Model Finding for Recursive Functions in SMT*

Andrew Reynolds[1], Jasmin Christian Blanchette[2,3], and Cesare Tinelli[4]

[1] École Polytechnique Fédérale de Lausanne (EPFL), Switzerland
[2] Inria Nancy & LORIA, Villers-lès-Nancy, France
[3] Max-Planck-Institut für Informatik, Saarbrücken, Germany
[4] Department of Computer Science, The University of Iowa, USA

Abstract

SMT solvers have recently been extended with techniques for finding models in presence of universally quantified formulas in some restricted fragments. This paper introduces a translation which reduces axioms specifying a large class of recursive functions, including well-founded (terminating) functions, to universally quantified formulas for which these techniques are applicable. An empirical evaluation confirms that the approach improves the performance of existing solvers on benchmarks from two sources. The translation is implemented as a preprocessor in the solver CVC4.

1 Introduction

Many solvers based on SMT (satisfiability modulo theories) can reason about quantified formulas using incomplete instantiation-based methods [8, 20]. These methods work well in the context of proving (i..e, showing unsatisfiability), but they are of little help for finding models (i.e., showing satisfiability). Often, a single universal quantifier in one of the axioms of a problem is enough to prevent the discovery of models.

In the past few years, techniques have been developed to find models for quantified formulas in SMT. Ge and de Moura [11] introduced a complete instantiation-based procedure for formulas in the essentially uninterpreted fragment. This fragment is limited to universally quantified formulas where all variables occur as direct subterms of uninterpreted functions—e.g., $\forall x.\ \mathsf{f}(x) \approx \mathsf{g}(x) + 5$. Other syntactic criteria extend this fragment slightly, including cases when variables occur as arguments of arithmetic predicates. Subsequently, Reynolds et al. [21, 22] introduced techniques for finding finite models for quantified formulas over uninterpreted types and types having a fixed finite interpretation. These techniques can find a model for a formula such as $\forall x, y : \tau.\ x \approx y \vee \neg\, \mathsf{f}(x) \approx \mathsf{f}(y)$, where τ is an uninterpreted type.

Unfortunately, none of these fragments can accommodate the vast majority of quantified formulas that correspond to recursive function definitions: The essentially uninterpreted fragment does not allow the argument of a recursive function to be used inside a complex term on the right-hand side, whereas the finite model finding techniques are not applicable for functions over infinite domains such as the integers or algebraic datatypes. A simple example where both

*This research is partially supported by the Inria technological development action "Contre-exemples utilisables par Isabelle et Coq" (CUIC).

approaches fail is $\forall x : \mathsf{Int}.\ \mathsf{ite}\big(x \leq 0,\ \mathsf{p}(x) \approx 1,\ \mathsf{p}(x) \approx 2 * \mathsf{p}(x-1)\big)$. This state of affairs is unsatisfactory, given the frequency of recursive definitions in practice and the impending addition of a dedicated command for introducing them, `define-fun-rec`, to the SMT-LIB standard [2].

We present a method for translating formulas involving recursive function definitions into formulas where finite model finding techniques can be applied. The recursive functions must meet a semantic criterion to be admissible (Section 2). This criterion is met by well-founded (terminating) recursive function definitions.

We define a translation for a class of formulas involving admissible recursive function definitions (Section 3). The main insight is that a recursive definition $\forall x : \tau.\ \mathsf{f}(x) \approx t$ can be translated to $\forall a : \alpha_\tau.\ \mathsf{f}(\gamma_\mathsf{f}(a)) \approx t[\gamma_\mathsf{f}(a)/x]$, where α_τ is an uninterpreted abstract type and γ_f converts the abstract type to the concrete type. Additional constraints ensure that the abstract values that are relevant to the formula's satisfiability exist. The translation preserves satisfiability and unsatisfiability, and makes finite model finding possible for problems in this class. Detailed proofs of correctness are included in a technical report [18].

Our empirical evaluation on benchmarks from the IsaPlanner proof planner [12] and the Leon verifier [4] provides evidence that this translation improves the effectiveness of the SMT solvers CVC4 and Z3 for finding countermodels to verification conditions (Section 4). The approach is implemented as a preprocessor in CVC4 (Section 5).

2 Preliminaries

Our setting is a monomorphic (or many-sorted) first-order logic like the one defined by SMT-LIB [2]. A *signature* Σ consists of a set Σ^{ty} of first-order types (or sorts) and a set Σ^{f} of function symbols over these types. We assume that signatures always contain a Boolean type Bool and constants $\top, \bot : \mathsf{Bool}$ for truth and falsity, an infix equality predicate $\approx : \tau \times \tau \to \mathsf{Bool}$ for each $\tau \in \Sigma^{\mathrm{ty}}$, standard Boolean connectives ($\neg$, $\wedge$, $\vee$, etc.), and an if–then–else function symbol $\mathsf{ite} : \mathsf{Bool} \times \tau \times \tau \to \tau$ for each $\tau \in \Sigma^{\mathrm{ty}}$. For each $\tau \in \Sigma^{\mathrm{ty}}$, we fix an infinite set Σ^{v}_τ of *variables of type* τ and define Σ^{v} as $\bigcup_{\tau \in \Sigma^{\mathrm{ty}}} \Sigma^{\mathrm{v}}_\tau$. Σ-terms are built as usual over functions symbols in Σ and variables in Σ^{v}. Formulas are terms of type Bool. We write t^τ to denote terms of type τ and $\mathcal{T}(t)$ to denote the set of subterms in t. Given a term u, we write $u[\bar{t}/\bar{x}]$ to denote the result of replacing all occurrences of $\bar{x}$ with $\bar{t}$ in u.

A Σ-*interpretation* I maps each type $\tau \in \Sigma^{\mathrm{ty}}$ to a nonempty set τ^I, the *domain* of τ in I, each function symbol $\mathsf{f} : \tau_1 \times \cdots \times \tau_n \to \tau$ in Σ^{f} to a total function $\mathsf{f}^I : \tau_1^I \times \cdots \times \tau_n^I \to \tau^I$, and each variable $x : \tau$ of Σ^{v} to an element of τ^I. A *theory* is a pair $T = (\Sigma, \mathscr{I})$ where Σ is a signature and $\mathscr{I}$ is a class of Σ-interpretations, the *models* of T, closed under variable reassignment (i.e., for every $I \in \mathscr{I}$, every Σ-interpretation that differs from I only on the variables of Σ^{v} is also in $\mathscr{I}$). A Σ-formula φ is T-*satisfiable* if it is satisfied by some interpretation in $\mathscr{I}$. A formula φ T-*entails* ψ, written $\varphi \models_T \psi$, if all interpretations in $\mathscr{I}$ that satisfy φ also satisfy ψ. Two formulas φ and ψ are T-*equivalent* if each T-entails the other. If $T_1 = (\Sigma_1, \mathscr{I}_1)$ is a theory and Σ_2 is a signature with $\Sigma_1^{\mathrm{f}} \cap \Sigma_2^{\mathrm{f}} = \emptyset$, the *extension of* T_1 *to* Σ_2 is the theory $T = (\Sigma, \mathscr{I})$ where $\Sigma^{\mathrm{f}} = \Sigma_1^{\mathrm{f}} \cup \Sigma_2^{\mathrm{f}}$, $\Sigma^{\mathrm{ty}} = \Sigma_1^{\mathrm{ty}} \cup \Sigma_2^{\mathrm{ty}}$, and $\mathscr{I}$ is the set of all Σ-interpretations I whose Σ_1-reduct is a

model of T_1. We refer to the symbols of Σ_2 that are not in Σ_1 as *uninterpreted*. For the rest of the paper, we fix a theory $T = (\Sigma, \mathscr{I})$ with uninterpreted symbols constructed as above.

Unconventionally, we consider *annotated quantified formulas* of the form $\forall_{\mathsf{f}}\, \overline{x}.\, \varphi$, where $\mathsf{f} \in \Sigma^{\mathrm{f}}$ is uninterpreted. Their semantics is the same as for standard quantified formulas $\forall \overline{x}.\, \varphi$. Given $\mathsf{f} : \tau_1 \times \cdots \times \tau_n \to \tau$, an annotated quantified formula $\forall_{\mathsf{f}}\, \overline{x}.\, \varphi$ is a *function definition* (*for* f) if $\overline{x}$ is a tuple of variables $x_1 : \tau_1, \ldots, x_n : \tau_n$ and φ is a quantifier-free formula T-equivalent to $\mathsf{f}(\overline{x}) \approx t$ for some term t of type τ. We write $\exists \overline{x}.\, \varphi$ as an abbreviation for $\neg\, \forall \overline{x}.\, \neg\, \varphi$.

Definition 1. A formula φ is in *definitional form with respect to* $\{\mathsf{f}_1, \ldots, \mathsf{f}_n\} \subseteq \Sigma^{\mathrm{f}}$ if it is of the form $(\forall_{\mathsf{f}_1}\, \overline{x}_1.\, \varphi_1) \wedge \cdots \wedge (\forall_{\mathsf{f}_n}\, \overline{x}_n.\, \varphi_n) \wedge \psi$, where $\mathsf{f}_1, \ldots, \mathsf{f}_n$ are distinct, $\forall_{\mathsf{f}_i}\, \overline{x}_i.\, \varphi_i$ is a function definition for $i = 1, \ldots, n$, and ψ contains no function definitions. We call ψ the *conjecture* of φ.

In the signature Σ, we distinguish a subset $\Sigma^{\mathrm{dfn}} \subseteq \Sigma^{\mathrm{f}}$ of *defined* uninterpreted function symbols. We consider Σ-formulas that are in definitional form with respect to Σ^{dfn}.

Definition 2. Given a set of function definitions $\Delta = \{\forall_{\mathsf{f}_1}\, \overline{x}.\, \varphi_1, \ldots, \forall_{\mathsf{f}_n}\, \overline{x}.\, \varphi_n\}$, a ground formula ψ is *closed under function expansion with respect to* Δ if $\psi \vDash_T \bigwedge_{i=1}^{n} \{\varphi_i[\overline{t}/\overline{x}] \mid \mathsf{f}_i(\overline{t}) \in \mathcal{T}(\psi)\}$. The set Δ is *admissible* if for every T-satisfiable formula ψ closed under function expansion with respect to Δ, the formula $\psi \wedge \bigwedge \Delta$ is also T-satisfiable.

Admissibility is a semantic criterion that must be satisfied for each function definition before applying the translation described in Section 3. It is interesting to connect it to the standard notion of *well-founded* function definitions, often called *terminating* definitions in a slight abuse of terminology. In such definitions, all recursive calls are decreasing with respect to a well-founded relation, which must be supplied by the user or inferred automatically using a termination prover. This ensures that the function is uniquely defined at all points.

First-order logic has no built-in notion of computation or termination. To ensure that a function specification is well-founded, it is sufficient to require that the function would terminate when seen as a functional program, under *some* evaluation order. For example, the definition $\forall x : \mathsf{Int}.\ \mathsf{ite}\big(x \leq 0,\ \mathsf{p}(x) \approx 1,\ \mathsf{p}(x) \approx 2 * \mathsf{p}(x-1)\big)$, where T is integer arithmetic extended with the uninterpreted symbol $\mathsf{p} : \mathsf{Int} \to \mathsf{Int}$, can be shown well-founded under a strategy that evaluates the condition of an ite before evaluating the relevant branch, ignoring the other branch. Krauss developed these ideas in the more general context of higher-order logic [14, Section 2].

Theorem 1. *If Δ is a set of well-founded function definitions for Σ^{dfn}, then it is admissible.*

An example of an inadmissible set is $\{\forall_{\mathsf{f}}\, x : \mathsf{Int}.\ \mathsf{f}(x) \approx \mathsf{f}(x) + 1\}$, where T is integer arithmetic extended to a set of uninterpreted symbols $\{\mathsf{f}, \mathsf{g} : \mathsf{Int} \to \mathsf{Int}, \ldots\}$. The reason is that the formula $\top$ is (trivially) closed under function expansion with respect to this set, and there is no model of T satisfying f's definition. A more subtle example is $\{\forall_{\mathsf{f}}\, x : \mathsf{Int}.\ \mathsf{f}(x) \approx \mathsf{f}(x)$, $\forall_{\mathsf{g}}\, x : \mathsf{Int}.\ \mathsf{g}(x) \approx \mathsf{g}(x) + \mathsf{f}(x)\}$. While this set has a model where f and g are interpreted as the constant function 0, it is not admissible since $\mathsf{f}(0) \approx 1$ is closed under function expansion and yet there exists no interpretation satisfying both $\mathsf{f}(0) \approx 1$ and g's definition.

$$
\begin{aligned}
&\mathcal{A}_0(t^\tau, p) = \\
&\quad \text{if } \tau = \mathsf{Bool} \text{ and } t = \mathsf{b}(t_1, \ldots, t_n) \text{ then} \\
&\qquad \text{let } (t'_i, \chi_i) = \mathcal{A}_0(t_i, \mathsf{pol}(\mathsf{b}, i, p)) \text{ for } i = 1, \ldots, n \text{ in} \\
&\qquad \text{let } \chi = \chi_1 \wedge \cdots \wedge \chi_n \text{ in} \\
&\qquad \text{if } p = \mathsf{pos} \text{ then } \big(\mathsf{b}(t'_1, \ldots, t'_n)) \wedge \chi, \top\big) \\
&\qquad \text{else if } p = \mathsf{neg} \text{ then } \big(\mathsf{b}(t'_1, \ldots, t'_n) \vee \neg \chi, \top\big) \\
&\qquad \text{else } \big(\mathsf{b}(t'_1, \ldots, t'_n), \chi\big) \\
&\quad \text{else if } t = \forall_{\mathsf{f}} \overline{x}.\ u \text{ then} \\
&\qquad \text{let } (u', \chi) = \mathcal{A}_0(u, p) \text{ in } \big(\forall a : \alpha_{\mathsf{f}}.\ u'[\overline{\gamma}_{\mathsf{f}}(a)/\overline{x}], \top\big) \\
&\quad \text{else if } t = \forall \overline{x}.\ u \text{ then} \\
&\qquad \text{let } (u', \chi) = \mathcal{A}_0(u, p) \text{ in } \big(\forall \overline{x}.\ u', \forall \overline{x}.\ \chi\big) \\
&\quad \text{else} \\
&\qquad \big(t, \bigwedge \{\exists a : \alpha_{\mathsf{f}}.\ \overline{\gamma}_{\mathsf{f}}(a) \approx \overline{s} \mid \mathsf{f}(\overline{s}) \in \mathcal{T}(t), \mathsf{f} \in \Sigma^{\mathrm{dfn}}\}\big) \\
&\mathcal{A}(\varphi) = \text{let } (\varphi', \chi) = \mathcal{A}_0(\varphi, \mathsf{pos}) \text{ in } \varphi'
\end{aligned}
$$

Figure 1: Definition of translation $\mathcal{A}$

3 The Translation

For the rest of the section, let φ be a Σ-formula in definitional form with respect to Σ^{dfn} whose definitions are admissible. We present a method that constructs an extended signature $\mathcal{E}(\Sigma)$ and an $\mathcal{E}(\Sigma)$-formula φ' such that φ' is T-satisfiable if and only if φ is T-satisfiable—i.e., φ and φ' are *equisatisfiable* (*in* T). The idea behind this translation is to use an uninterpreted type to abstract the set of *relevant* tuples for each defined function f and restrict the quantification of f's definition to a variable of this type. Informally, the relevant tuples $\bar{t}$ of a function f are the ones for which the interpretation of $\mathsf{f}(\bar{t})$ is relevant to the satisfiability of φ. More precisely, for each $\mathsf{f} : \tau_1 \times \cdots \times \tau_n \to \tau \in \Sigma^{\mathrm{dfn}}$, the extended signature $\mathcal{E}(\Sigma)$ contains an uninterpreted *abstract type* α_{f} and n uninterpreted *concretization functions* $\gamma_{\mathsf{f},1} : \alpha_{\mathsf{f}} \to \tau_1, \ldots, \gamma_{\mathsf{f},n} : \alpha_{\mathsf{f}} \to \tau_n$.

The translation $\mathcal{A}$ defined in Figure 1 translates the Σ-formula φ into the $\mathcal{E}(\Sigma)$-formula φ'. It relies on the auxiliary function $\mathcal{A}_0$, which takes two arguments: the term t to translate and a polarity p for t, which is either pos, neg, or none. $\mathcal{A}_0$ returns a pair (t', χ), where t' is a term of the same type as t and χ is an $\mathcal{E}(\Sigma)$-formula.

The translation alters the formula φ in two ways. First, it restricts the quantification on function definitions for f to the corresponding uninterpreted type α_{f}, inserting applications of the concretization functions $\gamma_{\mathsf{f},i}$ as needed. Second, it augments φ with additional constraints of the form $\exists a : \alpha_{\mathsf{f}}.\ \overline{\gamma}_{\mathsf{f}}(a) \approx \overline{s}$, where $\overline{\gamma}_{\mathsf{f}}(a) \approx \overline{s}$ abbreviates the formula $\bigwedge_{i=1}^{n} \gamma_{\mathsf{f},i}(a) \approx s_i$ with $\overline{s} = (s_1, \ldots, s_n)$. These existential constraints ensure that the restricted definition for f covers all relevant tuples of terms, namely those occurring in applications of f that are relevant to the satisfiability of φ.

If t is an application of a predicate symbol b, including the operators $\neg$, $\wedge$, $\vee$, $\approx$, and ite, $\mathcal{A}_0$ calls itself recursively on the arguments t_i and polarity $\mathsf{pol}(\mathsf{b}, i, p)$, with pol defined as

$$\mathsf{pol}(\mathsf{b}, i, p) = \begin{cases} p & \text{if either } \mathsf{b} \in \{\wedge, \vee\} \text{ or } \mathsf{b} = \mathsf{ite} \text{ and } i \in \{2, 3\} \\ -p & \text{if } \mathsf{b} = \neg \\ \mathsf{none} & \text{otherwise} \end{cases}$$

where $-p$ is neg if p is pos, pos if p is neg, and none if p is none. The term t is then reconstructed as $\mathsf{b}(t'_1, \ldots, t'_n)$ where each t'_i is the result of the recursive call with argument t_i. If the polarity p associated with t is pos, $\mathcal{A}_0$ conjunctively adds to $\mathsf{b}(t'_1, \ldots, t'_n)$ the constraint χ derived from the subterms. Dually, if p is neg, it adds a disjunction with the negated constraint, to achieve the same overall effect. It p is none, the constraint χ is returned to the caller. If t is a function definition, $\mathcal{A}_0$ constructs a quantified formula over a single variable a of type α_{f} and replaces all occurrences of $\overline{x}$ in the body of that formula with $\overline{\gamma}_{\mathsf{f}}(a)$. (Since function definitions are top-level conjuncts, χ must be $\top$ and can be ignored.) If t is an unannotated quantified formula, $\mathcal{A}_0$ calls itself on the body with the same polarity. Otherwise, t is either an application of an uninterpreted predicate symbol or a term of a type other than Bool. Then, the returned constraint is a conjunction of formulas of the form $\exists a : \alpha_{\mathsf{f}}.\ \overline{\gamma}_{\mathsf{f}}(a) \approx \overline{s}$ for each subterm $\mathsf{f}(\overline{s})$ of t such that $\mathsf{f} \in \Sigma^{\mathrm{dfn}}$. Such constraints, when asserted positively, ensure that some element in the abstract domain α_{f} is the preimage of the argument tuple $\overline{s}$.

Example 1. Let T be linear arithmetic with the uninterpreted symbols $\{\mathsf{c} : \mathsf{Int}, \mathsf{s} : \mathsf{Int} \to \mathsf{Int}\}$. Let φ be the Σ-formula

$$\forall_{\mathsf{s}} x : \mathsf{Int}.\ \mathsf{ite}\big(x \leq 0,\ \mathsf{s}(x) \approx 0,\ \mathsf{s}(x) \approx x + \mathsf{s}(x-1)\big) \wedge \mathsf{s}(\mathsf{c}) > 100 \tag{1}$$

The definition of s specifies that it returns the sum of all positive integers up to x. The formula φ is in definitional form with respect to Σ^{dfn} and states that the sum of all positive numbers up to some constant c is greater than 100. It is satisfiable with a model that interprets c as 14 or more. Due to the universal quantifier, current SMT techniques are unable to find a model for φ. The signature $\mathcal{E}(\Sigma)$ extends Σ with the type α_{s} and the uninterpreted function symbol $\gamma_{\mathsf{s}} : \alpha_{\mathsf{s}} \to \mathsf{Int}$. The result of $\mathcal{A}(\varphi)$, after simplification, is the $\mathcal{E}(\Sigma)$-formula

$$\begin{aligned} &\big(\forall a : \alpha_{\mathsf{s}}.\ \mathsf{ite}\big(\gamma_{\mathsf{s}}(a) \leq 0,\ \mathsf{s}(\gamma_{\mathsf{s}}(a)) \approx 0, \\ &\qquad\qquad \mathsf{s}(\gamma_{\mathsf{s}}(a)) \approx \gamma_{\mathsf{s}}(a) + \mathsf{s}(\gamma_{\mathsf{s}}(a) - 1) \wedge \exists b : \alpha_{\mathsf{s}}.\ \gamma_{\mathsf{s}}(b) \approx \gamma_{\mathsf{s}}(a) - 1\big)\big) \\ &\wedge \mathsf{s}(\mathsf{c}) > 100 \wedge \exists a : \alpha_{\mathsf{s}}.\ \gamma_{\mathsf{s}}(a) \approx \mathsf{c} \end{aligned} \tag{2}$$

The universal quantifier in formula (2) ranges over an uninterpreted type α_{s}, making it amenable to the finite model finding techniques by Reynolds et al. [21,22], implemented in CVC4, which search for a finite interpretation for α_{s}. Furthermore, since all occurrences of the quantified variable a are beneath applications of the uninterpreted function γ_{s}, the formula is in the essentially uninterpreted fragment, for which Ge and de Moura [11] provide a complete instantiation procedure, implemented in Z3. As expected, CVC4 and Z3 run indefinitely on formula (1), whereas they produce a model for (2) within 100 milliseconds. ■

Note that the translation $\mathcal{A}$ results in formulas whose models (i.e., satisfying interpretations) are generally different from those of φ. One model I for formula (2) in the above example interprets α_{s} as a finite set $\{u_0,\ldots,u_{14}\}$, γ_{s} as a finite map $u_i \mapsto i$ for $i = 0,\ldots,14$, c as 14, and s as the almost constant function $\lambda x : \mathsf{Int}.\ \mathsf{ite}(x \approx 0,\ 0,\ \mathsf{ite}(x \approx 1,\ 1,\ \mathsf{ite}(x \approx 2,\ 3,\ \mathsf{ite}(\ldots,\ \mathsf{ite}(x \approx 13, 91, 105)\ldots))))$. In other words, s is interpreted as a function mapping x to the sum of all positive integers up to x when $0 \leq x \leq 13$, and 105 otherwise. The Σ-reduct of I is not a model of the original formula (1), since I interprets $\mathsf{s}(n)$ as 105 when $n < 0$ or $n > 14$.

However, under the assumption that the function definitions in Σ^{dfn} are admissible, $\mathcal{A}(\varphi)$ is equisatisfiable with φ for any input φ. Moreover, the models of $\mathcal{A}(\varphi)$ contain pertinent information about the models of φ. For example, the model I for formula (2) given above interprets c as 14 and $\mathsf{s}(n)$ as $\sum_{i=1}^{n} i$ for $0 \leq n \leq 14$, and there exists a model of formula (1) that also interprets c and $\mathsf{s}(n)$ in the same way (for $0 \leq n \leq 14$). In general, for every model of $\mathcal{A}(\varphi)$, there exists a model of φ that coincides with it on its interpretation of all function symbols in $\Sigma^{\mathrm{f}} - \Sigma^{\mathrm{dfn}}$. Furthermore, the model of $\mathcal{A}(\varphi)$ will also give correct information for the defined functions at all points belonging to the domains of the corresponding abstract types. This can sometimes help users debug their specifications.

Theorem 2. *If φ is a Σ-formula in definitional form with respect to Σ^{dfn} and the set of function definitions Δ corresponding to Σ^{dfn} is admissible, then φ and $\mathcal{A}(\varphi)$ are equisatisfiable in T.*

We give an intuition of the above theorem in the context of an example. For a set of ground Σ-literals L, let $\mathrm{X}(L)$ be the set $\{\exists a : \alpha_{\mathsf{f}}.\ \overline{\gamma}_{\mathsf{f}}(a) \approx \overline{s} \mid \mathsf{f}(\overline{s}) \in \mathcal{T}(L),\ \mathsf{f} \in \Sigma^{\mathrm{dfn}}\}$.

Example 2. Let us revisit the formulas in Example 1. If the original formula (1) is T-satisfiable, the translated formula (2) is clearly also T-satisfiable since α_{s} can be interpreted as the integers and γ_{s} as the identity function. Conversely, we claim that (2) is T-satisfiable only if (1) is T-satisfiable, noting that the set $\{\forall_{\mathsf{s}} x.\ \varphi_{\mathsf{s}}\}$ is admissible, where φ_{s} is the formula $\mathsf{ite}\big(x \leq 0,\ \mathsf{s}(x) \approx 0,\ \mathsf{s}(x) \approx x + \mathsf{s}(x-1)\big)$. Clearly, any interpretation I satisfying formula (2) satisfies $L_0 \cup \mathrm{X}(L_0)$, where $L_0 = \{\mathsf{s}(\mathsf{c}) > 100\}$ and $\mathrm{X}(L_0)$ consists of the single constraint $\exists a : \alpha_{\mathsf{s}}.\ \gamma_{\mathsf{s}}(a) \approx \mathsf{c}$. Since I also satisfies both the translated function definition for s (the first conjunct of (2)) and $\mathrm{X}(L_0)$, it must also satisfy

$$\mathsf{ite}\big(\mathsf{c} \leq 0,\ \mathsf{s}(\mathsf{c}) \approx 0,\ \mathsf{s}(\mathsf{c}) \approx \mathsf{c} + \mathsf{s}(\mathsf{c}-1) \wedge \exists b : \alpha_{\mathsf{s}}.\ \gamma_{\mathsf{s}}(b) \approx \mathsf{c} - 1\big)$$

The existential constraint in the above formula ensures that whenever I satisfies the set $L_1 = L_0 \cup \{\neg\, \mathsf{c} \leq 0,\ \mathsf{s}(\mathsf{c}) \approx \mathsf{c} + \mathsf{s}(\mathsf{c}-1)\}$, I satisfies $\mathrm{X}(L_1)$ as well. Hence, by repeated application of this reasoning, it follows that a model of formula (2) that interprets c as n must also satisfy ψ:

$$\mathsf{s}(\mathsf{c}) > 100 \wedge \bigwedge_{i=0}^{n-1}\big(\neg\,(\mathsf{c} - i \leq 0) \wedge \mathsf{s}(\mathsf{c}-i) \approx \mathsf{c} - i + \mathsf{s}(\mathsf{c}-i-1)\big) \wedge \mathsf{c} - n \leq 0 \wedge \mathsf{s}(\mathsf{c}-n) \approx 0$$

This formula is closed under function expansion since it entails $\varphi_{\mathsf{s}}[(\mathsf{c}-i)/x]$ for $i = 0,\ldots,n$, and it contains only s applications corresponding to $\mathsf{s}(\mathsf{c}-i)$ for $i = 0,\ldots,n$. Since $\{\forall_{\mathsf{s}} x.\ \varphi_{\mathsf{s}}\}$ is admissible, there exists a Σ-interpretation satisfying $\psi \wedge \forall_{\mathsf{s}} x.\ \varphi_{\mathsf{s}}$, which entails formula (1). ■

4 Evaluation

In this section, we evaluate both the overall impact of the translation introduced in Section 3 and the performance of individual SMT techniques. We gathered 245 benchmarks from two sources, which we will refer to as Isa and Leon. The first source consists of the 79 benchmarks from the IsaPlanner suite [12] that do not contain higher-order functions. These benchmarks have been used recently as challenge problems for a variety of inductive theorem provers. They heavily involve recursive functions and are limited to a theory of algebraic datatypes with a signature that contains uninterpreted function symbols over these datatypes. The second source consists of 166 benchmarks from the Leon repository,[1] which were constructed from verification conditions about simple Scala programs. These benchmarks also heavily involve recursively defined functions over algebraic datatypes, but cover a wide variety of additional theories, including bit vectors, arrays, and both linear and nonlinear arithmetic. All benchmarks are in definitional form with respect to a set of well-founded functions. Nearly all of them are unsatisfiable, except for a handful in the Leon set.

For each of these 245 benchmarks, we considered up to three randomly selected mutated forms of its conjecture ψ. In particular, we considered unique conjectures that are obtained as a result of swapping a subterm of ψ at one position with another of the same type at another position. Note that benchmarks created in this way have a high likelihood of having small, easy-to-find countermodels. In total, we considered 213 mutated forms of conjectures from Isa and 427 mutated forms of conjectures from Leon. We will call these sets Isa-Mut and Leon-Mut, respectively. Thus, our benchmark set consist of 885 benchmarks (640 mutants plus the original 245). We considered these 885 benchmarks both before and after the translation $\mathcal{A}$. For example, Isa contains 79 original benchmarks φ and 79 translated benchmarks $\mathcal{A}(\varphi)$.

For solvers, we considered the SMT solver Z3 [9], which runs heuristic methods for quantifier instantiation [8] as well as methods for finding models for quantified formulas [11]. We also considered three configurations of CVC4 [1] which we will refer to as CVC4d, CVC4f, and CVC4i. The default configuration CVC4d runs heuristic and conflict-based techniques for quantifier instantiation [20], but does not include techniques for finding models. CVC4f runs heuristic instantiation and the finite model finding procedure due to Reynolds et al. [21, 22]. CVC4i incorporates techniques for automating inductive reasoning in SMT [19].

The results are summarized in Figures 2 and 3. The benchmarks and more detailed results are available online.[2] The figures are divided into benchmarks triggering *unsat* and *sat* responses and further into benchmarks before and after the translation $\mathcal{A}$. The raw evaluation data reveals no cases in which a solver answered *unsat* on a benchmark φ and *sat* on its corresponding benchmark $\mathcal{A}(\varphi)$, or vice versa. This is consistent with our expectations and Theorem 2, since these benchmarks contain only well-founded function definitions.

Figure 2 shows that for untranslated benchmarks (the "φ" columns), the number of *sat* responses is very low across all configurations. This confirms the shortcomings of existing SMT techniques for finding models for benchmarks containing recursively defined functions.

[1] `https://github.com/epfl-lara/leon/`

[2] `http://lara.epfl.ch/~reynolds/SMT2015-recfun/`

	Z3		CVC4d		CVC4f		CVC4i	
	φ	$\mathcal{A}(\varphi)$	φ	$\mathcal{A}(\varphi)$	φ	$\mathcal{A}(\varphi)$	φ	$\mathcal{A}(\varphi)$
Isa	0	0	0	0	0	0	0	0
Leon	0	2	0	0	0	**9**	0	0
Isa-Mut	0	35	0	0	0	**153**	0	0
Leon-Mut	11	75	6	6	6	**169**	6	6
Total	11	112	6	6	6	**331**	6	6

Figure 2: Number of *sat* responses on benchmarks without and with $\mathcal{A}$ translation

	Z3		CVC4d		CVC4f		CVC4i	
	φ	$\mathcal{A}(\varphi)$	φ	$\mathcal{A}(\varphi)$	φ	$\mathcal{A}(\varphi)$	φ	$\mathcal{A}(\varphi)$
Isa	14	15	15	15	15	15	**61**	14
Leon	73	78	80	80	80	76	**96**	78
Isa-Mut	17	18	18	18	18	18	**44**	17
Leon-Mut	83	98	103	98	104	95	**117**	98
Total	187	209	216	211	217	204	**318**	217

Figure 3: Number of *unsat* responses on benchmarks without and with $\mathcal{A}$ translation

The translation $\mathcal{A}$ (the "$\mathcal{A}(\varphi)$" columns) has a major impact. CVC4f finds 331 of the 885 benchmarks to be satisfiable, including 9 benchmarks in the nonmutated Leon benchmark set. The performance of Z3 for countermodels also improves dramatically, as it finds 101 more benchmarks to be satisfiable, including 10 that are not solved by CVC4f. We conclude that the translation $\mathcal{A}$ enables SMT solvers to find countermodels for conjectures involving recursively defined functions whose definitions are admissible.

Over *unsat* benchmarks, CVC4i is the clear winner, finding 318 total benchmarks to be unsatisfiable. Moreover, the translation $\mathcal{A}$ helps Z3 for *unsat* responses as well: Z3 solves a total of 209 with the translation, whereas it solves only 187 without it. In contrast, CVC4d finds 211 benchmarks unsatisfiable with the translation. In general, the translation $\mathcal{A}$ does not significantly degrade performance over unsatisfiable benchmarks, and in some cases it actually aids the solver in determining unsatisfiability. The exception is CVC4i, whose inductive arguments are guided by the types of the quantified variables.

5 Front-End Support in CVC4

The translation $\mathcal{A}$ significantly improves the effectiveness of current SMT techniques for finding models of formulas involving recursive function definitions. We have implemented it in the development version of CVC4 (version 1.5 prerelease). Function definitions $\forall_{\mathsf{f}}\, \overline{x}.\, \varphi$ can be written using the `define-fun-rec` command from the new version (2.5) of the SMT-LIB standard [2]. Thus, formula (1) from Example 1 can be specified as

```
(define-fun-rec s ((x Int)) Int (ite (<= x 0) 0 (+ x (s (- x 1)))))
```

```
(declare-fun c () Int)
(assert (> (s c) 100))
(check-sat)
```

When reading this input, CVC4 adds $\forall_{\mathsf{s}} x.\ \mathsf{s}(x) \approx \mathsf{ite}\big(x \leq 0,\ 0,\ \mathsf{s}(x-1)\big)$ to its list of assertions, which after rewriting becomes $\forall_{\mathsf{s}} x.\ \mathsf{ite}\big(x \leq 0,\ \mathsf{s}(x) \approx 0,\ \mathsf{s}(x) \approx \mathsf{s}(x-1)\big)$. If CVC4's finite model finding mode for recursive functions is enabled (using the command-line option `--fmf-fun`), it will replace its list of known assertions based on the translation $\mathcal{A}$ before checking for satisfiability. Accordingly, the solver will output the approximation of the interpretation it used for recursive function definitions. For the example above, it outputs a partial model of `s` where only the values of $\mathsf{s}(x)$ for $x = 0, \ldots, 14$ are given.

The `--fmf-fun` option tells CVC4 to assume that functions introduced using `define-fun-rec` are admissible. We stress that admissibility must be discharged separately by the user—e.g., using a syntactic criterion or a termination prover. If some function definitions are not admissible, CVC4 may answer *sat* for an unsatisfiable problem.

6 Related Work

We described the most closely related work, by Ge and de Moura [11] and by Reynolds et al. [21, 22], in the text already. The finite model finding support in the instantiation-based iProver [13] is also close, given the similarities with SMT.

Some finite model finders are based on a reduction to a decidable logic, typically propositional logic. The translation is parameterized by upper or exact finite bounds on the cardinalities of the atomic types. This procedure was pioneered by McCune in the earlier versions of MACE [16]. Other conceptually similar finders are Paradox [7] and FM-Darwin [3] for first-order logic with equality; the Alloy Analyzer and its back-end Kodkod [24] for first-order relational logic; and Refute [25] and Nitpick [5] for higher-order logic.

An alternative is to perform an exhaustive model search directly on the original problem. Given fixed cardinalities, the search space is represented as multidimensional tables. The procedure tries different values in the function and predicate tables, checking each time if the problem is satisfied. This approach was pioneered by FINDER [23] and SEM [26].

Kuncak and Jackson [15] presented an idiom for encoding datatypes and recursive functions in Alloy, by approximating datatypes by finite subterm-closed substructures. This idiom was extended by Dunets et al. [10], who presented a translation scheme for primitive recursion. Their definedness guards play a similar role to our existential constraints.

The higher-order model finder Nitpick [5] for the Isabelle/HOL proof assistant relies on another variant of Kuncak and Jackson's approach inside a Kleene-style three-valued logic. The three-valued logic approach extends each approximated type with an unknown value, which is propagated by function application. This scheme works reasonably well in Nitpick, because it builds on a relational logic, but our initial experiments with CVC4 suggest that it is more efficient to avoid unknowns by adding existential constraints.

The Leon system [4] implements a procedure that can produce both proofs and counterexamples for properties of terminating functions written in a subset of Scala. Leon is based on an SMT solver. It avoids quantifiers altogether by unfolding recursive definitions up to a certain depth. Our translation $\mathcal{A}$ works in an analogous manner, where instead the SMT solver is invoked only once and quantifier instantiation is used in lieu of function unfolding.

Model finding is concerned with satisfying arbitrary logical constraints. Some tools are tailored for problems that correspond to total functional programs. QuickCheck [6] for Haskell is an early example, based on random testing. Bounded exhaustive testing and narrowing are other successful strategies. These tools are often much faster than model finders, but they typically cannot cope with underspecification and nonexecutable functions.

7 Conclusion

We presented a translation scheme that extends the scope of finite model finding techniques in SMT, allowing one to use them to find models of quantified formulas over infinite types, such as integers and algebraic datatypes. In future work, it would be interesting to evaluate the approach against other counterexample generators, notably Leon and Nitpick, and enrich the benchmark suite with problems exercising CVC4's support for coalgebraic datatypes [17]. We also plan to integrate CVC4 as a counterexample generator in proof assistants. Further work would also include identifying additional sufficient conditions for admissibility, thereby enlarging the applicability of the translation scheme presented here.

Acknowledgment. Viktor Kuncak and Stephan Merz have made this work possible. We would also like to thank Damien Busato-Gaston and Emmanouil Koukoutos for providing the initial set of benchmarks used in the evaluation, and the anonymous reviewers for their suggestions and comments.

References

[1] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanovic, T. King, A. Reynolds, and C. Tinelli. CVC4. In G. Gopalakrishnan and S. Qadeer, editors, *CAV 2011*, volume 6806 of *LNCS*, pages 171–177. Springer, 2011.

[2] C. Barrett, P. Fontaine, and C. Tinelli. The SMT-LIB Standard: Version 2.5. Technical report, The University of Iowa, 2015. Available at `http://smt-lib.org/`.

[3] P. Baumgartner, A. Fuchs, H. de Nivelle, and C. Tinelli. Computing finite models by reduction to function-free clause logic. *J. Applied Logic*, 7(1):58–74, 2009.

[4] R. Blanc, V. Kuncak, E. Kneuss, and P. Suter. An overview of the Leon verification system—Verification by translation to recursive functions. In *Scala '13*. ACM, 2013.

[5] J. C. Blanchette and T. Nipkow. Nitpick: A counterexample generator for higher-order logic based on a relational model finder. In M. Kaufmann and L. C. Paulson, editors, *ITP 2010*, volume 6172 of *LNCS*, pages 131–146. Springer, 2010.

[6] K. Claessen and J. Hughes. QuickCheck: A lightweight tool for random testing of Haskell programs. In *ICFP '00*, pages 268–279. ACM, 2000.

[7] K. Claessen and N. Sörensson. New techniques that improve MACE-style model finding. In *MODEL*, 2003.

[8] L. de Moura and N. Bjørner. Efficient E-matching for SMT solvers. In F. Pfenning, editor, *CADE-21*, volume 4603 of *LNCS*, pages 183–198. Springer, 2007.

[9] L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In C. R. Ramakrishnan and J. Rehof, editors, *TACAS 2008*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.

[10] A. Dunets, G. Schellhorn, and W. Reif. Automated flaw detection in algebraic specifications. *J. Autom. Reasoning*, 45(4):359–395, 2010.

[11] Y. Ge and L. de Moura. Complete instantiation for quantified formulas in satisfiability modulo theories. In *CAV '09*, volume 5643 of *LNCS*, pages 306–320. Springer, 2009.

[12] M. Johansson, L. Dixon, and A. Bundy. Case-analysis for rippling and inductive proof. In *ITP 2010*, pages 291–306, 2010.

[13] K. Korovin. Non-cyclic sorts for first-order satisfiability. In P. Fontaine, C. Ringeissen, and R. A. Schmidt, editors, *FroCoS 2013*, volume 8152 of *LNCS*, pages 214–228. Springer, 2013.

[14] A. Krauss. *Automating Recursive Definitions and Termination Proofs in Higher-Order Logic*. Ph.D. thesis, Technische Universität München, 2009.

[15] V. Kuncak and D. Jackson. Relational analysis of algebraic datatypes. In M. Wermelinger and H. Gall, editors, *ESEC/FSE 2005*. ACM, 2005.

[16] W. McCune. A Davis–Putnam program and its application to finite first-order model search: Quasigroup existence problems. Technical report, Argonne National Laboratory, 1994.

[17] A. Reynolds and J. C. Blanchette. A decision procedure for (co)datatypes in SMT solvers. In A. Felty and A. Middeldorp, editors, *CADE-25*, LNCS. Springer, 2015.

[18] A. Reynolds, J. C. Blanchette, and C. Tinelli. Model finding for recursive functions in SMT. Tech. report, `http://lara.epfl.ch/~reynolds/SMT2015-recfun/`, 2015.

[19] A. Reynolds and V. Kuncak. Induction for SMT solvers. In D. D'Souza, A. Lal, and K. G. Larsen, editors, *VMCAI 2015*, volume 8931 of *LNCS*, pages 80–98. Springer, 2014.

[20] A. Reynolds, C. Tinelli, and L. de Moura. Finding conflicting instances of quantified formulas in SMT. In *FMCAD 2014*, pages 195–202. IEEE, 2014.

[21] A. Reynolds, C. Tinelli, A. Goel, and S. Krstić. Finite model finding in SMT. In N. Sharygina and H. Veith, editors, *CAV 2013*, volume 8044 of *LNCS*, pages 640–655. Springer, 2013.

[22] A. Reynolds, C. Tinelli, A. Goel, S. Krstić, M. Deters, and C. Barrett. Quantifier instantiation techniques for finite model finding in SMT. In M. P. Bonacina, editor, *CADE-24*, volume 7898 of *LNCS*, pages 377–391. Springer, 2013.

[23] J. K. Slaney. FINDER: Finite domain enumerator system description. In A. Bundy, editor, *CADE-12*, volume 814 of *LNCS*, pages 798–801. Springer, 1994.

[24] E. Torlak and D. Jackson. Kodkod: A relational model finder. In O. Grumberg and M. Huth, editors, *TACAS 2007*, volume 4424 of *LNCS*, pages 632–647. Springer, 2007.

[25] T. Weber. *SAT-Based Finite Model Generation for Higher-Order Logic*. Ph.D. thesis, Technische Universität München, 2008.

[26] J. Zhang and H. Zhang. SEM: A system for enumerating models. In C. S. Mellish, editor, *IJCAI-95*, volume 1, pages 298–303. Morgan Kaufmann, 1995.