAMPLE 1.1.
A discrimination tree:



2. Fundamentals

In this section we introduce some fundamental notions which will be used later in the paper.

DEFINITION 2.1. We assume that there is a fixed set of variables V, and a fixed set of function symbols F. *Constants* are 0-ary function symbols. We assume that each function symbol has a fixed arity. We write $\#f$ for the arity of f. If v is a variable, then we define $\#v =$

0. A *term* is obtained by applying finitely often one of the following construction rules:

1. A variable is a term.
2. If $t_1, \dots, t_n$ are terms, f is an n-ary function symbol, then $f(t_1, \dots, t_n)$ is a term.

Because of our emphasis on implementation, and the fact that from the implementational point of view terms, literals and atoms have the same structure, we will not be careful in distinguishing them.
In order to be able to deal with discrimination trees we will use the prefix notation of literals. In our opinion the prefix notation is underestimated in automated theorem proving. Left/right traversal of a literal is extremely easy to implement for literals in prefix notation. It is true that direct access of subterms is expensive for literals in prefix notation, but direct access of subterms is hardly ever necessary in resolution theorem proving. In fact the only place that we know is the computation of the recursive path order, where a multiset comparison on subterms has to be made. Another advantage of the prefix representation is the absence of pointers. Pointers may occupy a significant amount of the memory that is occupied by a term.
The advantage of the prefix representation is also observed in ([2]). There literals are stored in prefix notation in a linked list with pointers backward and forward, and with additional pointers to the ends of subterms. The reason that the pointers to subterms are kept in ([2]) is that in the context of Knuth-Bendix completion replacements may be necessary. Also there may be associative and commutative operators in the context of term rewriting. We think however that simple resolution without associative and commutative operators is best implemented with literals in prefix notation.

DEFINITION 2.2. Let t be a term or literal. The *prefix* representation of t is obtained as follows:

1. If t is a variable then the prefix representation of t equals t.
2. If t has form $f(t_1, \dots, t_n)$, then the prefix representation of t equals the concatenation of f and the prefix representations of t_1 up to t_n.

EXAMPLE 2.3. The following literals have the following prefix representations: The literal $p(X, s(Y))$ has prefix representation (p, X, s, Y). The literal $p(a, f(a, b))$ has prefix representation (p, a, f, a, b). Finally $s(0) < s(s(0))$ has prefix representation $(<, s, 0, s, s, 0)$.

It is for the prefix notation that we imposed the condition that the arity of an operator is fixed. There is a 1-1 correspondence between the set of terms and the set of prefix representations, so the prefix representation can be really used as a method of storing terms.

DEFINITION 2.4. A *substitution* is a set Θ of the form $\Theta = \{X_1 := t_1, \dots, X_n := t_n\}$, with $X_i \neq t_i$, and if $X_i = X_j$, then $t_i = t_j$. Θ should be read as a prescription to simultaneously replace in a literal all X_i by t_i. We write $A\Theta$ for the result of applying Θ on A. The *composition* of two substitutions $\Sigma_1 \cdot \Sigma_2$ is defined as $\{V := V\Sigma_1\Sigma_2 \mid V \neq V\Sigma_1\Sigma_2\}$. Literal A is an *instance* of literal B, if there is a substitution Θ, such that $B\Theta = A$. Two literals A and B are *equivalent* if A is an instance of B, and B is an instance of A.

For the composition of two substitutions Θ_1 and Θ_2 holds for all literals A, $A(\Theta_1 \cdot \Theta_2) = (A\Theta_1)\Theta_2$.

DEFINITION 2.5. Let $A_1, \dots, A_m$ and $B_1, \dots, B_m$ be sequences of literals. A *unifier* is a substitution Θ such that for all $i, A_i\Theta = B_i\Theta$. A *most general unifier* of $A_1, \dots, A_m, B_1, \dots, B_m$ is a substitution that satisfies the following conditions:

1. Θ is a unifier of $A_1, \dots, A_m, B_1, \dots, B_m$.
2. For every unifier Σ of $A_1, \dots, A_m, B_1, \dots, B_m$ there is a substitution Ξ, such that $\Sigma = \Theta \cdot \Xi$.

In practice there is a technical difficulty that different B_j come from different clauses, and that therefore occurrences of the same variable in different B_j should be treated as different variables. We will solve this problem by representing variables as a pair $(term, var)$.

Two equivalent literals behave completely the same when it is attempted to unify them with another literal. Since we are interested in extracting unifiers and unifiable literals, we should not distinguish equivalent literals in a discrimination tree. For this reason we need a canonical element of each equivalence class of equivalent literals:

DEFINITION 2.6. Let $(a_1, \dots, a_n)$ be the prefix representation of a literal. We call $(a_1, \dots, a_n)$ *normalized* if every occurrence of a variable V_{i+1} is preceded by a variable V_i. A literal A is normalized if its prefix representation is normalized.

In this definition we assume that there is a natural enumeration $V_0, V_1, \dots$ of the variables. We will store only normalized literals in a discrimination tree. We will store the renaming with which the real literal can be obtained in the leafs of the discrimination tree.

EXAMPLE 2.7. The following literals have the following normalizations: $p(V_2)$ has normalization $P(V_0)$, the literal $q(V_2, V_1, V_0)$ has normalization $q(V_0, V_1, V_2)$, the literal $q(V_1, V_2, V_1)$ has normalization $q(V_0, V_1, V_0)$, the literal $q(V_1, V_2, V_2)$ has normalization $q(V_0, V_1, V_1)$, and the literal $q(V_1, V_1, V_1)$ has normalization $q(V_0, V_0, V_0)$.

We formally define discrimination trees:

DEFINITION 2.8. A *discrimination tree* is a finite tree, such that

1. the branches of T are labelled with function symbols and variables, such that each path of T is labelled with the prefix representation of a normalized literal.

2. The end nodes of T are labelled with *nonempty* sets of references of the form $\{(\Theta_1, i_1), \ldots, (\Theta_n, i_n)\}$. Here the Θ_j are the substitutions that are necessary to retrieve the unnormalized variant of the literal. The i_j are indices, or references to the place where the literal can be found.

3. Let $(a_0, \ldots, a_i)$ be an initial segment of the prefix notation of a literal. There is only one initial segment of a path of T, that is labelled with $(a_0, \ldots, a_i)$.

In principle empty label sets are harmless but they are a waste, because when they are present it is possible that a path will be constructed which does not result in a unifier.

DEFINITION 2.9. A *retrieval problem* is defined by an ordered pair $(\mathcal{A}, \mathcal{T})$, where

1. $\mathcal{A}$ is a sequence of literals $A_1, \ldots, A_m$, with $m > 0$.

2. $\mathcal{T}$ is a sequence of corresponding discrimination trees $T_1, \ldots, T_m$.

A *solution* $\mathcal{S}$ of the retrieval problem consists of a sequence of literals P_i where each P_i is a path in T_i, together with a most general unifier Θ of $A_1, \ldots, A_m$, and $P_1, \ldots, P_m$.

EXAMPLE 2.10. For example, for binary resolution, the retrieval problem will be defined by $(X, \neg\, X)$, with $S_1 = S_2 =$ the set of literals that are allowed to be resolved upon. For hyperresolution, based on a nucleus
$\{\neg\, A_1, \ldots, \neg\, A_m, C_1, \ldots, C_q\}$, the retrieval problem will be defined by $(A_1, \ldots, A_m)$, and $S_1 = \cdots = S_m =$ the set of literals that are allowed to be resolved upon.

ALGORITHM:

```
procedure  equal($\alpha_1, \alpha_2$)
$\lambda := 1$;  $\rho := 1$
while  $\rho > 0$  do
begin
        if  $\alpha_{1,\lambda} \neq \alpha_{2,\lambda}$ then
                return  $\bot$
        $\rho := \rho + \#\alpha_{1,\lambda} - 1$
        $\lambda := \lambda + 1$
end
return  $\top$.
```

When a solution has been found the corresponding references will be transferred to a procedure that construct the resolvents.

3. Unification with Prefix Terms

The plan was the following: To modify the standard unification algorithm for prefix terms into an algorithm which postpones the occurs check, in order to avoid unwanted restrictions on the order in which the literals are read. In this section we will give the standard unification algorithm for prefix terms. In the next section we will modify it in order to solve the problem with the occurs check.

DEFINITION 3.1. A *range* is a triple $r = (\alpha, \lambda, \rho)$, in which

1. α refers to a literal in prefix representation,
2. λ is an index in α,
3. ρ is the number of terms in the range.

The range (α, λ, ρ) refers to ρ subterms of α, beginning at the index λ. If $\alpha = (<, s, 0, s, s, 0)$, then $(\alpha, 1, 1)$ refers to the complete term, $(\alpha, 3, 2)$ refers to $(0, s, s, 0)$, and $(\alpha, 5, 1)$ refers to $(s, 0)$.
Left/right traversal of terms in prefix representation is very easy. The following algorithm checks if two terms are equal:
The expression $\#\alpha_{1,\lambda}$ means: The arity of the functor on the λ-th position in term α_1. It is not difficult to prove the correctness of this algorithm. The main difficulty is checking that the algorithm maintains the range (α, λ, ρ) correctly. This is seen as follows: Write $\alpha = (\alpha_1, \dots, \alpha_n)$.

Assume that $(\alpha_\lambda, \dots, \alpha_n)$ consists of the concatenation of the prefix representations of ρ terms. Then:

– If α_λ is a variable, then $(\alpha_{\lambda+1}, \dots, \alpha_n)$ consists of $\rho - 1$ terms.

– If α_λ is a function symbol f with arity m and arguments t_i then $(\alpha_\lambda, \dots, \alpha_n)$ consists of a term $f(t_1, \dots, t_m)$, followed by $\rho - 1$ terms. $(\alpha_{\lambda+1}, \dots, \alpha_n)$ consists of m terms, namely $t_1, \dots, t_m$, followed by $\rho - 1$ terms. So $(\alpha_{\lambda+1}, \dots, \alpha_n)$ consists of $\rho + m - 1$ terms.

For unification it is necessary to compare two terms in the context of a substitution. For this it is necessary to maintain stacks of ranges:

DEFINITION 3.2. A *range list* P is a list of ranges $P = (\alpha_1, \lambda_1, \rho_1), \dots, (\alpha_p, \lambda_p, \rho_p)$.
Range list P is *normalized* if either P is empty, or $\rho_p \neq 0$.
We define for a normalized range list $P = (\alpha_1, \lambda_1, \rho_1), \dots, (\alpha_p, \lambda_p, \rho_p)$ the following properties:

1. The *context* of P, written as $T(P)$, equals α_p.

2. The *lookahead* symbol of P, written as $\Lambda(P)$ is the first symbol of the range $(\alpha_p, \lambda_p, \rho_p)$, so $\Lambda(P) = \alpha_{p,\lambda_p}$.

3. The *index* of P, notation $I(P)$, equals λ_p.

The reason that we do not accept that $\rho_p = 0$, is that in that case $(\alpha_p, \lambda_p, \rho_p)$ refers to a range of 0 elements and $\Lambda(P), I(P), T(P)$ are not well-defined.
The stacks are necessary because it may be necessary to follow a substitution. If for example we are processing a term (p, X, Y), and there is an assignment $X := (s, 0)$ then at the moment we reach the position of X, we must start processing $(s, 0)$, and after that return to (p, X, Y).

DEFINITION 3.3. A substitution will be represented as a sequence of objects of the form $(a_1, v_1) := (\overline{a}_1, \overline{l}_1), \dots, (a_s, v_s) := (\overline{a}_s, \overline{l}_s)$.

The substitution should be interpreted as the composition $\Theta_1 \cdot \ldots \cdot \Theta_q$. Not all substitutions can be represented in this manner, but all idempotent substitutions can, and this is sufficient for our purpose.
The simple assignment $(a_i, v_i) := (\overline{a}_i, \overline{l}_i)$ means that variable v_i originating from literal a_i should be assigned the literal starting at $\overline{a}_{i,\overline{l}_i}$.
Variables have to be represented by pairs (literal, var), because we consider occurrences in different literals of the same variable as different variables.
We give algorithms to make a step in a range list, to normalize a range list, and to follow a substitution: The additional step in normalize is

ALGORITHM:

```
procedure step($P$)
Write $P$ as $(\alpha_1, \lambda_1, \rho_1), \dots, (\alpha_p, \lambda_p, \rho_p)$
return $(\alpha_1, \lambda_1, \rho), \dots, (\alpha_p, \lambda_p + 1, \rho + \#\alpha_{p,\lambda_p} - 1)$
```

```
procedure normalize($P$)
Write $P$ as $(\alpha_1, \lambda_1, \rho_1), \dots, (\alpha_p, \lambda_p, \rho_p)$
if $P = ()$ then
        return $P$
else
        if $\rho_p = 0$ then
        begin
                $P := (\alpha_1, \lambda_1, \rho_1), \dots, (\alpha_{p-1}, \lambda_{p-1}, \rho_{p-1})$; $p := p - 1$
                $P :=$ step($P$)
                $P :=$ normalize($P$)
        end
```

```
procedure followsubst($P, \Theta$)
Write $\Theta$ as $[(a_1, v_1) := (\overline{a}_1, \overline{l}_1)], \dots, [(a_s, v_s) := (\overline{a}_s, \overline{l}_s)]$
for $i := 1$ to $s$ do
        if $T(P) = a_i \wedge \Lambda(P) = v_i$ then
                return followsubst($P \cdot (\overline{a}_i, \overline{l}_i, 1), \Theta$)
return $P$
```

necessary because the range $(\alpha_p, \lambda_p, \rho_p)$ represents the value of the variable on position $\alpha_{p-1,\lambda_{p-1}}$. If $(\alpha_p, \lambda_p, \rho_p)$ is finished then this variable should be skipped. In followsubst both the variable, and the literal have to be compared because when a variable occurs in different literals these occurrences should be treated as different variables.
We will give an algorithm which constructs the result of a substitution to illustrate the use of range lists. The function processvar is a function that assigns distinct variables to different pairs (literal, variable):
For the unification algorithm we need two more components. They are both straigthforward. The occurs check has the same structure as substitute. Procedure occurs checks if the variable v in the context of literal a occurs in the literal starting at the μ-th position of literal β. Procedure unify extends substitution Θ to a unifier of α and β if this

ALGORITHM:

```
procedure  substitute($\alpha, \Theta$)
$P := (\alpha, 1, 1)$
$\beta := ()$;  $j := 1$
while  $P \neq ()$  do
begin
if  $\Lambda(P)$ is a variable then
        $\beta_j :=$ processvar($T(P), \Lambda(P)$)
        $j := j + 1$
else
        $\beta_j := \Lambda(P)$;  $j := j + 1$
        $P :=$ step($P$)
        $P :=$ normalize($P$)
        $P :=$ followsubst($P, \Theta$)
end
        $j := j - 1$
return  $\beta$
```

is possible. Otherwise it returns $\perp$

4. The Adapted Unification Algorithm

Now we will adapt the unification algorithm such that it will make its decisions by seeing only the next character on the input tape. This is necessary because the occurs check may force the algorithm to read a complete term, which may destroy the possibility of reading terms simultaneously.
Suppose that we want to retrieve two literals matching (X, X) from two discrimination trees T_1 and T_2. In the initial situation the unification algorithm will start with the equations $X =$ a path of T_1, and $X =$ a path of T_2. In order to be able to make the assignment $X := T_1$, it is necessary to check that X does not occur in T_1. This means that the algorithm has to read a complete literal from T_1, and that the possibility of being able to read T_1 and T_2 simultaneously is lost.

ALGORITHM:

```
procedure occurs($a, v, \beta, \mu, \Theta$)
$Q := (\beta, \mu, 1)$
while $Q \neq ()$ do
begin
        if $a = T(Q) \wedge v = \Lambda(Q)$ then
                return $\top$
        $Q :=$ step($Q$)
        $Q :=$ normalize($Q$)
        $Q :=$ followsubst($Q, \Theta$)
end
return $\perp$
```

```
procedure skipterm($P$)
Write $P = (\alpha_1, \lambda_1, \rho_1), \ldots, (\alpha_p, \lambda_p, \rho_p)$
$\rho := \rho_p - 1$
while $\rho < \rho_p$ do
begin
        $\rho_p := \rho_p + \#\alpha_{p,\lambda_p} - 1$
        $\lambda_p := \lambda_p + 1$
end
return $P$
```

We will postpone the occurs check by adapting followsubst. When a substitution Θ violates the occurs check there is a variable v for which $v\Theta$ equals a compound term containing v. This can be detected by procedure followsubst, because it has to enter a term for variable v, which it has already entered before. When followsubst encounters a variable which can be substituted, it will check the range list if it is not already reading a subterm caused by the same variable. If this happens the substitution violates the occurs check and should be rejected. Procedure followsubst will return $\perp$ if it notices that the occurs check fails:

The unification algorithm has to be adapted as follows: When it encounters an equation $V = t$, it will add $V := t$ to the substitution, as usual. But it cannot skip the term t as the standard algorithm would do, because then the occurs check will not be made. Instead it

ALGORITHM 3:

```
procedure  unify($\alpha, \beta, \Theta$)
$P := (\alpha, 1, 1)$;  $Q := (\beta, 1, 1)$
while  $P \neq ()$  do
begin
        if  $\Lambda(P)$ is a variable then
        begin
                if  $\Lambda(Q)$ is a variable then
                begin
                        if  $\Lambda(P) \neq \Lambda(Q) \vee T(P) \neq T(Q)$ then
                                $\Theta := \Theta \cdot [(T(P), \Lambda(P)) := (T(Q), I(Q))]$
                        step($P$);  step($Q$)
                end
                else
                begin
                        if  occurs($T(P), \Lambda(P), T(Q), I(Q), \Theta$)
                                return  $\perp$
                        $\Theta := \Theta \cdot [(T(P), \Lambda(P)) := (T(Q), I(Q))]$
                        $P :=$ skipterm($P$);  $Q :=$ skipterm($Q$)
                end
        end else
        begin
                if  $\Lambda(Q)$ is a variable then
                begin
                        if  occurs($T(Q), \Lambda(Q), T(P), I(P), \Theta$)
                                return  $\perp$
                        $\Theta := \Theta \cdot [(T(Q), \Lambda(Q)) := ((T(P), I(P))]$
                        $P :=$ skipterm($P$);  $Q :=$ skipterm($Q$)
                end
                else
                begin
                        if  $\Lambda(P) \neq \Lambda(Q)$ then
                                return  $\perp$
                        $P :=$ step($P$);  $Q :=$ step($Q$)
                end
        end
        $P :=$ normalize($P$);  $Q :=$ normalize($Q$)
        $P :=$ followsubst($P$);  $Q :=$ followsubst($Q$)
end
return  $\Theta$
```

ALGORITHM:

```
procedure  followsubst$(P, \Theta)$
Write $P$ as $(\alpha_1, \lambda_1, \rho_1), \ldots, (\alpha_p, \lambda_p, \rho_p)$.
Write $\Theta$ as $(a_1, v_1) := (\overline{a}_1, \overline{l}_1), \ldots, (a_s, v_s) := (\overline{a}_s, \overline{l}_s)$

for  $i := 1$  to  $s$  do
begin
        if  $a_i = T(P) \wedge v_i = \Lambda(P)$ then
        begin
                for  $j := 1$  to  $p - 1$  do
                begin
                        if  $\alpha_j = T(P) \wedge \alpha_{j,\lambda_j} = \Lambda(P)$ then
                        return  $\perp$
                end
                $P := P \cdot (\overline{a}_i, \overline{l}_i, 1)$
                return  followsubst$(P, \Theta)$
        end
end
```

will make the substitution $V := t$ on the left side, which results in the tautologeous equation $t = t$, and then scan the terms t.

Algorithm 4 is simpler than Algorithm 3, but also less efficient. The condition in the second if-statement should be interpreted as follows: For two variables a necessary condition for equality is that they originate from the same literal. For functional symbols this condition is not present. Therefore two equal objects can be considered equal by the algorithm if they are functional, or variables originating from the same literal.

5. The Algorithm for Retrieval

The retrieval algorithm will start with m discrimination trees T_i and m literals for which a unifier has to be retrieved. It will construct literals $\alpha_1, \ldots, \alpha_m$, which are selected from the T_i and a unifier Θ. We will use recursion for backtracking.

ALGORITHM 4:

```
procedure unify($\alpha, \beta, \Theta$)
$P := (\alpha, 1, 1)$
$Q := (\beta, 1, 1)$
while $P \neq ()$ do
begin
        $P :=$ followsubst($P, \Theta$)
        $Q :=$ followsubst($Q, \Theta$)
        if $P = \perp \vee Q = \perp$ then
                return $\perp$
        if $\Lambda(P) = \Lambda(Q) \wedge (T(P) = T(Q) \vee \Lambda(P)$ is not a variable) then
                $P :=$ step($P$); $Q :=$ step($Q$)
        else
        if $\Lambda(P)$ is a variable then
                $\Theta := \Theta \cdot [(T(P), \Lambda(P)) := (T(Q), I(Q))]$
        else
                if $\Lambda(Q)$ is a variable then
                        $\Theta := \Theta \cdot [(T(Q), \Lambda(Q)) := (T(P), I(P))]$
                else
                        return $\perp$
        $P :=$ normalize($P$); $Q :=$ normalize($Q$)
end
return $\Theta$
```

DEFINITION 5.1. A *partial literal* is a sequence $(a_1, \ldots, a_n)$, such that $(a_1, \ldots, a_{n-1})$ can be extended to the prefix notation of a literal. a_n is a special character $\perp$ to indicate the end.

Examples are $(p, X, \perp)$, and $(<, s, s, 0, s, \perp)$. Algorithm 5 will start with all $\alpha_i = (\perp)$, i.e. a completely unread literal. During the process it will try to extend the α_i to complete literals. The parameters have the following meanings: $(A_1, \ldots, A_m)$ are the literals that should be unified. $(T_1, \ldots, T_m)$ are the discrimination trees with which the corresponding A_i are to be unified. $(\alpha_1, \ldots, \alpha_m)$ are the selections of the T_i that will be constructed. $(P_1, \ldots, P_m)$ and $(Q_1, \ldots, Q_m)$ are range lists. Θ is the substitution. Procedure retrieve should be called initially as follows:
retrieve$((A_1, \ldots, A_m), (T_1, \ldots, T_m), ((\perp), \ldots, (\perp)),$
$((A_1, 1, 1), \ldots, (A_m, 1, 1)), ((\alpha_1, 1, 1), \ldots, (\alpha_m, 1, 1)), ())$.
The P_i start at the beginning of the A_i, and the Q_i start at the beginning of the α_i. Θ is initially empty. Procedure retrieve backtracks by means of recursion. At each time the algorithm will try to process the

ALGORITHM 5:

```
procedure  retrieve($(A_1, \ldots, A_m), (T_1, \ldots, T_m), (\alpha_1, \ldots, \alpha_m),$
                     $(P_1, \ldots, P_m), (Q_1, \ldots, Q_m), \Theta)$
for  $i := 1$  to  $m$  do
    while  $P_i \neq () \wedge \Lambda(P_i) \neq \perp \wedge \Lambda(Q_i) \neq \perp$  do
    begin
        $P_i :=$ followsubst$(P_i, \Theta)$;  $Q_i :=$ followsubst$(Q_i, \Theta)$
        if  $P_i = \perp \vee Q_i = \perp$ then
            return
        if  $\Lambda(P_i) = \Lambda(Q_i) \ \wedge (T(P_i) = T(Q_i) \vee \Lambda(P_i)$ is not a variable) then
            $P_i :=$ step($\mathrm{P_i}$);  $Q_i :=$ step($\mathrm{Q_i}$);
        else
            if  $\Lambda(P_i)$ is a variable then
                $\Theta := \Theta \cdot [((T(P_i), \Lambda(P_i)) := (T(Q_i), I(Q_i))]$
            else
                if  $\Lambda(Q_i)$ is a variable then
                    $\Theta := \Theta \cdot [(T(Q_i), \Lambda(Q_i)) := (T(P_i), I(P_i))]$
                else
                    return
        $P_i :=$ normalize$(P_i)$;  $Q_i :=$ normalize$(Q_i)$
    end
if for all $i$,  $P_i = ()$ then
    presentsolution$(\alpha_1, \ldots, \alpha_m, \Theta)$;  return
choose the best $k$ with $1 \leq k \leq m$, for which $\alpha_k$ is not completely read.
for the possible manners of extending $\alpha_k$ do
begin
    $f :=$  the first/the next possible extension of $\alpha_k$.
    retrieve$((A_1, \ldots, A_m), (T_1, \ldots, T_m), (\alpha_1, \ldots, \alpha_k \cdot (f), \ldots, \alpha_m),$
             $(P_1, \ldots, P_m), (Q_1, \ldots, Q_m), \Theta)$
end
```

terms as far as possible. When it is not possible to continue, either the end is reached, in that case retrieve will give the solutions, or it has processed all terms as far as they are read, and in that case it will try to extend one of the partial literals.
In order to decide which α_k to extend we use a heuristic. If $\Lambda(P_i) = \perp$, and $T(P_i) = \alpha_k$, then α_k is a candidate for extension. (The same if $\Lambda(Q_i) = \perp$, and $T(Q_i) = \alpha_k$). Our choice strategy is based on the following two observations:

1. Suppose it is the case that $\Lambda(P_i) = \perp$, and $T(P_i) = \alpha_k$. Now if $\Lambda(Q_i)$ is a function symbol, this is more restrictive than when $\Lambda(Q_i)$ is a variable, because unification of the term on α_k with a variable will always succeed, while unification of the term on α_k with a term with a known functor has a chance to fail.
2. The α_k which has the least number of branches with variables should be preferred, because variables will unify with everything, so we can expect the least number of choices from the α_k which has the least number of variables on branches.

Based on these observations, weights can be assigned to the α_k as follows:

- If $\Lambda(P_i) = \perp$, and $T(P_i) = \alpha_k$, and $\Lambda(Q_i)$ is a function symbol or a constant, then $W_P(\alpha_k, i) = 50$.
- If $\Lambda(P_i) = \perp$, and $T(P_i) = \alpha_k$, and $\Lambda(Q_i)$ equals $\perp$, or a variable, then $W_P(\alpha_k, i) = 30$.
- If $\Lambda(Q_i) = \perp$, and $T(Q_i) = \alpha_k$, and $\Lambda(P_i)$ is function symbol or a constant, then $W_Q(\alpha_k, i) = 50$.
- If $\Lambda_(Q_i) = \perp$, and $T(Q_i) = \alpha_k$, and $\Lambda(P_i)$ equals $\perp$ or a variable, then $W_Q(\alpha_k, i) = 30$.
- If $\Lambda(P_i) = \perp$, and $T(P_i) = \alpha_k$, then $V_P(\alpha_k, i) = -$ the number of extensions of α_k that are labelled with a variable.
- If $\Lambda(Q_i) = \perp$, and $T(Q_i) = \alpha_k$, then $V_Q(\alpha_k, i) = -$ the number of extensions of α_k that are labelled with a variable.

Then for every α_k the total weight can be computed as follows:

$$\sum_{i \in \{1,\ldots,m\}} W_P(\alpha_k, i) + W_Q(\alpha_k, i) + V_P(\alpha_k, i) + V_Q(\alpha_k, i).$$

Using this Algorithm 5 can extend the α_k with the highest weight.

6. Complexity

The complexity situation is as follows: The task of constructing all solutions of a retrieval problem is provably exponential, because the number of solutions may be exponentially large. The question of deciding whether or not at least one solution exists is NP-complete, (Theorem 6.2) hence there is no hope for an algorithm that is polynomial in the number of solutions.

We compare Algorithm 5 to non-simultaneous extraction. (Called the one-by-one method in the sequel) There exist cases where the one-by-one strategy is better (Theorem 6.5), and cases where the strategy of Theorem 5 is better (Theorem 6.4). The situation is typical for NP-complete problems where one rarely finds an algorithm that is optimal in all cases.
We need the following definition, before proving NP-completeness:

DEFINITION 6.1. We write $T(\{A_1, \ldots, A_n\})$ for the discrimination tree, that contains the literals $A_1, \ldots, A_n$.

THEOREM 6.2. *Let $(\mathcal{A}, \mathcal{T})$ be a retrieval problem.*

1. *The decision problem of deciding whether or not $(\mathcal{A}, \mathcal{T})$ has at least one solution, is NP-complete.*

2. *The problem of enumerating the solutions is NP-hard.*

Proof. It is clear that (1) $\Rightarrow$ (2). (1) can be proven from the NP-completeness of the subsumption problem, but we prefer to give a direct transformation from the $3SAT$problem.
Let $\mathcal{I} = \{\{\gamma_{1,1}, \gamma_{1,2}, \gamma_{1,3}\}, \ldots, \{\gamma_{m,1}, \gamma_{m,2}, \gamma_{m,3}\}\}$ be a $3SAT$ instance. The $\gamma_{i,j}$ are propositional literals: They are equal to a propositional atom, or the negation of a propositional atom. The $\{\gamma_{i,1}, \gamma_{i,2}, \gamma_{i,3}\}$ are clauses, i.e. disjunctions. Let $At = \{\alpha_1, \ldots, \alpha_n\}$ be the set of propositional atoms in $\mathcal{I}$. We construct a retrieval problem $(\mathcal{A}, \mathcal{T})$, such that $(\mathcal{A}, \mathcal{T})$ has a solution if and only if $\mathcal{I}$ has a model.

1. Assign to every propositional atom α_i a fresh variable V_i, and a fresh symbol a_i. For every literal $\gamma_{i,j}$, $V(\gamma_{i,j})$ is the variable that corresponds to the atom from which $\gamma_{i,j}$ is built.

2. Assign to every clause in $\mathcal{I}$ a fresh symbol c_i. $U(i, \gamma_{i,1}, \gamma_{i,2}, \gamma_{i,3})$ is defined as follows: Let

$$\Sigma_i = \{c_i(f,f,f), c_i(f,f,t), c_i(f,t,f), c_i(f,t,t),$$

$$c_i(t,f,f), c_i(t,f,t), c_i(t,t,f), c_i(t,t,t)\}$$

   be the set of all possible truth value assignments to 3 propositional atoms. The set Σ_i' is obtained by deleting that literal (exactly one) from Σ_i that is not consistent with the clause $\{\gamma_{i,1}, \gamma_{i,2}, \gamma_{i,3}\}$. Then $U(i, \gamma_{i,1}, \gamma_{i,2}, \gamma_{i,3})$ equals $T(\Sigma_i')$.

Construct:

$$\mathcal{A} = (a_1(V_1), \ldots, a_n(V_n),$$

$$c_1(V(\gamma_{1,1}), V(\gamma_{1,2}), V(\gamma_{1,3})), \ldots, c_m(V(\gamma_{m,1}),\ V(\gamma_{m,2}),\ V(\gamma_{m,3}))\ \ ),$$

and

$$\mathcal{T} = (T(\{a_1(f), a_1(t)\}), \ldots, T(\{a_n(f), a_n(t)\}),$$
$$U(1, \gamma_{1,1}, \gamma_{1,2}, \gamma_{1,3}), \ldots, U(m, \gamma_{m,1}, \gamma_{m,2}, \gamma_{m,3})\ \ ).$$

Now assume that $\mathcal{I}$ has a solution, i.e. an interpretation I that makes all the clauses $\{\gamma_{i,1}, \gamma_{i,2}, \gamma_{i,3}\}$ true. We define a solution of the retrieval problem from $\Theta = \{V_1 := w_1, \ldots, V_n := w_n\}$, where $w_i = t$ if I interprets α_i as t, and $w_i = f$ if I interprets α_i as f. Clearly each $a_i(V_i)$ can retrieve a branch in $T(\{a_i(f), a_i(t)\})$. Each $c_i(V_(\gamma_{i,1}), V(\gamma_{i,2}), V(\gamma_{i,3}))$ must be retrievable from $U(i, \gamma_{i,1}, \gamma_{i,2}, \gamma_{i,3})$, because I satisfies the i-th clause of $\mathcal{I}$.
Now assume that the retrieval problem has a solution. Let Θ be the most general unifier that belongs to the solution. Θ must be of the form $\{V_1 := w_1, \ldots, W_n := w_n\}$, where each $w_i \in \{f, t\}$, because each $a_i(V_i)$ must be retrieved from $T(\{a_i(f), a_i(t)\})$. Define an interpretation I for each propositional atom α_i, from $I(\alpha_i) = w_i$. Because each $U(i, \gamma_{i,1}, \gamma_{i,2}, \gamma_{i,3})$ contains the truth values that are compatible with the i-th clause, every clause of $\mathcal{I}$ must be true in I. This completes the transformation.
It remains to show that the problem is in NP. In order to do this it is sufficient to show that the problem of deciding whether or not two sequences of literals can be unified, is in P. This was proven in ([12]), but for the sake of completeness we outline the proof in the following lemma.

LEMMA 6.3. *The following transformation rules on equations define an algorithm that can decide in polynomial time (in the total size of the literals), whether or not two sequences of literals* $(A_1, \ldots, A_n)$, *and* $(B_1, \ldots, B_n)$ *have a unifier:*

start *The initial set of equations equals* $\{A_1 = B_1, \ldots, A_n = B_n\}$.

sym *If there is an equation* $t_1 = t_2$, *then add* $t_2 = t_1$.

decomp *If there is an equation of the form* $p(t_1, \ldots, t_m) = p(u_1, \ldots, u_m)$, *then add the equations* $t_1 = u_1, \ldots, t_m = u_m$.

trans *If there are equations of the form* $x = t_1$, *and* $x = t_2$, *where* x *is a variable, then add* $t_1 = t_2$.

Let $\overline{E}$ *be the closure of the initial set of equations, under the rules* **sym**, **decomp** *and* **trans**. *There are the following end conditions:*

structdif *If there is an equation of the form* $p(t_1, \ldots, t_m) = q(u_1, \ldots, u_n)$, *in* $\overline{E}$, *where either* $p \neq q$, *or* $m \neq n$, *then there is no unifier.*

occurs *If the set of equations $\overline{E}$ violates the occurs check, (i.e. $x = t(x)$ follows by transitivity and substitutivity from $\overline{E}$), for a variable x and a term which has x as a subterm), then there is no unifier.*

yes *In the remaining case there is a unifier.*

Proof. The process of constructing $\overline{E}$ is polynomial. All equations consist purely of subterms of the initial literals A_i and B_j. The number of possible subterms is equal to the size of the A_i and B_j, and the number of possible equations is hence quadratic. Because of this the closure $\overline{E}$ can be constructed in polynomial time. After that the end conditions can be checked in polynomial time. The correctness of the rules can be easily checked.

So the problem of deciding whether or not a retrieval problem has a solution is in NP, and hence (stricly seen) it is not known whether or not the problem needs exponential time. However the problem becomes provably exponential as soon as one changes the problem to the task of enumerating all solutions, because the the number of solutions may be exponential. Also in the case that there is only one solution, the size of this solution may be exponentially large.
The fact that the retrieval problem is NP-complete sets the scene for the problems that we can expect when one compares Algorithm 5 to the one-by-one approach. Typically, almost for every pair of algorithms that solve an NP-complete problem, there is an instance on which A_1 performs exponentially better than A_2, and an instance on which A_2 performs exponentially better than A_1.
For example it generally agreed upon that orderered resolution performs better on the SAT-problem, than unordered resolution. However one can find a sequence of SAT-instances on which unordered resolution performs exponentially better than ordered resolution. Our algorithm compares in the same manner with the one-by-one approach: There are instances where the algorithm behaves exponentially better, and there are instances where the algorithm behaves exponentially worse. However it is sensible to prune the search tree by always choosing the branch with the lowest number of alternatives.
We first show that Algorithm 5 can be exponentially better than the one-by-one approach.

THEOREM 6.4. *There exists a sequence of retrieval problems $(\mathcal{A}_n, \mathcal{T}_n)$, with $n > 0$, that have exactly one solution. The one-by-one algorithm will take time $O(2^n)$, if it selects the trees from left to right. Algorithm 5 takes time $O(n)$.*

Proof. Construct the following retrieval problem, for each $n > 0$:

– $\mathcal{A}_n = (p_1(V_1), \ldots, p_n(V_n),\ q_1(V_1), \ldots, q_n(V_n))$.

– $\mathcal{T}_n = (T(\{p_1(a), p_1(b)\}), \ldots, T(\{p_n(a), p_n(b)\}),$
$T(\{q_1(b), q_1(c)\}), \ldots, T(\{q_n(b), q_n(c)\})\ \ )$

The one-by-one algorithm will enumerate all possible matchings for the $p_1(V_1)$, up to $p_n(V_n)$, which takes time $O(2^n)$. Only one of the solutions will pass the test on the q_i.
On the other hand, Algorithm 5, as soon as it makes a choice for one of the p_i, will have a preference to read the corresponding q_i. In this manner it will not backtrack and take only $O(n)$ time.

Unfortunately it is also possible to deceive Algorithm 5. Its weakness lies in the fact that it extends the tree that appears to offer the highest degree of determinism. In doing so it looks only at the next symbol. It is however possible that the tree that appears to be the most deterministic if one looks ahead one symbol, is not if one looks ahead two symbols. Suppose that the tree T contains literal $p(a, a)$, that one tries to retrieve $p(X, b)$, and that the algorithm has read up to the p. Because of the variable X, the algorithm will be reluctant to continue reading this tree, and try to read from another tree, which possibly involves backtracking. However if it would, it would notice that this tree will not lead to a solution of the retrieval problem. On this the following is based:

THEOREM 6.5. *There exists a sequence of retrieval problems $(\mathcal{A}_n, \mathcal{T}_n)$, that has exactly one solution, where the one-by-one algorithm will take time $O(n)$, and Algorithm 5 takes time $O(2^n)$.*

Proof. Construct the following sequence:

– $\mathcal{A}_n = (p_1(a, V_1, V_1), \ldots, p_n(a, V_n, V_n)).$

– $\mathcal{T}_n = (T(\{p_1(a, a, a), p_1(X, a, b)\}), \ldots, T(\{p_n(a, a, a), p_n(X, a, b)\})\ ).$

The problem has only one solution, defined by $\Theta = \{V_i := a \mid 1 \leq i \leq n\}$.
Because of the symmetry it does not matter in which order the one-by-one algorithm selects the trees, so assume that it is from left to right. For each p_i it will find the correct path in $T(\{p_i(a, a, a), p_i(X, a, b)\})$ in at most two guesses, so the total time will be $O(n)$.
On the other hand Algorithm 5 will do the following: In all trees it will prefer reading the first two symbols over reading the third symbol. This is due to the fact that in the $p_i(a, V_i, V_i)$ the first two symbols are constant symbols, and that the third is a variable. As a consequence it will make n choices, giving 2^n possiblities, before trying to match the V_i.

In Theorem 6.4 the superiority of Algorithm 5 depends highly on the order in which the one-by-one approach selects the trees. However we

believe that the idea of trying to postpone the choices with a large number of alternatives, is sound.
Even if one considers Algorithm 5 too complicated, then at least one should pay some attention to the order in which the unifiers are retrieved. It is possible to give an example where the superiority does not depend on the order, but this example is not so impressive, because the time of the one-by-one approach is not exponential. It is an open problem whether or not there exists an example in which Algorithm 5 takes polynomial time, and the one-by-one approach takes exponential time, independent of the order in which the trees are selected.

THEOREM 6.6. *There exists a sequence of retrieval problems* $(\mathcal{A}_n, \mathcal{T}_n)$, *with* $n > 0$, *that have one solution, for which the one-by-one algorithm takes time* $O(\frac{1}{2}n)$ *and Algorithm 5 takes time* $O(\log(n))$.

Proof. Define

$$A_n = T(\{p(0,\dots,0),\dots,p(1,\dots,1)\}),$$

$$B_n = T(\{q(1,\dots,1),\dots,q(2,\dots,2)\}).$$

A_n enumerates all combinations of 0 and 1 of length $\log(n)$. B_n enumerates all combinations of 1 and 2 of length $\log(n)$.
Then the sequence is defined by:

$$\mathcal{A}_n = (A_n, B_n).$$

$$\mathcal{T}_n = (p(X_1,\dots,X_n), q(X_1,\dots,X_n)\ ).$$

The one-by-one algorithm will read either A_n or B_n first. Doing so it will try all the $2^{\log(n)}$ paths, which takes $O(n)$ time.
Algorithm 5 will read $p(X_1,\dots,X_n)$ and $q(X_1,\dots,X_n)$ simultaneously and take $O(\log(n))$ time.

The situation that occured in the proof of Theorem 6.5 cannot occur in the case where one tries to retrieve literals of the form $(p(V_1,\dots,V_n, \neg\, p(V_1,\dots,V_n))$, with the V_i distinct, as was the case in the construction in the proof of Theorem 6.6. In such cases one can expect the highest benefit from Algorithm 5.

7. Conclusions

We have presented a unification algorithm which is able to retrieve multiple literals simultaneously from multiple discrimination trees. One can expect on average that this will reduce the number of paths that have to be tried. This is achieved by using knowledge that has been gained in one discrimination tree immediately in the other trees. This is an improvement over the existing one-by-one algorithms, presented in ([8]), because the algorithms given there extract only one unifier at a time. At this moment the algorithm is being implemented in a resolution based theorem prover.

8. Acknowledgements

This research has been supported by NWO. I would like to thank the Computer Science Department of the Delft University of Technology for the use of their equipment. The paper from which this article is derived ([11]) was made while the author worked at the Catholic University of Tilburg. Thanks also to the anonymous referees, and to R. Pliuskevicius for insisting on a complexity analysis.

References

1. C-L. Chang, R. C-T. Lee, Symbolic Logic and Mechanical Theorem Proving, Academic Press, New York, 1973.
2. J. Christian, Flatterms, Discrimination Nets, and Fast Term Rewriting, Journal of Automated Reasoning 10, pp. 95-113, 1993.
3. G. Gottlob, A. Leitsch, Fast Subsumption Algorithms, EUROCAL 85, LNCS 204, pp. 64-77, 1985.
4. P. Graf, Extended Path-Indexing, CADE 12, Ed. Alan Bundy, pp. 514-528, 1994.
5. W.H. Joyner, Resolution Strategies as Decision Procedures, J. ACM 23 (1), pp. 398-417, 1976.
6. R. Kowalski, P.J. Hayes, Semantic Trees in Automated Theorem Proving, Machine Intelligence 4, ed. B. Meltzer and D. Michie, 1969.
7. D. W. Loveland, Automated Theorem Proving, A Logical Basis, North Holland Publishing Company, Amsterdam, New York, Oxford, 1978.
8. W. McCune, Experiments with Discrimination-Tree Indexing and Path Indexing for Term Retrieval, Journal of Automated Reasoning, Vol. 9, pp. 147-167, 1992.
9. W. McCune, OTTER 3.0 Reference Manual and Guide + source, obtainable from info.mcs.anl.gov, 1994.
10. H. de Nivelle, Resolution Games and Non-Liftable Resolution Orderings, In CSL'94, pp. 279-293, Springer Verlag, 1994.
11. H. de Nivelle, An Algorithm for the Retrieval of Unifiers from Discrimination Trees, in JELIA'96, ed. Peireira and Orlowska, 1996.
12. M.S. Paterson, M.N. Wegman, Linear Unification, J. Comput. System Sci., 16-2, pp. 158-167, 1978.
13. J. A. Robinson, A Machine Oriented Logic Based on the Resolution Principle, Journal of the ACM, Vol. 12, pp. 23-41, 1965.
14. J. A. Robinson, Automated Deduction with Hyperresolution, International Journal of Computer Mathematics 1, pp. 227-234, 1965.
15. M. Stickel, the Path-Indexing Method for Indexing Terms, Technical Note 473, Artificial Intelligence Center SRI International, Menlo Park CA, 1989.

16. L. Wos, A Note on McCune's Article on Discrimination Trees, Journal of Automated Reasoning 9, pp. 145-146, 1992.
17. N.K. Zamov: On a Bound for the Complexity of Terms in the Resolution Method, Trudy Mat. Inst. Steklov 128, pp. 5-13, 1972.