Model Finding for Recursive Functions in SMT

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

[1] Department of Computer Science, The University of Iowa, USA
[2] Inria Nancy – Grand Est & LORIA, Villers-lès-Nancy, France
[3] Max-Planck-Institut für Informatik, Saarbrücken, Germany

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 three sources. The translation is implemented as a preprocessor in the CVC4 solver and in a new higher-order model finder called Nunchaku.

1 Introduction

Many solvers based on SMT (satisfiability modulo theories) can reason about quantified formulas using incomplete instantiation-based methods [11, 25]. 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 [14] 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. [26, 27] 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 $\forall x, y : \tau.\ x \approx y \lor \neg\, \mathsf{f}(x) \approx \mathsf{f}(y)$, 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 approaches fail is

$$\forall x : \mathsf{Int}.\ \mathsf{p}(x) \approx \mathsf{ite}\big(x \leq 0,\ 1,\ 2 * \mathsf{p}(x-1)\big)$$

This state of affairs is unsatisfactory, given the frequency of recursive definitions in practice and the addition of a command for introducing them, `define-fun-rec`, to the SMT-LIB standard [3].

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. It is not met by inconsistent definitions such as $\forall x : \mathsf{Int}.\ \mathsf{f}(x) \approx \mathsf{f}(x) + 1$.

We define a translation for a class of formulas involving admissible recursive function definitions (Section 3). A recursive definition $\forall x : \tau.\ \mathsf{f}(x) \approx \mathit{rhs}$ is translated to $\forall a : \alpha_\tau.\ \mathsf{f}(\gamma_\mathsf{f}(a)) \approx \mathit{rhs}[\gamma_\mathsf{f}(a)/x]$, where α_τ is an uninterpreted abstract type and $\gamma_\mathsf{f} : \alpha_\tau \to \tau$ 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, for admissible definitions, unsatisfiability, and makes finite model finding possible for problems in this class.

The approach is implemented as a preprocessor in CVC4 and in the Nunchaku model finder (Section **??**). We evaluated the two evaluation on benchmarks from IsaPlanner [16], Leon [5], and Isabelle/HOL, to demonstrate that this translation improves the effectiveness of the SMT solvers CVC4 and Z3 for finding countermodels to verification conditions (Section 4). Unlike earlier work, our approach can be combined with off-the-shelf SMT solvers (Section **??**).

An earlier version of this paper was presented at the SMT 2015 workshop in San Francisco [**?**]. This paper extends the workshop paper with proof sketches, an expanded implementation section covering Nunchaku and relevant CVC4 optimizations, and the evaluation on Isabelle benchmarks produced by Nunchaku.

2 Preliminaries

Our setting is a monomorphic (or many-sorted) first-order logic like the one defined by SMT-LIB [3]. 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 $\mathrm{ite} : \mathsf{Bool} \times \tau \times \tau \to \tau$ for each $\tau \in \Sigma^{\mathrm{ty}}$. We fix an infinite set Σ^{v}_τ of *variables of type* τ for each $\tau \in \Sigma^{\mathrm{ty}}$ 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. When applied to terms, the symbol $=$ denotes syntactic equality.

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

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 $(\forall_{\mathsf{f}_1} \bar{x}_1.\ \varphi_1) \wedge \cdots \wedge (\forall_{\mathsf{f}_n} \bar{x}_n.\ \varphi_n) \wedge \psi$, where $\mathsf{f}_1, \ldots, \mathsf{f}_n$ are distinct function symbols, $\forall_{\mathsf{f}_i} \bar{x}_i.\ \varphi_i$ is a function definition for $i = 1, \ldots, n$, and ψ contains no function definitions. We call ψ the *goal* 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 $\Delta = \{\forall_{\mathsf{f}_1} \bar{x}.\ \varphi_1, \ldots, \forall_{\mathsf{f}_n} \bar{x}.\ \varphi_n\}$, a ground formula ψ is *closed under function expansion with respect to* Δ if

$$\psi \models_T \bigwedge\nolimits_{i=1}^{n} \{\varphi_i[\bar{t}/\bar{x}] \mid \mathsf{f}_i(\bar{t}) \in \mathcal{T}(\psi)\}$$

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 useful 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 $\forall x : \mathsf{Int}.\ \mathsf{p}(x) \approx \mathsf{ite}\big(x \leq 0,\ 1,\ 2 * \mathsf{p}(x-1)\big)$, 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. Logically, such dependencies can be captured by congruence rules. Krauss developed these ideas in the more general context of higher-order logic [18, Section 2].

Theorem 3. *If Δ is a set of well-founded function definitions for $\Sigma^{\mathrm{dfn}} = \{\mathsf{f}_1, \ldots, \mathsf{f}_n\}$, then it is admissible.*

Proof sketch. Let ψ be a satisfiable formula closed under function expansion with respect to Δ. We show that $\psi \wedge \bigwedge \Delta$ is also satisfiable. Let I be a model of ψ, and let I_0 be the restriction of I to the function symbols in $\Sigma^{\mathrm{f}} - \Sigma^{\mathrm{dfn}}$. Because well-founded definitions uniquely characterize the interpretation of the functions they define, there exists a Σ-interpretation $\mathcal{J}$ that extends I_0 such that $\mathcal{J} \models \Delta$. Since ψ is closed under function expansion, it already constrains the functions in Σ^{dfn} recursively as far as is necessary for interpreting ψ. Thus, any point v for which $\mathsf{f}_i^I(v)$ is needed for interpreting ψ will have its expected value according to its definition and hence coincide with $\mathcal{J}$. And since ψ^I does not depend on the interpretation at the other points, $\mathcal{J}$ is, like I, a model of ψ. Since $\mathcal{J} \models \Delta$ by assumption, we have $\mathcal{J} \models \psi \wedge \bigwedge \Delta$ as desired. $\square$

Another useful fragment of function definitions is the class of *productive* corecursive functions. Corecursive functions are functions to a coalgebraic datatype. These functions can be ill founded, without their being inconsistent. Productive corecursive functions are functions that progressively reveal parts of their potentially infinite output [1, 31]. For example, given a type of infinite streams constructed by $\mathsf{scons} : \mathsf{int} \times \mathsf{stream} \to \mathsf{stream}$, the definition $\forall_{\mathsf{e}}\, x.\ \mathsf{e}(x) \approx \mathsf{scons}(x,\ \mathsf{e}(x+1))$ falls within this fragment: Each call to e produces one constructor before entering the nested call. Like terminating recursion, productive corecursion totally specify the functions it defines. It is even possible to mix recursion and corecursion in the same function [8]. Theorem 3 can be extended to cover such specifications, based on the observation that unfolding a corecursive definition infinitely computes a unique infinite object.

Beyond totality, an admissible set can contain underspecified functions such as $\forall_{\mathsf{f}}\, x : \mathsf{Int}.\ \mathsf{f}(x) \approx \mathsf{f}(x-1)$ or $\forall_{\mathsf{g}}\, x.\ \mathsf{g}(x) \approx \mathsf{g}(x)$. We conjecture that one can ignore all tail-recursive calls (i.e., calls that occupy the right-hand side of the definition, potentially under some ite branch) when establishing well-foundedness or productivity, without affecting admissibility.

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 with respect to this set and yet there exists no interpretation satisfying both $\mathsf{f}(0) \approx 1$ and g's definition.

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

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

$\mathcal{A}(\varphi) =$ let $(\varphi', \chi) = \mathcal{A}_0(\varphi, \mathsf{pos})$ in $\varphi'$
```

Fig. 1. Definition of translation $\mathcal{A}$

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 $\exists a : \alpha_{\mathsf{f}}.\; \bar{\gamma}_{\mathsf{f}}(a) \approx \bar{s}$, where $\bar{\gamma}_{\mathsf{f}}(a) \approx \bar{s}$ abbreviates the formula $\bigwedge_{i=1}^{n} \gamma_{\mathsf{f},i}(a) \approx s_i$ with $\bar{s} = (s_1, \ldots, s_n)$. 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. The constraints are generated as deep in the formula as possible, based on our knowledge of the polarities of logical connectives, to allow models in which the domains interpreting the α_{f} types are as small as possible.

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 and returns $\top$ as the constraint. 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 $\bar{x}$ in the body of that formula with $\bar{\gamma}_\mathsf{f}(a)$. (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; a quantifier is prefixed to the quantified formula and constraint returned by the recursive call. 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 $\exists a : \alpha_\mathsf{f}.\ \bar{\gamma}_\mathsf{f}(a) \approx \bar{s}$ for each subterm $\mathsf{f}(\bar{s})$ 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 $\bar{s}$.

Example 4. Let T be linear integer arithmetic with the uninterpreted symbols $\{\mathsf{c} : \mathsf{Int}, \mathsf{s} : \mathsf{Int} \to \mathsf{Int}\}$. Let φ be the Σ-formula

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

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

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

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. [26, 27], 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 [14] 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 α_{f}. This can sometimes help users debug their specifications.

We sketch the correctness of translation $\mathcal{A}$. For a set of ground literals L, we write $\mathrm{X}(L)$ to denote the set of constraints that force the concretization functions to have the necessary elements in their range for determining the satisfiability of L with respect to the function definitions in the translation. Formally,

$$\mathrm{X}(L) = \{\exists a : \alpha_{\mathsf{f}}.\ \overline{\gamma}_{\mathsf{f}}(a) \approx \bar{t} \mid \mathsf{f}(\bar{t}) \in \mathcal{T}(L), \mathsf{f} \in \Sigma^{\mathrm{dfn}}\}$$

The following lemma states the central invariant behind the translation $\mathcal{A}$.

Lemma 5. *Let ψ be a formula not containing function definitions, and let I be an $\mathcal{E}(\Sigma)$-interpretation. Then I satisfies $\mathcal{A}(\psi)$ if and only if I satisfies $L \cup \mathrm{X}(L)$, where L is a set of ground Σ-literals that entail ψ.*

Proof sketch. By definition of $\mathcal{A}$ and case analysis on the return values of $\mathcal{A}_0$. □

Corollary 6. *If ψ is a formula not containing function definitions, then $\mathcal{A}(\psi)$ entails ψ.*

Theorem 7. *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.*

Proof sketch. First, we show that if φ is satisfied by an Σ-interpretation I, then $\mathcal{A}(\varphi)$ is satisfied by an $\mathcal{E}(\Sigma)$-interpretation $\mathcal{J}$. Let $\mathcal{J}$ be the $\mathcal{E}(\Sigma)$-interpretation that interprets all types $\tau \in \Sigma^{\mathrm{ty}}$ as τ^{I}, all functions $\mathsf{f} \in \Sigma^{\mathrm{f}}$ as f^{I}, and for each function $\mathsf{f} : \tau_1 \times \cdots \times \tau_n \to \tau$ in Σ^{dfn}, interprets α_{f} as $\tau_1^{I} \times \cdots \times \tau_n^{I}$ and each $\gamma_{\mathsf{f},i}$ as the ith projection on such tuples for $i = 1, \ldots, n$. Since $\mathcal{J}$ satisfies φ, it satisfies a set of ground literals L that entail φ. Furthermore, $\mathcal{J}$ satisfies every constraint of the form $\exists a : \alpha_{\mathsf{f}}.\ \overline{\gamma}_{\mathsf{f}}(a) \approx \bar{t}$, since by our construction of $\mathcal{J}$ there exists a value $v \in \alpha_{\mathsf{f}}{}^{\mathcal{J}}$ such that $v = \bar{t}^{\mathcal{J}}$. Thus, $\mathcal{J}$ satisfies $L \cup \mathrm{X}(L)$, and by Lemma 5 we conclude $\mathcal{J}$ satisfies $\mathcal{A}(\varphi)$.

Second, we show that if $\mathcal{A}(\varphi)$ is satisfied by a $\mathcal{E}(\Sigma)$-interpretation $\mathcal{J}$, then φ is satisfied by a Σ-interpretation I. Since φ is in definitional form with respect to the functions defined by Δ, it must be of the form $\Delta \wedge \varphi_0$. First, we define a sequence of Σ-literals sets $L_0 \subseteq L_1 \subseteq \cdots$ such that $\mathcal{J}$ satisfies $L_i \cup \mathrm{X}(L_i)$ for $i = 0, 1, \ldots$. Since $\mathcal{J}$ satisfies $\mathcal{A}(\varphi_0)$, by Lemma 5, $\mathcal{J}$ satisfies a set of literals $L \cup \mathrm{X}(L)$ where L is a set of Σ-literals that entail φ_0. Let $L_0 = L$. For each $i \geq 0$, let ψ_i be the formula $\bigwedge \{\mathcal{A}(\varphi_{\mathsf{f}}[\bar{t}/\bar{x}]) \mid$

$\mathsf{f}(\bar{t}) \in \mathcal{T}(L_i), \mathsf{f} \in \Sigma^{\mathrm{dfn}}\}$, where $\forall_{\mathsf{f}}\, \bar{x}.\ \varphi_{\mathsf{f}} \in \Delta$. Since $\mathcal{J}$ satisfies $\mathcal{A}(\forall_{\mathsf{f}}\, \bar{x}.\ \varphi_{\mathsf{f}})$ and $\mathrm{X}(L_i)$, we know that $\mathcal{J}$ also satisfies ψ_i. Thus by Lemma 5, $\mathcal{J}$ satisfies a set of literals $L \cup \mathrm{X}(L)$ where L is a set of Σ-literals that entail ψ_i. Let $L_{i+1} = L_0 \cup L$. Let L_∞ be the limit of this sequence (i.e., $\ell \in L_\infty$ if and only if $\ell \in L_i$ for some i), and let ψ be the Σ-formula $\bigwedge L_\infty$. To show that ψ is closed under function expansion with respect to Δ, we first note that by construction ψ entails ψ_∞. For any function symbol f and terms $\bar{t}$, since $\varphi_{\mathsf{f}}[\bar{t}/\bar{x}]$ does not contain function definitions, by Corollary 6, $\mathcal{A}(\varphi_{\mathsf{f}}[\bar{t}/\bar{x}])$ entails $\varphi_{\mathsf{f}}[\bar{t}/\bar{x}]$. Thus, ψ entails $\{\varphi_{\mathsf{f}}[\bar{t}/\bar{x}] \mid \mathsf{f}(\bar{t}) \in \mathcal{T}(\psi), \mathsf{f} \in \Sigma^{\mathrm{dfn}}\}$, meaning that ψ is closed under function expansion with respect to Δ. Furthermore, ψ entails φ_0 since $L_0 \subseteq L_\infty$. Since ψ is a T-satisfiable formula that is closed under function expansion with respect to Δ and Δ is admissible, by definition there exists a Σ-interpretation I satisfying $\psi \wedge \Delta$, which entails $\Delta \wedge \varphi_0$, i.e., φ. □

The intuition of the above proof is as follows. First, $\mathcal{A}(\varphi)$ cannot be unsatisfiable when φ is satisfiable since any Σ-interpretation that satisfies φ can be extended in a straightforward way to an $\mathcal{E}(\Sigma)$-interpretation that satisfies $\mathcal{A}(\varphi)$, by interpreting the abstract types in the same way as the cartesian products they abstract, thereby satisfying all existential constraints introduced by $\mathcal{A}$. Conversely, if a model is found for $\mathcal{A}(\varphi)$, existential constraints introduced by $\mathcal{A}$ ensure that this model also satisfies a Σ-formula that is closed under function expansion and that entails the goal of φ. This implies the existence of a model for φ, assuming Δ is admissible.

Example 8. Let us revisit the formulas in Example 4. 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 $\mathsf{ite}\big(x \le 0,\ \mathsf{s}(x) \approx 0,\ \mathsf{s}(x) \approx x + \mathsf{s}(x-1)\big)$. 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

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

The existential constraint in the above formula ensures that whenever I satisfies the set $L_1 = L_0 \cup \{\neg\, \mathsf{c} \le 0,\ \mathsf{s}(\mathsf{c}) \approx \mathsf{c} + \mathsf{s}(\mathsf{c}-1)\}$, 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:

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

This formula is closed under function expansion since it entails $\varphi_{\mathsf{s}}[(\mathsf{c}-i)/x]$ for $i = 0,\dots,n$, and it contains only s applications corresponding to $\mathsf{s}(\mathsf{c}-i)$ for $i = 0,\dots,n$. 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 Implementations

We have implemented the translation $\mathcal{A}$ in two separate systems, as a preprocessor in CVC4 (version 1.5 prerelease) and in the CVC4-based higher-order model finder Nunchaku. This section describes how the translation is implemented in these systems, as well as optimizations used by CVC4 for finding models of translated problems.

4.1 CVC4

In CVC4, function definitions $\forall_{\mathsf{f}}\,\bar{x}.\ \varphi$ can be written using the define-fun-rec command from SMT-LIB 2.5 [3]. Formula (1) from Example 4 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 the annotated quantified formula

$$\forall_{\mathsf{s}}\, x.\ \mathsf{s}(x) \approx \mathsf{ite}\big(x \leq 0,\ 0,\ \mathsf{s}(x-1)\big)$$

to its list of assertions, which after rewriting becomes

$$\forall_{\mathsf{s}}\, x.\ \mathsf{ite}\big(x \leq 0,\ \mathsf{s}(x) \approx 0,\ \mathsf{s}(x) \approx \mathsf{s}(x-1)\big)$$

By specifying the command-line option `--fmf-fun`, users can enable CVC4's finite model finding mode for recursive functions. In this mode, CVC4 will replace its list of known assertions based on the $\mathcal{A}$ translation 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 model of s where only the values of $\mathsf{s}(x)$ for $x = 0, \ldots, 14$ are correctly given:

```
(model
  (define-fun s (($x1 Int)) Int
    (ite (= $x1 14) 105 (ite (= $x1 13) 91 (ite (= $x1 12) 78
      (ite (= $x1 11) 66 (ite (= $x1 10) 55 (ite (= $x1 4) 10
        (ite (= $x1 9) 45 (ite (= $x1 8) 36 (ite (= $x1 7) 28
          (ite (= $x1 6) 21 (ite (= $x1 3) 6 (ite (= $x1 5) 15
            (ite (= $x1 2) 3 (ite (= $x1 1) 1 0))))))))))))))))
  (define-fun c () Int 14))
```

With the `--fmf-fun` option enabled, CVC4 assumes that functions introduced using `define-fun-rec` are admissible. 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. Indeed, if we add the inconsistent definition

```
(define-fun-rec h ((x Int)) Int (+ (h x) x))
```

to the above problem and run CVC4 with the `--fmf-fun` option, it wrongly answers *sat*.

CVC4 implements a few optimizations designed to help finding finite models of $\mathcal{A}(\varphi)$. Like other systems, the finite model finding capability of CVC4 incrementally fixes bounds on the cardinalities of uninterpreted types and increases these bounds until it encounters a model. When multiple types are present, it uses a fairness scheme that bounds the sum of cardinalities of all uninterpreted types [**?**]. For example, if a signature has two uninterpreted types τ_1 and τ_2, it will first search for models where $|\tau_1| + |\tau_2|$ is at most 2, then 3, 4, and so on. To accelerate the search for models, we implemented an optimization based on statically inferring *monotonic* types. A monotonic type is one in which models can always be extended with additional elements of that type [**?**, **?**]. Types α_{f} introduced by our translation $\mathcal{A}$ are monotonic, because $\approx$ is never used directly on such types [**?**]. CVC4 takes advantage of this by fixing the bounds for all monotonic types simultaneously. That is, if τ_1 and τ_2 are inferred monotonic (whether they are present in the original problem or introduced by our translation), the solver fixes the bound for both types to be 1, then 2, and so on. This scheme allows the solver greater flexibility compared with the default scheme, and comes with no loss of generality with respect to models, since monotonic types can always be extended to have equal cardinalities.

By default, CVC4 uses techniques to minimize the number of literals it considers when constructing propositional satisfying assignments for formulas [**?**]. However, we have found such techniques degrade performance for finite model finding on problems having recursive functions that are defined by cases. For this reason, we disable the techniques for problems produced from our translation.

4.2 Nunchaku

Nunchaku is a new higher-order model finder designed to be integrated with several proof assistants. The first version, 0.1, was released in January 2016 with support for (co)algebraic datatypes, (co)recursive functions and (co)inductive predicates. Support for higher-order functions is planned for the next release. We have developed a preliminary Isabelle frontend and are planning further frontends for Coq, the TLA$^+$ Proof System, and possibly other proof assistants.

Nunchaku is the spiritual successor to Nitpick for Isabelle/HOL [7], but is developed as a standalone OCaml program, with its own input language. Whereas Nitpick generates a succession of problems where cardinalities of finite types grow at each step, Nunchaku translates its input to one first-order logic program that targets the finite model finding fragment of CVC4, including (co)algebraic datatypes [23]. Using CVC4 also allows Nunchaku to provide efficient arithmetic reasoning and to detect unsatisfiability in addition to satisfiability.

The input syntax was inspired by that other systems based on higher-order logic (e.g., Isabelle/HOL) and by functional programming languages (e.g., OCaml). The following simple problem gives a taste of the syntax:

```
data nat := Zero | Suc nat.

pred even : nat -> prop :=
  even Zero;
  forall n. odd n => even (Suc n)
and odd : nat -> prop :=
  forall n. even n => odd (Suc n).

val m : nat.
goal even m && ~ (m = Zero).
```

The problem defines a datatype (`nat`) and two mutually recursive inductive predicates (`even` and `odd`), it declares a constant `m`, and specifies a goal to satisfy ("*m* is even and nonzero"). Nunchaku quickly finds the following partial model:

```
val m := Suc (Suc Zero).
val odd := fun x. if x = Suc Zero then true else ?__.
val even := fun x. if x = Suc (Suc Zero) || x = Zero then true
                   else ?__.
```

The partial model gives sufficient information to the user to evaluate the goal: "2 is even if 1 is odd, 1 is odd if 0 is even, and 0 is even."

Nunchaku parses and types the input problem before applying a sequence of translations, each reducing the distance to the target fragment. In our example, the predicates `even` and `odd` are *polarized* (specialized into a pair of predicates such that one is used in positive positions and the other in negative positions), *m* is skolemized into a fresh constants, then translated into admissible recursive functions, before another pass applies the encoding described in this paper. If a model is found, it is translated back to the input language, with `?__` placeholders indicating unknown values.

Conceptually, the sequence of transformation is a two-way pipeline built by composing pairs (encode, decode) of transformations. For each such pair, encode transforms a problem over a signature Σ in a logic $\mathscr{L}$ to a problem over a signature Σ' in a logic $\mathscr{L}'$, and decode translates a model in $\mathscr{L}'$ over Σ' into a model in $\mathscr{L}$ over Σ, in the spirit of institution theory [**?**]. The pipeline currently consists of the following phases:

Type inference infers types and checks definitions;
Type skolemization replaces $\exists\alpha.\ \varphi[\alpha]$ with $\varphi[\tau]$, where τ is a fresh type;
Monomorphization specializes polymorphic definitions on their type arguments and removes unused definitions;
Elimination of equations translates multiple-equation definitions of recursive functions into a single nested pattern matching;
Polarization specializes predicates into a version used in positive positions and a version used in negative positions;
Unrolling adds a decreasing argument to possibly ill-founded predicates to make them well founded;
Skolemization introduces Skolem symbols for term variables;
Elimination of (co)inductive predicates recasts a multiple-clause (co)inductive predicate specification into a recursive equation;
Recursion elimination performs the encoding from Section 3;
Elimination of pattern matching rewrites pattern-matching expressions using datatype discriminators and selectors;
CVC4 invocation runs CVC4 to obtain a model.

5 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 602 benchmarks from three sources, which we will refer to as IsaPlanner, Leon, and Nunchaku-Mut:

- IsaPlanner consists of the 79 benchmarks from the IsaPlanner suite [16] 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.
- Leon 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.
- Nunchaku-Mut consists of 357 benchmarks originating from Isabelle/HOL. They involve (co)recursively defined functions over (co)algebraic datatypes and uninterpreted functions but no other theories. They were obtained by mutation of negated Isabelle theorems, as was done for evaluating Nitpick [7]. Benchmarks created by mutation have a high likelihood of having small, easy-to-find models.

The IsaPlanner and Leon benchmarks are expressed in SMT-LIB 2.5 and are in definitional form with respect to a set of well-founded functions. The Leon tool was used to generate SMT-LIB files. A majority of these benchmarks are unsatisfiable. For each of the 245 benchmarks, we considered up to three randomly selected mutated forms of its goal ψ. In particular, we considered unique formulas that are obtained as a result of exchanging a subterm of ψ at one position with another of the same type at another position. In total, we considered 213 mutated forms of theorems from IsaPlanner and 427 mutated forms of theorems from Leon. We will call these sets IsaPlanner-Mut and Leon-Mut, respectively. Each of these benchmarks exists in two versions: without and with the $\mathcal{A}$ translation. Problems with $\mathcal{A}$ were produced by running CVC4's preprocessor on each benchmark.

For Nunchaku-Mut, the Isabelle Nunchaku frontend was used to generate thousands of Nunchaku problems from Isabelle/HOL theory files involving lists, trees, and other functional data structures. Nunchaku was then used to generate SMT-LIB files, again in two versions: without and with the $\mathcal{A}$ translation. Problems requiring higher-order logic were discarted, since Nunchaku does not yet support them, leaving 357 problems.

Among SMT solvers, we considered Z3 [12] and CVC4 [2]. Z3 runs heuristic methods for quantifier instantiation [11] as well as methods for finding models for quantified formulas [14]. For CVC4, we considered four configurations, referred to as CVC4h, CVC4f, CVC4fh, and CVC4fm. The configuration CVC4h runs heuristic and conflict-based techniques for quantifier instantiation [25], but does not include techniques for finding models. The other configurations run the finite model finding procedure due to

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

	Z3		CVC4h		CVC4f		CVC4fh		CVC4fm	
	φ	$\mathcal{A}(\varphi)$	φ	$\mathcal{A}(\varphi)$	φ	$\mathcal{A}(\varphi)$	φ	$\mathcal{A}(\varphi)$	φ	$\mathcal{A}(\varphi)$
IsaPlanner	0	0	0	0	0	0	0	0	0	0
IsaPlanner-Mut	0	41	0	0	0	**153**	0	**153**	0	**153**
Leon	0	2	0	0	0	9	0	9	0	**10**
Leon-Mut	11	78	6	6	6	**189**	6	**189**	6	**189**
Nunchaku-Mut	3	27	0	0	3	199	2	**200**	2	199
Total	14	148	6	6	8	550	8	**551**	8	**551**

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

	Z3		CVC4h		CVC4f		CVC4fh		CVC4fm	
	φ	$\mathcal{A}(\varphi)$	φ	$\mathcal{A}(\varphi)$	φ	$\mathcal{A}(\varphi)$	φ	$\mathcal{A}(\varphi)$	φ	$\mathcal{A}(\varphi)$
IsaPlanner	14	**15**	**15**	**15**	1	**15**	**15**	**15**	1	**15**
IsaPlanner-Mut	**18**	**18**	**18**	**18**	4	**18**	**18**	**18**	4	**18**
Leon	74	79	**80**	**80**	17	78	**80**	77	17	78
Leon-Mut	84	98	**104**	98	24	100	**104**	98	24	100
Nunchaku-Mut	**61**	59	46	53	45	59	44	59	45	59
Total	251	269	263	264	91	**270**	261	267	91	**270**

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

Reynolds et al. [26, 27]. The configuration CVC4fh additionally incorporates heuristic quantifier instantiation as described in Section 2.3 of [27], and CVC4fm incorporates the fairness scheme for monotonic types as described in Section **??**.

The results are summarized in Figures 2 and 3. Bold indicates the maximum of a row. 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 7, 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. The translation $\mathcal{A}$ (the "$\mathcal{A}(\varphi)$" columns) has a major impact. CVC4f finds 550 of the 1242 benchmarks to be satisfiable, including 6 benchmarks in the non-mutated Leon benchmark set. The two optimizations for finite model finding in CVC4 (configurations CVC4fh and CVC4fm) led to a net gain of one satisfiable benchmark each with respect to CVC4f. The performance of Z3 for countermodels also improved dramatically, as it finds 134 more benchmarks to be satisfiable, including 5 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.

[2] `http://lara.epfl.ch/~reynolds/IJCAR2016-recfun/`

Moreover, the translation $\mathscr{A}$ helps all configurations for *unsat* responses as well. Z3 solves a total of 269 with the translation, whereas it solves only 251 without it. Surprisingly, the configuration CVC4f, which is not tailored for handling unsatisfiable benchmarks, solves 270 *unsat* benchmarks overall, which is more than both CVC4h and Z3. These results suggest that the translation do not degrade the performance of SMT solvers for unsatisfiable problems involving recursive functions, and instead often improve their performance. It would be interesting to try this out in Sledgehammer [**?**] and to try Nunchaku as a proof tool.

6 Related Work

We described the most closely related work, by Ge and de Moura [14] and by Reynolds et al. [26, 27], in the text already. The finite model finding support in the instantiation-based iProver [17] is also close, given the similarities with SMT.

Some finite model finders are based on a reduction to a decidable logic, typically propositional logic. They translate the input problem to the weaker logic and pass it to a solver for that 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 (originally styled MACE) [22]. Other conceptually similar finders are Paradox [10] and FM-Darwin [4] for first-order logic with equality; the Alloy Analyzer and its back-end Kodkod [30] for first-order relational logic; and Refute [32] and Nitpick [7] 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 [29] and SEM [33] and serves as the basis of many more model finders, notably the Alloy Analyzer's precursor [15] and the later versions of Mace [21].

Most of the above tools cannot cope with algebraic datatypes or other infinite types. Kuncak and Jackson [19] presented an idiom for encoding datatypes and recursive functions in Alloy, by approximating datatypes by finite subterm-closed substructures. The approach finds sound (fragments of) models for formulas in the existential–bounded-universal fragment(i.e., formulas whose prenex normal forms contain no unbounded universal quantifiers ranging over datatypes). This idiom was further developed by Dunets et al. [13], who presented a translation scheme for primitive recursion. Their definedness guards play a similar role to the existential constraints generated by our translation $\mathscr{A}$. An approach related in scope to ours is given in [**?**], which establishes the satisfiability of formulas in the presence of admissible axioms over infinite domains by proving their negation is entailed.

The higher-order model finder Nitpick [7] for the Isabelle/HOL proof assistant relies on another variant of Kuncak and Jackson's approach inside a Kleene-style three-valued logic, inspired by abstract interpretation. It was also the first tool of its kind to support corecursion and coalgebraic datatypes [6]. 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 [5] 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, which is increased on a per-need basis. 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. It would be worth investigating how existing approaches for function unfolding can inform approaches for dedicated quantifier instantiation techniques for function definitions, and vice versa.

Model finding is concerned with satisfying arbitrary logical constraints. Some tools are tailored for problems that correspond to total functional programs. QuickCheck [9] for Haskell is an early example, based on random testing. Bounded exhaustive testing [28] and narrowing [20] 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, Nitpick, and Quickcheck, and enrich the benchmark suite with more problems exercising CVC4's support for coalgebraic datatypes [23]. We are also working on an encoding of higher-order functions in SMT-LIB, as a generalization to the current translation scheme, for Nunchaku. 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 set of Leon benchmarks used in the evaluation, and the anonymous reviewers for their suggestions and comments. Cruanes is supported by the Inria technological development action "Contre-exemples utilisables par Isabelle et Coq" (CUIC).

References

[1] R. Atkey and C. McBride. Productive coprogramming with guarded recursion. In G. Morrisett and T. Uustalu, editors, *ICFP '13*, pages 197–208. ACM, 2013.

[2] 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.

[3] 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/`.

[4] 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.

[5] 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.

[6] J. C. Blanchette. Relational analysis of (co)inductive predicates, (co)inductive datatypes, and (co)recursive functions. *Softw. Qual. J.*, 21(1):101–126, 2013.

[7] 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.

[8] J. C. Blanchette, A. Popescu, and D. Traytel. Foundational extensible corecursion—A proof assistant perspective. In J. Reppy, editor, *ICFP '15*. ACM, 2015.

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

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

[11] 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.

[12] 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.

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

[14] 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.

[15] D. Jackson. Nitpick: A checkable specification language. In *FMSP '96*, pages 60–69, 1996.

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

[17] 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.

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

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

[20] F. Lindblad. Property directed generation of first-order test data. In M. Morazán, editor, *TFP 2007*, pages 105–123. Intellect, 2008.

[21] W. McCune. Prover9 and Mace4. `http://www.cs.unm.edu/~mccune/prover9/`.

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

[23] 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.

[24] 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.

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

[26] 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.

[27] 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.

[28] C. Runciman, M. Naylor, and F. Lindblad. SmallCheck and Lazy SmallCheck: Automatic exhaustive testing for small values. In A. Gill, editor, *Haskell 2008*, pages 37–48. ACM, 2008.

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

[30] 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.

[31] D. A. Turner. Elementary strong functional programming. In P. H. Hartel and M. J. Plasmeijer, editors, *FPLE '95*, volume 1022 of *LNCS*, pages 1–13. Springer, 1995.

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

[33] 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.