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. 2). 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. 3). 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. 4). The rules are sound, and the treatment of binders is correct even in the presence of name clashes. Moreover, with suitable data structures, 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. Detailed proofs of the metatheory are included in a technical report [2], together with more explanations and examples.

We implemented the approach in veriT (Sect. 5), 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 [5], which are reconstructed in the proof assistants Coq [1] and Isabelle/HOL [6] and in the GAPT system [10]. 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.

Conventions Our setting is a many-sorted classical first-order logic as defined by the SMT-LIB standard [4]. A signature $\Sigma = (\mathcal{S}, \mathcal{F})$ consists of a set $\mathcal{S}$ of sorts and a set $\mathcal{F}$ of function symbols. 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 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.

We use the symbol $=$ for syntactic equality on terms. We reserve the names $\mathsf{a}, \mathsf{f}, \mathsf{p}, \mathsf{q}$ for function symbols; x, y 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 set of its free variables is written $FV(t)$. The notation $t[\bar{x}_n]$ stands for a term that may depend on $\bar{x}_n$; $t[\bar{s}_n]$ is the corresponding term where the terms $\bar{s}_n$ are 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 substitution $\rho[\bar{x}_n \mapsto \bar{s}_n]$ 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).

2 Inference System

The inference rules used by our framework depend on a notion of *context* defined by the grammar $\Gamma ::= \varnothing \mid \Gamma, x \mid \Gamma, \bar{x}_n \mapsto \bar{s}_n$. Each context entry either *fixes* a variable x or defines a *substitution* $\{\bar{x}_n \mapsto \bar{s}_n\}$. 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 $\mathit{subst}(\Gamma)$ defined by $\mathit{subst}(\varnothing) = \{\}$, $\mathit{subst}(\Gamma, x) = \mathit{subst}(\Gamma)[x \mapsto x]$, and $\mathit{subst}(\Gamma, \bar{x}_n \mapsto \bar{t}_n) = \mathit{subst}(\Gamma) \circ \{\bar{x}_n \mapsto \bar{t}_n\}$. In the second equation, the $[x \mapsto x]$ update shadows any replacement of x induced by Γ. 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 paper are presented below.

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

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

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

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

- $\textsc{Taut}_{\mathscr{T}}$ relies on an oracle $\models_{\mathscr{T}}$ to derive arbitrary lemmas in a theory $\mathscr{T}$. 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.

- 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.
- 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. The side condition prevents an unwarranted variable capture. In the antecedent, the renaming is expressed by a substitution in the context.
- SKO$_\exists$ and 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.
- LET exploits (let) to expand a 'let' expression. 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. By systematically renaming variables in BIND, we can satisfy most side conditions trivially.

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 [7, 19, 21], 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:

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

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.

3 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.

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 of $\Gamma \rhd t \simeq u$ obtained by applying rule R to $\bar{\mathscr{D}}_n$. The plugin functions are responsible for invoking *apply* as appropriate.

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 $\mathit{build\_let}(\Delta, \Delta', \bar{x}_n, \bar{r}_n, t', \mathit{process}(\Delta', t'))$

'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. 2. The plugin functions are as follows:

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

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

function $\mathit{ctx\_app}(\Gamma, \mathsf{f}, \bar{t}_n, i)$
  return Γ

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

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

function *build_quant*(Γ, Γ', Q, x, φ, ψ)
  $y \leftarrow \Gamma'(x)$
  apply(BIND, 1, Γ, $Qx.\varphi$, $Qy.\psi$)
  return $Qy.\psi$

function *build_var*(Γ, x)
  apply(REFL, 0, Γ, 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. 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 2. Following up on Example 1, assume $\varphi = \text{let } x \simeq \mathsf{a} \text{ in } \mathsf{p}(x,x)$. Given the above plugin functions, *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 $\triangleright\ \mathsf{a} \simeq \mathsf{a}$ 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 $\triangleright\ \varphi \simeq \mathsf{p}(\mathsf{a},\mathsf{a})$ 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.

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 as defined in Sect. 2 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 $(Q, p) \in \{(\exists, +), (\forall, -)\}$ **then**
    $\Gamma' \leftarrow \Gamma, x \mapsto$ *sko_term*(Γ, Q, x, φ)
  else
    $\Gamma' \leftarrow \Gamma, x$
  return (Γ', p)

The polarity 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 [20] creates an application of a fresh function symbol to all variables fixed in the context. To comply

with the inference system, the application of $\text{SKO}_\exists$ or $\text{SKO}_\forall$ in *build_quant* instructs the proof module to systematically replace the Skolem term with the corresponding ε term when outputting the proof.

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; 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, $Qx.\varphi$, $Qx.\psi$)
  return $Qx.\psi$

function *build_app*(Γ, $\bar{\Gamma}'_n$, f, $\bar{t}_n$, $\bar{u}_n$)
  apply(CONG, n, Γ, $\mathsf{f}(\bar{t}_n)$, $\mathsf{f}(\bar{u}_n)$)
  if $\mathsf{f}(\bar{u}_n)$ has form $u+0$ or $0+u$ **then**
    apply(TAUT_+, 0, Γ, $\mathsf{f}(\bar{u}_n)$, u)
    apply(TRANS, 2, Γ, $\mathsf{f}(\bar{t}_n)$, u)
    return u
  else
    return $\mathsf{f}(\bar{u}_n)$

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: $\mathsf{f}(\bar{t}_n) \simeq \mathsf{f}(\bar{u}_n) \simeq u$.

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 allows us to perform arithmetic simplification in the substituted terms of a 'let' expression in a single pass.

The combination of 'let' expansion and skolemization is less straightforward. Consider the formula $\varphi = \text{let } y \simeq \exists x.\, \mathsf{p}(x) \text{ in } y \to y$. 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. There is also a way to perform all the transformations in a single instance of the framework, described in our technical report [2].

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 $\text{TAUT}_{\mathsf{Bool}}$ 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.

Some transformations, such as symmetry breaking [9] 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.

4 Theoretical Properties

The first two metatheoretical results below concern the soundness of the inference rules and the correctness of the recursion algorithm that generates proofs in that system. The other results have to do with the cost of proof generation and checking.

Theorem 1 (Soundness of Inferences). *If the judgment* $\Gamma \rhd t \simeq u$ *is derivable using the inference system with the theories* $\mathscr{T}_1, \ldots, \mathscr{T}_n$, *then* $\models_{\mathscr{T}_1 \cup \cdots \cup \mathscr{T}_n \cup \simeq \cup \varepsilon \cup \mathrm{let}} \Gamma(t) \simeq u$.

Theorem 2 (Total Correctness of Recursion). *For the instances presented in Sect. 3, the contextual recursion algorithm always produces correct proofs.*

Observation 3 (Complexity of Recursion). *For the instances presented in Sect. 3, the 'process' function is called at most once on every subterm of the input.*

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, skolemization is at best quadratic in the worst case.

Observation 4 (Overhead of Proof Generation). *For the instances presented in Sect. 3, the number of 'apply' calls is proportional to the number of subterms in the input.*

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.

Observation 5 (Cost of Proof Checking). *Checking an inference step can be performed in constant time if checking the side condition takes 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. Fortunately, most of the side conditions can be checked efficiently. For example, simplification proofs can be checked in linear time because $subst(\Gamma)$ is always the identity. 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, exploiting sophisticated invariants.

5 Implementation

The ideas presented in this paper have been implemented in two tools. We implemented the contextual recursion algorithm and the transformations described in Sect. 3 in the SMT solver veriT [8], 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. 2 using Isabelle/HOL [18], to convince ourselves that veriT's output can easily be reconstructed.

Isabelle The Isabelle/HOL proof assistant is based on classical higher-order logic (HOL), 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. 4 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 $\textsc{Taut}_{\mathscr{T}}$ additionally carries a theorem that represents the oracle $\models_{\mathscr{T}}$, 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 $\mathsf{case}_\times : (\alpha \to \beta \to \gamma) \to \alpha \times \beta \to \gamma$; 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 $(t_1, (\cdots (t_{n-1}, t_n) \cdots))$.

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 [11]. 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 $t \simeq u$.

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

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

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

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.

We were able to reuse its existing proof module and proof format [5]. A proof is a list of inferences, each of which consists of an identifier, 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. 2. 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.

In contrast with the abstract proof module described in Sect. 3, 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.

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.

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. It currently produces coarse-grained proofs.

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 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',

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

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 relies on two main sets of benchmarks from SMT-LIB [4] without bit vectors and nonlinear arithmetic (currently not supported by veriT): the 20 916 benchmarks in the quantifier-free (QF) categories, and the 30 250 benchmarks labeled as unsatisfiable in the non-QF categories. 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 table below shows the number of problems solved in total by each configuration.

		Old code	New code
BASIC	without proofs	42 235	42 258
	with proofs	42 104	42 118
EXTENDED	without proofs	42 324	42 389
	with proofs	N/A	42 271
COMPLETE	without proofs	42 585	42 613
	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 compared with the old code and coarse-grained proofs. Moreover, allowing Boolean and arithmetic simplifications leads to some improvements. We expect that generating proofs for the global transformations would lead to substantial improvements on quantifier-free problems.

6 Related Work

Most automatic provers that support the TPTP syntax for problems generate proofs in TSTP format [24]. 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 [22] combines clausification, skolemization, and variable renaming into a single inference, whereas Vampire [15] appears to cleanly separate preprocessing transformations. SPASS's [25] custom proof format does not record preprocessing steps; reverse engineering is necessary to make sense of its output, and optimizations ought to be disabled [6, Sect. 7.3].

Most SMT solvers can parse the SMT-LIB [4] format, but each solver has its own output syntax. Z3's proofs can be quite detailed [17], but rewriting steps often combine many rewrites rules. CVC4's format is an instance of LF [13] with Side Conditions [23]; despite recent progress [12, 14], neither skolemization nor quantifier instantiation are currently recorded in the proofs. Proof production in Fx7 [16] 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.

7 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 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 paper 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.: Tech. report associated with this paper (2017), `http://matryoshka.gforge.inria.fr/pubs/processing_report.pdf`

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

[5] 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)

[6] 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)

[7] 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)

[8] 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)

[9] 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)
[10] 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)
[11] Gordon, M.J.C., Milner, R., Wadsworth, C.P.: Edinburgh LCF: A Mechanised Logic of Computation, LNCS, vol. 78. Springer (1979)
[12] 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)
[13] Harper, R., Honsell, F., Plotkin, G.D.: A framework for defining logics. In: LICS '87. pp. 194–204. IEEE Computer Society (1987)
[14] 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)
[15] 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)
[16] 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)
[17] 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)
[18] Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL: A Proof Assistant for Higher-Order Logic, LNCS, vol. 2283. Springer (2002)
[19] de Nivelle, H.: Translation of resolution proofs into short first-order proofs without choice axioms. Inf. Comput. 199(1-2), 24–54 (2005)
[20] 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)
[21] 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)
[22] 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)
[23] Stump, A.: Proof checking technology for satisfiability modulo theories. Electr. Notes Theor. Comput. Sci. 228, 121–133 (2009)
[24] 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)
[25] 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)