Datatypes with Shared Selectors (Technical Report)

Andrew Reynolds¹, Arjun Viswanathan¹, Haniel Barbosa¹, Cesare Tinelli¹, and Clark Barrett²

¹ University of Iowa, Iowa City, USA
² Department of Computer Science, Stanford University

Abstract. We introduce a new theory of algebraic datatypes where selector symbols can be shared between multiple constructors, thereby reducing the number of terms considered by current SMT-based solving approaches. We show the satisfiability problem for the traditional theory of algebraic datatypes can be reduced to problems where selectors are mapped to shared symbols based on a transformation provided in this paper. The use of shared selectors addresses a key bottleneck for an SMT-based enumerative approach to the Syntax-Guided Synthesis (SyGuS) problem. Our experimental evaluation of an implementation of the new theory in the solver cvc4 on syntax-guided synthesis and other domains shows evidence that the use of shared selectors improves state-of-the-art SMT-based approaches for datatype constraints.

1 Introduction

Algebraic datatypes (also known as inductive or recursive datatypes) are composite types commonly used for expressing finite data structures in computer science applications, such as lists or trees. Reasoning efficiently about datatypes is thus paramount in such fields as program analysis and verification, which has led to numerous approaches for automating solving in this setting. In this paper, we follow the semantic approach introduced by Barrett et al. [10], which is generally the basis for datatype decision procedures in satisfiability modulo theories (SMT) solvers [11].

In traditional presentations of the theory of algebraic datatypes [10,20], a datatype is freely generated by function symbols called constructors, while its elements are accessed with function symbols called selectors, or projections, which are specific for each constructor, each representing one of its arguments. Datatypes also have discriminators, or testers, associated with each constructor. They are predicates indicating whether a given term was built with a specific constructor.

The satisfiability of quantifier-free formulas in the theory of algebraic datatypes is decidable. A basic decision procedure for this problem [10,20] used by a number of SMT solvers operates by progressively unrolling datatypes: it tries to satisfy constraints by guessing top-level constructors in order to build values for the constraint variables incrementally. Concretely, if \( x \) is a datatype variable and \( c \) is an \( n \)-ary constructor for the datatype, the procedure may guess the equality constraint \( x \approx c(x_1, \ldots, x_n) \) where \( x_1, \ldots, x_n \) are fresh variables. If such a choice leads to an inconsistency, the procedure backtracks and tries different constructors until it determines that the constraints are satisfiable or no more choices are possible. During this process, lemmas in the form of
quantifier-free clauses may be learned by the procedure that prevent the procedure from making guesses already shown to be infeasible. However, these lemmas may include selectors, and because each selector is associated with only a single constructor, the generality and hence the usefulness of such lemmas is limited.

To address this limitation we introduce a new theory of datatypes that allows certain selectors to be shared by multiple constructors. Thus, information acquired when reasoning with a previous constructor, i.e., the learned lemmas on the applications of its selectors, can be reused when an argument of the same type is considered in another constructor. We illustrate this point with the following example which will be used as a running example throughout the paper.

Example 1. Consider a binary tree whose internal nodes store either one or two integer values, and whose leaves store both a Boolean and an integer value. A datatype Tree modeling this data structure has three constructors: one (N₁) taking an integer and two Tree elements as arguments, another (N₂) taking two integers and two Tree elements as arguments, and a third (L) taking as arguments a Boolean and an integer element. We write this datatype in the following BNF-style notation:

\[ \text{Tree} = N₁(\text{Int, Tree, Tree}) \mid N₂(\text{Int, Int, Tree, Tree}) \mid L(\text{Bool, Int}) \]

We assume each constructor has selectors associated with them. The subfields (i.e., the immediate subterms) of terms constructed by N₁ are accessed, respectively, by the selectors S^{N₁,1}, S^{N₁,2}, and S^{N₁,3} of type Tree → Int, Tree → Tree and Tree → Tree. The selectors for the other constructors are similar. The subfields of terms constructed by N₂ are accessed by the selectors S^{N₂,1}, S^{N₂,2}, S^{N₂,3}, and S^{N₂,4}; the subfields of L by S^{L,1} and S^{L,2}. We also assume each constructor is associated with a tester predicate, i.e. isN₁, isN₂, and isL, each of which takes a Tree as an argument. Given term t of type Tree, consider the following set of clauses:

\[ \{ \neg \text{isN₁}(t) \lor S^{N₁,1}(t) \geq 0, \neg \text{isL}(t) \lor S^{L,2}(t) \geq 0 \} \]  \hspace{1cm} (1)

The first clause states that when t has top symbol N₁, then its first subfield (which is of integer type) is non-negative. Similarly, the second says when t has top symbol L, its second subfield is non-negative.

Consider now a different kind of selector symbol S^{Int,1} of type Tree → Int which maps each value of type Tree to the first (i.e., leftmost) subfield of t of type Int, regardless of the top constructor symbol of t. We will refer to such selectors as shared selectors. While this datatype requires nine selectors in the standard sense, five shared selectors suffice to access all possible subfields of a value of type Tree: two to access the Tree subfields, two to access the Int subfields, and one to access the Bool subfield of L. In particular, clause set [1] can be rewritten as follows using the shared selector S^{Int,1}:

\[ \{ \neg \text{isN₁}(t) \lor S^{\text{Int,1}}(t) \geq 0, \neg \text{isL}(t) \lor S^{\text{Int,1}}(t) \geq 0 \} \]  \hspace{1cm} (2)

stating that when t has top symbol N₁ or L, its first integer child is non-negative.

In Example 1 the second set of clauses had one unique arithmetic constraint whereas the first set had two. In practice, reducing the number of unique constraints can substantially improve the performance of SMT solvers. Our experiments provide evidence
that shared selectors lead to a significant reduction in the number of unique constraints for several classes of benchmarks from real applications, with resulting SMT solver performance improvements that are proportional to the magnitude of this reduction.

**Contributions** We introduce a conservative extension of the (generic) theory of algebraic datatypes that features shared selectors. We show how using shared selectors instead of standard (unshared) selectors can improve the performance of current satisfiability procedures for the theory and also, as a result, the performance of procedures for syntax-guided synthesis. Specifically:

1. We formalize the new theory and show that constraints in the original signature can be reduced to equisatisfiable constraints, all of whose selectors are shared selectors. We present a decision procedure for the satisfiability of quantifier-free formulas in this theory, which is a modification of an earlier procedure for datatypes [20].
2. We provide details on an SMT-based approach for syntax-guided synthesis [22], and demonstrate how this approach can significantly benefit from native support in the SMT solver for a theory of datatypes with shared selectors.
3. We present an extensive experimental evaluation of our implementation in the SMT solver cvc4 [7] on benchmarks from SMT-LIB [8] and from the most recent edition of SyGuS-COMP [3], the syntax-guided synthesis competition. This evaluation shows that the use of shared selectors can reduce the number of terms introduced during solving, thus leading to more problems solved with respect to the state of the art.

**Outline** After preliminaries in Section 2, we formalize the new theory of datatypes in Section 3. In Section 3.2, we define a satisfiability-preserving transformation between (datatype) constraints containing only standard selectors and constraints containing only shared selectors. In Section 4, we present a decision procedure for satisfiability in the theory based on a preliminary elimination of standard selectors. Section 5 introduces syntax-guided synthesis and explains how this application benefits from datatypes with shared selectors. Section 6 presents our experimental evaluation. We cover related work in Section 7 and offer concluding remarks in Section 8.

## 2 Preliminaries

Our setting is a many-sorted classical first-order logic similar in essence to the one adopted by the SMT-LIB standard [9]. A signature $\Sigma = (\mathcal{Y}, \mathcal{F})$ consists of a set $\mathcal{Y}$ of first-order types, or sorts, and a set $\mathcal{F}$ of first-order function symbols over these types. Each symbol $f \in \mathcal{F}$ is associated with a list $\tau_1, \ldots, \tau_n$ of argument types and a return type $\tau$, written $f : \tau_1 \times \cdots \times \tau_n \rightarrow \tau$ or just $f : \tau$ if $n = 0$. The function arity($f$) returns $n$. We assume that any signature contains a $\text{Bool}$ type and constants true, false: $\text{Bool}$, a family $(\approx : \tau \times \tau \rightarrow \text{Bool})_{\tau \in \mathcal{Y}}$ of equality symbols, a family $(\text{ite} : \text{Bool} \times \tau \times \tau \rightarrow \tau)_{\tau \in \mathcal{Y}}$ of if-then-else symbols, and the Boolean connectives $\neg, \land, \lor$ with their expected types. Function symbols of $\text{Bool}$ return type play the role of predicate symbols.

Typed terms are built as usual over function symbols from $\mathcal{F}$ and typed variables from a fixed family $(\mathcal{V}_\tau)_{\tau \in \mathcal{Y}}$ of pairwise-disjoint infinite sets. Formulas are terms of
type \texttt{Bool}. Equivalence is equality ($\approx$) on \texttt{Bool}. The syntax $t \neq u$ is short for $\neg(t \approx u)$. We reserve the names $a, c, f, g, p, q$ for function symbols; $x, y, z$ for variables; $r, s, t, u$ for terms (which may be formulas); and $\phi, \psi$ for formulas. We use the symbol $=$ for equality at the meta-level. We write $t^\tau$ to indicate that $t$ is a term of type $\tau$. The set of all terms occurring in a term $t$ is denoted by $T(t)$. When convenient, we write an enumeration of (meta)symbols $a_1, \ldots, a_n$ as $\bar{a}$. If $b_1, \ldots, b_k$ is another enumeration, $\bar{a} \bar{b}$ denotes the enumeration $a_1, \ldots, a_n, b_1, \ldots, b_k$.

Given a signature $\Sigma = (\mathcal{Y}, \mathcal{F})$, a $\Sigma$-interpretation $I$ maps: each $\tau \in \mathcal{Y}$ to a non-empty set $\tau^I$, the domain of $\tau$ in $I$, with $\text{Bool}^I = \{\top, \bot\}$; each $x \in \mathcal{V}_\tau$ to an element of $\tau^I$; each $f \in \mathcal{F}$ s.t. $f : \tau_1 \times \cdots \times \tau_n \rightarrow \tau$ to a total function $u^I : \tau_1^I \times \cdots \times \tau_n^I \rightarrow \tau^I$ when $n > 0$ and to an element of $\tau^I$ when $n = 0$, with true$^I = \top$ and false$^I = \bot$. The interpretation $I$ induces as usual a mapping from terms $t$ of type $\tau$ to elements $t^I$ of $\tau^I$. If $x_1, \ldots, x_n$ are variables and $v_1, \ldots, v_n$ are well-typed values for them, we denote by $I[x_1 \mapsto v_1, \ldots, x_n \mapsto v_n]$ the $\Sigma$-interpretation that maps each $x_i$ to $v_i$, and is otherwise identical to $I$. A satisfiability relation between $\Sigma$-interpretations and $\Sigma$-formulas is defined inductively as usual.

A theory is a pair $\mathcal{T} = (\Sigma, I)$ where $\Sigma$ is a signature and $I$ is a non-empty class of $\Sigma$-interpretations, the models of $\mathcal{T}$, that is closed under variable reassignment (i.e., every $\Sigma$-interpretation that differs from one in $I$ only in how it interprets the variables is also in $I$) and isomorphism. A $\Sigma$-formula $\phi$ is $\mathcal{T}$-satisfiable (respectively $\mathcal{T}$-unsatisfiable) if it is satisfied by some (resp., no) interpretation in $I$. A satisfying interpretation for $\phi$ models (or is a model of) $\phi$. A formula $\phi$ is valid in $\mathcal{T}$ (or $\mathcal{T}$-valid), written $\models_{\mathcal{T}} \phi$, if every model of $\mathcal{T}$ is a model of $\phi$.

### 3 Theory of Datatypes With Shared Selectors

In this section, we consider a theory $\mathcal{D}$ of algebraic datatypes over some signature $\Sigma = (\mathcal{Y}, \mathcal{F})$ and then extend it conservatively to an expanded signature with shared selectors. The terms of $\mathcal{D}$ are quantifier-free. As a technical convenience, we treat free variables as constants in a suitable expansion of $\Sigma$. The types of $\mathcal{D}$ are partitioned into a set of datatypes $\mathcal{Y}_d$, and a set of other types $\mathcal{Y}_{ord}$. We use the metavariables $\delta, \epsilon$ to refer to datatypes and $\tau, \upsilon$ for arbitrary first-order types. Each datatype $\delta$ is equipped with one or more constructors, distinguished function symbols from $\mathcal{F}$ with return type $\delta$. For every argument $k$ of a constructor $C : \tau_1 \ldots \tau_n \rightarrow \delta$ for $\delta$, we assume $\mathcal{F}$ contains a (standard) selector $\delta_k^{C,k} : \delta \rightarrow \tau_k$. We omit $\delta$ from the selector name when it is understood or not important. We refer the reader to the SMT-LIB 2 reference document [9] or Barrett et al. [10] for a formal definition of this theory.$^1$ We recall salient properties of its symbols as needed.

To start, each interpretation of a datatype in the theory, when reduced to the datatype constructors, is isomorphic to a term (or Herbrand) algebra. Concretely, this means that if $\delta \in \mathcal{Y}_d$ is a datatype whose constructors are $\{C_1, \ldots, C_n\}$, then the following formulas

---

$^1$ The two references differ on how they make selectors (which are naturally partial functions) total. We follow the SMT-LIB 2 standard here.
are all $D$-valid for all distinct $i, j \in \{1, \ldots, m\}$
\[
\forall x_1, \ldots, x_n, z_1, \ldots, z_p, C_i(x_1, \ldots, x_n) \neq C_j(z_1, \ldots, z_p) \quad \text{(Distinctness)}
\]
\[
\forall x_1, \ldots, x_n, z_1, \ldots, z_n.
\]
\[
C_i(x_1, \ldots, x_n) \equiv C_i(z_1, \ldots, z_n) \rightarrow x_1 \approx z_1 \land \ldots \land x_n \approx z_n
\]
\[
\forall x. \, \text{isC}_i(x) \lor \cdots \lor \text{isC}_n(x) \quad \text{(Exhaustiveness)}
\]
Above, we write $\text{isC}_i(t)$ to denote the predicate that holds if and only if the top symbol of $t$ is $C_i$. Strictly speaking, we do not need to extend our signature with the symbols $\text{isC}_i(t)$, called discriminators or testers in the literature, as $\text{isC}(t)$ can be considered an abbreviation for the equality $t \approx C(S^{C_1}(t), \ldots, S^{C_m}(t))$ where $m = \text{arity}(C)$. Moreover, interpretations must also respect acyclicity, which states that constructor terms cannot be equal to any of their proper subterms.

Since all models of $D$ interpret a datatype $\delta$ as the same Herbrand algebra, we will say that $\delta$ is finite if that algebra is finite. For simplicity, we will assume that every type $\tau$ in $D$ that is not a datatype is interpreted as an infinite set in every model of $D$. This is not a strong restriction in practice, since types with some fixed, finite cardinality $n$ can be treated as datatypes with $n$ nullary constructors.

The relationship between an $n$-ary constructor $C$ and each of its selectors $S^{C,k}$ with $k = 1, \ldots, n$ is captured by the following $D$-valid formula:
\[
\forall x_1, \ldots, x_n, S^{C,k}_\delta(C(x_1, \ldots, x_n)) \approx x_k \quad \text{(Standard selection)}
\]

### 3.1 Shared Selectors

We extend the signature of $D$ with additional selectors which we call shared selectors and denote as $S^{\tau,k}_\delta$, for each datatype $\delta$ and type $\tau$ in $D$ and each natural number $k$.

Intuitively, a shared selector $S^{\tau,k}_\delta$ for $\delta$, when applied to a $\delta$-term $C(t_1, \ldots, t_n)$ returns the $k$-th argument of $C$ that has type $\tau$, if one exists.

**Example 2.** Consider again the Tree datatype introduced in Example 1:

$$
\text{Tree} = N_1(\text{Int, Tree, Tree}) \mid N_2(\text{Int, Int, Tree, Tree}) \mid L(\text{Bool, Int})
$$

For term
\[
t = N_1(1, N_2(2, 3, L(\top, 4), L(\bot, 5)), L(\top, 6))
\]
the equalities $S^{\text{Int,1}}_\delta(t) \approx 1$, $S^{\text{Int,2}}_\delta(S^{\text{Tree,1}}_\delta(t)) \approx 3$, and $S^{\text{Int,1}}_\delta(S^{\text{Tree,2}}_\delta(S^{\text{Tree,1}}_\delta(t))) \approx 6$ are all valid in our extension of $D$ to shared selectors.

To define shared selectors formally, let us first define a partial function $\text{stoa}$ (for selector to argument) that takes as input a natural $k$, a type $\tau$, and a constructor $C$, and returns the index of the $k$-th argument of $C$ of type $\tau$. We leave $\text{stoa}$ undefined if $C$ has fewer than $k$ arguments of type $\tau$.

**Example 3.** For the Tree datatype, $\text{stoa}(1, \text{Int, N}_1) = 1$, $\text{stoa}(2, \text{Tree, N}_1) = 3$ and $\text{stoa}(1, \text{Int, L}) = 2$, whereas $\text{stoa}(2, \text{Int, N}_1)$, $\text{stoa}(1, \text{Bool, N}_2)$, and $\text{stoa}(1, \text{Tree, L})$ are undefined.
$H(t, M) = \text{match } t \text{ with }$

\[
\begin{align*}
    x &\rightarrow x \\
    C(t_1, \ldots, t_n) &\rightarrow C(H(t_1, M), \ldots, H(t_n, M)) \\
    S^r_{\delta}(t_1) &\rightarrow S^r_{\delta}(H(t_1, M)) \\
    S^c_{\delta}(t_1) &\rightarrow \begin{cases} S^{r, \text{atos}(\tau, C, k)}_{\delta}(H(t_1, M)) & \text{if } M(t_1) = C \\
    S^{r, \text{err}(C, k)}_{\delta}(H(t_1, M)) & \text{otherwise} \end{cases}
\end{align*}
\]

where $S^c_{\delta} : \delta \rightarrow \tau$

\[\text{Fig. 1: Definition of } H(t, M)\]

More formally, in our extension of theory $\mathcal{D}$ with shared selectors, which we also refer to as $\mathcal{D}$ for convenience, the following holds for all datatypes $\delta$, constructors $C$ of $\delta$, and shared selectors $S^r_{\tau}$, whenever $\text{stoa}(k, \tau, C)$ is defined:

\[\forall x_1, \ldots, x_n. S^r_{\delta}(C(x_1, \ldots, x_n)) \approx x_i, \text{ where } i = \text{stoa}(k, \tau, C) \quad (\text{Shared selection})\]

It is not difficult to argue that every $\Sigma$-formula $\varphi$ without shared selectors is valid in the extended theory if and only if it is valid in the original theory.

### 3.2 From Standard Selectors to Shared Selectors

The satisfiability problem for constraints, i.e., finite sets of literals, over the original theory of datatypes (without shared selectors) is decidable [10]. In this section, we introduce a transformation $H$ that reduces arbitrary constraints in our extended theory $\mathcal{D}$ (which may have both standard and shared selectors) to constraints with no standard selectors. Applying this transformation as an initial step allows us to determine the satisfiability of arbitrary $\Sigma$-constraints by means of a decision procedure for $\Sigma$-constraints involving shared selectors only.

To define this transformation, let $\max_{\Sigma}$ denote some natural number that is greater than the arity of all constructors in $\Sigma$. We define the dual of the $\text{stoa}$ function from Subsection 3.1 as the partial function $\text{atos}$ (for argument to selector) that takes as input a type $\tau$, a constructor $C : \tau_1 \times \cdots \times \tau_n \rightarrow \delta$, and a natural number $k \leq n$, and returns the number of times $\tau$ occurs in $\tau_1, \ldots, \tau_k$.

Figure 1 defines the transformation $H$, which takes as arguments a $\Sigma$-term $t$, and a mapping $M$ that consists of an entry of the form $s \mapsto C$ for each datatype term $s$ in $T(t)$ where $C$ is one of the constructors for the type of $s$. Without loss of generality, we assume that all applications of shared selectors $S^r_{\delta}$ occurring in $t$ are such that $k < \max_{\Sigma}$. The transformation $H$ leaves variables unchanged; for terms whose top symbol is a constructor or a shared selector, $H$ behaves homorphically. For terms $t$ with a standard selector $S^c_{\delta} : \delta \rightarrow \tau$ as top symbol, we distinguish whether the argument $t_1$ is mapped to $C$ by $M$ or not. In the first case, we replace $S^c_{\delta}$ by the shared selector

\[\text{Fig. 1: Definition of } H(t, M)\]
The above construction is well-defined due to our definition of \( err \). In particular, \( err(C, k) \) is defined uniquely for each (constructor, natural number) pair. We show \( H(t, M_I) \) is well-defined for all terms \( t \in T(E) \) by structural induction on \( t \).

**Theorem 1.** \( E \) is \( \mathcal{D} \)-satisfiable iff \( H(E, M) \) is \( \mathcal{D} \)-satisfiable for some \( M \).

**Proof.** We split the statement into its two implications. The proof relies on the construction of a mapping \( M \) from a model of \( E \).

\( \Rightarrow \) If \( E \) is satisfied by some \( \Sigma \)-model \( I \) of \( \mathcal{D} \), there exists a mapping \( M_I \) and \( \Sigma \)-model \( J_I \) of \( \mathcal{D} \) such that \( H(E, M_I) \) is satisfied by \( J_I \). We show this by a particular construction for \( M_I \) and \( J_I \). Let the mapping \( M_I \) be \( t \mapsto C \mid \models \text{isC}(t), t \in T(E) \). Construct \( J_I \) as follows. First, all types \( \tau \) and constructors are interpreted by \( J_I \) the same as in \( I \). Furthermore, we interpret all variables and standard selectors in \( J_I \) the same as in \( I \). It remains to state how shared selectors are interpreted in \( J_I \). Notice that our transformation generates shared selectors of the form \( S_{\delta, \text{err}(\tau, C, k)}^T \). We distinguish these in the following construction.

\[ S_{\delta, \text{err}(\tau, C, k)}^T = S_{\delta}^{C, k} \] and \( S_{\delta}^{\tau, k} = S_{\delta}^{\tau, k} \) for all other shared selectors.

The above construction is well-defined due to our definition of \( err \). In particular, \( err(C, k) \) is defined uniquely for each (constructor, natural number) pair. We show \( H(t, M_I) \) is well-defined for all terms \( t \in T(E) \) by structural induction on \( t \).
– Case \( t = x \), for some variable \( x \): By the definition of \( \mathcal{H} \) and the construction of \( \mathcal{J} \), we have that \( \mathcal{H}(x, M_F)^J = x^J = x^J \).

\[ \square \]

– Case \( t = C(t_1, ..., t_n) \): We have that

\[
\mathcal{H}(C(t_1, ..., t_n), M_F)^J \\
= C^J(\mathcal{H}(t_1, M_F)^J, ..., \mathcal{H}(t_n, M_F)^J) \text{ by the definition of } \mathcal{H} \\
= C^J(t_1^J, ..., t_n^J) \text{ by the induction hypothesis} \\
= C^J(t_1^J, ..., t_n^J), \text{ by the construction of } \mathcal{J} \\
= C(t_1, ..., t_n)^J.
\]

– Case \( t = S_{\delta}^{\tau,k}(t_1) \): Firstly,

\[
\mathcal{H}(S_{\delta}^{\tau,k}(t_1), M_F)^J \\
= (S_{\delta}^{\tau,k})^J(\mathcal{H}(t_1, M_F)^J) \text{ by the definition of } \mathcal{H} \\
= (S_{\delta}^{\tau,k})^J(t_1^J) \text{ by the induction hypothesis}
\]

By assumption, we have that \( k < \text{max}_S \). Hence, by the construction of \( \mathcal{J} \), \( (S_{\delta}^{\tau,k})^J(t_1^J) = (S_{\delta}^{\tau,k})^J(t_1^J) \), which is \( S_{\delta}^{\tau,k}(t_1) \).

– Case \( t = S_{\delta}^{\tau,k}(t_1) \) where \( S_{\delta}^{\tau,k} : \delta \to \tau \):

• If \( M_F(t_1) = C \), then

\[
\mathcal{H}(S_{\delta}^{\tau,k}(t_1), M_F)^J \\
= (S_{\delta}^{\tau,\text{ato}(\tau, C, k)})^J(\mathcal{H}(t_1, M_F)^J) \text{ by the definition of } \mathcal{H} \\
= (S_{\delta}^{\tau,\text{ato}(\tau, C, k)})^J(t_1^J) \text{ by the induction hypothesis}
\]

By definition of ato and since \( k \) is a valid argument position of \( C \), we have that \( \text{ato}(\tau, C, k) \leq \text{arity}(C) < \text{max}_S \). Hence, by the construction of \( \mathcal{J} \), we have that \( (S_{\delta}^{\tau,\text{ato}(\tau, C, k)})^J(t_1^J) = (S_{\delta}^{\tau,\text{ato}(\tau, C, k)})^J(t_1^J) \). Since \( M_F(t_1) = C \), by construction of \( M_F \), we have that \( t_1^J \) is a term of the form \( C(s_1, ..., s_n) \), and hence \( (S_{\delta}^{\tau,\text{ato}(\tau, C, k)})^J(t_1^J) = \text{ato}(\text{ato}(\tau, C, k) \tau, C) = s_k \). Furthermore, \( s_k = S_{\delta}^{\tau,k}(t_1) \).

• If \( M_F(t_1) \neq C \), then

\[
\mathcal{H}(S_{\delta}^{\tau,k}(t_1), M_F)^J \\
= (S_{\delta}^{\tau,\text{err}(C,k)})^J(\mathcal{H}(t_1, M_F)^J) \text{ by the definition of } \mathcal{H}, \\
= (S_{\delta}^{\tau,\text{err}(C,k)})^J(t_1^J) \text{ by the induction hypothesis}, \\
= (S_{\delta}^{\tau,k})^J(t_1^J) \text{ by the construction of } \mathcal{J}, \\
= S_{\delta}^{\tau,k}(t_1)^J. \\
\]

\[ \square \]
Since $I$ satisfies $E$ and since $\mathcal{H}(t, M_I)^J = t^I$ for all terms $t \in T(E)$, we have that $J$ satisfies the equalities and disequalities in $\mathcal{H}(E, M_I)$ of the form $(\neg)\mathcal{H}(t_1, M_I) \approx \mathcal{H}(t_2, M_I)$. By construction of $M_I$, we have that $J$ satisfies the constraints in $\mathcal{H}(E, M_I)$ of the form $isC(t)$ where $t \mapsto C \in M_I$. Hence, $J$ satisfies $\mathcal{H}(E, M_I)$.

"⇐": If $\mathcal{H}(E, M_I)$ is satisfied by some $\Sigma$-model $J$ of $D$ for some mapping $M$, then $E$ is satisfied by some $\Sigma$-model $I$ of $D$. We show this by constructing $I$ as follows. First, all types, constructors, variables and have the same interpretation in $I$ as in $J$. Furthermore, all shared selectors have the same interpretation in $I$ as in $J$. We interpret standard selectors in $I$ as follows.

\[
S^C_{\delta}(t_1)^I = \begin{cases} 
S^\tau_{\delta, atos(\tau, C, k)}(t_1)^I & \text{if } M(t) = C \\
S^\tau_{\delta, err(C, k)}(t_1)^I & \text{otherwise}
\end{cases}
\]

We show $t^I = \mathcal{H}(t, M_I)^J$ for all terms $t \in T(E)$ by structural induction on $t$ as follows.

- Case $t = x$, for some variable $x$: By the definition of $\mathcal{H}$ and the construction of $J$, we have that $\mathcal{H}(x, M_I)^J = x^J = x^I$. \hfill \Box$
- Case $t = C(t_1, \ldots, t_n)$: We have that
  \[
  \mathcal{H}(C(t_1, \ldots, t_n), M_I)^J \\
  = C^J(\mathcal{H}(t_1, M_I)^J, \ldots, \mathcal{H}(t_n, M_I)^J) \text{ by the definition of } \mathcal{H}, \\
  = C^I(t_1^I, \ldots, t_n^I) \text{ by the induction hypothesis}, \\
  = C(t_1^I, \ldots, t_n^I), \text{ by the construction of } I \\
  = C(t_1, \ldots, t_n)^I.
  
  \hfill \Box$
- Case $t = S^\tau_{\delta, k}(t_1)$:
  \[
  \mathcal{H}(S^\tau_{\delta, k}(t_1), M_I)^J \\
  = (S^\tau_{\delta, k})^J(\mathcal{H}(t_1, M_I)^J) \text{ by the definition of } \mathcal{H}, \\
  = (S^\tau_{\delta, k})^I(t_1^I) \text{ by the induction hypothesis}, \\
  = (S^\tau_{\delta, k})^I(t_1^I) \text{ by the construction of } I, \\
  = S^\tau_{\delta, k}(t_1)^I
  \]
  \hfill \Box$
- Case $t = S^C_{\delta}(t_1)$ where $S^C_{\delta} : \delta \to \tau$: 

9
Corollary 1. For some index sets $I$ and $J$, and set $E$ of $\Sigma$-literals without standard selectors, let

$$E_0 = E \cup \{ \delta_{c_i}(x_i) \approx y_i \mid i \in I, j \in J \}$$

and

$$E_1 = E \land \{ \text{iter}(\delta_{c_i}(x_i), r_{alsi}(r, c_i, k_i))(x_i), \delta_{\text{err}}(c_i, k_i)(x_i) \approx y_i \mid i \in I, j \in J \}.$$

The sets $E_0$ and $E_1$ are equisatisfiable in $D$.

Proof. For an interpretation $I$, then let $M_I$ be the mapping $[t \mapsto C | I \models \text{isC}(t), t \in T(E_0 \cup E_1)].$ We first show that $I$ satisfies $\mathcal{H}(E_0, M_I)$ if and only if $I$ satisfies $E_1$.

First, for each (dis)equality $(\sim)t_1 \approx t_2 \in E$, we have that $\mathcal{H}(E_0, M_I)$ contains a (dis)equality of the form $(\sim)\mathcal{H}(t_1, M_I) \approx \mathcal{H}(t_2, M_I)$, which by definition of $\mathcal{H}$ is equivalent to $(\sim)t_1 \approx t_2$, since $E$ does not contain standard selectors. Second, by definition of $M_I$, we have that $I \models [\text{isC}(t) \mid t \mapsto C \in M_I]$. Since $\mathcal{H}(y_i, M_I) = y_i$ for all $y_i$, it suffices to show $\mathcal{H}(\delta_{c_i}(x_i), r_{alsi}(r, c_i, k_i))(x_i)$ is equal to $\text{iter}(\delta_{c_i}(x_i), r_{alsi}(r, c_i, k_i))(x_i)$, $\delta_{\text{err}}(c_i, k_i)(x_i)$ is equal to $\text{iter}(\delta_{c_i}(x_i), \delta_{\text{err}}(c_i, k_i))(x_i)$, and $\delta_{\text{err}}(c_i, k_i)(x_i)$ is equal to $\text{iter}(\delta_{c_i}(x_i), \delta_{\text{err}}(c_i, k_i))(x_i)$. Hence, $\mathcal{H}(\delta_{c_i}(x_i), M_I)$ is equal to $\text{iter}(\delta_{c_i}(x_i), \delta_{\text{err}}(c_i, k_i))(x_i)$.
Thus, $I$ satisfies $\mathcal{H}(E_0, M_I)$ if and only if $I$ satisfies $E_1$. Thus, by Theorem 1, we have that $E_0$ is satisfiable if and only if $E_1$ is satisfiable.

Using this corollary, we can reduce (possibly after some literal flattening) the satisfiability of an arbitrary set of $\Sigma$-constraints $E_0$ to a set of $\Sigma$-constraints $E_1$ not containing standard selectors. In particular, our implementation in the SMT solver cvc4 replaces each application of the form $S^{c_{ji}, k_i}(x_i)$ by the term $\text{ite}(\text{isC}_{ji}(x_i), S^{\tau, \text{axC}(\tau, c_{ji}, k_i)}(x_i), S^{\tau, \text{err}(C_{ji}, k_i)}(x_i))$ during a preprocessing pass on the input formula.

### 4 Decision Procedure for Datatypes with Shared Selectors

This section describes a tableau-like calculus for deciding constraint satisfiability in $D$ with constraint variables interpreted existentially. The calculus is parametrized by the theory’s signature $\Sigma$. By the results of the previous section, we can restrict with no loss of generality the input language to sets of equalities and disequalities between $\Sigma$-terms with no standard selectors and no discriminators. Since our calculus is based on similar calculi for datatypes that have been presented in detail in previous work [10, 20], we focus on our modifications to accommodate shared selectors.

The derivation rules of the calculus operate on a current set $E$ of constraints as specified in Figure 2. A derivation rule can be applied to $E$ if its premises are met. Some of those premises check membership in the congruence closure $E^*$ of $E$, the smallest superset of $E$ that is closed under entailment in the theory of equality. A rule’s conclusion either modifies $E$ or replaces it by $\bot$ to indicate unsatisfiability. The Split rule has multiple alternative conclusions, denoting branching. The rules are applied to build a derivation tree, i.e., a tree whose nodes are finite sets of (dis)equalities, with an initial constraint set $E_0$ as its root and child nodes obtained by a non-redundant application of a derivation rule to the parent. A derivation tree is generated progressively from $E_0$ by applying a derivation rule to a leaf node. We say that $E_0$ has a derivation tree $D$ if $E_0$ is the root of $D$. An application of a rule is redundant if (one of) its conclusion(s) leaves $E$ unchanged. A node is saturated if it admits only redundant rule applications. A derivation tree is closed if all of its leaf nodes are $\bot$. Intuitively, rules are applied until the derivation tree being generated is closed (indicating that the initial set $E_0$ is $D$-unsat) or a saturated leaf node is found (indicating that $E_0$ is $D$-sat).

In the calculus, all reasoning based on the general properties of equality is encapsulated in the rule Conflict, which detects that congruent terms are forced to be distinct. The remaining rules perform datatype reasoning proper, with Decompose computing a downward equality closure based on the injectivity of constructors and Clash detecting failures based on the distinctness of constructors. The Cycle rule recognizes when a constructor term must be equivalent to one of its subterms, which is forbidden in all models of the theory.

The procedure also incrementally unrolls terms by branching on different constructors, with the Split rule performing case distinctions on constructors for various terms occurring in $E$. The main modification from the previous calculi for the theory of

---

2 Such tests are effective by well-known results about the theory of equality [6].
Proposition 1 (Termination). All derivation trees in the calculus are finite.

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

\[
S_t^0 = \{ t \} \quad S_t^{i+1} = S_t^i \cup \{ S_\delta^{\tau, n}(u) \mid u^\delta \in S_t^i, \delta \in \mathcal{J}_{dr}, |\delta| \text{ is finite}, S_\delta^{\tau, n} \in \mathcal{F}_\text{set} \}
\]

and let \( S_t^\infty \) be the limit of this sequence. This is a finite set for each \( t \), since all selector chains applied to \( t \) are finite. Let \( S_t^\infty \) be the union of all sets \( S_t^\infty \) where \( t \in D \), and let

\[
T^\infty(E) = T\left( E \cup \left\{ C_i(S_\delta^{\tau, \text{ato}(\tau_i, C_i, 1)}(t), \ldots, S_\delta^{\tau_i, \text{ato}(\tau_i, C_i, n_i)}(t)) \mid t^\delta \in N \cup S_t^\infty, C_i \in \mathcal{F}_\text{cir}^{\delta} \right\} \right)
\]
In a derivation tree with root node $E$, it can be shown by induction on the rules of the calculus that each non-root nor $\bot$-node $F$ is such that $T(F) \subseteq T^{\infty}(E)$, and hence contains an equality between two terms from $T^{\infty}(E)$ not occurring in its parent node. Thus, the depth of a branch in a derivation tree with root node $E$ is at most $|T^{\infty}(E)|^2$, which is finite since $T^{\infty}(E)$ is finite.

**Proposition 2 (Refutation Soundness).** If a constraint set $E_0$ has a closed derivation tree, then $E_0$ is $D$-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 Cycle, then $E$ is trivially $D$-unsatisfiable, due to equality reasoning and for $D$ not accepting models in which distinctness is violated or a constructor term has the same value as one of its subterms, respectively. If a child node of $E$ is a closed derivation tree whose root node $E \cup t \approx u$ is obtained by applying Inject, since $C_1(t) \approx C_0(i) \in E^*$, by injectivity and equality reasoning, $E \models t \approx u$. Thus, by the induction hypothesis, $E \cup t \approx u$ is $D$-unsatisfiable and thus $E$ is $D$-unsatisfiable.

The remaining case is that child nodes of $E$ are closed derivation trees whose root nodes are the result of applying Split on a term $t^\delta$. By the induction hypothesis $E \cup t \approx C_i(S_{\delta}^{\tau_1,\alpha}(t), \ldots, S_{\delta}^{\tau_n,\alpha}(t))$ is $D$-unsatisfiable for each $C_i \in \mathcal{F}_i^{\delta}$.

By exhaustiveness all models of $D$ entail exactly one $t \approx C_i(S_{\delta}^{\tau_1,\alpha}(t), \ldots, S_{\delta}^{\tau_n,\alpha}(t))$. Since Theorem 1 guarantees that in a model $I$ of $D$ in which $t$ is interpreted as a term constructed with $C_i$ it holds that $S_{\delta}^{C_i,t}(I) = S_{\delta}^{C_i,t}(I)^I$, for $1 \leq j \leq n$, then we can conclude that $E$ is $D$-unsatisfiable.

**Proposition 3 (Solution Soundness).** If a constraint set $E_0$ has a derivation tree with a saturated node, then $E_0$ is $D$-satisfiable.

**Proof.** The proof relies on the construction of a specific term-generated interpretation $I$ from the set of equality literals $F$ in a saturated node of a tree whose root is $E$. We will show that by construction $I$ models $E$.

We build $I$ by assigning to terms as their values constructor terms from their congruence classes in $F^\ast$. Since Inject and Clash cannot be applied on $F$, no congruence class in $F^\ast$ contains more than one constructor term modulo congruence, i.e. no two terms with different constructors $C_1(t)$ and $C_2(\bar{u})$ and all terms $C_1(\bar{t}_1), \ldots, C_1(\bar{t}_n)$ in the same class are equivalent modulo congruence, since $\bar{t}_1, \ldots, \bar{t}_n$ are also congruent. Since Cycle cannot be applied on $F$, each term modulo congruence is also acyclic. Finally, since Split cannot be applied as well, every term of the form $S_{\delta}^{C_i,t}(I)$ is such that $t$ contains a constructor term in its congruence class.

To finish building $I$ it remains to specify how it interprets terms $t$ whose congruence classes do not contain constructor terms. If $t$ is not a selector application, it can be assigned any distinct value from its respective domain in $I$ according to two conditions: the value has not been assigned before to a term in a congruence class; and it is distinct from the resulting values computed, after such an assignment, to previously unassigned terms. The first condition can always be trivially satisfied since if a term is not congruent to a constructor term, it must have an infinite type, and therefore there are infinitely many distinct values to be assigned. To satisfy the second condition it suffices to perform the
assignment by adding an equality to $F$, derive a saturated node $F'$, which is always different from $\bot$, and perform the next assignment with a distinct value according to constructor terms of $F'$ modulo congruence.

In the case that $t$ is a selector application $S^\tau_k(u)$, we distinguish two cases, depending on the congruence class of $u$ containing a constructor term modulo congruence $C(u_1, \ldots, u_n)$ with $\text{stoi}(k, \tau, C)$ being defined or not. If it is defined, $S^\tau_k(u)' = \text{stoi}(k, \tau, C)'$. Otherwise we apply an analogous process as before of assigning a distinct value for $S^\tau_k(u)$.

It is easy to show that $I$ satisfies distinctness, injectivity, exhaustiveness, shared selection, and acyclicity by construction. Therefore it satisfies all properties of a $\mathcal{D}$-model. Finally, we show that $I$ satisfies all equalities and disequalities in $F$. Since $I$ can be successfully built from any saturated node by assigning the same value to congruent terms, it trivially satisfies all equalities in $F$. Since Conflict cannot be applied, no two congruent terms occur in a disequality in $F$. Since each congruence class is assigned a distinct value by $I$, it satisfies all disequalities. Therefore $I$ is a model of $F$, and since $E \subseteq F$, $I$ is also a model of $E$. 

**Theorem 2.** Constraint satisfiability in the theory $\mathcal{D}$ of datatypes with shared selectors is decidable.

**Proof.** Completeness, i.e. derivation trees with root $E$ are closed if $E$ is $\mathcal{D}$-unsatisfiable or have a saturated node if $E$ is $\mathcal{D}$-satisfiable, is a direct consequence of Propositions 1, 2, and 3. Therefore, since the rules from Figure 2 are sound and complete, the calculus constitutes a decision procedure for the satisfiability of $\Sigma$-constraints. 

5 Using Shared Selectors for Syntax-Guided Synthesis

In this section, we show how the theory of datatypes with shared selectors can substantially improve the performance of an approach by Reynolds et al. [22] for performing syntax-guided synthesis (SyGuS) [1] directly within an SMT solver.

Syntax-guided synthesis is the problem of automatically synthesizing a stateless, recursion-free program that satisfies a given specification, but with the addition of explicit syntactic restrictions on the solution space. These restrictions specify that the function must be built with selected operators over basic types (such as arithmetic and Boolean operators) and belong to the language generated by a given grammar. Grammars allow users to specify formally a set of candidates for the desired function, thus reducing the search effort of a SyGuS solver.

More technically, a syntax-guided synthesis problem for a function $f$ in a background theory $T$ of the basic types consists of:

1. a set of semantic restrictions, or specification, given by a (second-order) $T$-formula of the form $\exists f. \forall \bar{x}. \varphi[f, \bar{x}]$, and
2. a set of syntactic restrictions given by a grammar $R$.

A solution for $f$ is a lambda term $\lambda \bar{y}. e$ of the same type as $f$, such that (i) $\forall \bar{x}. \varphi[\lambda \bar{y}. e, \bar{x}]$ is valid in $T$ (modulo beta-reductions) and (ii) $e$ is in the language generated by $R$. 

14
The SMT solver cvc4 incorporates a SyGuS solver that automatically encodes
the solution space of a SyGuS problem as a set of algebraic datatypes mirroring
the problem’s syntactic restrictions \cite{22}. A deep embedding of the datatypes in the problem’s
background theory \( T \), realized as a set of automatically generated axioms, provides a
semantics for datatype values in terms of the semantic values in \( T \).

**Example 5.** Consider the problem of synthesizing a binary function \( f \) over the integers
such that \( f \) is commutative (i.e., \( \exists f \forall xy. f(x, y) \approx f(y, x) \)), and with the solution space
for \( f \) defined by a context-free grammar \( R \) with start symbol \( A \) and production rules:

\[
A \rightarrow x \mid y \mid 0 \mid 1 \mid A + A \mid A - A \mid \text{ite}(B, A, A) \quad B \rightarrow A \geq A \mid A \approx A \mid \neg B
\]

The following mutually recursive datatypes capture the grammar \( R \). The datatypes
themselves correspond to \( R \)’s non-terminals (e.g., \( a \) corresponds to \( A \)), their constructors
correspond to production rules (e.g., \( X \) corresponds to \( A \rightarrow x \)):

\[
\begin{align*}
a &= X \mid Y \mid \text{Zero} \mid \text{One} \mid \text{Plus}(a, a) \mid \text{Minus}(a, a) \mid \text{ite}(b, a, a) \\
b &= \text{Geq}(a, a) \mid \text{Eq}(a, a) \mid \text{Neg}(b)
\end{align*}
\]

Datatypes like the one above can be associated with the programs they represent through
evaluation functions that map datatype values, expressed as ground constructor terms,
to expressions over the basic types. For example, the evaluation function \( \text{eval}_a \) for \( a \)
is a function of type \( a \times \text{Int} \times \text{Int} \rightarrow \text{Int} \), and the specific term \( \text{eval}_a(\text{Plus}(X, X), 2, 3) \) is
interpreted as \( (x + x)\{x \mapsto 2, y \mapsto 3\} = 2 + 2 = 4 \). The evaluation functions in this
case are defined by a set of axioms that can be handled by any SMT solver that, like
cvc4, supports the combined theory of datatypes, linear arithmetic, and uninterpreted
functions. The SyGuS problem for \( f \) in this example can then be stated as the first-order
formula:

\[
\forall xy. \text{eval}_a(d, x, y) \approx \text{eval}_a(d, y, x)
\]

where \( d \) is a fresh constant of type \( a \). This formula has models in which \( d \) is interpreted
as Zero or \( \text{Plus}(X, Y) \), which correspond to solutions \( f = x y. 0 \) and \( f = x y. x + y \)
for the original formula, respectively\footnote{For a thorough description of this approach, see \cite{22}.}.

Since cvc4 is a DLPT-based solver \cite{11}, for a problem like the one in the example
above, it will find a possible solution for \( d \) by first guessing its top constructor symbol
with an application of the \texttt{Split} rule from Figure\footnote{For a thorough description of this approach, see \cite{22}.}. The effect of the rule is achieved in practice with the generation of splitting lemmas such as the following, which we write here with discriminators and standard selectors for simplicity:

\[
\begin{align*}
isX(d) \vee isY(d) \vee \cdots \vee is\text{ite}(d) \quad (4) \\
isX(S^{\text{Plus}, 1}(d)) \vee isY(S^{\text{Plus}, 1}(d)) \vee \cdots \vee is\text{ite}(S^{\text{Plus}, 1}(d)) \quad (5) \\
is\text{Geq}(S^{\text{Lin}, 1}(d)) \vee is\text{Eq}(S^{\text{Lin}, 1}(d)) \vee is\text{Neg}(S^{\text{Lin}, 1}(d)) \quad (6) \\
isX(S^{\text{Lin}, 2}(d)) \vee isY(S^{\text{Lin}, 2}(d)) \vee \cdots \vee is\text{ite}(S^{\text{Lin}, 2}(d)) \quad (7)
\end{align*}
\]
The solver will subsequently guess the top constructor for other subterms of \( d \)'s value. These guesses are represented symbolically by selector chains, i.e. zero or more applications of selectors to \( d \); for example, \( S_{\text{Plus}}^1(d) \) is a selector chain that corresponds to the first child of \( d \) (if we think of the value of \( d \) as a tree) when \( d \) is an application of \( \text{Plus} \); \( S_{\text{Plus}}^1(S_{\text{Plus}}^1(d)) \) is a selector chain that corresponds to the first child of the first child of \( d \) when \( d \) and its first child are both applications of \( \text{Plus} \); and so on.

The bottleneck in solving (3) is the large number of splitting lemmas for selector chains introduced during search which, depending on the datatypes involved, is often highly exponential. Our key observation is that datatypes generated by the SyGuS approach sketched above very often include constructors with arguments of the same type. In Example 5, both \( a \) and \( b \) have multiple constructors with arguments of type \( a \). Using shared selectors, we can reduce the number of selectors in the example from 7 to 3 for \( a \) and from 5 to 3 for \( b \). Moreover, using shared selectors in selector chains makes splitting lemmas relevant in multiple contexts. For example, a splitting lemma for a selector chain \( S_{\text{Plus}}^1(d) \) is relevant when \( d \) is either \( \text{Plus} \), \( \text{Minus} \) or \( \text{Ite} \); likewise \( S_{\text{Plus}}^1(S_{\text{Plus}}^1(d)) \) is relevant when \( d \) and its first child of type \( a \) are applications of either \( \text{Plus} \), \( \text{Minus} \) or \( \text{Ite} \). Notice that by using the decision procedure for shared selectors from Section 4, lemmas (5) and (7) would instead both be provided to the SAT engine as:

\[
\text{isX}(S_{\text{Int}}^1(d)) \lor \text{isY}(S_{\text{Int}}^1(d)) \lor \cdots \lor \text{isIte}(S_{\text{Int}}^1(d))
\]

Using shared selectors can lead to reducing the number of other kinds of lemmas as well. During synthesis, cvc4 implements symmetry breaking techniques to avoid spending time on repeated candidates that are equivalent in \( T \) \([22,23]\). These redundant candidates are avoided by adding blocking clauses to the SAT engine that are also expressed in terms of discriminators applied to selector chains.

**Example 6.** Consider again the function \( f \), grammar \( R \), and datatypes \( a \) and \( b \) from Example 5. Assume that the solver considers \( X \) as a candidate solution for \( d \), and later considers another candidate solution, \( \text{Plus}(X, \text{Zero}) \). Since the corresponding arithmetic terms \( x \) and \( x + 0 \) are equivalent in integer arithmetic, the solver infers a lemma template of the form:

\[
\neg \text{isPlus}(z) \lor \neg \text{isX}(S_{\text{Int}}^1(z)) \lor \neg \text{isZero}(S_{\text{Int}}^2(z))
\]

to block a redundant candidate solution like (the one corresponding to) \( x + 0 \). This is achieved by instantiating the template with the substitution \( \{ z \mapsto d \} \) for variable \( z \). More interestingly, \( z \) can be instantiated with other selector chains to rule out entire families of redundant candidate solutions. For instance, the lemma obtained with \( \{ z \mapsto S_{\text{Int}}^1(d) \} \) rules out all terms that have \( x + 0 \) as their first child of type \( a \), such as the terms \( (x + 0) + y \), \( \text{ite}(x \geq y, x + 0, y) \) and \( (x + 0) - 1 \), which are equivalent to the smaller expressions \( x + y \), \( \text{ite}(x \geq y, x, y) \) and \( x - 1 \), respectively, and hence redundant as candidate solutions. Sharing selectors allows the same blocking clause to be reused for the different constructors, whereas standard selectors would require three different clauses in this case, with \( z \mapsto S_{\text{Plus}}^1(d), z \mapsto S_{\text{Ite}}^2(d), \) and \( z \mapsto S_{\text{Minus}}^1(d) \), respectively.
6 Experiments

We implemented our calculus for the theory of datatypes with shared selectors in cvc4 Version 1.5, together with a preprocessing pass to convert standard selectors in input formulas to shared ones and other modifications to the existing decision procedure for datatypes, as described in Sections 3.2 and 4. We discuss here our evaluation of two configurations of cvc4, with and without support for shared selectors, on two different sets of benchmarks: the SyGuS benchmark suite from the 2017 SyGuS competition [4]; and a subset of the SMT-LIB [8] benchmarks containing datatype computations. Our experiments were performed on the StarExec logic solving service [26].

6.1 Syntax-guided Synthesis Benchmarks

The benchmarks from the 2017 SyGuS competition are divided into five families across four tracks:

- The “General” track - problems over the theories of linear integer arithmetic (LIA) or bit-vectors.
- The conditional linear integer arithmetic track (CLIA) - problems over the theory of LIA.
- The “Invariant” synthesis track - invariant synthesis benchmarks specifying the problem with pre- and post-conditions, and a transition relation, over LIA.
- Programming by example track - consists of the bit-vector and strings subtracks, where problems are semantically constrained by examples.

We measured the impact of shared selectors by comparing, for the two configurations of cvc4, the total number of solved problems, and the average over the benchmarks of: solving time, number of decisions performed by the SAT engine, quantifier-free terms.

---

generated, and number of selectors in the signature. Averages were computed over the set of problems solved by both configurations. We used a timeout of 30 minutes per benchmark.

A summary of the results is given in Figure 5. The first two columns show the evaluated family and the number of benchmarks in it, while the other columns present the statistics listed above, with average times expressed in seconds. The number of problems solved by both configurations is given in parentheses in the third column. The results clearly show that sharing selectors reduces the number of selectors in the signature, which generally leads to fewer terms and SAT decisions, with a positive impact on solving speed and number of problems solved. Except for the invariant family, the cvc4 configuration with shared selectors solves more problems than the one without. The impact of shared selectors is particularly significant for the bit-vector benchmark suite (PBE_BV), with a reduction of over 80% in the average number of selectors. In that case, cvc4 is over eight times faster with shared selectors than without, solving 412 more problems, thus reducing the percentage of unsolved problems in this category from over 65% to less than 12%. Significant improvements can also be observed in the PBE_Strings and General families, with the percentages of unsolved problems being reduced from over 40% to almost 13% and from over 55% to almost 40%, respectively.

We present a per-problem comparison in the scatter plots of Figure 4, which clearly show that for the vast majority of the benchmarks, sharing selectors reduces the number of SAT decisions and improves the solving time, often by orders of magnitude.

Comparison against other SyGuS solvers We also compared cvc4’s performance with the state-of-the-art SyGuS solver EUSolver [2, 5]. For fairness, in this comparison we combine the results of the above configurations of cvc4 with its other approach for solving single-invocation synthesis problems (see [22] for details), which impacts the CLIA and General families of benchmarks. The results are summarized in Figure 5. We obtained the following results for the problems solved by EUSolver and cvc4 with and without shared selectors: 71/73/73 for CLIA, 404/391/334 for General, 42/46/46 for Invariant, 739/665/253 for PBE_BV, and 68/93/64 for PBE_Strings. These numbers show that overall cvc4 is significantly more competitive with shared selectors than without, surpassing EUSolver’s performance in three of the five families.
<table>
<thead>
<tr>
<th>Family</th>
<th>#</th>
<th>EUS</th>
<th>cvc4-si-sh</th>
<th>cvc4-si-std</th>
</tr>
</thead>
<tbody>
<tr>
<td>General</td>
<td>535</td>
<td>404</td>
<td>391</td>
<td>334</td>
</tr>
<tr>
<td>CLIA</td>
<td>73</td>
<td>71</td>
<td>73</td>
<td>73</td>
</tr>
<tr>
<td>Invariant</td>
<td>67</td>
<td>42</td>
<td>46</td>
<td>46</td>
</tr>
<tr>
<td>PBE_BV</td>
<td>750</td>
<td>739</td>
<td>665</td>
<td>253</td>
</tr>
<tr>
<td>PBE_Strings</td>
<td>108</td>
<td>68</td>
<td>93</td>
<td>64</td>
</tr>
</tbody>
</table>

Fig. 5: Performance of cvc4 and EUSolver on benchmarks from five families of SyGuS Comp 2017.

<table>
<thead>
<tr>
<th>Family</th>
<th>#</th>
<th>Solved: sh / std</th>
<th>Time</th>
<th>Decs</th>
<th>Terms</th>
<th>Sels</th>
</tr>
</thead>
<tbody>
<tr>
<td>Leon</td>
<td>410</td>
<td>179 / 175 (175)</td>
<td>0.96</td>
<td>9920 / 9925</td>
<td>718 / 929</td>
<td>8.67 / 23.10</td>
</tr>
<tr>
<td>Sledgehammer</td>
<td>321</td>
<td>113 / 112 (112)</td>
<td>0.47</td>
<td>6949 / 6942</td>
<td>185 / 185</td>
<td>10.50 / 12.76</td>
</tr>
<tr>
<td>Nunchaku</td>
<td>158</td>
<td>67 / 67 (67)</td>
<td>0.49</td>
<td>7149 / 6653</td>
<td>1373 / 1297</td>
<td>6.22 / 7.22</td>
</tr>
</tbody>
</table>

Fig. 6: Performance of cvc4 on benchmarks from three families of SMT LIB.

### 6.2 Datatype benchmarks from SMT LIB

We also considered all SMT-LIB benchmarks containing datatypes. Among these, we excluded from consideration 14,387 benchmarks that do not have any shareable selectors, as cvc4 with and without shared selectors perform the same on these benchmarks. The remaining 889 benchmarks are divided into three families:

- A set of benchmarks generated by Leon [12] (and Nunchaku and cvc4) for counterexample generation for higher-order theorem provers (AUFBVDTLIA logic). We will refer to this set as ‘Leon’.
- Benchmarks generated for verification in Isabelle [19] by the Sledgehammer tool [14] (UFDT logic). This set will be referred to as ‘Sledgehammer’.
- A benchmark set generated for higher order theorem provers by Nunchaku [21] (and Leon, and cvc4) (UFDT logic). This is the ‘Nunchaku’ set.

We summarize our results over the two configurations of cvc4, with and without shared selectors, in Figure 6 following the same schema as in Figure 5. We used a timeout of 60 seconds, since in this setting we evaluate SMT solvers as backends of verification and ITP tools, which require fast answers. The configuration with shared selectors solved at least all the benchmarks as the one without. The Leon benchmark set shows the most significant impact of sharing selectors, with a reduction of over 60% in the average number of selections, and 4 more problems solved. It is important to remark that cvc4 employs heuristic instantiation techniques for solving these benchmarks, which can be very sensible for changes in the signature. The relevant number to measure then is the number of solved benchmarks, which either remains the same or improves with shared selectors, even if the average solving time is marginally worse.

Comparison against other SMT solvers To put the shared selector version of cvc4 in context with the state of the art, we also compared it with the only two provers that can
reason about datatypes and support the SMT-LIB format: z3 [16] and Vampire [17]. On the Nunchaku and Sledgehammer benchmarks, the comparison of problems solved between cvc4, z3 and Vampire is 67/29/30 and 113/119/138, respectively. The comparison on the Leon set excludes Vampire, since it does not support the theory of bit-vectors; the split between cvc4 and z3 is 179/173 on that set. The results show that cvc4 compares favorably with the other tools.

7 Related Work

The motivation of our work is to reduce the number of terms considered by a decision procedure for the theory of algebraic of datatypes, based on procedures introduced in previous work [10] [20]. Thus, our contributions apply to other systems that handle datatypes semantically, such as SMBC [15] and the SMT solver Z3 [16]. On systems that reason about datatypes axiomatically, such as the first-order theorem prover and SMT solver Vampire [17], and the higher-order systems Isabelle [19] and Dafny [18] whether to share selectors and how to handle them is simply a matter of axiomatizing the datatypes theory accordingly. For example, the axiomatization in Vampire avoids selectors altogether [17, Sect. 4.3], while in Isabelle users are encouraged to write specifications directly with shared selectors [13, Sec. 3].

Most SyGuS solvers employ a variation of counter-example guided inductive synthesis (CEGIS), introduced by Solar-Lezama [25]. While cvc4 benefits from sharing selectors by representing syntax restrictions with datatypes, other systems use an outer layer with an underlying reasoning engine, for instance using an SMT solver to verify the correctness of candidate solutions, but not for performing the enumerative search [5].

8 Conclusion

We have presented an extension of the theory of algebraic datatypes that adds shared selectors. We have discussed and proved correct a calculus for deciding the constraint satisfiability problem in the new theory. Moreover, we have described how algebraic datatypes can be leveraged in an SMT solver to solve syntax-guided synthesis problems and explained how the use of shared selectors in this setting may lead to significant performance gains. Our experiments demonstrate that an implementation of the new calculus in the cvc4 solver significantly enhances its performance on syntax-guided synthesis problems and is responsible for making cvc4 the best known solver for certain classes of problems.

In future work, we plan to generalize our approach so that distinct selector chains can be compressed to a single application of the same selector symbol. This requires more sophisticated criteria for recognizing when two selector chains for a datatype cannot be simultaneously constrained for arbitrary values of that datatype. We believe that this further extension can be done in a manner similar to the one presented here and expect that this will lead to further performance improvements.
References

