An Efficient Proof-Producing Framework for Formula Processing

Haniel Barbosa\textsuperscript{1,2}, Jasmin Christian Blanchette\textsuperscript{3,1,4}, and Pascal Fontaine\textsuperscript{1}

\textsuperscript{1} Inria, Loria, and Université de Lorraine, Nancy, France
\textsuperscript{2} Universidade Federal do Rio Grande do Norte, Natal, Brazil
\textsuperscript{3} Vrije Universiteit Amsterdam, Netherlands
\textsuperscript{4} Max-Planck-Institut für Informatik, Saarbrücken, Germany

Abstract. We present a framework for processing formulas in automatic theorem provers, with proof generation. 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, allowing us to dramatically simplify the code base while making the solver faster when proofs are enabled.

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, optional processing features are even disabled to avoid gaps in proofs, at a high cost in proof search performance.

We propose a framework to address these issues. Proofs are expressed using an extensible set of inference rules (Sect.\textsuperscript{5}). The succedent of a rule is an equality between the original term and the translated term. (We 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.
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 large inputs, this can have a substantial impact on 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. Detailed proofs of the metatheory are included in a technical report [2].

We implemented the approach in veriT (Sect. 6), an SMT solver that is competitive on problems combining equality, linear arithmetic, and quantifiers [3]. The solver is known for its detailed proofs [6], which are reconstructed in the proof assistants Coq [1] and Isabelle/HOL [7]. By moving to 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. With proof output enabled, veriT is faster and solves slightly more benchmark problems than before. The more precise justifications can be communicated to other systems. As a proof of concept, we implemented a simple checker in Isabelle/HOL.

### 2 Conventions

Our setting is a many-sorted classical first-order logic as defined by the SMT-LIB standard [5] or TPTP TFF [33]. Our results are also applicable to richer formalisms such as higher-order logic (simple type theory) [17]. A signature \( \Sigma = (S, F) \) consists of a set \( S \) of sorts and a set \( F \) of function symbols over these sorts. Nullary function symbols are called constants. We assume that the signature contains a \( \text{Bool} \) sort and constants \( \text{true}, \text{false} : \text{Bool} \), a family \(( \approx : \sigma \times \sigma \to \text{Bool})_{\sigma \in S}\) of function symbols interpreted as equality, and the connectives \( \neg, \land, \lor, \text{and } \to \). Formulas are terms of type \( \text{Bool} \), and equivalence is equality \((\simeq)\) on \( \text{Bool} \). Terms are built over function symbols from \( F \) and variables from a fixed family of infinite sets \((\forall \phi)_{\phi \in \mathcal{S}}\). In addition to \( \forall \) and \( \exists \), we rely on two more binders: Hilbert’s choice operator \( \varepsilon x. \phi \) and a ‘let’ construct, let \( \bar{x} \equiv \bar{s} \in \ldots \), which simultaneously assigns \( n \) variables that can be used in the body \( t \).

We use the symbol \( = \) for syntactic equality on terms and \( =_a \) for syntactic equality up to renaming of bound variables. We reserve the names \( a, c, f, g, p, q \) for function symbols; \( x, y, z \) for variables; \( r, s, t, u \) for terms (which may be formulas); \( \varphi, \psi \) for formulas; and \( \forall \) for quantifiers \((\forall \varphi \\text{and } \exists \). 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 $t[\bar{s}_n]$ stands for a term that may depend on $\bar{s}_n$; $t[\bar{s}_n]$ is the corresponding term where the terms $\bar{s}_n$ are substituted for $\bar{s}_n$; bound variables in $t$ are renamed to avoid capture. Following these conventions, Hilbert’s choice and ‘let’ are characterized by

$$\models \exists x. \varphi[x] \rightarrow \varphi[e.\varphi] \quad (e_1)$$

$$\models (\forall x. \varphi \simeq \psi) \rightarrow (e.\varphi) \simeq (e.\psi) \quad (e_2)$$

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

Substitutions $\rho$ 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 $\rho$. The application of a substitution $\rho$ 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., $\rho$ is applied first).

### 3 Inference System

The inference rules used by our framework depend on a notion of context defined by the grammar $\Gamma ::= \emptyset \mid \Gamma, x \mid \Gamma, \bar{x}_n \mapsto \bar{s}_n$. The empty context $\emptyset$ is also denoted by a blank. Each entry in a context 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 $\Gamma$ on the left. If a context introduces the same variable several times, the rightmost entry shadows the others.

Abstractly, a context $\Gamma$ fixes a set of variables and specifies a substitution $\text{subst}(\Gamma)$ defined by $\text{subst}(\emptyset) = \{\}$, $\text{subst}(\Gamma, x) = \text{subst}(\Gamma)[x \mapsto x]$, and $\text{subst}(\Gamma, \bar{x}_n \mapsto \bar{s}_n) = \text{subst}(\Gamma) \circ \{\bar{x}_n \mapsto \bar{s}_n\}$. In the second equation, the $[x \mapsto x]$ update is necessary to shadow any replacement of $x$ induced by $\Gamma$. We write $\Gamma(t)$ to abbreviate the capture-avoiding substitution $\text{subst}(\Gamma)(t)$.

Transformations of terms (and formulas) are justified by judgments of the form $\Gamma \triangleright t \simeq u$, where $\Gamma$ 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 $\Gamma$. Semantically, the judgment expresses the equality of the terms $\Gamma(t)$ and $u$ for all variables fixed by $\Gamma$. 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.

- **Tautology** ($\Gamma \triangleright t \simeq u$ if $\models \varphi[t] \Rightarrow u$)
- **Transitivity** ($\Gamma \triangleright s \simeq t \Gamma \triangleright t \simeq u \Gamma \triangleright s \simeq u$)
- **Congruence** ($\Gamma \triangleright f(t_i) \simeq f(u_i)$)
- **Bind** ($\Gamma, y, x \mapsto y \triangleright \varphi \simeq \psi \Gamma \triangleright (Qx.\varphi) \simeq (Qy.\psi)$)
- **Skolemization** ($\Gamma, x \mapsto (\exists x.\varphi) \triangleright \varphi \simeq \psi \Gamma \triangleright (\forall x.\varphi) \simeq \psi$)
- **Let** ($\Gamma \triangleright (\text{let } \bar{x}_n \simeq \bar{r}_n \text{ in } t) \simeq u$ if $\Gamma(s_i) = s_i$ for all $i \in [n]$)
These rules deserve some explanation:

- **Taut** relies on an oracle \( \models \) 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 \( \Gamma(t) =_a u \).

- **Trans** needs the side condition because the term \( t \) appears both on the left-hand side of \( \simeq \) (where it is subject to \( \Gamma \)'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. If \( x = y \), the context is \( \Gamma, x \mapsto x \), which has the same meaning as \( \Gamma, x \).

- **SkO \(_3\) and SkO \(_\forall\)** exploit \( \varepsilon \) to replace a quantified variable with a suitable witness, simulating skolemization. We can think of the \( \varepsilon \) 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 \( \text{let} \) 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 rechecked 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’s choice to represent fresh symbols if necessary.

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

\[
\begin{align*}
\Gamma \vdash a \simeq a & \quad \text{Cong} \\
\Gamma, x \mapsto a \vdash x \simeq a & \quad \text{Refl} \\
\Gamma, x \mapsto a \vdash x \simeq a & \quad \text{Refl} \\
\Gamma, x \mapsto a \vdash p(x, x) \simeq p(a, a) & \quad \text{Cong} \\
\Gamma, x \mapsto a \vdash \text{let } x \simeq a \text{ in } p(x, x) & \simeq p(a, a) & \text{Let}
\end{align*}
\]

**Example 2.** The following derivation tree, in which \( \varepsilon_x \) abbreviates \( \varepsilon x. \neg p(x) \), justifies the skolemization of the quantifier in the formula \( \neg \forall x. p(x) \):

\[
\begin{align*}
\Gamma, x \mapsto \varepsilon_x & \vdash x \simeq \varepsilon_x & \text{Refl} \\
\Gamma, x \mapsto \varepsilon_x & \vdash p(x) \simeq p(\varepsilon_x) & \text{Cong} \\
\Gamma, x \mapsto \varepsilon_x & \vdash (\forall x. p(x)) \simeq p(\varepsilon_x) & \text{SkO}_\forall \\
\Gamma, x \mapsto \varepsilon_x & \vdash (\neg \forall x. p(x)) \simeq \neg p(\varepsilon_x) & \text{Cong}
\end{align*}
\]

Skolemization can be applied regardless of polarity. Normally, we skolemize only positive existential quantifiers and negative universal quantifiers. However, skolemizing
other quantifiers is sound for proving. The trouble is that it is generally incomplete, if we introduce Skolem symbols and forget their definitions in terms of Hilbert’s 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 \( \text{TAUT}_T \) and the equality rules \( \text{TRANS} \) and \( \text{CONG} \):

\[
\begin{align*}
\Gamma &\vdash k \simeq k & (\text{TUT}_k) \\
\Gamma &\vdash 1 \times 0 \simeq 0 & (\text{CONG}) \\
\Gamma &\vdash k + 1 \times 0 \simeq k & (\text{TUT}_r) \\
\Gamma &\vdash k + 1 \times 0 \simeq (k + 1 \times 0 < k) & (\text{CONG}) \\
\end{align*}
\]

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.

The output is generated by a proof module, which maintains a stack of derivations trees. The procedure \( \text{apply}(R, n, \Gamma, t, u) \) pops \( n \) derivation trees \( \mathcal{D}_n \) from the stack and pushes the derivation tree of \( \Gamma \vdash t \simeq u \) obtained by applying rule \( R \) to \( \mathcal{D}_n \). The plugin functions are responsible for invoking \( \text{apply} \) 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 \( \Delta \) 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: \( \text{ctx}_\text{let}, \text{ctx}_\text{quant}, \) and \( \text{ctx}_\text{app} \) update the context when entering the body of a binder or when moving from a function symbol to one of its arguments; \( \text{build}_\text{let}, \text{build}_\text{quant}, \text{build}_\text{app}, \) and \( \text{build}_\text{var} \) return the processed binder, function application, or variable and produce the corresponding proof as a side effect.

function \( \text{process}(\Delta, t) \) is

match \( t \)

\[\begin{align*}
\text{case} \ x: & \quad \text{return} \ \text{build}_\text{var}(\Delta, x) \\
\text{case} \ f(\bar{t}_n): & \quad \Delta_i' \leftarrow (\text{ctx}_\text{app}(\Delta, f, \bar{t}_n, i))_{i=1}^n \\
& \quad \text{return} \ \text{build}_\text{app}(\Delta, \Delta_i', f, \bar{t}_n, (\text{process}(\Delta_i', t_i))_{i=1}^n)
\end{align*}\]
The plugin functions are as follows:

The first instance of the contextual recursion algorithm expands ‘let’ expressions and

\[ \Delta' \leftarrow ctx\_quant(\Delta, Q, x, \varphi) \]
\[ \text{return } build\_quant(\Delta, \Delta', Q, x, \varphi, \text{process}(\Delta', \varphi)) \]

\[ \text{case let } \bar{x}_n \simeq \bar{r}_n \text{ in } t' : \]
\[ \Delta' \leftarrow ctx\_let(\Delta, \bar{x}_n, \bar{r}_n, t') \]
\[ \text{return } build\_let(\Delta, \Delta', \bar{x}_n, \bar{r}_n, t', \text{process}(\Delta', t')) \]

### 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** \(ctx\_let(\Gamma, \bar{x}_n, \bar{r}_n, t)\) is

\[ \begin{align*}
\text{return } & \Gamma, \bar{x}_n \mapsto (\text{process}(\Gamma, r_i))_{i=1}^n \\
\end{align*} \]

**Function** \(build\_let(\Gamma, \Gamma', \bar{x}_n, \bar{r}_n, t, u)\) is

\[ \begin{align*}
\text{return } & u \\
\end{align*} \]

**Function** \(ctx\_quant(\Gamma, Q, x, \varphi)\) is

\[ \begin{align*}
y & \leftarrow \text{fresh variable} \\
\text{return } & \Gamma, y, x \mapsto y \\
\end{align*} \]

**Function** \(build\_quant(\Gamma, \Gamma', Q, x, \varphi, \psi)\) is

\[ \begin{align*}
y & \leftarrow \Gamma'(x) \\
\text{return } & Qy, \psi \\
\end{align*} \]

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 4.** Following up on Example 1 let \(\varphi = \text{let } x \simeq a \text{ in } p(x, x)\). Given the above plugin functions, \(\text{process}(\varnothing, \varphi)\) returns \(p(a, a)\). It is instructive to study the evolution of the stack during the execution of \(\text{process}\). First, in \(ctx\_let\), the \(a\) is processed recursively; the call to \(build\_app\) pushes a nullary \(\text{CONG}\) step with succedent \(\varphi \simeq a\) onto the stack. Then the term \(p(x, x)\) is processed. For each of the two occurrences of \(x\), \(build\_var\) pushes a \(\text{REFL}\) step onto the stack. Next, \(build\_app\) applies a \(\text{CONG}\) step to justify rewriting under \(p\): The two \(\text{REFL}\) steps are popped, and a binary \(\text{CONG}\) is pushed. Finally, \(build\_let\) performs a LET inference with succedent \(\varphi \simeq p(a, a)\) to complete the proof: The two \(\text{CONG}\) 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 $\Gamma$ 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**: Computes the context for a given term $\varphi$, assuming that ‘let’ expressions have been expanded. The context is a list of fixed variables.
- **function build_quant**: Applies a fresh function symbol to all variables fixed in the context.
- **function sko_term**: Skolemizes a term by replacing existential quantifiers with Skolem terms.

The polarity is updated by `ctx_app`, which is not shown. For example, `ctx_app((\exists, \neg, \varphi, 1)` returns $(\Gamma, +)$, because if $\neg\varphi$ occurs negatively in a larger formula, then $\varphi$ 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 [29] creates an application of a fresh function symbol to all variables fixed in the context. To comply with the inference system, the application of $\text{SKO}_0$ or $\text{SKO}_1$ in `build_quant` instructs the proof module to systematically replace the Skolem term with the corresponding $\varepsilon$ term when outputting the proof.

**Example 5.** Let $\varphi = \neg\forall x. \, p(x)$. The call `process((\forall, +), \varphi)` skolemizes $\varphi$ into $\neg p(c)$, 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 $x \mapsto c$ is added to the context. Applying $\text{SKO}_1$ instructs the proof module to replace $c$ with $\varepsilon x. \, \neg p(x)$. 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**: Computes the context for a given term $\varphi$, assuming that ‘let’ expressions have been expanded.
- **function build_app**: Builds a new application of a fresh function symbol to all variables fixed in the context.
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 \( f(\bar{u}_n) \) can be simplified further into a term \( u \), it performs a transitive chain of reasoning: \( f(\bar{u}_n) \simeq f(\bar{u}_n) \simeq u \).

Example 6. Let \( \varphi = k + 1 \times 0 < k \). Assuming that the framework has been instantiated with theory simplification for additive and multiplicative identity, invoking \( \text{process}(\emptyset, \varphi) \) returns the formula \( k < k \). 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. Assuming 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.

The combination of ‘let’ expansion and skolemization is less straightforward. Consider the formula \( \varphi = \exists x. p(x) \) in \( y \rightarrow y \). When processing the subformula \( \exists x. p(x) \), 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; the second pass skolemizes and simplifies terms.

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`:

```plaintext
function ctx_let(\( \Gamma, p \), \( \bar{x}_n, \bar{\bar{r}}_n, i \)) is
  for \( i = 1 \) to \( n \) do
    apply(REFL, 0, \( \Gamma, x_i, \Gamma(r_i) \))
  \( \Gamma' \leftarrow \Gamma, \bar{x}_n \rightarrow (\Gamma(r_i))_{i=1}^n \)
return \( \Gamma', p \)
```

In contrast with the corresponding function for ‘let’ expansion (Sect. 4.2), `ctx_let` does not process the terms \( \bar{\bar{r}}_n \), 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` case renames bound variables apart (Sect. 4.3). The `ctx_app` function is as for skolemization, whereas `build_app` is as for theory simplification (Sect. 4.4).

For the formula \( \varphi \) introduced above, \( \text{process}(\emptyset, +, \varphi) \) returns \( \exists x. p(x) \rightarrow p(c) \), where \( c \) is a fresh Skolem constant. The substituted term \( \exists x. p(x) \) is put unchanged into the substitution used to expand the ‘let’ expression. It is processed each time \( y \) arises in the body \( y \rightarrow y \). 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 TAUT\textsubscript{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 $\forall x. x \simeq a \rightarrow p(x)$ into $p(a)$—are similar to ‘let’ expansion and can be represented in much the same way in our framework.

A less convincing application of the framework is simplification based on associativity and commutativity of function symbols. The rewritings require traversing the terms to be simplified. Since process visits terms in postorder, the complexity of the simplification would be quadratic. By contrast, a custom processing that applies depth-first preorder traversal can perform the simplifications with a linear complexity.

Some transformations, such as symmetry breaking [13] 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.

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 encode the judgments in a well-understood theory of binders: the simply typed $\lambda$-calculus. A context and a term are encoded together as a single $\lambda$-term. We call these somewhat nonstandard $\lambda$-terms metaterms. They are defined by the grammar

$$M ::= t | \lambda x. M | (\lambda \bar{x}. M) \bar{t}.$$

A metaterm is either a term $t$ decorated with a box $[]$, a $\lambda$-abstraction, or the application of an $n$-tuple of terms to an uncurried $\lambda$-abstraction that simultaneously binds $n$ distinct variables.

**Theorem 1 (Soundness of Inferences).** If the judgment $\Gamma \vdash t \simeq u$ is derivable using the inference system with the theories $T_1, \ldots, T_n$, then $\models T_1 \cup \cdots \cup T_n \cup \simeq \cup \epsilon \cup \text{let} \Gamma (t) \simeq u$.

**Proof.** Our proof strategy [2] is as follows. We start by showing how to encode an arbitrary judgment as the equality of two metaterms, $M \simeq N$. Then we present a set of inference rules for deriving judgments $M \simeq N$. Each rule in the inference system presented in Sect. 3 has a corresponding encoded rule—for example:

$$\frac{(M[i] \simeq N[i])_{i=1}^n}{M[f(\bar{i})] \simeq N[f(\bar{u})]}\quad \text{CONG} \quad \frac{M[(\lambda x. \varphi)(\epsilon x. \varphi)] \simeq N}{M[\exists x. \varphi] \simeq N}\quad \text{SKO}$$

where $M[i]$ represents a metaterm with $[\square]$ as its box. Relying on basic properties of the $\lambda$-calculus, we show that the encoded inference system is sound. Finally, we show that each inference rule presented in Sect. 3 can be simulated by the corresponding encoded rule, allowing us to conclude that the system of Sect. 3 is sound. ⊓⊔

Next, we turn to the contextual recursion algorithm that generates derivations in that system. The first question we look into 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 met. First, we need to introduce some terminology. A term is shadowing-free if nested binders...
always bind variables with different names; for example, \( \forall x. (\forall y. p(x, y)) \land (\forall y. q(y)) \) is shadowing-free. The set of variables fixed by \( \Gamma \) is written \( \text{fix}(\Gamma) \), and the set of variables replaced by \( \Gamma \) is written \( \text{repl}(\Gamma) \). They are defined as follows:

\[
\begin{align*}
\text{fix}(\emptyset) &= \{\} \\
\text{fix}(\Gamma, x) &= \{x\} \cup \text{fix}(\Gamma) \\
\text{repl}(\emptyset) &= \{\} \\
\text{repl}(\Gamma, x) &= \text{repl}(\Gamma)
\end{align*}
\]

Trivial substitutions \( x \mapsto x \) are ignored, since they have no effect. The set of variables introduced by \( \Gamma \) is defined by \( \text{intr}(\Gamma) = \text{fix}(\Gamma) \cup \text{repl}(\Gamma) \). A context \( \Gamma \) is consistent if all the fixed variables are mutually distinct and the two sets of variables are disjoint (i.e., \( \text{fix}(\Gamma) \cap \text{repl}(\Gamma) = \{\} \)). A judgment \( \Gamma \vdash t \simeq u \) is canonical if \( \Gamma \) is consistent, \( \text{FV}(t) \subseteq \text{intr}(\Gamma) \), \( \text{FV}(u) \subseteq \text{fix}(\Gamma) \), and \( \text{BV}(u) \cap \text{intr}(\Gamma) = \{\} \). The canonical inference system is a variant of the system of Sect. 3 in which all judgments are canonical and rules \text{TRANS}, \text{BIND}, and \text{LET} have no side conditions.

**Theorem 2 (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 \( \Gamma \) part of the context is relevant. We will write process(\( \Gamma, t \)) even if the first argument actually has the form (\( \Gamma, p \)) for skolemization. The pre- and postconditions of a process(\( \Gamma, t \)) call that returns the term \( u \) are

\[
\begin{align*}
\text{PRE1 } & \Gamma \text{ is consistent;} \\
\text{PRE2 } & \text{FV}(t) \subseteq \text{intr}(\Gamma); \\
\text{PRE3 } & \text{BV}(t) \cap \text{fix}(\Gamma) = \{\};
\end{align*}
\]

\[
\begin{align*}
\text{POST1 } & u \text{ is shadowing-free;} \\
\text{POST2 } & \text{FV}(u) \subseteq \text{fix}(\Gamma); \\
\text{POST3 } & \text{BV}(u) \cap \text{intr}(\Gamma) = \{\}.
\end{align*}
\]

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

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 stated above. Correctness follows from the observation that any inference in the canonical inference system is also an inference in the original inference system: The constraints on canonical judgments ensure that the side conditions of \text{TRANS}, \text{BIND}, and \text{LET} are always satisfied.

**Observation 3 (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.

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.

Observation 4 (Overhead of Proof Generation). For the instances presented in Sects. 4.2 to 4.4, the number of calls to the ‘apply’ function 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; 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 5 (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. Fortunately, most of the side conditions can be checked efficiently. For example, simplification proofs can be checked in linear time because subst(Γ) 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.

6 Implementation

The ideas presented in this paper have been implemented in two tools. We programmed the contextual recursion algorithm and the transformations described in Sect. 4 in the SMT solver veriT [10]. In addition, we developed a prototypical proof checker for the inference system described in Sect. 3 using Isabelle/HOL [27], to convince ourselves that veriT’s output can easily be reconstructed.

6.1 Isabelle/HOL

Isabelle/HOL is a proof assistant 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. 5 to represent judgments in HOL. The proof checker is included in the development version of Isabelle.

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 TAUFX additionally carries a theorem that represents the oracle \(\models \varphi\), and rules TRANS and LET are labeled with the terms that occur only in the antecedent (\(t\) and \(\delta_n\)). Terms and metaterms are translated to HOL terms, and judgments \(M \equiv N\) are translated to HOL equalities \(t \simeq u\), where \(t\) and \(u\) are HOL terms. Uncurried \(\lambda\)-applications are encoded using a polymorphic combinator \(\text{case}_\text{prod} \colon (\alpha \to \beta \to \gamma) \to \alpha \times \beta \to \gamma\); in Isabelle/HOL, \(\lambda(x,y). \top\) is syntactic sugar for \(\text{case}_\text{prod} (\lambda x. Ay. t)\). 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 \(\{18\}\). We hard-coded a few dozen examples to test different cases, such as this one:

\[
N (\text{Cong}, [N (\text{Bind}, [N (\text{Cong}, [N (\text{Refl}, [[]])], N (\text{Bind}, [N (\text{Sko}_\text{All}, [N (\text{Refl}, [[]])]])])])])
\]

the reconstruction function returns the HOL theorem \(t \simeq u\).

6.2 veriT

We implemented the contextual recursion framework in the development version of the SMT solver veriT\(^2\), replacing large parts of the previous non-proof-producing, hard-to-maintain code for processing formulas. We reused the solver’s existing proof module and proof format.

Proofs A veriT proof \([6]\) is a list of inferences. Each inference consists of an identifier (e.g., \(c0\)), the name of the inference rule, the identifiers of the dependencies, and the derived clause. 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 \(\neg \forall x. p(x)\) from Example 2 is presented as

\[
(c0 \ (\text{Sko}_\text{All} : \text{conclusion} ((\forall x. p(x)) \simeq p(\text{ex} \cdot \neg p(x))))
\begin{align*}
: & \text{args} \ (x \mapsto (\text{ex} \cdot \neg p(x))) \\
: & \text{subproof} ((c1 \ (\text{Refl} : \text{conclusion} (x \simeq (\text{ex} \cdot \neg p(x)))))) \\
: & (c2 \ (\text{Cong} : \text{clauses} (c1) : \text{conclusion} (p(x) \simeq p(\text{ex} \cdot \neg p(x)))))) \\
: & (c3 \ (\text{Cong} : \text{clauses} (c0) : \text{conclusion} ((\neg \forall x. p(x)) \simeq \neg p(\text{ex} \cdot \neg p(x))))))
\end{align*}
\]

In contrast with the abstract proof module described in Sect. 4, veriT leaves most \(\text{REFL}\) steps implicit. The other inference rules are generalized to cope with missing \(\text{REFL}\) judgments on the stack. The proof module also associates terms in the inferences with some other terms when printing proofs. This is used 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.

\(^2\) https://members.loria.fr/HBarbosa/papers/2017-proofs/veriT-proofs.tar.gz
Transformations  The implementation of contextual recursion uses a single global context that is updated before and after processing a subterm. The context consists of a set of fixed variables, a substitution, and a polarity. Substitutions are composed eagerly to avoid spurious recomputations. If the context is empty, the result of processing a subterm is cached. For skolemization, a separate cache is used for each polarity.

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 eventually inserted into the larger resolution proof produced by veriT. This is achieved through an inference of the form

\[
\varphi \quad \mathcal{D} \quad \neg (\varphi \simeq \psi) \lor \neg \varphi \lor \psi \quad \text{TAUT}_\varphi
\]

\[
\psi
\]

\[
\text{RESOLVE}
\]

where \( \varphi \) is the original formula, \( \psi \) is the processed formula, and \( \mathcal{D} \) is a derivation of \( \varphi \simeq \psi \). The derivation \( \mathcal{D} \) may itself depend on instances of rule TAUT \( \varphi \), 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 more efficiently implemented as a dedicated procedure (Sect. 4.6).

Evaluation  We compared the performance of different configurations of veriT to evaluate the impact of the use of the new contextual recursion algorithm and the production of more detailed proofs. Our experimental data is available online\(^1\). Configurations are distinguished according to which sets of transformations they perform, whether they produce proofs, and whether they use contextual recursion.

SOME 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, exclusive ORs, and ‘if–then–else’ with quantifiers in subterms of unknown polarity. MORE adds Boolean and arithmetic simplifications to the transformations performed by SOME; for these, no proofs were generated before. ALL also performs global rewriting simplifications and symmetry breaking, for which no proofs are generated, in addition to the transformations performed by MORE. Given a base configuration, +P further indicates that proof production is enabled, and +C indicates that the new contextual recursion algorithm is used instead of the older code it subsumes. The proofs generated by the old code have a coarser granularity than those generated by the new algorithm.

\(^1\) https://members.loria.fr/HBarbosa/papers/2017-proofs
The evaluation was made on two main sets of benchmarks from SMT-LIB [5]: the 20916 benchmarks in the quantifier-free (QF) categories QF_ALIA, QF_AUFLIA, QF_IDL, QF_LIA, QF_LRA, QF_UF, QF_UFIDL, QF_UFLIA, and QF_UFLRA; and the 30250 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. The time limit was set to 30 s, a reasonable value for interactive use from a proof assistant. The table below indicates the number of benchmark problems solved in each configuration and the total number of problems:

<table>
<thead>
<tr>
<th></th>
<th>SOME</th>
<th>+P</th>
<th>+PC</th>
<th>MORE</th>
<th>+PC</th>
<th>ALL</th>
<th>+C</th>
<th>TOTAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>QF</td>
<td>13489</td>
<td>13344</td>
<td>13351</td>
<td>13527</td>
<td>13410</td>
<td>13804</td>
<td>13814</td>
<td>20916</td>
</tr>
<tr>
<td>NON-QF</td>
<td>28754</td>
<td>28596</td>
<td>28593</td>
<td>28798</td>
<td>28615</td>
<td>28791</td>
<td>28791</td>
<td>30250</td>
</tr>
</tbody>
</table>

Moving to the new contextual recursion code, and producing more detailed proofs, does not impact performance negatively. Moreover, allowing Boolean and arithmetic simplifications leads to some modest improvements, especially for the benchmarks with quantifiers. Adding proof generation to the global transformations would most likely have a substantial impact on quantifier-free 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 [34]. A TSTP proof consists of a list of inferences, like a veriT proof. 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 [31] combines clausification, Skolemization, and variable renaming into a single inference, whereas Vampire [22] appears to cleanly separate preprocessing transformations. SPASS’s [35] custom proof format does not record preprocessing steps; reverse engineering is necessary to make sense of its output, and optimizations ought to be disabled [7, Sect. 7.3]. SMT solvers typically support the SMT-LIB [5] format, but each solver has its own output syntax. Z3’s proofs can be quite detailed [26], but rewriting steps often combine many rewrites rules. CVC4’s format is an instance of LF with Side Conditions (LFSC) [32], an extension of LF [21]; currently, neither skolemization nor quantifier instantiation are recorded in the proofs.

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 λx.x, of type A → A, is a proof that A implies A. Frameworks such as LF, LFSC, and Dedukti [12] provide a way to specify inference systems and proof checkers based on proof terms. Our encoding into λ-terms is reminiscent of LF. The encoded rules also bear a superficial resemblance to deep inference [29].

Isabelle and the proof assistants from the HOL family (HOL4, HOL Light, HOL Zero, and ProofPower–HOL) are based on the LCF architecture [18]. 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 [19]. In Cambridge LCF, Paulson introduced an idiom, conversions,
for expressing rewriting strategies [30]. A conversion is an ML function from terms
t to theorems of the form \( t \sim u \). Basic conversions perform \( \beta \)-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. 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.

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 independently
rechecked by the proof assistant. The TRAMP subsystem of \( \Omega \)MEGA is one of the finest
examples [24]. For integrating superposition provers with Coq, De Nivelle studied how
to build efficient proof terms for clausification and skolemization [28]. For SMT, the
main integrations with proof reconstruction are CVC Lite in HOL Light [23], haRVey
(veriT’s predecessor) in Isabelle/HOL [16], Z3 in HOL4 and Isabelle/HOL [8,9], veriT
in Coq [1], and CVC4 in Coq [15]. 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 [7].

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 [36]. GAPT
imports proofs generated by resolution provers with clausifiers to a sequent calculus and
uses other provers and solvers to transform the proofs [14]. Dedukti’s \( \lambda \Pi \)-calculus mod-
ulo [12] has been used to encode resolution and superposition proofs [11], among others.
\( \lambda \)Prolog provides a general proof-checking framework that allows nondeterminism,
enabling combinations of proof search and proof checking [25].

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.

Our solver veriT 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, and which are beyond the scope of our framework (e.g., symmetry breaking,
rewriting based on global assumptions).
Acknowledgment We thank Simon Cruanes for discussing many aspects of the framework with us as it was emerging, and we thank Rob Lewis, Stephan Merz, Anders Schlichtkrull, and Mark Summerfield for suggesting 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 SC² (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
