A Decision Procedure for (Co)datatypes in SMT Solvers

Andrew Reynolds, Jasmin Christian Blanchette

Noname manuscript No.
(will be inserted by the editor)

A Decision Procedure for (Co)datatypes in SMT Solvers

Andrew Reynolds · **Jasmin Christian Blanchette**

the date of receipt and acceptance should be inserted later

Abstract We present a decision procedure that combines reasoning about datatypes and codatatypes. The dual of the acyclicity rule for datatypes is a uniqueness rule that identifies observationally equal codatatype values, including cyclic values. The procedure decides universal problems and is composable via the Nelson–Oppen method. It has been implemented in CVC4, a state-of-the-art SMT solver. An evaluation based on problems generated from formalizations developed with Isabelle demonstrates the potential of the procedure.

1 Introduction

Freely generated algebraic datatypes are ubiquitous in functional programs and formal specifications. They are especially useful to represent finite data structures in computer science applications but also arise in formalized mathematics. They can be implemented efficiently and enjoy properties that can be exploited in automated reasoners.

To represent infinite objects, a natural choice is to turn to coalgebraic datatypes, or *codatatypes*, the non-well-founded dual of algebraic *datatypes*. Despite their reputation for being esoteric, codatatypes have a role to play in computer science. The verified C compiler CompCert [24], the verified Java compiler JinjaThreads [25], and the formalized Java memory model [26] all depend on codatatypes to capture 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). Intuitively, the values of a codatatype consist of all well-typed finite *and infinite* ground constructor terms, and only those. As a simple example, the coalgebraic specification

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

In memoriam Morgan Deters 1979–2015

Andrew Reynolds
École Polytechnique Fédérale de Lausanne (EPFL), Switzerland

Jasmin Christian Blanchette
Inria & LORIA, Nancy, France
Max-Planck-Institut für Informatik, Saarbrücken, Germany

introduces a type that models the natural numbers Z, $\mathsf{S}(\mathsf{Z})$, $\mathsf{S}(\mathsf{S}(\mathsf{Z}))$, …, in Peano 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 $\mathsf{S}(\mathsf{S}(\mathsf{S}(\ldots)))$, which uniquely identifies ∞.

Datatypes and codatatypes are an integral part of many proof assistants, including Agda, Coq, Isabelle, Matita, and PVS. In recent years, datatypes have emerged in a few automatic theorem provers as well. The SMT-LIB [2] format, supported by most SMT (satisfiability modulo theories) solvers, is being extended with a syntax for datatypes. In this article, we introduce a unified decision procedure for universal problems involving datatypes and codatatypes in combination (Section 3). The procedure is described abstractly as a calculus and is composable via the Nelson–Oppen method [29]. It generalizes the procedure by Barrett et al. [3], 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

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

of infinite sequences or streams over an element type τ is allowed, even though it has no base case. By contrast, the corresponding datatype specification

$$\textbf{datatype}\ \mathit{fstream}_\tau = \mathsf{FSCons}(\tau, \mathit{fstream}_\tau)$$

would be rejected as non-well-founded [9]. 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 are needed to ensure completeness (solution soundness) on universal problems. They cannot be finitely axiomatized, so they naturally belong in a decision procedure. Third, *it must be possible to express cyclic* (*regular*) *values as closed terms and to enumerate them.* This is necessary when generating models. The μ-binder notation associates a name with a subterm; it is used to represent cyclic values in the metatheory and in the generated models. For example, the μ-term $\mathsf{SCons}(1, \mu s.\, \mathsf{SCons}(0, \mathsf{SCons}(9, s)))$ stands for the lasso-shaped sequence $1,0,9,0,9,\ldots$.

Our procedure is implemented in the SMT solver CVC4 [1] as a combination of rewriting and a theory solver (Section 4). It consists of about 2000 lines of C++ code, most of which are shared between datatypes and codatatypes. The code is integrated in the development version of the solver and is expected to be part of the CVC4 1.5 release. An evaluation on problems generated from Isabelle/HOL [30] formalizations using the Sledgehammer tool [5] demonstrates the usefulness of the approach (Section 5).

An earlier version of this article was presented at the CADE-25 conference in Berlin, Germany [32]. This article extends the conference paper with additional background material on (co)datatypes (Section 2), more detailed metatheoretical proofs (Section 3), a description of the generation of models with μ-binders in CVC4 (Section 4), and a more comprehensive evaluation (Section 5).

Related Work. Barrett et al. [3] provide a good account of related work on datatypes as of 2007, in addition to describing their implementation in CVC3. Since then, datatypes have been added not only to CVC4 (a complete rewrite of CVC3) but also to the SMT solver Z3 [28] in unpublished work by Leonardo de Moura and to a SPASS-like superposition prover by Wand [41]. In his Ph.D. thesis [4], Bjørner introduced a decision procedure for

(co)datatypes in STeP, the Stanford Temporal Prover. Closely related are the automatic structural induction in SMT solvers [34] and superposition provers [11,20], the (co)datatype and (co)induction support in Dafny [23], and the (semi-)decision procedures for datatypes implemented in Leon [39] and RADA [31]. Datatypes are supported by the higher-order model finder Refute [42] for Isabelle. Its successor, Nitpick [7], can also generate models involving cyclic codatatype values. Cyclic values have been studied extensively under the heading of regular or rational trees—see Carayol and Morvan [10] and Djellou et al. [12] for recent work. The μ-notation is inspired by the μ-calculus [13,22].

Conventions. Our setting is a monomorphic (or many-sorted) first-order logic. A signature $\Sigma = (\mathcal{Y}, \mathcal{F})$ consists of a set of types $\mathcal{Y}$ and a set of function symbols $\mathcal{F}$. Types are atomic sorts and interpreted as nonempty domains. The set $\mathcal{Y}$ must contain a distinguished type *bool* interpreted as the set of truth values. The metavariables δ, ε range over (co)datatypes, whereas τ, υ range over arbitrary types.

Function symbols are written in a sans-serif font (e.g., f, g) to distinguish them from variables (e.g., x, y). Symbol names starting with an uppercase letter (e.g., S) are reserved for constructors. With each function symbol f is associated a list of argument types $\tau_1, \ldots, \tau_n$ (for $n \geq 0$) and a return type τ, written $\mathsf{f} : \tau_1 \times \cdots \times \tau_n \to \tau$; this notation collapses to $\mathsf{f} : \tau$ if $n = 0$. Functions invocations $\mathsf{f}(t_1, \ldots, t_n)$ apply the n-ary function symbol f to n arguments $t_1 : \tau_1, \ldots, t_n : \tau_n$ of the right types. Nullary function symbols, also called constants, can appear without parentheses in terms. The set $\mathcal{F}$ must at least contain $\mathsf{true}, \mathsf{false} : \mathit{bool}$, interpreted as truth values. The only predicate is equality, written $\approx$; it belongs to the logical symbols. Other predicates can be represented as functions to *bool*, with $\mathsf{p}(\ldots)$ abbreviating $\mathsf{p}(\ldots) \approx \mathsf{true}$. The notation t^τ stands for a term t of type τ; it should not be confused with $t : \tau$, which is a statement expressing that t has type τ. When applied to terms, the standard equality symbol $=$ denotes syntactic equality. The operator $\bigwedge_i \varphi_i$ abbreviates a conjunction $\varphi_1 \wedge \cdots \wedge \varphi_n$. Finally, $\bar{x}$ abbreviates a list or tuple $x_1, \ldots, x_n$.

2 (Co)datatypes

We fix a signature $\Sigma = (\mathcal{Y}, \mathcal{F})$. The types are partitioned into $\mathcal{Y} = \mathcal{Y}_{\mathrm{dt}} \uplus \mathcal{Y}_{\mathrm{codt}} \uplus \mathcal{Y}_{\mathrm{ord}}$, where $\mathcal{Y}_{\mathrm{dt}}$ are the *datatypes*, $\mathcal{Y}_{\mathrm{codt}}$ are the *codatatypes*, and $\mathcal{Y}_{\mathrm{ord}}$ are the remaining *ordinary types* (which can be interpreted or not). The function symbols are partitioned into $\mathcal{F} = \mathcal{F}_{\mathrm{ctr}} \uplus \mathcal{F}_{\mathrm{sel}}$, where $\mathcal{F}_{\mathrm{ctr}}$ are the *constructors* and $\mathcal{F}_{\mathrm{sel}}$ are the *selectors*. There is no need to consider further function symbols because they can be abstracted away as variables when combining theories. Exceptionally, it is convenient to use numeric constants (0, 1, ...) in examples. Σ-terms are standard first-order terms over Σ, without μ-binders.

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 mutually (co)recursive (co)datatypes, and finally any other function symbols. Each (co)datatype specification consists of ℓ mutually recursive types that are either all datatypes or all codatatypes. Polymorphic types, nested (co)recursion, and datatype–codatatype mixtures fall outside this fragment.[1] We allow ourselves some notational parameteriza-

[1] In principle, rank-1 (top-level) polymorphism [8] does not raise any special difficulties. Nesting datatypes inside datatypes, and likewise for codatatypes, can be reduced to the mutual case [17]. So the only genuinely interesting cases missing are mixed nested (co)recursion as well as (co)recursion through a non-(co)datatype (both of which make sense, at least in a higher-order setting [6]).

tion through subscripts—for example, stream_τ denotes a family of ground types including $\mathit{stream}_{\mathit{int}}$, $\mathit{stream}_{\mathit{bool}}$, and $\mathit{stream}_{\mathit{stream}_{\mathit{real}}}$.

Each (co)datatype δ is equipped with $m \geq 1$ constructors, and each constructor for δ takes zero or more arguments and returns a δ value. The argument types must be either ordinary, among the already known (co)datatypes, or among the (co)datatypes being introduced. To every argument corresponds a selector. The names for the (co)datatypes, the constructors, and the selectors must be distinct and different from existing names.[2] Schematically:

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

with $\mathsf{C}_{ij} : \tau^1_{ij} \times \cdots \times \tau^{n_{ij}}_{ij} \to \delta_i$ and $\mathsf{s}^k_{ij} : \delta_i \to \tau^k_{ij}$. Defaults are assumed for the selector names if they are omitted. 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 $\mathsf{d}_{ij} : \delta_i \to \mathit{bool}$. Instead of extending $\mathcal{F}$, we let $\mathsf{d}_{ij}(t)$ be an abbreviation for $t \approx \mathsf{C}_{ij}\big(\mathsf{s}^1_{ij}(t),\ldots,\mathsf{s}^{n_{ij}}_{ij}(t)\big)$.

Here are a few examples of legal specifications of (co)datatype families:

$$\begin{aligned}\textbf{datatype}\ \mathit{list}_\tau &= \mathsf{Nil} \mid \mathsf{Cons}(\mathsf{hd}\colon \tau, \mathsf{tl}\colon \mathit{list}_\tau)\\ \textbf{codatatype}\ \mathit{llist}_\tau &= \mathsf{LNil} \mid \mathsf{LCons}(\mathsf{lhd}\colon \tau, \mathsf{ltl}\colon \mathit{llist}_\tau)\\ \textbf{datatype}\ \mathit{tree}_\tau &= \mathsf{Node}(\tau, \mathit{forest}_\tau)\\ \textbf{and}\ \mathit{forest}_\tau &= \mathsf{FNil} \mid \mathsf{FCons}(\mathit{tree}_\tau, \mathit{forest}_\tau)\end{aligned}$$

Because all types must be inhabited (nonempty), a datatype specification is admissible only if a ground constructor term can be exhibited. This rules out non-well-founded specifications such as that of *fstream* in Section 1. For codatatypes, no admissibility check is necessary because there is always a term, finite or infinite, that witnesses nonemptiness [9].

A type δ depends on another type ε if ε is the type of an argument to one of δ's constructors. Semantically, a set of types is *mutually* (*co*)*recursive* if and only if the associated dependency graph is strongly connected. A type is (*co*)*recursive* if it belongs to such a set of types. Types can be declared together in a mutual fashion even if they are not actually mutually (co)recursive. The semantic notion is more precise and is the one that interests us.

Non(co)recursive type specifications such as either of

$$\begin{aligned}\textbf{datatype}\ \mathit{option}_\tau &= \mathsf{None} \mid \mathsf{Some}(\tau)\\ \textbf{codatatype}\ \mathit{complex} &= \mathsf{Complex}(\mathsf{re}\colon \mathit{real}, \mathsf{im}\colon \mathit{real})\end{aligned}$$

are permitted. At the semantic level, it makes no difference whether such types are introduced as datatypes or as codatatypes.

One way to characterize datatypes is as the initial model of the selector–constructor equations [3]. A drawback of this approach is that it does not naturally account for selectors applied to wrong constructors. Barrett et al. address this by parameterizing the construction by default values, but this gives rise to spurious equalities between unrelated terms. For example, given

$$\textbf{datatype}\ x = \mathsf{C}(\mathsf{s}\colon \mathit{int}) \mid \mathsf{D} \mid \mathsf{E}$$

we would have the spurious equality $\mathsf{s}(\mathsf{D}) \approx \mathsf{s}(\mathsf{E})$. This flaw could be corrected, but the added complexity seems to suggest that selectors are better characterized axiomatically.

[2] It can be convenient to specify the same selector for several constructors associated with the same (co)datatype, as long as the argument types coincide. However, this is disallowed by SMT-LIB, so we do not consider it here.

A related semantic view of datatypes is as initial algebras of suitable functors. Codatatypes are then defined dually as final coalgebras [37]. The datatypes are generated by their constructors, whereas the codatatypes are viewed through their selectors.

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:

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

Expressed in the algebraic jargon, exhaustiveness helps ensure that "no junk" exists, whereas distinctness and injectivity guarantee that "no confusion" can arise. The result of selectors applied to the wrong constructor is left completely unspecified. Datatypes are additionally characterized by an induction principle for proving a conjunction of properties $P_1,\ldots,P_\ell$ over arbitrary values $v_1 : \delta_1,\ldots,v_\ell : \delta_\ell$:

$$\text{Induction:} \quad \frac{\bigwedge_{i=1}^{\ell} \bigwedge_{j=1}^{m_i} \forall x_1 \ldots x_{n_{ij}}.\, \big(\bigwedge_{k=1}^{n_{ij}} \mathit{IH}[x_k]\big) \longrightarrow P_i[\mathsf{C}_{ij}(x_1,\ldots,x_{n_{ij}})]}{\bigwedge_{i=1}^{\ell} P_i[v_i]}$$

The notation $\mathit{IH}[x]$ denotes either $P_{i'}[x]$ if there exists some i' such that the formula is type-correct or else $\top$ (truth). The induction principle ensures that the interpretation of datatypes is standard. For the natural numbers constructed from Z and S, induction prohibits models that contain infinite values $\mathsf{S}(\mathsf{S}(\ldots))$.

For codatatypes, the dual notion is called coinduction. It makes it possible to derive the equality of pairs of codatatype values $v_1, w_1 : \delta_1, \ldots, v_\ell, w_\ell : \delta_\ell$, based on suitable coinduction witnesses $R_1,\ldots,R_\ell$:

$$\text{Coinduction:} \quad \frac{\bigwedge_{i=1}^{\ell} R_i[v_i, w_i] \qquad \bigwedge_{i=1}^{\ell} \big(\forall x\, y.\, R_i[x,y] \longrightarrow \bigwedge_{j=1}^{m_i} \big(\mathsf{d}_j(x) \longrightarrow \mathsf{d}_j(y) \wedge \bigwedge_{k=1}^{n_{ij}} \mathsf{s}_{ij}^k(x) \sim \mathsf{s}_{ij}^k(y)\big)\big)}{\bigwedge_{i=1}^{\ell} v_i \approx w_i}$$

The notation $x \sim y$ stands for $R_{i'}[x,y]$ if there exists some i' such that the formula is type-correct or $x \approx y$ otherwise. The second premise ensures that the coinduction witnesses are bisimulations. The first premise and the conclusion capture the notion that equality is the largest bisimulation on codatatypes. Thus, the coinduction principle encodes a form of extensionality: Two values that yield the same observations must be equal, where the observations are made through selectors and discriminators.

Example 1 The induction principle for list_τ is given below:

$$\frac{P[\mathsf{Nil}] \quad \forall x, xs.\, P[xs] \longrightarrow P[\mathsf{Cons}(x,xs)]}{P[vs]}$$

Assuming that lnull and its negation are the discriminators associated with LNil and LCons, the coinduction principle for llist_τ is

$$\frac{R[vs, ws] \qquad \begin{array}{l} \forall xs\, ys.\, R[xs,ys] \longrightarrow \big(\mathsf{lnull}(xs) \longrightarrow \mathsf{lnull}(ys)\big) \\ \qquad \wedge \big(\neg\, \mathsf{lnull}(xs) \longrightarrow \neg\, \mathsf{lnull}(ys) \wedge \mathsf{hd}(xs) \approx \mathsf{hd}(ys) \wedge R[\mathsf{tl}(xs), \mathsf{tl}(ys)]\big) \end{array}}{vs \approx ws}$$

■

Codatatypes are guaranteed to contain all values corresponding to infinite ground constructor terms. In general, this cannot be captured by a first-order axiomatization, since there may be uncountably many of them. For example, $\mathit{stream}_{\mathit{int}}$ is isomorphic to the uncountable function space $\mathit{nat} \to \mathit{int}$. Bjørner gives a rigorous treatment of this aspect [4].

Given a signature Σ, $\mathcal{DC}$ refers to the *theory of datatypes and codatatypes*, which defines a class of Σ-interpretations $\mathscr{J}$, namely those that satisfy the properties mentioned in this section, including (co)induction. The interpretations in $\mathscr{J}$ share the same interpretation for constructor terms and correctly applied selector terms (up to isomorphism) but may differ on variables and wrongly applied selector terms. A formula is *$\mathcal{DC}$-satisfiable* if there exists an interpretation in $\mathscr{J}$ that satisfies it. For deciding universal formulas, induction can be replaced by the acyclicity axiom schema, which states that constructor terms cannot be equal to any of their proper subterms [3]. Dually, coinduction can be replaced by the uniqueness schema, which asserts that codatatype values are fully characterized by their expansion [37, Theorem 8.1, $2 \Leftrightarrow 5$].

For datatypes, any recursive specification gives rise to an infinite datatype. Paradoxically, this does not extend to codatatypes: Some codatatypes are so degenerate as to be finite even though they have infinite values. A simple example is **codatatype** $a = \mathsf{A}(a)$, which is corecursive and yet has a cardinality of one; its unique value is $\mu a.\, \mathsf{A}(a)$. Other specimens are $\mathit{stream}_{\mathit{unit}}$ and both b and c in the specification

$$\begin{aligned} &\textbf{codatatype}\ b = \mathsf{B}(b, c, b, \mathit{unit}) \\ &\quad\textbf{and}\ c = \mathsf{C}(a, b, c) \end{aligned}$$

assuming *unit* is a datatype with the single constructor $\mathsf{Unity} : \mathit{unit}$. We call such codatatypes *corecursive singletons*, or simply singletons. For the decision procedure, it will be crucial to detect these. A type may also be a corecursive singleton only in some models. If the example above is altered to make *unit* an uninterpreted type, b and c will be singletons precisely when *unit* is interpreted as a singleton. Fortunately, given cardinalities for the ordinary types, it is easy to characterize this degenerate case.

Lemma 1 *Let δ be a corecursive codatatype. For any interpretation in $\mathscr{J}$, the domain interpreting δ is either infinite or a singleton. In the latter case, δ necessarily has a single constructor, whose arguments have types that are interpreted as singleton domains.*

Proof By definition, the type is equipped with at least one (directly or indirectly) corecursive constructor C. If it additional has second corecursive constructor D, it is possible to encode infinitely many alternation patterns—e.g., $\mathsf{C}(\mathsf{D}(\mathsf{C}(\mathsf{C}(\ldots))))$—all of which correspond to distinct values (by distinctness and injectivity). If the type has a noncorecursive constructor E, it is possible to create terms of arbitrary depth—e.g., $\mathsf{C}(\ldots(\mathsf{C}(\mathsf{E}))\ldots)$. In either case, there can be no finite models.

Therefore, C must be the only constructor. If any of its noncorecursive arguments has a cardinality greater than 1, it is possible to encode alternation patterns using it—e.g., $\mathsf{C}(0, \mathsf{C}(1, \mathsf{C}(0, \mathsf{C}(0, \ldots))))$—which again excludes finite models. Otherwise, the coinduction principle ensures that the type has at most one value. □

3 The Decision Procedure

Given a fixed signature Σ, the decision procedure for the universal theory of (co)datatypes determines the $\mathcal{DC}$-satisfiability of finite sets E of literals: equalities and disequalities be-

tween Σ-terms, whose variables are interpreted existentially. The decision procedure is formulated as a tableau-like calculus. Proving a universal quantifier-free conjecture is reduced to showing that its negation is unsatisfiable. The presentation is inspired by Barrett et al. [3] but at a higher level, using unoriented equations instead of oriented ones.

To simplify the presentation, we make a few assumptions about Σ. 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 [3].

3.1 A Calculus for $\mathcal{DC}$

Our calculus for $\mathcal{DC}$ consists of derivation rules. A derivation rule can be applied to E if its premises are met. The conclusion either specifies equalities to be added to E or is $\bot$ (contradiction). One of the rules has multiple conclusions, 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 sets of rules, given in Figs. 1 to 3, corresponding to three phases of the calculus. 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.

Phase 1: Computing the Bidirectional Closure (Fig. 1). 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. The Conflict rule recognizes when an equality and its negation both occur in E, in which case E has no model.

Let $\mathcal{T}(E)$ denote the set of Σ-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 $\mathsf{C}(\bar{t}) \in [t]$ among these.

Phase 2: Applying Acyclicity and Uniqueness (Fig. 2). The rules in this phase are described in terms of a mapping $\mathcal{A}$ that assigns to each equivalence class a μ-term as its representative.

Formally, *μ-terms* are defined recursively as being either a variable x or an applied constructor $\mu x.\, \mathsf{C}(\bar{t})$ for some $\mathsf{C} \in \mathcal{F}_{\mathrm{ctr}}$ and μ-terms $\bar{t}$ of the expected types. The variable x need

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

$$\frac{\bar{t} \approx \bar{u} \in E \quad \mathsf{f}(\bar{t}), \mathsf{f}(\bar{u}) \in \mathcal{T}(E)}{\mathsf{f}(\bar{t}) \approx \mathsf{f}(\bar{u}) \in E}\ \textsf{Cong} \qquad \frac{t \approx u,\ t \not\approx u \in E}{\bot}\ \textsf{Conflict}$$

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

Fig. 1 Derivation rules for bidirectional closure

$$\frac{\delta \in \mathcal{Y}_{\mathrm{dt}} \quad \mathcal{A}[t^\delta] = \mu x.\, u \quad x \in \mathrm{FV}(u)}{\bot}\ \textsf{Acyclic} \qquad \frac{\delta \in \mathcal{Y}_{\mathrm{codt}} \quad \mathcal{A}[t^\delta] =_\alpha \mathcal{A}[u^\delta]}{t \approx u \in E}\ \textsf{Unique}$$

Fig. 2 Derivation rules for acyclicity and uniqueness

$$\frac{\begin{array}{c} t^\delta \in \mathcal{T}(E) \quad \mathcal{F}^\delta_{\mathrm{ctr}} = \{\mathsf{C}_1, \ldots, \mathsf{C}_m\} \\ \big(\mathsf{s}(t) \in \mathcal{T}(E) \text{ and } \mathsf{s} \in \mathcal{F}^\delta_{\mathrm{sel}}\big) \text{ or } \big(\delta \in \mathcal{Y}_{\mathrm{dt}} \text{ and } \delta \text{ is finite}\big) \end{array}}{t \approx \mathsf{C}_1\big(\mathsf{s}^1_1(t), \ldots, \mathsf{s}^{n_1}_1(t)\big) \in E \quad \cdots \quad t \approx \mathsf{C}_m\big(\mathsf{s}^1_m(t), \ldots, \mathsf{s}^{n_m}_m(t)\big) \in E}\ \textsf{Split}$$

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

Fig. 3 Derivation rules for branching

not occur free in the μ-binder's body, in which case the binder can be omitted. $\mathrm{FV}(t)$ 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 μ-term $\mathcal{A}[t^\tau]$ describes a class of τ 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.

The mapping $\mathcal{A}$ is defined 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:

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

$\mathrm{FV}(\mathcal{A})$ 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.

Example 2 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 $\mathsf{C} \in \mathcal{F}_{\mathrm{ctr}}$. A possible sequence of unfolding steps is given below, omitting trivial entries such as $[w] \mapsto \widetilde{w}$.

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 $x \in \mathrm{FV}(u)$), then E is $\mathcal{DC}$-unsatisfiable. For uniqueness, if $[t]$, $[u]$ are two codatatype equivalence classes whose values $\mathcal{A}[t]$, $\mathcal{A}[u]$ are α-equivalent, then $t \approx u$. 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 3 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

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

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:

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

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

Phase 3: Branching (Fig. 3). 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 δ constructor term. This is enforced in the third phase by the Split rule. Another rule, Single, focuses on the degenerate case where two terms have the same corecursive singleton type and are therefore equal. 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.

3.2 Termination and Correctness of the Calculus

We now show the termination and correctness of the calculus. Correctness means the following: If there exists a closed derivation tree with root node E, then E is $\mathcal{DC}$-unsatisfiable; and if there exists a derivation tree with root node E that contains a saturated node, then E is $\mathcal{DC}$-satisfiable.

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

Proof Consider a derivation tree with root node E. Let $D \subseteq \mathcal{T}(E)$ be the set of terms whose types are finite datatypes, and let $S \subseteq \mathcal{T}(E)$ be the set of terms occurring as arguments to selectors. For each term $t \in D$, let

$$S_t^0 = \{t\} \qquad S_t^{i+1} = S_t^i \cup \{\mathsf{s}(u) \mid u^\delta \in S_t^i,\, \delta \in \mathcal{Y}_{\mathrm{dt}},\, |\delta| \text{ is finite},\, \mathsf{s} \in \mathcal{F}_{\mathrm{sel}}^\delta\}$$

and let S_t^∞ be the limit of this sequence. This is a finite set for each t, because all chains of selectors applied to t are finite. Let S^∞ be the union of all sets S_t^∞ where $t \in D$, and let $\mathcal{T}^\infty(E)$ be the set of subterms of $E \cup \{\mathsf{C}_j\big(\mathsf{s}_j^1(t),\ldots,\mathsf{s}_j^{n_j}(t)\big) \mid t^\delta \in S \cup S^\infty,\, \mathsf{C}_j \in \mathcal{F}_{\mathrm{ctr}}^\delta\}$. In a derivation tree with root node E, it can be shown by induction on the rules of the calculus that each non-root node F is such that $\mathcal{T}(F) \subseteq \mathcal{T}^\infty(E)$, and hence contains an equality between two terms from $\mathcal{T}^\infty(E)$ not occurring in its parent node. Thus, the depth of a branch in a derivation tree with root node E is at most $|\mathcal{T}^\infty(E)|^2$, which is finite since $\mathcal{T}^\infty(E)$ is finite. □

Theorem 2 (Refutation Soundness) *If there exists a closed derivation tree with root node E, then E is $\mathcal{DC}$-unsatisfiable.*

Proof The proof is by structural induction on the derivation tree with root node E. If the tree is an application of Conflict, Clash, or Acyclic, then E is $\mathcal{DC}$-unsatisfiable. For Conflict, this is a consequence of equality reasoning. For Clash, this is a consequence of distinctness. For Acyclic, the construction of $\mathcal{A}$ indicates that the class of values that term t can take in models of E is infeasible. If the child nodes of E are closed derivation trees whose root nodes are the result of applying Split on t^δ, by the induction hypothesis $E \cup t \approx \mathsf{C}_j\big(\mathsf{s}_j^1(t),\ldots,\mathsf{s}_j^{n_j}(t)\big)$ is $\mathcal{DC}$-unsatisfiable for each $\mathsf{C}_j \in \mathcal{F}_{\mathrm{ctr}}^\delta$. Since by exhaustiveness, all models of $\mathcal{DC}$ entail exactly one $t \approx \mathsf{C}_j\big(\mathsf{s}_j^1(t),\ldots,\mathsf{s}_j^{n_j}(t)\big)$, E is $\mathcal{DC}$-unsatisfiable. Otherwise, the child node of E is a closed derivation tree whose root node $E \cup t \approx u$ is obtained by applying one of the rules Refl, Sym, Trans, Cong, Inject, Unique, or Single. In each of these cases, $E \vDash_{\mathcal{DC}} t \approx u$. For Refl, Sym, Trans, Cong, this is a consequence of equality reasoning. For Inject, this is a consequence of injectivity. For Unique, the construction of $\mathcal{A}$ indicates that the values of t and u are equivalent in all models of E. For Single, t and u must have the same value since the cardinality of their type is one. By the induction hypothesis, $E \cup t \approx u$ is $\mathcal{DC}$-unsatisfiable and thus E is $\mathcal{DC}$-unsatisfiable. □

It remains to show the converse of the previous theorem: If a derivation tree with root node E contains a saturated node, then E is $\mathcal{DC}$-satisfiable. The proof relies on a specific interpretation $\mathcal{I}$ that satisfies E.

First, we define the set of interpretations of the theory $\mathcal{DC}$, which requires custom terminology concerning μ-terms. Given a μ-term t with subterm u, the *expansion of u with respect to t* is the μ-term $\langle u \rangle_t^\emptyset$, abbreviated to $\langle u \rangle_t$, as returned by the function

$$\langle x \rangle_t^B = \begin{cases} x & \text{if } x \in B \\ \mu x.\, \mathsf{C}\big(\langle \bar{u} \rangle_t^{B \uplus \{x\}}\big) & \text{if } \mu x.\, \mathsf{C}(\bar{u}) \text{ binds this occurrence of } x \notin B \text{ in } t \end{cases}$$

$$\langle \mu x.\, \mathsf{C}(\bar{u}) \rangle_t^B = \begin{cases} x & \text{if } x \in B \\ \mu x.\, \mathsf{C}\big(\langle \bar{u} \rangle_t^{B \uplus \{x\}}\big) & \text{otherwise} \end{cases}$$

The recursion will eventually terminate because each recursive call adds one bound variable to B and there are finitely many distinct bound variables in a μ-term. Intuitively, the expansion of a subterm is a self-contained μ-term that denotes the same value as the original subterm—e.g., $\langle \mu y.\, \mathsf{D}(x) \rangle_{\mu x.\, \mathsf{C}(\mu y.\, \mathsf{D}(x))} = \mu y.\, \mathsf{D}(\mu x.\, \mathsf{C}(y))$.

The μ-term u is a *self-similar subterm* of t if u is a proper subterm of t, t and u are of the forms $\mu x.\, \mathsf{C}(t_1,\ldots,t_n)$ and $\mu y.\, \mathsf{C}(u_1,\ldots,u_n)$, and $\langle t_k \rangle_t =_\alpha \langle u_k \rangle_t$ for all k. The μ-term t is

normal if it does not contain self-similar subterms and all of its proper subterms are also normal. Thus, $t = \mu x.\, \mathsf{C}(\mu y.\, \mathsf{C}(y))$ is not normal because $\mu y.\, \mathsf{C}(y)$ is a self-similar subterm of t. Their arguments have the same expansion with respect to t: $\langle \mu y.\, \mathsf{C}(y)\rangle_t = \mu y.\, \mathsf{C}\big(\langle y\rangle_t^{\{y\}}\big) = \mu y.\, \mathsf{C}(y)$ is α-equivalent to $\langle y\rangle_t = \mu y.\, \mathsf{C}\big(\langle y\rangle_t^{\{y\}}\big) = \mu y.\, \mathsf{C}(y)$. The term $u = \mu x.\, \mathsf{C}(\mu y.\, \mathsf{C}(x))$ is also not normal, since $\mu y.\, \mathsf{C}(x)$ is a self-similar subterm of u, noting that $\langle \mu y.\, \mathsf{C}(x)\rangle_u = \mu y.\, \mathsf{C}\big(\langle x\rangle_u^{\{y\}}\big) = \mu y.\, \mathsf{C}\big(\langle \mu x.\, \mathsf{C}(\mu y.\, \mathsf{C}(x))\rangle_u^{\{y\}}\big) = \mu y.\, \mathsf{C}\big(\mu x.\, \mathsf{C}(\langle \mu y.\, \mathsf{C}(x)\rangle_u^{\{x,y\}})\big) = \mu y.\, \mathsf{C}(\mu x.\, \mathsf{C}(y))$ is α-equivalent to $\langle x\rangle_u = u$.

For any μ-term t of the form $\mu x.\, \mathsf{C}(\bar{u})$, its *normal form* $\lfloor t \rfloor$ is obtained by replacing all of the self-similar subterms of t with x and by recursively normalizing the other subterms. For variables, $\lfloor x \rfloor = x$. Thus, $\lfloor \mu x.\, \mathsf{C}(\mu y.\, \mathsf{C}(x)) \rfloor = \mu x.\, \mathsf{C}(x)$.

We now define the class of interpretations for $\mathcal{DC}$. $\mathcal{I}(\tau)$ denotes the interpretation type τ in $\mathcal{I}$—that is, a nonempty set of domain elements for that type. $\mathcal{I}(\mathsf{f})$ denotes the interpretation of a function f in $\mathcal{I}$. If $\mathsf{f} : \tau_1 \times \cdots \times \tau_n \to \tau$, then $\mathcal{I}(\mathsf{f})$ is a total function from $\mathcal{I}(\tau_1) \times \cdots \times \mathcal{I}(\tau_n)$ to $\mathcal{I}(\tau)$. All types are interpreted as sets of μ-terms, but only values of types in $\mathcal{Y}_{\mathrm{codt}}$ may contain cycles.

Definition 1 (Normal Interpretation) An interpretation $\mathcal{I}$ is *normal* if the following conditions are met:

1. For each type τ, $\mathcal{I}(\tau)$ includes a maximal set of closed normal μ-terms of that type that are unique up to α-equivalence and acyclic unless $\tau \in \mathcal{Y}_{\mathrm{codt}}$.
2. For each constructor term $\mathsf{C}(\bar{t})$ of type τ, $\mathcal{I}(\mathsf{C})(\mathcal{I}(\bar{t}))$ is the value in $\mathcal{I}(\tau)$ that is α-equivalent to $\lfloor \mu x.\, \mathsf{C}(\mathcal{I}(\bar{t})) \rfloor$, where x is fresh.
3. For each selector term $\mathsf{s}_j^k(t)$ of type τ, if $\mathcal{I}(t)$ is $\mu x.\, \mathsf{C}_j(\bar{u})$, then $\mathcal{I}(\mathsf{s}_j^k)\big(\mathcal{I}(t)\big)$ is the value in $\mathcal{I}(\tau)$ that is α-equivalent to $\langle u_k\rangle_{\mathcal{I}(t)}$.

Not all normal interpretations are models of codatatypes, because models must contain all possible infinite terms, not only cyclic ones. However, acyclic infinite values are not interesting for deciding universal formulas: For such formulas, it is trivial to extend any normal interpretation with extra domain elements to obtain a genuine model if desired.

When constructing a model $\mathcal{I}$ of E, it remains only to specify how $\mathcal{I}$ interprets wrongly applied selector terms and variables. For the latter, this will be based on the mapping $\mathcal{A}$ computed in phase 2 of the calculus.

First, we need the following definitions. We write $t =_\alpha^x u$ if μ-terms t and u are syntactically equivalent for some renaming that avoids capturing any variable other than x. For example, $\mu x.\, \mathsf{D}(x) =_\alpha^y \mu x.\, \mathsf{D}(y)$ (by renaming y to x), $\mu x.\, \mathsf{C}(x,x) =_\alpha^x \mu y.\, \mathsf{C}(x,y)$, and $\mu x.\, \mathsf{C}(z,x) =_\alpha^z \mu y.\, \mathsf{C}(z,y)$, but $\mu x.\, \mathsf{D}(x) \neq_\alpha^x \mu x.\, \mathsf{D}(y)$ and $\mu x.\, \mathsf{C}(x,x) \neq_\alpha^y \mu y.\, \mathsf{C}(x,y)$. For a variable x^τ and a normal interpretation $\mathcal{I}$, we let $\mathcal{V}_\mathcal{I}^x(\mathcal{A})$ denote the set consisting of all values $v \in \mathcal{I}(\tau)$ such that $v =_\alpha^x \langle u\rangle_t$ for some subterm u of a term t occurring in the range of $\mathcal{A}$. This set describes shapes of terms to avoid when assigning a μ-term to x.

The *completion* $\mathcal{A}^\star$ of $\mathcal{A}$ for a normal interpretation $\mathcal{I}$ assigns values from $\mathcal{I}$ to unassigned variables in the domain of $\mathcal{A}$. We construct $\mathcal{A}^\star$ by initially setting $\mathcal{A}^\star := \lfloor \mathcal{A} \rfloor$ and by exhaustively applying the following rule:

$$\frac{\widetilde{x}^\tau \in \mathrm{FV}(\mathcal{A}^\star) \quad \mu\widetilde{x}.\, t =_\alpha v \quad v \in \mathcal{I}(\tau) \quad v \notin \mathcal{V}_\mathcal{I}^{\widetilde{x}}(\mathcal{A}^\star)}{\mathcal{A}^\star := \lfloor \mathcal{A}^\star[\widetilde{x} \mapsto \mu\widetilde{x}.\, t] \rfloor}$$

Given an unassigned variable in $\mathcal{A}^\star$, this rule assigns it a fresh value—one that does not occur in $\mathcal{V}_\mathcal{I}^{\widetilde{x}}(\mathcal{A}^\star)$ modulo α-equivalence—excluding not only existing terms in the range of $\mathcal{A}^\star$ but also terms that could emerge as a result of the update. Since this update removes one

variable from $\mathrm{FV}(\mathcal{A}^\star)$ and does not add any variables to $\mathrm{FV}(\mathcal{A}^\star)$, the process eventually terminates. We normalize all terms in the range of $\mathcal{A}^\star$ at each step.

To ensure disequality literals are satisfied by an interpretation based on $\mathcal{A}^\star$, it suffices that $\mathcal{A}^\star$ is injective modulo α-equivalence. This invariant holds initially, and the last premise in the above rule ensures that it is maintained. The set $\mathcal{V}_{\mathcal{I}}^{\widetilde{x}}(\mathcal{A}^\star)$ is an overapproximation of the values that, when assigned to $\widetilde{x}$, will cause values in the range of $\mathcal{A}^\star$ to become α-equivalent. For infinite codatatypes, it is always possible to find fresh values v because $\mathcal{V}_{\mathcal{I}}^{\widetilde{x}}(\mathcal{A}^\star)$ is a finite set.

Example 4 Let δ be a codatatype with the constructors $\mathsf{C}, \mathsf{D}, \mathsf{E} : \delta \to \delta$. Let E be the set $\{u \approx \mathsf{C}(z),\ v \approx \mathsf{D}(z),\ w \approx \mathsf{E}(y),\ x \approx \mathsf{C}(v),\ v \approx s,\ z \not\approx v\}$. After applying the calculus to saturation on E, the mapping $\mathcal{A}$ is as follows:

$$\begin{array}{lll} \mathcal{A}[u] = \mu\widetilde{u}.\,\mathsf{C}(\widetilde{z}) & \mathcal{A}[w] = \mu\widetilde{w}.\,\mathsf{E}(\widetilde{y}) & \mathcal{A}[y] = \widetilde{y} \\ \mathcal{A}[v] = \mu\widetilde{v}.\,\mathsf{D}(\widetilde{z}) & \mathcal{A}[x] = \mu\widetilde{x}.\,\mathsf{C}(\mu\widetilde{v}.\,\mathsf{D}(\widetilde{z})) & \mathcal{A}[z] = \widetilde{z} \end{array}$$

Here, $[u]$ denotes the equivalence class $\{u, \mathsf{C}(z)\}$, $[v]$ denotes $\{v, \mathsf{C}(z), s\}$, and so on. To construct a completion $\mathcal{A}^\star$, we must choose values for $\widetilde{y}$ and $\widetilde{z}$, which are free in $\mathcal{A}$. Modulo α-equivalence, $\mathcal{V}_{\mathcal{I}}^{\widetilde{z}}(\mathcal{A}) = \{\mu a.\,\mathsf{C}(a),\ \mu a.\,\mathsf{D}(a),\ \mu a.\,\mathsf{C}(\mathsf{D}(a)),\ \mathsf{C}(\mu a.\,\mathsf{D}(a))\}$. Now consider a normal interpretation $\mathcal{I}$ that evaluates variables in E based on $\mathcal{A}$: $\mathcal{I}(u) = \mathcal{A}[u]$, $\mathcal{I}(v) = \mathcal{A}[v]$, and so on. Assigning a value for $\mathcal{A}[z]$ that is α-equivalent to a value in $\mathcal{V}_{\mathcal{I}}^{\widetilde{z}}(\mathcal{A})$ may cause values in the range of $\mathcal{A}$ to become α-equivalent, which in turn may cause E to be falsified by $\mathcal{I}$. For example, assign $\mu\widetilde{z}.\,\mathsf{D}(\widetilde{z})$ for $\widetilde{z}$. After the substitution, $\mathcal{A}[v] = \mu\widetilde{v}.\,\mathsf{D}(\mu\widetilde{z}.\,\mathsf{D}(\widetilde{z}))$, which has normal form $\mu\widetilde{v}.\,\mathsf{D}(\widetilde{v})$, which is α-equivalent to $\mu\widetilde{z}.\,\mathsf{D}(\widetilde{z})$. However, this contradicts the disequality $z \not\approx v$ in E. On the other hand, if the value assigned to $\widetilde{z}$ is fresh, the values in the range of $\mathcal{A}$ remain α-disequivalent. We can assign a value such as $\mu\widetilde{z}.\,\mathsf{E}(\widetilde{z})$, $\mu\widetilde{z}.\,\mathsf{D}(\mathsf{C}(\widetilde{z}))$, or $\mu\widetilde{z}.\,\mathsf{C}(\mathsf{C}(\mathsf{D}(\widetilde{z})))$ to $\widetilde{z}$.

Legal substitutions for $\widetilde{z}$ may cause the range of $\mathcal{A}$ to contain abnormal terms. For example, after assigning $\mu\widetilde{z}.\,\mathsf{D}(\mathsf{C}(\widetilde{z}))$ to $\widetilde{z}$, we have $\mathcal{A}[u] = \mu\widetilde{u}.\,\mathsf{C}(\mu\widetilde{z}.\,\mathsf{D}(\mathsf{C}(\widetilde{z})))$, with normal form $\mu\widetilde{u}.\,\mathsf{C}(\mu\widetilde{z}.\,\mathsf{D}(\widetilde{u}))$. This explains why the term is normalized in the rule's conclusion. ■

In the following lemma about $\mathcal{A}^\star$, $\mathrm{Var}(t) = \begin{cases} t & \text{if } t \text{ is a variable} \\ x & \text{if } t \text{ is of the form } \mu x.\,u. \end{cases}$

Lemma 2 *If $\mathcal{A}$ is constructed for a saturated set E and $\mathcal{A}^\star$ is a completion of $\mathcal{A}$ for a normal interpretation $\mathcal{I}$, the following properties hold for all $[x]$ and $[y]$ in the domain of $\mathcal{A}^\star$:*

(1) *$\mathcal{A}^\star[x^\tau]$ is α-equivalent to a value in $\mathcal{I}(\tau)$.*
(2) *$\mathcal{A}^\star[x] = \langle t \rangle_{\mathcal{A}^\star[y]}$ for all subterms t of $\mathcal{A}^\star[y]$ with $\mathrm{Var}(t) = \widetilde{x}$.*
(3) *$\mathcal{A}^\star[x] =_\alpha \mathcal{A}^\star[y]$ if and only if $[x] = [y]$.*

Proof To show (1), we first show that $\mathcal{A}^\star$ contains no free variables. Assume by contradiction that $\mathcal{A}^\star$ contains a free variable $\widetilde{y}$ for some $[y]$ of type τ. Then it must be the case that $[y]$ does not contain a constructor term, or else $\widetilde{y}$ would not occur as a free variable in $\mathcal{A}$. Consider the case when τ is finite. By assumption, $\tau \notin \mathcal{Y}_{\mathrm{ord}}$. Since Split does not apply to E, we have $\tau \notin \mathcal{Y}_{\mathrm{dt}}$. If $\tau \in \mathcal{Y}_{\mathrm{codt}}$, then τ is corecursive by assumption, and by Lemma 1, the cardinality of τ must be one. Since Single does not apply, there is only one equivalence class of type τ in E, and thus there are no terms in $\mathcal{V}_{\mathcal{I}}^{\widetilde{y}}(\mathcal{A}^\star)$ of type τ. This is a contradiction, since completion can assign the value in the domain of τ to $\widetilde{y}$. Now, consider the case when τ is infinite. This is also a contradiction, since there are only a finite number of

closed terms in $\mathcal{V}_{\mathcal{J}}^{\widetilde{y}}(\mathcal{A}^\star)$, and thus completion can assign a value not occurring in $\mathcal{V}_{\mathcal{J}}^{\widetilde{y}}(\mathcal{A}^\star)$ to $\widetilde{y}$. By construction, $\mathcal{A}^\star[x]$ is normal. Since Acyclic does not apply, $\mathcal{A}[x]$ is acyclic when $\tau \in \mathcal{Y}_{\text{dt}}$. Moreover, the construction of $\mathcal{A}^\star$ applies substitutions of the form $\{\widetilde{y} \mapsto t\}$, where t is acyclic when the type of $\widetilde{y}$ is not a codatatype. Thus, $\mathcal{A}^\star[x]$ is acyclic when $\tau \notin \mathcal{Y}_{\text{codt}}$. Therefore, by definition, $\mathcal{A}^\star[x]$ is α-equivalent to a value in $\mathcal{J}(\tau)$.

We now show that (2) and (3) hold initially for $\mathcal{A}$. For all equivalence classes $[z]$, each pair of constructor terms $\mathsf{C}_j(\bar{t})$ and $\mathsf{C}_{j'}(\bar{u})$ in $[z]$ are such that $j = j'$, since Clash does not apply, and are such that $[\bar{t}] = [\bar{u}]$, since Inject does not apply. Thus, $\mathcal{A}$ was constructed by applying a sequence of substitutions where all substitutions for variables $\widetilde{z}$ were uniquely of the form $\{\widetilde{z} \mapsto \mathsf{C}_j(\widetilde{t_1},\ldots,\widetilde{t_n})\}$ when $[z]$ contains a constructor $\mathsf{C}_j(t_1,\ldots,t_n)$. Suppose $\mathcal{A}[y]$ has a subterm t such that $\text{Var}(t) = \widetilde{x}$. Both $\mathcal{A}[x]$ and the subterm t of $\mathcal{A}[y]$ were constructed by applying a sequence of substitutions of the form mentioned above to $\widetilde{x}$. Moreover, free variables $\widetilde{z}$ in t that are bound in $\mathcal{A}[y]$ are interpreted in the expansion of $\langle t \rangle_{\mathcal{A}[y]}$ as a term constructed by a sequence of substitutions of the form mentioned above to $\widetilde{z}$. Thus, we have $\langle t \rangle_{\mathcal{A}[y]} = \langle \mathcal{A}[x] \rangle_{\mathcal{A}[x]} = \mathcal{A}[x]$, and (2) holds for $\mathcal{A}$. Property (3) holds for $\mathcal{A}$ since Unique does not apply.

We now show that $\mathcal{A} = \lfloor \mathcal{A} \rfloor$. Assume by contradiction $\mathcal{A}[x] \neq \lfloor \mathcal{A}[x] \rfloor$ for some $\mathcal{A}[x]$ of minimal size. We have that $\mathcal{A}[x]$ is of the form $\mu\widetilde{x}.\, \mathsf{C}(t_1,\ldots,t_n)$. Due to the construction of $\mathcal{A}$, we know $[x]$ contains a constructor $\mathsf{C}(z_1,\ldots,z_n)$ and $\text{Var}(t_i) = z_i$ for some i. Since $\mathcal{A}[x]$ is a minimal, it contains a subterm of the form $\mu\widetilde{y}.\, \mathsf{C}(u_1,\ldots,u_n)$ where $\langle t_i \rangle_{\mathcal{A}[x]} =_\alpha \langle u_i \rangle_{\mathcal{A}[x]}$ for some i. Due to the construction of $\mathcal{A}$, $[y]$ contains a constructor $\mathsf{C}(w_1,\ldots,w_n)$ and $\text{Var}(u_i) = w_i$ for some i. Since Cong does not apply, we have $[w_j]$ and $[z_j]$ are distinct for some j. By (2), $\langle t_j \rangle_{\mathcal{A}[x]} = \mathcal{A}[z_j]$ and $\langle u_j \rangle_{\mathcal{A}[x]} = \mathcal{A}[w_j]$, which are not α-equivalent by (3), contradicting the fact that $\mu\widetilde{y}.\, \mathsf{C}(u_1,\ldots,u_n)$ is a self-similar subterm of $\mathcal{A}[x]$. Thus, $\mathcal{A} = \lfloor \mathcal{A} \rfloor$, and (2) and (3) hold for $\lfloor \mathcal{A} \rfloor$.

We now show that if (2) and (3) hold for some $\mathcal{A}_1$, they also hold for $\lfloor \mathcal{A}_1\sigma \rfloor$, where σ is a substitution of the form $\{\widetilde{x} \mapsto \mu\widetilde{x}.\, t\}$, $\widetilde{x} \in FV(\mathcal{A}_1)$, and $\mu\widetilde{x}.\, t$ is not α-equivalent to a term in $\mathcal{V}_{\mathcal{J}}^{\widetilde{x}}(\mathcal{A}_1)$. To show (2), by assumption of (2) on $\mathcal{A}_1$, we have $\langle u \rangle_{\mathcal{A}_1[y]} = \mathcal{A}_1[x]$ for all subterms u of $\mathcal{A}_1[y]$ where $\text{Var}(t) = \widetilde{x}$. Thus, $\langle u \rangle_{\mathcal{A}_1\sigma[y]} = \mathcal{A}_1\sigma[x]$ and $\langle u \rangle_{\lfloor \mathcal{A}_1\sigma \rfloor[y]} = \lfloor \mathcal{A}_1\sigma \rfloor[x]$. To show (3), consider two distinct equivalence classes $[y]$ and $[z]$, and assume by contradiction that $\lfloor \mathcal{A}_1\sigma \rfloor[y] =_\alpha \lfloor \mathcal{A}_1\sigma \rfloor[z]$. Due to (3) for $\mathcal{A}_1$, $[y]$ and $[z]$ must have (minimal) subterms where t_1 occurs in $\mathcal{A}_1[y]$ the same position p as t_2 occurs in $\mathcal{A}_1[z]$, and $t_1 \neq_\alpha t_2$. If t_1 (resp. t_2) is a free variable that is not $\widetilde{x}$, then $\lfloor \mathcal{A}_1\sigma \rfloor[y]$ (resp. $\lfloor \mathcal{A}_1\sigma \rfloor[z]$) contains t_1 (resp. t_2) at position p, and $\lfloor \mathcal{A}_1\sigma \rfloor[z]$ (resp. $\lfloor \mathcal{A}_1\sigma \rfloor[y]$) does not. If t_1 is of the form $\mu w_1.\, \mathsf{C}(\bar{t})$ and t_2 is of the form $\mu w_2.\, \mathsf{D}(\bar{u})$, then the expansion of $\lfloor \mathcal{A}_1\sigma \rfloor[y]$ and $\lfloor \mathcal{A}_1\sigma \rfloor[z]$ are α-disequivalent at position p. Since t_1 and t_2 are minimal, say t_1 is of the form $\mu w.\, \mathsf{C}(\bar{t})$, and t_2 is $\widetilde{x}$. Since σ maps $\widetilde{x}$ to a closed μ-term $\mu\widetilde{x}.\, t$, we have that $FV(t_1) \subseteq \{\widetilde{x}\}$, or else the expansion of $\lfloor \mathcal{A}_1\sigma \rfloor[y]$ and $\lfloor \mathcal{A}_1\sigma \rfloor[z]$ are α-disequivalent at position p since they do not contain the same free variables. Since $\langle t_1 \rangle_{\mathcal{A}_1[y]}$ is normal, there is a closed μ-term $v \in \mathcal{V}_{\mathcal{J}}^{\widetilde{x}}(\mathcal{A}_1)$ such that $v =_\alpha^{\widetilde{x}} \langle t_1 \rangle_{\mathcal{A}_1[y]}$. Thus, by assumption on the selection of $\mu\widetilde{x}.\, t$, we have $\mu\widetilde{x}.\, t \neq_\alpha^{\widetilde{x}} \langle t_1 \rangle_{\mathcal{A}_1[y]}$, which implies that the expansion of $\lfloor \mathcal{A}_1\sigma \rfloor[y]$ and $\lfloor \mathcal{A}_1\sigma \rfloor[z]$ are α-disequivalent at position p.

Thus, by induction on the number of applications of the above rule used to obtain $\mathcal{A}^\star$, we have that $\mathcal{A}^\star$ satisfies (2) and (3). □

Intuitively, this lemma states three properties of $\mathcal{A}^\star$ that together ensure that a normal interpretation $\mathcal{J}$ can be constructed that satisfies E. Property (1) states that the values in the range of $\mathcal{A}^\star$ are α-equivalent to a value in normal interpretation. This means they are closed, normal, and acyclic when required. Property (2) states that the interpretation of all subterms in the range of $\mathcal{A}^\star$ depends on its associated variable only. In other words, the interpretation

of a subterm t where $\mathrm{Var}(t) = \widetilde{x}$ is equal to $\mathcal{A}^\star[x]$, independently of the context. Property (3) states that $\mathcal{A}^\star$ is injective (modulo α-equivalence), which ensures that distinct values are assigned to distinct equivalence classes.

Example 5 Consider $\mathcal{A}$ from Example 4. While extending $\mathcal{A}$ to its corresponding completion $\mathcal{A}^\star$, we may, for instance, assign $\mu\widetilde{z}.\,\mathsf{E}(\widetilde{z})$ to $\widetilde{z}$, and subsequently assign $\mu\widetilde{y}.\,\mathsf{D}(\widetilde{y})$ to $\widetilde{y}$. We obtain the following mapping $\mathcal{A}^\star$ from equivalence classes to μ-terms:

$$\begin{array}{lll}
\mathcal{A}^\star[u] = \mu\widetilde{u}.\,\mathsf{C}(\mu\widetilde{z}.\,\mathsf{E}(\widetilde{z})) & \mathcal{A}^\star[w] = \mu\widetilde{w}.\,\mathsf{E}(\mu\widetilde{y}.\,\mathsf{D}(\widetilde{y})) & \mathcal{A}^\star[y] = \mu\widetilde{y}.\,\mathsf{D}(\widetilde{y}) \\
\mathcal{A}^\star[v] = \mu\widetilde{v}.\,\mathsf{D}(\mu\widetilde{z}.\,\mathsf{E}(\widetilde{z})) & \mathcal{A}^\star[x] = \mu\widetilde{x}.\,\mathsf{C}(\mu\widetilde{v}.\,\mathsf{D}(\mu\widetilde{z}.\,\mathsf{E}(\widetilde{z}))) & \mathcal{A}^\star[z] = \mu\widetilde{z}.\,\mathsf{E}(\widetilde{z})
\end{array}$$

Notice that $\mathcal{A}^\star$ satisfies the properties from Lemma 2 for a normal interpretation $\mathcal{I}$: All terms in the range of $\mathcal{A}^\star$ are closed and normal, $\mathcal{A}^\star[s] = \langle t \rangle_{\mathcal{A}^\star[y]}$ for all subterms t where $\mathrm{Var}(t) = \widetilde{s}$ for all s, and all terms in the range of $\mathcal{A}^\star$ are pairwise α-disequivalent. We may construct an interpretation $\mathcal{I}$ where variables are interpreted as the value that is α-equivalent to one from $\mathcal{I}(\delta)$ and that is uniquely associated with its equivalence class by $\mathcal{A}^\star$. ■

Theorem 3 (Solution Soundness) *If there exists a derivation tree with root node E containing a saturated node, then E is $\mathcal{DC}$-satisfiable.*

Proof Let F be a saturated node in a derivation tree with root node E. We consider a normal interpretation $\mathcal{I}$ that interprets wrongly applied selectors based on equality information in F and that interprets the variables of F based on the completion $\mathcal{A}^\star$. For the variables, let $\mathcal{I}(x^\tau)$ be the value in $\mathcal{I}(\tau)$ that is α-equivalent with $\mathcal{A}^\star[x]$ for each variable $x \in \mathcal{T}(F)$, which by Lemma 2(1) is guaranteed to exist.

We first show that $\mathcal{I}$ satisfies all equalities $t_1 \approx t_2 \in F$. To achieve this, we show by structural induction on t^τ that $\mathcal{I}(t) =_\alpha \mathcal{A}^\star[t]$ for all terms $t \in \mathcal{T}(F)$, which implies $\mathcal{I} \models t_1 \approx t_2$ since $\mathcal{I}$ is normal.

If t is a variable, then $\mathcal{I}(t) =_\alpha \mathcal{A}^\star[t]$ by construction.

If t is a constructor term of the form $\mathsf{C}(u_1,\ldots,u_n)$, then $\mathcal{I}(t)$ is α-equivalent with $\lfloor \mu x.\, \mathsf{C}(\mathcal{I}(u_1),\ldots,\mathcal{I}(u_n)) \rfloor$ for some fresh x, which by the induction hypothesis is α-equivalent with $\lfloor \mu x.\, \mathsf{C}(\mathcal{A}^\star[u_1],\ldots,\mathcal{A}^\star[u_n]) \rfloor$. Call this term t'. Since Inject and Clash do not apply to F, by the construction of $\mathcal{A}^\star$ we have that $\mathcal{A}^\star[t]$ is a term of the form $\mu\widetilde{t}.\,\mathsf{C}(w_1,\ldots,w_n)$ where $\mathrm{Var}(w_i) = \widetilde{u_i}$ for each i. Thus by Lemma 2(2), $\langle w_i \rangle_{\mathcal{A}^\star[t]} = \mathcal{A}^\star[u_i]$. For each i, let ${u_i}'$ be the ith argument of t'. We have that $\langle {u_i}' \rangle_{t'} =_\alpha \mathcal{A}^\star[u_i]$, and thus $\langle {u_i}' \rangle_{t'} =_\alpha \langle w_i \rangle_{\mathcal{A}^\star[t]}$. Thus, $\mathcal{I}(t) =_\alpha t' =_\alpha \mathcal{A}^\star[t]$, and we have $\mathcal{I}(t) =_\alpha \mathcal{A}^\star[t]$.

If t is a selector term $\mathsf{s}^k_j(u)$, since Split does not apply to F, $[u]$ must contain a term of the form $\mathsf{C}_{j'}\big(\mathsf{s}^1_{j'}(u),\ldots,\mathsf{s}^n_{j'}(u)\big)$ for some j'. Since Inject and Clash are not applicable, by construction $\mathcal{A}^\star[u]$ must be of the form $\mu\widetilde{u}.\,\mathsf{C}_{j'}(w_1,\ldots,w_n)$, where $\mathrm{Var}(w_i) = \widetilde{\mathsf{s}^i_{j'}(u)}$ for each i, and thus by Lemma 2(2), $\langle w_i \rangle_{\mathcal{A}^\star[u]} = \mathcal{A}^\star[\mathsf{s}^i_{j'}(u)]$. If $j = j'$, then $\mathcal{I}(t)$ is α-equivalent with $\langle w_k \rangle_{\mathcal{A}^\star[u]}$, which is equal to $\mathcal{A}^\star[\mathsf{s}^k_j(u)] = \mathcal{A}^\star[t]$. If $j \neq j'$, since Cong does not apply, any term of the form $\mathsf{s}^k_j(u')$ not occurring in $[t]$ is such that $[u] \neq [u']$. By the induction hypothesis and Lemma 2(3), $\mathcal{I}(u) \neq \mathcal{I}(u')$ for all such u, u'. Thus, we may interpret $\mathcal{I}(\mathsf{s}^k_j)(\mathcal{I}(u))$ as the value in $\mathcal{I}(\tau)$ that is α-equivalent with $\mathcal{A}^\star[t]$.

We now show that all disequalities in F are satisfied by $\mathcal{I}$. Assume $t \not\approx u \in F$. Since Conflict does not apply, $t \approx u \notin F$ and thus $[t]$ and $[u]$ are distinct. Since $\mathcal{I}(t) =_\alpha \mathcal{A}^\star[t]$ and $\mathcal{I}(u) =_\alpha \mathcal{A}^\star[u]$, by Lemma 2(3), $\mathcal{I}(t) \neq \mathcal{I}(u)$, and thus $\mathcal{I} \models t \not\approx u$.

Since F contains only equalities and disequalities, we have $\mathcal{I} \models F$, and since $E \subseteq F$, we conclude that $\mathcal{I} \models E$. □

Example 6 Continuing Example 5, let $\mathcal{J}$ be a normal interpretation, where up to renaming of μ-bound variables, we have

$$\begin{array}{rrr} \mathcal{J}(u) = \mathsf{C}(\mu\widetilde{z}.\,\mathsf{E}(\widetilde{z})) & \mathcal{J}(w) = \mathsf{E}(\mu\widetilde{y}.\,\mathsf{D}(\widetilde{y})) & \mathcal{J}(y) = \mu\widetilde{y}.\,\mathsf{D}(\widetilde{y}) \\ \mathcal{J}(s) = \mathcal{J}(v) = \mathsf{D}(\mu\widetilde{z}.\,\mathsf{E}(\widetilde{z})) & \mathcal{J}(x) = \mathsf{C}(\mathsf{D}(\mu\widetilde{z}.\,\mathsf{E}(\widetilde{z}))) & \mathcal{J}(z) = \mu\widetilde{z}.\,\mathsf{E}(\widetilde{z}) \end{array}$$

It is easy to see that $\mathcal{J}$ satisfies the constraints

$$u \approx \mathsf{C}(z) \qquad v \approx \mathsf{D}(z) \qquad w \approx \mathsf{E}(y) \qquad x \approx \mathsf{C}(v) \qquad v \approx s \qquad z \not\approx v \qquad \blacksquare$$

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 is constructive in that it provides a method for constructing a model for a saturated configuration, by means of the mapping $\mathcal{A}^\star$.

4 Implementation in CVC4

The decision procedure was presented at a high level of abstraction, omitting quite a few details. This section describes the main aspects of the implementation within the SMT solver CVC4: the integration of the procedure into CDCL(T) [15], the construction of models with μ-terms, and the extension of the procedure to quantified formulas.

4.1 A Theory Solver for $\mathcal{DC}$

The decision procedure is implemented as a theory solver of CVC4—that is, a specialized procedure for determining the satisfiability of conjunctions of literals for its theory. Given a theory $T = T_1 \cup \cdots \cup T_n$ and a set of input clauses F in conjunctive normal form, the CDCL(T) procedure incrementally builds partial assignments of truth values to the atoms of F such that no clause in F is falsified. We can regard such a partial assignment as a set M of true literals. By a variant [19] of the Nelson–Oppen method [29], each T_i-solver takes as input the union M_i of

- the purified form of T_i-literals occurring in M, where fresh variables replace terms containing symbols not belonging to T_i;
- additional (dis)equalities between variables of types not belonging to T_i.

Each T_i-solver either reports that a subset C of M_i is T_i-unsatisfiable, in which case $\neg C$ is added to F, adds a clause to F, or does nothing. When M is a complete assignment for F, a theory solver can choose to do nothing only if M_i is indeed T_i-satisfiable.

Assume E is initially the set M_i described above. With each equality $t \approx u$ added to E, we associate a set of equalities from M_i that together entail $t \approx u$, which we call its *explanation*. Similarly, each $\mathcal{A}[x]$ is assigned an explanation—that is, a set of equalities from M_i that entail that the values of $[x]$ in models of E are of the form $\mathcal{A}[x]$. For example, if $x \approx \mathsf{C}(x) \in M_i$, then $x \approx \mathsf{C}(x)$ is an explanation for $\mathcal{A}[x] = \mu\widetilde{x}.\,\mathsf{C}(\widetilde{x})$. If multiple rules can derive the same conclusion, the solver simply keeps the first explanation it encounters.

The rules of the calculus are implemented as follows. For all rules with conclusion $\bot$, we report the union of the explanations for all premises is $\mathcal{DC}$-unsatisfiable. The implementation does not use any techniques to minimize the size of this set, which in some cases may be non-minimal. For Split, we add the exhaustiveness clause $t \approx \mathsf{C}_1\big(\mathsf{s}_1^1(t),\ldots,\mathsf{s}_1^{n_1}(t)\big) \vee \cdots \vee t \approx$

$\mathsf{Acyclic}(t)$:

1. Let U be the datatype equivalence classes of E.
2. Repeat until $U \neq \emptyset$:
   2.1. For some $[t] \in U$, $\mathsf{Traverse}([t], U, \emptyset)$.

$\mathsf{Traverse}([t], U, P)$:

1. If $[t] \in P$, $\mathsf{Acyclic}(t)$.
2. If $[t] \in U$:
   2.1. If $\mathsf{C}(t_1, \ldots, t_k) \in [t]$, then for each $j = 1, \ldots k$:
      2.1.1. $\mathsf{Traverse}([t_j], U, P \cup \{[t]\})$.
   2.2. $U := U \setminus \{[t]\}$.

Fig. 4 Algorithm for applying the Acyclic rule

$\mathsf{Unique}(t, u)$:

1. Let $\mathcal{U}$ be a partition of the equivalence classes of E such that $[t] =_{\mathcal{U}} [u]$ for distinct $[t], [u]$ if and only if t, u have type $\delta \in \mathcal{Y}_{\text{codt}}$, $\mathsf{C}(t_1, \ldots, t_k) \in [t]$, and $\mathsf{C}(u_1, \ldots, u_k) \in [u]$.
2. Repeat until $\mathcal{U}$ is unchanged:
   2.1. $\mathcal{U}' := \emptyset$.
   2.2. For each $U_i \in \mathcal{U}$:
      2.2.1. Let $\mathcal{U}_i$ be a partition of U_i such that $[t] =_{\mathcal{U}_i} [u]$ for distinct $[t], [u]$ if and only if
         – $\mathsf{C}(t_1, \ldots, t_k) \in [t]$ and $\mathsf{C}(u_1, \ldots, u_k) \in [u]$, and
         – $[t_j] =_{\mathcal{U}} [u_j]$ for each $j = 1, \ldots, k$.
      2.2.2. $\mathcal{U}' := \mathcal{U}' \cup \mathcal{U}_i$.
   2.3. $\mathcal{U} := \mathcal{U}'$.
3. If distinct $[t], [u] \in U$ for some $U \in \mathcal{U}$, $\mathsf{Unique}(t, u)$.

Fig. 5 Algorithm for applying the Unique rule

$\mathsf{C}_m\big(\mathsf{s}_m^1(t), \ldots, \mathsf{s}_m^{n_m}(t)\big)$ to F. Decisions on which branch to take are thus performed externally by the SAT solver. All other rules add equalities to the internal state of the theory solver. The rules in phase 1 are performed eagerly—that is, for partial satisfying assignments M—while the rules in phases 2 and 3 are performed only for complete satisfying assignments M.

Before constructing a model for F, the theory solver constructs neither μ-terms nor the mapping $\mathcal{A}$. Instead, it relies on the algorithms in Figs. 4 and 5 for determining whether the rules Acyclic and Unique apply to the set E.

For Acyclic, Fig. 4 considers the set U of all datatype equivalence classes of E. We choose an arbitrary $[t]$ in U and call the recursive subprocedure Traverse, which takes as input the current equivalence class we are processing, the set of equivalence classes U we have yet to process, and the set of equivalence classes P we are currently processing. If a call to $\mathsf{Traverse}([t], U, P)$ is such that $[t] \in P$, we know that the rule Acyclic applies to t. Otherwise, if we have yet to process $[t]$, as indicated by the condition $[t] \in U$, then if $[t]$ contains a constructor term $\mathsf{C}(t_1, \ldots, t_k)$, then we recursively call Traverse on each of $[t_1], \ldots, [t_n]$. If this succeeds, we remove $[t]$ from U.

For Unique, Fig. 5 considers a partition $\mathcal{U}$ of the equivalence classes of our set E such that codatatype equivalence classes having a constructor term with top symbol C are placed in the same subset, for each constructor C. We write $[t] =_{\mathcal{U}} [u]$ to denote that $[t]$ and $[u]$ reside in the same subset within $\mathcal{U}$. We then refine $\mathcal{U}$ by constructing a partition $\mathcal{U}_i$ of each $U_i \in \mathcal{U}$ such that distinct $[t]$ and $[u]$ reside in the same subset of $\mathcal{U}_i$ if and only if they

contain constructor terms $\mathsf{C}(t_1,\ldots,t_k)$ and $\mathsf{C}(u_1,\ldots,u_k)$ where each t_j, u_j are such that $[t_j]$ and $[u_j]$ reside in the same subset of $\mathcal{U}$ for each $j = 1,\ldots,k$. Let $\mathcal{U}'$ be the union of each of these partitions. We update $\mathcal{U}$ to $\mathcal{U}'$. This refinement is repeated until $\mathcal{U}$ is left unchanged. If the resulting partition $\mathcal{U}$ contains a subset U having two distinct equivalence classes $[t]$ and $[u]$, then Unique applies to t and u. This algorithm is analogous to Hopcroft's algorithm for minimizing deterministic finite automata [18]—terms correspond to states, argument positions $1,\ldots,k$ correspond to input symbols, and the top constructor symbol of a term generalizes the accepting/rejecting status of a state.

The implementation of the decision procedure uses several optimizations following the lines of Barrett et al. [3]. We briefly mention the main ones. Discriminators are part of the signature and not abbreviations. This requires extending the decision procedure with several rules, which apply uniformly to datatypes and codatatypes. This approach often leads to better performance because it introduces terms less eagerly to $\mathcal{T}(E)$. Selectors are collapsed eagerly: If $\mathsf{s}_j^k(t) \in \mathcal{T}(E)$ and $t = \mathsf{C}_j(u_1,\ldots,u_n)$, the solver directly adds $\mathsf{s}_j^k(t) \approx u_k$ to E, whereas the presented calculus would apply Split and Inject before adding this equality. To reduce the number of unique literals considered by the calculus, we compute a normal form for literals as a preprocessing step. In particular, we replace $u \approx t$ by $t \approx u$ if t is smaller than u with respect to some term ordering, replace $\mathsf{C}_j(\bar{t}) \approx \mathsf{C}_{j'}(\bar{u})$ with $\bot$ when $j \neq j'$, replace all selector terms of the form $\mathsf{s}_j^k(\mathsf{C}_j(t_1,\ldots,t_n))$ by t_k, and replace occurrences of discriminators $\mathsf{d}_j(\mathsf{C}_{j'}(\bar{t}))$ by $\top$ or $\bot$ based on whether $j = j'$. Finally, finite ordinary types are not converted to finite datatypes. We instead rely on standard extensions of the Nelson–Oppen method [40], which impose requirements regarding terms having finite sorts are shared between multiple theories. In particular, given an ordinary type τ of finite cardinality n belonging to theory T_i, the theory solver for T_i must communicate enough equalities over shared terms to ensure that all other theory solvers, including the one for $\mathcal{DC}$, interpret τ as a set having cardinality at most n.

As Barrett et al. observed for their procedure, it is both theoretically and empirically beneficial to delay applications of Split as long as possible. Similarly, Acyclic and Unique are fairly expensive because they require traversing the equivalence classes, which is why they are part of phase 2.

4.2 Model Construction

When instructed to do so, the implementation produces models for satisfiable inputs. As described in Section 3.2, given a saturated set E, we construct a map $\mathcal{A}^\star$ from the equivalence classes of E to closed normal μ-terms. Recall that our construction of $\mathcal{A}^\star$ requires choosing values $\mu\widetilde{x}.\, t$ for each $\widetilde{x} \in \mathrm{FV}(\mathcal{A}^\star)$ such that $\mu\widetilde{x}.\, t$ does not occur in the set $\mathcal{V}_j^{\widetilde{x}}(\mathcal{A}^\star)$. To choose values, we use a fair enumerator for each (co)datatype τ, which lists all the values of type τ in a normal interpretation.

For codatatypes, we enumerate a stream of every μ-term of our signature Σ in a fair manner, discarding those that have free variables or are not normal. For those that are closed and normal, we then check whether they occur in the set $\mathcal{V}_j^{\widetilde{x}}(\mathcal{A}^\star)$. This set does not need to be explicitly constructed. To determine if a term $\mu\widetilde{x}.\, t$ occurs in the set $\mathcal{V}_j^{\widetilde{x}}(\mathcal{A}^\star)$, we check if it is α-equivalent to a closed term $(\mu\widetilde{y}.\, u)\{\widetilde{x} \mapsto z\}$ for some $\mu\widetilde{y}.\, u$ in the range of $\mathcal{A}^\star$ and variable z. This can be efficiently achieved by matching the term $\mu\widetilde{x}.\, t$ with $\mu\widetilde{y}.\, u$.

4.3 Extension to Quantified Formulas

While the decision procedure is restricted to universal conjectures, users often want to solve problems that feature universal axioms and existential conjectures. Many SMT solvers, including CVC4, can reason about quantified formulas using incomplete instantiation-based techniques [27,36]. These techniques extend naturally to quantified formulas involving datatypes and codatatypes.

However, the presence of quantifiers poses an additional challenge in the context of (co)datatypes. Quantified formulas can entail an upper bound on the cardinality of an uninterpreted type u. Since it assumes that uninterpreted types have infinite cardinality, the calculus presented in Section 3 is incomplete since it may fail to recognize cases where Split and Single should be applied. This does not impact the correctness of the procedure in this setting, since the solver is already incomplete for quantified formulas.

Nonetheless, two techniques help increase the precision of the solver. First, we can apply Split to datatype terms whose cardinality depends on the finiteness of uninterpreted types. Second, we can conditionally apply Single to codatatype terms whose type potentially has cardinality one. For example, the codatatype stream_u has cardinality one precisely when u has cardinality one. If there exist two equivalence classes $[s]$ and $[t]$ for this type, the implementation adds the clause $(\exists x\, y^u.\ x \not\approx y) \vee s \approx t$ to F, which states that either the cardinality of u is greater than one or s must be equal to t.

5 Evaluation on Isabelle Problems

The decision procedure for (co)datatypes is useful both for proving (via negation, in the refutational style) and for model finding [16, 35]. 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

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

on streams, 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 that both correspond to the term $\mu s.\, \mathsf{SCons}(0, s)$, 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 [5] as translator from Isabelle to SMT-LIB. We included all the formalizations from the Isabelle distribution (Distro, 1179 goals) and the *Archive of Formal Proofs* (AFP, 3014 goals) [21] that define codatatypes falling within the supported fragment. We also included formalizations about Bird and Stern–Brocot trees (SBT, 265 goals) [14]. To exercise the datatype support, formalizations about finite lists and trees were added to the first two benchmark sets. The formalizations were selected before conducting any experiments. The experimental data are available online.[3]

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

[3] http://lara.epfl.ch/~reynolds/JAR-CADE2015-cdt/ or http://www21.in.tum.de/~blanchet/JAR-CADE2015-cdt/

	Distro		AFP		SBT		Overall	
	CVC4	Z3	CVC4	Z3	CVC4	Z3	CVC4	Z3
No (co)datatypes	287	282	765	771	47	44	1099	1097
Datatypes without Acyclic	298	–	771	–	47	–	1116	–
Full datatypes	298	294	775	783	47	44	1120	1121
Codatatypes without Unique	288	–	797	–	47	–	1132	–
Full codatatypes	288	–	797	–	**52**	–	1137	–
Full (co)datatypes	**299**	–	**806**	–	**52**	–	**1157**	–

Table 1 Number of solved goals with 16 lemmas per goal

	Distro		AFP		SBT		Overall	
	CVC4	Z3	CVC4	Z3	CVC4	Z3	CVC4	Z3
No (co)datatypes	617	560	1503	1271	89	80	2209	1911
Datatypes without Acyclic	617	–	**1504**	–	90	–	2211	–
Full datatypes	617	560	**1504**	1263	90	78	2211	1901
Codatatypes without Unique	617	–	1501	–	90	–	2208	–
Full codatatypes	**620**	–	1502	–	98	–	**2220**	–
Full (co)datatypes	619	–	1501	–	**99**	–	2219	–

Table 2 Number of solved goals with 256 lemmas per goal

2015) and to Z3 4.3.2 for comparison, each running for up to 60 s on StarExec [38]. Problems not involving any (co)datatypes were left out. Due to the lack of machinery in Isabelle for parsing CVC4 proofs and reconstructing inferences about (co)datatypes, the solvers are trusted as oracles.

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-generated problems.

The results are summarized in Tables 1 and 2. The decision procedure makes a difference across all three benchmark suites. 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%. Table 2 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. This potentially points to poor interactions between the decision procedure and the quantifier instantiation module [36].

Overall, four proofs were found thanks to the acyclicity rule and 17 required uniqueness. Interestingly, no proofs were lost by enabling these rules. Among the 17 proofs requiring uniqueness, some were simple arguments of the form **by** *coinduction auto* in Isabelle [6], while others involved more elaborate reasoning, including the following example about Stern–Brocot trees:

lemma *num_mod_den_unique*: $x = \mathsf{Node}\ 0\ \mathsf{num}\ x \Longrightarrow x = \mathsf{num\_mod\_den}$
proof (*coinduction arbitrary*: *x rule*: *tree.coinduct_strong*)
  case (*Eq_tree x*) **show** *?case*
    by (*subst* (1 2 3 4) *Eq_tree*) (*simp add*: *eqTrueI*[OF *Eq_tree*])
qed

The tree num_mod_den is defined as $\mathsf{num\_mod\_den} = \mathsf{Node}\ 0\ \mathsf{num}\ \mathsf{num\_mod\_den}$.

6 Conclusion

We introduced a decision procedure for the universal theory of datatypes and codatatypes. Our main contribution has been the support for codatatypes. Both the metatheory and the implementation in CVC4 rely 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 [33]; in this context, the acyclicity and uniqueness rules are crucial to exclude spurious countermodels. As future work, it would be useful to implement proof reconstruction for (co)datatype inferences in Isabelle. In addition, it might be worthwhile to extend SMT solvers with dedicated support for (co)recursion.

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 theory solver for datatypes in CVC4. Our present and former bosses, Viktor Kuncak, Stephan Merz, Tobias Nipkow, Cesare Tinelli, and Christoph Weidenbach, have either encouraged the research on codatatypes or at least benevolently tolerated it, both of which we are thankful for. Peter Gammie and Andreas Lochbihler pointed us to useful benchmarks. Andrei Popescu helped clarify our thoughts regarding codatatypes and indicated related work. Dmitriy Traytel took part in discussions about degenerate codatatypes. Pascal Fontaine, Andreas Lochbihler, Andrei Popescu, Christophe Ringeissen, Mark Summerfield, Dmitriy Traytel, and the anonymous reviewers suggested many textual improvements. 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

1. Barrett, C., Conway, C.L., Deters, M., Hadarean, L., Jovanović, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: G. Gopalakrishnan, S. Qadeer (eds.) CAV '11, *LNCS*, vol. 6806, pp. 171–177. Springer (2011)
2. Barrett, C., Fontaine, P., Tinelli, C.: The SMT-LIB standard: Version 2.5. Tech. rep., University of Iowa (2015). Available at `http://smt-lib.org/`
3. Barrett, C., Shikanian, I., Tinelli, C.: An abstract decision procedure for satisfiability in the theory of inductive data types. J. Satisf. Boolean Model. Comput. **3**, 21–46 (2007)
4. Bjørner, N.S.: Integrating decision procedures for temporal verification. Ph.D. thesis, Stanford University (1998)
5. Blanchette, J.C., Böhme, S., Paulson, L.C.: Extending Sledgehammer with SMT solvers. J. Autom. Reasoning **51**(1), 109–128 (2013)
6. Blanchette, J.C., Hölzl, J., Lochbihler, A., Panny, L., Popescu, A., Traytel, D.: Truly modular (co)datatypes for Isabelle/HOL. In: G. Klein, R. Gamboa (eds.) ITP 2014, *LNCS*, vol. 8558, pp. 93–110. Springer (2014)

7. Blanchette, J.C., Nipkow, T.: Nitpick: A counterexample generator for higher-order logic based on a relational model finder. In: M. Kaufmann, L.C. Paulson (eds.) ITP 2010, *LNCS*, vol. 6172, pp. 131–146. Springer (2010)
8. Blanchette, J.C., Paskevich, A.: TFF1: The TPTP typed first-order form with rank-1 polymorphism. In: M.P. Bonacina (ed.) CADE-24, *LNCS*, vol. 7898, pp. 414–420. Springer (2013)
9. Blanchette, J.C., Popescu, A., Traytel, D.: Witnessing (co)datatypes. In: J. Vitek (ed.) ESOP 2015, *LNCS*, vol. 9032, pp. 359–382. Springer (2015)
10. Carayol, A., Morvan, C.: On rational trees. In: Z. Ésik (ed.) CSL 2006, *LNCS*, vol. 4207, pp. 225–239. Springer (2006)
11. Cruanes, S.: Extending superposition with integer arithmetic, structural induction, and beyond. Ph.D. thesis, École polytechnique (2015). Available at `https://who.rocq.inria.fr/Simon.Cruanes/files/thesis.pdf`
12. Djelloul, K., Dao, T., Frühwirth, T.W.: Theory of finite or infinite trees revisited. Theor. Pract. Log. Prog. **8**(4), 431–489 (2008)
13. Endrullis, J., Grabmayer, C., Klop, J.W., van Oostrom, V.: On equal μ-terms. Theor. Comput. Sci. **412**(28), 3175–3202 (2011)
14. Gammie, P., Lochbihler, A.: The Stern–Brocot tree. Archive of Formal Proofs (2016). `http://afp.sf.net/entries/Stern_Brocot.shtml`, Formal proof development
15. Ganzinger, H., Hagen, G., Nieuwenhuis, R., Oliveras, A., Tinelli, C.: DPLL(T): Fast decision procedures. In: R. Alur, D. Peled (eds.) CAV '04, *LNCS*, vol. 3114, pp. 175–188. Springer (2004)
16. Ge, Y., de Moura, L.: Complete instantiation for quantified formulas in satisfiability modulo theories. In: CAV '09, *LNCS*, vol. 5643, pp. 306–320. Springer (2009)
17. Gunter, E.L.: Why we can't have SML-style `datatype` declarations in HOL. In: L.J.M. Claesen, M.J.C. Gordon (eds.) TPHOLs '92, *IFIP Transactions*, vol. A-20, pp. 561–568. North-Holland/Elsevier (1993)
18. Hopcroft, J.: An $n \log n$ algorithm for minimizing states in a finite automaton. In: Z. Kohavi, A. Paz (eds.) Theory of Machines and Computations, pp. 189–196. Academic Press (1971)
19. Jovanović, D., Barrett, C.: Sharing is caring: Combination of theories. In: C. Tinelli, V. Sofronie-Stokkermans (eds.) FroCoS 2011, *LNCS*, vol. 6989, pp. 195–210. Springer (2011)
20. Kersani, A., Peltier, N.: Combining superposition and induction: A practical realization. In: P. Fontaine, C. Ringeissen, R.A. Schmidt (eds.) FroCoS 2013, *LNCS*, vol. 8152, pp. 7–22. Springer (2013)
21. Klein, G., Nipkow, T., Paulson, L. (eds.): Archive of Formal Proofs. `http://afp.sf.net/`
22. Kozen, D.: Results on the propositional μ-calculus. Theor. Comput. Sci. **27**, 333–354 (1983)
23. Leino, K.R.M., Moskal, M.: Co-induction simply—Automatic co-inductive proofs in a program verifier. In: C.B. Jones, P. Pihlajasaari, J. Sun (eds.) FM 2014, *LNCS*, vol. 8442, pp. 382–398. Springer (2014)
24. Leroy, X.: A formally verified compiler back-end. J. Autom. Reasoning **43**(4), 363–446 (2009)
25. Lochbihler, A.: Verifying a compiler for Java threads. In: A.D. Gordon (ed.) ESOP 2010, *LNCS*, vol. 6012, pp. 427–447. Springer (2010)
26. Lochbihler, A.: Making the Java memory model safe. ACM Trans. Program. Lang. Syst. **35**(4), 12:1–65 (2014)
27. de Moura, L., Bjørner, N.: Efficient E-matching for SMT solvers. In: F. Pfenning (ed.) CADE-21, *LNCS*, vol. 4603, pp. 183–198. Springer (2007)
28. de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: C.R. Ramakrishnan, J. Rehof (eds.) TACAS 2008, *LNCS*, vol. 4963, pp. 337–340. Springer (2008)
29. Nelson, G., Oppen, D.C.: Simplification by cooperating decision procedures. ACM Trans. Program. Lang. Syst. **1**(2), 245–257 (1979)
30. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL: A Proof Assistant for Higher-Order Logic, *LNCS*, vol. 2283. Springer (2002)
31. Pham, T., Whalen, M.W.: RADA: A tool for reasoning about algebraic data types with abstractions. In: B. Meyer, L. Baresi, M. Mezini (eds.) ESEC/FSE '13, pp. 611–614. ACM (2013)
32. Reynolds, A., Blanchette, J.C.: A decision procedure for (co)datatypes in SMT solvers. In: A. Felty, A. Middeldorp (eds.) CADE-25, *LNCS*, vol. 9195, pp. 197–213. Springer (2015)
33. Reynolds, A., Blanchette, J.C., Tinelli, C.: Model finding for recursive functions in SMT. In: V. Ganesh, D. Jovanović (eds.) SMT 2015 (2015)
34. Reynolds, A., Kuncak, V.: Induction for SMT solvers. In: D. D'Souza, A. Lal, K.G. Larsen (eds.) VMCAI 2015, *LNCS*, vol. 8931, pp. 80–98. Springer (2014)
35. Reynolds, A., Tinelli, C., Goel, A., Krstić, S., Deters, M., Barrett, C.: Quantifier instantiation techniques for finite model finding in SMT. In: M.P. Bonacina (ed.) CADE-24, *LNCS*, vol. 7898, pp. 377–391. Springer (2013)
36. Reynolds, A., Tinelli, C., de Moura, L.: Finding conflicting instances of quantified formulas in SMT. In: FMCAD 2014, pp. 195–202. IEEE (2014)
37. Rutten, J.J.M.M.: Universal coalgebra—A theory of systems. Theor. Comput. Sci. **249**, 3–80 (2000)

38. Stump, A., Sutcliffe, G., Tinelli, C.: StarExec: A cross-community infrastructure for logic solving. In: S. Demri, D. Kapur, C. Weidenbach (eds.) IJCAR 2014, *LNCS*, vol. 8562, pp. 367–373. Springer (2014)
39. Suter, P., Köksal, A.S., Kuncak, V.: Satisfiability modulo recursive programs. In: E. Yahav (ed.) SAS 2011, *LNCS*, vol. 6887, pp. 298–315. Springer (2011)
40. Tinelli, C., Zarba, C.G.: Combining nonstably infinite theories. J. Autom. Reasoning **34**(3), 209–238 (2005)
41. Wand, D.: Polymorphic+typeclass superposition. In: L. de Moura, B. Konev, S. Schulz (eds.) PAAR 2014 (2014)
42. Weber, T.: SAT-based finite model generation for higher-order logic. Ph.D. thesis, Technische Universität München (2008). Available at `http://mediatum.ub.tum.de/doc/676608/file.pdf`