# Extending SMT Solvers to Higher-Order Logic*

Haniel Barbosa[1], Andrew Reynolds[1], Daniel El Ouraoui[2],
Cesare Tinelli[1], and Clark Barrett[3]

[1] The University of Iowa, Iowa City, USA
[2] University of Lorraine, CNRS, Inria, and LORIA, Nancy, France
[3] Stanford University, Stanford, USA

**Abstract.** SMT solvers have throughout the years been able to cope with increasingly expressive formulas, from ground logics to full first-order logic (FOL). In contrast, the extension of SMT solvers to higher-order logic (HOL) is mostly unexplored. We propose a pragmatic extension for SMT solvers to support HOL reasoning natively without compromising performance on FOL reasoning, thus leveraging the extensive research and implementation efforts dedicated to efficient SMT solving. We show how to generalize data structures and the ground decision procedure to support partial applications and extensionality, as well as how to reconcile quantifier instantiation techniques with higher-order variables. We also discuss a separate approach for redesigning an HOL SMT solver from the ground up via new data structures and algorithms. We apply our pragmatic extension to the CVC4 SMT solver and discuss a redesign of the veriT SMT solver. Our evaluation shows they are competitive with state-of-the-art HOL provers and often outperform the traditional encoding into FOL.

## 1 Introduction

Higher-order (HO) logic is a pervasive setting for reasoning about numerous real-world applications. In particular, it is widely used in proof-assistants (also known as interactive theorem provers) to provide trustworthy, formal, and machine-checkable proofs of theorems. A major challenge in these applications is to automate as much as possible the production of these formal proofs, thereby reducing the burden of proof on the users. An effective approach to achieve stronger automation in proof assistants is to rely on less expressive but more automatic theorem provers to discharge some of the proof obligations. Systems such as HOLyHammer, Miz$\mathbb{AR}$, Sledgehammer, and Why3, which provide a one-click connection from proof-assistants to first-order (FO) provers, have led in recent years to considerable improvements in proof-assistant automation [14]. A similar layered approach is also used by automatic HO provers such as Leo-III [43] and Satallax [17], which regularly invoke FO provers to discharge intermediate goals that depend solely on FO reasoning. However, as noted in previous work [12, 30, 48], in both cases the reduction to FOL has its own disadvantages: full encodings into FO, such as those performed by the *hammers*, may lead to issues with performance, soundness, or completeness. On the other hand, the combination of FO and HO reasoning

* This work was partially supported by the National Science Foundation under award 1656926.

in automatic HO provers may suffer from the HO prover itself having to perform substantial FO reasoning, since it is not optimized for FO proving. This would be the case in HO problems with a large FO component, which occur often in practice. We aim to overcome these shortcomings by extending Satisfiability Modulo Theories (SMT) [8] solvers, a class of highly successful automatic FO provers, to natively support HOL.

The two main challenges for extending SMT solvers to HOL lie in dealing with *partial function applications* and with *functional variables*, i.e., quantifier variables of higher-order type. The former mainly affects term representation and core algorithms, which in FOL are based on the fact that all function symbols are fully applied. The latter impacts quantifier instantiation techniques, which must now account for quantified variables occurring in function symbol positions. Moreover, often HO problems can only be proven if functional variables are instantiated with synthesized $\lambda$-terms, typically via HO unification [23], which is undecidable in general.

*Contributions* We present two approaches for extending SMT solvers to natively support HO reasoning (HOSMT). The first one, the *pragmatic* approach (Section 3), targets existing state-of-the-art SMT solvers with large code bases and complex data structures optimized for the FO case. In this approach, we extend a solver with only minimal modifications to its core data structures and algorithms. In the second approach, the *redesign* approach (Section 4), we rethink a solver's data structures and develop new algorithms aimed specifically at HO reasoning. This approach may lead to better results but is better suited to *lightweight* solvers, i.e., less optimized solvers with a smaller code base. Moreover, this approach provides more flexibility to later develop new techniques especially suited for higher-order reasoning. A common theme of both approaches is that the instantiation algorithms are *not* extended with HO unification. This is a significant enough challenge that we plan to explore in a later phase of this work. We include proofs, more examples, and related work in a technical report [5].

We present an extensive experimental evaluation (Section 5) of our pragmatic and redesign approaches as implemented respectively in the state-of-the-art SMT solver CVC4 [6] and the lightweight solver veriT [16]. Besides comparisons against state-of-the-art HO provers, we also evaluate these solvers against themselves, comparing a native HO encoding using the extensions in this paper to the base versions of the solvers with the more traditional FO encoding (not using the extensions).

*Related work.* The pioneering work of Robinson [41] on using a translation to reduce higher-order reasoning to first-order logic inspired the successful tools such as Sledgehammer [36] and CoqHammer [19] that build on this idea by automating HO reasoning via automatic FO provers. Earlier works on native HO proving are, e.g., Andrews's higher-order resolution [1] and Kohlhase's higher-order tableau [29], inspire the modern day HO provers such as LEO-II [11] and Leo-III [43], implementing variations of HO resolution, and Satallax [17], based on a HO tableau calculus guided by a SAT solver. Our approach however is conceptually closer to recent work by Blanchette et al. [9,48] on *gracefully* generalizing the superposition calculus [2,33] to support higher-order reasoning. As a first step, they have targeted the $\lambda$-free fragment of higher-order logic, presenting a refutationally complete calculus [9] and an initial implementation as a prototype extension of the Zipperposition prover [18]. More recently they integrated

their approach into the state-of-the-art FO prover $E$ [48], showing competitive results against state-of-the-art HO provers. Their next step, as is ours, is to extend their calculus to superposition with $\lambda$-terms while preserving their completeness guarantees.

## 2 Preliminaries

Our monomorphic higher-order language $\mathscr{L}$ is defined in terms of right-associative binary *sort constructors* $\to$, $\times$ and pairwise-disjoint countably infinite sets $\mathcal{S}$, $\mathcal{X}$ and $\mathcal{F}$, of *atomic sorts*, *variables*, and *function symbols*, respectively. We use the notations $\bar{a}_n$ and $\bar{a}$ to denote the tuple $(a_1, \ldots, a_n)$ or the cross product $a_1 \times \cdots \times a_n$, depending on context, with $n \geq 0$. We extend this notation to pairwise binary operations over tuples in the natural way. A *sort* $\tau$ is either an element of $\mathcal{S}$ or a *functional sort* $\bar{\tau}_n \to \tau$ from sorts $\bar{\tau}_n = \tau_1 \times \cdots \times \tau_n$ to sort $\tau$. The elements of $\mathcal{X}$ and $\mathcal{F}$ are annotated with sorts, so that $x : \tau$ is a variable of sort $\tau$ and $\mathsf{f} : \bar{\tau}_n \to \tau$ is an *$n$-ary* function symbol of sort $\bar{\tau}_n \to \tau$. We identify function symbols of sort $\bar{\tau}_0 \to \tau$ with function symbols of sort $\tau$, which we call *constants* when $\tau$ is not a functional sort. Whenever convenient, we drop the sort annotations when referring to symbols.

The set of terms is defined inductively: every variable $x : \tau$ is a term of sort $\tau$. For variables $\bar{x}_n : \bar{\tau}_n$ and a term $t : \tau$ of sort $\tau$, the expression $\lambda \bar{x}_n.\, t$ is a term of sort $\bar{\tau}_n \to \tau$, called a *$\lambda$-abstraction*, with *bound* variables $\bar{x}_n$ and *body* $t$. A variable occurrence is *free* in a term if it is not bound by a $\lambda$-abstraction. For a term $t : \bar{\tau}_n \to \tau$ and terms $t_1 : \tau_1, \ldots, t_m : \tau_m$ with $m \leq n$, the expression $\mathsf{f}(\bar{t}_n)$ is a term, called an *application of* $\mathsf{f}$, the *head* of the application, to the *arguments* $\bar{t}_m$. The application is *total* and has sort $\tau$ if $m = n$; it is *partial* and has sort $\tau_{m+1} \times \cdots \times \tau_n \to \tau$ if $m < n$. A *$\lambda$-application* is an application whose head is a $\lambda$-abstraction. The subterm relation is defined recursively: a term is a subterm of itself; if a term is an application, all subterms of its arguments are also its subterms. Note this is not the standard definition of subterms in HOL, which also includes application heads and all partial applications. The *set of all subterms in a term $t$* is denoted by $\mathbf{T}(t)$. We assume $\mathcal{S}$ contains a sort $o$, the Boolean sort, and that $\mathcal{F}$ contains Boolean constants $\top$, $\bot$, a Boolean unary function $\neg$, Boolean binary functions $\wedge$, $\vee$, and, for every sort $\tau$, a family of equality symbols $\simeq : \tau \times \tau \to o$ and a family of symbols $\mathsf{ite} : o \times \tau \times \tau \to \tau$. These symbols are interpreted in the usual way as, respectively, logical constants, connectives, identity, and *if-then-else* (ITE). We refer to terms of sort $o$ as *formulas* and to terms of sort $\bar{\tau} \to o$ as *predicates*. An *atom* is a total predicate application. A *literal* or *constraint* is an atom or its negation. We assume the language contains the $\forall$ and $\exists$ binders over formulas, defined as usual, in addition to the $\lambda$ binder. A formula or a term is *ground* if it is binder-free. We use the symbol $=$ for syntactic equality on terms. We reserve the names $\mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{f}, \mathsf{g}, \mathsf{h}, \mathsf{p}$ for function symbols; $w, x, y, z$ for variables in general; $F, G$ for variables of functional sort; $r, s, t, u$ for terms; and $\varphi, \psi$ for formulas. The notation $t[\bar{x}_n]$ stands for a term whose free variables are included in the tuple of distinct variables $\bar{x}_n$; $t[\bar{s}_n]$ is the term obtained from $t$ by a simultaneous substitution of $\bar{s}_n$ for $\bar{x}_n$.

We assume $\mathcal{F}$ contains a family $@ : (\bar{\tau}_n \to \tau) \times \tau_1 \to (\tau_2 \times \cdots \times \tau_n \to \tau)$ of *application symbols* for all $n > 1$. We use it to model (curried) applications of terms of functional sort $\bar{\tau}_n \to \tau$. For example, given a function symbol $\mathsf{f} : \tau_1 \times \tau_2 \to \tau_3$

and application symbols $@ : (\tau_1 \times \tau_2 \to \tau_3) \times \tau_1 \to (\tau_2 \to \tau_3)$ and $@ : (\tau_2 \to \tau_3) \times \tau_2 \to \tau_3$, $@(\mathsf{f}, t_1)$ and $@(@(\mathsf{f}, t_1), t_2)$ have, respectively, the same denotation as $\lambda x_2 : \tau_2 . f(t_1, x_2)$ and $\mathsf{f}(t_1, t_2)$.

An *applicative encoding* is a well-known approach for performing HO reasoning using FO provers. This encoding converts every functional sort into an atomic sort, every $n$-ary symbol into a nullary symbol, and uses $@$ to encode applications. Thus, all applications, partial or not, become total, and quantification over functional variables becomes quantification over regular FO variables. We adopt Henkin semantics [10, 27] with extensionality and choice, as is standard in automatic HO theorem proving.

### 2.1 SMT solvers and quantified reasoning

SMT solvers that process quantified formulas can be seen as containing three main components: a preprocessing module, a ground solver, and an instantiation module. Given an input formula $\varphi$, the preprocessing module applies various transformations (for instance, Skolemization, clausification and so on) to it to obtain another, equisatisfiable, formula $\varphi'$. The ground solver operates on the formula $\varphi'$. It abstracts all of its atoms and quantified formulas and treats them as if they were propositional variables. The solver for ground formulas provides an *assignment* $E \cup Q$, where $E$ is a set of ground literals and $Q$ is a set of quantified formulas appearing in $\varphi'$, such that $E \cup Q$ propositionally entails $\varphi'$. The ground solver then determines the satisfiability of $E$ according to a decision procedure for a combination of background theories. If $E$ is satisfiable, the instantiation module of the solver generates new *instances*, ground formulas of the form $\neg(\forall \bar{x}. \psi) \vee \psi\sigma$ where $\forall \bar{x}. \psi$ is a quantified formula in $Q$ and $\sigma$ is a substitution from the variables in $\bar{x}$ to ground terms. These instances will be, after preprocessing, added conjunctively to the input of the ground solver, which will proceed to derive a new assignment $E' \cup Q'$, if possible. This interplay may terminate either if $\varphi'$ is proven unsatisfiable or if a model is found for an assignment $E \cup Q$ that is also a model of $\varphi'$.

Extending SMT solvers to HOL can be achieved by extending these three components so that: (1) the preprocessing module eliminates $\lambda$-abstractions; (2) the ground decision procedure supports a ground extensional logic with partial applications, which we denote QF_HOSMT; and (3) the instantiation module instantiates variables of functional type and takes into account partial applications and equations between functions. We can perform each of these tasks pragmatically without heavily modifying the solver, which is useful when extending highly optimized state-of-the-art SMT solvers (Section 3). Alternatively, we can perform these extensions in a more principled way by redesigning the solver, which better suits lightweight solvers (Section 4).

## 3 A pragmatic extension for HOSMT

We pragmatically extend the ground SMT solver to QF_HOSMT by removing $\lambda$-expressions (Section 3.1), checking ground satisfiability (Section 3.2), and generating models (Section 3.3). Extensions to the instantiation module are discussed in Section 3.4.

### 3.1 Eliminating λ-abstractions and partial applications of theory symbols

To ensure that the formulas that reach the core solving algorithm are $\lambda$-free, a preprocessing pass is used to first eliminate $\lambda$-applications and then eliminate any remaining $\lambda$-abstractions. The former are eliminated via $\beta$-reduction, with each application $(\lambda \bar{x}.\, t[\bar{x}])\ \bar{u}$ replaced by the equivalent term $t[\bar{u}]$. The substitution renames bound variables in $t$ as needed to avoid capture.

Two main approaches exist for eliminating (non-applied) $\lambda$-abstractions: reduction to combinators [35] and $\lambda$-lifting [28]. Combinators allow $\lambda$-terms to be synthesized during solving without the need for HO unification. This translation, however, introduces a large number of quantifiers and often leads to performance loss [13, Section 6.4.2]. We instead apply $\lambda$-lifting in our pragmatic extension.

In $\lambda$-lifting, each $\lambda$-abstraction is replaced by a fresh function symbol, and a quantified formula is introduced to define the symbol in terms of the original expression. Note this is similar to the typical approach used for eliminating ITE expressions in SMT solvers. The new function takes as arguments the variables bound by the respective $\lambda$-abstraction and the free variables occurring in its body. More precisely, $\lambda$-abstractions of the form $\lambda \bar{x}_n.\, t[\bar{x}_n, \bar{y}_m]$ of type $\bar{\tau}_n \to \tau$ with $\bar{y}_m : \bar{\upsilon}_m$ occurring in a formula $\varphi$ are lifted to (possibly partial) applications $\mathsf{f}(\bar{y}_m)$ where $\mathsf{f}$ is a fresh function symbol of type $\bar{\upsilon}_m \times \bar{\tau}_n \to \tau$. Moreover, the formula $\forall \bar{y}_m \bar{x}_n.\, \mathsf{f}(\bar{y}_m, \bar{x}_n) \simeq t[\bar{x}_n, \bar{y}_m]$ is added conjunctively to $\varphi$. To minimize the number of new functions and quantified formulas introduced, eliminated expressions are cached so that the same definition can be reused.

In the presence of a background theory $T$, the norm in SMT, a previous preprocessing step is also needed to make all applications of theory, or *interpreted*, symbols total: each term of the form $\mathsf{h}(\bar{t}_m)$, where $\mathsf{h} : \bar{\tau}_n \to \tau$ is a symbol of $T$ and $m < n$, is converted to $\lambda \bar{x}_{n-m}.\, \mathsf{h}(\bar{t}_m, \bar{x}_{n-m})$, which is then $\lambda$-lifted as above to an uninterpreted symbol $\mathsf{f}$, defined by the quantified formula $\forall \bar{y} \forall \bar{x}_{n-m}.\, \mathsf{f}(\bar{x}_{n-m}) \simeq \mathsf{h}(\bar{t}_m, \bar{x}_{n-m})$, with $\bar{y}$ collecting the free variables of $\bar{t}_m$.

We stress that careful engineering is required to perform $\lambda$-lifting correctly in an SMT solver not originally designed for it. For instance, using the existing machinery for ITE removal may be insufficient, since this may not properly handle instances occurring inside binders or as the head of applications.

### 3.2 Extending the ground solver to QF_HOSMT

Since we operate after preprocessing in a $\lambda$-free setting in which only uninterpreted functions may occur partially applied, lifting the ground solver to QF_HOSMT amounts to extending the solver for ground literals in the theory of Equality and Uninterpreted Functions (EUF) to handle partial applications and extensionality.

The decision procedure for ground EUF adopted by SMT solvers is based on classical congruence closure algorithms [24, 31]. While the procedure is easily extensible to HOL (with partial applications but no $\lambda$-abstractions) via a uniform applicative encoding [32], many SMT solvers require that function symbols occurring in (FO) terms be fully applied. Instead of redesigning the solver to accommodate partial applications, we apply a *lazy applicative encoding* where only partial applications are converted.

$$\frac{t \in \mathbf{T}(E)}{t \simeq t}\ \textsc{Refl} \qquad \frac{t \simeq u}{u \simeq t}\ \textsc{Sym} \qquad \frac{s \simeq t,\ t \simeq u}{s \simeq u}\ \textsc{Trans}$$

$$\frac{\bar{t}_n \simeq \bar{u}_n \quad \mathsf{f}(\bar{t}_n),\ \mathsf{f}(\bar{u}_n) \in \mathbf{T}(E)}{\mathsf{f}(\bar{t}_n) \simeq \mathsf{f}(\bar{u}_n)}\ \textsc{Cong} \qquad \frac{t \simeq u,\ t \not\simeq u}{\bot}\ \textsc{Conflict}$$

$$\frac{\mathsf{f}(\bar{t}_n),\ \mathsf{f} \in \mathbf{T}(E)}{\mathsf{f}(\bar{t}_n) \simeq @(\ldots\ (@(\mathsf{f},\ t_1), \ldots),\ t_n)}\ \textsc{App-encode}$$

$$\frac{\mathsf{f} \not\simeq \mathsf{g} \qquad \mathsf{f},\ \mathsf{g} : \bar{\tau}_n \to \tau \qquad n > 0}{\mathsf{f}(\mathsf{sk}_1,\ \ldots,\ \mathsf{sk}_n) \not\simeq \mathsf{g}(\mathsf{sk}_1,\ \ldots,\ \mathsf{sk}_n)}\ \textsc{Extensionality}$$

where $\mathsf{sk}_1,\ \ldots,\ \mathsf{sk}_n$ are fresh symbols of respective sorts $\tau_1,\ \ldots,\ \tau_n$.

Fig. 1: Derivation rules for checking satisfiability of QF_HOSMT constraints in EUF.

Concretely, during term construction, all partial applications are converted to total applications by means of the binary symbol $@$, while fully applied terms are kept in their regular representation. Determining the satisfiability of a set of EUF constraints $E$ containing terms in both representations is done in two phases: if $E$ is determined to be satisfiable by the regular first-order procedure, we introduce equalities between regular terms (i.e., fully applied terms without the $@$ symbol) and their applicative counterpart and recheck the satisfiability of the resulting set of constraints. However, we only introduce these equalities for regular terms which interact with partially applied ones. This interaction is characterized by function symbols appearing as members of congruence classes in the *$E$-graph*, the congruence closure of $E$ built by the EUF decision procedure. A function symbol occurs in an equivalence class if it is an argument of an $@$ symbol or if it appears in an equality between function symbols. The equalities between regular terms and their applicative encodings are kept internal to the $E$-graph, therefore not affecting other parts of the ground decision procedure.

*Example 1.* Given $\mathsf{f} : \tau \times \tau \to \tau$, $\mathsf{g}, \mathsf{h} : \tau \to \tau$ and $\mathsf{a} : \tau$, consider the set of constraints $E = \{@(\mathsf{f}, \mathsf{a}) \simeq \mathsf{g},\ \mathsf{f}(\mathsf{a}, \mathsf{a}) \not\simeq \mathsf{g}(\mathsf{a}),\ \mathsf{g}(\mathsf{a}) \simeq \mathsf{h}(\mathsf{a})\}$. We have that $E$ is initially found to be satisfiable. However, since $\mathsf{f}$ and $\mathsf{g}$ occur partially applied, we augment the set of constraints with a correspondence between the HO and FO applications of $\mathsf{f}, \mathsf{g}$:

$$E' = E \cup \{@(@(\mathsf{f}, \mathsf{a}), \mathsf{a}) \simeq \mathsf{f}(\mathsf{a}, \mathsf{a}),\ @(\mathsf{g}, \mathsf{a}) \simeq \mathsf{g}(\mathsf{a})\}$$

When determining the satisfiability of $E'$, the equality $@(@(\mathsf{f}, \mathsf{a}), \mathsf{a}) \simeq @(\mathsf{g}, \mathsf{a})$ will be derived by congruence and hence, $\mathsf{f}(\mathsf{a}, \mathsf{a}) \simeq \mathsf{g}(\mathsf{a})$ will be derived by transitivity, leading to a conflict. Notice that we do not require equalities between fully applied terms whose functions *do not* appear in the $E$-graph and their equivalent in the applicative encoding. In particular, the equality $\mathsf{h}(\mathsf{a}) \simeq @(\mathsf{h}, \mathsf{a})$ is not introduced in this example. •

We formalize the above procedure via the calculus in Figure 1. The derivation rules operate on a current set $E$ of constraints. A derivation rule can be applied if its premises

are met. A rule's conclusion either adds an equality literal to $E$ or replaces it by $\bot$ to indicate unsatisfiability. A rule application is *redundant* if its conclusion leaves $E$ unchanged. A constraint set is *saturated* if it admits only redundant rule applications.

Rules REFL, SYM, TRANS, CONG and CONFLICT are standard for EUF decision procedures based on congruence closure, i.e., the smallest superset of a set of equations that is closed under entailment in the theory of equality. The rule APP-ENCODE equates a full application to its applicative encoding equivalent, and it is applied only to applications of functions which occur as subterms in $E$. As mentioned above, this can only be the case if the function itself appears as an argument of an application, which happens when it is partially applied (as argument of $@$ or $\simeq$).

Rule EXTENSIONALITY is similar to how extensionality is handled in decision procedures for extensional arrays [21,44]. If two non-nullary functions are disequal in $E$, then a witness of their disequality is introduced. The extensionality property is characterized by the axiom $\forall \bar{x}_n.\, \mathsf{f}(\bar{x}_n) \simeq \mathsf{g}(\bar{x}_n) \Leftrightarrow \mathsf{f} \simeq \mathsf{g}$, for all functions $\mathsf{f}$ and $\mathsf{g}$ of the same type. The rule ensures the left-to-right direction of the axiom (the opposite one is ensured by APP-ENCODE together with the congruence closure rules). To simplify the presentation we assume that, for every term $@(\ldots\,(@(\mathsf{f},\, t_1), \ldots),\, t_m) : \bar{\tau}_n \to \tau \in \mathbf{T}(E)$, there is a fresh symbol $\mathsf{f}' : \bar{\tau}_n \to \tau$ such that $@(\ldots\,(@(\mathsf{f},\, t_1), \ldots),\, t_m) \simeq \mathsf{f}' \in E$.

*Example 2.* Consider the function symbols $\mathsf{f}, \mathsf{g} : \tau \to \tau$, $\mathsf{a} : \tau$, and the set of constraints $E = \{\mathsf{f} \simeq \mathsf{g},\ \mathsf{f}(\mathsf{a}) \not\simeq \mathsf{g}(\mathsf{a})\}$. The constraints are initially satisfiable with respect to the congruence closure rules, however, since $\mathsf{f}, \mathsf{g} \in \mathbf{T}(E)$, the rule APP-ENCODE will be applied twice to derive $\mathsf{f}(\mathsf{a}) \simeq @(\mathsf{f},\, \mathsf{a})$ and $\mathsf{g}(\mathsf{a}) \simeq @(\mathsf{g},\, \mathsf{a})$. Then, via CONG, from $\mathsf{f} \simeq \mathsf{g}$ we infer $@(\mathsf{f},\, \mathsf{a}) \simeq @(\mathsf{g},\, \mathsf{a})$, which leads to a conflict via transitivity. •

**Decision procedure** Any derivation strategy for the calculus that does not stop until it saturates or generates $\bot$ yields a decision procedure for the satisfiability of QF_HOSMT constraints in the EUF theory, according to the following results for the calculus.

**Proposition 1 (Termination).** *Every sequence of non-redundant rule applications is finite.*

**Proposition 2 (Refutation Soundness).** *A constraint set is unsatisfiable if $\bot$ is derivable from it.*

**Proposition 3 (Solution Soundness).** *Every saturated constraint set is satisfiable.*

Even though we could apply the rules in any order, for better performance we only apply APP-ENCODE and EXTENSIONALITY once other rules have only redundant applications. Moreover, APP-ENCODE has precedence over EXTENSIONALITY.

### 3.3 Model generation for ground formulas

When our decision procedure for QF_HOSMT saturates, it can produce a first-order model $M$ as a witness for the satisfiability of its input. Typically, the models generated

by SMT solvers for theories in first-order logic map uninterpreted functions $\mathsf{f} : \bar{\tau}_n \to \tau$ to functions, denoted $M(\mathsf{f})$, of the form

$$\lambda \bar{x}_n.\, \mathsf{ite}(x_1 \simeq t_1^1 \wedge \ldots x_n \simeq t_n^1, s_1, \ldots, \mathsf{ite}(x_1 \simeq t_1^{m-1} \wedge \ldots x_n \simeq t_n^{m-1}, s_{m-1},\, s_m) \ldots)$$

in which every entry but the last corresponds to an application $\mathsf{f}(t_1^i,\, \ldots,\, t_n^i)$, modulo congruence, occurring in the problem. In other words, functions are interpreted in models $M$ as almost constant functions.

In the presence of partial applications, this scheme can sometimes lead to functions with exponentially many entries. For example, consider the satisfiable formula

$$\mathsf{f}_1(\mathsf{a}) \simeq \mathsf{f}_1(\mathsf{b}) \wedge \mathsf{f}_1(\mathsf{b}) \simeq \mathsf{f}_2 \wedge \mathsf{f}_2(\mathsf{a}) \simeq \mathsf{f}_2(\mathsf{b}) \wedge \mathsf{f}_2(\mathsf{b}) \simeq \mathsf{f}_3 \wedge \mathsf{f}_3(\mathsf{a}) \simeq \mathsf{f}_3(\mathsf{b}) \wedge \mathsf{f}_3(\mathsf{b}) \simeq \mathsf{c}$$

in which $\mathsf{f}_1 : \tau \times \tau \times \tau \to \tau$, $\mathsf{f}_2 : \tau \times \tau \to \tau$, $\mathsf{f}_3 : \tau \to \tau$, and $\mathsf{a}, \mathsf{b}, \mathsf{c} : \tau$. To produce the model values of $\mathsf{f}_1$ as a list of total applications with three arguments into an element of the interpretation of $\tau$, we would need to account for 8 cases. In other words, we require 8 $\mathsf{ite}$ cases to indicate $\mathsf{f}_1(x, y, z) \simeq \mathsf{c}$ for all inputs where $x, y, z \in \{\mathsf{a}, \mathsf{b}\}$. The number of entries in the model is exponential on the "depth" of the chain of functions that each partial application is equal to, which can make model building unfeasible if just a few functions are chained as in the above example.

To avoid such an exponential behavior, model building assigns values for functions in terms of the other functions that their partial applications are equated to. In the above example $\mathsf{f}_1$ would have only two model values, depending on its application's first argument being $\mathsf{a}$ or $\mathsf{b}$, by using the model values of $\mathsf{f}_2$ applied on its two other arguments. In other words, we construct $M(\mathsf{f}_1)$ as the term:

$$\lambda xyz.\, \mathsf{ite}(x \simeq \mathsf{a},\, M(\mathsf{f}_2)(y,\, z), \mathsf{ite}(x \simeq \mathsf{b},\, M(\mathsf{f}_2)(y,\, z), \_))$$

where $M(\mathsf{f}_2)$ is the model for $\mathsf{f}_2$ and _ is an arbitrary value. The model value of $\mathsf{f}_2$ would be analogously built in terms of the model value of $\mathsf{f}_3$. This guarantees a polynomial construction for models in terms of the number of constraints in the problem in the presence of partial applications.

*Extensionality and finite sorts* Model construction assigns different values to terms not asserted equal. Therefore, if non-nullary functions $\mathsf{f}, \mathsf{g} : \bar{\tau}_n \to \tau$ occur as terms in different congruence classes but are not asserted disequal, we ensure they are assigned different model values by introducing disequalities of the form $\mathsf{f}(\bar{\mathsf{sk}}_n) \not\simeq \mathsf{g}(\bar{\mathsf{sk}}_n)$ for fresh $\bar{\mathsf{sk}}_n$. This is necessary because model values for functions are built based on their applications occurring in the constraint set. However, such disequalities are only always guaranteed to be satisfied if $\bar{\tau}_n, \tau$ are infinite sorts.

*Example 3.* Let $E$ be a saturated set of constraints s.t. $\mathsf{p}_1, \mathsf{p}_2, \mathsf{p}_3 : \tau \to o \in \mathbf{T}(E)$ and $E \not\models \mathsf{p}_1 \simeq \mathsf{p}_2 \vee \mathsf{p}_1 \simeq \mathsf{p}_3 \vee \mathsf{p}_2 \simeq \mathsf{p}_3 \vee \mathsf{p}_1 \not\simeq \mathsf{p}_2 \vee \mathsf{p}_1 \not\simeq \mathsf{p}_3 \vee \mathsf{p}_2 \not\simeq \mathsf{p}_3$. In the congruence closure of $E$ the functions $\mathsf{p}_1, \mathsf{p}_2, \mathsf{p}_3$ each occur in a different congruence class but are not asserted disequal, so a naive model construction would, in order to build their model values, introduce disequalities $\mathsf{p}_1(\mathsf{sk}_1) \not\simeq \mathsf{p}_2(\mathsf{sk}_1)$, $\mathsf{p}_1(\mathsf{sk}_2) \not\simeq \mathsf{p}_3(\mathsf{sk}_2)$, and $\mathsf{p}_2(\mathsf{sk}_3) \not\simeq \mathsf{p}_3(\mathsf{sk}_3)$, for fresh $\mathsf{sk}_1, \mathsf{sk}_2, \mathsf{sk}_3 : \tau$. However, if $\tau$ has cardinality one these disequalities make $E$ unsatisfiable, since $\mathsf{sk}_1, \mathsf{sk}_2, \mathsf{sk}_3$ must be equal and $o$ has cardinality 2. •

To prevent this issue, whenever the set of constraints $E$ is saturated, we introduce, for every pair of functions $\mathsf{f}, \mathsf{g} : \bar{\tau}_n \to \tau \in \mathbf{T}(E)$ s.t. $n > 0$ and $E \not\models \mathsf{f} \simeq \mathsf{g} \vee \mathsf{f} \not\simeq \mathsf{g}$, the splitting lemma $\mathsf{f} \simeq \mathsf{g} \vee \mathsf{f} \not\simeq \mathsf{g}$. In the above example this would amount to add the lemmas $\mathsf{p}_1 \simeq \mathsf{p}_2 \vee \mathsf{p}_1 \not\simeq \mathsf{p}_2$, $\mathsf{p}_1 \simeq \mathsf{p}_3 \vee \mathsf{p}_1 \not\simeq \mathsf{p}_3$, and $\mathsf{p}_2 \simeq \mathsf{p}_3 \vee \mathsf{p}_2 \not\simeq \mathsf{p}_3$, thus ensuring that the decision procedure detects the inconsistency before saturation.

### 3.4 Extending the quantifier instantiation module to HOMST

The main quantifier instantiation techniques in SMT solving are trigger-based [22], conflict-based [4, 38], model-based [26, 40], and enumerative [37]. Lifting any of them to HOSMT presents its own challenges. We focus here on extending the $E$-matching [20] algorithm, the keystone of trigger-based instantiation, the most commonly used technique in SMT solvers. In this technique, instantiations are chosen for quantified formulas $\varphi$ based on *triggers*. A trigger is a term (or set of terms) containing the free variables occurring in $\varphi$. Matching a trigger term against ground terms in the current set of assertions $E$ results in a substitution that is used to instantiate $\varphi$.

The presence of higher-order constraints poses several challenges for E-matching. First, notice that the $@$ symbol is an overloaded operator. Applications of this symbol can be selected as terms that appear in triggers. Special care must be taken so that applications of $@$ are not matched with ground applications of $@$ whose arguments have different types. Second, functions can be equated in higher-order logic. As a consequence, a match may involve a trigger term and a ground term with different head symbols. Third, since we use a lazy applicative encoding, our ground set of terms may contain a mixture of partially and fully applied function applications. Thus, our indexing techniques must be robust to handle combinations of the two. The following example demonstrates the last two challenges.

*Example 4.* Consider $E$ with the equality $@(\mathsf{f}, \mathsf{a}) \simeq \mathsf{g}$ and the term $\mathsf{f}(\mathsf{a}, \mathsf{b})$ where $\mathsf{f} : \tau \times \tau \to \tau$ and $\mathsf{g} : \tau \to \tau$. Notice that $\mathsf{g}(x)$ is equivalent modulo $E$ to the term $\mathsf{f}(\mathsf{a}, \mathsf{b})$ under the substitution $x \mapsto \mathsf{b}$. Such a match is found by indexing all terms that are applications of *either* $@(\mathsf{f}, \mathsf{a})$ or $\mathsf{g}$ in a common term index. This ensures when matching $\mathsf{g}(x)$, the term $\mathsf{f}(\mathsf{a}, \mathsf{b})$, whose applicative counterpart is $@(@(\mathsf{f}, \mathsf{a}), \mathsf{b})$, is considered.

We extended the regular first-order $E$-matching algorithm of CVC4 as described in this section. Extensions to the other instantiation techniques of CVC4, such as model-based quantifier instantiation, are left as future work.

*Extending expressivity via axioms* Even though not synthesizing $\lambda$-abstractions prevents us from fully lifting the above instantiation techniques to HOL, we remark that, as we see in Section 5, this pragmatic extension very often can prove HO theorems, many times even at higher rates than full-fledged HO provers. Success rates can be further improved by using well-chosen axioms to prove problems that otherwise cannot be proved without synthesizing $\lambda$-abstractions.

*Example 5.* Consider the ground formula $\varphi = \mathsf{a} \not\simeq \mathsf{b}$ with $\mathsf{a}$, $\mathsf{b}$ of sort $\tau$ and the quantified formula $\psi = \forall F, G : \tau \to \tau.\, F \simeq G$. Intuitively $\psi$ states that all functions of sort $\tau \to \tau$ are equal. However, this is inconsistent with $\varphi$, which forces $\tau$ to contain at least

two elements and therefore $\tau \to \tau$ to contain at least four functions. For a prover to detect this inconsistency it must apply an instantiation like $\{F \mapsto (\lambda w.\, \mathsf{a}),\ G \mapsto (\lambda w.\, \mathsf{b})\}$ to $\psi$, which would need HO unification. However, adding the axiom

$$\forall F : \tau \to \tau.\, \forall x, y : \tau.\, \exists G : \tau \to \tau.\, \forall z : \tau.\, G(z) \simeq \mathsf{ite}(z \simeq x,\ y,\ F(z)) \qquad \text{(SAX)}$$

makes the problem provable without the need to synthesize $\lambda$-abstractions. •

We denote the above axiom as the *store axiom* (SAX) because it simulates how arrays are updated via the store operation. As we note in Section 5, introducing this axiom for all functional sorts occurring in the problem often allows our pragmatically extended solver to prove problems it would not be able to prove otherwise. Intuitively, the reason is that instances can be generated not only from terms in the original problem, but also from the larger set of functions representable in the formula signature.

## 4 Redesigning a solver for HOSMT

In the previous section we discussed how to address the challenges of HO reasoning in SMT while minimally changing the SMT solver. Alternatively, we can redesign the solver to support HO features directly. However, this requires a redesign of the core data structures and algorithms. We propose one such redesign below. We again assume that the solver operates on formulas with no $\lambda$-abstraction and no partial applications of theory symbols, which can be achieved via preprocessing (Section 3.1).

### 4.1 Redesigning the core ground solver for HOSMT

Efficient implementations of the congruence closure (CC) procedure for EUF reasoning operate on Union-Find data structures and have asymptotic time complexity $\mathcal{O}(n \log n)$. To accommodate partial applications, we propose a simpler algorithm which operates on an $E$-graph where nodes are terms, and edges are relations (equality, congruence, disequality) between them. An equivalence class is a connected component without disequality edges. All operations on the graph (incremental addition of new constraints, backtracking, conflict analysis, proof production) are implemented straightforwardly. This simpler implementation comes at the cost of higher worse-case time complexity (the CC algorithm becomes quadratic) but integrates better with various other features such as term addition, support of injective functions, rewriting or even computation, in particular for $\beta$- and $\eta$-conversion, which now can be done during solving rather than as preprocessing. In the redesigned approach, the solver keeps two term representations, a curried representation and a regular one. In the regular one, partial and total applications are distinguished by type information. The curried representation is used only by the congruence closure algorithm. It is integrated with the rest of the solver via an interface with translation functions `curry` and `uncurry` between the two different representations. For conciseness, instead of writing $@(\ldots\ (@(\mathsf{f},\ t_1), \ldots),\ t_n)$ below, we use the curried notation $(\cdots\ ((\mathsf{f}\ t_1) \cdots)\ t_n)$, omitting parenthesis when unambiguous.

*Example 6.* Given $\mathsf{f} : \tau \times \tau \to \tau$, $\mathsf{g}, \mathsf{h} : \tau \to \tau$ and $\mathsf{a} : \tau$, consider the constraints $\{\mathsf{f}(\mathsf{a}) \simeq \mathsf{g},\ \mathsf{f}(\mathsf{a}, \mathsf{a}) \not\simeq \mathsf{g}(\mathsf{a}),\ \mathsf{g}(\mathsf{a}) \simeq \mathsf{h}(\mathsf{a})\}$. The congruence closure module will operate on $\{\mathsf{f}\,\mathsf{a} \simeq \mathsf{g},\ \mathsf{f}\,\mathsf{a}\,\mathsf{a} \not\simeq \mathsf{g}\,\mathsf{a},\ \mathsf{g}\,\mathsf{a} \simeq \mathsf{h}\,\mathsf{a}\}$, thanks to the `curry` translation. •

SMT solvers generally perform theory combination via equalities over terms shared between different theories. Given the different term representations kept between the CC procedure and the rest of the solver, to ensure that theory combination is done properly, the redesigned core ground solver keeps track of terms shared with other theory solvers. Whenever an equality is inferred on a term whose translation is shared with another theory, a shared equality is sent out in terms of the translation.

*Example 7.* Consider the function symbols $\mathsf{f} : \mathsf{Int} \to \mathsf{Int}$, $\mathsf{p} : \mathsf{Int} \to o$, $\mathsf{a}, \mathsf{b}, \mathsf{c}_1, \mathsf{c}_2, \mathsf{c}_3, \mathsf{c}_4 : \mathsf{Int}$, the set of arithmetic constraints $\{\mathsf{a} \leq \mathsf{b},\ \mathsf{b} \leq \mathsf{a},\ \mathsf{p}(\mathsf{f}(\mathsf{a}) - \mathsf{f}(\mathsf{b})),\ \neg\mathsf{p}(0),\ \mathsf{c}_1 \simeq \mathsf{c}_3 - \mathsf{c}_4,\ \mathsf{c}_2 \simeq 0\}$, and the set of curried equality constraints $E = \{\mathsf{p}\,\mathsf{c}_1,\ \neg(\mathsf{p}\,\mathsf{c}_2),\ \mathsf{c}_3 \simeq \mathsf{f}\,\mathsf{a},\ \mathsf{c}_4 \simeq \mathsf{f}\,\mathsf{b}\}$. The equalities $\mathsf{c}_3 \simeq \mathsf{f}\,\mathsf{a}$ and $\mathsf{c}_4 \simeq \mathsf{f}\,\mathsf{b}$ keep track of the fact that $\mathsf{f}\,\mathsf{a}$ and $\mathsf{f}\,\mathsf{b}$ are shared. The arithmetic module deduces $\mathsf{a} \simeq \mathsf{b}$, which is added to $E' = E \cup \{\mathsf{a} \simeq \mathsf{b}\}$. By congruence, $\mathsf{f}\,\mathsf{a} \simeq \mathsf{f}\,\mathsf{b}$ is derived, which propagates $\mathsf{c}_3 \simeq \mathsf{c}_4$ to the arithmetic solver. With this new equality, arithmetic reasoning derives $\mathsf{c}_1 \simeq \mathsf{c}_2$, whose addition to the equality constraints produces the unsatisfiable constraint set $E' \cup \{\mathsf{c}_1 \simeq \mathsf{c}_2\}$. •

*Extensionality* The EXTENSIONALITY rule (Figure 1) is sufficient for handling extensionality at the ground level. However, it has shortcomings when quantifiers, even just first-order ones, are considered, as shown in the example below. In the redesigned solver, extensionality is better handled via axioms.

*Example 8.* Consider the constraints $E = \{\mathsf{h}\,\mathsf{f} \simeq \mathsf{b},\ \mathsf{h}\,\mathsf{g} \not\simeq \mathsf{b},\ \forall x.\, \mathsf{f}(x) \simeq \mathsf{a}, \forall x.\, \mathsf{g}(x) \simeq \mathsf{a}\}$, with $\mathsf{h} : \tau \to \tau \to \tau$, $\mathsf{f}, \mathsf{g} : \tau \to \tau$, $\mathsf{a}, \mathsf{b} : \tau$. The pragmatic solver could prove this problem unsatisfiable only with a ground decision procedure that derives consequences of disequalities, since deriving $\mathsf{f} \not\simeq \mathsf{g}$ is necessary to derive $\mathsf{f}(\mathsf{sk}) \not\simeq \mathsf{g}(\mathsf{sk})$, via extensionality, which then leads to a conflict. But SMT solvers are well known not to propagate all disequalities for efficiency considerations. In contrast, with the axiom $\forall F, G : \bar{\tau}_n \to \tau.\, F \not\simeq G \Rightarrow F(\mathsf{sk}_1, \ldots, \mathsf{sk}_n) \not\simeq G(\mathsf{sk}_1, \ldots, \mathsf{sk}_n)$, the instantiation $\{F \mapsto \mathsf{f},\ G \mapsto \mathsf{g}\}$ (which may be derived, e.g., via enumerative instantiation, since $\mathsf{f}, \mathsf{g} \in \mathbf{T}(E)$), provides the splitting lemma $\mathsf{f} \simeq \mathsf{g} \vee \mathsf{f}(\mathsf{sk}) \not\simeq \mathsf{g}(\mathsf{sk})$. The case $E \cup \{\mathsf{f} \simeq \mathsf{g}\}$ leads to a conflict by pure ground reasoning, while the case $E \cup \{\mathsf{f}\,\mathsf{sk} \not\simeq \mathsf{g}\,\mathsf{sk}\}$ leads to a conflict from the instances $\mathsf{f}(\mathsf{sk}) \simeq \mathsf{a},\ \mathsf{g}(\mathsf{sk}) \simeq \mathsf{a}$ of the quantified formulas in $E$. •

### 4.2 Quantifier Instantiation module

In the pragmatic approach, the challenges for the $E$-matching procedure lied in properly accounting for the @ symbol, functional equality, and the mixture of partial and total applications, all of which lead to different term representations, in the term indexing data structure. In the redesign approach, the second challenge remains the same, and term indexing is extended in the same manner of Section 3.4 to cope with it. The first and third challenge present themselves in a different way, however, since the curried representation of terms is only used inside the $E$-graph of the new CC procedure. To apply $E$-matching properly, term indexing is extended to perform query by types, returning all the subterms of a given type that occur in the $E$-graph, but translated back to the uncurried representation.

*Example 9.* Consider $E = \{\mathsf{f}(\mathsf{a}, \mathsf{g}(\mathsf{b}, \mathsf{c})) \simeq \mathsf{a}, \forall F.\, F(\mathsf{a}) \simeq \mathsf{h},\ \forall y.\, \mathsf{h}(y) \not\simeq \mathsf{a}\}$ and the set of triggers $\{F(\mathsf{a}), \mathsf{h}(y)\}$ where $\mathsf{a}, \mathsf{b}, \mathsf{c} : \tau$, $\mathsf{h} : \tau \to \tau$ and $\mathsf{f}, \mathsf{g} : \tau \times \tau \to \tau$. The set of ground curried terms in $E$ is $\{\mathsf{f}\,\mathsf{a}\,(\mathsf{g}\,\mathsf{b}\,\mathsf{c}), \mathsf{f}\,\mathsf{a}, \mathsf{g}\,\mathsf{b}, \mathsf{g}\,\mathsf{b}\,\mathsf{c}, \mathsf{f}, \mathsf{g}, \mathsf{a}, \mathsf{b}, \mathsf{c}\}$. To do $E$-matching with $F(\mathsf{a})$ and $\mathsf{h}(y)$ the index returns the sets of uncurried subterms $\{\mathsf{f}(\mathsf{a}, \mathsf{g}(\mathsf{b}, \mathsf{c})), \mathsf{a}, \mathsf{g}(\mathsf{b}, \mathsf{c}), \mathsf{b}, \mathsf{c}\}$ and $\{\mathsf{f}(\mathsf{a}), \mathsf{g}(\mathsf{b})\}$ for the types $\tau$ and $\tau \to \tau$, respectively. •

Since we do not perform HO unification, to instantiate functional variables it suffices to extend the standard $E$-matching algorithm applied by SMT solvers by accounting for function applications with variable heads. When matching a term $F(\bar{s}_n)$ with a ground term $t$ the procedure essentially matches $F$ with the head of ground terms $\mathsf{f}(\bar{t}_n)$ congruent to $t$, as long as each $s_i$ in $\bar{s}_n$ can be matched with each $t_i$ in $\bar{t}_n$. In the above example, matching the trigger $F(\mathsf{a})$ with the term $\mathsf{f}(\mathsf{a})$ yields the substitution $\{F \mapsto \mathsf{f}\}$.

## 5 Evaluation

We have implemented the above techniques in the state-of-the-art CVC4 solver and in the lightweight veriT solvers. We distinguish between two main versions of each solver: one that performs a full applicative encoding (Section 2) into FOL a priori, denoted @cvc and @vt, and another that implements the pragmatic (Sections 3) or redesigned (Section 4) extensions to HOL within the solvers, denoted cvc and vt. Both CVC4 modes eliminate $\lambda$-abstractions via $\lambda$-lifting. Neither veriT configuration supports benchmarks with $\lambda$-abstractions. The CVC4 configurations that employ the "store axiom" (Section 3.4) are denoted by having the suffix -sax.

We use the state-of-the-art HO provers Leo-III [43], Satallax [17,25] and Ehoh [42, 48] as baselines in our evaluation. The first two have refutationally complete calculi for extensional HOL with Henkin semantics, while the third only supports $\lambda$-free HOL without first-class Booleans. For Leo-III and Satallax we use their configurations from the CASC competition [47], while for Ehoh we report on their best non-portfolio configuration from Vukmirović et al., Ehoh hb, [48].

We split our account between the case of proving HO theorems and that of producing countermodels for HO conjectures since the two require different strengths from the system considered. We discus only two of them, CVC4 and Satallax, for the second evaluation. The reason is that Leo-III and veriT do not provide models and Ehoh is not model-sound with respect to Henkin semantics, only with respect to $\lambda$-free Henkin semantics. We ran our experiments on a cluster equipped with Intel E5-2637 v4 CPUs running Ubuntu 16.04, providing one core, 60 seconds, and 8GB RAM for each job. The full experimental data is publicly available.[1]

We consider the following sets[2] of HO benchmarks: the 3,188 monomorphic HO benchmarks in TPTP [46], split into three subsets: the 530 problems that are both $\lambda$-free and without first-class Booleans (TH0); the 743 that are only $\lambda$-free ($o$TH0); and the 1,915 that are neither ($\lambda o$TH0). The next sets are Sledghammer (SH) benchmarks

[1] http://matryoshka.gforge.inria.fr/pubs/hosmt/

[2] Since veriT does not parse TPTP, its reported results are on the equivalent benchmarks as translated by CVC4 into the HOSMT language [3].

| Solver | Total | TH0 | $o$TH0 | $\lambda o$TH0 | $\text{JD}_{\text{lift}}^{32}$ | $\text{JD}_{\text{combs}}^{32}$ | $\text{JD}_{\text{lift}}^{512}$ | $\text{JD}_{\text{combs}}^{512}$ | $\lambda o\text{SH}^{1024}$ |
|---|---|---|---|---|---|---|---|---|---|
| # | 9032 | 530 | 743 | 1915 | 1253 | 1253 | 1253 | 1253 | 832 |
| @cvc | 4318 | 384 | 344 | 940 | 457 | 459 | 655 | 667 | **412** |
| @cvc-sax | 4348 | 390 | 373 | 937 | 456 | 457 | 655 | **668** | **412** |
| cvc | 4232 | 389 | 342 | 865 | 463 | 447 | **667** | 654 | 405 |
| cvc-sax | 4275 | 389 | 376 | 883 | 458 | 443 | **667** | 654 | 405 |
| Leo-III | **4410** | **402** | 452 | 1178 | **491** | **482** | 609 | 565 | 231 |
| Satallax | 3961 | 392 | **457** | **1215** | 394 | 390 | 407 | 404 | 302 |
| @vt | | 370 | 332 | | 404 | 396 | 525 | 529 | |
| vt | | 369 | 346 | | 426 | 424 | 550 | 556 | |
| Ehoh | | 394 | | | 489 | 481 | 637 | 630 | |

Table 1: Proved theorems per benchmark set. Best results are in **bold**.

from the Judgment Day test harness [15], consisting of 1,253 provable goals *manually* chosen from different Isabelle theories [34] and encoded into $\lambda$-free monomorphic HOL problems without first-class Booleans. The encoded problems are such that they are provable only if the original goal is. These problems are split into four subsets, $\text{JD}_{\text{lift}}^{32}$, $\text{JD}_{\text{combs}}^{32}$, $\text{JD}_{\text{lift}}^{512}$, and $\text{JD}_{\text{combs}}^{512}$ depending, respectively, on whether they have 32 or 512 Isabelle lemmas, or facts, and whether $\lambda$-abstractions are removed via $\lambda$-lifting or via SK-style combinators. The last set, $\lambda o\text{SH}^{1024}$, has 832 SH benchmarks from 832 provable goals *randomly* selected from different Isabelle theories, encoded with 1,024 facts and preserving $\lambda$s and first-class Booleans. Considering a varying number of facts in the SH benchmarks emulates the needs of increasingly larger problems in interactive verification, while different $\lambda$ handling schemes allow us to measure from which alternative each particular solver benefits more.

We point out that our extensions of CVC4 and veriT do not significantly compromise their performance on FO benchmarks. The pragmatic extension of CVC4 has virtually the same performance as the original solver on SMT-LIB [7], the standard SMT test suite. The redesigned veriT does have a considerably lower performance. However, while it is, for example, three times slower on the QF_UF category of SMT-LIB due to its slower ground solver for EUF, it still performs better on this category than CVC4. This shows that despite the added cost of supporting higher-order reasoning, the FO performance of veriT is still on par with the state of the art.

### 5.1 Proving HO theorems

The number of theorems proved by each solver configuration per benchmark set is given in Table 1. Grayed out cells represent unsupported benchmark sets. Figure 2 compares benchmarks solved per time. It only includes benchmark sets supported by all solvers (namely TH0 and the JD benchmarks).

As expected, the results vary significantly between benchmark sets. Leo-III and Satallax have a clear advantage on TPTP, which contains a significant number of small logical problems meant to exercise the HO features of a prover. Considering the TPTP benchmarks from less to more expressive, i.e., including first-class Booleans and then

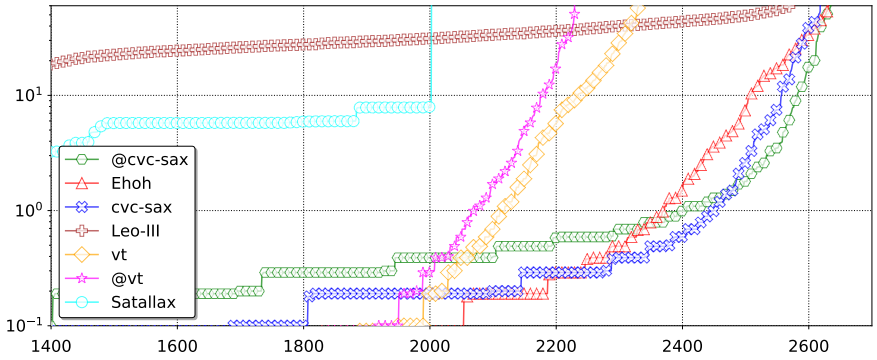


Fig. 2: Execution times in secs on 5,543 benchmarks, from TH0 and JD, supported by all solvers.

$\lambda$s, we see the advantages of these systems only increase. We also observe that both @cvc and cvc, but especially the latter, benefit from -sax as more complex benchmarks are considered in TPTP, showing that the disadvantage of not synthesizing $\lambda$-abstractions can sometimes be offset by well-chosen axioms. Nevertheless, the results on $\lambda o$TH0 show that this axiom alone is far from enough to offset the gap between @cvc and cvc, with cvc giving up more often from lack of instantiations to perform.

Sledghammer-generated problems stem from formalization efforts across different applications. As others note [45, 48], the bottleneck in solving these problems is often scalability and efficient FO reasoning, rather than a refined handling of HO constructs, especially as more facts are considered. Thus, the ability to synthesize $\lambda$-abstractions is not sufficient for scalability as more facts are considered, and Ehoh and the CVC4 extensions eventually surpass the native HO provers. In particular, in the largest set we considered, $\lambda o\text{SH}^{1024}$, both @cvc and cvc have significant advantages. As in $\lambda o$TH0, @cvc also solves more problems than cvc in $\lambda o\text{SH}^{1024}$, which we attribute again to @cvc being able to perform more instantiations than cvc On commonly solved problems, however, cvc is often faster than @cvc, albeit by a small margin: 15% on average.

Both CVC4 configurations dominate $\text{JD}^{512}$ with a significantl margin over Ehoh and Leo-III. Comparing the results between using $\lambda$-lifting or combinators, the former favors cvc and the latter, @cvc. These results, as well as the previously discussed ones, indicate that for unsatisfiable benchmarks the pragmatic extension of CVC4 should not, in its current state, substitute an encoding-based approach but complement it. In fact, a virtual best solver of all the CVC4 configurations, as well as others employing interleaved enumerative instantiation [37], in portfolio, would solve 703 problems in $\text{JD}^{512}_{\text{lift}}$, 702 in $\text{JD}^{512}_{\text{combs}}$, 453 in $\lambda o\text{SH}^{1024}$, and 408 in TH0, the most in these categories, even also considering a virtual best solver of all Ehoh configurations from [48]. The CVC4 portfolio would also solve 482 problems in $\text{JD}^{32}_{\text{lift}}$, and 482 in $\text{JD}^{32}_{\text{combs}}$, doing almost as well as Leo-III, and 1,001 problems in $\lambda o$TH0, The virtual best CVC4 has a success rate 3 percentage points higher than @cvc on Sledghammer benchmarks, as well as overall, which represents a significant improvement when considering the usage of these solvers as backends for interactive theorem provers.

| Solver | Total | TH0 | $o$TH0 | $\lambda o$TH0 | $\text{JD}^{32}_{\text{lift}}$ | $\text{JD}^{32}_{\text{combs}}$ | $\text{JD}^{512}_{\text{lift}}$ | $\text{JD}^{512}_{\text{combs}}$ | $\lambda o\text{SH}^{1024}$ |
|---|---|---|---|---|---|---|---|---|---|
| # | 9032 | 530 | 743 | 1915 | 1253 | 1253 | 1253 | 1253 | 832 |
| *@cvc-fmf-sax* | 224 | 58 | **43** | 80 | 20 | 18 | **1** | **1** | **3** |
| *cvc-fmf* | **482** | **90** | 17 | **205** | **93** | **73** | **1** | **1** | 2 |
| *Satallax* | 186 | 72 | 15 | 98 | 0 | 0 | 0 | 0 | 1 |

Table 2: Conjectures with found countermodels per benchmark set. Best results in **bold**.

Differently from the pragmatic extension in CVC4, which provides more of an alternative to the full applicative encoding, the redesigned veriT is an outright improvement, with vt consistently solving more problems and with better solving times than @vt, especially on harder problems, as seen by the wider separation between them after 10s in Figure 2. Overall, veriT's performance, consistently with it being a lightweight solver, lags behind CVC4 and Ehoh as bigger benchmarks are considered. However, it is respectable compared with Leo-III's and ahead of Satallax's performance, thus validating the effort of redesigning the solver for a more refined handling of higher-order constructs and suggesting that further extensions should be beneficial.

### 5.2 Providing countermodels to HO conjectures

The number of countermodels found by each solver configuration per benchmark set is given in Table 2. We consider the two CVC4 extension, @cvc and cvc, run in in finite-model-finding mode (-fmf) [39]. The builtin HO support in cvc is vastly superior to @cvc when it comes to model finding, as cvc-fmf greatly outperforms @cvc-fmf-sax. We note that @cvc-fmf is only model-sound if combined with -sax. Differently from cvc-fmf, which fails to provide a model as soon as it is faced with quantification over a functional sort, in @cvc-fmf functional sorts are encoded as atomic sorts. Thus it needs the extra axiom to ensure model soundness. For example, @cvc-fmf considers Example 5 satisfiable while @cvc-fmf-sax properly reports it unsatisfiable.

The high number of countermodels in $\text{JD}^{32}$ indicates, not surprisingly, that providing few facts makes several SH goals unprovable. Nevertheless, it is still useful to know where exactly the Sledghammer generation is being "incomplete" (i.e., making originally provable goals unprovable), something that is difficult to determine without effective model finding procedures.

## 6 Concluding remarks

We have presented extensions for SMT solvers to handle HOSMT problems. The pragmatic extension of CVC4, which can be implemented in other state-of-the-art SMT solver with similar level of effort, performs similarly to the standard encoding-based approach despite its limited support for HO instantiation. Moreover, it allows numerous new problems to be solved by CVC4, with a portfolio approach performing very competitively and often ahead of state-of-the-art HO provers. The redesigned veriT on the other hand consistently outperforms its standard encoding-based counterpart, showing it can be the basis for future advancements towards stronger HO automation.

*Acknowledgments* We are grateful to Jasmin Blanchette and Pascal Fontaine for numerous discussions throughout the development of this work, for providing funding for research visits and for suggesting many improvements. We also thank Jasmin for generating several of the benchmarks with which we evaluate our approach; Simon Cruanes and Martin Riener for many fruitful discussions on the intricacies of HOL; Andres Nötzli for help with the table and plot scripts; Mathias Fleury, Hans-Jörg Schurr and Sophie Tourret for suggesting many improvements. This work was partially supported by the National Science Foundation under Award 1656926 and the European Research Council (ERC) under starting grant Matryoshka (713999).

## References

1. Peter B. Andrews. Resolution in type theory. *J. Symb. Log.*, 36(3):414–432, 1971.
2. Leo Bachmair and Harald Ganzinger. Rewrite-based equational theorem proving with selection and simplification. *Journal of Logic and Computation*, 4(3):217–247, 1994.
3. Haniel Barbosa, Jasmin Christian Blanchette, Simon Cruanes, Daniel El Ouraoui, and Pascal Fontaine. Language and proofs for higher-order SMT (work in progress). In Catherine Dubois and Bruno Woltzenlogel Paleo, editors, *PXTP 2017*, volume 262 of *EPTCS*, pages 15–22, 2017.
4. Haniel Barbosa, Pascal Fontaine, and Andrew Reynolds. Congruence closure with free variables. In Axel Legay and Tiziana Margaria, editors, *TACAS 2017*, volume 10206 of *LNCS*, pages 214–230. Springer, 2017.
5. Haniel Barbosa, Andrew Reynolds, Daniel El Ouraoui, Cesare Tinelli, and Clark Barrett. Extending SMT solvers to higher-order logic. Technical report, The University of Iowa, May 2019.
6. Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *CAV 2011*, pages 171–177. Springer, 2011.
7. Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The SMT-LIB standard: version 2.6. Technical report, Department of Computer Science, The University of Iowa, 2017.
8. Clark Barrett, Roberto Sebastiani, Sanjit Seshia, and Cesare Tinelli. Satisfiability modulo theories. In Armin Biere, Marijn J. H. Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, volume 185 of *FAIA*, chapter 26, pages 825–885. IOS Press, 2009.
9. Alexander Bentkamp, Jasmin Christian Blanchette, Simon Cruanes, and Uwe Waldmann. Superposition for lambda-free higher-order logic. In Didier Galmiche, Stephan Schulz, and Roberto Sebastiani, editors, *IJCAR 2018*, volume 10900 of *LNCS*, pages 28–46. Springer, 2018.
10. Christoph Benzmüller and Dale Miller. Automation of higher-order logic. In Jörg H. Siekmann, editor, *Computational Logic*, volume 9 of *Handbook of the History of Logic*, pages 215–254. Elsevier, 2014.
11. Christoph Benzmüller, Nik Sultana, Lawrence C. Paulson, and Frank Theiss. The higher-order prover LEO-II. *J. Autom. Reason.*, 55:389–404, 2015.
12. Ahmed Bhayat and Giles Reger. Set of support for higher-order reasoning. In Boris Konev, Josef Urban, and Philipp Rümmer, editors, *PAAR-2018*, volume 2162 of *CEUR Workshop Proceedings*, pages 2–16. CEUR-WS.org, 2018.
13. Jasmin Christian Blanchette. *Automatic proofs and refutations for higher-order logic*. PhD thesis, Technical University Munich, 2012.
14. Jasmin Christian Blanchette, Cezary Kaliszyk, Lawrence C. Paulson, and Josef Urban. Hammering towards QED. *J. Formaliz. Reas.*, 9(1):101–148, 2016.

15. Sascha Böhme and Tobias Nipkow. Sledgehammer: judgement day. In Jürgen Giesl and Reiner Hähnle, editors, *IJCAR 2010*, volume 6173 of *LNCS*, pages 107–121. Springer, 2010.
16. Thomas Bouton, Diego Caminha B. de Oliveira, David Déharbe, and Pascal Fontaine. veriT: an open, trustable and efficient SMT-solver. In Renate A. Schmidt, editor, *CADE–22*, volume 5663 of *LNCS*, pages 151–156. Springer, 2009.
17. Chad E. Brown. Satallax: an automatic higher-order prover. In Bernhard Gramlich, Dale Miller, and Uli Sattler, editors, *IJCAR 2012*, volume 7364 of *LNCS*, pages 111–117. Springer, 2012.
18. Simon Cruanes. Superposition with structural induction. In Clare Dixon and Marcelo Finger, editors, *FroCoS 2017*, volume 10483 of *LNCS*, pages 172–188. Springer, 2017.
19. Łukasz Czajka and Cezary Kaliszyk. Hammer for Coq: automation for dependent type theory, 2018.
20. Leonardo de Moura and Nikolaj Bjørner. Efficient e-matching for SMT solvers. In Frank Pfenning, editor, *CADE–21*, volume 4603 of *LNCS*, pages 183–198. Springer, 2007.
21. Leonardo de Moura and Nikolaj Bjørner. Generalized, efficient array decision procedures. In *FMCAD 2009*, pages 45–52. IEEE, 2009.
22. David Detlefs, Greg Nelson, and James B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52:365–473, 2005.
23. Gilles Dowek. Higher-order unification and matching. In John Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, volume II, pages 1009–1062. Elsevier and MIT Press, 2001.
24. Peter J. Downey, Ravi Sethi, and Robert Endre Tarjan. Variations on the common subexpression problem. *J. ACM*, 27:758–771, 1980.
25. Michael Färber and Chad E. Brown. Internal guidance for Satallax. In Nicola Olivetti and Ashish Tiwari, editors, *IJCAR 2016*, volume 9706 of *LNCS*, pages 349–361. Springer, 2016.
26. Yeting Ge and Leonardo de Moura. Complete instantiation for quantified formulas in satisfiabiliby modulo theories. In Ahmed Bouajjani and Oded Maler, editors, *CAV 2009*, volume 5643 of *LNCS*, pages 306–320. Springer, 2009.
27. Leon Henkin. Completeness in the theory of types. *J. Symb. Log.*, 15(2):81–91, 1950.
28. R. J. M. Hughes. Super combinators: a new implementation method for applicative languages. In *Symposium on LISP and Functional Programming*, pages 1–10, 1982.
29. Michael Kohlhase. Higher-order tableaux. In Peter Baumgartner, Reiner Hähnle, and Joachim Posegga, editors, *TABLEAUX '95*, volume 918 of *LNCS*, pages 294–309. Springer, 1995.
30. Jia Meng and Lawrence C. Paulson. Translating higher-order clauses to first-order clauses. *J. Autom. Reas.*, 40(1):35–60, 2008.
31. Greg Nelson and Derek C. Oppen. Fast decision procedures based on congruence closure. *J. ACM*, 27:356–364, 1980.
32. Robert Nieuwenhuis and Albert Oliveras. Fast Congruence Closure and Extensions. *Information and Computation, IC*, 2005(4):557–580, 2007.
33. Robert Nieuwenhuis and Albert Rubio. Paramodulation-based theorem proving. In Alan Robinson and Andrei Voronkov, editors, *Handbook of automated reasoning*, volume 1, pages 371–443. Elsevier Science, 2001.
34. Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: a proof Assistant for higher-order logic*, volume 2283 of *LNCS*. Springer, 2002.
35. Kohei Noshita. Translation of turner combinators in O(n log n) space. *IPL*, 20:71 – 74, 1985.
36. Lawrence C. Paulson and Jasmin Christian Blanchette. Three years of experience with Sledgehammer, a practical link between automatic and interactive theorem provers. In Geoff Sutcliffe, Stephan Schulz, and Eugenia Ternovska, editors, *IWIL-2010*, volume 2 of *EPiC*, pages 1–11. EasyChair, 2012.

37. Andrew Reynolds, Haniel Barbosa, and Pascal Fontaine. Revisiting enumerative instantiation. In Dirk Beyer and Marieke Huisman, editors, *TACAS 2018*, volume 10806 of *LNCS*, pages 112–131. Springer, 2018.
38. Andrew Reynolds, Cesare Tinelli, and Leonardo de Moura. Finding conflicting instances of quantified formulas in SMT. In *FMCAD 2014*, pages 195–202. IEEE, 2014.
39. Andrew Reynolds, Cesare Tinelli, Amit Goel, and Sava Krstić. Finite model finding in SMT. In Natasha Sharygina and Helmut Veith, editors, *CAV 2013*, volume 8044 of *LNCS*, pages 640–655. Springer, 2013.
40. Andrew Reynolds, Cesare Tinelli, Amit Goel, Sava Krstić, Morgan Deters, and Clark Barrett. Quantifier instantiation techniques for finite model finding in SMT. In Maria Paola Bonacina, editor, *CADE–24*, volume 7898 of *LNCS*, pages 377–391. Springer, 2013.
41. John Alan Robinson. Mechanizing higher order logic. *Machine Intelligence*, 4:151–170, 1969.
42. Stephan Schulz. E - a brainiac theorem prover. *AI Commun.*, 15:111–126, 2002.
43. Alexander Steen and Christoph Benzmüller. The higher-order prover Leo-III. In Didier Galmiche, Stephan Schulz, and Roberto Sebastiani, editors, *IJCAR 2018*, volume 10900 of *LNCS*, pages 108–116. Springer, 2018.
44. Aaron Stump, Clark W. Barrett, David L. Dill, and Jeremy R. Levitt. A decision procedure for an extensional theory of arrays. In *LICS 2001*, pages 29–37. IEEE Computer Society, 2001.
45. Nik Sultana, Jasmin Christian Blanchette, and Lawrence C. Paulson. LEO-II and Satallax on the Sledgehammer test bench. *J. Applied Logic*, 11:91–102, 2013.
46. Geoff Sutcliffe. The TPTP problem library and associated infrastructure. *J. Autom. Reason.*, 43:337–362, 2009.
47. Geoff Sutcliffe. The CADE ATP system competition - CASC. *AI Magazine*, 37:99–101, 2016.
48. Petar Vukmirović, Jasmin Christian Blanchette, Simon Cruanes, and Stephan Schulz. Extending a brainiac prover to lambda-free higher-order logic. In Tomas Vojnar and Lijun Zhang, editors, *TACAS 2019*, LNCS. Springer, 2019.