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 a lack of algorithms for computing interpolants for data structures used in software programs. We present efficient procedures to construct interpolants for the theories of arrays, sets, and multisets 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 proving properties of programs that manipulate these data structures.

Categories and Subject Descriptors: D.2.4 [Software Engineering]: Software/Program Verification.

General Terms: Languages, Verification, Reliability.

Keywords: Interpolation, data structure verification, CEGAR.

1. INTRODUCTION

Counterexample-guided abstraction refinement (CEGAR) [5, 2, 15] with interpolation based abstraction refinement [14, 17, 26, 18] has recently received a lot of attention as a robust technique for abstract static analysis of systems. In CEGAR, one attempts to prove a safety property starting with a crude abstraction on system states. If a counterexample is discovered in an attempt to do a proof, it is checked if the counterexample is indeed realized in the system. If so, the method has found a bug. Otherwise, if the counterexample is *spurious*, that is, it arises because the abstraction is too crude, the abstraction is refined and the analysis is repeated using this refined abstraction. The refinement of a spurious counterexample uses an *interpolant*, that is, an over-approximation of the set of states reachable by executing a prefix of the counterexample (i) which can be described using the variables live at the end of the prefix, and (ii) which is enough to determine the infeasibility of executing the suffix of the counterexample.

Similarly, in hardware verification, interpolants have been used [25] as a substitute for the expensive post-image computation to construct invariants: one finds a set (an interpolant) between the set of states S_k reachable in k-steps (constructed using bounded model checking) and the set of error states E that can be expressed as a formula over the common variables between S_k and E. This 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, where the logic is first-order, together with certain interpreted theories.

Formally, an interpolant ψ between two formulas α and β whose conjunction is unsatisfiable is a formula over the common variables of α and β that is implied by α and whose conjunction with β is unsatisfiable. Most applications of interpolants have so far been restricted to propositional logic, and the theories of equality with uninterpreted functions together with linear arithmetic [25, 26, 14, 17, 18]. 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 [2, 15] and low level state machine properties such as correct usage of locks or files [15]. In particular, model checking programs that use data structures and properties that depend on the correct use of data structures has not been possible.

Until recently, the automatic model checking of imperative programs manipulating data structures has mainly been limited to correctness properties of the *implementation* of data structures [33, 27], 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. In this paper, we concentrate on the other half, automatically checking properties of applications that *use* these data structures, *given* a correct implementation of the data structure. In practice, programmers use well-tested (and reasonably bug-free) library implementations of common data structures such as lists, sets, or multisets (e.g., from C++ STL or Java system classes), so checking client programs assuming correct implementations of the libraries is an important problem.

*This research was funded in part by the NSF grants CCR-0113611, CCR-0098114, CCR-0203051, CCF-0427202 and CNS-0541606.

SIGSOFT'06/FSE-14, November 5–11, 2006, Portland, Oregon, USA.

Using module-level abstractions is a fundamental software engineering principle to handle complexity [31], and we decompose our correctness proofs at module boundaries. This is not a new idea: Hoare [16] suggested 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. We show how modular verification and counterexample-guided abstraction refinement can be combined using interpolants for theories of data structures. Our technical contribution is to provide powerful abstraction capabilities provided by interpolants in order to make much of this reasoning automatic.

First, we show that (not necessarily quantifier-free) interpolants always *exist* for unsatisfiable formulas in any recursively enumerable theory and quantifier elimination in the theory is a necessary and sufficient condition for the existence of quantifier-free interpolants. Using this characterization, one immediately gets the existence of interpolants for many theories (such as (real and Presburger) arithmetic, arrays, lists, sets, and multisets). Together with quantifier elimination, this guarantees the existence of quantifier-free interpolants for linear arithmetic, lists, and sets with cardinality constraints (for arrays and multisets, this shows interpolants need not be quantifier-free).

Second, we provide a *reduction* technique to efficiently *compute* interpolants in many theories of practical interest that can use existing interpolating theorem provers as black boxes. In particular, using the reduction approach for compiling various quantifier-free theories for complex data structures to the combination theory of equality with uninterpreted functions and the theory of linear arithmetic proposed in [19], we show that from an interpolant in the reduced theory, one can construct an interpolant in the original theory. This is particularly attractive since very efficient implementations of these combined theories exist [7, 35, 20] and are already interpolant producing [26]. In our experience, developing efficient implementation and integration of new theories into decision procedures (including carefully tuned heuristics) is an nontrivial and difficult 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 practical interpolating decision procedures whenever such reductions exist. In particular, we get practical interpolating decision procedures for the quantifier-free theories of arrays, sets, and multisets.

We have implemented such interpolating decision procedures for the quantifier-free theories of arrays, sets, and multisets on top of the Foci interpolating decision procedure for linear arithmetic with uninterpreted functions [26]. This was used in the software model checker Blast [15]. By using more general reasoning about data structures, Blast is able to prove interesting properties of programs that manipulate data structures.

The organization of the paper reflects our attempts both to informally review how our results can be used in CEGAR-based software verification (Section 2) and to rigorously provide the technical details of our interpolant construction (Sections 3,4,5). Section 2 provides an informal overview of software verification based on counterexample-guided abstraction refinement with an emphasis on the role of interpolants for refinement and the reduction technique. In Section 3, we formalize the notion of reductions of a (more complex) theory to another simpler theory. Reductions are applied in Section 4 to obtain interpolants for several theories of practical interest, which can be implemented on top of existing interpolating theorem provers (Section 5).

Other Related Work. There has been a lot of work on modular construction and verification of software since the early papers [31] and [16], and the fundamental studies in abstract data types [23, 13]. Recent attempts for automatic modular verification of software using verification condition generation, explicit pre- and post-conditions, and decision procedures include [10, 22]. Our work is similar in spirit to these efforts. 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 [5, 2, 15, 3]. The paper [3] performs modular verification, but does not handle data structures.

Orthogonal work in separation logic also attempts system verification in the presence of data structures [28, 30], however, the notion of automatic abstractions has so far been less central. Work on shape analysis and related techniques [33, 27] complement our work by proving that data structure implementations are correct w.r.t. abstract specifications.

2. MOTIVATING EXAMPLE

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 `assume[p]`, that represent a boolean condition `p` that must be true for the operation to be executed, and (3) function calls $\mathtt{y} := f(\overline{\mathtt{x}})$ that call a function f with the actual arguments $\overline{\mathtt{x}}$ and writes the return value into `y`. We assume that control flow is represented explicitly, e.g., through a control flow graph.

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

```
   Example() {
01   x := emptyset();
02   while (*) {
03        t := *;
04        if(t->tag==0)
05            x := add(x,t);
06   }
07   q := choose(x);
08   if (q≠0 && q->tag != 0)
09        error();
     }
```

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

Figure 1: (A) Simple client (B) set signature

menting 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 functions. The client `Example` and the set library together form 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* (ℓ, s) of the program consists of a program location ℓ 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 (ℓ, 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 (ℓ, s). A program location ℓ is reachable if some state (ℓ, s) is reachable. For a closed program $(\mathsf{Lib}, \mathsf{Client})$ and a location ℓ of Client, the *safety verification problem* asks if ℓ is not reachable in the program $(\mathsf{Lib}, \mathsf{Client})$. We say $(\mathsf{Lib}, \mathsf{Client})$ *satisfies the safety property* associated with ℓ, written $\mathsf{Lib} \| \mathsf{Client} \models \ell$, if ℓ 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 ℓ.

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 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 $\mathtt{y} \rightarrow tag = 0$, 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 ℓ, 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 (ℓ, s) is in this set. If not, then ℓ is not reachable. However, if ℓ 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 [5, 2, 15] automatically find either (a) a concrete program execution to ℓ or (b) a new and more precise over-approximation of the set of reachable states; this process is repeated until either the location ℓ is proved to be unreachable, or a concrete counterexample trace to ℓ is obtained.

There are two pragmatic issues, however, that arise in safety verification problem. First, when both the client and the library are large programs, the construction of the reachable set of states 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 [15] 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 Lib (i.e., a program with at least as many behaviors as Lib), and separately check that (1) the closed program (A, Client) satisfies the safety property associated with ℓ and (2) A is indeed an abstraction of Lib. We use *abstract datatype definitions* (ADTs) as abstractions of a library Lib.

An ADT $A = (T, \mu)$ for a library Lib with signature Σ_{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 Σ_{Lib} as well as a map μ 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 $x_1, \ldots, x_n$, and y' of sorts $\sigma_1, \ldots, \sigma_n, \sigma$ 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 it entails $\psi(c_1, \ldots, c_n, d)$ as well. Formally, we use the proof rule [16]

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

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. Here, we focus on the verification problem

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

There are other orthogonal, approaches to prove the second obligation $\mathsf{Lib} \preceq \mathsf{A}$ [33, 10].

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:

$$\begin{array}{ll} \mathtt{y} := \mathtt{emptyset}() & \mathtt{y} = \emptyset \\ \mathtt{y} := \mathtt{add}(\mathtt{x}, \mathtt{t}) & \mathtt{y} = \mathtt{x} \cup \{\mathtt{t}\} \\ \mathtt{y} := \mathtt{choose}(\mathtt{x}) & (\mathtt{y} = 0 \wedge \mathtt{x} = \emptyset) \vee (\mathtt{y} \neq 0 \wedge \mathtt{y} \in \mathtt{x}) \end{array}$$

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 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 [14] 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 Client using a library Lib, an ADT A such that $\mathsf{Lib} \preceq A$, a set of predicates Π_0 over the program state, and a location ℓ of the program. It returns "reachable" if some execution reaches the location ℓ, and "safe" otherwise. The algorithm maintains a *current abstraction* Π, 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 Reach that constructs an over-approximation of the reachable states using the current abstraction. This phase constructs a tree $\mathcal{G}$ 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 φ over the predicates in the current abstraction. The formula φ 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 [11, 15].

The second (**Step 2**) checks if the location ℓ 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 ℓ is reachable in the tree, the path to ℓ may (a) either represent a real bug, (b) or be a *spurious counterexample* in that ℓ 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 ℓ 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.

Algorithm 1 Safety Verification

Input: client program Client using ADT A
Input: initial abstraction Π_0, program location ℓ
Output: "reachable" if location ℓ is reachable,
Output: "safe" otherwise

$\Pi := \Pi_0$
Step 1: $\mathcal{G} := \mathsf{Reach}(\mathsf{Client}, A, \Pi)$
Step 2:
if ℓ is unreachable in $\mathcal{G}$ **then**
  return "safe"
else
  pick an abstract trace t from $\mathcal{G}$ that reaches ℓ
  if t can be concretely simulated **then**
    return "reachable"
  else
    Step 3: $\Pi := \Pi \cup \mathsf{Refine}(t)$; **goto Step 1:**
  end if
end if

In case Step 2 finds the current path is spurious, a refinement step (**Step 3**) is used to *refine* the current abstraction by adding new predicates derived from analyzing t.

Theorem 4. *[5, 2, 15] If Algorithm 1 returns "safe" for a client program* Client*, an ADT* A*, a set of initial predicates* Π_0*, and a location* ℓ*, then* $\mathsf{Client}\|A \models \ell$*. If it returns "reachable" then* ℓ *is reachable in* $\mathsf{Client}\|A$*.* ▫

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

2.4 Forward Search

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 condition* computation, that takes a formula φ over the current abstraction and a program operation $\mathtt{op}$, and produces a new formula that represents a superset of the set of states that can be reached from the states in φ by executing the $\mathtt{op}$.

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., a variable $\mathtt{x}$ is renamed $\mathtt{x}'$). Intuitively, s denotes the valuation at the "current state" and s' denotes the valuation at the "next state." For every operation $\mathtt{op}$, 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 (primed) variables in the next state after the operation $\mathtt{op}$ 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 Π be a set of predicates over the program variables V. We write Π' to denote the set where each predicate in Π is primed. For any $\mathtt{op}$, the *predicate abstraction* of the

[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, e.g., using *ownership* type systems [4] that ensure data is not modified while inside a data structure.

transition relation $T(\texttt{op}, V, V')$, written $T_\Pi(\texttt{op}, V, V')$, is the smallest boolean formula (in the implication order) over the atomic predicates $\Pi \cup \Pi'$ that contains $T(\texttt{op}, V, V')$. The computation of T_Π makes calls to a decision procedure for the theory over which the predicates in Π are interpreted [11]. In a later subsection below, we informally discuss the reduction approach for generating decision procedures for quantifier-free theories over data structures including sets, arrays, and multisets.

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

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 (using the abstract post computation) each outgoing operation of every reached location [11, 15]. 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

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

The location 09 is abstractly reachable. This is expected, since we are not tracking the state $q \rightarrow \mathit{tag}$ of the elements in the set.

2.5 Refinement

If the forward search tree contains a path t from the root node to a node with label ℓ, 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 [14]. 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 [14].

To check the satisfiability of a trace formula which expresses a property of operations of a complex data structure, the reduction approach is used to reduce the formula to an equisatisfiable formula in the theory of uninterpreted symbols (and Presburger arithmetic in certain cases). This is followed by a decision procedure call to check if the reduced 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. And, this is done using *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 ψ_i between φ^- and φ^+ (see the definition in the next section) has the property that (1) φ^- implies ψ_i, that is, the predicate ψ_i holds after the prefix of the trace up to i is executed, (2) $\psi_i \wedge \varphi^+$ is unsatisfiable, that is, the predicate ψ_i at location i is enough to show infeasibility of the suffix of the trace, and (3) ψ_i is over the common variables between φ^- and $\varphi+$, that is, over the *live* variables. If at each point i, we then track the predicate ψ_i (where we rename variables back to their original names), then we have enough information to rule out the trace.[2] In this way, the refinement step reduces to interpolant computation.

2.6 Reduction and Interpolation

We now give an informal overview of how interpolants for theories over complex data structures including arrays, sets, multisets, etc., can be computed. A more technical discussion is given in Section 3.

Reduction. Abstract post condition 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. So if a program uses container data structures including sets (as in the example above), multisets, arrays, lists, etc., formulas expressing their properties will involves operations on these data structures. When checking programs using an ADT (T, μ), one has to additionally implement a decision procedure for the theory T. Engineering a fast and scalable decision procedure is a difficult and nontrivial task, in general. For the theory of equality with uninterpreted functions and the theory of Presburger arithmetic, there are fast implementations [7, 35] that are used by software model checkers such as SLAM, Blast, or Magic to discharge the satisfiability checks.

In [19], we proposed a *reduction* approach, which compiles a query made in the theory T to an equisatisfiable query made in the theory of equality (and Presburger 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

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

in which $\in$ is treated as an uninterpreted predicate symbol. It is easy to see that the above formula is valid in the

[2] One technical point: as shown in [14], the interpolants must be generated from the same proof of unsatisfiability.

theory of equality with uninterpreted functions (by instantiating the quantifier with q) and so by the correctness of the reduction, the original formula is valid in the theory of sets.

The main idea of the reduction approach is to define the operations on a data structure using a small subset of operations, the equality predicate, and in certain cases, operations such as $+$ and $\leq$ on numbers. This small set of operations are then treated as uninterpreted symbols. The reduction has the property that it preserves satisfiability, i.e., a formula in the original theory is equisatisfiable with the reduced formula in the theory of uninterpreted symbols (and Presburger arithmetic). For example, any formula in the theory of arrays can be reduced to an equisatisfiable formula expressed only using the read operation; similarly, any formula in the theory of sets can be reduced to an equisatisfiable formula expressed using the $\in$ predicate. In the case of multisets, in addition to $\in$, we need a function count and addition on natural numbers. While the theories of arrays and sets can be reduced to the theory of uninterpreted symbols and equality, the theory of multisets is reduced to the combination of the theory of uninterpreted symbols with equality and Presburger arithmetic. In Section 4, we provide the formal compilation algorithms for the theory of arrays, sets, and multisets, which are commonly used data structures.

There is another technicality however. The predicates used in predicate abstraction need not be quantifier-free. For many theories, including the theories of arrays and multisets, 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 [7] 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.

Interpolation. Given a trace formula expressed in terms of operations on complex data structures such as sets (and other data structures), the reduction technique can be used for computing interpolants as well. Here, we give a brief informal overview of the approach.

Using the reduction approach, a trace formula is first reduced to an equisatisfiable formula in the theory of uninterpreted symbols with equality (and Presburger arithmetic). Algorithms for computing interpolants for the theory of equality with uninterpreted functions and arithmetic are known and efficient implementations exist; see [26] for details. An interpolant can thus be constructed for the reduced trace formula expressed in the theory of uninterpreted symbols with equality (and Presburger arithmetic) using these algorithms. The interpolant is then mapped back to the original theories (see Figure 4 for an outline). In Section 3, we prove that if a theory T reduces to another theory R whose signature is a subset of T, then an interpolating decision procedure for R can be used to compute an interpolant in T.

In the process, quantifiers may get introduced. If the theory admits quantifier elimination, it is then possible to get rid of these quantifiers and get quantifier-free interpolants. More often, though, the interpolants will have quantifiers.

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

Figure 2: Trace, trace formula, and interpolants

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. □

In Section 4, we prove using compactness that interpolants exist for any recursively axiomatizable theory. In general, such an interpolant may have quantifiers. If a theory admits quantifier-elimination, then a quantifier-free interpolant can be generated also. The reduction approach is then used to generate interpolating decision procedures for quantifier-free theories for complex data structures by compiling formulas in these theories to formulas in the theory of uninterpreted symbols with equality (and Presburger arithmetic if needed). In general, the interpolants so produced may have quantifiers. We show that this is inevitable: quantifier-free interpolants may not exist even if the original formulas are quantifier-free.

2.7 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.

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 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

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

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

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

This section is a rigorous technical presentation of interpolation and reduction, and it provides characterizations for the existence of interpolants. We prove two main results. The first (Theorem 8) shows that every recursively enumerable theory admits interpolation, although the interpolant may not be quantifier-free. A quantifier-free interpolant is guaranteed to exist 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 the 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 do not get quantifier-free interpolants.

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). A *constant* is a function of arity zero. For a signature Σ, we write Σ^S (respectively, Σ^F, Σ^P) for S (respectively F, P). For signatures Σ_1 and Σ_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 σ, we fix a set $\mathcal{X}_\sigma$ of free constant symbols of sort σ which are disjoint from the function symbols Σ^F. We also fix a set $\mathcal{X}_{\mathsf{bool}}$ of free propositional symbols.

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

The set of Σ-atoms is the smallest set such that (1) each propositional symbol $u \in \mathcal{X}_{\mathsf{bool}}$ is a Σ-atom, (2) $s \approx t$ is a Σ-atom if s and t are Σ-terms of the same sort, and (3) $p(t_1, \ldots, t_n)$ is a Σ-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 Σ-term of sort σ_i for $i = 1, \ldots, n$.

The set of quantifier-free Σ-formulas is the smallest set such that (1) each Σ-atom is a Σ-formula, (2) if φ, ψ, χ are Σ-formulas, so are $\neg\varphi$, $\varphi \wedge \psi$. The set of Σ-formulas is the smallest set such that (1) every quantifier-free Σ-formula is a Σ-formula, and (2) if φ is a Σ-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 Σ-formulas. We shall use the usual derived formulas $\varphi \vee \psi$, $\varphi \to \psi$, $\varphi \leftrightarrow \psi$. We use the shorthand $s \not\approx t$ for $\neg(s \approx t)$. We write $\mathit{vars}(\varphi)$ for the free constant symbols in φ. We omit the prefix Σ- 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 Σ-structure $\mathcal{A}$ over X is a map which interprets

1. each sort $\sigma \in S$ as a non-empty domain A_σ,
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 $\{\mathit{true}, \mathit{false}\}$,
4. each function symbol $f \in F$ of arity $\sigma_1 \times \ldots \times \sigma_n \to \sigma$ as a function $f^\mathcal{A} : A_{\sigma_1} \times \ldots \times A_{\sigma_n} \to A_\sigma$,
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 Σ-formula φ with free constants $X_0 \subseteq X$, we denote by $\varphi^\mathcal{A}$, the evaluation of φ under $\mathcal{A}$ (defined in the usual way). For a formula φ, we write $\mathcal{A} \models \varphi$ if $\varphi^\mathcal{A} = \mathit{true}$. A formula φ is *satisfiable* if $\mathcal{A} \models \varphi$ for some structure $\mathcal{A}$ over $\mathit{vars}(\varphi)$.

3.2 Theories

A Σ-*theory* is a set of Σ-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 φ and ψ are T-equivalent for a theory T if $\varphi \leftrightarrow \psi$ is in T. If Σ is a signature, $T_\approx^\Sigma$ denotes the theory of equality over Σ, that is, $T_\approx^\Sigma$ is the set of all valid Σ-sentences.

Given a Σ-theory T, a T-model is a Σ-structure that satisfies all sentences in T. A Σ-formula φ over a set V of free constant symbols 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 Σ-theory T is the problem of deciding, for every Σ-formula φ, whether or not φ is T-satisfiable. The *quantifier-free satisfiability problem* of a Σ-theory T is the problem of deciding, for every quantifier-free Σ-formula φ, whether or not φ is T-satisfiable. For every signature Σ, the

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

satisfiability problem of $T_{\approx}^{\Sigma}$ is undecidable [38], whereas the quantifier-free satisfiability problem of $T_{\approx}^{\Sigma}$ is decidable [1].

A Σ-theory T *eliminates quantifiers* if for every Σ-formula φ it is possible to effectively compute a quantifier-free Σ-formula ψ such that φ and ψ are T-equivalent and $vars(\psi) \subseteq vars(\varphi)$. Examples of theories that eliminate quantifiers include the theory T_{int} of linear integer arithmetic [32], the theory T_{rat} of linear rational arithmetic [39], the theory T_{real} of real arithmetic [37], the theory T_{data} of recursively defined data structures [29], and the theory T_{set} of sets [21].

The theory of equality $T_{\approx}^{\Sigma}$ over Σ does not eliminate quantifiers. Assume by contradiction that $T_{\approx}^{\Sigma}$ eliminates quantifiers. Given any Σ-formula φ, a quantifier free Σ-formula ψ can be effectively computed such that φ and ψ are equivalent (hence, equisatisfiable). Since the quantifier-free satisfiability problem of $T_{\approx}^{\Sigma}$ is decidable, we can effectively decide whether φ is satisfiable. But this implies that the satisfiability problem of $T_{\approx}^{\Sigma}$ is decidable, a contradiction. Using similar arguments, we will prove in the next section that theories that do not eliminate quantifiers also include the theory T_{array} of arrays and the theory T_{bag} of multisets.

3.3 Interpolation

Let T be a Σ-theory, and let φ and ψ be Σ-formulas such that $\varphi \wedge \psi$ is T-unsatisfiable. A Σ-formula α is a *T-interpolant* of (φ, ψ) if the following three conditions hold:

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

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

For every signature Σ, the theory of equality $T_{\approx}^{\Sigma}$ over Σ is interpolating [6]. In particular, if φ and ψ are Σ-formulas such that $\varphi \wedge \psi$ is unsatisfiable, then a $T_{\approx}^{\Sigma}$-interpolant α of (φ, ψ) can be extracted from any first-order proof Π of the unsatisfiability of $\varphi \wedge \psi$ in time linear in the size of the proof Π. Methods for extracting $T_{\approx}^{\Sigma}$-interpolants from first-order proofs exist for Gentzen-like calculi [36], resolution, and tableaux [9]. 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 occurring in it are of the form $x \approx y$, $x \approx f(x_1, \ldots, x_n)$, or $p(x_1, \ldots, x_n)$, where $x, y, x_1, \ldots, x_n$ are variables. It is easy to see that every formula φ can be converted to an equivalent flat formula φ' (by introducing new variables). This flat formula φ' is called the *flat form* of φ. 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 φ' is the flat form of φ and ψ' the flat form of ψ.

Proposition 7. *Let $\varphi \wedge \psi$ be a quantifier-free T-unsatisfiable Σ-formula. A T-interpolant of (φ, ψ) can be computed from a flat form $\varphi' \wedge \psi'$ of $\varphi \wedge \psi$.* $\square$

We characterize (quantifier-free) T-interpolating theories. Our main result 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. $\square$

Proof. Let T be a recursively enumerable Σ-theory, and let φ and ψ be Σ-formulas such that $\varphi \wedge \psi$ is T-unsatisfiable. We want to effectively compute a T-interpolant α of (φ, ψ).

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 Π of the T_0-unsatisfiability of $\varphi \wedge \psi$. From Π, one can effectively extract a $T_{\approx}^{\Sigma}$-interpolant α 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, α is also a T-interpolant of (φ, ψ).

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

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

From Theorem 8, it follows that examples of theories that are interpolating, include the theory T_{int} of integer linear arithmetic, the theory T_{rat} of rational linear arithmetic, the theory T_{real} of real arithmetic, the theory T_{data} of recursively defined data structures, the theory T_{array} of arrays, the theory T_{set} of sets, and the theory T_{bag} of multisets. Further, except for the theory T_{array} of arrays and the theory T_{bag} of multisets, all other theories listed above are quantifier-free interpolating.

Example 9. Arithmetic. The theory T_{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_{int} is recursively enumerable and eliminates quantifiers [32]. Consequently, by Theorem 8, T_{int} is quantifier-free interpolating.

The theory T_{rat} of *rational linear arithmetic* is the theory of the structure of rational numbers $\langle \mathbb{Q}, 0, 1, +, \leq \rangle$. The theory T_{rat} is recursively enumerable, and eliminates quantifiers [39]. Consequently, by Theorem 8, T_{rat} is quantifier-free interpolating.

The theory T_{real} of *real arithmetic* is the theory of the structure of real numbers $\langle \mathbb{R}, 0, 1, +, \times, \leq \rangle$. The theory T_{real} is recursively enumerable, and eliminates quantifiers [37]. Consequently T_{real} is quantifier-free interpolating. $\square$

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

$$\mathsf{car}(\mathsf{cons}(x, y)) = x\,, \qquad \mathsf{cdr}(\mathsf{cons}(x, y)) = y\,,$$
$$\mathsf{cons}(\mathsf{car}(x), \mathsf{cdr}(x)) = x\,, \qquad x \not\approx t(x)\,,$$

where t is a term built up form x by using finitely many applications of the unary function symbols car and cdr. The theory T_{data} is recursively enumerable, and eliminates quantifiers [24]. Consequently, by Theorem 8, T_{data} is quantifier-free interpolating. $\square$

3.4 Reduction of Interpolants

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 [19], 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 [26].

Let T be a Σ-theory, and let R be an Ω-theory such that $\Omega \subseteq \Sigma$, $\Omega^{\mathrm{S}} = \Sigma^{\mathrm{S}}$, and $R \subseteq T$. As defined in [19], T *reduces* to R if there is a computable map from flat Σ-atoms to Ω-formulae such that, if we apply this map to a quantifier-free flat Σ-formula φ, obtaining an Ω-formula φ^*, then

1. φ and φ^* are T-equivalent.
2. If φ^* is R-satisfiable then φ is T-satisfiable.

It is shown in [19] that the quantifier-free theories of arrays, lists, sets and multisets can be reduced to quantifier-free theories of uninterpreted symbols with equality, constructors and Presburger arithmetic. Some of these reductions will be reviewed in Section 4; for details, an interested reader should consult [19]. The theorem below shows how the reduction approach can also be used to generate interpolants using interpolating decision procedures for theories of uninterpreted symbols and Presburger arithmetic.

Theorem 11. *Let T be a Σ-theory, and let R be an Ω-theory such that $\Omega \subseteq \Sigma$ and $\Omega^{\mathrm{S}} = \Sigma^{\mathrm{S}}$, and $R \subseteq T$. If*
(i) T reduces to R, and
(ii) It is possible to compute an R-interpolant of (φ, ψ) whenever $\varphi \wedge \psi$ is an R-unsatisfiable Ω-formula,
then it is possible to compute a T-interpolant of (φ, ψ) whenever $\varphi \wedge \psi$ is a T-unsatisfiable quantifier-free Σ-formula. $\square$

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

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 Σ-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. $\blacksquare$

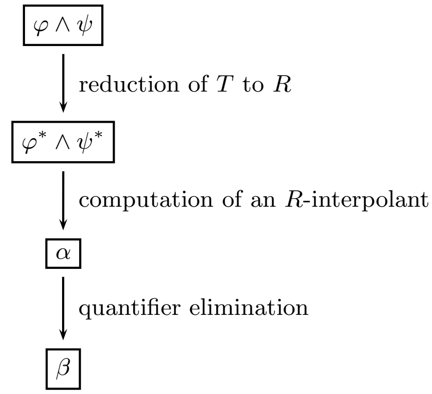

Figure 4: Computing quantifier-free T-interpolants when T reduces to R, and either T or R eliminates quantifiers. $\varphi \wedge \psi$ is a T-unsatisfiable flat quantifier-free Σ-formula. $\varphi^* \wedge \psi^*$ is an Ω-formula obtained from $\varphi \wedge \psi$ by a reduction function. α is an R-interpolant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

Note that in Theorem 11, if 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 β of (φ, ψ). This process is depicted in Figure 4.

4. INTERPOLANTS VIA REDUCTION

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

4.1 Arrays

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

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

The theory T_{array} is recursively enumerable. Consequently, by Theorem 8, T_{array} is interpolating. T_{array} does have a decidable quantifier-free satisfiability problem [34]. Below we show that: (a) the satisfiability problem of T_{array} is undecidable, (b) the theory T_{array} does not eliminate quantifiers, and consequently, (c) the theory T_{array} is not quantifier-free interpolating.

Theorem 12. *1. The satisfiability problem of T_{array} is undecidable.*
2. T_{array} does not eliminate quantifiers. $\square$

Proof (*Sketch*). Consider the theory T^2_{array} which is the same as T_{array}, but the sort elem and index are identified. Clearly, the satisfiability problem of T^2_{array} is undecidable [12]. We want to reduce the satisfiability problem of T^2_{array} to the satisfiability problem of T_{array}. This can be done as follows. Specify 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 $\mathsf{read}(h, i) \approx e_2 \wedge \mathsf{read}(f, i) \approx e_1$.

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

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

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

In fact, even if φ and ψ are quantifier-free, their interpolant may require quantification. Consider $h' \approx \mathsf{write}(h, i, e)$ and $(a \neq b) \wedge (\mathsf{read}(h, a) \not\approx \mathsf{read}(h', a)) \wedge (\mathsf{read}(h, b) \not\approx \mathsf{read}(h', b))$ whose conjunction is unsatisfiable, but there is no quantifier-free interpolant over the common variables h and h'. The interpolant for this case is: $\exists j((\mathsf{read}(h, j) \not\approx \mathsf{read}(h', j)) \wedge \forall k(k \neq j \implies \mathsf{read}(h, k) \approx \mathsf{read}(h', k)))$.

It is shown in [19] that T_{array} reduces to $T_{\approx}^{\Omega}$, where $\Omega^{\mathrm{S}} = \{\mathsf{elem}, \mathsf{index}, \mathsf{array}\}$, $\Omega^{\mathrm{F}} = \{\mathsf{read}\}$, and $\Omega^{\mathrm{P}} = \emptyset$. 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 $\mathsf{read}(a, i) \approx e \wedge (\forall_{\mathsf{index}}\ j)(j \not\approx i \to \mathsf{read}(a, j) \approx \mathsf{read}(b, j))$. Literals not mentioned are left unchanged. Consequently, by Theorem 11, and provided that $\varphi \wedge \psi$ is quantifier-free, we can reduce the problem of computing T_{array}-interpolants[4] of (φ, ψ) to the problem of computing $T_{\approx}^{\Omega}$-interpolants.

4.2 Sets

The theory T_{set} of *sets with finite cardinality constraints* has a signature Σ_{set} containing one sort elem for elements and one sort set for sets of elements. The operations allowed include the constant symbols $\emptyset$ (empty set) and $\mathbb{1}$ (full set)[5], 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 $\mathsf{elem} \to \mathsf{set}$, and the binary predicate symbol $\in$, of arity $\mathsf{elem} \times \mathsf{set}$, with the standard semantics. In addition, for each natural number k, the unary predicate symbols $|\cdot| \geq k$ and $|\cdot| \approx k$, both of arity set. The element domain is assumed to be finite. The theory T_{set} is the set of all Σ_{set}-sentences that are true in all standard set-structures.

The theory T_{set} is recursively enumerable, and eliminates quantifiers [21, Fact 1, page 7]. Consequently, by Theorem 8, T_{set} is quantifier-free interpolating. Further, it has a decidable satisfiability problem [21].

As shown in [19], T_{set} reduces to $T_{\approx}^{\Omega}$, where $\Omega^{\mathrm{S}} = \{\mathsf{elem}, \mathsf{set}\}$, $\Omega^{\mathrm{F}} = \emptyset$, and $\Omega^{\mathrm{P}} = \{\in\}$. Figure 5 shows the reduction function from flat Σ_{set}-literals to Ω-formulas (literals not mentioned are left unchanged). Consequently, by Theorem 11, and provided that $\varphi \wedge \psi$ is quantifier-free, we can reduce the problem of computing quantifier-free T_{set}-interpolants of (φ, ψ) to the problem of computing $T_{\approx}^{\Omega}$-interpolants.

Example 14. Let $\varphi\ :\ x \approx \{a\}$, and $\psi\ :\ x \approx \{b, c\} \wedge b \not\approx c$. After performing the reduction, we get

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

A Gentzen-style proof yields the interpolant

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

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

$$|x| \approx 0 \vee |x| \approx 1\,. \qquad \square$$

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

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

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_{bag} of *multisets* has a signature Σ_{bag} extending Σ_{int} with a sort elem for elements, and a sort bag for multisets, plus the following symbols: the constant symbol $[\![\,]\!]$, of sort bag; the function symbols: $[\![\cdot]\!]^{(\cdot)}$, of sort $\mathsf{elem} \times \mathsf{int} \to \mathsf{bag}$; $\sqcup$, $\uplus$, and $\sqcap$, of sort $\mathsf{bag} \times \mathsf{bag} \to \mathsf{bag}$; count, of sort $\mathsf{elem} \times \mathsf{bag} \to \mathsf{int}$. The theory T_{bag} is the set of all Σ_{bag}-sentences that are true in all standard bag-structures.

The theory T_{bag} is recursively enumerable. Consequently, by Theorem 8, T_{bag} is interpolating. Even though the quantifier-free satisfiability problem of T_{bag} is decidable [40], we show that: (a) the satisfiability problem of the full T_{bag} is undecidable, (b) the theory T_{bag} does not eliminate quantifiers, and (c) the theory T_{bag} is not quantifier-free interpolating.

Theorem 15. *1. The satisfiability problem of T_{bag} is undecidable.*
2. T_{bag} does not eliminate quantifiers. □

PROOF (*Sketch*). Consider the theory T_{bag}^2 which is the same as T_{bag}, but the sort elem and int are identified. Clearly, the satisfiability problem of T_{bag}^2 is undecidable [8]. To reduce the satisfiability problem of T_{bag}^2 to the satisfiability problem of T_{bag}, specify that a function $h : \mathsf{elem} \to \mathsf{int}$ is bijective with the formulas

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

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_{bag} eliminates quantifiers, and let φ be a Σ_{bag}-formula. Then it is possible

[4]Since T_{array} is not quantifier-free interpolating, these interpolants are, in general, not quantifier-free.
[5]introduced for technical reasons; see [41] for details.

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

Figure 5: Reduction function for sets

to effectively compute a quantifier free Σ_{bag}-formula ψ such that φ and ψ are T_{bag}-equivalent. It follows that φ and ψ are T_{bag}-equisatisfiable. Since the quantifier-free satisfiability problem of T_{bag} is decidable, we can effectively decide whether φ is T_{bag}-satisfiable. But this implies that the satisfiability problem of T_{bag} is decidable, a contradiction. ■

Corollary 16. The theory T_{bag} of multisets is not quantifier-free interpolating. □

In fact, even if φ and ψ 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, and whose interpolant is

$$\exists x.\mathsf{count}(x, h) \approx 1 \wedge \forall y.(y \not\approx x \Rightarrow \mathsf{count}(y, h) \approx 0)$$

However, there is no quantifier-free interpolant over the common variables h and h'.

In [19], it is proved that T_{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 $\Omega^{\mathrm{P}} = \emptyset$. Figure 6 shows the reduction function from flat Σ_{array}-literals to $(\Sigma_{\mathsf{int}} \cup \Omega)$-formulas (literals not mentioned are left unchanged).

Consequently, by Theorem 11, and provided that $\varphi \wedge \psi$ is quantifier-free, we can reduce the problem of computing T_{bag}-interpolants of (φ, ψ) to the problem of computing $T_{\mathsf{int}} \cup T_{\approx}^{\Omega}$-interpolants. Since T_{bag} is not quantifier-free interpolating, these interpolants are, in general, not quantifier-free.

5. IMPLEMENTATION USING FOCI

Foci [26] is an implementation of an interpolating decision procedure for the quantifier-free theory of equality and arithmetic. We now sketch how Foci can be used to implement our reduction procedure for generating interpolants for data structures even though the reduced formulas can have quantifiers. Let $\varphi^* \wedge \psi^*$ be an R-formula obtained by the reduction algorithms presented above of a quantifier-free 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. This works since in all three reductions above, the existential quantifiers are not in the scope of any universal quantifiers. In the resulting formula (that may contain universal quantifiers), 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 [9].

6. 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 over complex data structures of interest in program verification. Further, the reduction approach allows quick implementations by allowing hooking into already-existing efficient implementations of interpolating decision procedures, such as Foci.

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

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

Figure 6: Reduction from T_{bag}

generalizing from particular counterexamples as an interesting open problem. On the practical front, we are applying the more powerful predicate generation capabilities to prove larger programs using the Blast software model checker.

Acknowledgements. We thank Shriram Krishnamurthi and Corina Pasareanu for many detailed comments.

7. REFERENCES

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

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

[3] S. Chaki, E.M. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. *IEEE TSE*, 30(6):388–402, 2004.

[4] D.G. Clarke, J. Noble, and J.M. Potter. Simple ownership types for object containment. In *ECOOP 01*, pages 53–76.

[5] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *CAV 00*, LNCS 1855, pages 154–169. Springer, 2000.

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

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

[8] P.J. Downey. Undeciability of Presburger arithmetic with a single monadic predicate letter. Technical Report 18-72, Havard University, 1972.

[9] M.C. Fitting. *First-Order Logic and Automated Theorem Proving*. Springer, 1996.

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

[11] S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *CAV 97*, LNCS 1254, pages 72–83. Springer, 1997.

[12] Y. Gurevich. The decision problem for standard classes. *J. Symbolic Logic*, 41(2), 1976.

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

[14] T.A. Henzinger, R. Jhala, R. Majumdar, and K.L. McMillan. Abstractions from proofs. In *POPL 04*, pages 232–244. ACM, 2004.

[15] T.A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL 02*, pages 58–70. ACM, 2002.

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

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

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

[19] D. Kapur and C.G. Zarba. A reduction approach to decision procedures. 2005.

[20] D. Kapur and H. Zhang. An overview of rewrite rule laboratory (RRL). *J. Computer and Mathematics with applications*, 14(2):91–114, 1995.

[21] V. Kuncak and M. Rinard. The first-order theory of sets with cardinality constraints is decidable. Technical Report CSAIL 958, MIT, 2004.

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

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

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

[25] K.L. McMillan. Interpolation and SAT-based model checking. In *CAV 03*, LNCS 2725, pages 1–13. Springer, 2003.

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

[27] S. McPeak and G.C. Necula. Data structure specifications via local equality axioms. In *CAV 05*, LNCS 3576, pages 476–490. Springer, 2005.

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

[29] D.C. Oppen. Reasoning about recursively defined data structures. *J. ACM*, 27(3):403–411, 1980.

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

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

[32] M. 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.

[33] S. Sagiv, T.W. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM TOPLAS*, 24(3):217–298, 2002.

[34] A. Stump, C.W. Barret, D.L. Dill, and J. Levitt. A decision procedure for an extensional theory of arrays. In *LICS 01*, pages 29–37. IEEE, 2001.

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

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

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

[38] A.M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proc. London Math. Society*, 42:230–265, 1936.

[39] V. Weispfenning. The complexity of linear problems in fields. *J. Symbolic Computation*, 5(1/2):3–27, 1988.

[40] C.G. Zarba. Combining multisets with integers. In *CADE 02*, LNCS 2392, pages 363–376. Springer, 2002.

[41] C.G. Zarba. A quantifier elimination algorithm for a fragment of set theory involving the cardinality operator. In *International Workshop on Unification*, 2004.