A Decision Procedure for (Co)datatypes in SMT Solvers*

Andrew Reynolds
Department of Computer Science
The University of Iowa, USA

Jasmin Christian Blanchette
Inria Nancy – Grand Est & LORIA
Villers-lès-Nancy, France

Abstract

Datatypes and codatatypes are useful to represent finite and potentially infinite objects. We describe a decision procedure to reason about such types. The procedure has been integrated into CVC4, a modern SMT (satisfiability modulo theories) solver, which can be used both as a constraint solver and as an automatic theorem prover. An evaluation based on formalizations developed in the Isabelle proof assistant shows the potential of the procedure.

1 Introduction

In the past decade, satisfiability modulo theories (SMT) solvers [Nieuwenhuis *et al.*, 2006] have emerged as useful tools in a number of applications [De Moura and Bjørner, 2011]. These solvers are one of the most powerful ways to prove theorems in classical first-order logic, and are also useful for constraint solving, or model finding: Given some constraints, they can be used to find an assignment, or model, that satisfies them.

A great benefit of the SMT approach is that it provides a flexible framework for composing dedicated decision procedures and other solvers for various theories, including equality, linear arithmetic on $\mathbb{Z}$ and $\mathbb{R}$, bit vectors (for representing n-bit machine words), and first-order quantifiers ($\forall$, $\exists$). In this context, recurrent scientific questions include:

- Which theories would be useful to applications?
- Which theories could be implemented efficiently in SMT solvers?

This paper offers the following partial answer: freely generated algebraic and coalgebraic datatypes—also called *datatypes* and *codatatypes*. Datatypes are ubiquitous in functional programming and formal specifications. They are useful to represent finite data structures in computer science but also arise in formalized mathematics. And to represent infinite objects, a natural choice is to turn to codatatypes, their non-well-founded dual. Despite their reputation for esotericism, codatatypes have a role to play in computer science. For example, Leroy's verified C compiler [2009] and Lochbihler's formalized Java memory model [2014] both depend on codatatypes to express infinite processes.

Codatatypes are freely generated by their constructors, but in contrast with datatypes, infinite constructor terms are also legitimate values for codatatypes (Section 2). The values of a codatatype consist of all well-typed finite *and infinite* variable-free constructor terms, and only those. As a simple example, the codatatype specification

$$\textbf{codata}\ \mathit{enat} = \mathsf{Z} \mid \mathsf{S}(\mathit{enat})$$

(using an ML- or Haskell-like syntax) introduces a type that captures the natural numbers Z, $\mathsf{S}(\mathsf{Z})$, $\mathsf{S}(\mathsf{S}(\mathsf{Z}))$, …, in unary notation, extended with an infinite value $\infty = \mathsf{S}(\mathsf{S}(\mathsf{S}(\ldots)))$. The equation $\mathsf{S}(\infty) \approx \infty$ holds as expected, because both sides expand to the infinite term $\mathsf{S}(\mathsf{S}(\mathsf{S}(\ldots)))$, i.e., ∞.

Datatypes and codatatypes are an integral part of many proof assistants, including Coq, Isabelle, and PVS. In recent years, datatypes have emerged in a few automatic theorem provers as well. In this paper, we present a unified decision procedure for universal problems involving datatypes and codatatypes in combination (Section 3). The procedure is described abstractly as a calculus and can be composed with other theories in an SMT solver. It generalizes the procedure by Barrett *et al.* [2007], which covers only datatypes.

Datatypes and codatatypes share many properties, so it makes sense to consider them together. There are, however, at least three important differences. First, *codatatypes need not be well-founded.* For example, the type

$$\textbf{codata}\ \mathit{stream}_\tau = \mathsf{SCons}(\tau, \mathit{stream}_\tau)$$

of infinite sequences or streams over an element type τ is allowed, even though it has no base case. Second, *a uniqueness rule takes the place of the acyclicity rule of datatypes.* Cyclic constraints such as $x \approx \mathsf{S}(x)$ are unsatisfiable for datatypes, thanks to an acyclicity rule, but satisfiable for codatatypes. For the latter, a uniqueness rule ensures that two values having the same infinite expansion are equal; from $x \approx \mathsf{S}(x)$ and $y \approx \mathsf{S}(y)$, it deduces $x \approx y$. These two rules cannot be finitely axiomatized, so they naturally belong in a decision procedure. Third, *it must be possible to express cyclic values as closed terms.* For this, we use the μ-binder notation to give a name to a repeated subterm. Thus, the μ-term $\mathsf{SCons}(1, \mu s.\ \mathsf{SCons}(0, \mathsf{SCons}(9, s)))$ stands for the sequence $1,0,9,0,9,0,9,\ldots$.

Our procedure is implemented in the SMT solver CVC4 [Barrett *et al.*, 2011]. It consists of about 2000 lines of C++.

*In memoriam Morgan Deters 1979–2015

An evaluation on problems generated from Isabelle formalizations demonstrates its usefulness (Section 4).

The original version of this paper was presented at the 25th Conference on Automated Deduction (CADE-25) in Berlin, Germany [Reynolds and Blanchette, 2015]. An more comprehensive article will appear in a special issue of the *J. Autom. Reasoning*. We refer to these for a description of related work.

2 (Co)datatypes

Our setting is a monomorphic (many-sorted) first-order logic. We fix a signature consisting of a set of types and a set of function symbols. The types are partitioned into the *datatypes* $\mathcal{Y}_{\mathrm{dt}}$, the *codatatypes* $\mathcal{Y}_{\mathrm{codt}}$, and the are the remaining *ordinary types* $\mathcal{Y}_{\mathrm{ord}}$. The function symbols are partitioned into the *constructors* $\mathcal{F}_{\mathrm{ctr}}$ and the *selectors* $\mathcal{F}_{\mathrm{sel}}$. There is no need to consider further function symbols because they can be abstracted away as variables when combining theories.

In an SMT problem, the signature is typically given by specifying first the uninterpreted types in any order, then the (co)datatypes with their constructors and selectors in groups of ℓ mutually recursive datatypes or corecursive codatatypes, and finally any other function symbols.

Each (co)datatype δ is equipped with $m \geq 1$ constructors, and each constructor takes zero or more arguments and returns a δ value. To every argument corresponds a selector. The names for the (co)datatypes, the constructors, and the selectors must be fresh. Schematically:

$$\begin{aligned}(\mathbf{co})\mathbf{data}\ \delta_1 &= \mathsf{C}_{11}(\left[\mathsf{s}^1_{11}{:}\right]\tau^1_{11},\ldots,\left[\mathsf{s}^{n_{11}}_{11}{:}\right]\tau^{n_{11}}_{11}) \mid \cdots \mid \mathsf{C}_{1m_1}(\ldots)\\ &\vdots\\ \mathbf{and}\ \delta_\ell &= \mathsf{C}_{\ell 1}(\ldots) \mid \cdots \mid \mathsf{C}_{\ell m_\ell}(\ldots)\end{aligned}$$

with $\mathsf{C}_{ij} : \tau^1_{ij} \times \cdots \times \tau^{n_{ij}}_{ij} \to \delta_i$ and $\mathsf{s}^k_{ij} : \delta_i \to \tau^k_{ij}$. The δ constructors and selectors are denoted by $\mathcal{F}^\delta_{\mathrm{ctr}}$ and $\mathcal{F}^\delta_{\mathrm{sel}}$. For types with several constructors, it is useful to provide discriminators $\mathsf{d}_{ij} : \delta_i \to \mathit{bool}$. We let $\mathsf{d}_{ij}(t)$ be an abbreviation for $t \approx \mathsf{C}_{ij}(\mathsf{s}^1_{ij}(t),\ldots,\mathsf{s}^{n_{ij}}_{ij}(t))$.

Datatypes and codatatypes share many basic properties. All properties below are implicitly universally quantified and range over all i, j, j', and k within bounds:

$$\begin{aligned}\text{Distinctness:}\quad & \mathsf{C}_{ij}(\bar{x}) \not\approx \mathsf{C}_{ij'}(\bar{y}) \quad \text{if } j \neq j'\\ \text{Injectivity:}\quad & \mathsf{C}_{ij}(x_1,\ldots,x_{n_{ij}}) \approx \mathsf{C}_{ij}(y_1,\ldots,y_{n_{ij}}) \longrightarrow x_k \approx y_k\\ \text{Exhaustiveness:}\quad & \mathsf{d}_{i1}(x) \vee \cdots \vee \mathsf{d}_{im_i}(x)\\ \text{Selection:}\quad & \mathsf{s}^k_{ij}(\mathsf{C}_{ij}(x_1,\ldots,x_{n_{ij}})) \approx x_k\end{aligned}$$

Datatypes are additionally characterized by an induction principle. For the natural numbers constructed from Z and S, induction prohibits infinite values $\mathsf{S}(\mathsf{S}(\ldots))$. For codatatypes, the dual notion is called coinduction: Two values that yield the same observations must be equal, where the observations are made through selectors and discriminators. Codatatypes are also guaranteed to contain all values corresponding to infinite variable-free constructor terms.

3 The Decision Procedure

Given a fixed signature, the decision procedure for the universal theory of (co)datatypes determines the satisfiability of finite sets E of constraints: equalities ($\approx$) and disequalities ($\not\approx$) between first-order terms, whose variables are interpreted existentially. The decision procedure is formulated as an abstract calculus. Proving a universal quantifier-free conjecture is reduced to showing that its negation is unsatisfiable.

To simplify the presentation, we make a few assumptions about the signature. First, all codatatypes are corecursive. This is reasonable because noncorecursive codatatypes can be seen as nonrecursive datatypes. Second, all ordinary types have infinite cardinality. Without quantifiers, the constraints E cannot entail an upper bound on the cardinality of any uninterpreted type, so it is safe to consider these types infinite. As for ordinary types interpreted finitely by other theories (e.g., bit vectors), each interpreted type having finite cardinality n can be viewed as a datatype with n nullary constructors.

Our calculus for the theory of (co)datatypes consists of derivation rules. A derivation rule can be applied to E if its premises are met. The conclusion either specifies an equality to be added to E or is $\bot$ (contradiction). One of the rules has multiple conclusions separated by $\|$, denoting branching. An application of a rule is *redundant* if one of its non-$\bot$ conclusions leaves E unchanged. A *derivation tree* is a tree whose nodes are finite sets of equalities, such that child nodes are obtained by a nonredundant application of a derivation rule to the parent. A derivation tree is *closed* if all of its leaf nodes are $\bot$. A node is *saturated* if no nonredundant instance of a rule can be applied to it.

The derivation rules are partitioned into three phases, given in Figures 1, 2, and 3. The first phase computes the bidirectional closure of E. The second phase makes inferences based on acyclicity (for datatypes) and uniqueness (for codatatypes). The third phase performs case distinctions on constructors for various terms occurring in E. The rules belonging to a phase have priority over those of subsequent phases. The rules are applied until the derivation tree is closed or all leaf nodes are saturated.

3.1 Phase 1: Computing the Bidirectional Closure

In conjunction with Refl, Sym, and Trans, the Cong rule computes the congruence (upward) closure, whereas the Inject and Clash rules compute the unification (downward) closure. For unification, equalities are inferred based on the injectivity of constructors by Inject, and failures to unify equated terms are recognized by Clash. Conflict recognizes when an equality and

$$\frac{t \in \mathcal{T}(E)}{t \approx t \in E}\ \textsf{Refl} \qquad \frac{t \approx u \in E}{u \approx t \in E}\ \textsf{Sym} \qquad \frac{s \approx t,\ t \approx u \in E}{s \approx u \in E}\ \textsf{Trans}$$

$$\frac{\bar{t} \approx \bar{u} \in E \quad \mathsf{f}(\bar{t}), \mathsf{f}(\bar{u}) \in \mathcal{T}(E)}{\mathsf{f}(\bar{t}) \approx \mathsf{f}(\bar{u}) \in E}\ \textsf{Cong}$$

$$\frac{t \approx u,\ t \not\approx u \in E}{\bot}\ \textsf{Conflict} \qquad \frac{\mathsf{C}(\bar{t}) \approx \mathsf{C}(\bar{u}) \in E}{\bar{t} \approx \bar{u} \in E}\ \textsf{Inject}$$

$$\frac{\mathsf{C}(\bar{t}) \approx \mathsf{D}(\bar{u}) \in E \quad \mathsf{C} \neq \mathsf{D}}{\bot}\ \textsf{Clash}$$

Figure 1: Derivation rules for bidirectional closure

its negation both occur in E, in which case E is unsatisfiable.

Let $\mathcal{T}(E)$ denote the set of terms occurring in E. At the end of the first phase, E induces an equivalence relation over $\mathcal{T}(E)$ such that two terms t and u are equivalent if and only if $t \approx u \in E$. Thus, we can regard E as a set of equivalence classes of terms. For a term $t \in \mathcal{T}(E)$, we write $[t]$ to denote the equivalence class of t in E. Moreover, at the end of this phase, each equivalence class $[t]$ contains at most one constructor term that is unique up to congruence. Thus, in the subsequent phases, when considering the case that $[t]$ contains constructor terms, it is enough to select an arbitrary constructor term from $[t]$ among these.

3.2 Phase 2: Applying Acyclicity and Uniqueness

The rules in this phase are described in terms of a mapping $\mathcal{A}$ that assigns to each equivalence class a μ-term.

Formally, *μ-terms* are defined recursively as being either a variable x or an applied constructor $\mu x.\mathsf{C}(\bar{t})$ for some $\mathsf{C} \in \mathcal{F}_{\text{ctr}}$ and μ-terms $\bar{t}$ of the expected types. The variable x need not occur free in the μ-binder's body, in which case the binder can be omitted. $\text{FV}(t)$ denotes the set of free variables occurring in the μ-term t. A μ-term is *closed* if it contains no free variables. It is *cyclic* if it contains a bound variable. The *α-equivalence* relation $t =_\alpha u$ indicates that the μ-terms t and u are syntactically equivalent for some capture-avoiding renaming of μ-bound variables—e.g., $\mu x.\,\mathsf{D}(y,x) =_\alpha \mu z.\,\mathsf{D}(y,z)$, but $\mu x.\,\mathsf{C}(x)$, $\mu x.\,\mathsf{D}(y,x)$, $\mu x.\,\mathsf{D}(z,x)$, and $\mu y.\,\mathsf{D}(y,x)$ are all α-disequivalent. Two μ-terms can denote the same value despite being α-disequivalent—e.g., $\mu x.\,\mathsf{S}(x) \neq_\alpha \mu y.\,\mathsf{S}(\mathsf{S}(y))$.

The mapping $\mathcal{A}$ is constructed as follows. With each equivalence class $[u]$, we associate a fresh variable $\widetilde{u}$ of the same type as u. For a term $t \in \mathcal{T}(E)$, we write $\widetilde{t}$ to denote the variable associated with the equivalence class $[t]$. Initially, we set $\mathcal{A}[u] := \widetilde{u}$ for each equivalence class $[u]$. Because $\widetilde{u}$ is unconstrained, this indicates that there are initially no constraints on the values for any equivalence class $[u]$. The mapping $\mathcal{A}$ is refined by applying the following unfolding rule exhaustively:

$$\frac{\widetilde{u} \in \text{FV}(\mathcal{A}) \quad \mathsf{C}(t_1,\ldots,t_n) \in [u] \quad \mathsf{C} \in \mathcal{F}_{\text{ctr}}}{\mathcal{A} := \mathcal{A}[\widetilde{u} \mapsto \mu\widetilde{u}.\,\mathsf{C}(\widetilde{t_1},\ldots,\widetilde{t_n})]}$$

$\text{FV}(\mathcal{A})$ denotes the set of free variables occurring in $\mathcal{A}$'s range, and $\mathcal{A}[x \mapsto t]$ denotes the variable-*capturing* substitution of t for x in $\mathcal{A}$'s range. It is easy to see that the height of terms produced as a result of the unfolding is bounded by the number of equivalence classes of E, and thus the construction of $\mathcal{A}$ will terminate.

The μ-term $\mathcal{A}[t]$ describes a class of values that t and other members of t's equivalence class can take in models of E. When τ is a datatype, a cyclic μ-term describes an infeasible class of values.

$$\frac{t : \delta \in \mathcal{Y}_{\text{dt}} \quad \mathcal{A}[t] = \mu x.\,u \quad x \in \text{FV}(u)}{\bot}\ \textsf{Acyclic}$$

$$\frac{t,u : \delta \in \mathcal{Y}_{\text{codt}} \quad \mathcal{A}[t] =_\alpha \mathcal{A}[u]}{t \approx u \in E}\ \textsf{Unique}$$

Figure 2: Derivation rules for acyclicity and uniqueness

Example 1. Suppose that E contains four distinct equivalence classes $[w]$, $[x]$, $[y]$, and $[z]$ such that $\mathsf{C}(w,y) \in [x]$ and $\mathsf{C}(z,x) \in [y]$ for some $\mathsf{C} \in \mathcal{F}_{\text{ctr}}$. A possible sequence of unfolding steps is given below, omitting trivial entries $[t] \mapsto \widetilde{t}$.

1. Unfold $\widetilde{x}$: $\mathcal{A} = \{[x] \mapsto \mu\widetilde{x}.\,\mathsf{C}(\widetilde{w},\widetilde{y})\}$
2. Unfold $\widetilde{y}$: $\mathcal{A} = \{[x] \mapsto \mu\widetilde{x}.\,\mathsf{C}(\widetilde{w},\mu\widetilde{y}.\,\mathsf{C}(\widetilde{z},\widetilde{x})),$ $[y] \mapsto \mu\widetilde{y}.\,\mathsf{C}(\widetilde{z},\widetilde{x})\}$
3. Unfold $\widetilde{x}$: $\mathcal{A} = \{[x] \mapsto \mu\widetilde{x}.\,\mathsf{C}(\widetilde{w},\mu\widetilde{y}.\,\mathsf{C}(\widetilde{z},\widetilde{x})),$ $[y] \mapsto \mu\widetilde{y}.\,\mathsf{C}(\widetilde{z},\mu\widetilde{x}.\,\mathsf{C}(\widetilde{w},\widetilde{y}))\}$

The resulting $\mathcal{A}$ indicates that the values for x and y in models of E must be of the forms $\mathsf{C}(\widetilde{w},\mathsf{C}(\widetilde{z},\mathsf{C}(\widetilde{w},\mathsf{C}(\widetilde{z},\ldots))))$ and $\mathsf{C}(\widetilde{z},\mathsf{C}(\widetilde{w},\mathsf{C}(\widetilde{z},\mathsf{C}(\widetilde{w},\ldots))))$, respectively. ∎

Given the mapping $\mathcal{A}$, the Acyclic and Unique rules work as follows. For acyclicity, if $[t]$ is a datatype equivalence class whose values $\mathcal{A}[t] = \mu x.\,u$ are cyclic (expressed by $x \in \text{FV}(u)$), then E is unsatisfiable. For uniqueness, if $[t]$, $[u]$ are two codatatype equivalence classes whose values $\mathcal{A}[t]$, $\mathcal{A}[u]$ are α-equivalent, then t is equal to u. Comparison for α-equivalence may seem too restrictive, since $\mu x.\,\mathsf{S}(x)$ and $\mu y.\,\mathsf{S}(\mathsf{S}(y))$ specify the same value despite being α-disequivalent, but the rule will make progress by discovering that the subterm $\mathsf{S}(y)$ of $\mu y.\,\mathsf{S}(\mathsf{S}(y))$ must be equal to the entire term, as demonstrated next.

Example 2. Let $E = \{x \approx \mathsf{S}(x),\ y \approx \mathsf{S}(\mathsf{S}(y))\}$. After phase 1, the equivalence classes are $\{x, \mathsf{S}(x)\}$, $\{y, \mathsf{S}(\mathsf{S}(y))\}$, and $\{\mathsf{S}(y)\}$. Constructing $\mathcal{A}$ yields

$$\mathcal{A}[x] = \mu\widetilde{x}.\,\mathsf{S}(\widetilde{x}) \qquad \mathcal{A}[y] = \mu\widetilde{y}.\,\mathsf{S}(\mu\widetilde{\mathsf{S}(y)}.\,\mathsf{S}(\widetilde{y}))$$
$$\mathcal{A}[\mathsf{S}(y)] = \mu\widetilde{\mathsf{S}(y)}.\,\mathsf{S}(\mu\widetilde{y}.\,\mathsf{S}(\widetilde{\mathsf{S}(y)}))$$

Since $\mathcal{A}[y] =_\alpha \mathcal{A}[\mathsf{S}(y)]$, the Unique rule applies to derive $y \approx \mathsf{S}(y)$. At this point, phase 1 is activated again, yielding the equivalence classes $\{x, \mathsf{S}(x)\}$ and $\{y, \mathsf{S}(y), \mathsf{S}(\mathsf{S}(y))\}$. The mapping $\mathcal{A}$ is updated accordingly:

$$\mathcal{A}[x] = \mu\widetilde{x}.\,\mathsf{S}(\widetilde{x}) \qquad \mathcal{A}[y] = \mu\widetilde{y}.\,\mathsf{S}(\widetilde{y})$$

Since $\mathcal{A}[x] =_\alpha \mathcal{A}[y]$, Unique can finally derive $x \approx y$. ∎

3.3 Phase 3: Branching

If a selector is applied to a term t, or if t's type is a finite datatype, t's equivalence class must contain a constructor term. This is enforced in the third phase by the Split rule. Another rule, Single, focuses on the degenerate case where two terms are of a *singleton* type (one for which there exists only one value), and are therefore equal. Notice that corecursive singleton types may have infinite values. A simple example is

$$\frac{t : \delta \quad t \in \mathcal{T}(E) \quad \mathcal{F}^{\delta}_{\text{ctr}} = \{\mathsf{C}_1,\ldots,\mathsf{C}_m\} \quad \big(\mathsf{s}(t) \in \mathcal{T}(E) \text{ and } \mathsf{s} \in \mathcal{F}^{\delta}_{\text{sel}}\big) \text{ or } \big(\delta \in \mathcal{Y}_{\text{dt}} \text{ and } \delta \text{ is finite}\big)}{\big\|_{i=1}^{n}\, t \approx \mathsf{C}_i\big(\mathsf{s}_i^1(t),\ldots,\mathsf{s}_i^{n_i}(t)\big) \in E}\ \textsf{Split}$$

$$\frac{t,u : \delta \in \mathcal{Y}_{\text{codt}} \quad t,u \in \mathcal{T}(E) \quad \delta \text{ is a singleton}}{t \approx u \in E}\ \textsf{Single}$$

Figure 3: Derivation rules for branching

codata $a = \mathsf{A}(a)$, which is corecursive and yet has a cardinality of one; its unique value is $\mu a.\, \mathsf{A}(a)$. Both Split's finiteness assumption and Single's singleton constraint can be evaluated statically based on a recursive computation of the cardinalities of the constructors' argument types.

3.4 Termination and Correctness

We proved the following properties of derivation trees generated by the calculus [Reynolds and Blanchette, 2015].

Theorem 1 (Termination). *All derivation trees are finite.*

Theorem 2 (Refutation Soundness). *If there exists a closed derivation tree with root node E, then E is unsatisfiable.*

Theorem 3 (Solution Soundness). *If there exists a derivation tree with root node E containing a saturated node, then E is satisfiable.*

By Theorems 1, 2, and 3, the calculus is sound and complete for the universal theory of (co)datatypes. We may rightly call it a decision procedure for that theory. The proof of solution soundness provides a method for constructing a model for a saturated configuration.

4 Evaluation

The decision procedure for (co)datatypes is useful both for proving (via negation, in the refutational style) and for model finding [Ge and de Moura, 2009; Reynolds *et al.*, 2013]. It is in fact vital for finite model finding, because the acyclicity and uniqueness rules are necessary for solution soundness, without which the generated models would often be spurious. For example, given the constraints

$$\mathsf{zeros} \approx \mathsf{SCons}(0, \mathsf{zeros}) \quad \mathsf{repeat}(n) \approx \mathsf{SCons}(n, \mathsf{repeat}(n))$$

the conjecture $\mathsf{zeros} \approx \mathsf{repeat}(0)$ would be "refuted" by a spurious countermodel that interprets zero and $\mathsf{repeat}(0)$ by two distinct values $\mu s.\, \mathsf{SCons}(0, s)$, violating uniqueness.

By contrast, the contributions of the decision procedure to proving are less obvious; they depend on how often acyclicity and uniqueness are necessary for a proof. To evaluate this, we generated benchmark problems from existing interactive proof goals arising in existing Isabelle formalizations, using Sledgehammer [Blanchette *et al.*, 2013] as translator. We included all the formalizations from the Isabelle distribution (Distro, 1179 goals) and the *Archive of Formal Proofs* (AFP, 3014 goals) that define codatatypes falling within the supported fragment. We also included formalizations about Bird and Stern–Brocot trees (SBT, 265 goals). To exercise the datatype support, formalizations about finite lists and trees were added to the first two benchmark sets.

For each proof goal in each formalization, we used Sledgehammer to select either 16 or 256 lemmas, which were monomorphized and translated to SMT-LIB along with the goal. The resulting problem was given to the development version of CVC4 (from 15 September 2015) and to Z3 4.3.2 for comparison, each running for up to 60 s. Problems not involving any (co)datatypes were left out.

CVC4 was run on each problem several times, with the support for datatypes and codatatypes either enabled or disabled. The contributions of the acyclicity and uniqueness rules were also measured, by selectively enabling or disabling the rules. Even when the decision procedure is disabled, the problems may contain basic lemmas about constructors and selectors, allowing some (co)datatype reasoning. This is especially true for problems generated using 256 lemmas. The problems with 16 lemmas put more stress on the decision procedure but are less typical of Sledgehammer problems.

	$n = 16$		$n = 256$	
	CVC4	Z3	CVC4	Z3
No (co)datatypes	1099	1097	2209	1911
Datatypes without Acyclic	1116	–	2211	–
Full datatypes	1120	1121	2211	1901
Codatatypes without Unique	1132	–	2208	–
Full codatatypes	1137	–	**2220**	–
Full (co)datatypes	**1157**	–	2219	–

Table 4: Number of solved goals with n lemmas per goal

The results are summarized in Table 4. For the 16-lemma problems, it accounts for an overall success rate increase of over 5%. Moreover, every aspect of the procedure, including the more expensive rules, makes a contribution. For the 256-lemma problems, the difference is much smaller, at 0.5%. The table indicates that the theoretically stronger instances of the decision procedure do not always subsume the weaker ones in practice. The raw data reveal that the full procedure proved 27 goals that could not be proved without it, but failed for 17 goals that could be proved without it.

5 Conclusion

We presented a decision procedure for the universal theory of datatypes and codatatypes. Our approach relies on μ-terms to represent cyclic values. Although this aspect is primarily motivated by codatatypes, it makes a uniform account of datatypes and codatatypes possible—in particular, the acyclicity rule for datatypes exploits μ-terms to detect cycles. The empirical results on Isabelle benchmarks confirm that CVC4's new capabilities improve the state of the art.

This work is part of a wider program that aims at enriching automatic provers with high-level features and at reducing the gap between automatic and interactive theorem proving. We are currently interfacing CVC4's finite model finding capabilities for generating counterexamples in proof assistants [Reynolds *et al.*, 2016]. The acyclicity and uniqueness rules are crucial to exclude spurious counterexamples.

Acknowledgment

We owe a great debt to the development team of CVC4, including Clark Barrett and Cesare Tinelli, and in particular Morgan Deters, who jointly with the first author developed the initial version of the decision procedure for datatypes in CVC4. We also want to thank everyone who helped us with the longer versions of this paper. The second author's work was partially supported by the Deutsche Forschungsgemeinschaft project "Den Hammer härten" (grant NI 491/14-1) and the Inria technological development action "Contre-exemples utilisables par Isabelle et Coq" (CUIC).

References

[Barrett *et al.*, 2007] Clark Barrett, Igor Shikanian, and Cesare Tinelli. An abstract decision procedure for satisfiability in the theory of inductive data types. *J. Satisf. Boolean Model. Comput.*, 3:21–46, 2007.

[Barrett *et al.*, 2011] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification* (*CAV 2011*), volume 6806 of *LNCS*, pages 171–177. Springer, 2011.

[Blanchette *et al.*, 2013] Jasmin Christian Blanchette, Sascha Böhme, and Lawrence C. Paulson. Extending Sledgehammer with SMT solvers. *J. Autom. Reasoning*, 51(1):109–128, 2013.

[De Moura and Bjørner, 2011] Leonardo De Moura and Nikolaj Bjørner. Satisfiability modulo theories: Introduction and applications. *Commun. ACM*, 54(9):69–77, 2011.

[Ge and de Moura, 2009] Yeting Ge and Leonardo de Moura. Complete instantiation for quantified formulas in satisfiability modulo theories. In Ahmed Bouajjani and Oded Maler, editors, *Computer Aided Verification* (*CAV 2009*), volume 5643 of *LNCS*, pages 306–320. Springer, 2009.

[Leroy, 2009] Xavier Leroy. A formally verified compiler back-end. *J. Autom. Reasoning*, 43(4):363–446, 2009.

[Lochbihler, 2014] Andreas Lochbihler. Making the Java memory model safe. *ACM Trans. Program. Lang. Syst.*, 35(4):12:1–65, 2014.

[Nieuwenhuis *et al.*, 2006] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving SAT and SAT modulo theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). *J. ACM*, 53(6):937–977, 2006.

[Reynolds and Blanchette, 2015] Andrew Reynolds and Jasmin Christian Blanchette. A decision procedure for (co)datatypes in SMT solvers. In Amy Felty and Aart Middeldorp, editors, *Conference on Automated Deduction* (*CADE-25*), volume 9195 of *LNCS*, pages 197–213. Springer, 2015.

[Reynolds *et al.*, 2013] Andrew Reynolds, Cesare Tinelli, Amit Goel, Sava Krstić, Morgan Deters, and Clark Barrett. Quantifier instantiation techniques for finite model finding in SMT. In Maria Paola Bonacina, editor, *Conference on Automated Deduction* (*CADE-24*), volume 7898 of *LNCS*, pages 377–391. Springer, 2013.

[Reynolds *et al.*, 2016] Andrew Reynolds, Jasmin Christian Blanchette, Simon Cruanes, and Cesare Tinelli. Model finding for recursive functions in SMT. In Nicola Olivetti and Ashish Tiwari, editors, *International Joint Conference on Automated Reasoning* (*IJCAR 2016*), LNCS. Springer, 2016.