Optimized Encodings of Fragments of Type Theory in First Order Logic

Tanel Tammet and Jan M. Smith

Department of Computing Science,
Chalmers University of Technology and Univ. of Göteborg,
S-41296 Göteborg, Sweden.
e-mail: {tammet, smith}@cs.chalmers.se

1 Introduction

The subject of this paper is the problem of automated theorem proving in Martin-Löf's monomorphic type theory [17, 8], which is the underlying logic of the interactive proof development system ALF [2, 14].

In the scope of our paper the task of automated theorem proving in type theory is understood as demonstrating that a certain type is inhabited by constructing a term of that type. The problem of inhabitedness of a type A is understood in the following way: given a set of judgements Γ (these may be constant declarations, explicit definitions and defining equalities), find a term a such that $a \in A$ is derivable from Γ. The term a is explicitly constructed and, hence, the judgement $a \in A$ can be checked by ALF.

We consider the question of axiomatizing the problem of inhabitedness of a type as a formula both in the Horn fragment of the standard first-order logic (where classical and intuitionistic logic coincide) and in full first-order intuitionistic logic. Translations of Martin-Löf's type theory to theories based on predicate logic have been considered earlier by Aczel [1] and Smith [19], but with purposes different from automated theorem proving.

Our translation is also similar to the translation given by Felty and Miller [10] of the logical framework LF to the logic hh^{ω} of hereditary Harrop formulas with quantification at all non-predicate types. However, our work differs from theirs in that we consider translations of fragments of type theory into first order logic and that our main interest is in optimizing the translation for enhancing the efficiency of automated proof search for the problem of inhabitedness.

While it is not realistic to expect that fully automated methods will manage most of the hard tasks in theorem-proving, we believe that the automated methods can be used as a powerful tool when developing proofs interactively in a proof system like ALF.

T. Tammet has implemented a resolution-based theorem prover for the fragments F_1 and F_2 described in the paper. The prover is designed for use together with system ALF. It contains also a part for searching structural induction proofs. Structural induction expressed by pattern-matching is an important extension, used in ALF, of the underlying monomorphic type theory [8, 14].

2 Martin-Löf's Type Theory

In type theory we can form judgements of the forms

- A *type*, A is a type,
- $A = B$, A and B are equal types,
- $a \in A$, a is an object in the type A,
- $a = b \in A$, a and b are equal objects in the type A.

In general, a judgement is made in a context, i.e., a list of assumptions $x_1 \in A_1, \ldots, x_n \in A_n$ where for $j \leq n$, A_j may depend on $x_1, \ldots, x_{j-1}$.

There are basically two ways of introducing types in Martin-Löf's type theory: function types and inductively defined sets. Because of the possibility of introducing sets by induction, type theory is an open theory; it is in this sense that the theory may serve as a logical framework.

We denote the type of sets by Set. Given a set A we may form the type $El(A)$ of its elements; hence we have the rule $\dfrac{A \in \mathsf{Set}}{El(A)\ type}$ We will write A instead of $El(A)$, since it will always be clear from the context whether we mean A as a set (i.e., as an object in Set) or as a type.

If A is a type and B is a family of types for $x \in A$, then we can form the type of functions $(x \in A)B$ from A to B.

$$\frac{A\ type \qquad B\ type\ \ [x \in A]}{(x \in A)B\ type}$$

All free occurrences of x in B become bound in $(x \in A)B$. Given a function in $(x \in A)B$ we may apply it on an object in A:

$$\frac{c \in (x \in A)B \qquad a \in A}{c(a) \in B\{a/x\}}$$

where $B\{a/x\}$ denotes the result of substituting a for all free occurrences of x in B. A basic way of forming functions is by abstraction:

$$\frac{b \in B\ \ [x \in A]}{[x]b \in (x \in A)B}$$

A function applied on an object is defined by the ordinary β-rule.

$$\frac{a \in A \qquad b \in B\ \ [x \in A]}{([x]b)(a) = b\{a/x\} \in B\{a/x\}}$$

We also have the usual η-, α- and ξ-rules as well as substitution rules. We will often use the notation $(A)B$ when B does not contain any free occurrences of x. In order to increase the readability, we will write $(x_1 \in A_1; \ldots; x_n \in A_n)B$ instead of $(x_1 \in A_1) \ldots (x_n \in A_n)B$ and $b(a_1, \ldots, a_n)$ instead of $b(a_1) \ldots (a_n)$. Similarly, we will write $[x_1] \ldots [x_n]e$ as $[x_1, \ldots, x_n]e$.

The generality of type theory as a logical framework comes from the possibilities of introducing new constants. It is in this way that we can introduce the usual mathematical objects like natural numbers, functions, tuples etc. as well as sets expressing propositions. There are two kinds of constants: *primitive* and *defined*.

A set is defined by its introduction rules, i.e., by giving a collection of primitive constants with appropriate types. For example, the set of natural numbers is defined by declaring the constants $\mathsf{N} \in \mathsf{Set}$, $\mathsf{succ} \in (\mathsf{N})\mathsf{N}$, $0 \in \mathsf{N}$.

A defined constant can either be *explicitly* or *implicitly* defined. We declare an explicitly defined constant c by giving a definition of it: $c = a \in A$ For instance, we can make the following explicit definitions: $1 = \mathsf{succ}(0) \in \mathsf{N}$, $I_{\mathsf{N}} = [x]x \in (\mathsf{N})\mathsf{N}$, $I = [A, x]x \in (\mathsf{A} \in \mathsf{Set}; \mathsf{A})\mathsf{A}$.

The last example is the monomorphic identity function which when applied to an arbitrary set A yields the identity function on A.

We declare an implicitly defined constant by showing what definiens it has when we apply it to its arguments. An implicit definition may be recursive. The implicit constant $+$, expressing addition of natural numbers, is introduced by $+ \in (\mathsf{N}; \mathsf{N})\mathsf{N}$, $+(0, \mathsf{y}) = \mathsf{y}$, $+(\mathsf{succ}(\mathsf{x}), \mathsf{y}) = \mathsf{succ}(+(\mathsf{x}, \mathsf{y}))$.

The definition of $+$ is an example of an implicit constant defined by pattern-matching on the possible constructors of the set N. In Martin-Löf's original formulation, implicitly defined constants were only possible to introduce by primitive recursion schemes. We will, however, use the more general formulation with pattern-matching, proposed by Coquand [7]. For our approach to automated theorem proving in type theory, pattern-matching is important since it often makes it possible to avoid higher order functions.

A basic idea of type theory is the so called Curry-Howard isomorphism between propositions and sets: a proposition is represented as the set of its proofs. Hence, the type of propositions is identified with the type Set. Variables are used as names of assumptions and constants are used as rules. To apply a rule to a number of subproofs is done by applying a constant to the corresponding subproof objects.

A theory is presented by a list of typings and definitions of constants. When we read the constant as a name of a rule, then a primitive constant is usually a formation or introduction rule, an implicitly defined constant is an elimination rule (with the contraction rule expressed as the step from the definiendum to the definiens) and finally, an explicitly defined constant is a lemma or derived rule.

3 The Resolution Calculus

We will define some standard notions of the resolution method, restricting us to the Horn fragment where classical and intuitionistic provability coincide. For further details see, for example, [6] or [4].

An *atom* is a predicate symbol applied to zero or more terms. A *positive literal* is an atom. A *negative literal* is an atom preceded by the negation sign. A

clause is a finite set of literals. All variables in a clause are interpreted as being universally quantified. In classical logic a clause $\{L_1, L_2, \ldots, L_n\}$ is interpreted as the disjunction $L_1 \vee L_2 \vee \ldots \vee L_n$. A clause, literal, atom or a term is said to be *ground* if it contains no variables. A clause is said to be a *singleton clause* if it only contains a single literal. A *Horn clause* is a clause which contains at most one positive literal. We will often write Horn clauses as *sequents*; a sequent $L_1, \ldots, L_n \Rightarrow L$ is considered to be the same as a clause $\{\neg L_1, \ldots, \neg L_n, L\}$.

New clauses are derived by the rule of hyperresolution

$$\frac{L_1, \ldots, L_n \Rightarrow L \qquad \Rightarrow L'_1 \qquad \ldots \qquad \Rightarrow L'_n}{\Rightarrow L\sigma} \quad \sigma = mgu(L_1, L'_1) \ldots mgu(L_n, L'_n)$$

where $mgu(L, L')$ denote the most general unifier of the terms or literals L and L'.

For a clause set S we define $Res(S)$ as the set of all clauses derived from S by one step of hyperresolution. We define $\mathcal{R}^*$ by $\mathcal{R}^*(S) = \bigcup_i \mathcal{R}^i(S)$, where $\mathcal{R}^0(S) = S$, $\mathcal{R}^{i+1}(S) = \mathcal{R}^i(S) \cup Res(\mathcal{R}^i(S))$. We say that a clause C is *derivable from* a clause set S if $C \in \mathcal{R}^*(S)$.

4 Translating Non-Nested Function Types

In this section we will consider the fragment F_1 of type theory which corresponds to Horn clauses. A *function type* is a type of the form $(x_1 \in A_1; \ldots; x_n \in A_n)B$, where $0 < n$.

Definition *A type C belongs to the type fragment F_1 if either C does not contain function types or C has the form $(x_1 \in A_1; \ldots; x_n \in A_n)B$ where none of A_i and B is or contains a function type.*

We say that a judgement belongs to the judgement fragment F_1 if the type of the judgement belongs to F_1. When we speak about the fragment F_1 in the following, it will always be clear from the context whether we mean the type fragment or the judgement fragment.

Example 1. **1.** The type $(\mathsf{X} \in \mathsf{Set}; \mathsf{x} \in \mathsf{X}; \mathsf{y} \in \mathsf{X})\mathsf{X}$, corresponding to the implication $\mathsf{X} \to (\mathsf{X} \to \mathsf{X})$, is in F_1. The type $(\mathsf{X} \in \mathsf{Set}; \mathsf{x} \in \mathsf{X}; \mathsf{y} \in (\mathsf{z} \in \mathsf{X})\mathsf{X})\mathsf{X}$, corresponding to $\mathsf{X} \to ((\mathsf{X} \to \mathsf{X}) \to \mathsf{X})$ contains a function type $(\mathsf{z} \in \mathsf{X})\mathsf{X}$ in its third argument, thus it is not in the fragment F_1.

4.1 Translating Judgements and the Goal

We use a two-place first-order predicate In and a first-order equality predicate $=$ to translate judgements.

The intended meaning of $In(a, A)$ is that the term a is an element of the set denoted by the term A, i.e., $a \in A$. Each judgement in F_1 is encoded as a clause in the Horn fragment of first-order logic without quantifiers. All the (first-order) variables in a clause are understood as being universally quantified.

– **Application terms.** An application term $a \equiv f(g_1, \ldots, g_n)$ is translated by full uncurrying. So, in case f is a composite term $h(l_1, \ldots, l_k)$, then one step of uncurrying gives a term $h(l_1, \ldots, l_k, g_1, \ldots, g_n)$ for a. For instance, the term $\mathsf{f(x,y)(g(x))}$ is translated as $f(x, y, g(x))$.
  Notice that in the fragment F_1, all the occurrences of a function symbol in the translated terms will have the same number of arguments. The last fact justifies the syntactic correctness of the translation into first-order language without an extra layer of encoding for application terms.
– **Expressions declaring primitive constants.** A primitive constant $f \in (x_1 \in A_1; \ldots; x_n \in A_n)B$ is translated as the clause $In(x_1, A'_1), \ldots, In(x_n, A'_n) \Rightarrow In(f(x_1, \ldots, x_n), B')$, where each x_i is a variable, f is a constant symbol, A'_i and B' are translations of A_i and B, respectively.
  The only variables in the resulting clause are x_i $(1 \leq i \leq n)$. Notice that in the translation of an expression in the fragment F_1, the leftmost symbol of each subterm is a function symbol. This justifies the syntactic correctness of the translation into first-order language.
– **Expressions defining implicit constants.** The definition $f(t_1, \ldots, t_n) = g(h_1, \ldots, h_k)$, of an implicit constant f, is translated as the clause $\Rightarrow F' = G'$, where F' and G' are translations of $f(t_1, \ldots, t_n)$ and $g(h_1, \ldots, h_k)$, respectively.
– **Explicit definitions and equality of types.** Explicit definitions $c = a \in A$ and judgements of the form $A = B$ where A and B are types are not translated at all. Instead, all the type theory expressions containing occurrences of the left side c of some explicit definition or the left side A of an asserted equality of types are normalized by expanding the definition and reducing the resulting redexes before doing the translation.
– **The inhabitedness problem:** Given a set of judgements Γ, show the inhabitedness of a type $G \equiv (x_1 \in A_1; \ldots; x_n \in A_n)B$. We are moreover only interested of the case when B is not the constant Set.
  In order to avoid n explicit abstraction steps in the final part of the derivation, we use the pattern-matching formulation of the abstraction term inhabiting G. Thus we assume that G is a type of an implicitly defined constant g. Our goal is to construct a term t (corresponding to the body of the abstraction term) for the right hand side of the equation $g(x_1, \ldots, x_n) = t$ which defines a function inhabiting G.
  Let $\sigma \equiv \{c_1/x_n, \ldots, c_n/x_n\}$ be a substitution replacing the variables in G by new constant symbols (Skolem constants) $c_1, \ldots, c_n$ not occurring anywhere in G or in any of the judgements in Γ. The goal will be encoded as the problem of deriving a substitution instance of the clause $\Rightarrow In(x, B'\sigma)$ from the set of clauses $\Gamma' \cup E \cup A'$, where B' is the translation of B, Γ' is the set of translations of all the elements of Γ, E is the standard axiomatization of equivalence and substitutivity of equality for Γ', A' and B', and A' is the set of clauses

$$\Rightarrow In(c_1, A'_1\sigma), \ldots, \Rightarrow In(c_n, A'_n\sigma),$$

where each A'_i is a translation of A_i. The operation of replacing the variables in G by the new constants is a specific instance of the Skolemization procedure and, unlike full Skolemization, is correct for intuitionistic logic.
A substitution instance of $\Rightarrow In(x, B'\sigma)$ is derivable from the set $\Gamma' \cup E \cup A'$ if and only if the clause set $\Gamma' \cup E \cup A' \cup \{In(x, B'\sigma) \Rightarrow\}$ is refutable. We will say that the translation of the goal type G is the clause set $A' \cup \{In(x, B'\sigma) \Rightarrow \} \cup E'$, where E' is the set of equality substitution axioms for the function symbols in $A \cup B$.
There are well-known techniques enabling us to always construct the required substitution instance of $\Rightarrow In(x, B'\sigma)$ from the resolution refutation of Horn clause sets like $\Gamma' \cup E \cup A' \cup \{In(x, B'\sigma) \Rightarrow\}$; see [6] and [5]. It is easy to see that any refutation from a Horn clause set Γ where $In(x, B'\sigma) \Rightarrow$ is a single clause containing no positive literals will end with the hyperresolution step with premisses being the input clause $In(x, B'\sigma) \Rightarrow$ and some derived clause $\Rightarrow In(t, r)$. Due to the specific form of the clause sets obtained by the translations of type theory judgements, the terms t and r will moreover always be ground.
After finding the refutation we thus have an instance of the clause $\Rightarrow In(x, B'\sigma)$ with a ground term t replacing the variable x. The function body g is built of t by replacing the Skolem constant symbols $c_1, \ldots, c_n$ in t by the variables $x_1, \ldots, x_n$, respectively.

Example 2. In this example, we introduce an inductively defined predicate Leq, expressing the less than or equal relation on the natural numbers (we will add numbers for easier reference).

$$\mathsf{N} \in \mathsf{Set} \tag{1}$$
$$\mathsf{0} \in \mathsf{N} \tag{2}$$
$$\mathsf{s} \in (\mathsf{x} \in \mathsf{N})\mathsf{N} \tag{3}$$
$$\mathsf{Leq} \in (\mathsf{x}, \mathsf{y} \in \mathsf{N})\mathsf{Set} \tag{4}$$
$$\mathsf{leq_0} \in (\mathsf{x} \in \mathsf{N})\mathsf{Leq}(\mathsf{0}, \mathsf{x}) \tag{5}$$
$$\mathsf{leq_s} \in (\mathsf{x}, \mathsf{y} \in \mathsf{N}; \mathsf{z} \in \mathsf{Leq}(\mathsf{x}, \mathsf{y}))\mathsf{Leq}(\mathsf{s}(\mathsf{x}), \mathsf{s}(\mathsf{y})) \tag{6}$$

We denote these declarations by Γ. Consider the task of finding a term inhabiting the following type C:

$$(\mathsf{x} \in \mathsf{N})\mathsf{Leq}(\mathsf{s}(\mathsf{0}), \mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{x}))))$$

The translation $Tr(\Gamma)$ of Γ is the following clause set:

1: $\Rightarrow In(N, Set)$
2: $\Rightarrow In(0, N)$
3: $In(x, N) \Rightarrow In(s(x), N)$
4: $In(x, N), In(y, N) \Rightarrow In(Leq(x, y), Set)$
5: $In(x, N) \Rightarrow In(leq_0(x), Leq(0, x))$

6: $In(x,N), In(y,N), In(z, Leq(x,y)) \Rightarrow In(leq_s(x,y,z), Leq(s(x), s(y)))$

where x, y and z are the only variables. The translation of the goal is the following clause set C':

7: $\Rightarrow In(c,N)$
8: $In(u, Leq(s(0), s(s(s(c))))) \Rightarrow$

where c is a new constant symbol. We want to show that the clause set $Tr(\Gamma) \cup C'$ is refutable and find a term t such that the set $Tr(\Gamma) \cup C'\{t/u\}$ would also be refutable. Indeed, there exists the following resolution refutation:

9 (from 3,7): $\Rightarrow In(s(c), N)$
10 (from 3,9): $\Rightarrow In(s(s(c)), N)$
11 (from 5,10): $\Rightarrow In(leq_0(s(s(c))), Leq(0, s(s(c))))$
12 (from 6,2,10,11): $\Rightarrow In(leq_s(0, s(s(c)), leq_0(s(s(c)))), Leq(s(0), s(s(s(c)))))$
13 (from 8,12): $\Rightarrow$

giving us the term $leq_s(0, s(s(c)), leq_0(s(s(c))))$ as the required substitution instance for u in $In(u, Leq(s(0), s(s(s(c)))))$. By replacing the new constant c with a variable x we get the term $leq_s(0, s(s(x)), leq_0(s(s(x))))$ as the body of a function inhabiting the type $(\mathsf{x}{\in}\mathsf{N})\mathsf{Leq}(\mathsf{s}(\mathsf{0}), \mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{x})))))$.

We will below present an optimizations leading to a smaller input clause set which will avoid the explicit derivation of 9, 10 and 11 in the present example.

4.2 Translating Derivation Rules

Type Formation Rules For our purposes, the set of assumptions and the goal can be assumed to be already correctly formed types. Because the derivation rules of type theory preserve type correctness, we can ignore type formation rules without losing soundness.

Notice that our translation preserves the subformula structure. Concerning completeness, we see that we may ignore type construction rules as long as we only need types that are syntactically subtypes of the assumptions and the goal. Since F_1 corresponds to a fragment of first-order logic and since the normal form of any derivation in first-order logic has the subformula property, we may, hence, ignore type construction rules.

The Application rule and the Substitution Rules These are the main rules for our purposes. It is easy to see that for the fragment F_1 the application rule

$$\frac{c{\in}(x{\in}A)B \quad a{\in}A}{c(a){\in}B\{a/x\}}$$

can be assumed to have a "multiple form" instantiating all the arguments of a function at once:

$$\frac{c{\in}(x_1{\in}A_1; \ldots; x_n{\in}A_n)B \quad a_1{\in}A_1 \ \ldots \ a_n{\in}A_n}{c(a_1, \ldots, a_n){\in}B\{a_1/x_1, \ldots, a_n/x_n\}}$$

Indeed, since in F_1 function types do not occur as argument types of functions, if the result of a single-step application rule is a function, then this function can only be used as a left premiss of an application rule.

The translated form of the multiple application rule is:

$$\frac{In(x_1, A_1), \ldots, In(x_n, A_n) \Rightarrow In(c(x_1, \ldots, x_n), B) \Rightarrow In(a_1, A_1) \vdots \Rightarrow In(a_n, A_n)}{\Rightarrow In(c(a_1, \ldots, a_n), B\{a_1/x_1, \ldots, a_n/x_n\})}$$

The last rule is an instance of the hyperresolution rule combining multiple modus ponens with substitution limited to most general unifiers:

$$\frac{F_1, \ldots, F_n \Rightarrow F \qquad \Rightarrow F_1' \qquad \ldots \qquad \Rightarrow F_n'}{\Rightarrow F\sigma}\sigma = mgu(F_1, F_1') \ldots mgu(F_n, F_n')$$

4.3 Soundness and Completeness of the Translation

We will denote the result of translating a judgement or a set of judgements A as $Tr(A)$. We will denote the set of clauses obtained by translating a goal type A as $Trg(A)$.

We will show soundness and completeness of the translation and the hyperresolution calculus with respect to the fragment F_1 of type theory. By soundness we mean that if there is a hyperresolution refutation of $Trg(G) \cup Tr(\Gamma)$ for a goal type G and a set of judgements Γ, then there is a term t and type theory proof of $t \in G$ from Γ. By completeness we mean that if there is a type theory proof from Γ showing that some term inhabits a certain type G, then there is a hyperresolution refutation of $Trg(G) \cup Tr(\Gamma)$.

Lemma 1. *Removing all the equality rules of type theory except β-conversion preserves completeness for the fragment F_1.*

Proof. All the equality rules except β-conversion are covered by the standard first-order axiomatization of equivalence and substitutivity properties of the first-order equality predicate.

Lemma 2. *The type theory calculus obtained by removing the abstraction rule and the β-conversion rule preserves completeness for the fragment F_1.*

Proof. It is easy to see that for the fragment F_1 the abstraction rule is unnecessary. The reason is that F_1 corresponds to a fragment of first-order logic, and for the latter we have the subformula property, guaranteeing completeness without introducing any assumptions which are not subformulas of the conclusion.

Since we do not have assumptions, that is, formulas of the form $b \in B[x \in A]$ in the fragment F_1, and the abstraction rule is unnecessary, then the β-conversion rule (corresponding to the cut rule in first-order logic)

$$\frac{a \in A \quad b \in B[x \in A]}{((x)b)(a) = b\{a/x\} \in B\{a/x\}}$$

is also unnecessary.

The following is the standard *lifting lemma* of the theory of resolution calculus. (see e.g. [6]):

Lemma 3. *Let A and B be two clauses $A \equiv \{A_1, \ldots, A_n\}$ and $B \equiv \{A_1, \ldots, A_n\}$. Let A' and B' be two clauses with the following properties: $A' = A\sigma \cup A''$, $B' = B\rho \cup B''$ where σ and ρ are substitutions, A'' and B'' are arbitrary clauses. Whenever a new clause C' can be derived from A' and B' by resolution, such a clause C can be derived from A and B that $C' = C\mu \cup C''$, where μ is a substitution and C'' is a clause.*

Lemma 4. *The hyperresolution rule applied to the translation of type theory judgements is sound and complete for type theory with only application and substitution as derivation rules.*

Proof. The proof of soundness is by induction on the size of derivation. The base case is immediate. The induction step is also obvious, since every hyperresolution application corresponds to a derivation in type theory containing only substitutions and applications.

The proof of completeness is also by induction on the size of derivation. Again, the base case is obvious. For the induction step, assume that we have a substitution or application in type theory from premisses Γ with conclusion F. We have to show that there is a corresponding hyperresolution derivation from the clause set $Tr(\Gamma')$ giving a clause $Tr(F')$ such that Γ is a set of substitution instances of Γ' and F is a substitution instance of F'. The substitution step is obvious: take the translation of the unchanged premiss of the substitution derivation. The application step follows from the lifting lemma.

Theorem 5. *The translation and hyperresolution are sound and complete with respect to the fragment F_1 of type theory.*

Proof. Follows from the previous lemmas in this section.

4.4 Optimizing the Translation

We introduce several optimizations of our translation of the fragment F_1.

The First Optimization: O_1 Let Γ be a set of type theory judgements in the fragment F_1 and let G be a goal type in the fragment F_1. Observe that all the positive singleton clauses in $Tr(\Gamma) \cup Trg(G)$ are ground and all the variables in a positive literal of any non-singleton clause C in $Tr(\Gamma) \cup Trg(G)$ occur as first arguments in the negative literals of C. Therefore any nonempty clause derived from $Tr(\Gamma) \cup Trg(G)$ by hyperresolution is a singleton ground clause.

We present an optimization O_1 for translations of type theory judgements, which is crucial for improving the efficiency of automated proof search. The point of O_1 is allowing the derivation of non-ground singleton clauses, each covering a possibly infinite set of ground singleton clauses derivable from the non-optimized clause set.

The optimization O_1 is obtained by replacing all atoms with the binary predicate In by an atom with a unary predicate Inh, always discarding the first argument of In.

The encoded goal clause $G \equiv \ \Rightarrow In(x, B\sigma)$ turns into a ground clause $O_1(G) \equiv \Rightarrow Inh(B\sigma)$. After we have found a first-order derivation of the clause $O_1(G)$ from the set of optimized assumption clauses $O_1(\Gamma)$, we construct the corresponding derivation of a substitution instance of the non-optimized G from the set of non-optimized assumptions Γ.

Example 2. **3.** Consider the set of judgements A obtained by translating the type theory judgements in example 2. The optimization O_1 applied to the clauses in example 2 will give the following clause set $O_1(A)$.

1': $\Rightarrow Inh(Set)$
2': $\Rightarrow Inh(N)$
3': $Inh(N) \Rightarrow Inh(N)$
4': $Inh(N), Inh(N) \Rightarrow Inh(Set)$
5': $Inh(N) \Rightarrow Inh(Leq(0, x))$
6': $Inh(N), Inh(N), Inh(Leq(x, y)) \Rightarrow Inh(Leq(s(x), s(y)))$
7': $\Rightarrow Inh(N)$
8': $Inh(Leq(s(0), s(s(s(c))))) \Rightarrow$

There exists the following resolution refutation from $O_1(A)$:

11' (from 2',5'): $\Rightarrow Inh(Leq(0, x)$
12' (from 2',2',11',6'): $\Rightarrow Inh(Leq(s(0), s(x)))$
13' (from 8',12'): $\Rightarrow$

Let Γ be a clause set in the fragment F_1, let G be a goal type in F_1. $Tr(\Gamma)$ is a translated form of Γ and $Trg(G)$ is the translated form of G. $O_1(Tr(\Gamma))$ is the optimized form of $Tr(\Gamma)$.

For the simplicity of presentation we will assume that equality is not present in Γ or G and that $Trg(G)$ consists of a sole negative singleton clause. The case with the equality will be treated later in the Section 5. It is always possible to reformulate G and Γ as G' and Γ' by assuming that all the positive singleton clauses in the original $Trg(G)$ are members of $Tr(\Gamma')$. The reformulated $Trg(G')$ then consists of a sole negative singleton clause, which can be used in the hyperresolution refutation of $Tr(\Gamma') \cup Trg(G')$ only for the last inference, the derivation of the empty clause.

The crucial importance of the optimization O_1 stands in that whereas hyperresolution derives only ground clauses from $Tr(\Gamma)$, from $O_1(Tr(\Gamma))$ it is possible to derive new clauses containing variables. Such clauses stand for generally infinite sets of ground clauses derivable from $Tr(\Gamma)$.

In the example above, we can derive a non-ground clause $\Rightarrow Inh(leq(s(0), s(y)))$ from the clauses 2', 5' and 6', covering the infinite set of clauses $\Rightarrow In(p_1, leq(s(0), s(0))), \Rightarrow In(p_2, leq(s(0), s(s(0)))), \Rightarrow In(p_3, leq(s(0), s(s(s(0))))), \ldots\}$ derivable from A.

Build a new clause set A^i from the unoptimized clause set A in the example above by replacing the clause $8: \ In(u, leq(s(0), s(s(s(c))))) \Rightarrow$ with the clause $8^i: \ In(u, leq(s(0), s^i(c))) \Rightarrow$ where $s^i(c)$ stands for the term where s is applied i times. The shortest hyperresolution refutation of A^i consists of $i+3$ hyperresolution steps. The shortest hyperresolution refutation of $O_1(A^i)$ consists always of 3 steps, regardless of i. Due to the optimization we avoid deriving the previously necessary sequent $\Rightarrow In(s^i(c), N)$.

Soundness and completeness of the optimization O_1 The first order language of $Tr(\Gamma)$ is not typed. However, due to the construction of $Tr(\Gamma)$, all the terms occurring in the clauses of $Tr(\Gamma)$ can be seen as being typed by Γ. Every variable x occuring in a clause C in $Tr(\Gamma)$ occurs in a literal $In(x, t_x)$ in C, corresponding to a judgement $x \in t_x$ in Γ.

In the following we consider all the variables, constants, function symbols and terms (briefly: *objects*) in $Tr(\Gamma)$ to be typed. The type of an object in $Tr(\Gamma)$ is determined by Γ. The type of an object in $O_1(Tr(\Gamma))$ is the same as the type of a corresponding object in $Tr(\Gamma)$. A term t constructed from objects in $Tr(\Gamma)$ is *type-correct* iff the term t' corresponding to t in type theory is type-correct in the context of Γ.

By $\Delta \vdash_{h*} C$ we denote that a clause C is derivable from the clause set Δ by hyperresolution. Since any clause C such that $Tr(\Gamma) \vdash_{h*} C$ is ground, it is easy to see that C is type-correct.

Differently from $Tr(\Gamma)$, it is possible to derive non-ground clauses from $O_1(Tr(\Gamma))$. We extend the notion of type-correctness to clauses C such that $O_1(Tr(\Gamma)) \vdash_{h*} C$. Each variable x occurring in C can be traced (renaming taken into account) through the derivation tree of C to the occurrence of some variable y in $O_1(Tr(\Gamma))$. Thus we can say that it has a type of the variable y and we will extend the notion of type-correctness to the clauses derived from $O_1(Tr(\Gamma))$.

Lemma 6. *Consider a variable x occurring inside $O_1(Tr(\Gamma))$. Either x occurs as an argument of the predicate Inh, in which case it has a universal type* Set *or x is an argument a_i of some term $f(a_1, \ldots, a_n)$ occurring inside $O_1(Tr(\Gamma))$. In the last case the type of x is determined by the function symbol f, the position i of a_i in the term and and a subset of arguments $\Delta \subset \{a_1, \ldots, a_n\}$ such that $a_i \notin \Delta$. There exists a reflexive ordering $\succeq_d$ of the elements of $\{a_1, \ldots, a_n\}$, such that whenever $a_k \succeq_d a_l$ $(1 \leq k, l \leq n)$, the type of a_l cannot depend on the type of a_k.*

Proof. Follows from the rules of monomorphic type theory as a syntactic property.

Theorem 7. *The optimization O_1 preserves soundness and completeness.*

Proof. Let Γ be a clause set in the fragment F_1, let $Tr(\Gamma)$ be a translated form of Γ and let $O_1(Tr(\Gamma))$ be the optimized form of $Tr(\Gamma)$.

Completeness: if $Tr(\Gamma) \vdash_{h*} C$ holds, then there is such a clause C' and such a substitution σ that $C = C'\sigma$ and $O_1(Tr(\Gamma)) \vdash_{h*} O_1(C')$ holds. Consequently, if $Tr(\Gamma) \cup Trg(G)$ is refutable, then $O_1(Tr(\Gamma)) \cup O_1(Trg(G))$ is refutable.

The proof is easy, since O_1 does nothing but removes the first argument of each predicate symbol. Let D be a hyperresolution derivation of C from $Tr(\Gamma)$. Construct a new derivation D' by removing the first arguments from each occurrence of the predicate Inh in D and replacing Inh with In. Then D' is a substitution instance of a derivation giving $O_1(C')$ from $O_1(Tr(\Gamma))$. Therefore, due to the completeness of hyperresolution, $O_1(Tr(\Gamma)) \vdash_{h*} O_1(C')$ holds.

Soundness: If $O_1(Tr(\Gamma)) \vdash_{h*} C$ holds, then C is type-correct. Consequently (due to the completeness of $Tr(\Gamma)$ for type theory), if $O_1(Tr(\Gamma)) \cup O_1(Trg(G))$ is refutable, then $Tr(\Gamma) \cup Trg(G)$ is refutable.

Proof. We use induction over the hyperresolution derivation for the optimized case. The proof relies on that we are using monomorphic type theory where the previous lemma holds and that any applied substitution is a most general unifier of two existing terms.

Base case: obvious.

Induction step. Consider the hyperresolution rule:

$$\frac{L_1, \ldots, L_n \Rightarrow L \quad \Rightarrow L'_1 \quad \ldots \quad \Rightarrow L'_n}{\Rightarrow L\sigma}\sigma = mgu(L_1, L'_1) \ldots mgu(L_n, L'_n)$$

The literal L has a form $Inh(r)$. The induction step is proved by showing that $L\sigma$ is type-correct.

The proof is by induction over the substitution σ which is ordered by the dependency ordering $\succeq_d$. Base case is obvious, we proceed to proving the induction step.

An element t/x of $\sigma - \rho$ is obtained by unifying a variable x in a literal $L_i\rho$ with the term t in $L'_i\rho$, where $\rho \subset \sigma$. We have:

$$\forall u, p, v, s.(\{p/u\} \subset (\sigma - \rho) \,\&\, \{s/v\} \subset \rho) \Rightarrow v \succeq_d u$$

The variable x either occurs as a sole argument in the literal $Inh(x)$, in which case x has a universal type Set or as an argument a_i of some surrounding term $f(a_1, \ldots, a_n)$.

In the first case also t occurs as a sole argument of $Inh(t)$, the type of t is Set and thus $r\rho\{t/x\}$ is type-correct.

Consider the second case. Here also the term t occurs as an i-th argument of a surrounding term $f(a'_1, \ldots, a'_n)$. Both the leading function symbol f and the position index i in the surrounding term are the same for x and t. Thus either both the types of x and t do no depend on other terms or they both depend on a subset of arguments in the surrounding term. Consider the case where the types of x and t do not depend on other terms. In that case the type of t is the same as the type of x and thus $r\rho\{t/x\}$ is type-correct.

Consider the case where the types of x and t depend on other terms. Due to the previous lemma all these depended-upon terms form a subset of pairwise corresponding arguments D of $f(a_1, \ldots, a_n)$ and D' of $f(a'_1, \ldots, a'_n)$. The previous lemma shows that there exists a dependency ordering $\succeq_d$ so that the types

of the elements of D and D' do not depend on the types of x and t. Thus we can assume that $D = D' = D\sigma = D'\sigma$, i.e, the terms corresponding to D and D' in L_i and L'_i are already unified by ρ. Hence the type of t in $L'_i\rho$ is the same as the type of x in $L_i\rho$, thus $r\rho\{t/x\}$ is type-correct.

Observe that in some sense the unrestricted substitution rule (i.e. a substitution rule which uses arbitrary substitutions, not only these obtained by unification) would make the optimized calculus unsound in respect to the type theory, although the refutability of a set $O_1(Tr(\Gamma))$ with unrestricted substitution added as a derivation rule would still be equivalent to the refutability of $Tr(\Gamma)$.

The Second Optimization: O_2 The second optimization O_2 is applicable only to such clause sets which have been already optimized by O_1. O_2 is obtained by removing certain clauses and some atoms on the left sides of other clauses. It is essentially a preprocessing phase pre-applying obvious hyperresolution steps and removing redundant clauses and literals.

Definition *We say that a type* $f \in (x_1 \in A_1; \ldots; x_n \in A_n)\mathsf{Set}$ *is* essentially independent *in a given set of judgements iff it can be shown that* $f(x_1, \ldots, x_n)\rho$ *is inhabited for any substitution* ρ *such that* $f(x_1, \ldots, x_n)\rho$ *is correctly formed regarding types. We will call such* f *an* essentially independent constructor

For example, the special type Set, the type of natural numbers $\mathsf{N} \in \mathsf{Set}$, $\mathsf{0} \in \mathsf{N}$, $\mathsf{s} \in (\mathsf{x} \in \mathsf{N})\mathsf{N}$ and the parameterized list type $\mathsf{List} \in (\mathsf{x} \in \mathsf{Set})\mathsf{Set}$, $\mathsf{nil} \in (\mathsf{x} \in \mathsf{Set})\mathsf{List}(\mathsf{x})$, $\mathsf{cons} \in (\mathsf{x} \in \mathsf{Set}; \mathsf{y} \in \mathsf{x}; \mathsf{z} \in \mathsf{List}(\mathsf{x}))\mathsf{List}(\mathsf{x})$ are easily shown to be *essentially independent*.

The optimization O_2 takes a set of assumptions of the form given by the optimization O_1. It returns a modified set of assumptions and the goal where for any input judgement clause $s \equiv Inh(A_1), \ldots, Inh(A_m) \Rightarrow Inh(B)$ the following is done:

- if B is a composite term with an essentially independent constructor as a leading function symbol, remove the clause s.
- remove any premiss $Inh(A_i)$ such that A_i has an essentially independent constructor as a leading function symbol.

Example 3. **4.** Take the O_1-optimized clause set S from the example 3. You may also want to look at the original untranslated set of type theory judgements and the goal in the example 2. The result of the optimization S_2 applied to S is the following set of three clauses:

5": $\Rightarrow Inh(Leq(0, x))$
6": $Inh(Leq(x, y)) \Rightarrow Inh(Leq(s(x), s(y)))$
8": $Inh(Leq(s(0), s(s(s(c))))) \Rightarrow$

The last clause set has a short refutation:

1 (from 6", 5") $\Rightarrow Inh(Leq(s(0), s(y))$
2 (from 8", 1) $\Rightarrow$

In case the type term A in the goal clause $G \equiv \Rightarrow Inh(A)$ has an essentially independent leading function symbol, then the problem of constructing an element of A reduces to the simple problem of using the procedure of checking essential independency for constructing an element of A.

Soundness and completeness of the optimization O_2 follow from the definition of essentially independent types.

5 Building In Equality

For the Horn fragment where the translation of the F_1 class belongs, the explicit axiomatization of the equality predicate (except the reflexivity axiom $x = x$ which must be preserved) can be replaced by the following restricted form of paramodulation without losing completeness (see e.g. [12]).

$$\frac{\Rightarrow L[t] \quad \Rightarrow t' = g}{\Rightarrow L[g]\sigma}\sigma = mgu(t, t')$$

where $L[g]$ is obtained by replacing one occurrence of the term t in $L[t]$ by the term g. The term t (the term *paramodulated into*) in the paramodulation rule is prohibited to be a variable. The equality predicate in the rule is assumed to be commutative, i.e., $t' = g$ is the same as $g = t'$.

The definitional equality predicate $=$ in type theory extended with pattern-matching (as in Alf) is always assumed to define a terminating and confluent rewrite relation E. Thus we do not lose completeness in case we treat such equality axioms as rewrite rules, ie. if we rewrite any term in any derived first-order clause modulo the rewriting relation E. See [12]). However, we will still need to keep the paramodulation rule to guarantee completeness.

The soundness and completeness of using the paramodulation (or rewrite rules) for the O_1-optimized translations of type theory judgements is proved analogously to the earlier proof of soundness and completeness of the O_1-optimized translation.

Note that the other type of equality often used in the applications of type theory, the propositional equality Id, is not definable in F_1. However, we can build the propositional equality directly into the first order calculus.

We will use the following special form of the paramodulation rule along with the reflexivity axiom $\Rightarrow Id(A, x, x)$ instead of the standard equivalence and substitution rule schemes for Id:

$$\frac{\Rightarrow L[t] \quad \Rightarrow Id(A, t', g)}{\Rightarrow L[g]\sigma}\sigma = mgu(t, t')$$

where $L[g]$ is obtained by replacing one occurrence of the term t in $L[t]$ by the term g. The terms t and t' in the paramodulation rule are prohibited to be variables. Id is assumed to be commutative, i.e., $Id(A, t', g)$ is the same as $Id(A, g, t')$.

6 Translating Nested Function Types

6.1 The General Case

Consider full type theory. We obtain the translation Tri by modifying and extending the translation Tr given for the fragment F_1 to the implicational fragment of first-order intuitionistic logic as a target logic.

- **Application terms.** $f(g_1, g_2, \ldots, g_n)$ is translated as $ap(\ldots ap(ap(Tri(f), Tri(g_1)), Tri(g_2)), \ldots), Tri(g_n))$, where $Tri(t)$ denotes the result of the translation of t.
- **Abstraction terms.** An abstraction term $[x]t$ is translated by Schönfinkel's abstraction algorithm:
  $[x]x \to I$
  $[x]M \to ap(K, Tri(M))$ if M is a variable or constant, $M \neq x$.
  $[x](MN) \to ap(ap(S, [x]Tri(M)), [x]Tri(N))$.
  where I, S, K are special constants.
- **Type judgements.** A judgement $t \in (x \in A)B$ is translated as $\forall x.Tri(x \in A) \Rightarrow Tri(ap(t, x) \in B)$. A judgement $t \in C$ where C is not a function type is translated as $Inh(Tri(t), Tri(C))$. Constants and variables remain unchanged by the translation.
- **Goal.** The goal $G_1, \ldots, G_n \vdash_{tt} ? \in B$ of finding a term inhabiting B in the context of judgements $G_1, \ldots, G_n$ is translated as a formula $Tri(G_1) \Rightarrow (\ldots (Tri(G_n) \Rightarrow \exists x.Tri(x \in B)) \ldots)$.
- **Target logic.** An implicational fragment of first-order intuitionistic logic with the equality predicate plus the standard equalities $S_=, K_=, I_=$ for combinators S, K, I: $ap(I, x) = x$, $ap(ap(K, x), y) = x$, $ap(ap(ap(S, x), y), z) = ap(ap(x, z), ap(y, z))$.

It is known (see [3]) that the Schönfinkel's abstraction algorithm combined with the before-mentioned equality rules for S, K, I simulates weak β-reduction of lambda calculus $([x]MN) = M\{N/x\}$ but not the ξ rule: $M = M' \Rightarrow [x]M = [x]M'$.

From the view of automated theorem proving the main problem with using the equalities $S_=, K_=, I_=$ stands in that these equalities significantly expand the search space of proving a type theory goal. In the following we will consider a fragment where $S_=, K_=, I_=$ can be avoided without resorting to higher order unification.

6.2 Translating Types with Independent Nested Function Types

The following fragment F_2 is a superset of the previously considered fragment F_1. The motivation for considering F_2 stems from the fact that the problems arising in the general case disappear, but F_2 is strong enough to allow synthesis of conditional programs.

Definition *An* independent function type *is a type with the form* $(x \in A)B$ *such that* x *does not occur in* B.

A type C *in type theory belongs to the type fragment* F_2 *iff* C *is either not a function type or it has a form* $(x \in A)B$ *such that the following two cases hold:*

1. B *belongs to the type fragment* F_2
2. A *is not a function type or* A *is an independent function type and* x *does not occur in* B.

A type theory judgement C *belongs to the judgement fragment* F_2 *if* C *does not contain a type outside the type fragment* F_2.

The translation Tr' for F_2 is obtained from the previously considered translation Tri by applying an optimization O_1 (literals of the form $In(t, g)$ are converted to the form $Inh(g)$). Soundness and completeness proofs of the translation are obtained from the analogous proofs for the fragment F_1. The most convenient way to extend these proofs is to use the resolution method for intuitionistic logic proposed by G.Mints, see [15] and [16].

In addition to the optimization O_1, we will apply the analogue of the optimization O_2 to the formula given by the translation Tri.

We note that the introduction and elimination rules for disjunction and conjunction can be defined in F_2. These connectives will be on the different level than implication, but we can enhance the efficiency of the prover by building these connectives into the target logic, much like we can build propositional equality into target logic.

Since quantifiers cannot be defined in F_2, we can handle those only by building them into the target logic, ie. considering full intuitionistic logic instead of the implicational fragment.

7 Experiments with the Implementation

T.Tammet has implement an automated theorem prover Gandalf, which looks for type theory proofs in the fragments F_1 and F_2. Gandalf is written in Scheme and compiled to C. The timings presented in the following are obtained by running Gandalf on the SUN SS-10.

Gandalf takes a file containing type theory judgements and possibly several goal types, written in the ALF syntax. It converts the judgements and goals to first order language using the translation and optimizations described in the previous sections. After that it starts looking for proofs. Gandalf organizes proof search by iterative deepening on runtime. Suppose the input file contains a set of judgements Γ and a set of goal types $G_1, \ldots, G_n$. Gandalf will take the first time limit t_1 and allocate t_1/n seconds for the attempt to prove G_1. In case it fails, Gandalf will proceed to G_2, etc. In case it succeeds, all the following proof search subtasks will treat G_1 as a judgement, the remaining time from t_1 will be divided between goals $G_2, \ldots, G_n$ and Gandalf will proceed to the next goal G_2. In case a proof search attempt has been conducted for each goal $G_1, \ldots, G_n$ and

for some of these no proof is found, Gandalf will take a new time limit $t_2 = f * t_1$ for some factor f and will perform a new iteration for yet unproved goals with the new time limit t_2.

Consider the task of proving a separate subgoal G: $(x_1 \in A_1; \ldots; x_n \in A_n)B$. Gandalf will attempt to use structural induction, i.e., pattern-matching, for proving G. It will consider each argument type A_i in G, finding all these judgements in Γ which are introduction rules for A_i, that is, which are of the form $(y_1 \in B_1; \ldots; y_n \in B_m)C$ with C and A_i being unifiable. It will then generate a set T of different proof search tasks for G, each element of T corresponding to a subset of variables $x_1, \ldots, x_n$ being inducted upon. To every element (induction case) of the task in T it adds the induction assumptions. Induction assumptions are generated from the subterms of the pattern of the inductive case at hand. Only these assumptions are considered which are structurally smaller than the pattern of the particular induction case.

A certain amount of time will be allocated for proof search for each element of T. In case Gandalf manages to prove some element E of T, it has found the proof for G and it will continue with the next open goal from the set $G_1, \ldots, G_n$.

Consider the task of proving an element E in the set of proof search tasks for G. Each such element generally consists of a number of pure first order proof tasks $R_1, \ldots, R_k$, each R_i corresponding to one case of structural induction. E is proved only if each of $R_1, \ldots, R_k$ has been proved.

In case Gandalf finds a proof for some goal G_i, it will construct a sound type theory proof of G_i, using the ALF syntax. Each inductive case will have its own separate proof. Each such separate proof is constructed from the proof found for the corresponding first order clause set. In order to make this possible, Gandalf keeps track of all the necessary information while looking for the first order proofs.

The current implementation of Gandalf is only able to to construct type theory proofs for the fragment F_1. The part constructing type theory proofs from the resolution proofs for the intuitionistic fragment F_2 is being worked upon.

Gandalf does not attempt to construct lemmas. Thus it is completely up to the user to include all the necessary lemmas in the input file. Selecting lemmas to be proved (and later used for the main proof) is one of the main mechanisms available to the user in order to guide the blind search of Gandalf.

In addition to the selection of lemmas, there are a number of flags and parameters the user can give to Gandalf in order to guide the proof search. For example, it is possible to indicate that no induction should be used for a certain goal or that a certain subset of variables has to be inducted upon, etc.

7.1 Example. Correctness of Toy Compiler

The following example encodes the problem of proving correctness of a simple compiler. The example is presented in [8] with a proof created by a human user (C.Coquand) in interaction with the ALF system. We present a proof found by

the Gandalf system using a number of crucial *hints* given to Gandalf by the human user.

We want to compute a polynomial expression build of multiplication, addition, basic integers and one variable on a simple stack machine. The instructions for this machine are:

- Duplication of the top of the stack.
- Reversal of the top items on the stack.
- Replacement of the top items on the stack by an item that is their sum (respectively product).
- Replacement of the top item n on the stack by a given item n_0.

The problem is to compile a given polynomial expression $e(x)$ to a list l of instructions such that if we execute l on the stack, then the result of this computation is the stack with the value of the expression $e(x)$ pushed to the top.

The evaluation function is represented as a type theoretic function $\mathsf{Eval} \in (\mathsf{Expr}; \mathsf{N})\mathsf{N}$ such that $\mathsf{Eval}(\mathsf{e}, \mathsf{n})$ gives the value of the expression e when its unique variable is set equal to value n. The execution function which computes a final stack from the input stack and a list of instructions is represented as an inductively defined relation EXEC between a list of instructions and two stacks. This definition uses an auxiliary relation Exec between a single relation and two stacks.

The proof of the main theorem thm requires a lemma showing that EXEC is a transitive relation. Gandalf does not attempt to create lemmas automatically, thus lemma is present in the input file along with the assumption that it is already proved.

```
N : Set,  0 : N,  s : (N)N

plus : (N;N)N,   mult : (N;N)N

List : (Set)Set
  nil : (t:Set)List(t)
  cons : (t:Set; x:t; y:List(t))List(t)

Stack : Set,   null: Stack,   push:(N;r)Stack

Instr : Set

Dup : Instr, Rev : Instr, Add : Instr, Mul : Instr, Lit : (N)Instr

Expr : Set,
  Sum : (Expr;Expr)Expr,
  Pro : (Expr;Expr)Expr
  Num : (N)Expr
  Arg : Expr
```

```
Eval : (Expr;N)N
  Eval(Sum(e1,e2),n)=plus(Eval(e1,n),Eval(e2,n))
  Eval(Pro(f1,f2),n)=mult(Eval(f1,n),Eval(f2,n))
  Eval(Num(n1),n)=n1
  Eval(Arg,n)=n

append : (List(Instr);List(Instr))List(Instr)
  append(nil(Instr),x)=x
  append(cons(Instr,x,y),z)=cons(Instr,x,append(y,z))
  append(x,nil(Instr))=x

Exec : (Instr;Stack;Stack)Set
 Exec_Dup : (n:N;r:Stack)
            Exec(Dup,push(n,r),push(n,push(n,r)))
 Exec_Rev : (n1,n2:N;r:Stack)
            Exec(Rev,push(n1,push(n2,r)),push(n2,push(n1,r)))
 Exec_Add : (n1,n2:N;r:Stack)
            Exec(Add,push(n1,push(n2,r)),push(plus(n1,n2),r))
 Exec_Mul : (n1,n2:N;r:Stack)
            Exec(Mul,push(n1,push(n2,r)),push(mult(n1,n2),r))
 Exec_Lit : (n,m:N;r:Stack)
            Exec(Lit(n),push(m,r),push(n,r))

Program = List(Instr)  : Set

EXEC : (Program; Stack; Stack)Set
 exec_nil : (r:Stack)EXEC(nil(Instr),r,r)
 exec_seq : ( i:Instr; p:Program; r1,r2,r3:Stack;
              Exec(i,r1,r2); EXEC(p,r2,r3) )
            EXEC(cons(Instr,i,p),r1,r3)

Comp : (e:Expr)Program
 Comp(Sum(e1,e2)) =
   cons(Instr,
        Dup,
        append(Comp(e2),
               append(cons(Instr,Rev,nil(Instr)),
                  append(Comp(e1),cons(Instr,Add,nil(Instr))) )))
 Comp(Pro(f1,f2)) =
   cons(Instr,
        Dup,
        append(Comp(f2),
               append(cons(Instr,Rev,nil(Instr)),
                  append(Comp(f1),cons(Instr,Mul,nil(Instr))) )))
```

```
 Comp(Num(n)) =
   cons(Instr,Lit(n),nil(Instr))
 Comp(Arg) = nil(Instr)


lemma : ( p1,p2:Program; s1,s2,s3:Stack;
           EXEC(p1,s1,s2); EXEC(p2,s2,s3) )
         EXEC(append(p1,p2),s1,s3)

thm : (e:Expr; r:Stack; n:N)
       EXEC(Comp(e),push(n,r),push(Eval(e,n),r))

  thm(a,b,c) = ?(forceind(a),noind(b,c),method(both-ends))
```

The expression $\mathsf{thm}(\mathsf{a},\mathsf{b},\mathsf{c}) = ?(\mathsf{forceind}(\mathsf{a}), \mathsf{noind}(\mathsf{b},\mathsf{c}), \mathsf{method}(\mathsf{both\text{-}ends}))$ gives hints for the search of the proof of thm. The expressions $\mathsf{forceind}(\mathsf{a})$ and $\mathsf{noind}(\mathsf{b},\mathsf{c})$ tell Gandalf that it is necessary to use induction on the variable a, while it is prohibited to use induction on the variables b and c. In case we do not provide Gandalf with these two hints, it will spend a lot of time attempting to use induction schemas which are of no use when proving thm, and the proof is found only after a one-hour search.

The expression $\mathsf{method}(\mathsf{both\text{-}ends})$ tells Gandalf that it should look for proof by reasoning from both ends, i.e. doing both forward reasoning and backward reasoning simultaneously (implemented as binary unit resolution). The default strategy of Gandalf is using pure forward reasoning (implemented as hyper-resolution), which is ill suited for the current problem. It takes Gandalf almost two hours (deriving 337008 clauses) to prove even a single induction case $\mathsf{thm}(\mathsf{Sum}(\mathsf{x4},\mathsf{x5}),\mathsf{b},\mathsf{c})$ using pure forward reasoning.

We are planning to implement a heuristic module of Gandalf which would automatically make choices between reasoning methods.

Consider the definition of append used in the formulation of the problem. The current version of the proof checker ALF does not allow overlapping patterns, thus it fails to type-check the third case $\mathsf{append}(\mathsf{x}, \mathsf{nil}(\mathsf{Instr})) = \mathsf{x}$. However, this case preserves confluency of the definition of append. We have preferred to add it to the definition of append, since the extended rewriting relation clears up a noticeable amount of search space and enables the prover to find the proof faster. Alternatively, we could consider proving the equality $Id(List(Instr), append(x, nil(Instr)), x)$ as a separate lemma, which will have the same efficiency-boosting effect in case the prover is able to orient this equality to a rewriting rule.

The following is a proof found by Gandalf. We have deleted five arguments of essentially independent type from each application of the functions lemma and exec_seq in order to shorten the proof.

The proof is found in ca two and a half minutes. The first two induction cases $\mathsf{thm}(\mathsf{Sum}(\mathsf{x4},\mathsf{x5}),\mathsf{b},\mathsf{c})$ and $\mathsf{thm}(\mathsf{Pro}(\mathsf{x2},\mathsf{x3}),\mathsf{b},\mathsf{c})$ are both proved in one minute

(five thousand clauses derived, four thousand of these kept), the last two cases $\mathsf{thm}(\mathsf{Num}(\mathsf{x1}), \mathsf{b}, \mathsf{c})$ and $\mathsf{thm}(\mathsf{Arg}, \mathsf{b}, \mathsf{c})$ are proved in a fraction of a second (respectively 21 and 14 clauses derived), half a minute is wasted on an initial attempt to prove $\mathsf{thm}(\mathsf{Sum}(\mathsf{x4}, \mathsf{x5}), \mathsf{b}, \mathsf{c})$ with a half-minute time limit.

```
thm(Sum(x4,x5),b,c) =
  lemma(exec_seq(ExecDup(c,b),
                 thm(x5,push(c,b),c)),
        lemma(exec_seq(Exec_Rev(Eval(x5,c),c,b),
                       thm(x4,push(Eval(x5,c),b),c)),
              exec_seq(Exec_Add(Eval(x4,c),Eval(x5,c),b),
                   exec_nil(push(plus(Eval(x4,c),Eval(x5,c)),b)))))

thm(Pro(x2,x3),b,c) =
  lemma(exec_seq(Exec_Dup(c,b),
                 thm(x3,push(c,b),c)),
         lemma(exec_seq(Exec_Rev(Eval(x3,c),c,b),
                        thm(x2,push(Eval(x3,c),b),c)),
               exec_seq(Exec_Mul(Eval(x2,c),Eval(x3,c),b),
                    exec_nil(push(mult(Eval(x2,c),Eval(x3,c)),b)))))

thm(Num(x1),b,c) =
  exec_seq(Exec_Lit(x1,c,b),
           exec_nil(push(x1,b)))

thm(Arg,b,c) = exec_nil(push(c,b))
```

References

1. Peter Aczel. The strength of Martin-Löf's type theory with one universe. In *Proceedings of the Symposium on Mathematical Logic, Oulu, 1974*, pages 1–32. Report No 2, Department of Philosophy, University of Helsinki, 1977.
2. L. Augustsson, T. Coquand, and B. Nordström. A short description of Another Logical Framework. In *Proceedings of the First Workshop on Logical Frameworks, Antibes*, pages 39–42, 1990.
3. H. Barendregt. *The Lambda Calculus*. North Holland, 1981.
4. T.Tammet C.Fermüller, A.Leitsch and N.Zamov. *Resolution Methods for the Decision Problem*, volume 679 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, Berlin Heidelberg, 1993.
5. C.Green. Application of theorem-proving to problem solving. In *Proc. 1st Internat. Joint. Conf. Artificial Intelligence*, pages 219–239, 1969.
6. C.L.Chang and R.C.T Lee. *Symbolic Logic and Mechanical Theorem Proving*. Academic Press, 1973.
7. Thierry Coquand. Pattern matching with dependent types. In *Proceeding from the logical framework workshop at Båstad*, June 1992.

8. Thierry Coquand, Bengt Nordström, Jan M. Smith, and Björn von Sydow. Type theory and programming. *EATCS*, (52), February 1994.
9. T.Franzen D.Sahlin and S.Haridi. An intuitionistic predicate logic theorem prover. *Journal of Logic and Computation*, 2(5):619–656, 1992.
10. A. Felty and D. Miller. Encoding a Dependent-type λ-Calculus in a Logic Programming Language. In *Proceedings of CADE-10*. Lecture Notes in Artificial Intelligence 449, Springer Verlag, 1990.
11. W. D. Goldfarb. The undecidability of the second-order unification problem. *Theoretical Computer Science*, 13:225–230, 1981.
12. G.Peterson. A technique for establishing completeness results in theorem proving with equality. *SIAM J. of Comput*, 12:82–100, 1983.
13. Gérard Huet. A unification algorithm for typed λ-calculus. *Theoretical Computer Science*, 1(1):27–57, 1975.
14. Lena Magnusson. The new Implementation of ALF. In *The informal proceeding from the logical framework workshop at Båstad, June 1992*, 1992.
15. G. Mints. Gentzen-type systems and resolution rules. part i. propositional logic. In *COLOG-88*, volume 417 of *Lecture Notes in Computer Science*, pages 198–231. Springer Verlag, 1990.
16. G. Mints. Resolution strategies for the intuitionistic logic. In *Constraint Programming*, volume 131 of *NATO ASI Series F*, pages 289–311. Springer Verlag, 1994.
17. Bengt Nordström, Kent Petersson, and Jan M. Smith. *Programming in Martin-Löf's Type Theory. An Introduction.* Oxford University Press, 1990.
18. J. A. Robinson. A Machine-oriented Logic Based on the Resolution Principle. *ACM*, 12:23–41, 1965.
19. Jan Smith. An interpretation of Martin-Löf's type theory in a type-free theory of propositions. *Journal of Symbolic Logic*, 49(3):730–753, 1984.

This article was processed using the LaTeX macro package with LLNCS style