Scalable Fine-Grained Proofs for Formula Processing

Haniel Barbosa[1,2(✉)], Jasmin Christian Blanchette[3,1,4], and Pascal Fontaine[1]

[1] Université de Lorraine, CNRS, Inria, LORIA, Nancy, France
{haniel.barbosa,jasmin.blanchette,pascal.fontaine}@loria.fr
[2] Universidade Federal do Rio Grande do Norte, Natal, Brazil
[3] Vrije Universiteit Amsterdam, Amsterdam, The Netherlands
[4] Max-Planck-Institut für Informatik, Saarbrücken, Germany

Abstract. We present a framework for processing formulas in automatic theorem provers, with generation of detailed proofs. The main components are a generic contextual recursion algorithm and an extensible set of inference rules. Clausification, skolemization, theory-specific simplifications, and expansion of 'let' expressions are instances of this framework. With suitable data structures, proof generation adds only a linear-time overhead, and proofs can be checked in linear time. We implemented the approach in the SMT solver veriT. This allowed us to dramatically simplify the code base while increasing the number of problems for which detailed proofs can be produced, which is important for independent checking and reconstruction in proof assistants.

1 Introduction

An increasing number of automatic theorem provers can generate certificates, or proofs, that justify the formulas they derive. These proofs can be checked by other programs and shared across reasoning systems. Some users will also want to inspect this output to understand why a formula holds. Proof production is generally well understood for the core proving methods and for many theories commonly used in satisfiability modulo theories (SMT). But most automatic provers also perform some formula processing or preprocessing—such as clausification and rewriting with theory-specific lemmas—and proof production for this aspect is less mature.

For most provers, the code for processing formulas is lengthy and deals with a multitude of cases, some of which are rarely executed. Although it is crucial for efficiency, this code tends to be given much less attention than other aspects of provers. Developers are reluctant to invest effort in producing detailed proofs for such processing, since this requires adapting a lot of code. As a result, the granularity of inferences for formula processing is often coarse. Sometimes, processing features are even disabled to avoid gaps in proofs, at a high cost in proof search performance.

Fine-grained proofs are important for a variety of applications. We propose a framework to generate such proofs without slowing down proof search. Proofs are expressed using an extensible set of inference rules (Sect. 3). The succedent of a rule is an equality between the original term and the translated term. (It is convenient to consider formulas a special case of terms.) The rules have a fine granularity, making it possible to cleanly separate theories. Clausification, theory-specific simplifications, and expansion of 'let' expressions are instances of this framework. Skolemization may seem problematic, but with the help of Hilbert's choice operator, it can also be integrated into the framework. Some provers provide very detailed proofs for parts of the solving, but we are not aware of any publications about practical attempts to provide easily reconstructible proofs for processing formulas containing quantifiers and 'let' expressions.

At the heart of the framework lies a generic contextual recursion algorithm that traverses the terms to translate (Sect. 4). The context fixes some variables, maintains a substitution, and keeps track of polarities or other data. The transformation-specific work, including the generation of proofs, is performed by plugin functions that are given as parameters to the framework. The recursion algorithm, which is critical for the performance and correctness of the generated proofs, needs to be implemented only once. Another benefit of the modular architecture is that we can easily combine several transformations in a single pass, without complicating the code unduly or compromising the level of detail of the proof output. For very large inputs, this can improve performance.

The inference rules and the contextual recursion algorithm enjoy many desirable properties (Sect. 5). We show that the rules are sound and that the treatment of binders is correct even in the presence of name clashes. Moreover, assuming suitable data structures, we show that proof generation adds an overhead that is proportional to the time spent processing the terms. Checking proofs represented as directed acyclic graphs (DAGs) can be performed with a time complexity that is linear in their size.

We implemented the approach in veriT (Sect. 6), an SMT solver that is competitive on problems combining equality, linear arithmetic, and quantifiers [3]. Compared with other SMT solvers, veriT is known for its very detailed proofs [7], which are reconstructed in the proof assistants Coq [1] and Isabelle/HOL [8] and in the GAPT system [15]. As a proof of concept, we implemented a prototype checker in Isabelle/HOL.

By adopting the new framework, we were able to remove large amounts of complicated code in the solver, while enabling detailed proofs for more transformations than before. The contextual recursion algorithm had to be implemented only once and is more thoroughly tested than any of the monolithic transformations it subsumes. Our empirical evaluation reveals that veriT is as fast as before even though it now generates finer-grained proofs.

A shorter version of this report was presented at CADE-26 as a system description [2]. The current report includes more explanations and examples, detailed justifications of the metatheoretical claims, and extensive coverage of related work.

2 Conventions

Our setting is a many-sorted classical first-order logic as defined by the SMT-LIB standard [5] or TPTP TFF [40]. Our results are also applicable to richer formalisms such as higher-order logic (simple type theory) with polymorphism [18]. A signature $\Sigma = (\mathcal{S}, \mathcal{F})$ consists of a set $\mathcal{S}$ of sorts and a set $\mathcal{F}$ of function symbols over these sorts. Nullary function symbols are called constants. We assume that the signature contains a Bool sort and constants $\mathsf{true}, \mathsf{false} : \mathsf{Bool}$, a family $(\simeq : \sigma \times \sigma \to \mathsf{Bool})_{\sigma \in \mathcal{S}}$ of function symbols interpreted as equality, and the connectives $\neg$, $\wedge$, $\vee$, and $\longrightarrow$. Formulas are terms of type Bool, and equivalence is equality ($\simeq$) on Bool. Terms are built over function symbols from $\mathcal{F}$ and variables from a fixed family of infinite sets $(\mathcal{V}_\sigma)_{\sigma\in\mathcal{S}}$. In addition to $\forall$ and $\exists$, we rely on two more binders: Hilbert's choice operator $\varepsilon x.\,\varphi$ and a 'let' construct, let $\bar{x}_n \simeq \bar{s}_n$ in t, which simultaneously assigns n variables that can be used in the body t.

We use the symbol $=$ for syntactic equality on terms and $=_\alpha$ for syntactic equality up to renaming of bound variables. We reserve the names $\mathsf{a},\mathsf{c},\mathsf{f},\mathsf{g},\mathsf{p},\mathsf{q}$ for function symbols; x,y,z for variables; r,s,t,u for terms (which may be formulas); φ,ψ for formulas; and Q for quantifiers ($\forall$ and $\exists$). We use the notations $\bar{a}_n$ and $(a_i)_{i=1}^n$ to denote the tuple, or vector, $(a_1,\ldots,a_n)$. We write $[n]$ for $\{1,\ldots,n\}$.

Given a term t, the sets of its free and bound variables are written $FV(t)$ and $BV(t)$, respectively. The notation $t[\bar{x}_n]$ stands for a term that may depend on distinct variables $\bar{x}_n$; $t[\bar{s}_n]$ is the corresponding term where the terms $\bar{s}_n$ are simultaneously substituted for $\bar{x}_n$. Bound variables in t are renamed to avoid capture. Following these conventions, Hilbert choice and 'let' are characterized by

$$\models \exists x.\,\varphi[x] \longrightarrow \varphi[\varepsilon x.\,\varphi] \tag{ε_1}$$

$$\models (\forall x.\,\varphi \simeq \psi) \longrightarrow (\varepsilon x.\,\varphi) \simeq (\varepsilon x.\,\psi) \tag{ε_2}$$

$$\models (\text{let } \bar{x}_n \simeq \bar{s}_n \text{ in } t[\bar{x}_n]) \simeq t[\bar{s}_n] \tag{let}$$

Substitutions ρ are functions from variables to terms such that $\rho(x_i) \neq x_i$ for at most finitely many variables x_i. We write them as $\{\bar{x}_n \mapsto \bar{s}_n\}$; the omitted variables are mapped to themselves. The substitution $\rho[\bar{x}_n \mapsto \bar{s}_n]$ or $\rho[x_1 \mapsto s_1,\ldots,x_n \mapsto s_n]$ maps each variable x_i to the term s_i and otherwise coincides with ρ. The application of a substitution ρ to a term t is denoted by $\rho(t)$. It is capture-avoiding; bound variables in t are renamed as necessary. Composition $\rho' \circ \rho$ is defined as for functions (i.e., ρ is applied first).

3 Inference System

The inference rules used by our framework depend on a notion of *context* defined by the grammar

$$\Gamma ::= \varnothing \mid \Gamma, x \mid \Gamma, \bar{x}_n \mapsto \bar{s}_n$$

The empty context $\varnothing$ is also denoted by a blank. Each context entry either *fixes* a variable x or defines a *substitution* $\{\bar{x}_n \mapsto \bar{s}_n\}$. Any variables arising in the terms $\bar{s}_n$ will normally have been introduced in the context Γ on the left. If a context introduces the same variable several times, the rightmost entry shadows the others.

Abstractly, a context Γ fixes a set of variables and specifies a substitution $\mathit{subst}(\Gamma)$. The substitution is the identity for $\varnothing$ and is defined as follows in the other cases:

$$\mathit{subst}(\Gamma, x) = \mathit{subst}(\Gamma)[x \mapsto x] \qquad \mathit{subst}(\Gamma, \bar{x}_n \mapsto \bar{t}_n) = \mathit{subst}(\Gamma) \circ \{\bar{x}_n \mapsto \bar{t}_n\}$$

In the first equation, the $[x \mapsto x]$ update shadows any replacement of x induced by Γ. The examples below illustrate this subtlety:

$$\mathit{subst}(x \mapsto 7, x \mapsto \mathsf{g}(x)) = \{x \mapsto \mathsf{g}(7)\} \quad \mathit{subst}(x \mapsto 7, x, x \mapsto \mathsf{g}(x)) = \{x \mapsto \mathsf{g}(x)\}$$

We write $\Gamma(t)$ to abbreviate the capture-avoiding substitution $\mathit{subst}(\Gamma)(t)$.

Transformations of terms (and formulas) are justified by judgments of the form $\Gamma \rhd t \simeq u$, where Γ is a context, t is an unprocessed term, and u is the corresponding processed term. The free variables in t and u must appear in the context Γ. Semantically, the judgment expresses the equality of the terms $\Gamma(t)$ and u for all variables fixed by Γ. Crucially, the substitution applies only on the left-hand side of the equality.

The inference rules for the transformations covered in this report are presented below, followed by explanations.

$$\frac{}{\Gamma \rhd t \simeq u}\ \textsc{Taut}_{\mathcal{T}} \quad \text{if } \models_{\mathcal{T}} \Gamma(t) \simeq u \qquad \frac{\Gamma \rhd s \simeq t \quad \Gamma \rhd t \simeq u}{\Gamma \rhd s \simeq u}\ \textsc{Trans} \quad \text{if } \Gamma(t) = t$$

$$\frac{\big(\Gamma \rhd t_i \simeq u_i\big)_{i=1}^{n}}{\Gamma \rhd \mathsf{f}(\bar{t}_n) \simeq \mathsf{f}(\bar{u}_n)}\ \textsc{Cong} \qquad \frac{\Gamma, y, x \mapsto y \rhd \varphi \simeq \psi}{\Gamma \rhd (Qx.\,\varphi) \simeq (Qy.\,\psi)}\ \textsc{Bind} \quad \text{if } y \notin FV(Qx.\,\varphi)$$

$$\frac{\Gamma, x \mapsto (\varepsilon x.\,\varphi) \rhd \varphi \simeq \psi}{\Gamma \rhd (\exists x.\,\varphi) \simeq \psi}\ \textsc{Sko}_{\exists} \qquad \frac{\Gamma, x \mapsto (\varepsilon x.\,\neg\varphi) \rhd \varphi \simeq \psi}{\Gamma \rhd (\forall x.\,\varphi) \simeq \psi}\ \textsc{Sko}_{\forall}$$

$$\frac{\big(\Gamma \rhd r_i \simeq s_i\big)_{i=1}^{n} \quad \Gamma, \bar{x}_n \mapsto \bar{s}_n \rhd t \simeq u}{\Gamma \rhd (\text{let } \bar{x}_n \simeq \bar{r}_n \text{ in } t) \simeq u}\ \textsc{Let} \quad \text{if } \Gamma(s_i) = s_i \text{ for all } i \in [n]$$

- $\textsc{Taut}_{\mathcal{T}}$ relies on an oracle $\models_{\mathcal{T}}$ to derive arbitrary lemmas in a theory $\mathcal{T}$. In practice, the oracle will produce some kind of certificate to justify the inference. An important special case, for which we use the name REFL, is syntactic equality (up to renaming of bound variables); the side condition is then $\Gamma(t) =_{\alpha} u$. (We use $=_{\alpha}$ instead of $=$ because applying a substitution can rename bound variables.)
- TRANS needs the side condition because the term t appears both on the left-hand side of $\simeq$ (where it is subject to Γ's substitution) and on the right-hand side (where it is not). Without the side condition, the two occurrences of t in the antecedent could denote different terms.
- CONG can be used for any function symbol f, including the logical connectives.
- BIND is a congruence rule for quantifiers. The rule also justifies the renaming of the bound variable (from x to y). The side condition prevents an unwarranted variable capture. In the antecedent, the renaming is expressed by a substitution in the context. If $x = y$, the context is $\Gamma, x, x \mapsto x$, which has the same meaning as Γ, x.
- $\textsc{Sko}_{\exists}$ and $\textsc{Sko}_{\forall}$ exploit (ε_1) to replace a quantified variable with a suitable witness, simulating skolemization. We can think of the ε expression in each rule abstractly as a fresh function symbol that takes any fixed variables it depends on as arguments. In the antecedents, the replacement is performed by the context.
- LET exploits (let) to expand a 'let' expression. Again, a substitution is used. The terms $\bar{r}_n$ assigned to the variables $\bar{x}_n$ can be transformed into terms $\bar{s}_n$.

The antecedents of all the rules inspect subterms structurally, without modifying them. Modifications to the term on the left-hand side are delayed; the substitution is applied only in TAUT. This is crucial to obtain compact proofs that can be checked efficiently. Some of the side conditions may look computationally expensive, but there are ways to compute them fairly efficiently. Furthermore, by systematically renaming variables in BIND, we can satisfy most side conditions trivially, as we will prove in Sect. 5.

The set of rules can be extended to cater for arbitrary transformations that can be expressed as equalities, using Hilbert choice to represent fresh symbols if necessary. The usefulness of Hilbert choice for proof reconstruction is well known [10, 34, 37], but we push the idea further and use it to simplify the inference system and make it more uniform.

Example 1. The following derivation tree justifies the expansion of a ‘let’ expression:

$$\dfrac{\dfrac{}{\rhd\ \mathsf{a} \simeq \mathsf{a}}\ \text{\textsc{Cong}} \qquad \dfrac{\dfrac{}{x \mapsto \mathsf{a}\ \rhd\ x \simeq \mathsf{a}}\ \text{\textsc{Refl}} \quad \dfrac{}{x \mapsto \mathsf{a}\ \rhd\ x \simeq \mathsf{a}}\ \text{\textsc{Refl}}}{x \mapsto \mathsf{a}\ \rhd\ \mathsf{p}(x,x) \simeq \mathsf{p}(\mathsf{a},\mathsf{a})}\ \text{\textsc{Cong}}}{\rhd\ (\text{let } x \simeq \mathsf{a} \text{ in } \mathsf{p}(x,x)) \simeq \mathsf{p}(\mathsf{a},\mathsf{a})}\ \text{\textsc{Let}}$$

It is also possible to further process the substituted term, as in this derivation:

$$\dfrac{\dfrac{}{\rhd\ \mathsf{a} + 0 \simeq \mathsf{a}}\ \text{\textsc{Taut}}_{+} \qquad \dfrac{\vdots}{x \mapsto \mathsf{a}\ \rhd\ \mathsf{p}(x,x) \simeq \mathsf{p}(\mathsf{a},\mathsf{a})}\ \text{\textsc{Cong}}}{\rhd\ (\text{let } x \simeq \mathsf{a} + 0 \text{ in } \mathsf{p}(x,x)) \simeq \mathsf{p}(\mathsf{a},\mathsf{a})}\ \text{\textsc{Let}}$$

Example 2. The following derivation tree, in which ε_x abbreviates $\varepsilon x.\, \neg\, \mathsf{p}(x)$, justifies the skolemization of the quantifier in the formula $\neg \forall x.\, \mathsf{p}(x)$:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{x \mapsto \varepsilon_x\ \rhd\ x \simeq \varepsilon_x}\ \text{\textsc{Refl}}}{x \mapsto \varepsilon_x\ \rhd\ \mathsf{p}(x) \simeq \mathsf{p}(\varepsilon_x)}\ \text{\textsc{Cong}}}{\rhd\ (\forall x.\, \mathsf{p}(x)) \simeq \mathsf{p}(\varepsilon_x)}\ \text{\textsc{Sko}}_{\forall}}{\rhd\ (\neg \forall x.\, \mathsf{p}(x)) \simeq \neg\, \mathsf{p}(\varepsilon_x)}\ \text{\textsc{Cong}}$$

The CONG inference above $\text{\textsc{Sko}}_{\forall}$ is optional; we could have directly closed the derivation with REFL. In a prover, the term ε_x would be represented by a fresh Skolem constant c, and we would ignore c’s connection to ε_x during proof search.

Skolemization can be applied regardless of polarity. Normally, we skolemize only positive existential quantifiers and negative universal quantifiers. However, skolemizing other quantifiers is sound in the context of proving. The trouble is that it is generally incomplete, if we introduce Skolem symbols and forget their definitions in terms of Hilbert choice. To paraphrase Orwell, all quantifiers are skolemizable, but some quantifiers are more skolemizable than others.

Example 3. The next derivation tree illustrates the interplay between the theory rule $\text{\textsc{Taut}}_{\mathcal{T}}$ and the equality rules TRANS and CONG:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\rhd\ \mathsf{k} \simeq \mathsf{k}}\ \text{\textsc{Cong}} \quad \dfrac{}{\rhd\ 1 \times 0 \simeq 0}\ \text{\textsc{Taut}}_{\times}}{\rhd\ \mathsf{k} + 1 \times 0 \simeq \mathsf{k} + 0}\ \text{\textsc{Cong}} \qquad \dfrac{}{\rhd\ \mathsf{k} + 0 \simeq \mathsf{k}}\ \text{\textsc{Taut}}_{+}}{\rhd\ \mathsf{k} + 1 \times 0 \simeq \mathsf{k}}\ \text{\textsc{Trans}} \qquad \dfrac{}{\rhd\ \mathsf{k} \simeq \mathsf{k}}\ \text{\textsc{Cong}}}{\rhd\ (\mathsf{k} + 1 \times 0 < \mathsf{k}) \simeq (\mathsf{k} < \mathsf{k})}\ \text{\textsc{Cong}}$$

We could extend the tree at the bottom with an extra application of TRANS and $\text{TAUT}_<$ to simplify $\mathsf{k} < \mathsf{k}$ further to false. The example demonstrates that theories can be arbitrarily fine-grained, which often makes proof checking easier. At the other extreme, we could have derived $\rhd\ (\mathsf{k}+1\times 0 < \mathsf{k}) \simeq \mathsf{false}$ using a single $\text{TAUT}_{+\cup\times\cup<}$ inference.

Example 4. The tree below illustrates what can go wrong if we ignore side conditions:

$$\dfrac{\dfrac{\dfrac{}{\Gamma_1 \rhd \mathsf{f}(x) \simeq \mathsf{f}(x)}\,\textsc{Refl} \qquad \dfrac{\dfrac{}{\Gamma_2 \rhd x \simeq \mathsf{f}(x)}\,\textsc{Refl} \quad \dfrac{}{\Gamma_3 \rhd \mathsf{p}(y) \simeq \mathsf{p}(\mathsf{f}(\mathsf{f}(x)))}\,\textsc{Refl}}{\Gamma_2 \rhd (\text{let } y \simeq x \text{ in } \mathsf{p}(y)) \simeq \mathsf{p}(\mathsf{f}(\mathsf{f}(x)))}\,\textsc{Let}^*}{\Gamma_1 \rhd (\text{let } x \simeq \mathsf{f}(x) \text{ in let } y \simeq x \text{ in } \mathsf{p}(y)) \simeq \mathsf{p}(\mathsf{f}(\mathsf{f}(x)))}\,\textsc{Let}}{\rhd (\forall x.\ \text{let } x \simeq \mathsf{f}(x) \text{ in let } y \simeq x \text{ in } \mathsf{p}(y)) \simeq (\forall x.\ \mathsf{p}(\mathsf{f}(\mathsf{f}(x))))}\,\textsc{Bind}$$

In the above, $\Gamma_1 = x, x \mapsto x$; $\Gamma_2 = \Gamma_1, x \mapsto \mathsf{f}(x)$; and $\Gamma_3 = \Gamma_2, y \mapsto \mathsf{f}(x)$. The inference marked with an asterisk (*) is illegal, because $\Gamma_2(\mathsf{f}(x)) = \mathsf{f}(\mathsf{f}(x)) \neq \mathsf{f}(x)$. We exploit this to derive an invalid judgment, with a spurious application of f on the right-hand side. To apply LET legally, we must first rename the universally quantified variable x to a fresh variable z using the BIND rule:

$$\dfrac{\dfrac{\dfrac{}{\Gamma_1 \rhd \mathsf{f}(x) \simeq \mathsf{f}(z)}\,\textsc{Refl} \qquad \dfrac{\dfrac{}{\Gamma_2 \rhd x \simeq \mathsf{f}(z)}\,\textsc{Refl} \quad \dfrac{}{\Gamma_3 \rhd \mathsf{p}(y) \simeq \mathsf{p}(\mathsf{f}(z))}\,\textsc{Refl}}{\Gamma_2 \rhd (\text{let } y \simeq x \text{ in } \mathsf{p}(y)) \simeq \mathsf{p}(\mathsf{f}(z))}\,\textsc{Let}}{\Gamma_1 \rhd (\text{let } x \simeq \mathsf{f}(x) \text{ in let } y \simeq x \text{ in } \mathsf{p}(y)) \simeq \mathsf{p}(\mathsf{f}(z))}\,\textsc{Let}}{\rhd (\forall x.\ \text{let } x \simeq \mathsf{f}(x) \text{ in let } y \simeq x \text{ in } \mathsf{p}(y)) \simeq (\forall z.\ \mathsf{p}(\mathsf{f}(z)))}\,\textsc{Bind}$$

This time, we have $\Gamma_1 = z, x \mapsto z$; $\Gamma_2 = \Gamma_1, x \mapsto \mathsf{f}(z)$; and $\Gamma_3 = \Gamma_2, y \mapsto \mathsf{f}(z)$. LET's side condition is satisfied: $\Gamma_2(\mathsf{f}(z)) = \mathsf{f}(z)$.

Example 5. The dangers of capture are illustrated by the following tree, where ε_y stands for $\varepsilon y.\, \mathsf{p}(x) \wedge \forall x.\, \mathsf{q}(x,y)$:

$$\dfrac{\dfrac{\dfrac{}{x, y \mapsto \varepsilon_y \rhd (\mathsf{p}(x) \wedge \forall x.\, \mathsf{q}(x,y)) \simeq (\mathsf{p}(x) \wedge \forall x.\, \mathsf{q}(x,\varepsilon_y))}\,\textsc{Refl}^*}{x \rhd (\exists y.\, \mathsf{p}(x) \wedge \forall x.\, \mathsf{q}(x,y)) \simeq (\mathsf{p}(x) \wedge \forall x.\, \mathsf{q}(x,\varepsilon_y))}\,\textsc{Sko}_\exists}{\rhd (\forall x.\, \exists y.\, \mathsf{p}(x) \wedge \forall x.\, \mathsf{q}(x,y)) \simeq (\forall x.\, \mathsf{p}(x) \wedge \forall x.\, \mathsf{q}(x,\varepsilon_y))}\,\textsc{Bind}$$

The inference marked with an asterisk would be legal if REFL's side condition were stated using capturing substitution. The final judgment is unwarranted, because the free variable x in the first conjunct of ε_y is captured by the inner universal quantifier on the right-hand side.

To avoid the capture, we rename the inner bound variable x to z. Then it does not matter if we use capture-avoiding or capturing substitution:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{x, y \mapsto \varepsilon_y \rhd \mathsf{p}(x) \simeq \mathsf{p}(x)}\,\textsc{Refl} \qquad \dfrac{\dfrac{}{x, y \mapsto \varepsilon_y, x \mapsto z \rhd \mathsf{q}(x,y) \simeq \mathsf{q}(z,\varepsilon_y)}\,\textsc{Refl}}{x, y \mapsto \varepsilon_y \rhd (\forall x.\, \mathsf{q}(x,y)) \simeq (\forall z.\, \mathsf{q}(z,\varepsilon_y))}\,\textsc{Bind}}{x, y \mapsto \varepsilon_y \rhd (\mathsf{p}(x) \wedge \forall x.\, \mathsf{q}(x,y)) \simeq (\mathsf{p}(x) \wedge \forall z.\, \mathsf{q}(z,\varepsilon_y))}\,\textsc{Cong}}{x \rhd (\exists y.\, \mathsf{p}(x) \wedge \forall x.\, \mathsf{q}(x,y)) \simeq (\mathsf{p}(x) \wedge \forall z.\, \mathsf{q}(z,\varepsilon_y))}\,\textsc{Sko}_\exists}{\rhd (\forall x.\, \exists y.\, \mathsf{p}(x) \wedge \forall x.\, \mathsf{q}(x,y)) \simeq (\forall x.\, \mathsf{p}(x) \wedge \forall z.\, \mathsf{q}(z,\varepsilon_y))}\,\textsc{Bind}$$

4 Contextual Recursion

We propose a generic algorithm for term transformations, based on structural recursion. The algorithm is parameterized by a few simple plugin functions embodying the essence of the transformation. By combining compatible plugin functions, we can perform several transformations in one traversal. Transformations can depend on some context that encapsulates relevant information, such as bound variables, variable substitutions, and polarity. Each transformation can define its own notion of context that is threaded through the recursion.

The output is generated by a proof module that maintains a stack of derivation trees. The procedure $\mathit{apply}(R, n, \Gamma, t, u)$ pops n derivation trees $\bar{\mathscr{D}}_n$ from the stack and pushes the tree

$$\frac{\mathscr{D}_1 \quad \cdots \quad \mathscr{D}_n}{\Gamma \rhd t \simeq u}\, R$$

onto the stack. The plugin functions are responsible for invoking *apply* as appropriate.

4.1 The Generic Algorithm

The algorithm performs a depth-first postorder contextual recursion on the term to process. Subterms are processed first; then an intermediate term is built from the resulting subterms and is processed itself. The context Δ is updated in a transformation-specific way with each recursive call. It is abstract from the point of view of the algorithm.

The plugin functions are divided into two groups: *ctx_let*, *ctx_quant*, and *ctx_app* update the context when entering the body of a binder or when moving from a function symbol to one of its arguments; *build_let*, *build_quant*, *build_app*, and *build_var* return the processed term and produce the corresponding proof as a side effect.

function $\mathit{process}(\Delta, t)$
  match t
    case x:
      return $\mathit{build\_var}(\Delta, x)$
    case $\mathsf{f}(\bar{t}_n)$:
      $\bar{\Delta}'_n \leftarrow (\mathit{ctx\_app}(\Delta, \mathsf{f}, \bar{t}_n, i))_{i=1}^{n}$
      return $\mathit{build\_app}\big(\Delta, \bar{\Delta}'_n, \mathsf{f}, \bar{t}_n, (\mathit{process}(\Delta'_i, t_i))_{i=1}^{n}\big)$
    case $Qx.\,\varphi$:
      $\Delta' \leftarrow \mathit{ctx\_quant}(\Delta, Q, x, \varphi)$
      return $\mathit{build\_quant}(\Delta, \Delta', Q, x, \varphi, \mathit{process}(\Delta', \varphi))$
    case let $\bar{x}_n \simeq \bar{r}_n$ in t':
      $\Delta' \leftarrow \mathit{ctx\_let}(\Delta, \bar{x}_n, \bar{r}_n, t')$
      return $\mathit{build\_let}(\Delta, \Delta', \bar{x}_n, \bar{r}_n, t', \mathit{process}(\Delta', t'))$

4.2 ‘Let’ Expansion

The first instance of the contextual recursion algorithm expands ‘let’ expressions and renames bound variables systematically to avoid capture. Skolemization and theory simplification, presented below, assume that this transformation has been performed.

The context consists of a list of fixed variables and variable substitutions, as in Sect. 3. The plugin functions are as follows:

function *ctx_let*$(\Gamma,\ \bar{x}_n,\ \bar{r}_n,\ t)$
  return $\Gamma,\ \bar{x}_n \mapsto (\mathit{process}(\Gamma,\ r_i))_{i=1}^{n}$

function *build_let*$(\Gamma,\ \Gamma',\ \bar{x}_n,\ \bar{r}_n,\ t,\ u)$
  apply$(\textsc{Let},\ n{+}1,\ \Gamma,\ \text{let}\ \bar{x}_n \simeq \bar{r}_n\ \text{in}\ t,\ u)$
  return u

function *ctx_quant*$(\Gamma,\ Q,\ x,\ \varphi)$
  $y \leftarrow$ fresh variable
  return $\Gamma,\ y,\ x \mapsto y$

function *build_quant*$(\Gamma,\ \Gamma',\ Q,\ x,\ \varphi,\ \psi)$
  $y \leftarrow \Gamma'(x)$
  apply$(\textsc{Bind},\ 1,\ \Gamma,\ Qx.\,\varphi,\ Qy.\,\psi)$
  return $Qy.\,\psi$

function *ctx_app*$(\Gamma,\ \mathsf{f},\ \bar{t}_n,\ i)$
  return Γ

function *build_app*$(\Gamma,\ \bar{\Gamma}'_n,\ \mathsf{f},\ \bar{t}_n,\ \bar{u}_n)$
  apply$(\textsc{Cong},\ n,\ \Gamma,\ \mathsf{f}(\bar{t}_n),\ \mathsf{f}(\bar{u}_n))$
  return $\mathsf{f}(\bar{u}_n)$

function *build_var*$(\Gamma,\ x)$
  apply$(\textsc{Refl},\ 0,\ \Gamma,\ x,\ \Gamma(x))$
  return $\Gamma(x)$

The *ctx_let* and *build_let* functions process 'let' expressions. In *ctx_let*, the substituted terms are processed further before they are added to a substitution entry in the context. In *build_let*, the LET rule is applied and the transformed term is returned. Analogously, the *ctx_quant* and *build_quant* functions rename quantified variables systematically. This ensures that any variables that arise in the range of the substitution specified by *ctx_let* will resist capture when the substitution is applied (cf. Example 4). Finally, the *ctx_app*, *build_app*, and *build_var* functions simply reproduce the term traversal in the generated proof; they perform no transformation-specific work.

Example 6. Following up on Example 1, assume $\varphi = \text{let}\ x \simeq \mathsf{a}\ \text{in}\ \mathsf{p}(x,x)$. Given the above plugin functions, $\mathit{process}(\varnothing,\ \varphi)$ returns $\mathsf{p}(\mathsf{a},\mathsf{a})$. It is instructive to study the evolution of the stack during the execution of *process*. First, in *ctx_let*, the term a is processed recursively; the call to *build_app* pushes a nullary CONG step with succedent $\rhd\ \mathsf{a} \simeq \mathsf{a}$ onto the stack. Then the term $\mathsf{p}(x,\ x)$ is processed. For each of the two occurrences of x, *build_var* pushes a REFL step onto the stack. Next, *build_app* applies a CONG step to justify rewriting under p: The two REFL steps are popped, and a binary CONG is pushed. Finally, *build_let* performs a LET inference with succedent $\rhd\ \varphi \simeq \mathsf{p}(\mathsf{a},\mathsf{a})$ to complete the proof: The two CONG steps on the stack are replaced by the LET step. The stack now consists of a single item: the derivation tree of Example 1.

4.3 Skolemization

Our second transformation, skolemization, assumes that 'let' expressions have been expanded and bound variables have been renamed apart. The context is a pair $\Delta = (\Gamma,\ p)$, where Γ is a context as defined in Sect. 3 and p is the polarity ($+$, $-$, or ?) of the term being processed. The main plugin functions are those that manipulate quantifiers:

```
function ctx_quant((Γ, p), Q, x, φ)
  if (Q, p) ∈ {(∃,+), (∀,−)} then
    Γ′ ← Γ, x ↦ sko_term(Γ, Q, x, φ)
  else
    Γ′ ← Γ, x
  return (Γ′, p)
```

```
function build_quant((Γ, p), Δ′,
                     Q, x, φ, ψ)
  if (Q, p) ∈ {(∃,+), (∀,−)} then
    apply(SKO_Q, 1, Γ, Qx. φ, ψ)
    return ψ
  else
    apply(BIND, 1, Γ, Qx. φ, Qx. ψ)
    return Qx. ψ
```

The polarity component of the context is updated by *ctx_app*, which is not shown. For example, *ctx_app*$((\Gamma, -), \neg, \varphi, 1)$ returns $(\Gamma, +)$, because if $\neg\,\varphi$ occurs negatively in a larger formula, then φ occurs positively. The plugin functions *build_app* and *build_var* are as for 'let' expansion.

Positive occurrences of $\exists$ and negative occurrences of $\forall$ are skolemized. All other quantifiers are kept as is. The *sko_term* function returns an applied Skolem function symbol following some reasonable scheme; for example, outer skolemization [35] creates an application of a fresh function symbol to all variables fixed in the context. To comply with the inference system, the application of $\text{S}\small{\text{KO}}_\exists$ or $\text{S}\small{\text{KO}}_\forall$ in *build_quant* instructs the proof module to systematically replace the Skolem term with the corresponding ε term when outputting the proof.

Example 7. Let $\varphi = \neg\,\forall x.\ \mathsf{p}(x)$. The call *process*$((\varnothing, +), \varphi)$ skolemizes φ into $\neg\,\mathsf{p}(\mathsf{c})$, where c is a fresh Skolem constant. The initial *process* call invokes *ctx_app* on $\neg$ as the second argument, making the context negative, thereby enabling skolemization of $\forall$. The substitution $x \mapsto \mathsf{c}$ is added to the context. Applying $\text{S}\small{\text{KO}}_\forall$ instructs the proof module to replace c with $\varepsilon x.\,\neg\,\mathsf{p}(x)$. The resulting derivation tree is as in Example 2.

4.4 Theory Simplification

All kinds of theory simplification can be performed on formulas. We restrict our focus to a simple yet quite characteristic instance: the simplification of $u + 0$ and $0 + u$ to u. We assume that 'let' expressions have been expanded. The context is a list of fixed variables. The plugin functions *ctx_app* and *build_var* are as for 'let' expansion (Sect. 4.2); the remaining ones are presented below:

```
function ctx_quant(Γ, Q, x, φ)
  return Γ, x

function build_quant(Γ, Γ′, Q, x, φ, ψ)
  apply(BIND, 1, Γ, Qx. φ, Qx. ψ)
  return Qx. ψ
```

```
function build_app(Γ, Γ̄′_n, f, t̄_n, ū_n)
  apply(CONG, n, Γ, f(t̄_n), f(ū_n))
  if f(ū_n) has form u+0 or 0+u then
    apply(TAUT_+, 0, Γ, f(ū_n), u)
    apply(TRANS, 2, Γ, f(t̄_n), u)
    return u
  else
    return f(ū_n)
```

The quantifier manipulation code, in *ctx_quant* and *build_quant*, is straightforward. The interesting function is *build_app*. It first applies the CONG rule to justify rewriting the arguments. Then, if the resulting term $\mathsf{f}(\bar{u}_n)$ can be simplified further into a term u, it performs a transitive chain of reasoning: $\mathsf{f}(\bar{t}_n) \simeq \mathsf{f}(\bar{u}_n) \simeq u$.

Example 8. Let $\varphi = \mathsf{k} + 1 \times 0 < \mathsf{k}$. Assuming that the framework has been instantiated with theory simplification for additive and multiplicative identity, invoking $\mathit{process}(\varnothing, \varphi)$ returns the formula $\mathsf{k} < \mathsf{k}$. The generated derivation tree is as in Example 3.

4.5 Combinations of Transformations

Theory simplification can be implemented as a family of transformations, each member of which embodies its own set of theory-specific rewrite rules. If the union of the rewrite rule sets is confluent and terminating, a unifying implementation of *build_app* can apply the rules in any order until a fixpoint is reached. Moreover, since theory simplification modifies terms independently of the context, it is compatible with 'let' expansion and skolemization. This means that we can replace the *build_app* implementation from Sect. 4.2 or 4.3 with that of Sect. 4.4. In particular, this allows us to perform arithmetic simplification in the substituted terms of a 'let' expression in a single pass (cf. Example 1).

The combination of 'let' expansion and skolemization is less straightforward. Consider the formula $\varphi = \text{let } y \simeq \exists x.\, \mathsf{p}(x) \text{ in } y \rightarrow y$. When processing the subformula $\exists x.\, \mathsf{p}(x)$, we cannot (or at least should not) skolemize the quantifier, because it has no unambiguous polarity; indeed, the variable y occurs both positively and negatively in the 'let' expression's body. We can of course give up and perform two passes: The first pass expands 'let' expressions, and the second pass skolemizes and simplifies terms. The first pass also provides an opportunity to expand equivalences, which are problematic for skolemization.

There is also a way to perform all the transformations in a single instance of the framework. The most interesting plugin functions are *ctx_let* and *build_var*:

function $\mathit{ctx\_let}((\Gamma, p), \bar{x}_n, \bar{r}_n, t)$
  for $i = 1$ **to** n **do**
    $\mathit{apply}(\textsc{Refl}, 0, \Gamma, x_i, \Gamma(r_i))$
  $\Gamma' \leftarrow \Gamma, \bar{x}_n \mapsto (\Gamma(r_i))_{i=1}^{n}$
  return (Γ', p)

function $\mathit{build\_var}((\Gamma, p), x)$
  $\mathit{apply}(\textsc{Refl}, 0, \Gamma, x, \Gamma(x))$
  $u \leftarrow \mathit{process}((\Gamma, p), \Gamma(x))$
  $\mathit{apply}(\textsc{Trans}, 2, \Gamma, \Gamma(x), u)$
  return u

In contrast with the corresponding function for 'let' expansion (Sect. 4.2), *ctx_let* does not process the terms $\bar{r}_n$, which is reflected by the n applications of REFL, and it must thread through polarities. The call to *process* is in *build_var* instead, where it can exploit the more precise polarity information to skolemize the formula.

The *build_let* function is essentially as before. The *ctx_quant* and *build_quant* functions are as for skolemization (Sect. 4.3), except that the **else** cases rename bound variables apart (Sect. 4.2). The *ctx_app* function is as for skolemization, whereas *build_app* is as for theory simplification (Sect. 4.4).

For the formula φ given above, $\mathit{process}((\varnothing, +), \varphi)$ returns $(\exists x.\, \mathsf{p}(x)) \rightarrow \mathsf{p}(\mathsf{c})$, where c is a fresh Skolem constant. The substituted term $\exists x.\, \mathsf{p}(x)$ is put unchanged into the substitution used to expand the 'let' expression. It is processed each time y arises in the body $y \longrightarrow y$. The positive occurrence is skolemized; the negative occurrence is left as is. Using caching and a DAG representation of derivations, we can easily avoid the duplicated work that would arise if y occurred several times with positive polarity.

4.6 Scope and Limitations

Other possible instances of contextual recursion are the clause normal form (CNF) transformation and the elimination of quantifiers using one-point rules. CNF transformation is an instance of rewriting of Boolean formulas and can be justified by a $\text{T}\textsc{aut}_{\mathsf{Bool}}$ rule. Tseytin transformation can be supported by representing the introduced constants by the formulas they represent, similarly to our treatment of Skolem terms. One-point rules—e.g., the transformation of $\forall x.\ x \simeq \mathsf{a} \longrightarrow \mathsf{p}(x)$ into $\mathsf{p}(\mathsf{a})$—are similar to 'let' expansion and can be represented in much the same way in our framework. The rules for eliminating universal and existential quantifiers are as follows:

$$\frac{\Gamma \rhd s \simeq t \qquad \Gamma, x \mapsto t \rhd \varphi \simeq \psi}{\Gamma \rhd (\forall x.\ x \simeq s \longrightarrow \varphi) \simeq \psi}\ \text{1P}\textsc{t}_\forall \quad \text{if } x \notin FV(s) \text{ and } \Gamma(t) = t$$

$$\frac{\Gamma \rhd s \simeq t \qquad \Gamma, x \mapsto t \rhd \varphi \simeq \psi}{\Gamma \rhd (\exists x.\ x \simeq s \land \varphi) \simeq \psi}\ \text{1P}\textsc{t}_\exists \quad \text{if } x \notin FV(s) \text{ and } \Gamma(t) = t$$

The plugin functions used by *process* would also be similar as those for 'let' expansion, except that detecting the assignment at *ctx_quant* requires examining the body of the quantified formula to determine whether the one-point rule is applicable.

Some transformations, such as symmetry breaking [14] and rewriting based on global assumptions, require a global analysis of the problem that cannot be captured by local substitution of equals for equals. They are beyond the scope of the framework. Other transformations, such as simplification based on associativity and commutativity of function symbols, require traversing the terms to be simplified when applying the rewriting. Since *process* visits terms in postorder, the complexity of the simplifications would be quadratic, while a processing that applies depth-first preorder traversal can perform the simplifications with a linear complexity. Hence, applying such transformations optimally is also outside the scope of the framework.

5 Theoretical Properties

Before proving any properties of contextual recursion or proof checking, we establish the soundness of the inference rules they rely on. We start by encoding the judgments in a well-understood theory of binders: the simply typed λ-calculus. A context and a term will be encoded together as a single λ-term. We call these somewhat nonstandard λ-terms *metaterms*. They are defined by the grammar

$$M ::= \boxed{t} \mid \lambda x.\ M \mid (\lambda \bar{x}_n.\ M)\ \bar{t}_n$$

where x_i and t_i are of the same sort for each $i \in [n]$. A metaterm is either a term t decorated with a box $\boxed{}$, a λ-abstraction, or the application of an n-tuple of terms to an uncurried λ-abstraction that simultaneously binds n distinct variables. We let $=_{\alpha\beta}$ denote syntactic equality modulo α- and β-equivalence (i.e., up to renaming of bound variables and reduction of applied λ-abstractions). We use the letters M, N, P to refer to metaterms. The notion of type is as expected for simply typed λ-terms: The type of $\boxed{t}$ is the sort of t; the type of $\lambda x.\, M$ is $\sigma \to \tau$, where σ is the sort of x and τ the type of M; and the type of $(\lambda \bar{x}_n.\, M)\ \bar{t}_n$ is the type of M. It is easy to see that all metaterms contain exactly one boxed term. $M[t]$ denotes a metaterm whose box contains t, and $M[N]$ denotes the same metaterm after its box has been replaced with the metaterm N.

Encoded judgments will have the form $M \simeq N$. The λ-abstractions and applications represent the context, whereas the box stores the term. To invoke the theory oracle $\models_{\mathscr{T}}$, we will need to *reify* equalities on metaterms—i.e., map them to equalities on terms. Let M, N be metaterms of type $\sigma_1 \to \cdots \to \sigma_n \to \sigma$. We define $\mathit{reify}(M \simeq N)$ as $\forall \bar{x}_n.\, t \simeq u$, where $M =_{\alpha\beta} \lambda x_1 \ldots \lambda x_n.\, \boxed{t}$ and $N =_{\alpha\beta} \lambda x_1 \ldots \lambda x_n.\, \boxed{u}$. Thanks to basic properties of the λ-calculus, t and u are always defined uniquely up to the names of the bound variables. For example, if $M = \lambda u.\, (\lambda v.\, \boxed{\mathsf{p}(v)})\ u$ and $N = \lambda w.\, \boxed{\mathsf{q}(w)}$, we have $M =_{\alpha\beta} \lambda x.\, \boxed{\mathsf{p}(x)}$ and $N =_{\alpha\beta} \lambda x.\, \boxed{\mathsf{q}(x)}$, and the reification of $M \simeq N$ is $\forall x.\, \mathsf{p}(x) \simeq \mathsf{q}(x)$.

The inference rules presented in Sect. 3 can now be encoded as follows. We refer to these new rules collectively as the *encoded inference system*:

$$\frac{}{M \simeq N}\ \textsc{Taut}_{\mathscr{T}} \quad \text{if } \models_{\mathscr{T}} \mathit{reify}(M \simeq N)$$

$$\frac{M \simeq N \qquad N' \simeq P}{M \simeq P}\ \textsc{Trans} \quad \text{if } N =_{\alpha\beta} N' \qquad\qquad \frac{\big(M[t_i] \simeq N[u_i]\big)_{i=1}^{n}}{M[\mathsf{f}(\bar{t}_n)] \simeq N[\mathsf{f}(\bar{u}_n)]}\ \textsc{Cong}$$

$$\frac{M[\lambda y.\, (\lambda x.\, \varphi)\ y] \simeq N[\lambda y.\, \psi]}{M[Qx.\, \varphi] \simeq N[Qy.\, \psi]}\ \textsc{Bind} \quad \text{if } y \notin FV(Qx.\, \varphi)$$

$$\frac{M[(\lambda x.\, \varphi)\ (\varepsilon x.\, \varphi)] \simeq N}{M[\exists x.\, \varphi] \simeq N}\ \textsc{Sko}_{\exists} \qquad\qquad \frac{M[(\lambda x.\, \varphi)\ (\varepsilon x.\, \neg\varphi)] \simeq N}{M[\forall x.\, \varphi] \simeq N}\ \textsc{Sko}_{\forall}$$

$$\frac{\big(M[r_i] \simeq N[s_i]\big)_{i=1}^{n} \quad M[(\lambda \bar{x}_n.\, t)\ \bar{s}_n] \simeq N[u]}{M[\text{let } \bar{x}_n \simeq \bar{r}_n \text{ in } t] \simeq N[u]}\ \textsc{Let} \quad \text{if } M[s_i] =_{\alpha\beta} N[s_i] \text{ for all } i \in [n]$$

Lemma 1. *If the judgment $M \simeq N$ is derivable using the encoded inference system with the theories $\mathscr{T}_1, \ldots, \mathscr{T}_n$, then $\models_{\mathscr{T}} \mathit{reify}(M \simeq N)$ with $\mathscr{T} = \mathscr{T}_1 \cup \cdots \cup \mathscr{T}_n \cup \simeq \cup\, \varepsilon \cup \text{let}$.*

Proof. By structural induction on the derivation of $M \simeq N$. For each inference rule, we assume $\models_{\mathscr{T}} \mathit{reify}(M_i \simeq N_i)$ for each judgment $M_i \simeq N_i$ in the antecedent and show that $\models_{\mathscr{T}} \mathit{reify}(M \simeq N)$. Most of the cases implicitly depend on basic properties of the λ-calculus to reason about *reify*.

CASE $\textsc{Taut}_{\mathscr{T}'}$: Trivial because $\mathscr{T}' \subseteq \mathscr{T}$ by definition of $\mathscr{T}$.

CASES TRANS, CONG, AND BIND: These follow from the theory of equality ($\simeq$).

CASES $\textsc{Sko}_{\exists}$, $\textsc{Sko}_{\forall}$, AND LET: These follow from (ε_1) and (ε_2) or (let) and from the congruence of equality. ◻

A judgment $\Gamma \rhd t \simeq u$ is encoded as $L(\Gamma)[t] \simeq R(\Gamma)[u]$, where

$$\begin{aligned} L(\varnothing)[t] &= \boxed{t} & R(\varnothing)[u] &= \boxed{u} \\ L(x, \Gamma)[t] &= \lambda x.\, L(\Gamma)[t] & R(x, \Gamma)[u] &= \lambda x.\, R(\Gamma)[u] \\ L(\bar{x}_n \mapsto \bar{s}_n, \Gamma)[t] &= (\lambda \bar{x}_n.\, L(\Gamma)[t])\ \bar{s}_n & R(\bar{x}_n \mapsto \bar{s}_n, \Gamma)[u] &= R(\Gamma)[u] \end{aligned}$$

The L function encodes the substitution entries of Γ as λ-abstractions applied to tuples. Reducing the applied λ-abstractions effectively applies $\mathit{subst}(\Gamma)$. For example:

$$L(x \mapsto 7,\, x \mapsto \mathsf{g}(x))[x] = (\lambda x.\, (\lambda x.\, \boxed{x})\, (\mathsf{g}(x)))\; 7 =_{\alpha\beta} \boxed{\mathsf{g}(7)}$$
$$L(x \mapsto 7,\, x,\, x \mapsto \mathsf{g}(x))[x] = (\lambda x.\, \lambda x.\, (\lambda x.\, \boxed{x})\, (\mathsf{g}(x)))\; 7 =_{\alpha\beta} \lambda x.\, \boxed{\mathsf{g}(x)}$$

For any derivable judgment $\Gamma \rhd t \simeq u$, the terms t and u must have the same sort, and the metaterms $L(\Gamma)[t]$ and $R(\Gamma)[u]$ must have the same type. Another property is that $L(\Gamma)[t]$ is of the form $M[t]$ for some M that is independent of t and similarly for $R(\Gamma)[u]$, motivating the suggestive brackets around L's and R's term argument.

Lemma 2. *Let $\bar{x}_n$ be the variables fixed by Γ in order of occurrence. Then $L(\Gamma)[t] =_{\alpha\beta} \lambda x_1. \ldots \lambda x_n.\, \boxed{\Gamma(t)}$.*

Proof. By induction on Γ.

CASE $\varnothing$: $L(\varnothing)[t] = \boxed{t} = \boxed{\varnothing(t)}$.

CASE x, Γ: Let $\bar{y}_n$ be the variables fixed by Γ.

$$\begin{aligned} L(x,\Gamma)[t] &= \lambda x.\, L(\Gamma)[t] \\ &=_{\alpha\beta} \lambda x. \lambda y_1. \ldots \lambda y_n.\, \boxed{\Gamma(t)} && \{\text{by the induction hypothesis}\} \\ &= \lambda x. \lambda y_1. \ldots \lambda y_n.\, \boxed{(x,\Gamma)(t)} && \{\text{by } (*)\} \end{aligned}$$

where $(*)$ is the property that $\mathit{subst}(\Gamma) = \mathit{subst}(x,\Gamma)$ for all x and Γ, which is easy to prove by structural induction on Γ.

CASE $\bar{x}_n \mapsto \bar{s}_n, \Gamma$: Let $\bar{y}_n$ be the variables fixed by Γ, and let $\rho = \{\bar{x}_n \mapsto \bar{s}_n\}[\bar{y}_n \mapsto \bar{y}_n]$.

$$\begin{aligned} L(\bar{x}_n \mapsto \bar{s}_n, \Gamma)[t] &= (\lambda \bar{x}_n.\, L(\Gamma)[t])\; \bar{s}_n \\ &=_{\alpha\beta} (\lambda \bar{x}_n. \lambda y_1. \ldots \lambda y_n.\, \boxed{\Gamma(t)})\; \bar{s}_n && \{\text{by the induction hypothesis}\} \\ &=_{\alpha\beta} \lambda y_1. \ldots \lambda y_n.\, \boxed{\rho(\Gamma(t))} && \{\text{by } \beta\text{-reduction}\} \\ &= \lambda y_1. \ldots \lambda y_n.\, \boxed{(\bar{x}_n \mapsto \bar{s}_n, \Gamma)(t)} && \{\text{by } (**)\} \end{aligned}$$

where $(**)$ is the property that $\rho \circ \mathit{subst}(\Gamma) = \mathit{subst}(\bar{x}_n \mapsto \bar{s}_n, \Gamma)$ for all $\bar{x}_n$, $\bar{s}_n$, and Γ, which is easy to prove by structural induction on Γ. ◻

Lemma 3. *If the judgment $\Gamma \rhd t \simeq u$ is derivable using the original inference system, then $L(\Gamma)[t] \simeq R(\Gamma)[u]$ is derivable using the encoded inference system.*

Proof. By structural induction on the derivation of $\Gamma \rhd t \simeq u$.

CASE TAUT$_{\mathcal{T}}$: We have $\models_{\mathcal{T}} \Gamma(t) \simeq u$. Using Lemma 2, we can easily show that $\models_{\mathcal{T}} \Gamma(t) \simeq u$ is equivalent to $\models_{\mathcal{T}} \mathit{reify}(L(\Gamma)[t] \simeq R(\Gamma)[u])$, the side condition of the encoded TAUT$_{\mathcal{T}}$ rule.

CASE BIND: The encoded antecedent is $M[\lambda y.\, (\lambda x.\, \varphi)\; y] \simeq N[\lambda y.\, \psi]$ (i.e., $L(\Gamma, y, x \mapsto y)$ $[\varphi] \simeq R(\Gamma, y, x \mapsto y)[\psi]$), and the encoded succedent is $M[Qx.\, \varphi] \simeq N[Qy.\, \psi]$. By the induction hypothesis, the encoded antecedent is derivable. Thus, by the encoded BIND rule, the encoded succedent is derivable.

CASES CONG, SKO$_\exists$, AND SKO$_\forall$: Similar to BIND.

CASE TRANS: If $\Gamma(t) = t$, the substitution entries of Γ affect only variables that do not occur free in t. Hence, $R(\Gamma)[t] =_{\alpha\beta} L(\Gamma)[t]$, as required by the encoded TRANS rule.

CASE LET: Similar to TRANS. □

Incidentally, the converse of Lemma 3 does not hold, since the encoded inference rules allow metaterms that contain applied λ-abstractions on the right-hand side of $\simeq$.

Theorem 4 (Soundness of Inferences). *If the judgment $\Gamma \rhd t \simeq u$ is derivable using the original inference system with the theories $\mathscr{T}_1, \ldots, \mathscr{T}_n$, then $\models_{\mathscr{T}} \Gamma(t) \simeq u$ with $\mathscr{T} = \mathscr{T}_1 \cup \cdots \cup \mathscr{T}_n \cup \simeq \cup\, \varepsilon \cup$ let.*

Proof. This follows from Lemmas 1 and 3. The equivalence of $\models_{\mathscr{T}} \Gamma(t) \simeq u$ and $\models_{\mathscr{T}}$ $\mathit{reify}(L(\Gamma)[t] \simeq R(\Gamma)[u])$ can be established along the lines of case TAUT$_{\mathscr{T}}$ of Lemma 3. □

We turn to the contextual recursion algorithm that generates derivations in that system. The first question is, *Are the derivation trees valid?* In particular, it is not obvious from the code that the side conditions of the inference rules are always satisfied. First, we need to introduce some terminology. A term is *shadowing-free* if nested binders always bind variables with different names; for example, $\forall x.\ (\forall y.\ \mathsf{p}(x, y)) \wedge (\forall y.\ \mathsf{q}(y))$ is shadowing-free, while $\forall x.\ (\forall x.\ \mathsf{p}(x, y)) \wedge (\forall y.\ \mathsf{q}(y))$ is not. The set of variables *fixed* by Γ is written $\mathit{fix}(\Gamma)$, and the set of variables *replaced* by Γ is written $\mathit{repl}(\Gamma)$. They are defined as follows:

$$\begin{aligned} \mathit{fix}(\varnothing) &= \{\} & \mathit{repl}(\varnothing) &= \{\} \\ \mathit{fix}(\Gamma, x) &= \{x\} \cup \mathit{fix}(\Gamma) & \mathit{repl}(\Gamma, x) &= \mathit{repl}(\Gamma) \\ \mathit{fix}(\Gamma, \bar{x}_n \mapsto \bar{s}_n) &= \mathit{fix}(\Gamma) & \mathit{repl}(\Gamma, \bar{x}_n \mapsto \bar{s}_n) &= \{x_i \mid s_i \neq x_i\} \cup \mathit{repl}(\Gamma) \end{aligned}$$

Trivial substitutions $x \mapsto x$ are ignored, since they have no effect. The set of variables *introduced* by Γ is defined by $\mathit{intr}(\Gamma) = \mathit{fix}(\Gamma) \cup \mathit{repl}(\Gamma)$. A context Γ is *consistent* if all the fixed variables are mutually distinct and the two sets of variables are disjoint—i.e., $\mathit{fix}(\Gamma) \cap \mathit{repl}(\Gamma) = \{\}$.

A judgment $\Gamma \rhd t \simeq u$ is *canonical* if Γ is consistent, $FV(t) \subseteq \mathit{intr}(\Gamma)$, $FV(u) \subseteq \mathit{fix}(\Gamma)$, and $BV(u) \cap \mathit{intr}(\Gamma) = \{\}$. The *canonical inference system* is a variant of the system of Sect. 3 in which all judgments are canonical and rules TRANS, BIND, and LET have no side conditions.

Lemma 5. *Any inference in the canonical inference system is also an inference in the original inference system.*

Proof. It suffices to show that the side conditions of the original rules are satisfied.

CASE TRANS: Since the first judgment in the antecedent is canonical, $FV(t) \subseteq \mathit{fix}(\Gamma)$. By consistency of Γ, we have $\mathit{fix}(\Gamma) \cap \mathit{repl}(\Gamma) = \{\}$. Hence, $FV(t) \cap \mathit{repl}(\Gamma) = \{\}$ and therefore $\Gamma(t) = t$.

CASE BIND: Since the succedent is canonical, we have (1) $FV(Qx.\,\varphi) \subseteq \mathit{intr}(\Gamma)$ and (2) $BV(Qy.\,\psi) \cap \mathit{intr}(\Gamma) = \{\}$. From (2), we deduce $y \notin \mathit{intr}(\Gamma)$. Hence, by (1), we get $y \notin FV(Qx.\,\varphi)$.

CASE LET: Similar to TRANS. □

Theorem 6 (Total Correctness of Recursion). *For the instances presented in Sects. 4.2 to 4.4, the contextual recursion algorithm always produces correct proofs.*

Proof. The algorithm terminates because *process* is called initially on a finite input and recursive calls always have smaller inputs.

For the proof of partial correctness, only the Γ part of the context is relevant. We will write $\mathit{process}(\Gamma, t)$ even if the first argument actually has the form (Γ, p) for skolemization. The pre- and postconditions of a $\mathit{process}(\Gamma, t)$ call that returns the term u are

PRE1 Γ is consistent;	POST1 u is shadowing-free;
PRE2 $FV(t) \subseteq \mathit{intr}(\Gamma)$;	POST2 $FV(u) \subseteq \mathit{fix}(\Gamma)$;
PRE3 $BV(t) \cap \mathit{fix}(\Gamma) = \{\}$;	POST3 $BV(u) \cap \mathit{intr}(\Gamma) = \{\}$.

For skolemization and simplification, we may additionally assume that bound variables have been renamed apart by 'let' expansion, and hence that the term t is shadowing-free.

The initial call $\mathit{process}(\varnothing, t)$ trivially satisfies the preconditions on an input term t that contains no free variable. We must show that the preconditions for each recursive call $\mathit{process}(\Gamma', t')$ are satisfied and that the postconditions hold at the end of $\mathit{process}(\Gamma, t)$.

PRE1 (Γ' is consistent): First, we show that the fixed variables are mutually distinct. For 'let' expansion, all fixed variables are fresh. For skolemization and simplification, a precondition is that the input is shadowing-free. For any two fixed variables in Γ', the input formula must contain two quantifiers, one in the scope of the other. Hence, the variables must be distinct. Second, we show that $\mathit{fix}(\Gamma') \cap \mathit{repl}(\Gamma') = \{\}$. For 'let' expansion, all fixed variables are fresh. For skolemization, the condition is a direct consequence of the precondition that the input is shadowing-free. For simplification, we have $\mathit{repl}(\Gamma') = \{\}$.

PRE2 ($FV(t') \subseteq \mathit{intr}(\Gamma')$): We have $FV(t) \subseteq \mathit{intr}(\Gamma)$. The desired property holds because the *ctx_let* and *ctx_quant* functions add to the context any bound variables that become free when entering the body t' of a binder.

PRE3 ($BV(t') \cap \mathit{fix}(\Gamma') = \{\}$): The only function that fixes variable is *ctx_quant*. For 'let' expansion, all fixed variables are fresh. For skolemization and simplification, the condition is a consequence of the shadowing-freedom of the input.

POST1 (u is shadowing-free): The only function that builds quantifiers is *build_quant*. The $\mathit{process}(\Gamma', \varphi)$ call that returns the processed body ψ of the quantifier is such that $y \in \mathit{intr}(\Gamma')$ in the 'let' expansion case and $x \in \mathit{intr}(\Gamma')$ in the other two cases. The induction hypothesis ensures that ψ is shadowing-free and $BV(\psi) \cap \mathit{intr}(\Gamma') = \{\}$; hence, the result of *build_quant* (i.e., $Qy.\,\psi$ or $Qx.\,\psi$) is shadowing-free. Quantifiers can also emerge when applying a substitution in *build_var*. This can happen only if *ctx_let* has added a substitution entry to the context, in which case the substituted term is the result of a call to *process* and is thus shadowing-free.

POST2 ($FV(u) \subseteq \mathit{fix}(\Gamma)$): In most cases, this condition follows directly from the induction hypothesis POST2. The only case where a variable appears fixed in the context Γ' of a recursive call to *process* and not in Γ is when processing a quantifier, and then that variable is bound in the result. For variable substitution, it suffices to realize that the context in which the substituted term is created (and which fixes all the free variables of the term) is a prefix of the context when the substitution occurs.

POST3 ($BV(u) \cap intr(\Gamma) = \{\}$): In most cases, this condition follows directly from the induction hypothesis POST3: For every recursive call, $intr(\Gamma) \subseteq intr(\Gamma')$. Two cases require attention. For 'let' expansion, a variable may be replaced by a term with bound variables. Then the substituted term only contains variables that do not occur in the input. The variables introduced by Γ will be other fresh variables or variables from the input. The second case is when a quantified formula is built. For 'let' expansion, a fresh variable is used. For skolemization and simplification, we have $BV(Qx.\,\varphi) \cap fix(\Gamma) = \{\}$ (PRE3); hence $x \notin fix(\Gamma)$. Finally, we must show that $x \notin repl(\Gamma)$; this is a consequence of the shadowing-freedom of the input.

It is easy to see that each *apply* call generates a rule with an antecedent and a succedent of the right form, ignoring the rules' side conditions. Moreover, all calls to *apply* generate canonical judgments thanks to the pre- and postconditions proved above. Correctness follows from Lemma 5. □

Observation 7 (Complexity of Recursion). *For the instances presented in Sects. 4.2 to 4.4, the 'process' function is called at most once on every subterm of the input.*

Justification. It suffices to notice that a call to $process(\Delta, t)$ induces at most one call for each of the subterms in t. □

As a corollary, if all the operations performed in *process* excluding the recursive calls can be accomplished in constant time, the algorithm has linear-time complexity with respect to the input. There exist data structures for which the following operations take constant time: extending the context with a fixed variable or a substitution, accessing direct subterms of a term, building a term from its direct subterms, choosing a fresh variable, applying a context to a variable, checking if a term matches a simple template, and associating the parameters of the template with the subterms. Thus, it is possible to have a linear-time algorithm for 'let' expansion and simplification.

On the other hand, construction of Skolem terms is at best linear in the size of the context and of the input formula in *process*. Hence, skolemization is at best quadratic in the worst case. This is hardly surprising because in general, the formula $\forall x_1.\,\exists y_1.\ldots\forall x_n.\,\exists y_n.\ \varphi[\bar{x}_n, \bar{y}_n]$, whose size is proportional to n, is translated to $\forall x_1.\ldots\forall x_n.\ \varphi[\bar{x}_n, \mathsf{f}_1(\bar{x}_1), \mathsf{f}_2(\bar{x}_2), \ldots, \mathsf{f}_n(\bar{x}_n)]$, whose size is quadratic in n.

Observation 8 (Overhead of Proof Generation). *For the instances presented in Sects. 4.2 to 4.4, the number of calls to the 'apply' procedure is proportional to the number of subterms in the input.*

Justification. This is a corollary of Observation 7, since the number of *apply* calls per *process* call is bounded by a constant (3, in *build_app* for simplification). □

Notice that all arguments to *apply* must be computed regardless of the *apply* calls. If an *apply* call takes constant time, the proof generation overhead is linear in the size of the input. To achieve this performance, it is necessary to use sharing to represent contexts and terms in the output; otherwise, each call to *apply* might itself be linear in the size of its arguments, resulting in a nonlinear overhead on the generation of the entire proof.

Observation 9 (Cost of Proof Checking). *Checking an inference step can be performed in constant time if checking the side condition takes constant time.*

Justification. The inference rules involve only shallow conditions on contexts and terms, except in the side conditions. Using suitable data structures with maximal sharing, the contexts and terms can be checked in constant time. □

The above statement may appear weak, since checking the side conditions might itself be linear, leading to a cost of proof checking that can be at least quadratic in the size of the proof (measured as the number of symbols that represent it). Fortunately, most of the side conditions can be checked efficiently. For example, for simplification (Sect. 4.4), the BIND rule is always applied with $x = y$, which implies the side condition $y \notin FV(Qx.\,\varphi)$; and since no other rule contributes to the substitution, $subst(\Gamma)$ is the identity. Thus, simplification proofs can be checked in linear time. Moreover, certifying a proof by checking each step locally is not the only possibility. An alternative is to use an algorithm similar to the *process* function to check a proof in the same way as it has been produced. Such an algorithm can exploit sophisticated invariants on the contexts and terms.

6 Implementation

The ideas presented in this report have been implemented in two tools. We implemented the contextual recursion algorithm and the transformations described in Sect. 4 in the SMT solver veriT [11], showing that replacing the previous ad hoc code with the generic proof-producing framework had no significant detrimental impact on the solving times. In addition, we developed a prototypical proof checker for the inference system described in Sect. 3 using Isabelle/HOL [32], to convince ourselves that veriT's output can easily be reconstructed.

6.1 Isabelle

The Isabelle/HOL proof assistant is based on classical higher-order logic (HOL) [18], a variant of the simply typed λ-calculus. Thanks to the availability of λ-terms, we could follow the lines of the encoded inference system of Sect. 5 to represent judgments in HOL. The proof checker is included in the development version of Isabelle.[1]

Derivations are represented by a recursive datatype in Standard ML, Isabelle's primary implementation language. A derivation is a tree whose nodes are labeled by rule names. Rule TAUT$_{\mathcal{T}}$ additionally carries a theorem that represents the oracle $\models_{\mathcal{T}}$, and rules TRANS and LET are labeled with the terms that occur only in the antecedent (t and $\bar{s}_n$). Terms and metaterms are translated to HOL terms, and judgments $M \simeq N$ are translated to HOL equalities $t \simeq u$, where t and u are HOL terms. Uncurried λ-applications are encoded using a polymorphic combinator $\mathsf{case}_{\times} : (\alpha \to \beta \to \gamma) \to$

[1] `http://isabelle.in.tum.de/repos/isabelle/file/00731700e54f/src/HOL/ex/veriT_Preprocessing.thy`

$\alpha \times \beta \to \gamma$; in Isabelle/HOL, $\lambda(x,y).\,t$ is syntactic sugar for $\mathsf{case}_\times\ (\lambda x.\lambda y.\,t)$. This scheme is iterated to support n-tuples, represented by nested pairs $(t_1, (\cdots(t_{n-1}, t_n)\cdots))$.

To implement the inference rules, it is necessary to be able to locate any metaterm's box. There is an easy criterion: Translated metaterms are of the form $(\lambda\ldots.)\ldots$ or $\mathsf{case}_\times\ldots$, which is impossible for a translated term. Because reconstruction is not verified, there are no guarantees that it will always succeed, but when it does, the result is certified by Isabelle's LCF-style inference kernel [19]. We hard-coded a few dozen examples to test different cases, such as this one: Given the HOL terms $t = \neg\,\forall x.\ \mathsf{p} \wedge \exists x.\ \forall x.\ \mathsf{q}\ x\ x$ and $u = \neg\,\forall x.\ \mathsf{p} \wedge \exists x.\ \mathsf{q}\ (\varepsilon x.\,\neg\,\mathsf{q}\ x\ x)\ (\varepsilon x.\,\neg\,\mathsf{q}\ x\ x)$ and the ML tree

$$\mathsf{N}\ (\mathsf{Cong}, [\mathsf{N}\ (\mathsf{Bind}, [\mathsf{N}\ (\mathsf{Cong}, [\mathsf{N}\ (\mathsf{Refl}, []), \mathsf{N}\ (\mathsf{Bind}, [\mathsf{N}\ (\mathsf{Sko\_All}, [\mathsf{N}\ (\mathsf{Refl}, [])])])])])])$$

the reconstruction function returns the HOL theorem $t \simeq u$.

6.2 veriT

We implemented the contextual recursion framework in the SMT solver veriT,[2] replacing large parts of the previous non-proof-producing, hard-to-maintain code. Even though it offers more functionality (proof generation), the preprocessing module is about 20% smaller than before and consists of about 3000 lines of code. There are now only two traversal functions instead of 10. This is, for us, a huge gain in maintainability.

Proofs Previously, veriT provided detailed proofs for the resolution steps performed by the SAT solver and the lemmas added by the theory solvers and instantiation module. All transformations performed in preprocessing steps were represented in the proof in a very coarse manner, amounting to gaps in the proof. For example, when 'let' expressions were expanded in a formula, the only information present in the proof would be the formula before and after 'let' expansion.

When extending veriT to generate more detailed proofs, we were able to reuse its existing proof module and proof format [7]. A proof is a list of inferences, each of which consists of an identifier (e.g., .c0), the name of the rule, the identifiers of the dependencies, and the derived clause. The use of identifiers makes it possible to represent proofs as DAGs. We extended the format with the inference rules of Sect. 3. The rules that augment the context take a sequence of inferences—a *subproof*—as a justification. The subproof occurs within the scope of the extended context. Following this scheme, the skolemization proof for the formula $\neg\forall x.\ \mathsf{p}(x)$ from Example 2 is presented as

(.c0 (Sko_All :conclusion $((\forall x.\ \mathsf{p}(x)) \simeq \mathsf{p}(\varepsilon x.\,\neg\,\mathsf{p}(x)))$
  :args $(x \mapsto (\varepsilon x.\,\neg\,\mathsf{p}(x)))$
  :subproof ((.c1 (Refl :conclusion $(x \simeq (\varepsilon x.\,\neg\,\mathsf{p}(x)))))$
    (.c2 (Cong :clauses (.c1) :conclusion $(\mathsf{p}(x) \simeq \mathsf{p}(\varepsilon x.\,\neg\,\mathsf{p}(x)))))))$
(.c3 (Cong :clauses (.c0) :conclusion $((\neg\forall x.\ \mathsf{p}(x)) \simeq \neg\,\mathsf{p}(\varepsilon x.\,\neg\,\mathsf{p}(x)))))$

Formerly, no details of these transformations would be recorded. The proof would have contained only the original formula and the skolemized result, regardless of how many quantifiers appeared in the formula.

[2] `http://matryoshka.gforge.inria.fr/pubs/processing/veriT.tar.gz`

In contrast with the abstract proof module described in Sect. 4, veriT leaves REFL steps implicit for judgments of the form $\Gamma \rhd t \simeq t$. The other inference rules are generalized to cope with missing REFL judgments. In addition, when printing proofs, the proof module can automatically replace terms in the inferences with some other terms. This is necessary for transformations such as skolemization and 'if–then–else' elimination. We must apply a substitution in the replaced term if the original term contains variables. In veriT, efficient data structures are available to perform this.

Transformations The implementation of contextual recursion uses a single global context, augmented before processing a subterm and restored afterwards. The context consists of a set of fixed variables, a substitution, and a polarity. In our setting, the substitution satisfies the side conditions by construction. If the context is empty, the result of processing a subterm is cached. For skolemization, a separate cache is used for each polarity. No caching is attempted under binders.

Invoking *process* on a term returns the identifier of the inference at the root of its transformation proof in addition to the processed term. These identifiers are threaded through the recursion to connect the proof. The proofs produced by instances of contextual recursion are inserted into the larger resolution proof produced by veriT. This is achieved through an inference of the form

$$\dfrac{\varphi \qquad \mathscr{D} \qquad \overline{\neg\,(\varphi \simeq \psi) \vee \neg\,\varphi \vee \psi}\ \textsc{Taut}_{\simeq}}{\psi}\ \textsc{Resolve}$$

where φ is the original formula, ψ is the processed formula, and $\mathscr{D}$ is a derivation of $\rhd\, \varphi \simeq \psi$. The derivation $\mathscr{D}$ may itself depend on instances of rule $\textsc{Taut}_{\mathscr{T}}$, each with its own proof of the side condition that must also be included in the overall proof.

Transformations performing theory simplification were straightforward to port to the new framework: Their *build_app* functions simply apply rewrite rules until a fixpoint is reached. Porting transformations that interact with binders required special attention in handling the context and producing proofs. Fortunately, most of these aspects are captured by the inference system and the abstract contextual recursion framework, where they can be studied independently of the implementation.

Some transformations are performed outside of the framework. Proofs of CNF transformation are expressed using the inference rules of veriT's underlying SAT solver, so that any tool that can reconstruct SAT proofs can also reconstruct these proofs. Simplification based on associativity and commutativity of function symbols is implemented as a dedicated procedure, for efficiency reasons (Sect. 4.6). It currently produces coarse-grained proofs.

Evaluation To evaluate the impact of the new contextual recursion algorithm and of producing detailed proofs, we compare the performance of different configurations of veriT. Our experimental data is available online.[3] We distinguish three configurations. BASIC only applies transformations for which the old code provided some (coarse-grained) proofs. EXTENDED also applies transformations for which the old code did not

[3] `http://matryoshka.gforge.inria.fr/pubs/processing/`

provide any proofs, whereas the new code provides detailed proofs. COMPLETE applies all transformations available, regardless of whether they produce proofs.

More specifically, BASIC applies the transformations for 'let' expansion, skolemization, elimination of quantifiers based on one-point rules, elimination of 'if–then–else', theory simplification for rewriting n-ary symbols as binary, and elimination of equivalences and exclusive disjunctions with quantifiers in subterms. EXTENDED adds Boolean and arithmetic simplifications to the transformations performed by BASIC. COMPLETE performs global rewriting simplifications and symmetry breaking in addition to the transformations in EXTENDED.

The evaluation was carried out on two main sets of benchmarks from SMT-LIB [5]: the 20 916 benchmarks in the quantifier-free (QF) categories QF_ALIA, QF_AUFLIA, QF_IDL, QF_LIA, QF_LRA, QF_RDL, QF_UF, QF_UFIDL, QF_UFLIA, and QF_UFLRA; and the 30 250 benchmarks labeled as unsatisfiable in the non-QF categories AUFLIA, AUFLIRA, UF, UFIDL, UFLIA, and UFLRA. The categories with bit vectors and nonlinear arithmetic are not supported by veriT. Our experiments were conducted on servers equipped with two Intel Xeon E5-2630 v3 processors, with eight cores per processor, and 126 GB of memory. Each run of the solver uses a single core. The time limit was set to 30 s, a reasonable value for interactive use within a proof assistant.

The tables below indicate the number of benchmark problems solved by each configuration for the quantifier-free and non-quantifier-free benchmarks:

QF		Old code	New code
BASIC	without proofs	13 489	13 496
	with proofs	13 360	13 352
EXTENDED	without proofs	13 539	13 537
	with proofs	N/A	13 414
COMPLETE	without proofs	13 826	13 819
	with proofs	N/A	N/A

NON-QF		Old code	New code
BASIC	without proofs	28 746	28 762
	with proofs	28 744	28 766
EXTENDED	without proofs	28 785	28 852
	with proofs	N/A	28 857
COMPLETE	without proofs	28 759	28 794
	with proofs	N/A	N/A

These results indicate that the new generic contextual recursion algorithm and the production of detailed proofs do not impact performance negatively in any significant way compared with the old code. The time difference is less than 0.1%, and the small changes in solved problems are within the difference one can observe when renaming symbols or reordering axioms. In addition, fine-grained proofs are now provided, whereas before only the original formula and the result were given after each set of transformations, without any further details, which arguably did not even constitute a proof.

Allowing Boolean and arithmetic simplifications leads to some improvements, especially for the quantifier-free benchmarks. We expect that generating proofs for the global transformations would lead to substantial improvements on these problems.

7 Related Work

Most automatic provers that support the TPTP syntax for problems generate proofs in TSTP (Thousands of Solutions for Theorem Provers) format [41]. Like a veriT proof, a TSTP proof consists of a list of inferences. TSTP does not mandate any inference system; the meaning of the rules and the granularity of inferences vary across systems. For example, the E prover [38] combines clausification, skolemization, and variable renaming into a single inference, whereas Vampire [26] appears to cleanly separate preprocessing transformations. SPASS's [42] custom proof format does not record preprocessing steps; reverse engineering is necessary to make sense of its output, and optimizations ought to be disabled [8, Sect. 7.3].

Most SMT solvers can parse the SMT-LIB [5] format, but each solver has its own output syntax. Z3's proofs can be quite detailed [31], but rewriting steps often combine many rewrites rules. CVC4's format is an instance of LF [23] with Side Conditions (LFSC) [39]; despite recent progress [22, 25], neither skolemization nor quantifier instantiation are currently recorded in the proofs. Proof production in Fx7 [30] is based on an inference system whose formula processing fragment is subsumed by ours; for example, skolemization is more ad hoc, and there is no explicit support for rewriting.

Proof assistants for dependent type theory, including Agda, Coq, Lean, and Matita, provide very precise proof terms that can be checked by relatively simple checkers, meeting De Bruijn's criterion [4]. Exploiting the Curry–Howard correspondence, a proof term is a λ-term whose type is the proposition it proves; for example, the term $\lambda x.\ x$, of type $A \to A$, is a proof that A implies A. Proof terms have also been implemented in Isabelle [6], but they slow down the system considerably and are normally disabled. Frameworks such as LF, LFSC, and Dedukti [13] provide a way to specify inference systems and proof checkers based on proof terms. Our encoding into λ-terms is vaguely reminiscent of LF. The encoded rules also bear a superficial resemblance to deep inference [21].

Isabelle and the proof assistants from the HOL family (HOL4, HOL Light, HOL Zero, and ProofPower–HOL) are based on the LCF architecture [19]. Theorems are represented by an abstract data type. A small set of primitive inferences derives new theorems from existing ones. This architecture is also the inspiration behind automatic systems such as Psyche [20]. In Cambridge LCF, Paulson introduced an idiom, *conversions*, for expressing rewriting strategies [36]. A conversion is an ML function from terms t to theorems of the form $t \simeq u$. Basic conversions perform β-reduction and other simple rewriting. Higher-order functions combine conversions. Paulson's conversion library culminates with a function that replaces Edinburgh LCF's monolithic simplifier. Conversions are still in use today in Isabelle and the HOL systems. They allow a style of programming that focuses on the terms to rewrite—the proofs arise as a side effect. Our framework is related, but we trade programmability for efficiency on very large problems. Remarkably, both Paulson's conversions and our framework emerged as replacements for earlier monolithic systems.

Over the years, there have been many attempts at integrating automatic provers into proof assistants. To reach the highest standards of trustworthiness, some of these bridges translate the proofs found by the automatic provers so that they can be checked by the proof assistant. The TRAMP subsystem of ΩMEGA is one of the finest examples [28]. For integrating superposition provers with Coq, De Nivelle studied how to build efficient proof terms for clausification and skolemization [33]. For SMT, the main integrations with proof reconstruction are CVC Lite in HOL Light [27], haRVey (veriT's predecessor) in Isabelle/HOL [17], Z3 in HOL4 and Isabelle/HOL [9,10], veriT in Coq [1], and CVC4 in Coq [16]. Some of these simulate the proofs in the proof assistant using dedicated tactics, in the style of our simple checker for Isabelle (Sect. 6.1). Others employ reflection, a technique whereby the proof checker is specified in the proof assistant's formalism and proved correct; in systems based on dependent type theory, this can help keep proof terms to a manageable size. A third approach is to translate the SMT output into a proof text that can be inserted in the user's formalization; Isabelle/HOL supports veriT and Z3 in this way [8].

Proof assistants are not the only programs used to check machine-generated proofs. Otterfier invokes the Otter prover to check TSTP proofs from various sources [43]. GAPT imports proofs generated by resolution provers with clausifiers to a sequent calculus and uses other provers and solvers to transform the proofs [15, 24]. Dedukti's $\lambda\Pi$-calculus modulo [13] has been used to encode resolution and superposition proofs [12], among others. λProlog provides a general proof-checking framework that allows nondeterminism, enabling flexible combinations of proof search and proof checking [29].

8 Conclusion

We presented a framework to represent and generate proofs of formula processing and its implementation in veriT and Isabelle/HOL. The framework centralizes the delicate issue of manipulating bound variables and substitutions soundly and efficiently, and it is flexible enough to accommodate many interesting transformations. Although it was implemented in an SMT solver, there appears to be no intrinsic limitation that would prevent its use in other kinds of first-order, or even higher-order, automatic provers. The framework covers many preprocessing techniques and can be part of a larger toolbox.

Detailed proofs have been a defining feature of veriT for many years now. It now produces more detailed justifications than ever, but there are still some global transformations for which the proofs are nonexistent or leave much to be desired. In particular, supporting rewriting based on global assumptions would be essential for proof-producing inprocessing, and symmetry breaking would be interesting in its own right.

Acknowledgment We thank Simon Cruanes for discussing many aspects of the framework with us as it was emerging, and we thank Robert Lewis, Stephan Merz, Lawrence Paulson, Anders Schlichtkrull, Mark Summerfield, Sophie Tourret, and the CADE-26 anonymous reviewers for suggesting many textual improvements. This research has been partially supported by the Agence nationale de la recherche / Deutsche Forschungsgemeinschaft project SMArT (ANR-13-IS02-0001, STU 483/2-1) and by the European Union project SC^2 (grant agreement No. 712689). The work has also received funding from the European Research Council under the European Union's Horizon 2020 research and innovation program (grant agreement No. 713999, Matryoshka).

Experiments presented in this report were carried out using the Grid'5000 testbed (https://www.grid5000.fr/), supported by a scientific interest group hosted by Inria and including CNRS, RENATER, and several universities as well as other organizations.

References

[1] Armand, M., Faure, G., Grégoire, B., Keller, C., Théry, L., Werner, B.: A modular integration of SAT/SMT solvers to Coq through proof witnesses. In: Jouannaud, J.P., Shao, Z. (eds.) CPP 2011. LNCS, vol. 7086, pp. 135–150. Springer (2011)

[2] Barbosa, H., Blanchette, J.C., Fontaine, P.: Scalable fine-grained proofs for formula processing. In: de Moura, L. (ed.) CADE-26. LNCS, Springer (2017)

[3] Barbosa, H., Fontaine, P., Reynolds, A.: Congruence closure with free variables. In: Legay, A., Margaria, T. (eds.) TACAS 2017. LNCS, vol. 10206, pp. 214–230 (2017)

[4] Barendregt, H., Wiedijk, F.: The challenge of computer mathematics. Philosophical Transactions of the Royal Society of London—Series A, Mathematical and Physical Sciences 363(1835), 2351–2375 (2005)

[5] Barrett, C., Fontaine, P., Tinelli, C.: The SMT-LIB standard: Version 2.5. Tech. rep., University of Iowa (2015), http://smt-lib.org/

[6] Berghofer, S., Nipkow, T.: Proof terms for simply typed higher order logic. In: Aagaard, M., Harrison, J. (eds.) TPHOLs 2000. LNCS, vol. 1869, pp. 38–52. Springer (2000)

[7] Besson, F., Fontaine, P., Théry, L.: A flexible proof format for SMT: A proposal. In: Fontaine, P., Stump, A. (eds.) PxTP 2011. pp. 15–26 (2011)

[8] Blanchette, J.C., Böhme, S., Fleury, M., Smolka, S.J., Steckermeier, A.: Semi-intelligible Isar proofs from machine-generated proofs. J. Autom. Reasoning 56(2), 155–200 (2016)

[9] Böhme, S., Fox, A.C.J., Sewell, T., Weber, T.: Reconstruction of Z3's bit-vector proofs in HOL4 and Isabelle/HOL. In: Jouannaud, J., Shao, Z. (eds.) CPP 2011. LNCS, vol. 7086, pp. 183–198. Springer (2011)

[10] Böhme, S., Weber, T.: Fast LCF-style proof reconstruction for Z3. In: Kaufmann, M., Paulson, L. (eds.) ITP 2010. LNCS, vol. 6172, pp. 179–194. Springer (2010)

[11] Bouton, T., de Oliveira, D.C.B., Déharbe, D., Fontaine, P.: veriT: An open, trustable and efficient SMT-solver. In: Schmidt, R.A. (ed.) CADE-22. LNCS, vol. 5663, pp. 151–156. Springer (2009)

[12] Burel, G.: A shallow embedding of resolution and superposition proofs into the $\lambda\Pi$-calculus modulo. In: Blanchette, J.C., Urban, J. (eds.) PxTP 2013. EPiC Series in Computing, vol. 14, pp. 43–57. EasyChair (2013)

[13] Cousineau, D., Dowek, G.: Embedding pure type systems in the lambda-Pi-calculus modulo. In: Rocca, S.R.D. (ed.) TLCA 2007. LNCS, vol. 4583, pp. 102–117. Springer (2007)

[14] Déharbe, D., Fontaine, P., Merz, S., Paleo, B.W.: Exploiting symmetry in SMT problems. In: Bjørner, N., Sofronie-Stokkermans, V. (eds.) Proc. Conference on Automated Deduction (CADE). LNCS, vol. 6803, pp. 222–236. Springer (2011)

[15] Ebner, G., Hetzl, S., Reis, G., Riener, M., Wolfsteiner, S., Zivota, S.: System description: GAPT 2.0. In: Olivetti, N., Tiwari, A. (eds.) IJCAR 2016. LNCS, vol. 9706, pp. 293–301. Springer (2016)

[16] Ekici, B., Katz, G., Keller, C., Mebsout, A., Reynolds, A.J., Tinelli, C.: Extending SMTCoq, a certified checker for SMT (extended abstract). In: Blanchette, J.C., Kaliszyk, C. (eds.) HaTT 2016. EPTCS, vol. 210, pp. 21–29 (2016)

[17] Fontaine, P., Marion, J.Y., Merz, S., Nieto, L.P., Tiu, A.: Expressiveness + automation + soundness: Towards combining SMT solvers and interactive proof assistants. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006. LNCS, vol. 3920, pp. 167–181. Springer (2006)

[18] Gordon, M.J.C., Melham, T.F. (eds.): Introduction to HOL: A Theorem Proving Environment for Higher Order Logic. Cambridge University Press (1993)

[19] Gordon, M.J.C., Milner, R., Wadsworth, C.P.: Edinburgh LCF: A Mechanised Logic of Computation, LNCS, vol. 78. Springer (1979)

[20] Graham-Lengrand, S.: Psyche: A proof-search engine based on sequent calculus with an LCF-style architecture. In: Galmiche, D., Larchey-Wendling, D. (eds.) TABLEAUX 2013. LNCS, vol. 8123, pp. 149–156. Springer (2013)

[21] Guglielmi, A.: A system of interaction and structure. ACM Trans. Comput. Log. 8(1), 1 (2007)

[22] Hadarean, L., Barrett, C.W., Reynolds, A., Tinelli, C., Deters, M.: Fine grained SMT proofs for the theory of fixed-width bit-vectors. In: Davis, M., Fehnker, A., McIver, A., Voronkov, A. (eds.) LPAR-20. LNCS, vol. 9450, pp. 340–355. Springer (2015)

[23] Harper, R., Honsell, F., Plotkin, G.D.: A framework for defining logics. In: LICS '87. pp. 194–204. IEEE Computer Society (1987)

[24] Hetzl, S., Libal, T., Riener, M., Rukhaia, M.: Understanding resolution proofs through Herbrand's theorem. In: Galmiche, D., Larchey-Wendling, D. (eds.) Automated Reasoning with Analytic Tableaux and Related Methods (TABLEAUX). LNCS, vol. 8123, pp. 157–171. Springer (2013)

[25] Katz, G., Barrett, C.W., Tinelli, C., Reynolds, A., Hadarean, L.: Lazy proofs for dpll(t)-based SMT solvers. In: Piskac, R., Talupur, M. (eds.) FMCAD 2016. pp. 93–100. IEEE (2016)

[26] Kovács, L., Voronkov, A.: First-order theorem proving and Vampire. In: Sharygina, N., Veith, H. (eds.) CAV 2013, LNCS, vol. 8044, pp. 1–35. Springer (2013)

[27] McLaughlin, S., Barrett, C., Ge, Y.: Cooperating theorem provers: A case study combining HOL-Light and CVC Lite. Electr. Notes Theor. Comput. Sci. 144(2), 43–51 (2006)

[28] Meier, A.: TRAMP: Transformation of machine-found proofs into natural deduction proofs at the assertion level (system description). In: McAllester, D. (ed.) CADE-17. LNCS, vol. 1831, pp. 460–464. Springer (2000)

[29] Miller, D.: Proof checking and logic programming. In: Falaschi, M. (ed.) LOPSTR 2015. LNCS, vol. 9527, pp. 3–17. Springer (2015)

[30] Moskal, M.: Rocket-fast proof checking for SMT solvers. In: Ramakrishnan, C.R., Rehof, J. (eds.) Tools and Algorithms for Construction and Analysis of Systems (TACAS). LNCS, vol. 4963, pp. 486–500. Springer (2008)

[31] de Moura, L.M., Bjørner, N.: Proofs and refutations, and Z3. In: Rudnicki, P., Sutcliffe, G., Konev, B., Schmidt, R.A., Schulz, S. (eds.) LPAR 2008 Workshops. CEUR Workshop Proceedings, vol. 418. CEUR-WS.org (2008)

[32] Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL: A Proof Assistant for Higher-Order Logic, LNCS, vol. 2283. Springer (2002)

[33] de Nivelle, H.: Extraction of proofs from the clausal normal form transformation. In: Bradfield, J.C. (ed.) CSL 2002. LNCS, vol. 2471, pp. 584–598. Springer (2002)

[34] de Nivelle, H.: Translation of resolution proofs into short first-order proofs without choice axioms. Inf. Comput. 199(1-2), 24–54 (2005)

[35] Nonnengart, A., Weidenbach, C.: Computing small clause normal forms. In: Robinson, A., Voronkov, A. (eds.) Handbook of Automated Reasoning, vol. I, pp. 335–367. Elsevier and MIT Press (2001)

[36] Paulson, L.C.: A higher-order implementation of rewriting. Sci. Comput. Program. 3(2), 119–149 (1983)

[37] Paulson, L.C., Susanto, K.W.: Source-level proof reconstruction for interactive theorem proving. In: Schneider, K., Brandt, J. (eds.) TPHOLs 2007. LNCS, vol. 4732, pp. 232–245. Springer (2007)

[38] Schulz, S.: System description: E 1.8. In: McMillan, K., Middeldorp, A., Voronkov, A. (eds.) LPAR-19. LNCS, vol. 8312, pp. 735–743. Springer (2013)

[39] Stump, A.: Proof checking technology for satisfiability modulo theories. Electr. Notes Theor. Comput. Sci. 228, 121–133 (2009)

[40] Sutcliffe, G., Schulz, S., Claessen, K., Baumgartner, P.: The TPTP typed first-order form with arithmetic. In: Bjørner, N., Voronkov, A. (eds.) LPAR-18. LNCS, vol. 7180, pp. 406–419. Springer (2012)

[41] Sutcliffe, G., Zimmer, J., Schulz, S.: TSTP data-exchange formats for automated theorem proving tools. In: Zhang, W., Sorge, V. (eds.) Distributed Constraint Problem Solving and Reasoning in Multi-Agent Systems. Frontiers in Artificial Intelligence and Applications, vol. 112, pp. 201–215. IOS Press (2004)

[42] Weidenbach, C., Dimova, D., Fietzke, A., Kumar, R., Suda, M., Wischnewski, P.: SPASS version 3.5. In: Schmidt, R.A. (ed.) CADE-22, LNCS, vol. 5663, pp. 140–145. Springer (2009)

[43] Zimmer, J., Meier, A., Sutcliffe, G., Zhan, Y.: Integrated proof transformation services. In: Benzmüller, C., Windsteiger, W. (eds.) IJCAR WS 7 (2004)