A Pragmatic Approach to Equality Reasoning

Christoph Walther and Stephan Schweitzer
Fachgebiet Programmiermethodik
Technische Universität Darmstadt
www.informatik.tu-darmstadt.de/pm/

Abstract. We report about a first-order theorem prover which is implemented in the interactive verification tool \texttt{Verifun} to prove the base and step cases of an induction proof. The use in an interactive environment requires a terminating system providing a satisfying balance between theorem proving power and runtime performance as well as the supply of results being useful for carrying on with a proof attempt (by some user interaction, say) if a proof cannot be found. The latter requirement is particularly important because non-valid formulas are frequently encountered when proving theorems by induction. Our prover is based on symbolic evaluation, i.e. a method which combines symbolic execution of programs with techniques from classical theorem proving and term rewriting. We illustrate how to integrate the use of lemmas and induction hypotheses into symbolic evaluation and discuss the incorporation of equality reasoning in particular. We call our approach “pragmatic” because no interesting formal qualities (except soundness) can be assigned to it, but it successfully performs when running \texttt{Verifun} to prove statements about programs.

1 Introduction

We develop the \texttt{Verifun} system [1], [10], [12], an interactive system for the verification of statements about programs written in the functional programming language $\mathcal{L}$ [7]. This language consists of a definition principle for freely generated polymorphic data types, a definition principle for procedures operating on these data types based on recursion, case analyses, let-expressions and functional composition, and a definition principle for statements (called “lemmas” in the language) about the data types and the procedures. The data type $\mathbb{N}$ for natural numbers $\mathbb{N}$—built with the constructors 0 and the successor function $\mathbf{+}(...)$—and the data type $\mathbf{bool}$ consisting of the constructors $\mathbf{true}$ and $\mathbf{false}$ are predefined in $\mathcal{L}$. The definition principle for lemmas allows universal quantification only and uses case analyses (like in the procedure definitions) and the truth values to represent connectives. The general form of a lemma is given by $\text{lemma } \text{lem} <= \forall x_1: \tau_1, \ldots, x_n: \tau_n \text{ body}_\text{lem}$, where $\text{lem}$ is an identifier denoting the lemma’s name and $\text{body}_\text{lem}$, called the body of the lemma, is a boolean term built with the variables $x_i$ of type $\tau_i$ and the function symbols given by the data type and procedure definitions. Fig. 1 gives an example of a simple $\mathcal{L}$-program.

Technical Report VFR 06/02 — June 3, 2006
The proof of a lemma usually requires induction, the base and step formulas of which are represented by sequences of the form

\[ h_1, \ldots, h_n; \forall \ldots ; \forall ; \forall \ldots \vdash \text{goal} \]

(1)

where \( \{h_1, \ldots, h_n\} \) denotes the set of hypotheses defining the base or step case respectively. The set of induction hypotheses of a step case is given by \( \forall \ldots ; \forall \ldots ; \forall \ldots \), where the non-induction variables are universally quantified, and \( \text{goal} \), called the goalterm of the sequent, represents the induction conclusion. The induction hypotheses and the goalterm are boolean terms, and the hypotheses are literals.\(^1\) For example, when proving \( \forall x, y : \mathbb{N}. x + y = y + x \) by Peano-Induction upon \( x \), the sequent of the base case reads as \( x = 0 \); \( \vdash x + y = y + x \) and the step sequent is given as \( x \neq 0 \); \( \forall z : \mathbb{N}. \neg(x) + z = z + \neg(x) \vdash x + y = y + x \), where \( \neg(\ldots) \) stands for the predecessor function in \( \mathcal{L} \).

A sequent \( \text{seq} \)—given as in (1)—of a terminating \( \mathcal{L} \)-program \( P \) is \textit{valid} iff

\[ \forall \nu^* : \tau^*. h_1 \land \ldots \land h_n \land (\forall \nu_1^* : \tau_1^*. \ i_h_1) \land \ldots \land (\forall \nu_m^* : \tau_m^*. \ i_h_m) \rightarrow \text{goal} \]

(2)

is satisfied by each model of AX\( P \) plus the verified lemmas of \( P \), where AX\( P \) is a set of first-order formulas representing the definitions of the data types and procedures in \( P \). \( \nu^* \in \mathcal{V}^* \) is a sequence of the free variables in \( \text{seq} \) (given by a set \( \mathcal{V} \) of typed variables) and \( u_j^* \in \mathcal{U}^* \) are sequences of the universally quantified variables in the induction hypotheses \( i_h_j \) of \( \text{seq} \) (given by a set \( \mathcal{U} \) of typed variables). A sequent \( \text{seq} \) is \textit{true} iff (2) is satisfied by the initial model of AX\( P \), cf. [7], [6], [14], hence validity of \( \text{seq} \) entails truth of \( \text{seq} \) (but not vice versa).

The set of sequents defines the language of the \textit{HPL-calculus} (abbreviating \textit{Hypotheses, Programs and Lemmas}), which is the calculus in which the lemmas are proved. The application of a proof rule of this calculus to a sequent yields a finite set of sequents, which are obtained by altering the set of hypotheses, the set of induction hypotheses or the goalterm of the sequent to which the proof rule has been applied. Each proof rule is \textit{sound} in the sense that the truth of all resulting sequents entails the truth of the sequent to which the proof rule has been applied. A proof in the HPL-calculus is represented by a \textit{prooftree},\(^1\)

---

\(^1\) An atom \( a \) is an \textit{if}-free boolean term, and a literal is an atom or a negated atom \( if \{a, false, true\} \), subsequently abbreviated by \( \neg a \) or by \( t \# r \) for negated equations \( t = r \). \( \neg \) stands for the complement of a literal \( l \) and \( \overline{C} := \{T | l \in C\} \) for a clause \( C \).
the nodes of which are given by sequents. The root node of a proof tree for a lemma \textit{lem} is given by the initial sequent \( \vdash \text{body}_{\text{lem}} \), and the successor nodes are given by the sequents resulting from a proof rule application to the father node sequent. A proof of lemma \textit{lem} is obtained, if a proof tree can be built for \textit{lem} such that each leaf is a sequent of the form \ldots \( \vdash \text{true} \), see [12] for details.

2 Symbolic Evaluation

The \textit{HPL}-calculus provides a set of 15 proof rules to create proof trees. For example, \textit{Induction} creates the base and step sequents from an initial sequent wrt. some induction axiom, \textit{Use Lemma} applies a lemma to a sequent, \textit{Case Analysis} creates successor sequents by a case split, \textit{Unfold Procedure} “opens up” a procedure call, etc.

Goal terms are simplified by so-called \textit{computed HPL}-proof rules. For instance, \textit{Simplification} rewrites a sequent’s goal term using the definitions of the data types and procedures, the hypotheses and the induction hypotheses of the sequent and the lemmas already verified. These rewrites are performed by \textit{symbolic evaluation} which differs from the usual evaluation of terms by presence of the free variables of \( V \) in the hypotheses, the induction hypotheses and in the goal term of a sequent which must be treated as \textit{undefined constants}.

Symbolic evaluation is defined by another calculus, called the \textit{evaluation calculus}. The language of this calculus is given by the set \( T(\Sigma(P), V) \) of first-order terms, where \( \Sigma(P) \) stands for the signature of the function symbols defined by an \textit{L}-program \( P \), and \( V \) and \( U \) are the sets of typed variables, used for the free and the universally quantified variables respectively. The inference rules of the evaluation calculus, called \textit{evaluation rules}, are of the form

\[
\begin{aligned}
\frac{\text{term}}{\text{term'}} \quad \text{if COND}
\end{aligned}
\]

where COND stands for a side condition which must be satisfied for applying the evaluation rule. We write \( \text{term} \vdash_{H,A} \text{term'} \) if \( \text{term'} \) originates from \( \text{term} \) by an evaluation rule like (3) using a set \( H \) of literals containing the sequent’s hypotheses \( h_i \) at least and clauses from a finite set \( A \subset CL(\Sigma(P), V \cup U) \) which represents the sequent’s induction hypotheses \( ih_j \) as well as the verified lemmas of \( P \).

For \textit{terminating} \textit{L}-programs \( P \), symbolic evaluation is sound in the sense that each term equals its evaluated counterpart.\(^2\) I.e. \( \text{term} \vdash_{H,A} \text{term'} \) entails validity of

\[
\forall v^*:\tau^*. (\bigwedge_{h \in H} h \land \bigwedge_{D \in A} (\forall u_D: \tau_D. (\bigvee_{i \in D, l} l)) \rightarrow \text{term} = \text{term'})
\]

which means that (4) is satisfied by each model of \( AX_P \), where \( AX_P \) is the set of first-order formulas representing the definitions of the data types and procedures in \( P \), \( v^* \in V^* \) is a sequence of the free variables in \( \{H, A, \text{term}, \text{term'}\} \) and \( u^*_D \in U^* \) is a sequence of the universally quantified variables in \( D \).

\(^2\) Termination of \textit{L}-programs [13] is necessary (but not sufficient) for termination of symbolic evaluation as well as for soundness (cf. e.g. evaluation rule (4) of Fig. 2).
Symbolic evaluations are computed in \texttt{VeriFun} by the Symbolic Evaluator, i.e. an automated theorem prover which considers the evaluation rules in a fixed order. Starting with the goalterm \texttt{goal} of a sequent, a symbolic evaluation of \texttt{goal} is computed by applying the first applicable evaluation rule to \texttt{goal}, then applying the first applicable evaluation rule to the result of the recent evaluation step and so on, until eventually a term is obtained to which no further evaluation rule can be applied. The list of terms obtained thereby defines a deduction in the evaluation calculus, called a symbolic evaluation of the goalterm.

The challenge when developing this theorem prover is to find a satisfactory compromise between the competing requirements of runtime performance, termination and completeness, where “completeness” means that \texttt{true} is returned for each valid sequent. However, since non-valid (but not necessarily false) sequents are frequently subject to symbolic evaluation, completeness must be sacrificed in favor of termination as both requirements (assuming soundness) exclude each other by the non-decidability of first-order logic. So instead of completeness, one demands that a result which is useful for the continuation of an HPL-proof (by some user interaction, say) be returned at least if the symbolic evaluator fails to return \texttt{true} either by incompleteness or because a non-valid sequent was considered.\footnote{Of course, “usefulness” is an imprecise notion which can be judged empirically only by the degree of user interactions which decreases the “more useful” the results are.}

For instance, a non-valid (but true) step sequent is obtained when proving the commutativity of $+$ by induction. Therefore the system should terminate with a symbolically evaluated goalterm (viz. $\mathbf{+}(y + \neg(x)) = y + x$ in the example), which is useful to continue the proof (viz. to start a subsequent induction in the example).

\footnotesize
\begin{figure}[h]
\begin{center}
\begin{tabular}{|c|c|c|}
\hline
\textbf{if} \{\texttt{true}, \texttt{b}, \texttt{c}\} & \textbf{if} \{\texttt{false}, \texttt{b}, \texttt{c}\} & \textbf{if} \{\texttt{a}, \texttt{true}, \texttt{false}\} \\
\hline
\texttt{b} & \texttt{c} & \texttt{a} \\
\hline
(1) Keep then-part & (2) Keep else-part & (3) Skip alternatives \\
\hline
\textbf{if} \{\texttt{a}, \texttt{b}, \texttt{b}\} & \textbf{if} \{\texttt{a}, \texttt{b}, \texttt{c}\}, \texttt{if a} \vdash_{H,A} \texttt{a}' & \textbf{if} \{\neg \texttt{a}, \texttt{b}, \texttt{c}\} \\
\hline
(4) Skip condition & (5) Evaluate condition & (6) Skip negation \\
\hline
\textbf{if} \{\texttt{a}, \texttt{b}, \texttt{d}, \texttt{e}\}, \texttt{if} \{\texttt{c}, \texttt{d}, \texttt{e}\} & \texttt{a}, \texttt{if a} \in H & \texttt{a}, \texttt{if} \neg \texttt{a} \in H \\
\hline
(7) Distribute condition & (8) Affirmative hyp. & (9) Negative hyp. \\
\hline
\textbf{if} \{\texttt{a}, \texttt{b}, \texttt{c}\}, \texttt{if b} \vdash_{H \cup \{\neg \texttt{a}\}, A} \texttt{b}' & \textbf{if} \{\texttt{a}, \texttt{b}, \texttt{c}\}, \texttt{if c} \vdash_{H \cup \{\texttt{a}\}, A} \texttt{c}' & \\
\hline
(10) Evaluate then-part & (11) Evaluate else-part \\
\hline
\end{tabular}
\end{center}
\caption{Evaluation rules for propositional reasoning}
\end{figure}

2.1 Propositional Reasoning

The evaluation rules for propositional reasoning, cf. Fig. 2, form the base of the evaluation calculus. As we demand that the rules be applied in the presented order—thus making symbolic evaluation deterministic—condition $a$ in rules (3) – (11) must be different from true and false, and must be completely evaluated in rules (8) – (11). Except rules (3) and (7), all rules also apply to conditionals with alternatives having a type different from bool.

Rules (1), (2) and (5) define the semantics of if-conditionals like for non-symbolic evaluation of terms. Rule (5) is recursively defined and replaces the condition $a$ of a conditional by the one-step evaluation $a'$ of the condition. Rules (3) and (4) replace void case analyses, rule (6) replaces negated conditions (where $\neg a$ abbreviates $\{a, false, true\}$) and rule (7) rewrites a conditional expression corresponding to the application of DeMorgan’s laws when conventional connectives are used.

Differently to non-symbolic evaluation, symbolic evaluation also explores the alternatives of a conditional if the condition $a$ cannot be evaluated to a truth value: Rule (10) explores the then-part of a conditional using the (evaluated) atom $a$ as an additional hypothesis, and rule (11) similarly performs for the else-part.

2.2 Using Lemmas and Induction Hypotheses

To utilize induction hypotheses and verified lemmas for symbolic evaluation, a set of clauses $S_{lem}$ is computed for a lemma $lem$ by translating the lemma’s body $body_{lem}$ into conjunctive normal form, where each boolean subterm $\{a, b, c\}$ of $body_{lem}$ is treated as an abbreviation for $(\neg a \lor b) \land (a \lor c)$. An induction hypothesis $ih$ of a sequent is treated in the same way, yielding the clause set $S_{ih}$. Now the set $A$ of clauses available for symbolic evaluation of the goalterm of a sequent $seq$ is defined as the smallest clause set containing $S_{lem}$ for each verified lemma $lem$ as well as $S_{ih}$ for each induction hypothesis $ih$ of $seq$.

Clause set $A$, called the assumption set, is used in the following way, cf. Fig. 3: If some atom $t$ contained in a goalterm $goal$, i.e. $goal|_\pi = a$ for some $\pi \in \text{Occ}(goal)$, is subject to symbolic evaluation, it is searched for some clause $D \in A$ and some literal $lit \in D$ such that some matcher $\sigma$ of $lit$ and $a$ exists. Then it is tried to find a matcher $\theta$ of some literals in $\{l \in D \mid U(\sigma(l)) \neq \emptyset\}$ and some literals in $\theta$ such that $U(\theta(\sigma(D))) = \emptyset$. If this succeeds, it is tried to falsify all subgoals $lit' \in \theta(\sigma(D) \setminus \{lit\})$ recursively by symbolic evaluation, and—if successful—$\sigma(lit)$ must hold and $goal$ is soundly replaced by $goal [\pi \leftarrow true]$.

Likewise, the dual evaluation rule (13) Negative assumption of Fig. 3 replaces a goalterm $goal$ by $goal [\pi \leftarrow false]$ if the requirements for (12) Affirmative assumption are satisfied, except that $\sigma$ is a matcher of $lit$ and $a$, and $H \cup \{a\}$ is used to falsify the subgoals. Fig. 4 gives an example of a proof of $\neg n > 2n$ by symbolic evaluation.

---

*Occ(t)* is the set of all occurrences of $t$, $t|_\pi$ denotes the subterm of $t$ at occurrence $\pi$, and term $t [\pi \leftarrow r]$ is obtained from $t$ by replacing $t|_\pi$ by $r$, see e.g. [2], [3], [4].
evaluation of the subgoals. Since goalterms are recursively explored by rules

\[ a \text{ true}, \text{ if } \left\{ \begin{array}{ll}
\begin{array}{l}
\alpha = \sigma(\text{lit}) \text{ and lit}' \vdash_{H \cup \{\neg a\}} \text{false for some } \sigma, \\
\text{some } D \in A, \text{some } \text{lit} \in D, \text{some } \theta \\
\text{with } \mathcal{U}(\theta(\sigma(D))) = \emptyset \text{ and all } \text{lit}' \in \theta(\sigma(D \setminus \{\text{lit}\}))
\end{array}
\end{array} \right. \]

(12) Affirmative assumption

\[ a \text{ false}, \text{ if } \left\{ \begin{array}{ll}
\alpha = \sigma(\text{lit}) \text{ and lit}' \vdash_{H \cup \{\neg a\}} \text{false for some } \sigma, \\
\text{for some } D \in A, \text{some } \text{lit} \in D, \text{some } \theta \\
\text{with } \mathcal{U}(\theta(\sigma(D))) = \emptyset \text{ and all } \text{lit}' \in \theta(\sigma(D \setminus \{\text{lit}\}))
\end{array} \right. \]

(13) Negative assumption

\[ \neg a \vdash_{\emptyset,\{D_1,D_2,D_3\}} \text{true} \]

; by rules (5) and (2) using 1.

1. \[ a \vdash_{\emptyset,\{D_1,D_2,D_3\}} \text{false} \]

; by (13) Neg. assumption using 2.

2. \[ \neg a \vdash_{\emptyset,\{D_1,D_2,D_3\}} \text{false} \]

; by rules (5) and (1) using 3.

3. \[ 2n > a \vdash_{\emptyset,\{D_1,D_2,D_3\}} \text{true} \]

; by (12) Affirm. assump. using 4.

4. \[ n = 0 \vdash_{\emptyset,\{2n,\neg 2n, n, n = 0\},\{D_1,D_2,D_3\}} \text{false} \]

; by (13) Neg. assumption using 5.

5. \[ \neg a \vdash_{\emptyset,\{2n,\neg 2n, n, n = 0\},\{D_1,D_2,D_3\}} \text{false} \]

; by rules (5) and (1) using 6.

6. \[ a \vdash_{\emptyset,\{2n,\neg 2n, n, n = 0\},\{D_1,D_2,D_3\}} \text{true} \]

; by evaluation rule (8)

As a further support, the negation of the atom \( a \) if \( \theta \) does not hold. Soundness of using \( H \cup \{a\} \) upon application of (13) Negative assumption is justified analogously.

---

5 To prevent non-termination (thus introducing incompleteness) when applying the assumption rules, all atoms carry a so-called search limit (initially given by a global constant search depth) which is decremented in the subgoals, and no assumption rule applies to an atom with search limit = 0. See [6] for details.

6 Using \( H \cup \{\neg a\} \) when falsifying the subgoals is sound upon application of (12) Affirmative assumption: Replacing a by true is sound if a holds, and \( \bigwedge_{H \cup \{\neg a\}} h \leftrightarrow \bigwedge_{H} \neg h \) if a does not hold. Soundness of using \( H \cup \{a\} \) is justified analogously.
if \{ n > m, n > k, \ldots \}, n > m \in H \text{ holds when } n > k \text{ is to be evaluated symbolically. As } \sigma(D) \setminus \mathcal{T}(\Sigma(P), \mathcal{V}) = \{ \neg n > y, \neg y > k \} \text{ for the matcher } \sigma = \{ x/n, z/k \} \text{ of } x > z \text{ and } n > k, \theta = \{ y/m \} \text{ is obtained as matcher of } \neg n > y \text{ and } \neg n > m, \text{ hence } \neg n > m \text{ and } \neg m > k \text{ are obtained as subgoals to be falsified. The disproof of subgoal 4. in Fig. 4 yields another example for the benefit of guessing variable instantiations.}

Allowing matching only when applying clauses introduces another source of incompleteness. E.g., $2^n > n$ does not evaluate to true symbolically if $A$ consists of the transitivity clause for $>$ and clause $\{ 2^x > x \}$ only, as the required instance $2^n$ of $y$ cannot be “guessed” when $\{ \neg 2^n > y, \neg y > n \}$ is examined. Replacing the assumption rules by a complete inference machinery could in principle improve the situation, but our experiments with Resolution, e.g. [5], resulted in a drastic deterioration of runtime performance (and theorem proving power in turn).

### 2.3 Further Evaluation Rules

Given a procedure function $f(x_1 : \tau_1, \ldots, x_n : \tau_n) : \tau \leftarrow \text{body}_f$ and a goalterm containing a call of procedure $f$, i.e. $\text{goal}|_\pi = f(t_1, \ldots, t_n)$ for some $\pi \in \text{Occ(goal)}$, $\text{goal}$ may be replaced by $\text{goal} | [\pi \leftarrow \sigma(\text{body}_f)]$, where $\sigma$ is a matcher of $f(x_1, \ldots, x_n)$ and $f(t_1, \ldots, t_n)$. To mimic the call-by-value discipline when executing procedure calls symbolically, it is demanded in addition that the actual parameters $t_i$ are already symbolically evaluated, denoted by $t_i \not\in H.A$. Symbolic execution of procedure calls is implemented by evaluation rule \textit{Execute procedure call} of Fig. 5,\footnote{This evaluation rule only applies for completely defined L-programs, see [14].} where \textit{execute} denotes a system routine assessing heuristically whether it is useful to “open up” a procedure call, see [6] for details.

The evaluation calculus provides further inference rules, e.g. for coping with the data types, rearrangement of boolean subterms etc., which can be found in [6] and are omitted here. The remainder of the paper illustrates the extension of symbolic evaluation for equality reasoning.

### 3 Equality Reasoning

A first-order theory with equality, see e.g. [5], is given by the axioms of \textit{reflexivity} and \textit{substitutivity}

$$\forall x, y : \tau. \ x = y \land \phi[x,x] \rightarrow \phi[x,y] \ . \quad (5)$$
Both axioms entail that \( = \) satisfies the axioms of an equivalence relation as well as the congruence property

\[
\forall x_1, y_1 : \tau_1, \ldots, x_n, y_n : \tau_n. x_1 = y_1 \land \ldots \land x_n = y_n \rightarrow f(x_1, \ldots, x_n) \equiv f(y_1, \ldots, y_n)
\]

(6)

where \( f : \tau_1 \times \ldots \times \tau_n \rightarrow \tau \) with \( \tau \neq \text{bool} \) is any function symbol and \( \equiv \) stands for \( "=\)" if \( \tau \neq \text{bool} \) and for \( "\rightarrow\) otherwise. Hence in our setting, the semantics of the equality symbol \( = \) is given as the least congruence relation on the set \( T(\Sigma(P)) \) of ground terms in the initial model of \( AXP \), i.e. the least equivalence relation satisfying (6).

\textit{Reflexivity} of \( = \) is implemented by evaluation rule (15) Reflexivity of Fig. 6 and \textit{symmetry} of \( = \) is implemented by set membership and matching “modulo symmetry” which means that \( m = n \in \{ n = m \} \) and two matchers, viz. \( \{ x/n, y/m \} \) and \( \{ x/m, y/n \} \), are obtained if term \( x=y \) is matched against term \( n=m \). Finally, the \textit{transitivity axiom} for \( = \) is given explicitly as a clause in \( A \) to be used by the assumption rules like a verified lemma.

As we demand \( = \) being the least congruence relation in the initial model, constructor ground terms are related by \( = \) iff they are identical. This property of \( = \) is implemented by evaluation rules (16) Constructor uniqueness and (17) Constructor injectivity of Fig. 6, where \( \text{cons} \) and \( \text{cons}' \) are distinct constructors and \( \text{AND}(t_1 = r_1, \ldots, t_n = r_n) \) stands for the conjunction \( t_1 = r_1 \land \ldots \land t_n = r_n \) represented by if-conditional. For example, \( 0::l = + (n)::k \vdash_{H,A} \text{if} \{ 0 = + (n), l = k, \text{false} \} \vdash_{H,A} \text{false} \) by evaluation rules (17) and (16).

### 3.1 Functionality

So far, our settings define \( = \) as the least equivalence relation on the set \( T(\Sigma(P)) \) of ground terms and we continue with an implementation of the congruence property (6). We start with the functionality rules which provide a direct implementation of (6) for function symbols \( f : \tau_1 \times \ldots \times \tau_n \rightarrow \tau \) with \( \tau \neq \text{bool} \) and \( p : \tau_1 \times \ldots \times \tau_n \rightarrow \text{bool} \). In both cases, the rules split into an affirmative and a negative part, cf. Fig. 7.

When applying a functionality rule, search is restricted by demanding the presence of a trigger literal in the set \( H \) of hypotheses (or alternatively the redex being of the form \( f(t_1, \ldots, t_n) = f(r_1, \ldots, r_n) \) in case of rule (18)). This causes a functionality rule “to fire” only if facts already established suggest a successful application of the rule.
responding instance of
the general form of an
symbols disappear from a goalterm.

hence it is sound to replace an instance of \( 2 \)
\( n \)
\( x \)

hence each occurrence of
\( U \)
with
\( \text{such that} \)
\( U \)
function symbols. Given a
\( \text{evaluation rule} \)
which restrict the arguments of
\( \text{injective in fact. For example,} \)
\( \text{Of course, more than one clause set} \)
\( S_f^{\text{inj}} \)
may exist for \( f \), e.g. \( C^{\text{inj}} = \{ x = 1, y = 1 \} \)
might be considered in addition.

3.2 Injectivity and Cancellation

The constructor injectivity rule of Fig. 6 may also be applied to injective functions different from constructors. For example, \( \forall x,y \in \mathbb{N}. 2x = 2y \rightarrow x = y \) holds, hence it is sound to replace an instance of \( 2x = 2y \) in a goalterm by the corresponding instance of \( x = y \). Such a replacement is always useful as function symbols disappear from a goalterm.

For some functions, injectivity only holds for some subset of their domain, so the general form of an injectivity clause set for an \( n \)-ary function \( f \) is given by

\[
S_f^{\text{inj}} := \bigcup_{i=1}^{n} \left\{ C_f^{\text{inj}} \cup \{ x_i = y_i, f(x_1, \ldots, x_n) \neq f(y_1, \ldots, y_n) \} \right\}
\]

such that \( \mathcal{U}(C_f^{\text{inj}}) \subseteq \{ x_1, \ldots, x_n, y_1, \ldots, y_n \} \). Clause \( C_f^{\text{inj}} \) contains the literals which restrict the arguments of \( f \) to members of \( f \)'s domain for which \( f \) is injective in fact. For example, \( C_f^{\text{inj}} = \{ x = 0, y = 0 \} \) for the factorial function \( ! \), hence each occurrence of \( n! = m! \) may be replaced by \( n = m \) in a goalterm whenever \( n = 0 \) and \( m = 0 \) can be falsified. \( ^8 \) Such reasoning steps are implemented by evaluation rule (22) Injectivity of Fig. 8.

The cancellation rules are similar evaluation rules implemented for binary function symbols. Given a left-cancellation clause

\[
D_f^{\text{left}} = C_f^{\text{left}} \cup \{ x_1 \neq x_2, f(x_1, y) \neq f(x_2, z), y = z \}
\]

with \( \mathcal{U}(C_f^{\text{left}}) \subseteq \{ x_1, x_2, y, z \} \), each instance of \( f(x_1, y) = f(x_2, z) \) may be replaced by the corresponding instance of \( y = z \) in a goalterm whenever the cor-

\( ^8 \) Of course, more than one clause set \( S_f^{\text{inj}} \) may exist for \( f \), e.g. \( C_f^{\text{inj}} = \{ x = 1, y = 1 \} \)
f(t₁, ..., tₙ) = f(r₁, ..., rₙ) \land (t₁ = r₁, ..., tₙ = rₙ), \quad \{ S'''_{ij} \subseteq A \text{ and } \text{lit'} \vdash^{r \leftarrow}_{H,A} \text{false} \text{ for all } \text{lit'} \in \sigma(C''''_{ij}), \text{ where } \sigma(x_i) = t_i \text{ and } \sigma(y_i) = r_i \text{ for all } 1 \leq i \leq n \}

(23) Injectivity

\frac{f(s₁, t) = f(s₂, r)}{t = r}, \text{ if } \{ D'''_{ij} \in A \text{ and } \text{lit'} \vdash^{r \leftarrow}_{H,A} \text{false} \text{ for each } \text{lit'} \in \sigma(C''''_{ij}) \cup \{x₁ \neq x₂\}, \text{ where } \sigma = \{x₁/s₁, x₂/s₂, y/t, z/r\} \}

(22) Left cancellation

Fig. 8. Evaluation rules for injectivity and cancellation

responding instances of the literals in \( C'''_{ij} \cup \{x₁ \neq x₂\} \) can be falsified. E.g., \( C''_{ij} = \{x = 0, (x) = 0\} \) for the exponentiation function \( \uparrow \), hence each occurrence of \( n \uparrow i = n \uparrow (j \ast k) \) may be replaced by \( i = j \ast k \) in a goalterm whenever \( n = 0 \) and \( \neg \{n\} = 0 \) can be falsified. These reasoning steps are performed by evaluation rule (23) of Fig. 8 (and an analogue rule variant for right cancellation).

Injectivity and cancellation rules preserve equivalence by (6) and their benefit stems from the fact that they simplify equations \( t = r \) also if \( t = r \) neither can be proved nor falsified. For example, functionality rule (18) fails if \( i = j \ast k \neq I \) in the example above, and (13) Negative assumption fails (when using \( D''_{ij} \)) if \( i = j \ast k \neq I \) false (and consequently, injectivity and cancellation clauses need not be considered by the assumption rules).

3.3 Subterm Replacement

Subterm replacement—also called (term) reduction—is the most crucial inference step in equality reasoning. It is based on the substitutivity property (5) and allows to replace a goalterm \( \text{goal} \) by \( \text{goal}[\pi \leftarrow \sigma(\pi)] \) if there is some verified lemma \( \forall \ldots \pi \vdash l \equiv r \) with \( U(\pi) \supseteq U(r) \) and a matcher \( \sigma \) of the pattern term \( l \) and the target term \( \text{goal}[\pi] \). Subterm replacement is generalized to conditional equations \( \forall \ldots \text{cond} \rightarrow \pi \vdash r \) with \( U(\pi) \supseteq U(\text{cond}) \cup U(r) \) which then allows to replace \( \text{goal} \) by \( \text{goal}[\pi \leftarrow \text{if } \{\sigma(\text{cond}), \sigma(r), \text{goal}[\pi]\}] \).

Since (i) each non-boolean subterm of a goalterm is subject to replacement, (ii) equations can be used in any direction and (iii) different equations can be used for replacement of a certain subterm, a high degree of indeterminism is introduced into the reasoning machinery, usually resulting in a huge search space. Hence the challenge is to find a satisfactory compromise between theorem proving power and runtime performance when resolving this indeterminism.

We resolve the indeterminism of the first type by considering the subterms of a goalterm in a fixed order which gives priority to evaluation rules avoiding unnecessary replacements. For example, given an equation (*) \( \forall x : \text{list}[\text{@X}], \text{rev}(\text{rev}(x)) = x \), a term \( \text{hd}(\text{n} : \text{rev}(\text{rev}(\text{k}))) \) is replaced by \( n \) without considering (*). Otherwise the call by value discipline of interpretation of terms is mimicked by symbolic evaluation, which means that subterms are replaced innermost. For
instance, (*) is used to replace a term \( \varepsilon \not<\not> \) \( \text{rev}(\text{rev}(k)) \) by \( \varepsilon \not<\not> k \) in a first step, yielding \( k \) in the subsequent symbolic evaluation steps.

Using Oriented Equations To resolve the indeterminism coming with the use of equations in any direction, we define a relation \( \triangleright \) on terms to orient positive equations in a clause in the same way equations are oriented in term rewrite systems, see e.g. [2], [3], [4]. Using the \( \triangleright \) relation, subterm replacement is implemented in the following way: If some non-boolean subterm \( t \) of a goalterm \( \text{goal} \), i.e. \( \text{goal}\_\pi = t \) for some \( \pi \in \text{Occ}(\text{goal}) \), is considered by symbolic evaluation, it is searched for some clause \( D \in A \) and some equation \( l \not= r \in D \) satisfying \( l \triangleright r \) such that some matcher \( \sigma \) of \( l \) and \( t \) exists. Then it is tried to find a matcher \( \theta \) of \( \sigma(D \setminus \{l = r\}) \) and some subset of \( \overline{H} \), and—if successful—\( \text{goal} \) is soundly replaced by \( \text{goal}[\pi \leftarrow \sigma(r)] \), cf. Fig. 9.

Like for the assumption rules, matcher \( \theta \) is required for replacing all variables in \( \mathcal{U}(\sigma(D)) \) by terms not containing variables from \( \mathcal{U} \). Now if computation of \( \theta \) succeeds, \( \text{lit}' \vdash_{H,\theta}^{\rightarrow} \text{false} \) must hold for each subgoal \( \text{lit}' \in \theta(\sigma(D \setminus \{l = r\})) \), hence \( t = \sigma(r) \) is established. Instead of falsifying the subgoals by considering the hypotheses only, symbolic evaluation might be recursively called to falsify each subgoal \( \text{lit}' \) as implemented for the assumption rules of Fig. 3. However, our experiments revealed that attempts to compute \( \text{lit}' \vdash_{H,A}^{\rightarrow} \text{false} \) is far too expensive when using equations, caused by the huge number of replacement candidates which grows exponentially with the size of a goalterm. As it turned out by several experiments, this restriction upon falsification of the subgoals results in a satisfying balance between theorem proving power and runtime performance.

E.g., when verifying the permutation property of quicksort, cf. lemma \( \text{qs permutes} \) in Fig. 10, the lemmas \(+ \right\ succ\), \( \text{occurs append} \) and \( \text{occurs sm lg} \) of Fig. 10 are required. By our definition of \( \triangleright \) (cf. Def. 1 given subsequently), the equations in the lemmas are oriented as indicated by \( \Rightarrow \). The proof of \( \text{qs permutes} \) is by induction defined by the step sequent \( \{ k \not= \varepsilon \} : \{ \text{ih}1, \text{ih}2 \} \vdash n \not= \text{qs}(k) \Rightarrow n \not= k \), where the induction hypotheses \( \text{ih}1 = \forall n' : \text{N} \ n' \not= \text{qs}(\text{sm}(\text{hd}(k), \text{tl}(k))) \Rightarrow n' \not= \text{sm}(\text{hd}(k), \text{tl}(k)) \) and \( \text{ih}2 = \forall n'' : \text{N} \ n'' \not= \text{qs}(\text{lg}(\text{hd}(k), \text{tl}(k))) \Rightarrow n'' \not= \text{lg}(\text{hd}(k), \text{tl}(k)) \) are oriented as well. Fig. 10 displays a symbolic evaluation for proving the step case of \( \text{qs permutes} \) which frequently uses (24) Assumption replacement.

Orienting Equations When defining the orientation relation \( \triangleright \), care has to be taken to avoid infinite reductions. This would be achieved, of course, if \( \triangleright \) were defined as a noetherian (reduction) ordering as demanded in term rewrite systems,
function [infix] \((x:0X,k:0X):list[0X]):N <=
if \(k=e\) then 0 else if \(x = hd(k)\) then \(+ (x\#tl(k))\) else \(x\#tl(k)\) end end

function qs(k:0X):list[N] <=
if \(k=e\) then \(e\) else qs(sm(hd(k), tl(k))) <> \(hd(k) :: qs(lg(hd(k), tl(k)))\) end

lemma + right succ <= \(\forall x,y:N\ x + (+y) ==> (+x + y)\)

lemma occurs append <= \(\forall x:0X, k, l:0X\ x\#(k < l) \Rightarrow x\#k + x\#l\)

lemma occurs sm lg <= \(\forall n,m:N, k:0X\ n\#sm(m, k) + n\#lg(m, k) \Rightarrow n\#k\)

lemma qs permutes <= \(\forall m,N, k:0X\ n\#qs(k) \Rightarrow n\#k\)

\[n\#qs(k) = n\#k\]

\[+^H.A\]
if \(\{n = hd(k), n\#(qs(sm(hd(k), tl(k)))) <= hd(k) :: qs(lg(hd(k), tl(k))))\)
then \(+ (n\#tl(k))\) applying to \(qs\) and #

\[+^H.A\]
if \(\{n = hd(k), n\#(qs(sm(hd(k), tl(k)))) <= hd(k) :: qs(lg(hd(k), tl(k))))\)
then \(+ (n\#tl(k))\) using occurring append

\[+^H.A\]
if \(\{n = sm(hd(k), tl(k)), . . .\} = + (n\#tl(k)), . . .\)
then \(+ (n\#qs(lg(hd(k), tl(k))))\) using \(th_1\)

\[+^H.A\]
if \(\{n = hd(k), n\#sm(hd(k), tl(k)) + +^H.A\#qs(lg(hd(k), tl(k)))\)
then \(+ (n\#tl(k)), . . .\)
using # applying to #

\[+^H.A\]
if \(\{n = hd(k), n\#sm(hd(k), tl(k)) + +^H.A\#qs(lg(hd(k), tl(k)))\)
then \(+ (n\#tl(k)), . . .\)
using + right succ

\[+^H.A\]
if \(\{n = hd(k), n\#sm(hd(k), tl(k)) + +^H.A\#lg(hd(k), tl(k))\)
then \(+ (n\#tl(k)), . . .\)
using \(th_2\)

\[+^H.A\]
if \(\{n = hd(k), +^H.A\#tl(k)) + +^H.A\#tl(k))\)
then \(+ (n\#tl(k)), . . .\)
using occurs sm lg

\[+^H.A\]
if \(\{n = hd(k), true, . . .\} = + (n\#tl(k))\)
then \(+ (n\#tl(k))\) using \(th_2\)

\[+^H.A\]
true, \(n\#(qs(sm(hd(k), tl(k))) \Rightarrow hd(k) :: qs(lg(hd(k), tl(k))))\)
then \(+ (n\#tl(k))\) using \(th_2\)

\[+^H.A\]
true

Fig. 10. A symbolic evaluation using Assumption replacement
For terms \( l, r \in \mathcal{T}(\Sigma(P), \mathcal{V} \cup \mathcal{U}) \), \( l \triangleright r \) is defined as the smallest relation satisfying:

1. \( \mathcal{U}(l) \supseteq \mathcal{U}(r) \),
2. \( l \triangleright r \),
3. \( l \not\equiv r \),
4. \( l \not\equiv \vartheta(\text{body}_f) \), and
5. \( l \triangleright \vartheta(\text{body}_f) \) for each \( \vartheta := \{x_1/r_1, \ldots, x_n/r_n\} \) such that \( f(r_1, \ldots, r_n) \leq_{\tau} r \) for a procedure \( f(x_1, \ldots, x_n, \tau_0) \triangleleft \text{body}_f \).

An equation \( l = r \) is oriented to \( l \Rightarrow r \) iff \( l \triangleright r \) and \( r \not\triangleright l \).

Requirements (1) and (2) of Definition 1 define \( \triangleright \) as a subset of \( \triangleright \) such that no variables from \( \mathcal{U} \) remain in \( \sigma(r) \) for a matcher \( \sigma \) of \( l \) and some \( t \in \mathcal{T}(\Sigma(P), \mathcal{V}) \).

Requirement (3) excludes the replacement of \( \sigma(l) \) by \( \sigma(r) \) if \( l \) subsumes some subterm of \( r \), as—obviously—finite sequences of reductions might result otherwise. Requirement (4) demands that for each procedure call \( f(r_1, \ldots, r_n) \) in \( r \), the instantiated procedure body \( \vartheta(\text{body}_f) \) obtained by “opening up” this call does not contain a subterm which is subsumed by \( l \). This proviso is necessary for avoiding infinite symbolic evaluations, because subterm replacements alternate

\[ f(\langle x \rangle) = g(0, 0) \quad \text{and} \quad \forall x : \mathbb{N} \quad g(x, y) = f(\langle x \rangle) \]

is trivially verified for procedures \( f \) and \( g \) with procedure body \( = 0 \). As \( f(\langle x \rangle) \triangleright g(0, 0) \) as well as \( g(x, y) \triangleright f(\langle x \rangle) \) by Definition 1, an infinite reduction sequence \( f(1), g(0, 0), f(1), \ldots \) results.

Nevertheless, \( \text{VeriFun} \) provides a safety cut-off allowing a user to stop the symbolic evaluator any time and stopping automatically after 300,000 evaluation rules have been applied to a goalterm. So far, these features were only necessary to search for bugs in the implementation (and in experiments with artificial examples, of course).

For terms \( s, t \in \mathcal{T}(\Sigma(P), \mathcal{V} \cup \mathcal{U}) \), \( s \leq_{\tau} t \) denotes that \( s \) is a subterm of \( t \), \( s \leq t \) stands for containment (or encompassment) of \( s \) in \( t \), i.e. \( \sigma(s) \leq_{\tau} t \) for some substitution \( \sigma \), and \( \not\bowtie \) denotes weak containment, i.e. \( s \not\bowtie t \) if \( \nu(s) \leq_{\tau} t \) for some variable renaming \( \nu \) of \( s \) and \( t \). Commutative variants \( \leq_{\tau}^{\bowtie} \) and \( \not\bowtie^{\bowtie} \) exist for the subterm relation as well as for both containment relations, which compare terms wrt. permutation of arguments in calls of procedures for which commutativity (or symmetry) has been verified.
with applications of other symbolic evaluation rules, and in particular with evaluation rule (14) `Execute procedure call`, cf. Fig. 5. Therefore requirement (4) is needed to avoid that a reduction step using some oriented equation \(l \Rightarrow r\) enables evaluation rule `Execute procedure call` which in turn undoes the reduction, thus enabling a reduction using \(l \Rightarrow r\) again. Consider e.g.

\[
\text{lemma} \ plus \ left \ succ \leq \forall x, y : \mathbb{N} \ plus(+ (x), y) = plus(x, + (y)) \quad (7)
\]

about procedure

\[
\text{function} \ plus(x, y : \mathbb{N}) : \mathbb{N} \leq \text{if } y = 0 \text{ then } x \text{ else } plus(+ (x), - (y)) \text{ end} . \quad (8)
\]

Now if \(plus(+ (x), y) \geq plus(x, + (y))\) held, \(plus(+ (n), m)\) could be replaced by \(plus(n, + (m))\) using (7), and then `Execute procedure call` could be applied yielding \(plus(+ (n), m)\) again by subsequent evaluation steps. But as a subterm of \(\theta(body)\), viz. \(plus(+ (x), - (y))\), is an instance of \(plus(+ (x), y)\) and \(plus(+ (x), y) \leq \theta(body)\) holds. Hence \(plus(+ (x), y) \not\geq plus(x, + (y))\) by Def. 1(4), thus avoiding an infinite symbolic evaluation caused by a wrong orientation of the equation in (7). But if we replace the recursive call in (8) by \(\oplus (plus(x, - (y))\), \(\oplus (x, y) \geq \oplus (x, + (y))\) now holds as \(\oplus (x, y) \not\geq \oplus (x, - (y))\) (and all other requirements of Def. 1 are satisfied too). Hence \(\oplus (n, m) \vdash_{H(A)} plus(n, + (m)) \vdash_{H(A)} plus(n, - (m)) \not\vdash_{H(A)} \).

Finally, requirement (5) of Definition 1 is needed for the same reasons as requirement (4), but for cases in which procedures calls are symbolically executed under commutativity, cf. Section 4.

Relation \(\geq\) is defined using the so-called `orientation calculus`:

**Definition 2.** The language of the `orientation calculus` (\(O\)-calculus for short) is given by finite sets of pairs of terms from \(\mathcal{T}(\Sigma(P), \mathcal{V} \cup \mathcal{U})\) and the inference rules (called `orientation rules`) are given in Fig. 11.\(^{12}\)

For a pair of terms \(l, r \in \mathcal{T}(\Sigma(P), \mathcal{V} \cup \mathcal{U})\) we define \(l \geq r\) iff \(\{l, r\} \vdash_{O} \emptyset\), where \(\vdash_{O}\) denotes derivability in the `O-calculus`.

Derivation \(\vdash_{O}\) (and consequently \(\geq\)) is decidable: Let \(|l|\) denote the size of a term \(l\), let \(\{l_1, r_1\} \geq \{l_2, r_2\}\) iff \(|l_1| + |r_1| > |l_2| + |r_2|\), and let \(\gg\) be the multiset order imposed by \(\gg\) on the multisets of pairs of \(\mathcal{T}(\Sigma(P), \mathcal{V} \cup \mathcal{U})\). Then \(E_1 \vdash_{O} E_2\) entails \(E_1 \gg E_2\) for each \(E_1, E_2\) as easily recognized. With \(\gg\) being well-founded, \(\gg\) also is, cf. \([2],[3],[4]\). Hence with any application of an orientation rule, \(E\) strictly decreases wrt. a well-founded ordering which proves the statement.

For example, \((x + y) \ast (x - y) \geq x \ast x - y \ast y\) is obtained by the `O-deduction` of Fig. 12, hence the oriented equation \((x + y) \ast (x - y) \gg x \ast x - y \ast y\) is obtained as all other requirements of Definition 1 are satisfied as well.

\(^{12}\) \(E \oplus \{t, s\}\) stands for \(E \cup \{t, s\}\), \(f \gg_{\text{succ}} g\) iff \(f \in \Sigma(P)_{\text{proc}}\) and \(g \in \Sigma(body)\setminus\{f\}\), \(\gg_{\text{succ}}\) is the transitive closure of \(\gg_{\text{succ}}\), \#_{\text{succ}}(f) := \{g \in \Sigma(P)_{\text{proc}} \mid f \gg_{\text{succ}} g\}\), and \#\(_f\) counts the number of calls of conditionals in the body of a procedure \(f\), where \(\Sigma(P)_{\text{proc}}\) is the set of procedure functions symbols and \(\Sigma(P)_{\text{proc}}\) is the set of constructor functions symbols in an \(L\)-program \(P\).
\[ \frac{E \oplus (l, r)}{E}, \text{if } l \geq r \]  
\[ \frac{E \oplus (f(\ldots), g(\ldots))}{E}, \text{if } \{ f \notin \Sigma(P)_{\text{cons}} \text{ and } \} \]  
\[ \frac{E \oplus (f(l_1, \ldots, l_n), f(r_1, \ldots, r_n))}{E \cup \bigcup_{i=1}^{n} \{(l_i, r_i), (r_i, l_i)\}}, \text{if } \{ \{r_i, l_i\} \notin \emptyset \text{ for some } i \in \{1, \ldots, n\} \]  
\[ \frac{E \oplus (f(l_1, \ldots, l_n), g(r_1, \ldots, r_m))}{E \cup \bigcup_{i=1}^{n} \{(f(\ldots), r_i)\}}, \text{if } \{ \{r_i, f(\ldots)\} \notin \emptyset \text{ for some } j \in \{1, \ldots, m\}, f, g \notin \Sigma(P)_{\text{cons}} \text{ and } \]  
\[ \begin{align*} 
(i) & \quad g \notin \Sigma(f(\ldots)) \text{ as well as } f \in \Sigma(P)^{\text{proc}} \text{ if } g \in \Sigma(P)^{\text{proc}}, \text{ or} \\
(ii) & \quad f \in \Sigma(P)^{\text{proc}} \text{ and } f \geq_{\text{ases}} g, \text{ or} \\
(iii) & \quad n < m, \text{ or} \\
(iv) & \quad n = m, f, g \in \Sigma(P)^{\text{proc}} \text{ and } \#_{\text{ases}}(f) > \#_{\text{ases}}(g), \text{ or} \\
(v) & \quad n = m, f, g \in \Sigma(P)^{\text{proc}}, \#_{\text{ases}}(f) = \#_{\text{ases}}(g), \#_{f}(f) < \#_{f}(g) \text{ and } \]  
\[ \text{type}(f(\ldots)) = \text{type}(g(\ldots)) \]  
\[ \frac{E \oplus (f(l_1, \ldots, l_n), g(\ldots))}{E \cup \{(l_i, g(\ldots))\}}, \text{if } \{ \text{for some } i \in \{1, \ldots, n\}, f \notin \Sigma(P)_{\text{cons}}, \]  
\[ l_i \geq g(\ldots) \text{ and } g \not\geq_{\text{ases}} f \]  
\[ \frac{E \oplus (f(l_1, \ldots, l_n), g(\ldots))}{E \cup \{(l_i, g(\ldots))\}}, \text{if } \{ \text{for some } i \in \{1, \ldots, n\}, f \in \Sigma(P)_{\text{cons}}, \]  
\[ f(l_1, \ldots, l_n) \geq g(\ldots) \text{ and } g \not\geq_{\text{ases}} f \]  

Fig. 11. Inference rules of the orientation calculus

(1) \[ \{ (x + y) * (x - y), x * x - y * y \} \]
(2) \[ \{ (x + y, x * x - y * y) \} \]
(3) \[ \{ (x + y, x * x), (x + y, y * y) \} \]
(4) \[ \{ (x + y, x), (x + y, y * y) \} \]
(5) \[ \{ (x + y, y * y) \} \]
(6) \[ \{ (x + y) \} \]
(7) \[ \emptyset \]

Fig. 12. A deduction in the O-calculus

15
For all linear list $l$ with orientation rule (O2), hence e.g. those oriented equations are always useful because procedure function symbols are replaced by constructors in a goal term, thus enabling further symbolic evaluation.

**Fig. 13. Examples of oriented (conditional) equations**

Fig. 13 displays some verified (conditional) equations in the domain of natural numbers and in the list domain, all of which are oriented according to Def. 1.\(^\text{13}\) By rule (O1) of Fig. 11, Def. 1 also orients each equation of the form $f(\cdots x \cdots) = x$ with $x \in \mathcal{U}$ like idempotency laws, e.g. $gcd(x, x) = x$, cancellation laws, e.g. $x \neq 0 \rightarrow (x + y)/x = y$, $(x + y) - x = y$, and involutary laws, e.g. $rev(rev(k)) = k$. Equations of the form $f(\cdots) = c(\cdots)$ with $c$ being a constructor are oriented from left to right by orientation rule (O2), hence e.g. $k \neq rev(\text{cut}(k))$ or $\text{cut}(k) = last(k)$, $\text{cut}(k) = \text{cut}((k, k < l)) = (k, k < l) = l < k$, $\text{cut}(k, n \neq 0)$, $\text{cut}(k, n \neq 0)$, $\text{cut}(k, n \neq 0)$, etc. are obtained in addition. The reductions enabled by those oriented equations are always useful because procedure function symbols are replaced by constructors in a goal term, thus enabling further symbolic evaluation steps, e.g. $log_2(n) + \log_2(n) = \log_2(n)$, and associativity laws are always oriented, cf. orientation rule (O4), so e.g. $(x * y) * z = x * (y * z)$, $(k < l) < h = k < l < h$, etc. are obtained as well.

\(^{13}\) For a linear list $k$ and a natural numbers $n$ and $m$, $m|n$ abbreviates $(n \mod m) = 0$, $|k|$ computes the length of $k$, $last(k)$ computes the rightmost element of $k \neq$, $\text{cut}(k)$ removes $last(k)$ from $k \neq$, $\text{rev}(k)$ reverses $k$, $\text{rotate}(n, k)$ rotates $k$ $n$ times, $\text{not}(n, k)$ strips off the first $n$ elements from $k$ if $|k| \geq n$, $k \text{\slash} n$ deletes $n$ from $k$, $\Pi(k)$ multiplies the elements of $k$, $n \neq k$ counts the number of occurrences of $n$ in $k$, $\text{sm}(n, k)$ removes all elements $> n$ from $k$ and $\text{lg}(m, k)$ removes all elements $\leq n$ from $k$. 

16
Equations \( l = r \) with \( l \) and \( r \) being variants of each other are not (and must not be) oriented, cf. requirement (3) of Def. 1. Examples are commutativity laws and generalizations thereof, e.g. \((x - y) - z = (x - z) - y, (k\backslash n)\backslash m = (k\backslash m)\backslash n,\) \( ntl(nll(k, n), m) = ntl(nll(k, m), n), |k< > l| = |l< > k|,\) etc.

Finally, there are equations for which orientation by Definitions 1 or 2 fails. Examples are \( x + x \leftrightarrow 2x,\) quicksort\((k) \leftrightarrow mergesort\((k), k \neq \varepsilon \rightarrow hd\(\text{rev}(k)) = \text{last}(k), x>y \rightarrow x - \varepsilon (y) = \varepsilon (x - y), n \in k \rightarrow n \ast \Pi(k\backslash n) = \Pi(k)\) and \( k \neq \varepsilon \rightarrow \text{cut}(\text{rev}(k)) = \text{rev}(l(k)).\) Of course, failing to orient equations is a disadvantage. Moreover, it is not difficult to modify Definitions 1 and 2 such that these equations become oriented. However, the problem is that such modifications may also yield \( r \gtrsim l \) for terms \( l \) and \( r \) which satisfied \( l \succ r \) before, hence equations which had been oriented before become unoriented by the modifications. Our experiments revealed that making \( \gtrsim \) larger did not make \( \succ \) larger, but resulted in a smaller relation \( \succ \) (not surprisingly, enlarging \( \gtrsim \) also enlarges \( \gtrsim \) and \( \gtrsim \cap \gtrsim \) in turn, hence \( \succ \) becomes smaller). So the problem is to define \( \gtrsim \) in such a way that \( l \succ r \) for “as many” equations \( l \Rightarrow r \) is useful for in proofs as possible, where we determined “useful for as many as” experimentally by a large number of verification case studies which eventually resulted in our Definitions 1 and 2.

**Superposition** To resolve the indeterminism coming with different applicable equations, we proceed like in term rewrite systems, cf. [2], [3], [4]: Given a pair of clauses \( D_1 \cup \{l_1 \Rightarrow r_1\} \) and \( D_2 \cup \{l_2 \Rightarrow r_2\} \) in \( A,\) all critical pairs \( \langle s_1, s_2 \rangle \) obtained by superposition of \( l_1 \) and \( l_2 \) are computed. Next each pair \( \langle s_1, s_2 \rangle \) is simplified to \( \{l_1, l_2\} \) by defining \( s_1 \vdash_{\rho(D_1 \cup D_2),A} l_1, s_2 \vdash_{\rho(D_1 \cup D_2),A} l_2, \) where \( \rho \) is the most general unifier used to compute \( \langle s_1, s_2 \rangle.\) If \( t_1 = t_2,\) it does not matter which of the equations is preferred for replacement as any divergence is resolvable, so no further action is taken. Otherwise, clause \( \rho(D_1 \cup D_2) \cup \{t_1 = t_2\} \) is inserted into clause set \( A,\) where equation \( t_1 = t_2 \) is oriented if possible by Definition 1.

4 Conclusion

Theorem proving performance is further enhanced by incorporation of commutativity laws into some of the evaluation rules: The tests for set membership and equality as well as matching are performed modulo commutativity for each function symbol \( f \) with \( \{f(x, y) = f(y, x)\} \in A \) upon application of Affirmative/Negative hypothesis, Affirmative/Negative assumption and Reflexivity, cf. Figs. 2, 3, 6, hence e.g. \((n + m) > 0 \vdash_{\langle m + n \rangle > 0}, A\) true.\(^{14}\)

\(^{14}\) Unfortunately, matching must not be performed modulo commutativity when applying Assumption replacement, cf. Fig. 9, as this allows a looping use of associativity, viz. \((n + (m + k)) = m + (k + n) = k + (n + m) = n + (m + k).\) As a workaround, commutative variants of equations have to be provided explicitly by the user, who stipulates, for instance, \( x \ast (y + z) = x \ast y + x \ast z\) as well as \((x + y) \ast z = x \ast z + y \ast z.\) To overcome this weakness, we intend to integrate matching modulo associativity+commutativity into a future version of the system.
The execute? heuristic also recognizes commutativity so that, for instance, \( n + ^+ (m) \vdash _{H,A} ^+ (n + m) \) by rule Execute procedure call of Fig. 5 if commutativity of procedure + of Fig. 1 has been verified before. However, this feature only applies to “commutative” procedures \( f \) which satisfy \( \text{execute}! [f(x,y)] \rightarrow \neg \text{execute}! [f(t_1,t_2)] \) for each recursive call \( f(t_1,t_2) \) in body \( f \), cf. [6]. E.g., procedure plus as given by (8) must not be “opened up” under commutativity as otherwise \( \text{plus}(n, ^+ (m)) \vdash _{H,A} \text{plus}( ^+ (n), m) \vdash _{H,A} \text{plus}(n, ^+ (m)) \) results.

The cancellation rules of Fig. 8 recognize commutative, associative as well as commutative+associative function symbols. For example, \( n + (m + k) = l + (n + m) \vdash _{H,A} k = l \) by (23) Left cancellation if commutativity as well as associativity of + has been verified before. Similar to the cancellation rules, (conditional) idempotency for procedures \( f \) is implemented by an evaluation rule as well, provided \( C^\text{idem} \cup \{ x_1 \neq x_2 \}, f(x_1, x_2) = x_1 \} \in A \). One may think that a rule implementation of idempotency is not required by presence of the oriented equation \( f(x_1, x_2) \Rightarrow x_1 \), but implementation by an evaluation rule is advantageous if associativity or commutativity+associativity of \( f \) is recognized. For instance, \( \text{gcd}(n, \text{gcd}(m, n)) \vdash _{H,A} \text{gcd}(n, m) \) is obtained if idempotency, commutativity as well as associativity of \( \text{gcd} \) has been verified before.

Symbolic evaluation as presented here has been developed, refined and optimized by surveying theorem proving power and runtime performance for a large number of case studies, see [1], [7], [8], [9], [11], [12] for examples. It has been integrated into the \texttt{VeriFun} system and proved successful upon verification of functional programs. The \texttt{VeriFun} system is obtainable from the web [1].

Acknowledgement We are grateful to Markus Aderhold and Andreas Schlosser for useful comments on a draft of this paper.

References


15 Symbolic execution of procedure calls under commutativity necessitates requirement (5) of Definition 1, as otherwise infinite symbolic evaluations caused by alternating applications of Assumption replacement and Execute procedure call might result, cf. the discussion of requirement (4) of Definition 1 in Section 3.3.