# Interpolation for Data Structures [*]

Deepak Kapur
University of New Mexico
kapur@cs.unm.edu

Rupak Majumdar
UC Los Angeles
rupak@cs.ucla.edu

Calogero G. Zarba
Universität des Saarlandes
zarba@alan.cs.uni-sb.de

## Abstract

Interpolation based automatic abstraction is a powerful and robust technique for the automated analysis of hardware and software systems. Its use has however been limited to control-dominated applications because of lack of algorithms for computing interpolants for data structures used in software programs. This paper presents a general algorithm to construct interpolants for any recursively enumerable theory. In particular, efficient procedures to construct interpolants for the theories of arrays, sets, and multisets are discussed using the reduction approach for obtaining decision procedures for complex data structures. The approach taken is that of reducing the theories of such data structures to the theories of equality and linear arithmetic for which efficient interpolating decision procedures exist. This enables interpolation based techniques to be applied to programs that manipulate these data structures. These interpolating decision procedure are applied to the verification of C programs manipulating data structures using the software model checker Blast. Using the new procedures, Blast can check safety properties of C programs that manipulate data structures.

## 1. Introduction

Abstraction based model checking and automatic refinement of abstractions has received a lot of attention as a precise but scalable technique for verifying system properties [28, 3, 18, **?**, 20]. The key insight is to start with a crude abstraction on system states considered strong enough to prove a safety property of the system. If in an attempt to do a proof, a counter-example is discovered, it is checked whether the counter-example is indeed realized in the system, or it might have be spurious because of the abstraction being too crude. In the later case, the abstraction is refined using the notion of an *interpolant* of theories. This approach has come to be known as counterexample-guided abstraction refinement (CEGRA), where interpolants have been used for predicate discovery [17]: given an infeasible trace produced by the abstract model checker, an interpolant at a point in the trace determines an over-approximation of the set of reachable states that can execute the prefix of the trace in terms of the live variables that is enough to determine the infeasibility of executing the suffix.

Interpolants provide a robust abstraction technique in model checking. In hardware verification, interpolants have also been used [28] as a substitute for the expensive post-image computation to construct invariants: one finds an interpolant between the set of states reachable in $k$-steps (constructed using bounded model checking) and the set of error states. The interpolant is tried as an invariant. By providing a property-guided abstraction, the interpolant precludes the need to compute the strongest inductive invariant for the system. In infinite-state verification, a similar algorithm is used, but the logic is no longer propositional, but first order, usually with the theories of equality and arithmetic.

Most applications of interpolants have so far been restricted to propositional logic, and the theories of equality with uninterpreted functions together with linear arithmetic [28, 27, 17, 20, 21]. This severely restricts the kind of programs and properties that have been studied in the software model checking literature, which have so far been restricted to niche control-dominated applications such as device drivers [3, 18] and low level state machine properties such as correct usage of locks [12, 18]. As shown in this paper, providing interpolating theorem provers for data structure theories will significantly extend the applicability of software verification by providing robust approximation tools so far available only for equality and arithmetic.

One such application is that of using software model checkers to programs that use data structures and properties that depend on correct use of data structures. Until recently, the model checking of imperative programs manipulating data structures have mainly been limited to fundamental correctness properties of the *implementation* of data structures [36, 2, 29]: for example, to check that a list reverse routine does reverse an acyclic list. While an important area of research, this captures only half the problem. The other half, automatically checking properties of applications that *use* these data structures, *given* a correct implementation of the data structure has received less attention. In practice though, programmers use well-tested (and reasonably bug-free) library implementations of common data structures such as lists, sets, or bags (e.g., from C++ STL or Java system classes), and many bugs can be unearthed in the use of these data structures in client programs.

Using such module-level abstractions is a fundamental software engineering principle to handle complexity [34], and we decompose our correctness proofs accordingly at module boundaries. This is not a new idea: Hoare [19] suggests a modular proof decomposition for data structures into checking the implementation w.r.t. an abstract specification, and checking separately the use of the implementation assuming the abstract specification. It is shown in this paper how modular-verification and counterexample-guided abstraction refinement can be combined using interpolants for theories of data structures. Our contribution is to provide powerful abstraction capabilities provided by interpolants in order to make much of this reasoning automatic.

[*] This research was funded in part by the NSF grants CCF-0427202 and CNS-0541606.

We also present results about two novel aspects of interpolation in first order theories in this paper. The first is foundational: we study under what conditions we can guarantee and effectively compute a (quantifier-free) interpolant. We give a simple algorithm to compute (not necessarily quantifier-free) interpolants from the proof of unsatisfiability of any recursively enumerable theory and show that quantifier elimination in the theory is a necessary and sufficient condition for the existence of quantifier-free interpolants. Using this simple characterization, we provide interpolating decision procedures for the theories of (real and Presburger) arithmetic, arrays, lists, sets, and bags. Additionally, we show the existence of *quantifier free* interpolants for the theories of linear arithmetic, lists, and sets with cardinality constraints, and show that quantifier free interpolants do not exist for theories of arrays and bags.

The second aspect is pragmatic: we provide a *reduction* technique to find interpolants in many theories of practical interest that can use existing interpolating theorem provers as black boxes. In particular, we show a compilation of different theories to the combination theory of equality with uninterpreted functions and the theory of linear arithmetic (and free constructors) such that from an interpolant in the latter theory, one can construct an interpolant in the original theory.

This is particularly attractive since very efficient implementations of these combined theories exist [8, 37] and are already interpolant producing [27]. In our experience, developing efficient implementation and integration of new theories into decision procedures (including carefully tuned heuristics) is an expensive effort. Our compilation algorithm sidesteps this by providing an easy access to already developed tools through a simple and syntactic compilation step. Thus, our techniques provide the first practical interpolating decision procedures for the theories of arrays, sets, and bags.

We have implemented interpolating decision procedures for the quantifier-free theories of arrays, lists, sets with cardinality constraints, and bags on top of the Foci interpolating decision procedure for linear arithmetic with uninterpreted functions [27]. This was used in the software model checker Blast [18]. By using more general reasoning about data structures, Blast was able to prove interesting properties of programs that manipulate data structures.

The rest of the paper is organized as follows. In Section 2, we provide an overview of CEGAR based approach for verifying safety properties. The role of interpolants for refining abstractions guided by counter-examples is reviewed. In Section **??**, we prove prove the existence of interpolants for any recursively enumerable theory. It is also shown that quantifier-elimination in a theory is a necessary and sufficient condition for the existence of quantifier-free interpolants. In Subection 3.4, we formalize the notion of reductions of a (more complex) theory to another simpler theory. These reductions are applied in Section 4 to obtain interpolants for several theories of parctical interest. A brief review of a preliminary implementation of these reductions in Foci is given.

**Other Related Work.** Our work follows recent work on interpolation as a powerful abstraction technique [28, 27, 17, 20, 21, 43]. We build up on reduction based decision procedures of [22].

There has been a lot of work on modular construction and verification of software since the early papers [34] and [19], and the fundamental studies in abstract data types [25, 16]. Recent attempts for automatic modular verification of software using verification condition generation, explicit pre- and post-conditions, and decision procedures include [11, 24]. Our work is similar in spirit to these efforts. It uses for example, quantifier elimination procedures developed as part of the Hob infrastructure. However, instead of building the most precise verification condition up front, we use counterexample-guided refinement to incrementally build up invariants to the required precision [6, 3, 18, **?**]. The paper [4] performs modular verification, but does not handle data structures.

```
Example() {
x := emptyset();
while (*) {
     t := *;
     if(t->tag==0)
         x := add(x,t);
}
q := choose(x);
if (q≠0 && q->tag != 0)
     error();
}
```

```
type element
type set
emptyset : void → set
add : set × element → set
choose : set → element
```

**Figure 1.** (A) Simple client using sets (B) set signature

Orthogonal work in separation logic also attempts system verification in the presence of heap data structures [31, 33], however, the notion of automatic abstractions has so far been less central. Work on shape analysis and related techniques [36, 2, 29] complements our work by proving the implementation of data structures correct w.r.t. abstract specifications.

## 2. Abstract Data Types and Model Checking

The motivation for our work is the automatic verification of programs that manipulate data structures through a set of interface functions. We now demonstrate how interpolants for data structures enable automatic predicate discovery in software model checking based on counterexample-guided abstraction refinement (CEGAR) through a simple program that manipulates sets.

### 2.1 Programs and Data Types

We describe our technique on programs in a simple imperative programming language with typed variables and with the following basic operations: (1) assignments `x := e`, that set the value of the expression `e` to the variable `x`, (2) assume predicates $\texttt{assume}[\texttt{p}]$, that represent a boolean condition `p` that must be true for the operation to be executed, and (3) function calls $\texttt{y} := f(\overline{\texttt{x}})$ that call a function $f$ with the actual arguments $\overline{\texttt{x}}$ and writes the return value into `y`. We assume that control flow is represented explicitly, e.g., through a control flow graph, and return values are passed back using a special `ret` variable.

A *library* $\mathsf{Lib} = (\Sigma, \mathcal{C})$ consists of (1) a set $\Sigma$ of typed functions that represents the externally callable function names, and (2) a map $\mathcal{C}$ from functions $f \in \Sigma$ to their implementations. We write $\Sigma_{\mathsf{Lib}}$ for the signature of a library $\mathsf{Lib}$. We assume that for a function $f$ of type $\sigma_1 \times \ldots \times \sigma_n \to \sigma$, the implementation has $n$ input variables of the appropriate types, and the return value is of type $\sigma$. A *client* for a library $\mathsf{Lib}$ is a program that calls only the functions in the set $\Sigma_{\mathsf{Lib}}$. We assume all function calls in the client are type correct in that the types of called functions are respected. A *closed* program $(\mathsf{Lib}, \mathsf{Client})$ consists of a library $\mathsf{Lib}$ and a client $\mathsf{Client}$ of the library $\mathsf{Lib}$. While we have assumed that the client only interacts with one library and only calls functions in the library $\mathsf{Lib}$, these are for ease of exposition, and our techniques work even when the client has other function calls, or when the client uses several libraries.

**Example 1.** Instead of giving a formal definition, we introduce clients and libraries through an example. Figure 1(A) shows a client program `Example` that uses a library implementing a set data structure whose signature is shown in Figure 1(B). The program is motivated from the scheduler code in an OS kernel, where the set corresponds to tasks that are runnable: tasks are added to the run queue, and later, when they are removed, the kernel checks that any task from the run queue is runnable.

The client starts with an empty set (line 01), and in a while loop, adds elements to the set provided the tag field of the element is 0 (lines 02 to 06). Then it chooses an element `q` from the set (line 07), and checks that the both `q` is non-null and the tag field of `q` is not 0 (line 08). If the check succeeds, it means that an element whose tag is not zero has been returned from the set, and an error occurs (line 09). The library `set` provides the interface functions `emptyset`() to produce an empty set, `add`() to add an element to a set, and a function `choose`() that returns an arbitrary element from a set if the set is not empty and 0 otherwise. We omit the implementation of these interface functions. The client `Example`, together with the set library, forms a closed program. □

**Safety Verification Problem.** Let $V$ be the set of variables of a program. A *data state* is a type-preserving mapping of variables in $V$ to values in their domain. A *state* $(\ell, s)$ of the program consists of a program location $\ell$ and a data state $s$. A *region* is a set of states. We shall use first-order formulas over the program location and program variables to represent regions. The operations of the program define a binary *transition relation* on states which specifies the new state of the program that results when an operation `op` is performed from the current state. The transition relation is lifted to regions in the natural way.

Let $(\mathsf{Lib}, \mathsf{Client})$ be a closed program. A state $(\ell, s)$ of the program is *reachable* if there is some sequence of program operations (allowed by the control flow of the program) that takes the program from some initial state to $(\ell, s)$. A program location $\ell$ is reachable if some state $(\ell, s)$ is reachable. For a closed program $(\mathsf{Lib}, \mathsf{Client})$ and a location $\ell$ of $\mathsf{Client}$, the *safety verification problem* asks if $\ell$ is reachable in the program $(\mathsf{Lib}, \mathsf{Client})$. We say $(\mathsf{Lib}, \mathsf{Client})$ *satisfies the safety property* $\ell$, written $\mathsf{Lib}\|\mathsf{Client} \models \ell$, if $\ell$ is not reachable in $(\mathsf{Lib}, \mathsf{Client})$. It is known that any safety property can be reduced to checking (un)reachability of some location $\ell$.

**Example 2.** In the example of Figure 1, we want to check that the condition $q \rightarrow tag \neq 0$ at line 08 never holds, so that line 09 is unreachable. Informally, the program satisfies this safety property, since the set `x` starts of empty, and any element `y` in the set `x` added in the while loop in lines 02 to 06 satisfies `y` $\rightarrow tag = 0$, so that an arbitrary element $q$ chosen from the set satisfies $q \rightarrow tag = 0$. □

For a closed program $(\mathsf{Lib}, \mathsf{Client})$ and a program location $\ell$, one way to solve the safety verification problem is to compute an over-approximation of the set of all reachable states of the program and check if some state $(\ell, s)$ is in this set. If not, then $\ell$ is not reachable. However, if $\ell$ is reachable in this over-approximation, it may or may not be reachable in the original program. In this case, counterexample-guided abstraction refinement techniques [6, 3, 18] automatically find either (a) a concrete program execution to $\ell$ or (b) a new and more precise over-approximation of the set of reachable states and repeats until either the location $\ell$ is proved to be unreachable, or a concrete counterexample trace to $\ell$ is obtained. However, there are two pragmatic issues that arise in safety verification problem.

First, when both the client and the library are large programs, the construction of the reachable set is expensive and most techniques do not scale. Second, most automatic and scalable program analysis tools do not precisely reason about complex data and pointer manipulation. Hence, if the actual implementation of the library involves manipulation of heap data structures, these tools result in false alarms. Indeed, we were unable to verify the example in Figure 1 using the software model checker Blast [18] when we analyzed the client together with the implementation of the set library. This was because Blast was unable to reason precisely about the pointer manipulations in the set implementation. Therefore, we turn to *modular verification*.

## 2.2 Modular Verification

Instead of checking the full implementation of $(\mathsf{Lib}, \mathsf{Client})$, we decompose the proof obligation in the following way. First, we construct an abstraction $A$ of $\mathsf{Lib}$ (i.e., a program with at least as many behaviors as $\mathsf{Lib}$), and separately check that (1) the closed program $(A, \mathsf{Client})$ satisfies the safety property $\ell$ and (2) $A$ is indeed an abstraction of $\mathsf{Lib}$. We use *abstract datatype definitions* (ADTs) as abstractions of a library $\mathsf{Lib}$.

An ADT $A = (T, \mu)$ for a library $\mathsf{Lib}$ with signature $\Sigma_{\mathsf{Lib}}$, written $\mathsf{Lib} \preceq A$, consists of a first order theory $T$ (i.e., a set of first order logic sentences closed under deductions) whose signature contains $\Sigma_{\mathsf{Lib}}$ as well as a map $\mu$ associating with each function $f \in \Sigma_{\mathsf{Lib}}$ of type $\sigma_t \times \ldots \sigma_n \rightarrow \sigma$ a formula $\mu(f)(x_1, \ldots, x_n, y')$ with free variables $x_1$, …, $x_n$, and $y'$ of sorts $\sigma_1$, …, $\sigma_n$, $\sigma$ respectively.

The formula $\mu(f)$ is intended to replace the actual implementation of the function $f$ in the library with a declarative specification of its transition relation. That is, for every values $c_i \in \sigma_i$ (for $i = 1, \ldots, n$) and $d \in \sigma$, we have $d = f(c_1, \ldots, c_n)$ iff $\mu(f)(c_1, \ldots, c_n, d)$, and if the theory $T$ entails $\mu(f)(x_1, \ldots, x_n, y) \Rightarrow \psi(x_1, \ldots, x_n, y)$, then $\psi(c_1, \ldots, c_n, d)$ as well. Formally, we use the modular proof rule [19]

$$\frac{\mathsf{Client}\|\mathsf{A} \models \ell \qquad \mathsf{Lib} \preceq \mathsf{A}}{\mathsf{Client}\|\mathsf{Lib} \models \ell}\ \mathsf{ADT} \tag{1}$$

The rule breaks the verification effort into two parts: first, the implementation of the library is verified in isolation against an abstract logical specification, and second, the client code is verified using the abstract logical specification of the library. In this paper, we focus on the verification problem

$$\mathsf{C}\|\mathsf{A} \models \ell \tag{2}$$

There are other, orthogonal, approaches to prove the second obligation $\mathsf{Lib} \preceq \mathsf{A}$ [36, 11].

**Example 3.** An ADT for the set library in Figure 1(B) consists of the theory of sets together with the following mapping from functions in the library to formulas in first order logic over the signature of sets:

| | |
|---|---|
| $\mathtt{y} := \mathtt{emptyset}()$ | $\mathtt{y} = \emptyset$ |
| $\mathtt{y} := \mathtt{add}(\mathtt{x}, \mathtt{t})$ | $\mathtt{y} = \mathtt{x} \cup \{\mathtt{t}\}$ |
| $\mathtt{y} := \mathtt{choose}(\mathtt{x})$ | $(\mathtt{y} = 0 \wedge \mathtt{x} = \emptyset) \vee (\mathtt{y} \neq 0 \wedge \mathtt{y} \in \mathtt{x})$ |

These formulas declaratively specify the intent of each function in the interface. In our application of modular verification, we shall verify that the assertions in the client are satisfied assuming the library conforms to this ADT. The closed program $(\mathsf{Lib}, \mathsf{Client})$ is correct if in addition, we prove $\mathsf{Lib}$ indeed conforms to this ADT.[1] □

## 2.3 CEGAR

We now show how a counterexample-guided abstraction refinement algorithm using interpolant based predicate discovery [17] can solve the safety verification problem. We (1) briefly describe the main steps of the CEGAR loop, illustrating the steps on the `Example` code, and (2) point out the role of interpolation and reduction in the different phases.

Algorithm 1 shows the overall algorithm to check whether a set of states is reachable. The algorithm takes as input a client program $\mathsf{Client}$ using a library $\mathsf{Lib}$, an ADT $A$ such that $\mathsf{Lib} \preceq A$, a set of

[1] One issue is that in a language like C, even when an object is placed in a list, the programmer can still update the state of the object outside the list (e.g., through a pointer to the object). This generates a third proof obligation that we ignore for simplicity of exposition. This proof obligation can be discharged using an alias analysis that rules out updates to memory locations that are stored in a set.

**Algorithm 1** Safety Verification

```
Input: client program Client using ADT $A$
Input: initial abstraction $\Pi_0$, program label $L$
Output: "reachable" if label $L$ is reachable,
Output: "safe" otherwise
 1: $\Pi := \Pi_0$
 2: Step 1: $\mathcal{G} := \mathsf{Reach}(\mathsf{Client}, A, \Pi)$
 3: Step 2:
 4: if $L$ is unreachable in $\mathcal{G}$ then
 5:     return "safe"
 6: else
 7:     pick an abstract trace $t$ from $\mathcal{G}$ that reaches $L$
 8:     if $t$ can be concretely simulated then
 9:         return "reachable"
10:     else
11:         Step 3: $\Pi := \Pi \cup \mathsf{Refine}(t)$; goto Step 1:
12:     end if
13: end if
```

predicates $\Pi_0$ over the program state, and a location $\ell$ of the program. It returns "reachable" if some execution reaches the location $\ell$, and "safe" otherwise. At all points, the algorithm maintains a *current abstraction*, which is a set of first order predicates over the program state. The algorithm has three main steps.

The first (**Step 1**) is a forward search phase that constructs an over-approximation of the reachable states using the current abstraction. This phase constructs a tree representing an unfolding of the control flow automaton. Each edge of the tree is labeled with a program operation, and each node is labeled with a program location $l$ as well as a formula $\varphi$ over the predicates in the current abstraction. The formula $\varphi$ represents a superset of the set of states that can reach the program location $l$ by executing the program operations along the path from the root of the tree to the node. Since we only restrict attention to the predicates in the current abstraction, the tree represents, in general, an over-approximation of the actual reachable states.

The second (**Step 2**) checks if the location $\ell$ is reachable in the forward search tree. If not, the algorithm returns "safe". This is sound since the forward search tree is an over-approximation of the set of reachable states. However, if $\ell$ is reachable in the tree, the path to $\ell$ may (a) either represent a real bug, (b) or be a *spurious counterexample* in that $\ell$ is abstractly reachable because we have lost too much information by restricting to the current abstraction. This step performs a symbolic execution over a (possibly spurious) path to $\ell$ in the tree. If the symbolic constraints generated are satisfiable, then the path represents a real bug and the algorithm returns "reachable." Otherwise, we proceed to Step 3.

In case Step 2 finds the current path to be spurious (i.e., obtained because the current abstraction is too coarse), a refinement procedure (**Step 3**) is used to *refines* the current abstraction by adding new predicates derived from analyzing $t$.

The following is standard [6, 3, 18].

**Theorem 4.** *If Algorithm 1 returns "safe" for a client program* Client*, an ADT $A$, a set of initial predicates $\Pi_0$, and a location $\ell$, then* $\mathsf{Client}\|A \models \ell$*. If it returns "reachable" then $\ell$ is reachable in* $\mathsf{Client}\|A$. □

We now describe each part of the algorithm in more detail.

### 2.4 Forward Search and Reduction

We now describe the first phase: forward search. We start with some preliminary definitions.

**Abstract Postconditions.** The basic step in the forward reach is the *abstract post* computation, that takes a formula $\varphi$ over the current abstraction and a program operation op, and produces a new formula that represents a superset of the set of states that can be reached from the states in $\varphi$ by executing op.

Let Client be a client program, and let Lib be a library that conforms to the ADT $A = (T, \mu)$. Let $V$ be the set of variables in a program and let $V'$ be the set of variables where each variable in $V$ is primed (i.e., variable x is renamed x$'$). Intuitively, $s$ denotes the valuation at the "current state" and $s'$ denotes the valuation at the "next state." For any operation op, we define the transition relation $T(\mathtt{op}, V, V')$ as follows:

$$\begin{array}{lcl} T(\mathtt{x} := e, V, V') & = & \bigwedge_{y':y'\neq \mathtt{x}'} y' = y \wedge \mathtt{x}' = e \\ T(\mathtt{assume}(p), V, V') & = & p \wedge \bigwedge_{y' \in V'} y' = y \\ T(\mathtt{x} := f(\bar{z}), V, V') & = & \bigwedge_{y':y'\neq \mathtt{x}'} y' = y \wedge \mu(f)(\bar{z}, \mathtt{x}') \end{array}$$

The transition relation $T(\mathtt{op}, V, V')$ relates the values of the current (unprimed) variables with the next (primed) variables when the operation op is executed. Notice that we do not expand the function calls to the library, but instead we translate the effect of the function to its logical specification given by the ADT.

Let $\Pi$ be a set of predicates over the program variables $V$. We write $\Pi'$ to denote the set where each predicate in $\Pi$ is primed. For any op, the *predicate abstraction* of the transition relation $T(\mathtt{op}, V, V')$, written $T_\Pi(\mathtt{op}, V, V')$, is the smallest predicate (in the inclusion order) over the predicates $\Pi \cup \Pi'$ that contains $T(\mathtt{op}, V, V')$. The computation of $T_\Pi$ makes calls to a decision procedure for the theory over which the predicates in $\Pi$ are interpreted [14].

**Example 5.** Let $\Pi = \{q \in x\}$. Then $T_\Pi(x = \emptyset, V, V')$ is the abstract transition relation $(q \in x)'$ which states that after the operation, $q \in x$ becomes true. Similarly, $T_\Pi(y := \mathtt{choose}(x), V, V')$ is the relation $(q \in x) \Rightarrow (q \in x)'$ that says $q \in x$ is true afterward only if it was true before. □

**Reduction.** Abstract postcondition computation involves checking satisfiability of formulas constructed syntactically from the current set of predicates and the program operation. This requires a *decision procedure* for the theory over which the formula is interpreted. For the theory of equality with uninterpreted functions and the theory of arithmetic, there are fast implementations [8, 37] that are used by software model checkers such as SLAM, Blast, or Magic to discharge the satisfiability checks.

When checking programs using an ADT $(T, \mu)$, one has to additionally implement a decision procedure for the theory $T$. Engineering a fast and scalable decision procedure is a difficult task. Instead, we use a technique called *reduction*, which compiles a query made in the theory $T$ to an equi-satisfiable query made in the theory of equality or arithmetic, for which we already have fast decision procedures. This enables us to re-use existing efficient implementations by writing the (much simpler) compilation code for the new theory. For example, for the query $q \in x \wedge x = \emptyset \Rightarrow q \notin x$ made during predicate abstraction, we reduce the query to

$$q \in x \wedge (\forall e. e \notin x) \Rightarrow q \notin x$$

which is unsatisfiable in the theory of equality with uninterpreted functions and so by the correctness of the reduction, the original formula is unsatisfiable in the theory of sets. In Section 4, we provide compilation algorithms for the theory of arrays, sets, and multi-sets, which are commonly used data structures.

The predicates used in predicate abstraction need not be quantifier-free. However, for many theories, the quantifier-free fragment has a decision procedure to answer the satisfiability queries while the full theory may be undecidable. Thus, quantifier-free predicates are of particular interest. Unfortunately, as we show later, our predicate discovery technique may produce predicates with quantifiers. In practice, theorem provers like Simplify [8] have excellent heuristics to instantiate quantifiers, and despite their incompleteness, one can still perform predicate abstraction soundly, losing information where the theorem prover is unable to decide a particular query.

**Abstract Reachability.** Given the abstract post computation, the forward search algorithm starts with the initial location of the program and the formula $true$ and computes the forward search tree by expanding each outgoing operation of every reached location by applying the abstract transition relation for the operation [18]. The algorithm uses a worklist to maintain the set of reached nodes (labeled with a location and a region) that have yet to be explored, and in each step, picks a node from the worklist and expands it by applying the abstract transition relation for every outgoing operation from the location in the CFA. If the abstract successor is not already in the tree, it is added to the worklist to be processed later. Formally, we compute the least fixpoint of the abstract transition relation starting with the initial location. The approximation of the reachable state space consists of the set of node labelings of the forward search tree. The structure of the tree is used to construct counterexample traces. For example, the approximation of the reachable states for the predicates $\Pi = \{x = \emptyset, q = 0\}$ is given by the set

$$\begin{array}{l}
\{\langle 01, \mathit{true}\rangle,\\
\langle 02, x = \emptyset\rangle, \langle 02, x \neq \emptyset\rangle,\\
\langle 03, x = \emptyset\rangle, \langle 03, x \neq \emptyset\rangle,\\
\langle 04, x = \emptyset\rangle, \langle 04, x \neq \emptyset\rangle,\\
\langle 05, x = \emptyset\rangle, \langle 05, x \neq \emptyset\rangle,\\
\langle 06, x = \emptyset\rangle, \langle 06, x \neq \emptyset\rangle,\\
\langle 07, x = \emptyset\rangle, \langle 07, x \neq \emptyset\rangle,\\
\langle 08, x = \emptyset \wedge q = 0\rangle, \langle 08, x \neq \emptyset \wedge q \neq 0\rangle,\\
\langle 09, x \neq \emptyset \wedge q \neq 0\rangle\}
\end{array}$$

Since there is a reachable state will label 09, the location 09 is abstractly reachable in this tree. This is expected, since we are not tracking the state $q \rightarrow tag$ of the elements in the set.

## 2.5 Refinement and Interpolation

If the forward search tree contains a path $t$ from the root node to a node with label $\ell$, the refinement phase performs a symbolic execution to determine whether (a) $t$ is feasible in the concrete system (and hence a bug), or (b) $t$ is spurious, and in this case, refine the current abstraction to rule out this trace.

The refinement procedure constructs a *trace formula* from the trace [17]. The trace formula is a conjunction of constraints, one per instruction in the trace. Each constraint is the application of the transition relation to the current operation, where we give new names to each variable on each assignment. The original trace is feasible iff the trace formula is satisfiable [17]. We use reduction followed by a decision procedure call to check if the trace formula is satisfiable. If so, the current trace is a valid counterexample. If not, we go to the refinement step.

Given an unsatisfiable trace formula, the refinement procedure finds out predicates at each point of the trace such that if the abstract transition relation tracks these predicates at these points, the current spurious trace is ruled out. To see the connection to interpolants, let $\varphi_1 \wedge \ldots \varphi_i \ldots \wedge \varphi_n$ be an infeasible trace formula, and consider finding predicates for the point $i$. Partition the trace formula into $\varphi^- = \varphi_1 \wedge \ldots \varphi_i$ and $\varphi^+ = \varphi_{i+1} \wedge \ldots \wedge \varphi_n$ into the portion of the trace before (respectively after) the point $i$. Now, an interpolant $\psi_i$ between $\varphi^-$ and $\varphi^+$ has the property that (1) $\varphi^-$ implies $\psi_i$, that is, the predicate $\psi_i$ holds after the prefix of the trace up to $i$ is executed, (2) $\psi_i \wedge \varphi^+$ is unsatisfiable, that is, the predicate $\psi_i$ at location $i$ is enough to show infeasibility of the suffix of the trace, and (3) $\psi_i$ is over the common variables between $\varphi^-$ and $\varphi+$, that is, over the *live* variables. If at each point $i$, we then track the predicate $\psi_i$ (where we rename variables back to their original names),

| | | |
|---|---|---|
| `x :=` `emptyset()` | $x_0 = \emptyset$ | $x = \emptyset$ |
| `assume`$(\mathit{true})$ | $\mathit{true}$ | $x = \emptyset$ |
| $q :=$ `choose(x)` | $(q_0 = 0 \wedge x_0 = \emptyset)$ | |
| | $\vee(q_0 \neq 0 \wedge q_0 \in x_0)$ | $q = 0$ |
| `assume`$(q! = 0\&\&q \rightarrow tag! = 0)$ | $q_0 \neq 0 \wedge q_0 \rightarrow tag \neq 0$ | $\mathit{false}$ |

**Figure 2.** Trace, trace formula, and interpolants

then we have enough information to rule out the trace.[2] In this way, refinement reduces to interpolant computation. So far, there are implementations of interpolating decision procedures for the theory of equality with uninterpreted functions and arithmetic [27]. How can we use these to construct interpolants for our formulas, which also talk about the theory of sets (and other data structures)? Again, we use the reduction technique. Our interpolation construction first reduces the formula to that of equality with uninterpreted functions and arithmetic, uses existing interpolating decision procedures to construct an interpolant for the reduced formula, and then maps this interpolant back to the original theories (see Figure 4 for an outline). In the process, we may introduce quantifiers. If the theory admits quantifier elimination, we can get rid of these quantifiers and get quantifier-free interpolants. More often, though, the interpolants will have quantifiers.

**Example 6.** Suppose in Figure 1, we started with no predicates. In that case, the forward search returns the trace in Figure 2 reaches line 09. This corresponds to the program execution where the `while` block is not executed, and the `then` branch is taken. The middle column shows the constraints in the trace formula. Our reduction algorithm replaces each occurrence of $x_0 = \emptyset$ in the trace formula with $\forall e.e \notin x_0$. Our interpolation procedure now constructs the interpolants shown (after simplification) in the right column at each program point. Notice, for example, that the first two constraints in the trace imply $x_0 = \emptyset$, and this predicate is enough to show unsatisfiability of the rest of the trace. Finally, the only variable $x_0$ is live at this point (it is used in the future). These predicates are enough to show that this trace is infeasible. $\square$

## 2.6 Putting It All Together

We now illustrate the whole working of the algorithm on the Example from Figure 1. We start with an empty initial set of predicates. The forward search returns the trace shown in Figure 2, and the refinement process adds the predicates $x = \emptyset$ and $q = 0$.

We add this predicate, and perform the forward search again. This time, the previous counterexample is ruled out, since after executing the operations $x =$ `emptyset()` and $true$, we have that $x = \emptyset \wedge q = 0$, and the branch is not taken. However, there is a second counterexample, shown in Figure 3. The middle column again shows the reduced trace formula, and the third column shows the interpolants at each program point. When these predicates are added to the current abstraction, the forward search can prove that the location 09 is not reachable.

In practice, there are several optimizations to the basic algorithm above. First, the search for errors is performed interleaved with the forward search. Second, refinements are local, and the forward search only re-constructs parts of the state space that is affected by the reduction. Third, instead of computing the most precise but expensive predicate abstraction, an imprecise *Cartesian abstraction* is constructed [3, 18], and this is strengthened iteratively using the interpolants derived from the counterexample analysis [20] (in fact, as noted in [20], Cartesian abstraction is too weak for programs that manipulate data structures).

[2] One technical point: the interpolants must be generated from the same proof of unsatisfi ability, see [17] for details.

| | | |
|---|---|---|
| $x := \texttt{emptyset}();$ | $\forall e.e \notin x_0$ | $x = \emptyset$ |
| $true;$ | $true$ | $x = \emptyset$ |
| $t := *;$ | $t_0 = *$ | $x = \emptyset$ |
| $\texttt{assume}(t \rightarrow tag = 0);$ | $t_0 \rightarrow tag = 0$ | $x = \emptyset \wedge t \rightarrow tag = 0$ |
| $x := \texttt{add}(x, t);$ | $\forall e.e \in x_1 \Leftrightarrow (e \in x_0 \vee e = t)$ | $x \neq \emptyset \wedge \forall e.(e \in x \Rightarrow e \rightarrow tag = 0)$ |
| $q := \texttt{choose}(x);$ | $(q_0 = 0 \wedge \forall e.e \notin x_1) \vee (q_0 \neq 0 \wedge q_0 \in x_1)$ | $q \neq 0 \wedge q \rightarrow tag \neq 0$ |
| $\texttt{assume}(q! = 0\&\&q \rightarrow tag \neq 0)$ | $q_0 \neq 0 \wedge q_0 \rightarrow tag \neq 0$ | $false$ |

**Figure 3.** (a) Counterexample, (b) reduced trace formula, (c) interpolants

We have implemented support for reduction based interpolation for the theories of arrays, sets, and multisets in the Blast software model checker. In our preliminary experience, Blast, together with this added interpolation support, is able to prove many properties of programs using data structures when the data structures are represented as ADTs. For the same programs, the reachability returns a false positive if the actual implementation of the data structures are included. This is because the simple pointer analysis used by Blast to distinguish memory cells usually cannot distinguish between the different cells within the data structure implementation.

## 3. Interpolation and Reduction

We now introduce the formal definitions of interpolation and reduction, and provide characterizations for the existence of interpolants. We prove two main results in this section. The first (Theorem 8) shows that every recursively enumerable theory admits interpolation, although the interpolant may not be quantifier-free. The interpolant is quantifier-free if and only if in addition, the theory admits quantifier elimination. The second result (Theorem 11) provides a compilation technique that reduces the problem of computing interpolants for quantifier-free formulas in a theory $T$ to computing interpolants in a different theory $R$ through a compilation process. This enables us to use already implemented techniques for computing interpolants in $R$ to compute interpolants in $T$. Again, the interpolants may not be quantifier-free. On the positive side, we give conditions under which we do get quantifier free interpolants, and show e.g., that the theory of sets with cardinality constraints satisfy these conditions. On the negative side, we show that for the theories of arrays and multisets, these conditions are not satisfied, and we cannot expect to get quantifier-free interpolants. We start with standard definitions of many-sorted logics, theories, and interpolation.

### 3.1 Many Sorted Logics

**Syntax.** A *signature* $\Sigma = (S, F, P)$ consists of a set $S$ of *sorts*, a set $F$ of *function symbols*, and a set $P$ of *predicate symbols*, where the arities of the symbols in $F$ and $P$ are constructed using the sorts in $S$ (i.e., we consider the arity of a function or a predicate to be built-in the function or predicate symbol). For a signature $\Sigma$, we write $\Sigma^S$ (respectively, $\Sigma^F$, $\Sigma^P$) for $S$ (respectively $F$, $P$). For signatures $\Sigma_1$ and $\Sigma_2$, we write $\Sigma_1 \subseteq \Sigma_2$ if $\Sigma_1^S \subseteq \Sigma_2^S$, $\Sigma_1^F \subseteq \Sigma_2^F$, and $\Sigma_1^P \subseteq \Sigma_2^P$. The union and intersection of signatures is defined as the pointwise union and intersection of their component sets. For each sort $\sigma$ we fix a set $\mathcal{X}_\sigma$ of free constant symbols of sort $\sigma$ which are disjoint from the function symbols $\Sigma^F$. We also fix a set $\mathcal{X}_{\mathsf{bool}}$ of free propositional symbols.

For a signature $\Sigma$, the set of $\Sigma$-terms is the smallest set such that (1) each free constant symbol $u \in \mathcal{X}_\sigma$ is a $\Sigma$-term of sort $\sigma$ for all $\sigma \in \Sigma^S$, (2) each constant symbol $u \in \Sigma^F$ of sort $\sigma$ is a $\Sigma$-term of sort $\sigma$, and (3) $f(t_1, \ldots, t_n)$ is a $\Sigma$-term of sort $\sigma$, given $f \in \Sigma^F$ is a function symbol of arity $\sigma_1 \times \ldots \times \sigma_n \rightarrow \sigma$ and $t_i$ is a $\Sigma$-term of sort $\sigma_i$ for $i = 1, \ldots, n$.

The set of $\Sigma$-atoms is the smallest set such that (1) each propositional symbol $u \in \mathcal{X}_{\mathsf{bool}}$ is a $\Sigma$-atom, (2) $s \approx t$ is a $\Sigma$-atom if $s$ and $t$ are $\Sigma$-terms of the same sort, and (3) $p(t_1, \ldots, t_n)$ is a $\Sigma$-atom given that $p \in \Sigma^P$ is a predicate symbol of arity $\sigma_1 \times \ldots \times \sigma_n$ and $t_i$ is a $\Sigma$-term of sort $\sigma_i$ for $i = 1, \ldots, n$.

The set of quantifier-free $\Sigma$-formulas is the smallest set such that (1) each $\Sigma$-atom is a $\Sigma$-formula, (2) if $\varphi, \psi, \chi$ are $\Sigma$-formulas, so are $\neg\varphi$, $\varphi \wedge \psi$. The set of $\Sigma$-formulas is the smallest set such that (1) every quantifier-free $\Sigma$-formula is a $\Sigma$-formula, and (2) if $\varphi$ is a $\Sigma$-formula and $x \in \mathcal{X}_\sigma$ a free constant symbol, then $\forall x : \mathcal{X}_\sigma.\varphi$ and $\exists x : \mathcal{X}_\sigma.\varphi$ are $\Sigma$-formulas. We shall use the usual derived formulas $\varphi \vee \psi$, $\varphi \rightarrow \psi$, $\varphi \leftrightarrow \psi$. We also use the shorthand $s \not\approx t$ for $\neg(s \approx t)$. We write $vars(\varphi)$ for the free constant symbols in $\varphi$. We omit the prefix $\Sigma$- when it is clear from the context.

**Semantics.** For a signature $\Sigma = (S, F, P)$ and a set $X$ of free symbols over sorts in $S$, a $\Sigma$-structure $\mathcal{A}$ over $X$ is a map which interprets

1. each sort $\sigma \in S$ as a non-empty domain $A_\sigma$,
2. each free constant symbol $u \in X_\sigma$ as an element $u^{\mathcal{A}} \in A_\sigma$,
3. each free propositional symbol $u \in \mathcal{X}_{\mathsf{bool}}$ as a truth value in $\{true, false\}$,
4. each function symbol $f \in F$ of arity $\sigma_1 \times \ldots \times \sigma_n \rightarrow \sigma$ as a function $f^{\mathcal{A}} : A_{\sigma_1} \times \ldots \times A_{\sigma_n} \rightarrow A_\sigma$,
5. each predicate symbol $p \in P$ of arity $\sigma_1 \times \ldots \times \sigma_n$ as a relation $p^{\mathcal{A}} \subseteq A_{\sigma_1} \times \ldots \times A_{\sigma_n}$.

For a $\Sigma$-formula $\varphi$ with free variables $X_0 \subseteq X$, we denote by $\varphi^{\mathcal{A}}$ the evaluation of $\varphi$ under $\mathcal{A}$ (defined in the usual way). For a formula $\varphi$, we write $\mathcal{A} \models \varphi$ if $\varphi^{\mathcal{A}} = true$. A formula $\varphi$ is *satisfiable* if $\mathcal{A} \models \varphi$ for some structure $\mathcal{A}$ over $vars(\varphi)$.

### 3.2 Theories

A $\Sigma$-*theory* is a set of $\Sigma$-sentences closed under logical deduction.[3] A theory is *recursively enumerable* if the set of sentences in the theory is a recursively enumerable set. Two formulas $\varphi$ and $\psi$ are $T$-equivalent for a theory $T$ if $\varphi \leftrightarrow \psi$ is in $T$. If $\Sigma$ is a signature, $T_{\approx}^{\Sigma}$ denotes the theory of equality over $\Sigma$, that is, $T_{\approx}^{\Sigma}$ is the set of all valid $\Sigma$-sentences.

Given a $\Sigma$-theory $T$, a $T$-model is a $\Sigma$-interpretation that satisfies all sentences in $T$. A $\Sigma$-formula $\varphi$ over a set $V$ of variables is $T$-*valid* if it is satisfied by all $T$-models over $V$, is $T$-*satisfiable* if it is satisfied by some $T$-model over $V$, and is $T$-*unsatisfiable* if it is not $T$-satisfiable. The *satisfiability problem* of a $\Sigma$-theory $T$ is the problem of deciding, for every $\Sigma$-formula $\varphi$, whether or not $\varphi$ is $T$-satisfiable. The *quantifier-free satisfiability problem* of a $\Sigma$-theory $T$ is the problem of deciding, for every quantifier-free $\Sigma$-formula $\varphi$, whether or not $\varphi$ is $T$-satisfiable.

For every signature $\Sigma$, the satisfiability problem of $T_{\approx}^{\Sigma}$ is undecidable [5, 41] (indeed, semi-decidable [13]), whereas the quantifier-free satisfiability problem of $T_{\approx}^{\Sigma}$ is decidable [1].

[3] A set $T$ of $\Sigma$-sentences is closed under logical deduction if $\psi \in T$ whenever $\varphi \in T$ and $\varphi \rightarrow \psi$ is valid.

A $\Sigma$-theory $T$ *eliminates quantifiers* if for every $\Sigma$-formula $\varphi$ it is possible to effectively compute a quantifier-free $\Sigma$-formula $\psi$ such that $\varphi$ and $\psi$ are $T$-equivalent and $vars(\psi) \subseteq vars(\varphi)$.

Examples of theories that eliminate quantifiers include the theory $T_{\mathsf{int}}$ of linear integer arithmetic [35], the theory $T_{\mathsf{rat}}$ of linear rational arithmetic [42], the theory $T_{\mathsf{real}}$ of real arithmetic [40], the theory $T_{\mathsf{data}}$ of recursively defined data structures [32], and the theory $T_{\mathsf{set}}$ of sets [23].

For every signature $\Sigma$, the theory of equality $T_{\approx}^{\Sigma}$ over $\Sigma$ does not eliminate quantifiers. To see this, assume by contradiction that $T_{\approx}^{\Sigma}$ eliminates quantifiers, and let $\varphi$ be a $\Sigma$-formula. Then it is possible to effectively compute a quantifier free $\Sigma$-formula $\psi$ such that $\varphi$ and $\psi$ are equivalent. It follows that $\varphi$ and $\psi$ are equisatisfiable. Since the quantifier-free satisfiability problem of $T_{\approx}^{\Sigma}$ is decidable, we can effectively decide whether $\varphi$ is satisfiable. But this implies that the satisfiability problem of $T_{\approx}^{\Sigma}$ is decidable, a contradiction. Using similar arguments, we will prove that theories that do not eliminate quantifiers also include the theory $T_{\mathsf{array}}$ of arrays and the theory $T_{\mathsf{bag}}$ of multisets.

### 3.3 Interpolation

Let $T$ be a $\Sigma$-theory, and let $\varphi$ and $\psi$ be $\Sigma$-formulas such that $\varphi \wedge \psi$ is $T$-unsatisfiable. We say that a $\Sigma$-formula $\alpha$ is a *$T$-interpolant* of $(\varphi, \psi)$ if the following three conditions hold:

1. $\varphi \to \alpha$ is $T$-valid.
2. $\alpha \wedge \psi$ is $T$-unsatisfiable.
3. $vars(\alpha) \subseteq vars(\varphi) \cap vars(\psi)$.

A $\Sigma$-theory $T$ is *interpolating* if, for all $\Sigma$-formulas $\varphi, \psi$ such that $\varphi \wedge \psi$ is $T$-unsatisfiable, it is possible to effectively compute a $T$-interpolant $\alpha$ of $(\varphi, \psi)$. A $\Sigma$-theory $T$ is *quantifier-free interpolating* if, for all $\Sigma$-formulas $\varphi, \psi$ such that $\varphi \wedge \psi$ is $T$-unsatisfiable, it is possible to effectively compute a quantifier-free $T$-interpolant $\alpha$ of $(\varphi, \psi)$.

For every signature $\Sigma$, the theory of equality $T_{\approx}^{\Sigma}$ over $\Sigma$ is interpolating [7]. In particular, if $\varphi$ and $\psi$ are $\Sigma$-formulas such that $\varphi \wedge \psi$ is unsatisfiable, then a $T_{\approx}^{\Sigma}$-interpolant $\alpha$ of $(\varphi, \psi)$ can be extracted from any first-order proof $\Pi$ of the unsatisfiability of $\varphi \wedge \psi$ in time linear in the size of the proof $\Pi$. Methods for extracting $T_{\approx}^{\Sigma}$-interpolants from first-order proofs exist for Gentzen-like calculi [39], resolution, and tableaux [10]. We now extend these results to any recursively enumerable theory $T$.

We start with a simple normalization that simplifies the syntax of formulas in the following. A quantifier-free formula is *flat* if all atoms occuring in it are of the form $x \approx y$, $x \approx f(x_1, \dots, x_n)$, or $p(x_1, \dots, x_n)$, where $x, y, x_1, \dots, x_n$ are variables. It is easy to see that every formula $\varphi$ can be converted to an equivalent flat formula $\varphi'$ (by introducing new variables). This flat formula $\varphi'$ is called the *flat form* of $\varphi$. Further, since the conjunction or disjunction of flat formulas is also flat, we shall write, e.g., $\varphi' \wedge \psi'$ to denote the flat form of $\varphi \wedge \psi$ where $\varphi'$ is the flat form of $\varphi$ and $\psi'$ the flat form of $\psi$.

**Proposition 7.** *Let $\varphi \wedge \psi$ be a quantifier-free $T$-unsatisfiable $\Sigma$-formula. Then, in order to compute a $T$-interpolant of $(\varphi, \psi)$, one can just do the computation for a flat form $\varphi' \wedge \psi'$ of $\varphi \wedge \psi$.* □

We now characterize (quantifier-free) $T$-interpolating theories. Our main theorem shows that every recursively enumerable theory $T$ is $T$-interpolating, and additionally $T$ is quantifier-free interpolating iff additionally $T$ eliminates quantifiers.

**Theorem 8.** *1. Every recursively enumerable theory is interpolating.*

*2. Every recursively enumerable theory that eliminates quantifiers is quantifier-free interpolating.*

*3. Every quantifier-free interpolating theory eliminates quantifiers.* □

PROOF. Let $T$ be a recursively enumerable $\Sigma$-theory, and let $\varphi$ and $\psi$ be $\Sigma$-formulas such that $\varphi \wedge \psi$ is $T$-unsatisfiable. We want to effectively compute a $T$-interpolant $\alpha$ of $(\varphi, \psi)$.

By compactness, and since $T$ is recursively enumerable, one can effectively construct a finite subset $T_0 \subseteq T$ such that $\varphi \wedge \psi$ is $T_0$-unsatisfiable. Moreover, one can also construct a first-order proof $\Pi$ of the $T_0$-unsatisfiability of $\varphi \wedge \psi$. From $\Pi$, one can effectively extract a $T_{\approx}^{\Sigma}$-interpolant $\alpha$ of $(\bigwedge T_0 \wedge \varphi, \bigwedge T_0 \wedge \psi)$. (We write $\bigwedge T_0$ for the conjunction of the finitely many formulas in $T_0$.) Clearly, $\alpha$ is also a $T$-interpolant of $(\varphi, \psi)$.

For part (2), let $T$ be a recursively enumerable $\Sigma$-theory, and let $\varphi$ and $\psi$ be $\Sigma$-formulas such that $\varphi \wedge \psi$ is $T$-unsatisfiable. By Part (1), one can effectively construct a $T$-interpolant $\alpha$ of $(\varphi, \psi)$. Since $T$ eliminates quantifiers, one can effectively compute a quantifier-free $\Sigma$-formula $\beta$ such that $\alpha$ and $\beta$ are $T$-equivalent and $vars(\beta) \subseteq vars(\alpha)$. Clearly, $\beta$ is a quantifier-free $T$-interpolant of $(\varphi, \psi)$.

For part (3), let $T$ be a quantifier-free interpolating $\Sigma$-theory, let $\varphi$ be a $\Sigma$-formula, and let $\alpha$ be a quantifier-free interpolant of $(\varphi, \neg\varphi)$. Clearly, $\varphi$ and $\alpha$ are $T$-equivalent and $vars(\alpha) \subseteq vars(\varphi)$. ■

From Theorem 8, it follows that examples of theories that are interpolating include the theory $T_{\mathsf{int}}$ of integer linear arithmetic, the theory $T_{\mathsf{rat}}$ of rational linear arithmetic, the theory $T_{\mathsf{real}}$ of real arithmetic, the theory $T_{\mathsf{data}}$ of recursively defined data structures, the theory $T_{\mathsf{array}}$ of arrays, the theory $T_{\mathsf{set}}$ of sets, and the theory $T_{\mathsf{bag}}$ of multisets.

Further, it follows that examples of theories that are quantifier-free interpolating include the theory $T_{\mathsf{int}}$ of linear integer arithmetic, the theory $T_{\mathsf{rat}}$ of rational linear arithmetic, the theory $T_{\mathsf{real}}$ of real arithmetic, the theory $T_{\mathsf{data}}$ of recursively defined data structures, and the theory $T_{\mathsf{set}}$ of sets. In constrast, we will see that the theory $T_{\mathsf{array}}$ of arrays and the theory $T_{\mathsf{bag}}$ of multisets are not quantifier-free interpolating.

**Example 9. Arithmetic.** The theory $T_{\mathsf{int}}$ of *integer linear arithmetic* is the theory of the structure of integer numbers $\langle \mathbb{Z}, 0, 1, +, \leq, \equiv_n \rangle$. The theory $T_{\mathsf{int}}$ is recursively enumerable and eliminates quantifiers [35]. Consequently, by Theorem 8, $T_{\mathsf{int}}$ is quantifier-free interpolating. In addition, $T_{\mathsf{int}}$ has a decidable quantifier-free satisfiability problem (this is essentially integer linear programming), and is implemented in decision procedures such as Simplify and CVC [8, 37].

The theory $T_{\mathsf{rat}}$ of *rational linear arithmetic* is the theory of the structure of rational numbers $\langle \mathbb{Q}, 0, 1, +, \leq \rangle$. The theory $T_{\mathsf{rat}}$ is recursively enumerable, and eliminates quantifiers [42]. Consequently, by Theorem 8, $T_{\mathsf{rat}}$ is quantifier-free interpolating. In addition, $T_{\mathsf{rat}}$ has a decidable quantifier-free satisfiability problem (this is essentially linear programming).

The theory $T_{\mathsf{real}}$ of *real arithmetic* is the theory of the structure of real numbers $\langle \mathbb{R}, 0, 1, +, \times, \leq \rangle$. The theory $T_{\mathsf{real}}$ is recursively enumerable, and eliminates quantifiers [40]. Consequently, by Theorem 8, $T_{\mathsf{real}}$ is quantifier-free interpolating. In addition, $T_{\mathsf{real}}$ has a decidable quantifier-free satisfiability problem (through Fourier-Motzkin elimination). □

**Example 10. Recursively defined data structures.** The theory $T_{\mathsf{data}}$ of *recursively defined data structures* has a signature $\Sigma_{\mathsf{data}}$ containing one sort `data`, the binary function symbols `cons`, and the unary function symbols `car` and `cdr`. The theory $T_{\mathsf{data}}$ is axiomated by

$$\begin{aligned}
&\mathsf{cons}(\mathsf{car}(x), \mathsf{cdr}(x)) = x\,,\\
&\mathsf{car}(\mathsf{cons}(x, y)) = x\,,\\
&\mathsf{cdr}(\mathsf{cons}(x, y)) = y\,,\\
&x \not\approx t(x)\,,
\end{aligned}$$

where $t$ is a term built up form $x$ by using finitely many applictaions of the unary funcation symbols $\mathsf{car}$ and $\mathsf{cdr}$. The theory $T_{\mathsf{data}}$ is recursively enumerable, and eliminates quantifiers [26]. Consequently, by Theorem 8, $T_{\mathsf{data}}$ is quantifier-free interpolating. $T_{\mathsf{data}}$ has a decidable quantifier-free satisfiability problem [30, 32]. $\square$

### 3.4 Reduction

Theorem 8 provides a characterization of first order theories that admit (quantifier-free) interpolation. In practice, however, producing efficient implementations of interpolating decision procedures for every individual theory is a daunting engineering task. Further, the construction of the interpolant in the proof of Theorem 8 used compactness to construct a finite subset $T_0$ of $T$, which may be algorithmically inefficient. Instead, we use a compilation or reduction approach [22], where the satisfiability and interpolation-construction for a theory is proved by reducing to a different, often simpler, theory for which efficient satisfiability and interpolation procedures have already been implemented. In particular, the target for our reduction functions is the combination of the theory of equality with uninterpreted functions and linear arithmetic, for which interpolating decision procedures are available [27].

Let $T$ be a $\Sigma$-theory, and let $R$ be an $\Omega$-theory such that $\Omega \subseteq \Sigma$, $\Omega^{\mathrm{S}} = \Sigma^{\mathrm{S}}$, and $R \subseteq T$. We say that $T$ *reduces* to $R$ if there is a computable map from flat $\Sigma$-atoms to $\Omega$-formulae such that, if we apply this map to a quantifier-free flat $\Sigma$-formula $\varphi$, obtaining an $\Omega$-formula $\varphi^*$, then

1. $\varphi$ and $\varphi^*$ are $T$-equivalent.
2. If $\varphi^*$ is $R$-satisfiable then $\varphi$ is $T$-satisfiable.

**Theorem 11.** *Let $T$ be a $\Sigma$-theory, and let $R$ be an $\Omega$-theory such that $\Omega \subseteq \Sigma$ and $\Omega^{\mathrm{S}} = \Sigma^{\mathrm{S}}$, and $R \subseteq T$. Assume that*

*(i) $T$ reduces to $R$,*

*(ii) It is possible to compute an $R$-interpolant of $(\varphi, \psi)$ whenever $\varphi \wedge \psi$ is an $R$-unsatisfiable $\Omega$-formula.*

*Then it is possible to compute a $T$-interpolant of $(\varphi, \psi)$ whenever $\varphi \wedge \psi$ is a $T$-unsatisfiable quantifier-free $\Sigma$-formula.* $\square$

PROOF. Without loss of generality, let $\varphi \wedge \psi$ be a $T$-unsatisfiable flat quantifier-free $\Sigma$-formula, and let us construct $(\varphi^*, \psi^*)$. Note that $\varphi^* \wedge \psi^*$ is $R$-unsatisfiable. Thus, we can compute an $R$-interpolant $\alpha$ of $(\varphi^*, \psi^*)$. We claim that $\alpha$ is also a $T$-interpolant of $(\varphi, \psi)$.

Since $vars(\alpha) \subseteq vars(\varphi^*) \cap vars(\psi^*)$, $vars(\varphi) \subseteq vars(\varphi^*)$, and $vars(\psi^*) \subseteq vars(\psi)$, it follows that $vars(\alpha) \subseteq vars(\varphi) \cap vars(\psi)$.

Next, let $\mathcal{A} \models_T \varphi$. Then $\mathcal{A} \models_T \varphi^*$, which implies that $\mathcal{A}^{\Omega, vars(\varphi^*)} \models_R \varphi^*$, which implies $\mathcal{A}^{\Omega, vars(\varphi^*)} \models_R \alpha$, which implies $\mathcal{A} \models_T \alpha$. Summing up, $\varphi \rightarrow \alpha$ is $T$-valid.

Finally, assume by contradiction that $\alpha \wedge \psi$ is $T$-satisfiable. Then there exists a $\Sigma$-interpretation $\mathcal{A}$ such that $\mathcal{A} \models_T \alpha \wedge \psi$. It follows that $\mathcal{A} \models_T \alpha \wedge \psi^*$, which implies $\mathcal{A}^{\Omega, vars(\psi^*)} \models_R \alpha \wedge \psi^*$. But then, $\alpha \wedge \psi^*$ is $R$-satisfiable, a contradiction. ■

Note that if in Theorem 11 either one of the theories $T$ or $R$ eliminates quantifiers, then $T$ is quantifier-free interpolating and we can compute a quantifier-free $T$-interpolant $\beta$ of $(\varphi, \psi)$. This process is depicted in Figure 4.

$$\begin{tikzcd}
\boxed{\varphi \wedge \psi} \arrow[d, "\text{reduction of } T \text{ to } R"] \\
\boxed{\varphi^* \wedge \psi^*} \arrow[d, "\text{computation of an } R\text{-interpolant}"] \\
\boxed{\alpha} \arrow[d, "\text{quantifier elimination}"] \\
\boxed{\beta}
\end{tikzcd}$$

**Figure 4.** Computing quantifier-free $T$-interpolants when the $\Sigma$-theory $T$ reduces to the $\Omega$-theory $R$, and either $T$ or $R$ eliminates quantifiers. $\varphi \wedge \psi$ is a $T$-unsatisfiable flat quantifier-free $\Sigma$-formula. $\varphi^* \wedge \psi^*$ is an $\Omega$-formula obtained from $\varphi \wedge \psi$ by a reduction function. $\alpha$ is an $R$-interpolant of $(\varphi, \psi)$, as well as a $T$-interpolant of $(\varphi, \psi)$. $\beta$ is obtained by eliminating quantifiers from $\alpha$ in either the theory $T$ or the theory $R$. Finally, $\beta$ is a quantifier-free $T$-interpolant of $(\varphi, \psi)$.

## 4. Interpolation Algorithms for Data Structures

We now apply Theorem 11 to obtain interpolants for the theories of arrays, sets, and multisets.

### 4.1 Arrays

The theory $T_{\mathsf{array}}$ of *arrays* as a signature $\Sigma_{\mathsf{array}}$ containing a sort $\mathsf{elem}$ for elements, are sort $\mathsf{index}$ for indices, and a sort $\mathsf{array}$ of arrays, plus the function symbols $\mathsf{read}$ of arity $\mathsf{array} \times \mathsf{index} \longrightarrow \mathsf{elem}$, and $\mathsf{write}$ of arity $\mathsf{array} \times \mathsf{index} \times \mathsf{elem} \longrightarrow \mathsf{array}$. The theory $T_{\mathsf{array}}$ is axiomatized by

$$\begin{aligned}
&\mathsf{read}(\mathsf{write}(a, i, e), i) \approx e\,,\\
&i \not\approx j \;\rightarrow\; \mathsf{read}(\mathsf{write}(a, i, e), j) \approx \mathsf{read}(a, j)\,,\\
&(\forall_{\mathsf{index}}\, i)(\mathsf{read}(a, i) \approx \mathsf{read}(b, i)) \;\rightarrow\; a \approx b\,.
\end{aligned}$$

The theory $T_{\mathsf{array}}$ is recursively enumerable. Consequently, by Theorem 8, $T_{\mathsf{array}}$ is interpolating. In this section, we show that: (a) the satisfiability problem of $T_{\mathsf{array}}$ is undecidable, (b) the theory $T_{\mathsf{array}}$ does not eliminate quantifiers, and consequently, (c) the theory $T_{\mathsf{array}}$ is not quantifier-free interpolating. Notice that $T_{\mathsf{array}}$ does have a decidable quantifier-free satisfiability problem [38].

Moreover, we prove that $T_{\mathsf{array}}$ reduces to $T_{\approx}^{\Omega}$, where $\Omega^{\mathrm{S}} = \{\mathsf{elem}, \mathsf{index}, \mathsf{array}\}$, $\Omega^{\mathrm{F}} = \{\mathsf{read}\}$, and $\Omega^{\mathrm{P}} = \emptyset$. Consequently, by Theorem 11, and provided that $\varphi \wedge \psi$ is quantifier-free, we can reduce the problem of computing $T_{\mathsf{array}}$-interpolants[4] of $(\varphi, \psi)$ to the problem of computing $T_{\approx}^{\Omega}$-interpolants.

**Theorem 12.** *1. The satisfiability problem of the theory $T_{\mathsf{array}}$ of arrays is undecidable.*

*2. The theory $T_{\mathsf{array}}$ of arrays does not eliminate quantifiers.* $\square$

PROOF (*Sketch*). Consider the theory $T_{\mathsf{array}}^2$ which is the same as $T_{\mathsf{array}}$, but the sort $\mathsf{elem}$ and $\mathsf{index}$ are identified. Clearly, the satisfiability problem of $T_{\mathsf{array}}^2$ is undecidable (Gurevich [15]).

We want to reduce the satisfiability problem of $T_{\mathsf{array}}$ to the satisfiability problem of $T_{\mathsf{array}}^2$. This can be done as follows. Specify

[4] Since $T_{\mathsf{array}}$ is not quantifi er-free interpolating, these interpolants are, in general, not quantifi er-free.

that a function $h : \mathsf{index} \to \mathsf{elem}$ is bijective with the formulas

$$(\forall_{\mathsf{index}}\, i, j)(\mathsf{read}(h, i) \approx \mathsf{read}(h, j) \to i \approx j)\,,$$
$$(\forall_{\mathsf{elem}}\, e)(\exists_{\mathsf{index}}\, i)(\mathsf{read}(h, i) \approx e)\,.$$

Then, whenever we want to express that $e_1 = f(e_2)$, it suffices to say that $\mathsf{read}(h, i) \approx e_2 \land \mathsf{read}(f, i) \approx e_1$.

For part (2), assume by contradiction that $T_{\mathsf{array}}$ eliminates quantifiers, and let $\varphi$ be a $\Sigma_{\mathsf{array}}$-formula. Then it is possible to effectively compute a quantifier free $\Sigma_{\mathsf{array}}$-formula $\psi$ such that $\varphi$ and $\psi$ are $T_{\mathsf{array}}$-equivalent. It follows that $\varphi$ and $\psi$ are $T_{\mathsf{array}}$-equisatisfiable. Since the quantifier-free satisfiability problem of $T_{\mathsf{array}}$ is decidable, we can effectively decide whether $\varphi$ is $T_{\mathsf{array}}$-satisfiable. But this implies that the satisfiability problem of $T_{\mathsf{array}}$ is decidable, a contradiction. ■

By Theorem 8 and Theorem 12, we get the following.

**Corollary 13.** The theory $T_{\mathsf{array}}$ of arrays is not quantifier-free interpolating. □

In fact, even if $\varphi$ and $\psi$ are quantifier-free, their interpolant may require quantification. Consider $h' \approx \mathsf{write}(h, i, e)$ and $(a \neq b) \land (\mathsf{read}(h, a) \not\approx \mathsf{read}(h', a)) \land (\mathsf{read}(h, b) \not\approx \mathsf{read}(h', b))$ whose conjunction is unsatisfiable, but there is no quantifier-free interpolant over the common variables $h$ and $h'$.

**Proposition 14.** *The theory $T_{\mathsf{array}}$ of arrays reduces to the theory of equality $T^{\Omega}_{\approx}$, where $\Omega^{\mathrm{S}} = \{\mathsf{elem}, \mathsf{index}, \mathsf{array}\}$, $\Omega^{\mathrm{F}} = \{\mathsf{read}\}$, and $\Omega^{\mathrm{P}} = \emptyset$.* □

PROOF. The following map from flat $\Sigma_{\mathsf{array}}$-literals to $\Omega$-formulas will do the job (literals not mentioned are left unchanged). Every literal $a \approx_{\mathsf{array}} b$ is mapped to $(\forall_{\mathsf{index}}\, i)(\mathsf{read}(a, i) \approx \mathsf{read}(b, i))$, and every atom $a \approx \mathsf{write}(b, i, e)$ is mapped to $\mathsf{read}(a, i) \approx e \land (\forall_{\mathsf{index}}\, j)(j \not\approx i \to \mathsf{read}(a, j) \approx \mathsf{read}(b, j))$. Thus, by Theorem 11, it follows that $T_{\mathsf{array}}$ reduces to $T^{\Omega}_{\approx}$. ■

## 4.2 Sets

The theory $T_{\mathsf{set}}$ of *sets with finite cardinality constraints* has a signature $\Sigma_{\mathsf{set}}$ containing one sort $\mathsf{elem}$ for elements and one sort $\mathsf{set}$ for sets of elements, plus the following symbols:

- the constant symbols $\emptyset$ (empty set) and $\mathbb{1}$ (full set), both of sort $\mathsf{set}$.
- the binary function symbols $\cup$ (union), $\cap$ (intersection), and $\setminus$ (difference), of arity $\mathsf{set} \times \mathsf{set} \to \mathsf{set}$.
- the unary function symbol $\{\cdot\}$ (singleton), of arity $\mathsf{elem} \to \mathsf{set}$.
- the binary predicate symbol $\in$, of arity $\mathsf{elem} \times \mathsf{set}$.
- for each natural number $k$, the unary predicate symbols $|\cdot| \geq k$ and $|\cdot| \approx k$, both of arity $\mathsf{set}$.

A *standard* $\mathsf{set}$*-structure* is a $\Sigma_{\mathsf{set}}$-structure satisfying the following properties:

1. $A_{\mathsf{elem}}$ is finite.
2. $A_{\mathsf{set}} = \mathcal{P}(A_{\mathsf{elem}})$.
3. The symbols $\emptyset$, $\mathbb{1}$, $\cup$, $\cap$, $\setminus$, $\{\cdot\}$, and $\in$ are interpreted according to their standard meaning over sets; in particular, we have $\mathbb{1}^{\mathcal{A}} = A_{\mathsf{elem}}$.
4. $[|x| \geq k]^{\mathcal{A}} = \mathit{true}$ if and only if $\mathsf{card}(x^{\mathcal{A}}) \geq k$, for all sets $x \in A_{\mathsf{set}}$ and $k \in \mathbb{N}$, where the cardinality $\mathsf{card}(x)$ of a set $x$ is the number of elements in $x$.
5. $[|x| \approx k]^{\mathcal{A}} = \mathit{true}$ if and only if $\mathsf{card}(x^{\mathcal{A}}) = k$, for all sets $x \in A_{\mathsf{set}}$ and $k \in \mathbb{N}$.

The theory $T_{\mathsf{set}}$ is the set of all $\Sigma_{\mathsf{set}}$-sentences that are true in all standard $\mathsf{set}$-structures.

The theory $T_{\mathsf{set}}$ is recursively enumerable, and eliminates quantifiers [23, Fact 1, page 7]. Consequently, by Theorem 8, $T_{\mathsf{set}}$ is quantifier-free interpolating. Further, it has a decidable satisfiability problem [23].

In this section we prove that $T_{\mathsf{set}}$ reduces to $T^{\Omega}_{\approx}$, where $\Omega^{\mathrm{S}} = \{\mathsf{elem}, \mathsf{set}\}$, $\Omega^{\mathrm{F}} = \emptyset$, and $\Omega^{\mathrm{P}} = \{\in\}$. Consequently, by Theorem 11, and provided that $\varphi \land \psi$ is quantifier-free, we can reduce the problem of computing quantifier-free $T_{\mathsf{set}}$-interpolants of $(\varphi, \psi)$ to the problem of computing $T^{\Omega}_{\approx}$-interpolants.

**Proposition 15.** *The theory $T_{\mathsf{set}}$ of sets with finite cardinality constraints reduces to the theory of equality $T^{\Omega}_{\approx}$, where $\Omega^{\mathrm{S}} = \{\mathsf{elem}, \mathsf{set}\}$, $\Omega^{\mathrm{F}} = \emptyset$, and $\Omega^{\mathrm{P}} = \{\in\}$.* □

PROOF. Figure 5 shows the reduction function from flat $\Sigma_{\mathsf{set}}$-literals to $\Omega$-formulas (literals not mentioned are left unchanged). Thus, by Theorem 11, it follows that $T_{\mathsf{set}}$ reduces to $T^{\Omega}_{\approx}$. ■

**Example 16.** Let

$$\varphi: \quad x \approx \{a\}\,,$$
$$\psi: \quad x \approx \{b, c\} \land b \not\approx c\,.$$

After performing the reduction, we get

$$\varphi^*: \quad a \in x \land (\forall_{\mathsf{elem}}\, e)(e \in x \to e \approx a)\,,$$
$$\psi^*: \quad b \in x \land c \in x \land (\forall_{\mathsf{elem}}\, e)(e \in x \to (e \approx b \lor e \approx c)) \land b \not\approx c\,.$$

A Gentzen-style proof yields the interpolant

$$(\forall_{\mathsf{elem}}\, e_1, e_2)(e_1 \notin x \lor e_2 \notin x \lor e_1 \approx e_2)\,,$$

which is $T_{\mathsf{set}}$-equivalent to the quantifier-free formula

$$|x| \approx 0 \lor |x| \approx 1\,.$$ □

Unfortunately, the requirement in Theorem 11 that the original $T$-formula is quantifier-free cannot be relaxed. Take $\varphi$ to be a $T_{\mathsf{set}}$-unsatisfiable formula that says that there are exactly 3 sets:

$$(\exists_{\mathsf{set}} x, y, z)(x \not\approx y \land y \not\approx z \land x \not\approx z \land (\forall_{\mathsf{set}} w)(w \approx x \lor w \approx y \lor w \approx z))$$

When reduced to the theory of equality, the resulting formula is satisfiable, because the theory of equality does not know that the number of sets is $2^n$ where $n$ is the number of elements.

## 4.3 Multisets

The theory $T_{\mathsf{bag}}$ of *multisets* has a signature $\Sigma_{\mathsf{bag}}$ extending $\Sigma_{\mathsf{int}}$ with a sort $\mathsf{elem}$ for elements, and a sort $\mathsf{bag}$ for multisets, plus the following symbols:

- the constant symbol $[\![\,]\!]$, of sort $\mathsf{bag}$;
- the function symbols:
  - $[\![\cdot]\!]^{(\cdot)}$, of sort $\mathsf{elem} \times \mathsf{int} \to \mathsf{bag}$;
  - $\sqcup$, $\uplus$, and $\sqcap$, of sort $\mathsf{bag} \times \mathsf{bag} \to \mathsf{bag}$;
  - $\mathsf{count}$, of sort $\mathsf{elem} \times \mathsf{bag} \to \mathsf{int}$.

A *standard* $\mathsf{bag}$*-structure* $\mathcal{A}$ is a $\Sigma_{\mathsf{bag}}$-structure satisfying the following conditions:

- $\mathcal{A}^{\Sigma_{\mathsf{int}}}$ is the standard $\mathsf{int}$-structure;
- $A_{\mathsf{bag}} = \mathbb{N}^{A_{\mathsf{elem}}}$;
- the symbols $[\![\,]\!]$, $[\![\cdot]\!]^{(\cdot)}$, $\sqcup$, $\uplus$, and $\sqcap$ are interpreted according to their standard interpretation over multisets, i.e., $[\![\,]\!]^{\mathcal{A}}$ is the empty multiset, $[\![x]\!]^{(n)\mathcal{A}}$ constructs a multiset with $n$ copies of element $x$ if $n \geq 0$ and the empty multiset if $n < 0$, and $x \sqcup^{\mathcal{A}} y$ (resp. $x \uplus^{\mathcal{A}} y$, $x \sqcap^{\mathcal{A}} y$) is a multiset that maps $e$ to $\max(x(e), y(e))$ (resp. $x(e) + y(e)$, $\min(x(e), y(e))$).

$$
\begin{array}{lcl}
x \approx_{\mathsf{set}} y & \Longrightarrow & (\forall_{\mathsf{elem}}\, e)((e \in x \wedge e \in y) \vee (e \notin x \wedge e \notin y)) \\
x \approx \emptyset & \Longrightarrow & (\forall_{\mathsf{elem}}\, e)(e \notin x) \\
x \approx \mathbb{1} & \Longrightarrow & (\forall_{\mathsf{elem}}\, e)(e \in x) \\
x \approx y \cup z & \Longrightarrow & (\forall_{\mathsf{elem}}\, e)(e \in x \leftrightarrow (e \in y \vee e \in z)) \\
x \approx y \cap z & \Longrightarrow & (\forall_{\mathsf{elem}}\, e)(e \in x \leftrightarrow (e \in y \wedge e \in z)) \\
x \approx y \setminus z & \Longrightarrow & (\forall_{\mathsf{elem}}\, e)(e \in x \leftrightarrow (e \in y \wedge e \notin z)) \\
x \approx \{e_0\} & \Longrightarrow & e_0 \in x \wedge (\forall_{\mathsf{elem}}\, e)(e \in x \rightarrow e \approx e_0) \\
|x| \geq k & \Longrightarrow & (\exists_{\mathsf{elem}}\, e_1, \ldots, e_k) \left[ \left( \bigwedge_{i=1}^{k} e_i \in x \right) \wedge \left( \bigvee_{1 \leq i < j \leq k} (e_i \not\approx e_j) \right) \right] \\
|x| \approx k & \Longrightarrow & (\exists_{\mathsf{elem}}\, e_1, \ldots, e_k) \left[ \left( \bigwedge_{i=1}^{k} e_i \in x \right) \wedge \left( \bigvee_{1 \leq i < j \leq k} (e_i \not\approx e_j) \right) \wedge (\forall_{\mathsf{elem}}\, e)(e \in x \rightarrow \bigvee_{i=1}^{n} e \approx e_i) \right]
\end{array}
$$

**Figure 5.** Reduction function for sets

- $\mathsf{count}^{\mathcal{A}}(e, x) = x(e)$, for each $e \in A_{\mathsf{elem}}$ and $x \in A_{\mathsf{bag}}$.

The theory $T_{\mathsf{bag}}$ is the set of all $\Sigma_{\mathsf{bag}}$-sentences that are true in all standard $\mathsf{bag}$-structures.

The theory $T_{\mathsf{bag}}$ is recursively enumerable. Consequently, by Theorem 8, $T_{\mathsf{bag}}$ is interpolating. In this section, we show that: (a) the satisfiability problem of $T_{\mathsf{bag}}$ is undecidable, (b) the theory $T_{\mathsf{bag}}$ does not eliminate quantifiers, and (c) the theory $T_{\mathsf{bag}}$ is not quantifier-free interpolating. Notice that $T_{\mathsf{bag}}$ does have a decidable quantifier-free satisfiability problem [44].

Moreover, we prove that $T_{\mathsf{bag}}$ reduces to $T_{\mathsf{int}} \cup T_{\approx}^{\Omega}$, where $\Omega^{\mathrm{S}} = \{\mathsf{elem}, \mathsf{int}, \mathsf{bag}\}$, $\Omega^{\mathrm{F}} = \{\mathsf{count}\}$, and $\Omega^{\mathrm{P}} = \emptyset$. Consequently, by Theorem 11, and provided that $\varphi \wedge \psi$ is quantifier-free, we can reduce the problem of computing $T_{\mathsf{bag}}$-interpolants[5] of $(\varphi, \psi)$ to the problem of computing $T_{\mathsf{int}} \cup T_{\approx}^{\Omega}$-interpolants.

**Theorem 17.** *1. The satisfiability problem of the theory $T_{\mathsf{bag}}$ of bags is undecidable.*

*2. The theory $T_{\mathsf{bag}}$ of multisets does not eliminate quantifiers.* □

PROOF (*Sketch*). Consider the theory $T^2_{\mathsf{bag}}$ which is the same as $T_{\mathsf{bag}}$, but the sort $\mathsf{elem}$ and $\mathsf{int}$ are identified. Clearly, the satisfiability problem of $T^2_{\mathsf{bag}}$ is undecidable [9].

We want to reduce the satisfiability problem of $T_{\mathsf{bag}}$ to the satisfiability problem of $T^2_{\mathsf{bag}}$. This can be done as follows. Specify that a function $h : \mathsf{elem} \rightarrow \mathsf{int}$ is bijective with the formulas

$$(\forall_{\mathsf{elem}}\, a, b)(\mathsf{count}(h, a) \approx \mathsf{count}(h, b) \rightarrow a \approx b)\,,$$
$$(\forall_{\mathsf{int}}\, u)(\exists_{\mathsf{elem}}\, a)(\mathsf{count}(h, a) \approx u)\,.$$

Then, whenever we want to express that $u = f(v)$, it suffices to say that $\mathsf{count}(h, a) \approx v \wedge \mathsf{count}(f, a) \approx u$.

For part (2), assume by contradiction that $T_{\mathsf{bag}}$ eliminates quantifiers, and let $\varphi$ be a $\Sigma_{\mathsf{bag}}$-formula. Then it is possible to effectively compute a quantifier free $\Sigma_{\mathsf{bag}}$-formula $\psi$ such that $\varphi$ and $\psi$ are $T_{\mathsf{bag}}$-equivalent. It follows that $\varphi$ and $\psi$ are $T_{\mathsf{bag}}$-equisatisfiable. Since the quantifier-free satisfiability problem of $T_{\mathsf{bag}}$ is decidable, we can effectively decide whether $\varphi$ is $T_{\mathsf{bag}}$-satisfiable. But this implies that the satisfiability problem of $T_{\mathsf{bag}}$ is decidable, a contradiction. ■

**Corollary 18.** The theory $T_{\mathsf{bag}}$ of multisets is not quantifier-free interpolating. □

[5] Since $T_{\mathsf{bag}}$ is not quantifi er-free interpolating, these interpolants are, in general, not quantifi er-free.

In fact, even if $\varphi$ and $\psi$ are quantifier-free, their interpolant may require quantification. Consider $h \approx [\![e]\!]^{(1)}$ and $(a \neq b) \wedge (\mathsf{count}(a, h) \not\approx 0 \wedge (\mathsf{count}(b, h) \not\approx 0$ whose conjunction is unsatisfiable, but there is no quantifier-free interpolant over the common variables $h$ and $h'$.

**Proposition 19.** *The theory $T_{\mathsf{bag}}$ of multisets reduces to the theory $T_{\mathsf{int}} \cup T_{\approx}^{\Omega}$, where $\Omega^{\mathrm{S}} = \{\mathsf{elem}, \mathsf{int}, \mathsf{bag}\}$, $\Omega^{\mathrm{F}} = \{\mathsf{count}\}$, and $\Omega^{\mathrm{P}} = \emptyset$.* □

PROOF. Figure 6 shows the reduction function from flat $\Sigma_{\mathsf{array}}$-literals to $(\Sigma_{\mathsf{int}} \cup \Omega)$-formulas (literals not mentioned are left unchanged). Thus, by Theorem 11, it follows that $T_{\mathsf{bag}}$ reduces to $T_{\approx}^{\Omega}$. ■

## 4.4 Implementation using Foci

Foci [27] is an implementation of an interpolating decision procedure for the quantifier-free theory of equality and arithmetic. Our reduction introduces quantifiers, we now sketch how we can nevertheless use Foci to implement our procedure.

Let $\varphi^* \wedge \psi^*$ be an $R$-formula obtained by the reduction algorithms presented above of a $T$-formula $\phi \wedge \psi$ where $T$ is the theory of arrays, sets, or multisets, and $R$ is the theory of equality, or the theory of equality and arithmetic. Assume $\varphi^* \wedge \psi^*$ is a flat formula in negation normal form (i.e., negations are only applied to atomic formulas). Even though $\varphi^* \wedge \psi^*$ is not quantifier-free, we can obtain an $R$-interpolant for $\varphi^* \wedge \psi^*$ using a decision procedure for the quantifier-free fragment of these theories as follows.

First, we replace each existentially quantified variable using a fresh Skolem constant and remove the existential quantifiers. Notice that in all three reductions above, the existential quantifiers are not in the scope of any universals, therefore Skolem constants suffice. In the resulting formula (that may contain universal formulas), we instantiate each universally quantified variable with a constant that already appears in the formula. It can be shown that this is complete to show $R$-unsatisfiability for the theory $T$ of arrays, sets, or multisets. The resulting formula (after these instantiations) is quantifier-free, and we can use an Foci to compute a (quantifier-free) interpolant. However, because of the quantifier instantiations above, this interpolant will have Skolem constants introduced for the existential quantifiers. The interpolant for the theory $T$ will have quantifiers that can be added back to the output from Foci using, e.g., the tableau based interpolant computation method in [10].

$$
\begin{array}{lcl}
x \approx_{\mathsf{bag}} y & \Longrightarrow & (\forall_{\mathsf{elem}}\, e)(\mathsf{count}(x, e) \approx \mathsf{count}(y, e)) \\
x \approx [\![\,]\!] & \Longrightarrow & (\forall_{\mathsf{elem}}\, e)(\mathsf{count}(x, e) \approx 0) \\
x \approx y \sqcup z & \Longrightarrow & (\forall_{\mathsf{elem}}\, e)(\mathsf{count}(e, x) \approx \mathsf{max}(\mathsf{count}(e, y), \mathsf{count}(e, z))) \\
x \approx y \sqcap z & \Longrightarrow & (\forall_{\mathsf{elem}}\, e)(\mathsf{count}(e, x) \approx \mathsf{min}(\mathsf{count}(e, y), \mathsf{count}(e, z))) \\
x \approx y \uplus z & \Longrightarrow & (\forall_{\mathsf{elem}}\, e)(\mathsf{count}(e, x) \approx \mathsf{count}(e, y) + \mathsf{count}(e, z)) \\
x \approx [\![e_0]\!]^{(u)} & \Longrightarrow & \mathsf{count}(e_0, x) = \mathsf{max}(0, u) \wedge (\forall_{\mathsf{elem}}\, e)(e \not\approx e_0 \rightarrow \mathsf{count}(e, x) \approx 0)
\end{array}
$$

**Figure 6.** Reduction from $T_{\mathsf{bag}}$

## 5. Conclusion

Interpolation based abstraction is a powerful technique for approximate reachability (and safety) checking for systems. We have extended the potential of the technique by demonstrating how to compute interpolants for several theories of interest in program verification. Further, the reduction approach allows quick implementations by allowing hooking into already-existing efficient implementations. There are several directions of future work. One main limitation of the approach is that suitable predicates may not be derivable by considering individual counterexamples. For example, consider the program

```
for(i=0; i< n; i++) { a[i] := 0; }
for(i=0; i< n; i++) { assert(a[i] = 0); }
```

While correctness depends on the invariant $\forall 0 \leq i < n.a[i] = 0$ after the first loop, the interpolant based technique can come up with infinitely many predicates of the form $a[0] = 0, a[1] = 0, \ldots$. We leave the problem of generalizing from particular counterexamples as an interesting open problem. On the practical front, we are currently applying Blast together with these more powerful predicate generation capabilities to prove larger programs of practical interest.

## References

[1] Wilhelm Ackermann. *Solvable Cases of the Decision Problem*. North-Holland, 1954.

[2] Michael I. Schwartzbach Anders Moller. The pointer assertion logic engine. In *PLDI 01: Programming Language Design and Implementation*, pages 221–231. ACM, 2001.

[3] T. Ball and S.K. Rajamani. The SLAM project: debugging system software via static analysis. In *POPL 02: Principles of Programming Languages*, pages 1–3. ACM, 2002.

[4] S. Chaki, E.M. Clarke, A. Groce, S. Jha, and H. Veith. Modular verifi cation of software components in C. *IEEE Trans. Software Eng.*, 30(6):388–402, 2004.

[5] Alonzo Church. A note on the Entscheidungsproblem. *Journal of Symbolic Logic*, 1:101–102, 1936.

[6] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refi nement. In *CAV 00: Computer-Aided Verifi cation*, LNCS 1855, pages 154–169. Springer, 2000.

[7] William Craig. Linear reasoning: A new form of the Herbrand-Gentzen theorem. *Journal of Symbolic Logic*, 22(3):250–268, 1957.

[8] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005.

[9] Peter J. Downey. Undeciability of Presburger arithmetic with a single monadic predicate letter. Technical Report 18-72, Center for Research in Computing Technology, Havard University, 1972.

[10] Melvin C. Fitting. *First-Order Logic and Automated Theorem Proving*. Graduate Texts in Computer Science. Springer, 2nd edition, 1996.

[11] C. Flanagan, K.R.M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *PLDI 02: Programming Language Design and Implementation*, pages 234–245. ACM, 2002.

[12] J.S. Foster, T. Terauchi, and A. Aiken. Flow-sensitive type qualifi ers. In *PLDI 02: Programming Language Design and Implementation*, pages 1–12. ACM, 2002.

[13] Kurt Gödel. *Über die Vollständigkeit des Logikkalküls*. PhD thesis, University of Vienna, 1929.

[14] S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *CAV 97: Computer-aided Verifi cation*, LNCS 1254, pages 72–83. Springer, 1997.

[15] Yuri Gurevich. The decision problem for standar classes. *Journal of Symbolic Logic*, 41(2), 1976.

[16] J. Guttag. *The specifi cation and applicatons to programming of abstract data types*. PhD thesis, University of Toronto, 1975.

[17] T.A. Henzinger, R. Jhala, R. Majumdar, and K.L. McMillan. Abstractions from proofs. In *POPL 04: Principles of Programming Languages*, pages 232–244. ACM, 2004.

[18] T.A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL 02: Principles of Programming Languages*, pages 58–70. ACM, 2002.

[19] C.A.R. Hoare. Proof of correctness of data representations. *Acta Inf.*, 1:271–281, 1972.

[20] R. Jhala and K.L. McMillan. Interpolant-based transition relation approximation. In *CAV 05: Computer-Aided Verifi cation*, LNCS 3576, pages 39–51. Springer, 2005.

[21] R. Jhala and K.L. McMillan. A practical and complete approach to predicate abstraction. In *TACAS 06*. Springer, 2006.

[22] Deepak Kapur and Calogero G. Zarba. A reduction approach to decision procedures. `http://cs.unm.edu/~kapur/`, 2005.

[23] Viktor Kuncak and Martin C. Rinard. The fi rst-order theory of sets with cardinality constraints is decidable. Technical Report CSAIL 958, MIT, 2004.

[24] P. Lam, V. Kuncak, and M.C. Rinard. Hob: A tool for verifying data structure consistency. In *CC 05*, pages 237–241, 2005.

[25] B. Liskov and S. Zilles. Programming with abstract data types. In *Symposium on very high level programming languages*. 1974.

[26] A. Mal'cev. Axiomatizable classes of locally free algebras of certain types. *Sibirsk. Mat. Zh.*, 3:729–743, 1962.

[27] Kenneth L. McMillan. An interpolating theorem prover. *Theoretical Computer Science*, 345:101–121, 2005.

[28] K.L. McMillan. Interpolation and SAT-based model checking. In *CAV 03: Computer-Aided Verifi cation*, LNCS 2725, pages 1–13. Springer, 2003.

[29] S. McPeak and G.C. Necula. Data structure specifi cations via local equality axioms. In *CAV 05: Computer-Aided Verifi cation*, LNCS 3576, pages 476–490. Springer, 2005.

[30] G. Nelson. Techniques for program verifi cation. Technical Report CSL81-10, Xerox Palo Alto Research Center, 1981.

[31] P.W. O'Hearn, H. Yang, and J.C. Reynolds. Separation and information hiding. In *POPL 04*. ACM, 2004.

[32] Derek C. Oppen. Reasoning about recursively defi ned data structures.

*Journal of the ACM*, 27(3):403–411, 1980.

[33] M. Parkinson and G. Bierman. Separation logic and abstraction. In *POPL 05*. ACM, 2005.

[34] D.L. Parnas. The secret history of information hiding. In *Software pioneers: contributions to software engineering*. Springer, 2002.

[35] Mojzesz Presburger. Über die Vollständigkeit eines gewissen Systems der Arithmetik ganzer Zahlen, in welchen die Addition als einzige Operation hervortritt. In *Comptes Rendus du Premier Congrès des Mathématicienes des Pays Slaves*, pages 92–101, 1929.

[36] Shmuel Sagiv, Thomas W. Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Trans. Program. Lang. Syst.*, 24(3):217–298, 2002.

[37] A. Stump, C.W. Barrett, and D.L. Dill. Cvc: A cooperating validity checker. In *CAV 02: Computer-Aided Verifi cation*, LNCS 2404, pages 500–504. Springer, 2002.

[38] Aaron Stump, Clark W. Barret, David L. Dill, and Jeremy Levitt. A decision procedure for an extensional theory of arrays. In *Sixteenth Annual IEEE Symposium on Logic in Computer Science*, pages 29–37. IEEE Computer Society, 2001.

[39] G. Takeuti. *Proof Theory*. North-Holland, 1987.

[40] Alfred Tarski. *A Decision Method for Elementary Algebra and Geometry*. University of California Press, 1951.

[41] Alan M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42:230–265, 1936.

[42] Volker Weispfenning. The complexity of linear problems in fi elds. *Journal of Symbolic Computation*, 5(1/2):3–27, 1988.

[43] Greta Yorsh and Madanlal Musuvathi. A combination method for generating interpolants. In Robert Nieuwenhuis, editor, *Automated Deduction - CADE-20*, volume 3632 of *Lecture Notes in Computer Science*, pages 353–368. Springer, 2005.

[44] Calogero G. Zarba. Combining multisets with integers. In Andrei Voronkov, editor, *Automated Deduction – CADE-18*, volume 2392 of *Lecture Notes in Computer Science*, pages 363–376. Springer, 2002.