The 2006 Federated Logic Conference

The Seattle Sheraton Hotel and Towers
Seattle, Washington
August 10 - 22, 2006

IJCAR’06 Workshop

DISPROVING’06:
Non-Theorems, Non-Validity, Non-Provability
August 16th, 2006
Proceedings

Editors:
W. Ahrendt, P. Baumgartner, H. de Nivelle
A Fast Disprover for √eriFun

Markus Aderhold, Christoph Walther, Daniel Szallies, and Andreas Schlosser

Fachgebiet Programmiermethodik, Technische Universität Darmstadt, Germany
{aderhold, chr.walther, szallies, schlosser}@pm.tu-darmstadt.de

Abstract. We present a disprover for universal formulas stating conjectures about functional programs. The quantified variables range over freely generated polymorphic data types, thus the domains of discourse are infinite in general. The objective in the development was to quickly find counter-examples for as many false conjectures as possible without wasting too much time on true statements. We present the reasoning method underlying the disprover and illustrate its practical value in several applications in an experimental version of the verification tool √eriFun.

1 Introduction

As a common experience, specifications are faulty and programs do not meet their intention. Program bugs range from easily detected simple lapses (such as not excluding division by zero or typos when setting array bounds) to deep logical flaws in the program design which emerge elsewhere in the program and therefore are hard to discover.

But programmers' faults are not the only source of bugs. State-of-the-art verifiers synthesize conjectures about a program that are needed (or are at least useful) in the course of verification: Statements may be generalized to be qualified for a proof by induction, the verifier might generate termination hypotheses that ensure a procedure's termination, or it might synthesize conjectures justifying an optimization of a procedure. Sometimes these conjectures can be faulty, i.e. over-generalizations might result, the verifier comes up with a wrong idea for termination, or an optimization simply does not apply.

Verifying that a program meets its specification is a waste of time in all these cases, and therefore one should begin with testing the program beforehand. However, as testing is a time consuming and boring task, machine support is welcome (not to say needed) to relieve the human from the test-and-verify cycle. Program testing can be reformulated as a verification problem: A program conjecture $\phi$ fails the test if the negation of $\phi$ can be verified. However, for proving these negated conjectures, a special verifier—called a disprover—is needed.

In this paper, we present such a disprover for statements about programs written in the functional programming language $\mathcal{L}$ [14], which has been integrated into an experimental version [12] of the interactive verification tool √eriFun [15, 16]. The procedures of $\mathcal{L}$-programs operate over freely generated...
structure bool <= true, false
structure N <= 0, +\((\vdash : N)\)
structure list\[@A\] <= ε, \[infix\] ::(hd : @ A, tl : ... by Comon and Lescanne for solving
equational problems. As disproving is semi-decidable, a complete disprover can
60

Fig. 1. A simple \(\mathcal{L}\)-program

polymorphic data types and are defined by using recursion, case analyses, let-expressions, and functional composition. The data types \(\text{bool}\) for Boolean values and \(\text{N}\) for natural numbers \(\mathbb{N}\) as well as equality \(\equiv : \@ A \times \@ A \rightarrow \text{bool}\) and a procedure \(> : \@ N \times \@ N \rightarrow \text{bool}\) deciding the \(>\)-relation on \(\mathbb{N}\) are predefined in \(\mathcal{L}\). Figure 1 shows an example of an \(\mathcal{L}\)-program that defines a polymorphic data type \(\text{list}[@A]\), list concatenation \(<>\), and list reversal \(\text{rev}\). In this program, the symbols \(true, false\) are constructors of type \(\text{bool}\), \(\vdash\)\(\ldots\) is the selector of the \(\mathbb{N}\)-constructor \(+\)\(\ldots\), and \(hd\) and \(tl\) are the selectors of constructor \(::\) for lists. Subsequently, we let \(\Sigma(P)\) denote the signature of all function symbols defined by an \(\mathcal{L}\)-program \(P\), and \(\Sigma(P)^c\) is the signature of all constructor function symbols in \(P\). An operational semantics for \(\mathcal{L}\)-programs \(P\) is defined by an interpreter \(\text{eval}_P : \mathcal{T}(\Sigma(P)) \mapsto \mathcal{T}(\Sigma(P)^c)\) which maps ground terms to constructor ground terms of the respective monomorphic data types using the definition of the procedures and data types in \(P\), cf. [10,14,18].

In \(\mathcal{L}\), statements about programs are given by expressions of the form \(\text{lemma}\ name <= \forall x_1 : \tau_1, \ldots, x_n : \tau_n b\) (cf. Fig. 1), where \(b\)—called the body of the lemma—is a Boolean term built with the variables \(x_i\) (of type \(\tau_i\)) from a set \(\mathcal{V}\) of typed variables and the function symbols in \(\Sigma(P)\), where case analyses (like in procedure definitions) and the truth values are used to represent connectives. Hence the general form of the proof obligations we are concerned with are universal formulas \(\phi = \forall x_1 : \tau_1, \ldots, x_n : \tau_n b\). Disproving such a formula \(\phi\) is equivalent to proving its negation \(\neg\phi \equiv \exists x_1 : \tau_1, \ldots, x_n : \tau_n \neg b\), and as the domain of each type \(\tau_i\) can be enumerated, disproving \(\phi\) is a semi-decidable problem. A disproof of \(\phi\) (also called a witness of \(\neg\phi\)) can be represented by a constructor ground substitution \(\sigma\) such that \(\text{eval}_P(\sigma(b)) = false\). Consequently, disproving \(\phi\) can be viewed as solving the (semi-decidable) equational problem \(b \equiv false\).

To solve such an equation, we develop two disproving calculi that constitute the two phases of our disprover. The inference rules of both calculi are inspired by the calculus proposed in [6] by Comon and Lescanne for solving equational problems. As disproving is semi-decidable, a complete disprover can
be developed. However, as truth of universal formulas \( \phi \) is not semi-decidable by Gödel’s first incompleteness theorem, disproving \( \phi \) is undecidable. Therefore a complete (and sound) disprover need not terminate. But the use in an interactive environment—such as the one \texttt{VeriFun} provides—requires termination of all subsystems, hence completeness must be sacrificed in favor of termination. The use in an interactive environment also demands runtime performance, so particular care is taken to achieve early failure on non-disprovable conjectures.

In Section 2, we explain how we disprove universal formulas \( \phi \). In Section 3, we demonstrate the practical use of our disprover when \texttt{VeriFun} employs it in different disproving applications. We compare our proposal with related work in Section 4 and conclude with an outlook on future work in Section 5.

## 2 Disproving Universal Formulas

Our disproving method proceeds in two phases. The first phase is based on the elimination calculus (\( \mathcal{E} \)-calculus for short). Its language is given by \( \mathcal{L}_E := \{ \langle E, \sigma \rangle \in \mathcal{L}_E \times \text{Sub} \mid \forall \sigma \cap \text{dom}(\sigma) = \emptyset \} \). \( \mathcal{L}_E \) is the set of clauses in which atoms are built with terms from \( T(\Sigma(P), \mathcal{V}) \) and the predicate symbols \( \models \) of type \( \forall \mathcal{A} \times \forall \mathcal{A} \) and \( \supset \) of type \( \forall \mathbb{N} \times \forall \mathbb{N} \); negative literals are written \( t_1 \not\equiv t_2 \) or \( t_1 \not\supset t_2 \), respectively. \( \text{Sub} \) denotes the set of all constructor ground substitutions \( \sigma \), i.e. \( \sigma(v) \in T(\Sigma(P)^\mathcal{E}) \) for each \( v \in \text{dom}(\sigma) \). The inference rules of the \( \mathcal{E} \)-calculus (defined below) are of the form "\( \langle E, \sigma \rangle \langle \mathcal{E}(E), \sigma \circ \lambda \rangle \)”, if \( \text{COND} \)”, where \( \text{COND} \) stands for a side condition that has to be satisfied to apply the rule, and \( \lambda \in \text{Sub} \). An \( \mathcal{E} \)-deduction is a sequence \( \langle E_1, \sigma_1 \rangle, \ldots, \langle E_n, \sigma_n \rangle \) such that for each \( i \), \( \langle E_i, \sigma_i \rangle \) originates from \( \langle E_i, \sigma_i \rangle \) by applying an \( \mathcal{E} \)-inference rule, and \( \langle E_1, \sigma_1 \rangle \vdash_\mathcal{E} \langle E_n, \sigma_n \rangle \) denotes the existence of such an \( \mathcal{E} \)-deduction.

The second phase of our disproving method uses the solution calculus (\( \mathcal{S} \)-calculus for short). It operates on \( \mathcal{L}_S := \{ \langle E, \sigma \rangle \in \mathcal{L}_S \times \text{Sub} \mid \forall \sigma \cap \text{dom}(\sigma) = \emptyset \} \). \( \mathcal{L}_S \subset \mathcal{L}_E \) is the set of clauses in which atoms are formed with predicate symbols \( \models \), \( \supset \) and terms from \( T(\Sigma(P'), \mathcal{V}) \), where \( \Sigma(P') \) emerges from \( \Sigma(P) \) by removing the function symbols \( f \) and \( \mathcal{A} \) as well as all procedure function symbols. The form of the \( \mathcal{S} \)-inference rules (defined below) and deduction \( \vdash_\mathcal{S} \) are defined identically to the \( \mathcal{E} \)-calculus, and \( \langle E, \sigma \rangle \vdash_\mathcal{S} \langle E'', \sigma'' \rangle \) denotes the existence of a composed deduction \( \langle E, \sigma \rangle \vdash_\mathcal{E} \langle E', \sigma' \rangle \vdash_\mathcal{S} \langle E'', \sigma'' \rangle \).

A substitution \( \sigma \) is an \( \mathcal{E} \)-substitution for a clause \( E \in \mathcal{L}_E \), \( \sigma \in \text{Sub}_E \) for short, iff \( \sigma(v) \in T(\Sigma(P)^\mathcal{E}) \) for each \( v \in \mathcal{V}(E) \). We write \( \sigma \vdash l \) if an \{l\}-substitution \( \sigma \) solves an \( \mathcal{E} \)-literal \( l \), defined by \( \sigma \vdash t_1 \equiv t_2 \) iff \( \text{eval}_P(\sigma(t_1)) = \text{eval}_P(\sigma(t_2)) \) and \( \sigma \vdash t_1 \supset t_2 \) iff \( \text{eval}_P(\sigma(t_1)) > \text{eval}_P(\sigma(t_2)) \). An \( \mathcal{E} \)-clause \( E \) is solved by \( \sigma \in \text{Sub}_E \), \( \sigma \vdash E \) for short, iff \( \sigma \vdash l \) for each \( l \in E \). Both calculi are sound in the sense that \( \langle E, \sigma \rangle \vdash_\mathcal{E} \langle E', \sigma' \rangle \) entails \( \theta \vdash \sigma'(E) \) for each \( \theta \in \text{Sub}_E \) with \( \theta \vdash E' \), and \( \sigma \subseteq \sigma' \).

To disprove a conjecture \( \phi = \forall x_1 : \tau_1, \ldots, x_n : \tau_n \ b \), we search for a deduction \( \langle \{ b \equiv \text{false} \}, \varepsilon \rangle \vdash_\mathcal{S} \langle \emptyset, \sigma \rangle \).

\(^1\) Since the domain of each data type is at most countably infinite, we actually use monomorphic types \( \tau'_i \) instead of the polymorphic types \( \tau_i \) in \( \phi \) without loss of gen-
The inference rules of both calculi are given in the subsequent paragraphs. The most important rules are formally defined whereas others (denoted by rule numbers in italics) are only informally described for the sake of brevity. In order to reduce the depth of the terms in $\mathcal{E}$- and $\mathcal{S}$-literals, some of the rules introduce fresh variables (called “auxiliary unknowns” in [6]), which we denote by $w$ and $w'$. Terms are written as $t$, $t_1$ and $t_2$, and $v$ and $v'$ denote variables.

### 2.1 Inference rules of the $\mathcal{E}$-calculus

The $\mathcal{E}$-calculus consists of the inference rules (1)–(3) and (5)–(8) of Fig. 2 plus rules (4) and (9)–(10) described informally. The purpose of the $\mathcal{E}$-inference rules is to eliminate all occurrences of $if$, $\pi$, and of procedure function symbols so that some $(E, \sigma) \in L_\mathcal{E}$ is obtained by an $\mathcal{E}$-deduction $\langle \{b \equiv false\}, \epsilon \rangle \vdash_\mathcal{E} (E, \sigma)$. All rules are supplied with an additional side condition (*) demanding $E \notin L_\bot$ for each $(E, \sigma)$ they apply to, where $L_\bot$ is the set of all $\mathcal{E}$-clauses containing evident type $\tau_i'$ originates from type $\tau_i$ by instantiating each type variable in $\tau_i$ with type $\mathbb{N}$. E.g., to disprove $\forall k, l : \text{list}[\mathbb{N}, A] \ k < l \Rightarrow l < k$, the monomorphic instance $\forall k, l : \text{list}[\mathbb{N}] \ k < l \Rightarrow l < k$ is considered.

2. Assuming $t \equiv \text{cons}(\ldots)$ is sound, as well-typedness is demanded.

3. This elimination is possible whenever $b \equiv false$ is solvable, as each procedure call needs to be unfolded by rule (3) only finitely many times.
contradictions such as \{t \neq t, \ldots\}, \{0 > t, \ldots\} or \{t \doteq 0, t \doteq 1, \ldots\}. This proviso corresponds to the elimination of trivial disequations and the clash rule for equations in [6].

Rule (1) eliminates an if-conditional and rule (2) eliminates an inner procedure call from a literal.\(^4\) Rule (3) unfolds a call of procedure \(f\) that occurs as a direct argument in a literal. A procedure \(f\) is represented here by a set \(D_f\) of triples \((C, \overline{C}, r)\) such that \(r\) is the if-free result term in the procedure body of \(f\) obtained under the conditions \(C \cup \{\neg c \mid c \in \overline{C}\}\), where \(C\) and \(\overline{C}\) consist of if-free Boolean terms only. E.g., \(D_{<>}\) consists of two triples, viz. \(d_1 = (\{k \doteq \varepsilon\}, \emptyset, l)\) and \(d_2 = (\emptyset, \{k \doteq \varepsilon\}, hdl(k) := (tl(k) <> l))\), for procedure \(<\>\) of Fig. 1.

A further rule (4) translates inequations and equations expressed with symbols from \(\Sigma(P)\) into \(S\)-literals: e.g., “\(t_1 > t_2 \doteq \text{false}\)” is translated into “\(t_1 \neq t_2\)”. Rule (6) is like the decomposition rule from [6] for inequations, but restricted to constructors. Another rule (9)—corresponding to the elimination of trivial equations and clash for inequations in [6]—removes trivial literals such as \(t = t\) or \(0 \neq t\) from a clause \(E \in \mathcal{L}_E\) and supplies arbitrary values in \(\lambda\) for variables that disappear from the clause. Finally, literals are simplified by rule (10), which replaces subterms of the form \(\text{sel}_i(\text{cons}(t_1, \ldots, t_n))\) with \(t_i\). This rule does not exist in [6] and accounts for data type definitions with selectors.

\[\begin{align*}
(15) & \quad \frac{E \ni \{t_1 \odot t_2\}, \sigma}{E \cup \{w_1 \doteq t_1, w_2 \doteq t_2, w_1 \odot w_2\}, \sigma} \quad \text{, if } t_1, t_2 \notin \mathcal{V} \\
(16) & \quad \frac{E \ni \{v \neq t\}, \sigma}{E \cup \{v \neq t, v \doteq \text{cons'(w_1, \ldots, w_n)}\}, \sigma} \quad \text{, if } t \in \mathcal{V} \text{ or } t = \text{cons(...)} \\
(17) & \quad \frac{E \ni \{t_1 \neq t_2\}, \sigma}{E \cup \{t_2 \doteq t_1\}, \sigma} \quad \text{, if } t_1, t_2 \notin \mathcal{V} \\
(18) & \quad \frac{E \ni \{t_1 \neq t_2\}, \sigma}{E \cup \{t_1 \doteq t_2\}, \sigma} \quad \text{, if } t_1, t_2 \in \mathcal{T}(\Sigma(P'), \mathcal{V})
\end{align*}\]

Fig. 3. Inference rules of the \(S\)-calculus

\(^4\) For a literal \(l = t_1 \odot t_2, \upharpoonright_{\pi}\) is the subterm of \(l\) at occurrence \(\pi \in \text{Occ}(l)\), and \(l|_{\pi \leftarrow t}\) is obtained from \(l\) by replacing \(l|_{\pi\,n}\) in \(l\) with \(t\). We use “\(\upharpoonright\)” as a shorthand for the succedents of different rules with the same premises and side conditions. All rules are applied “modulo symmetry” of \(\doteq\) and \(\neq\) if possible (e.g., see rule (5)).

2.2 Inference rules of the \(S\)-calculus

The \(S\)-calculus consists of the inference rules (5)–(19), to which the additional side condition (*) applies as well. Rules (5)–(10) are the same as in the \(E\)-calculus, rules (11)–(14) are “structural” rules to merge (in)equations, to replace variables with terms, and to solve equations \(v \doteq t\) with \(t \in \mathcal{T}(\Sigma(P)\varepsilon)\) by substitutions \(\lambda := \{v/t\}\).

Rules (15)–(18) are given in Fig. 3. Rule (15) removes non-variable arguments from an \(S\)-literal, rule (16) is basically a case analysis on \(v\) using some

63
constructor cons’, and rules (17) and (18) eliminate negative literals. Rule (19) invokes a constraint solver. We call an S-literal \( t_1 \circ t_2 \) a constraint literal iff at least one of the \( t_i \) is a variable of type \( \mathbb{N} \), \( \Sigma(t_1) \cup \Sigma(t_2) \subseteq \{0, +, -\} \), and \( \circ \in \{\div, \succ\} \); \( C \) is the set of all constraint literals. When none of the other S-rules is applicable, rule (19) passes the constraint literals \( E \cap C \) to a modified version of INDIGO [5]: To terminate on cyclic constraints like \( \{x \succ y, y \succ x\} \), we simply limit the number of times a constraint can be used by the number of variables in \( E \cap C \). If \( E \cap C \) can be satisfied, we get a solving assignment \( \lambda \in \text{Sub}_{E \cap C} \); otherwise rule (19) fails.

### 2.3 Search Heuristic and Implementation

By the inherent indeterminism of both calculi, search is required for computing \( \mathcal{E} \) - and \( \mathcal{S} \) - deductions: An \( \mathcal{E} \)-clause \( E \) to be solved defines an infinite \( \mathcal{E} \)-search tree \( T^E_{\mathcal{E}} \) whose nodes are labeled with elements from \( \mathbb{L}_E \). The root node of \( T^E_{\mathcal{E}} \) is labeled with \( \langle E, \varepsilon \rangle \), and \( \langle E'', \sigma'' \rangle \) is a successor of node \( \langle E', \sigma' \rangle \) iff \( \langle E'', \sigma'' \rangle \) originates from \( \langle E', \sigma' \rangle \) by applying some \( \mathcal{E} \)-inference rule. The leaves of \( T^E_{\mathcal{E}} \) are given by the \( \mathcal{E} \)-success and the \( \mathcal{E} \)-failure nodes: \( \langle E', \sigma' \rangle \) is an \( \mathcal{E} \)-success node iff \( E' \in L_S \setminus L_{\perp} \), and \( \langle E', \sigma' \rangle \) is an \( \mathcal{E} \)-failure node iff \( E' \in L_\perp \). A path from the root node to an \( \mathcal{E} \)-success node is called an \( \mathcal{E} \)-solution path. All these notions carry over to \( \mathcal{S} \)-search trees \( T^S_{\mathcal{S}} \) by replacing \( \mathcal{E} \) with \( \mathcal{S} \) literally, except that an \( \mathcal{S} \)-node labeled with \( \langle E', \sigma' \rangle \) is an \( \mathcal{S} \)-success node iff \( E' = \emptyset \), and \( \langle E', \sigma' \rangle \) is an \( \mathcal{S} \)-failure node iff \( E' \neq \emptyset \) and no \( \mathcal{S} \)-inference rule applies to \( \langle E', \sigma' \rangle \).

An \( \mathcal{E} \)-clause \( E \) to be solved defines an infinite \( \mathcal{S} \circ \mathcal{E} \)-search tree \( T^E_{\mathcal{S} \circ \mathcal{E}} \), which originates from \( T^E_{\mathcal{E}} \) by replacing each \( \mathcal{E} \)-success node \( \langle E', \sigma' \rangle \) with the \( \mathcal{S} \)-search tree \( T^S_{\mathcal{S}} \). An \( \mathcal{S} \circ \mathcal{E} \)-solution path in \( T^E_{\mathcal{S} \circ \mathcal{E}} \) is an \( \mathcal{E} \)-solution path \( p \) followed by an \( \mathcal{S} \)-solution path that starts at the \( \mathcal{E} \)-success node of path \( p \).

To disprove a conjecture \( \phi = \forall x_1 : \tau_1, \ldots, x_n : \tau_n b \), the \( \mathcal{S} \circ \mathcal{E} \)-search tree \( T^E_{\mathcal{S} \circ \mathcal{E}} \) is explored to find an \( \mathcal{S} \circ \mathcal{E} \)-solution path. In order to guarantee termination of the search, only a finite part of \( T^E_{\mathcal{S} \circ \mathcal{E}} \) may be explored. Additional side conditions for the \( \mathcal{E} \)- and \( \mathcal{S} \)-inference rules ensure that one rule does not undo another rule’s work. Rules that do not require a choice, e.g. rule (5), are preferred to those that need some choice, e.g. rules (6) and (16).

The most significant restriction in exploring \( T^E_{\mathcal{S} \circ \mathcal{E}} \) (supporting termination at the cost of completeness) comes from an additional side condition (**) of rule (3), called the paramodulation rule in [8]: Definition triples \( d := (C, \mathcal{U}, r) \in D_f \) with \( f \notin \Sigma(r) \), called non-recursive definition triples, can be applied as often

---

5 We use \( \succ \) and \( \not\succ \) in \( L_E \) only to handle calls of the predefined procedure \( \triangleright \) more efficiently by a constraint solver.

6 In our setting there is no need to assign priorities to constraints, so we can simplify the algorithm by treating all constraints as “required” constraints.

7 \( E' \in L_\perp \) is sufficient but not necessary for \( \langle E', \sigma' \rangle \) being an \( \mathcal{S} \)-failure node, as the constraint solver called by rule (19) might fail on some \( \mathcal{S} \)-clause \( E' \in L_S \setminus L_\perp \).

---
as possible. However, if \( f \in \Sigma(r) \), we need to limit the usage of \( d \). Side condition (***) demands that recursive definition triples be used at most once on each side of a literal in each branch of \( T^{E_{\{b \neq \text{false}, c\}}} \). This leads to a fast disprover that works well on simple examples, cf. Sect. 3. We call this restriction *simple paramodulation*.

To increase the deductive performance of the disprover, we can allow more applications of a recursive definition triple \( d \) by considering the “context” of \( f \)-procedure calls. Each procedure call \( f(\ldots) \) in the original formula \( \phi \) is labeled with \((N, \ldots, N) \in \mathbb{N}^k \) for a constant \( N \in \mathbb{N} \), e. g., \( N = 2 \), and \( k = |D_f| \). *Context-sensitive paramodulation* modifies side condition (***) in the following way: A recursive definition triple \( d_i := (C_i, \overline{C}_i, r_i) \in D_f \) may only be used if procedure call \( f(\ldots) \) is labeled with \((n_1, \ldots, n_k) \) such that \( n_i > 0 \). The recursive calls of \( f \) in \( r_i \) are labeled with \((n_1, \ldots, n_{i-1}, n_i - 1, n_{i+1}, \ldots, n_k) \), and the other procedure calls in \( r_i \) are labeled with \((N, \ldots, N) \). Context-sensitive paramodulation still allows only finitely many applications of rule (3), as the labels decrease with each rule application. Section 3 gives examples that illustrate the difference between these alternatives in practice. Note that simple paramodulation is *not* a special case of context-sensitive paramodulation (by setting \( N = 1 \)), because it does not distinguish between different occurrences of a procedure as context-sensitive paramodulation does.

For efficiency reasons (wrt. memory consumption), we explore \( T^{E_{\{b \neq \text{false}, c\}}} \) with a *depth-first* strategy, whereas \( T^{(E', \sigma')}_{S} \) is examined *breadth-first* to avoid infinite applications of rule (16). Two technical optimizations considerably speed up the search for an \( S \circ E \)-solution path. Firstly, caching allows to prune a branch that has already been considered in another derivation. The cache hit rates are about 20%. Secondly, while exploring \( T^{E_{\{b \neq \text{false}, c\}}} \), we can already start a subsidiary \( S \)-search on \( S \)-literals from a clause \( E \notin L_{\bot} \) (even though node \((E, \sigma) \) of \( T^{E_{\{b \neq \text{false}, c\}}} \) is not an \( E \)-success node) and feed the results back to the \( E \)-search node \((E, \sigma) \). For instance, if we derive \( x \equiv + (y) \) from \( E \), we can discard \( E \)-branches that consider the case \( x \equiv 0 \). In conjunction with *simple paramodulation*, this (empirically) leads to early failure on unsolvable examples.

### 3 Using the Disprover

In this section we illustrate the use and the performance of our disprover when it is employed as a subsystem of *VeriFun* [12]. Unless otherwise stated, we use simple paramodulation. We distinguish between conjectures provided by the user and conjectures speculated by the system.

#### 3.1 User-Provided Conjectures

Before trying to verify a program statement, it is advisable to make sure that it does not contain lapses that render it false. E. g., in arithmetic we are often interested in cancelation lemmas such as \( x^k = x^z \rightarrow y = z \). However, the disprover
finds the witness \( \{x/0, y/0, z/1\} \) falsifying the conjecture. Excluding \( x=0 \) does not help, as now the witness \( \{x/1, y/0, z/1\} \) is quickly computed. But excluding \( x=1 \) as well causes the disprover to fail, hence we are expectant that verification of \( \forall x, y, z: \mathbb{N} \ x \neq 0 \land \neg(x \neq 0 \land x^y = x^z \rightarrow y = z) \) will succeed. If we conjecture the associativity of exponentiation, \( (x^y)^z = x^{yz} \), the disprover finds the witness \( \{x/2, y/0, z/0\} \). For the injectivity conjecture of the factorial function, i.e. \( \forall x, y: \mathbb{N} \ x! = y! \rightarrow x = y \), the disprover comes up with the witness \( \{x/1, y/0\} \) and fails if we demand \( x \neq 0 \land y \neq 0 \) in addition. For \( \forall k, l: \text{list}[@A] \ k \ll l \ll k \), the solution \( \{k/0::\varepsilon, l/1::\varepsilon\} \) is computed.

All conjectures from above are disproved within less than a second.\(^8\) One might argue that these disproofs are quite simple, so they should be easy to find. \texttt{VeriFun}'s old disprover [1] basically substitutes the variables with values (or value templates like \( n::k \) for lists) of a limited size and uses a heuristic search strategy to track down a counter-example quickly if one exists. However, such a strategy does not lead to early failure on true conjectures: The old disprover fails after \( 46 \) s on the conjecture that procedure \( \text{perm} \) (deciding whether two lists are a permutation of each other) computes a symmetric relation.\(^9\) The new disprover fails after just a second.

The disprover also helps to find simple flaws in the definition of lemmas or procedures. For instance, it disproves lemma "\( \text{rev} \ll \)" (cf. Fig. 1), yielding \( \{k/0::\varepsilon, l/1::\varepsilon\} \). Also, the termination hypothesis for \( \ll \) is disproved at once if one inadvertently writes \( tl(l) \) in the recursive call of \( \ll \) (instead of \( tl(k) \)). Similar errors are the use of \( \geq \) instead of \( > \) in program conjectures and procedure definitions.

To illustrate the consequences of simple context-sensitive paramodulation, consider formula \( \forall k: \text{list}[@A] \ \text{rev}(k) = k. \) As the smallest solution is \( \{k/0::1::\varepsilon\} \), we need to open \( \text{rev} \) twice. Thus the disprover fails to find this witness with simple paramodulation, but succeeds with context-sensitive paramodulation. The same effect is observed with lemma "\( \text{rev} \ll \)" or with \( \forall x: \mathbb{N} \ x^2 > x^2 \). However, as most conjectures do not need extensive search, we prefer to save time and offer this alternative only as an option to the user who is willing to spend more time on the search for a disproof.

### 3.2 Conjectures Speculated by the System

When generalizing statements by machine, a disprover is needed to detect over-generalizations. E.g., \texttt{VeriFun}'s generalization heuristic [1] tries to generalize \( \phi = \forall k, l: \text{list}[@A] \ \text{half}([k \ll l]) = \text{half}([l \ll k]) \) to \( \phi' = \forall k, l: \text{list}[@A] \ k \ll l \ll l \ll k. \) Our disprover quickly fails on \( \phi' \) and succeeds on \( \phi'' \) (see above), hence generalization \( \phi' \) is a good candidate for a proof by induction, whereas \( \phi'' \) is recognized as an over-generalization of \( \phi. \)

\(^8\) All timing details refer to our single-threaded \texttt{Java} implementation on a 3.2 GHz hyper-threading CPU, where the \texttt{Java} VM was assigned 300 MB of main memory.

\(^9\) The old disprover examined the conjecture for lists of length \( \leq 2 \) and natural numbers between 0 and 2.
Another example of such a generate-and-test cycle is recursion elimination: For user-defined procedures, \texttt{Verifun} synthesizes so-called difference and domain procedures which represent information that is useful for automated analysis of termination [13, 17] and for proving absence of “exceptions” [18] (caused by division by 0, for example). Both kinds of procedures may contain unnecessary recursive calls, which complicate subsequent proofs. Therefore the system generates recursion elimination formulas [13] justifying a sound replacement of some recursive calls with truth values. For those formulas that the system could not prove, the user has to decide whether to support the system either by interactively constructing a proof or by giving a witness to disprove the conjecture. He can also ignore the often unreadable conjectures (which most users do), not being aware that missing a true recursion elimination formula means much more work in subsequent proofs.

For example, for the domain procedure of a tautology checker (cf. procedure \(\texttt{\neg\neg}\) in [14]), \texttt{Verifun} generates 62 recursion elimination formulas. Our disprover falsifies all of them within 33 s. Without a disprover, we wasted four times longer on futile proof attempts from which we cannot conclude anything. With the old disprover, it took more than five times longer to disprove 59 formulas; it failed on the others. For other domain or difference procedures, the disprover performs equally well, so in the vast majority of cases the user does not need to worry about recursion elimination any more. This is a tremendous improvement in user-friendliness.

4 Related Work

The problem of automatically disproving statements in the context of program verification has been tackled in various research projects.

Protzen [8] describes a calculus to disprove universal conjectures in the INKA system [4]. While it apparently performs quite well on false system-generated conjectures, it has a rather poor performance on true ones; if the input conjecture is true, it searches until it reaches an explicit limit of the search depth.

A disprover for KIV is presented in [9]. The existing proof calculus is modified so that it is able to construct disproofs. This interleaves the incremental instantiation of variables and simplifying proof steps. For solvable cases “good results” are reported, whereas performance on unsolvable problems is not communicated.

Ahrendt has developed a complete disprover for free data type specifications [2]. Since the interpretation of function symbols is left open in this loose semantics approach, one needs to consider all models satisfying the axioms when proving the non-consequence of a conjecture \(\phi\). Similarly, the ALLOY modeling system [7] can investigate properties of under-specified models. The corresponding constraint analyzer checks only models with a bounded number of elements in each primitive type, so (like our disprover) it is incomplete. Differently from these approaches, we consider only a fixed interpretation of function symbols (given by the interpreter \texttt{evalP}) in our setting.
Isabelle supports a “quickcheck” command [3] to test a conjecture by substituting random values for the variables several times. A comparison of the success rates and the performance of this approach with our results is planned as future work.

Coral [11] is a system designed to find non-trivial flaws in security protocols. It is based on the Comon-Nieuwenhuis method for proof by consistency and uses a parallel architecture for consistency checking and so-called induction derivation to ensure termination. Finding an attack on a protocol may take several hours with Coral.

5 Conclusion

In the design of our disprover we tried to minimize the time wasted on true conjectures. We achieved this by limiting the application of the paramodulation rule. Apart from this, we do not need any explicit depth limits. In particular, there is no explicit limit on the size of a witness. We also reduce the cost of simplifications by restricting them to selector and constructor calls. By incorporating a constraint solver [5] for inequalities on the predefined data type N for N, we further improved the performance.

We identified several applications of our disprover that considerably improve the productivity when working with the ĖriFun system. The main application of our disprover is bulk processing (such as recursion elimination) or automatic generalization. While it is possible to approximate completeness arbitrarily well to find deeper flaws in a program (conjecture), this would tremendously increase the time wasted on true conjectures. The advantage of our disprover is that it is successful in most solvable cases and quickly gives up in unsolvable cases, as practical experiments reveal.

In future work, we intend to investigate further heuristics for the paramodulation rule, which primarily controls the power of the disprover. We also intend to examine whether the use of verified lemmas supports the disproving process. Finally, it would be interesting to look at combinations of various disproving strategies. When we are aware of the strengths and weaknesses of different strategies, we could possibly decide beforehand which one is most suitable for a specific problem.

References