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

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

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.

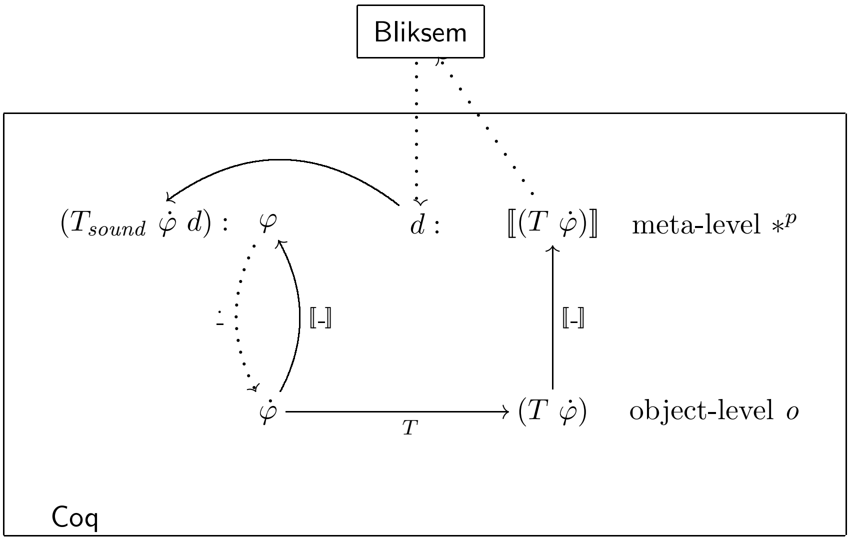

Figure 1. Schematic overview of the general procedure. Arrows correspond to application in Coq, dotted arrows are not performed by Coq. The term $[\![(T\ \dot{\varphi})]\!]$ is computed by Coq and exported to Bliksem. Bliksem is to return a proof term d, which is imported back in Coq. Then $(T_{sound}\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

index set I_L *is defined by the equations:*

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

where $\emptyset$ *is the empty set (i.e. without contructors),* $\mathbb{1}$ *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:*

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

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* $p, q : o,\ \ p' : A \to o,\ \ t_1, \ldots, t_k : A,\ \ \ i : I_{l_{rel}}$ *and* $l_{rel}(i) = k$*.*

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

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* $(\mathcal{R}\ i)$.

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

We use $\wedge, \vee, \exists$ for Coq's predefined logical connectives. Note that '→' (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

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

is not primitive recursive, since $\dot{\neg}p$ and $\dot{\neg}q$ are not subformulae of $\dot{\neg}(p \dot{\wedge} q)$. 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 $nnf : o \to pol \to o$ makes use of the so-called polarity ($\oplus$ or $\ominus$) of an input formula.*

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

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

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.

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

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

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

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.*

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

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 nonempty, 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 $\mathcal{R}$.

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

which indeed follows from the more general:

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

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:

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

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

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

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:*

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

The term $[\![(\mathit{mcf}\ \ \dot{\varphi})]\!]$ computes a format $C_1 \to \cdots \to C_n \to \bot$. 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 ($\rightarrow$,$\forall$) and a typed lambda calculus with product types (with $\rightarrow$ 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:

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

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:

$$C_1 \rightarrow \cdots \rightarrow C_n \rightarrow \bot$$

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:

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

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

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

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

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

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

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

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 $[C] \to [D]$.

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

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

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

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:

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

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 $\dot{\exists}$.

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

We define *foo*.

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

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 $\mathit{taut} = \dot{\varphi}$.)

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

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 $Acc_{\to}(S)$ we then obtain confluence. The induction step we have to prove is that confluence is preserved under the inductive definition of $Acc_{\to}$. In other words, we have to prove that x is confluent if the induction hypothesis (IH) holds, that is, every y such that $x \to y$ 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 $\to$-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.

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

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

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 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.