Data Structure Specifications via Local Equality Axioms

Scott McPeak George C. Necula

University of California, Berkeley
{smcpeak,necula}@cs.berkeley.edu

Abstract. We describe a program verification methodology for specifying global shape properties of data structures by means of axioms involving arbitrary predicates on scalar fields and pointer equalities in the neighborhood of a memory cell. We show that such local invariants are both natural and sufficient for describing a large class of data structures. We describe a complete decision procedure for such a class of axioms. The decision procedure is not only simpler and faster than in other similar systems, but has the advantage that it can be extended easily with reasoning for any decidable theory of scalar fields.

1 Introduction

This paper explores a program verification strategy where programs are annotated with invariants, and decision procedures are used to prove them. A key element of such an approach is the specification language, which must precisely capture shape and alias information but also be amenable to automatic reasoning. Type systems and alias analyses are often too imprecise. There are very expressive specification languages (e.g., reachability predicates [14], shape types [3]) with either negative or unknown decidability results. A few systems such as TVLA [16] and PALE [12] have similar expressivity and effectiveness, but use logics with *transitive closure* and thus incur additional restrictions.

Our approach is based on the idea that one can specify shape by specifying *local* pointer equalities, such as "for every list node n, n.`next`.`prev` $= n$", which specifies a doubly-linked list. This simple idea generalizes to describe a wide variety of shapes, as subsequent examples will show. And, as each specification constrains only a bounded fragment of the heap around a distinguished element n (unlike with transitive closure), it is fairly easy to reason about.

There are two main contributions of this work. First, we present a *methodology* for specifying shapes of data structures, using local specifications. The specifications use arbitrary predicates on scalar fields and equality between pointer expressions to constrain the shape of the data structure. We show that such local specifications can express indirectly a number of important global properties.

The second contribution is a *decision procedure* for a class of local shape specifications as described above. The decision procedure is based on the idea that local shape specifications have the property that any counterexamples are also local. This decision procedure is not only simple to implement but fits naturally in a cooperating decision procedure framework that integrates pointer-shape reasoning with reasoning about scalar values, such as linear arithmetic, or uninterpreted functions.

A related contribution is to the field of automated deduction, for dealing with universally quantified assumptions: the matching problem is that of finding sufficient instantiations of universally quantified facts to prove a goal. Performing too few instantiations endangers completeness while performing too many compromises the performance of the algorithm and often even its termination. For the class of universally quantified axioms that we consider here we show a complete and terminating matching rule. This is a valuable result in a field where heuristics are the norm [13, 1].

Our experimental results are encouraging. We show that we can describe the same data structures that are discussed in the PALE publications, with somewhat better performance results; we can also encode some data structures that are not expressible using PALE. We also show that the matching rules are not only complete, but lead to a factor of two improvement in performance over the heuristics used by Simplify [1], a mature automatic theorem prover. Furthermore, unlike matching in Simplify, our algorithm will always terminate.

2 Methodology Example

We follow a standard program verification strategy, with programmer-specified invariants for each loop and for the function start (precondition) and end (postcondition). We use a symbolic verification condition generator to extract a verification condition for each path connecting two invariants. The emphasis in this paper is on the specification mechanism for the invariants and the decision procedure for proving the resulting verification conditions.

Suppose we wish to verify the procedure in Figure 1, part of a hypothetical process scheduler written in a Java-like language. It has the job of removing a process from the list of runnable processes, in preparation for putting it into the list of sleeping processes. Each list is doubly-linked.

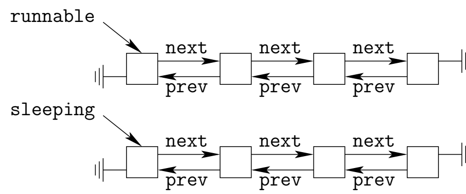
We capture the data structure invariants using the set of axioms in Figure 1, where the quantifier ranges over list cells. These axioms constitute the data structure invariant, and must hold at the start and end of the function. Axioms A1 and A2 express the local invariant that the `next` and `prev` fields are inverses. Axiom A3 says that all the processes in a list have the same state (`RUN` or `SLP`), and axiom A4 says that all the runnable processes have non-increasing priorities.

To begin verifying `insert`, we need a formal precondition:

$$PRE: \quad \mathtt{x} \neq \mathtt{null} \ \wedge \ \mathtt{x.prev} \neq \mathtt{null} \ \wedge \ \mathtt{x.state} = \mathtt{RUN}$$

The verification condition generator produces verification conditions, which are implications where the left-hand side consists of the function precondition along with the current path predicates, and the right-hand side is the goal to prove. We use the standard strategy to show validity of the verification condition by

```
1 // precondition: x is runnable (RUN) and not first in list
2 void remove(Process x) {
3   x.prev.next = x.next;
4   if (x.next)
5     x.next.prev = x.prev;
6   x.state = SLP;
7   x.next = x.prev = null;
8 }
```



$A1.$	$\forall p.\ p \neq \texttt{null} \land p.\texttt{next} \neq \texttt{null}$		$\Rightarrow p.\texttt{next}.\texttt{prev} = p$
$A2.$	$\forall p.\ p \neq \texttt{null} \land p.\texttt{prev} \neq \texttt{null}$		$\Rightarrow p.\texttt{prev}.\texttt{next} = p$
$A3.$	$\forall p.\ p \neq \texttt{null} \land p.\texttt{next} \neq \texttt{null}$		$\Rightarrow p.\texttt{state} = p.\texttt{next}.\texttt{state}$
$A4.$	$\forall p.\ p \neq \texttt{null} \land p.\texttt{next} \neq \texttt{null}$	$\land\ p.\texttt{state} = \texttt{RUN}$	$\Rightarrow p.\texttt{prio} \geq p.\texttt{next}.\texttt{prio}$

Fig. 1. A scheduler `remove` function and its data structure axioms.

showing that its negation is unsatisfiable. For example, to prove that we do not dereference `null` in `x.next` on line 3, we must show unsatisfiability of $PRE \ \land$ $\texttt{x} = \texttt{null}$. This can be done without any reference to the axioms.

What is harder to show is that the axioms hold when the function returns. Consider first showing that axiom A3 still holds, which is non-trivial since the axiom depends on the mutated fields `next` and `state`. An update $q.\texttt{f} = v$ is modeled by saying that the function modeling field `f` is changed into $\texttt{f}[q \mapsto v]$, with semantics

$$p.\texttt{f}[q \mapsto v] = \begin{cases} v & \text{if } p = q \\ p.\texttt{f} & \text{otherwise} \end{cases} \tag{upd}$$

The updated values of the `next` and `state` fields relevant to A3 are:

$$\begin{aligned} \texttt{next'} &= \texttt{next}[\texttt{x.prev} \mapsto \texttt{x.next}][\texttt{x} \mapsto \texttt{null}] \\ \texttt{state'} &= \texttt{state}[\texttt{x} \mapsto \texttt{SLP}] \end{aligned}$$

We have to verify that A3 still holds:

$$\forall q.\ q \neq \texttt{null} \land q.\texttt{next'} \neq \texttt{null} \ \Rightarrow\ q.\texttt{state'} = q.\texttt{next'}.\texttt{state'} \tag{goal}$$

To prove a formula involving updated fields, such as q.`next'`, we first eliminate the field updates by performing the case analysis suggested by the update equation (upd). In each case, what remains to be shown is that a conjunction of literals involving field accesses is unsatisfiable in the presence of some universally quantified local equality axioms.

For our example, there are twelve cases to consider: $q.\texttt{next'}$ has cases $q = \texttt{x}$, $q = \texttt{x.prev} \neq \texttt{x}$, and $q \neq \texttt{x} \land q \neq \texttt{x.prev}$; $q.\texttt{state'}$ has cases $q = \texttt{x}$ and $q \neq \texttt{x}$; and $q.\texttt{next'}.\texttt{state'}$ has cases $q.\texttt{next'} = \texttt{x}$ and $q.\texttt{next'} \neq \texttt{x}$. Several cases can be shown unsatisfiable without relying on the axioms.

Four cases require using the axioms; one such case is when $q = \texttt{x.prev} \neq \texttt{x}$ and $q.\texttt{next'} = \texttt{x}$. Consequently $q.\texttt{state'} = q.\texttt{state}$, $q.\texttt{next'} = \texttt{x.next}$ and $q.\texttt{next'}.\texttt{state'} = \texttt{x.next.state}$. We must show $q.\texttt{state} = \texttt{x.next.state}$. We

Injectivity	$\forall p.\ p \neq \texttt{null} \wedge p.\texttt{a} \neq \texttt{null} \Rightarrow p.\texttt{a}.\texttt{b} = p$
Transitivity	$\forall p.\ p \neq \texttt{null} \wedge p.\texttt{a} \neq \texttt{null} \Rightarrow p.\texttt{a}.\texttt{b} = p.\texttt{b}$
Order	$\forall p.\ p \neq \texttt{null} \wedge p.\texttt{a} \neq \texttt{null} \Rightarrow p.\texttt{a}.\texttt{b} > p.\texttt{b}$
Grid	$\forall p.\ p \neq \texttt{null} \wedge p.\texttt{a} \neq \texttt{null} \wedge p.\texttt{f} \neq \texttt{null} \Rightarrow p.\texttt{a}.\texttt{f} = p.\texttt{f}.\texttt{a}$

Fig. 2. Four important axiom forms.

first instantiate $A2$ at `x` (written $A2[\texttt{x}/p]$) to derive $\texttt{x.prev.next} = \texttt{x}$, implying $q.\texttt{next} = \texttt{x}$. Then we instantiate $A3[q/p]$ to derive $q.\texttt{state} = \texttt{x.state}$, and finally $A3[\texttt{x}/p]$ to derive $\texttt{x.state} = \texttt{x.next.state}$.

The essence of the above discussion is that the verification strategy will perform case analysis based on the memory writes and will generate facts that must be proved unsatisfiable using a number of axiom instantiations. The only difficulty is how to decide what instances of axioms to use; a strategy that instantiates too few axioms will not be complete (we might fail to prove unsatisfiability), while a strategy that instantiates too many might not terminate.

2.1 Unrestricted Scalar Predicates

Scalar predicates let us connect the shape of a data structure with the data stored within it. One advantage of our specification strategy is that we we can combine our satisfiability procedure with that of any predicate that works within the framework of a Nelson-Oppen theorem prover [15]. While other approaches often abstract scalars as boolean fields [12, 16], we can reason about them precisely. For example, in order to verify that the function `remove` shown in Figure 1 preserves the priority ordering axiom A4, we need transitivity of $\geq$, so we use a satisfiability procedure for partial orders.

2.2 Useful Axiom Patterns

In the process of specifying data structures, we have identified several very useful axiom patterns; four are shown in Figure 2. For example, axiom A1 of the scheduler example is an instance of the *injectivity* pattern. Such an injectivity axiom implies useful must-not-alias facts such as $x \neq y \wedge x.\texttt{a} \neq \texttt{null} \ \Rightarrow\ x.\texttt{a} \neq y.\texttt{a}$.

Injectivity can specify tree shapes as well, for example:

$$\forall p.\ p \neq \texttt{null} \wedge p.f \neq \texttt{null} \ \Rightarrow\ p.f.\texttt{inv} = p \ \wedge\ p.f.\texttt{kind} = \text{“}f\text{”}$$

where f is a tree field name such as `left` or `right`, `kind` is a scalar field, and there is one axiom for each f. Such axioms specify that the fields f are *mutually injective*, because `inv` is the inverse of their union.

The axioms A3 and A4 from the example are similar in that they relate the value of a certain field across the `next` field, which is in some sense *transitive*. A transitivity axiom can be used to say that two memory cells are in separate instances of a data structure; we used axiom A3 to require that the `runnable` and the `sleeping` lists are disjoint. More generally, we can prove the powerful

Globals	x	$\in$	Var	Pointer equalities	E	$::=$	$t_1 = t_2$
Variables	p	$\in$	Var	Pointer disequal.	D	$::=$	$t_1 \neq t_2$
Pointer fields	L	$\in$	PField	Scalar fields	S, R	$\in$	SField
Pointer paths	α, β	$::=$	$\epsilon \mid \alpha.L$	Scalar predicates	P	$\in$	Pred
Pointer terms	t, u	$::=$	$\texttt{null} \mid x \mid p.\alpha$	Scalar constraints	C	$::=$	$P(t_1.S, t_2.R)$

Fig. 3. Specification language.

must-not-reach fact $x.\mathtt{b} \neq y.\mathtt{b} \;\Rightarrow\; x.\mathtt{a}^n \neq y.\mathtt{a}^m$, where $x.\mathtt{a}^n$ means the object reached from x after following the a field n times. But note that transitivity axioms cannot express must-reach facts.

Axiom A4 is a non-strict *order* axiom. In its strict form it can be used to specify the absence of cycles. In this pattern, any transitive and anti-reflexive binary predicate can be used in place of $>$.

We defer discussion of the grid pattern to Section 6. A common theme in these patterns is that we use *equalities* and scalar predicates to imply *disequalities*, which are needed to reason precisely about updates. The interplay between equality axioms and the entailed disequalities is central to our approach.

2.3 Ghost Fields

Often, a data structure does not physically have a field that is needed to specify its shape. For example, a singly-linked list does not have the back pointers needed to witness injectivity of the forward pointers. In such cases, we propose to simply add *ghost fields* (also known as auxiliary variables, or specification fields), which are fields that our verification algorithm treats as real, but do not actually exist when the code is compiled and run. Thus, to specify a singly-linked list, we add `prev` as a ghost field and instead specify a doubly-linked list. Updates to ghost fields can sometimes be inferred automatically, but not always [9].

3 The Specification Language

Figure 3 describes the main syntactic elements of the specification language. This is a two-sorted logic, with pointer values and scalar values. Scalar predicates may have any arity. We use the notation $\mathcal{E}$ for disjunctions of pointer equalities, $\mathcal{D}$ for disjunctions of pointer disequalities, and $\mathcal{C}$ for disjunctions of scalar constraints.

Core Language. We have two languages, one a subset of the other. In the core language, a *data structure specification* is a finite set of axioms of the form

$$\forall p.\;\; \mathcal{E} \vee \mathcal{C} \qquad \text{(core)}$$

where p ranges over pointer values, with an additional syntactic constraint to be described later. Axioms in the core language cannot have pointer disequalities, but can have scalar field disequalities in $\mathcal{C}$. This language is expressive enough to describe many common structures, including those of Figures 1 and 2. Section 4 describes a complete satisfiability procedure for this language.

Extended Language. The extended language has axioms of the form

$$\forall p.\ \ \mathcal{E} \vee \mathcal{C} \vee \mathcal{D} \tag{ext}$$

This language is more expressive; e.g., it allows us to insist that certain pointers not be `null`, or to describe additional kinds of reachability, or to require that a structure be cyclic, among other things. Unfortunately, the extended language includes specifications with undecidable theories (see below). However, we extend the satisfiability procedure for the core language to handle many extended axioms as well, including all forms that we have encountered in our experiments.

Nullable Subterms. Data structure specification axioms naturally have the following *nullable subterms* (NS) property: for any pointer term $t.L$ or any scalar term $t.S$ that appears in the body of an axiom, the axiom also contains the equality $t = \mathtt{null}$ among its disjuncts. This is because fields are not defined at `null`. We require that all axioms have the NS property.

Discussion. The keystone of our technical result is the observation that the NS property ensures the decidability of the axioms in the core language. In contrast, if we allow axioms of the form $\forall p.\ p.\alpha = p.\beta$ (in the core language, but without the NS property), then we could encode any instance of the (undecidable) "word problem" [7] as an axiom set and a satisfiability query.

Intuitively, an axiom with the nullable subterms property can be satisfied by setting to `null` any unconstrained subterms. This avoids having to materialize new terms, which in turn ensures termination. Notice however, that if we allow arbitrary pointer disequalities we can cancel the effect of the NS condition. For example, the axiom

$$\forall p.\ \ p = \mathtt{null}\ \vee\ p.\mathtt{a} \neq \mathtt{null}$$

forces $t.\mathtt{a}$ to be non-`null` for any non-`null` t. Thus the unrestricted use of the disequalities in $\mathcal{D}$ makes satisfiability undecidable. In our experiments we observed that pointer disequalities are needed less frequently than other forms, which motivates separate treatment of the core and extended languages.

4 A Satisfiability Algorithm

4.1 The Algorithm

The purpose of the algorithm is to determine whether a set of local equality axioms and a set of ground (unquantified) facts is satisfiable. When used on axioms without pointer disequalities (core language) the algorithm always terminates with a definite answer. However, in the presence of axioms with pointer disequalities (extended language) the algorithm may return "maybe satisfiable". Note that pointer disequalities in the ground facts do not endanger completeness.

The basic idea of the algorithm is to exploit the observation that in a data structure described by local axioms, when the facts are satisfiable, they are satisfiable by a "small" model. Essentially, the algorithm attempts to construct such a model by setting any unknown values to `null`.

```
unsat(G, DS) =
  if G is contradictory then true
  elseif DS = (DS′ ∧ false) then true
  elseif a term u ∈ G matches axiom ∀p. ℰ ∨ 𝒞 ∨ 𝒟,
         not yet instantiated for class of u, then
    for each t1 = t2 ∈ ℰ,
      unsat(G ∧ t1[u/p] = t2[u/p], DS);
    for each P(t1.S, t2.R) ∈ 𝒞,
      unsat(G ∧ P(t1[u/p].S, t2[u/p].R), DS);
    unsat(G, DS ∧ 𝒟[u/p])
  elseif DS = true then
    raise Satisfiable(G)
  elseif DS = (DS′ ∧ (t1 ≠ t2 ∨ 𝒟))
         with t1 ∈ G and t2 ∈ G, then
    unsat(G ∧ t1 ≠ t2, DS′);
    unsat(G, DS′ ∧ 𝒟)
  else   /* search for a cyclic model */
    for each term t1 ∉ G where t1 ≠ t2 ∈ DS,
      for each term t3 ∈ G,
        unsat(G ∧ t1 = t3, DS);
    raise MaybeSatisfiable   /* give up */
```

Fig. 4. The basic decision algorithm.

The algorithm's central data structure is an equality graph (e-graph) G [15], a congruence-closed representation of the ground facts. We use the notation $t \in G$ to mean term t is *represented* in G, and $G \wedge f$ to mean the e-graph obtained by adding representatives of the terms in formula f to G, and asserting f.

The algorithm must decide which axioms to instantiate and for which terms. We first define when a ground pointer term u matches an equality disjunct $t_1 = t_2$ in an axiom with bound (quantified) variable p, as follows:

- Either $t_1[u/p] \in G$ or $t_2[u/p] \in G$, when neither t_1 nor t_2 is `null`, or
- t_1 is `null` and $t_2[u/p] \in G$ (or vice-versa).

We say that a term matches an axiom if it *matches all its pointer equality disjuncts*. An axiom is instantiated with any term that matches the axiom.

These rules implement an "all/most" strategy of instantiation. For example, an axiom

$$\forall p. \ \ \ldots \ \vee \ p.\alpha.\mathtt{a} = p.\beta.\mathtt{b}$$

must include (because of NS) disjuncts $p.\alpha = \mathtt{null}$ and $p.\beta = \mathtt{null}$. For a ground term u to match this axiom, it must match every equality disjunct (from the definition above), so $u.\alpha$, $u.\beta$, and either $u.\alpha.\mathtt{a}$ or $u.\beta.\mathtt{b}$ must be represented. Consequently, asserting the literal $u.\alpha.\mathtt{a} = u.\beta.\mathtt{b}$ will require representing at most one new term, *but no new equivalence classes*.

If G is an e-graph, and DS is a conjunction (i.e., a set) of disjunctions of pointer disequalities, we define a satisfiability procedure $\mathtt{unsat}(G, DS)$ that returns `true` if the facts represented in G along with DS and the axioms are unsatisfiable, raises the exception `Satisfiable` if it is satisfiable, and raises the exception `MaybeSatisfiable` if the procedure cannot decide satisfiability (in presence of axioms with pointer disequalities). Figure 4 contains the pseudocode for this procedure. This procedure is used by first representing the facts in an e-graph G, and then invoking $\texttt{unsat}(G, \texttt{true})$.

Lines 2 and 3 identify contradictions in the e-graph. The judgment "G is contradictory" may make use of a separate satisfiability procedure for the scalar predicates. We write $DS = (DS' \wedge \texttt{false})$ to deconstruct DS. The heart of the algorithm is in lines 4–10, which instantiate axioms with terms that match, given the matching rules described above, and performs the case analysis dictated by the instantiated axioms. Note that the pointer disequalities are deferred by collecting them in DS (line 10). When line 12 is reached, the axioms are completely instantiated and all cases fully analyzed, but no contradiction has been found. Thus, the original $G \wedge DS$ was satisfiable; the current G is a witness.

Lines 13–16 handle pointer disequalities where both sides are already represented in the e-graph. Finally, lines 17–20 attempt to find a satisfying assignment for the unrepresented terms in DS by setting them equal to other terms that *are* represented. The algorithm searches for cyclic models to accommodate data structures like cyclic lists.

Recalling the proof from Section 2 that A3 holds after `next` and `state` have been modified, we can now explain how the algorithm would prove it. First, the update rules are used to split the proof into cases; among those cases is $q \neq \texttt{x} \wedge q = \texttt{x.prev} \wedge \texttt{x.next} \neq \texttt{x}$, which (along with the precondition) is used to initialize G, and $\texttt{unsat}(G, \texttt{true})$ is invoked. Next, the algorithm determines that the term `x` matches axiom $A2$, because $\texttt{x.prev} \in G$, and begins asserting its disjuncts. Only the $\texttt{x.prev.next} = \texttt{x}$ disjunct is not immediately refuted. Asserting this literal causes the new term `x.prev.next` to be represented, by putting it into the same equivalence class as `x`. Since $q = \texttt{x.prev}$, the term $q.\texttt{next}$ is now represented, and so q matches $A3$, and $A3[q/p]$ is instantiated. Finally, as `x.next` is represented, $A3[\texttt{x}/p]$ is also instantiated. After these three instantiations, case analysis and reasoning about equality are sufficient.

4.2 Analysis of the Algorithm

Correctness when result is "Unsatisfiable". The algorithm is always right when it claims unsatisfiability, which in our methodology means that the goal is proved. The (omitted) proof is a straightforward induction on the recursion.

Correctness when result is "Satisfiable". When the algorithm raises $\texttt{Satisfiable}(G')$, there is a model Ψ that satisfies the original $G \wedge DS \wedge \mathcal{A}$, where $\mathcal{A}$ is the set of axioms:

$$\Psi(u) = \begin{cases} u^* & \text{if } u \in G' \\ \texttt{null} & \text{otherwise} \end{cases}$$

where by u^* we denote the representative of u in the e-graph G'. Ψ satisfies $\mathcal{A}$ because for every pointer term u (represented or not) and axiom A, either $A[u/p]$

has been instantiated, so G' satisfies $A[u/p]$ (and so does Ψ), or else u does not match A, in which case there is a pointer equality $t_1 = t_2$ in A that does not match. By the definition of matching, there are two cases:

– Neither t_1 nor t_2 are `null`, and $t_1[u/p] \notin G$ and $t_2[u/p] \notin G$. But then $\Psi(t_1[u/p]) = \Psi(t_2[u/p]) = \texttt{null}$, satisfying $A[u/p]$.
– t_1 is `null` and $t_2[u/p] \notin G$. Then $\Psi(t_2[u/p]) = \texttt{null}$, satisfying $A[u/p]$.

That is, the matching rules guarantee that if an axiom has not been instantiated, it can be satisfied by setting an unrepresented term to `null`.

Termination. The essence of the termination argument is that none of the steps of the algorithm increases the number of pointer equivalence classes in the e-graph. Whenever a new pointer term t might be created in the process of extending the e-graph, it is created while adding the fact $t = t'$, where t' is already in the e-graph. This is ensured by the matching rule that requires one side of each equality to be present in the e-graph. Furthermore, if t is of the form $t''.L$, then by the NS property, the disjunct $t'' = \texttt{null}$ is also part of axiom and thus the matching rule requires t'' to be already in the e-graph. Thus t is the only new term and is added to an existing equivalence class. There are only a finite number of equality and scalar constraints over a fixed number of pointer equivalence classes; once all entailed constraints are found, the algorithm halts.

Soundness and completeness on the Core Language. For the core language, the algorithm stays above line 13, because DS is always empty (`true`). As it always terminates, it must always return either Satisfiable or Unsatisfiable.

Complexity. The algorithm has exponential complexity, because of the case analysis that is performed in processing updates and conditional axioms. If we have n nodes in the e-graph representing the facts, a axioms with at most k disjuncts each, then there could be a total of $n \times a$ axiom instantiations. We can think of the algorithm exploring a state space that resembles a tree. Each node in the tree corresponds to an instance of an axiom, and has a branching factor k. Consequently its running time is $O(k^{na}S(n))$, where $S(n)$ is the complexity bound for the underlying satisfiability procedure for scalar constraints.

5 Limitations

5.1 Never-`null` pointers

Data structures sometimes have pointers that are never `null`. This can be specified with a disequality axiom such as $\forall p.\ p = \texttt{null} \lor p.\texttt{a} \neq \texttt{null}$. Under the right circumstances, such an axiom may necessitate the creation of a new pointer equivalence class in order to find a model.

The problem with creating new equivalence classes is it jeopardizes termination: by creating a new class, a new axiom might fire, creating more classes, etc. To address this, we use a heuristic to decide when to stop growing the interpretation (the e-graph) and give up: the programmer supplies an expansion horizon k, and the algorithm never creates an equivalence class more than k "hops" away from the original set of classes. In practice, the bound $k = 2$ works well.

5.2 Termination Conditions

One troublesome case that arises with some frequency is *termination conditions* on transitive fields. Consider the axioms

$$\begin{array}{l} \forall p.\;\; p = \texttt{null} \;\vee\; p.\texttt{next} = \texttt{null} \;\vee\; p.\texttt{next}.\texttt{tag} = p.\texttt{tag} \\ \forall p.\;\; p = \texttt{null} \;\vee\; p.\texttt{next} = \texttt{null} \;\vee\; p.\texttt{next}.\texttt{prev} = p \\ \forall p.\;\; p = \texttt{null} \;\vee\; p.\texttt{next} \neq \texttt{null} \;\vee\; p.\texttt{tag} = 3 \end{array}$$

These axioms say that everywhere along a chain of `next`s, which is injective, the `tag` fields have the same value. Moreover, the *last* element in the chain has a `tag` field with the value 3. Combined with the facts $x \neq \texttt{null}$, $x.\texttt{prev} = \texttt{null}$ and $x.\texttt{tag} = 2$, the only models are infinite, since a finite model cannot be cyclic, owing to the injectivity of `next` and $x.\texttt{prev} = \texttt{null}$, nor can it terminate, since then $2 = 3$. Our algorithm would be forced to raise `MaybeSatisfiable`.

5.3 Reachability

To directly reason about reachability requires a logic with transitive closure. However, our language does not have transitive closure, and this is by design, since deciding (for example) satisfiability in the presence of transitive closure is difficult [5]. However, with a little creativity, one can often find ways around using reachability. The most common use of reachability is to express *disjointness*. For example, writing $\rightsquigarrow$ to mean "reaches", one might write

$$\forall p.\;\; \neg(x \rightsquigarrow p \;\wedge\; y \rightsquigarrow p)$$

to mean that x and y point at disjoint lists. Instead, we would use a transitivity axiom to define a data structure label, and then say that x and y have different labels. Another use of reachability is to specify *acyclicity*, e.g.:

$$\forall p.\;\; x \rightsquigarrow p \;\Rightarrow\; p.\texttt{next} \not\rightsquigarrow p$$

says that x points at an acyclic list. Often, an injectivity axiom is sufficient. However, as injectivity doesn't rule out all cycles, we sometimes must additionally use the "order" axiom of Section 2.2.

6 Experiments

To evaluate the expressiveness of our specification language, we compare it to the Pointer Assertion Logic Engine (PALE) [12]. To measure the effectiveness of our decision algorithm, we compare it to the Simplify [1] theorem prover. For several example programs, we measure the lines of annotation, which axiom forms are used (from Figure 2, plus disequalities), and the time to verify on a 1.1GHz Athlon Linux PC with 768MB RAM. See Figure 5.

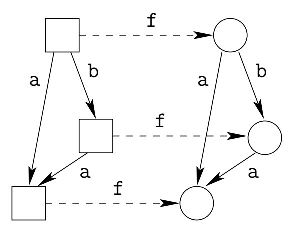
We are able to verify all of the examples distributed with PALE, however PALE requires less than half the lines of annotation. There are three reasons:

		Annot. lines		Axiom Forms					Verify Time (s)		
example	LOC	PALE	Ours	$\neq$	inj	trans	order	grid	PALE	Simplify	Ours
`bubblesort`	47	10	31		2				1.9	0.9	0.6
`doublylinked`	73	35	69		4	1			2.6	2.9	1.9
`taillist`	73	38	94	3	4	4			2.4	1.8	1.3
`redblacktree`	146	112	224		4		4		19.4	36.5	22.0
`gc_copy`	38		58	1	1			2		50.2	15.6
`btree`	163		185	2	7	4	6			96.7	26.4
`pc_keyb`	1116		163	2						40.1	38.1
`scull`	534		360	4	5					58.2	46.4

Fig. 5. Programs verified, with comparison to PALE when available.

(1) in PALE, the backbone tree is concisely specified by the `data` keyword, whereas our axioms must spell it out with injectivity, (2) PALE specifications use transitive closure, whereas we use the more verbose transitivity axioms, and (3) our language does not assume the program is type-safe (it is C, after all), so we have explicit axioms to declare where typing invariants hold. We plan to add forms of syntactic sugar to address these points.

We also selected two data structures that cannot be specified in PALE. `gc_copy` is the copy phase of a copying garbage collector, and has the job of building an isomorphic copy of the from-space, as shown at right. This isomorphism, embodied by the forwarding pointers `f`, is specified by the "grid" or "homomorphism" axiom $\forall p.\ \ \ldots\ \Rightarrow\ p.\mathtt{a}.\mathtt{f} = p.\mathtt{f}.\mathtt{a}$. This axiom (along with injectivity of `f`) describes a data structure where the targets of `f` pointers are isomorphic images of the sources.



The `btree` benchmark is a B+-tree implementation, interesting in part because we can verify that the tree is *balanced*, by using scalar constraints

$$\forall p.\quad p \neq \mathtt{null} \land p.\mathtt{left} \neq \mathtt{null}\ \Rightarrow\ p.\mathtt{level} = p.\mathtt{left}.\mathtt{level} + 1$$
$$\forall p.\quad p \neq \mathtt{null} \land p.\mathtt{right} \neq \mathtt{null}\ \Rightarrow\ p.\mathtt{level} = p.\mathtt{right}.\mathtt{level} + 1$$

along with a disequality constraint that forces all leaves to have the same `level`. However, this specification is also noteworthy because there is no bound k on the size of models as advocated in Section 5.1: the facts $x.\mathtt{level} = 100$ and $x \neq \mathtt{null}$ (plus the axioms) are satisfiable only by models with at least 100 elements. Fortunately, such pathological hypotheses do not seem to arise in practice.

Finally, `pc_keyb` and `scull` are two Linux device drivers. These real-world examples demonstrate the applicability of the technique, and the benefits of integrated pointer and scalar reasoning; for example, their data structures feature arrays of pointers, which are difficult to model in PALE or TVLA.

An important contribution of this work is the particular matching rules used to instantiate the axioms. Simplify has heuristics for creating matching rules for arbitrary universally-quantified facts. As expected, we find that our matching rules, which instantiate the axioms in strictly fewer circumstances, lead to better performance (compare timing columns "Simplify" and "Ours" in Figure 5). In fact, Simplify's heuristics will often lead to infinite matching loops, especially while trying to find counterexamples for invalid goals. This is not to denounce Simplify's heuristics—they do a good job for a wide variety of axiom sets—just to emphasize that in the case of axioms expressing data structure shape within our methodology, one can have a more efficient and predictable algorithm.

7 Related Work

As explained in Section 1, most existing approaches to specifying shape are either too imprecise or too difficult to reason about automatically. Here, we consider alternative approaches with similar expressiveness and effectiveness.

PALE [12], the Pointer Assertion Logic Engine, uses graph types [6] to specify a data structure as consisting of a spanning tree backbone augmented with auxiliary pointers. One disadvantage is the restriction that the data structure have a tree backbone. Disequality constraints that force certain pointers to not be `null` are not possible, since every tree pointer is implicitly nullable. Cyclic structures are at best awkward, and grid structures such as the garbage collector example in Section 6 are impossible. The second disadvantage is that only boolean scalar fields are allowed. Thus, all scalar values must first be abstracted into a set of boolean ghost fields, and updates inserted accordingly.

TVLA [11, 16], the Three Valued Logic Analyzer, uses abstract interpretation over a heap description that includes 1/2 or "don't know" values. It obtains shape precision through the use of *instrumentation predicates*, which are essentially ghost boolean fields with values defined by logical formulas. In general, the programmer specifies how instrumentation predicates evolve across updates, though TVLA can conservatively infer update rules that are often sufficiently precise. The primary advantage of TVLA is it infers loop invariants automatically, by fixpoint iteration. The disadvantage is that the obligation of proving that a shape is preserved across updates is delegated to the instrumentation predicate evolution rules, which are not fully automated. Also, as with PALE, scalar values must be abstracted as boolean instrumentation predicates.

PALE and TVLA include transitive closure operators, and hence can reason directly about reachability, but they pay a price: PALE has non-elementary complexity and requires tree backbones, and TVLA has difficulty evolving instrumentation predicates when they use transitive closure. The difficulties of reasoning about transitive closure have been recently explored by Immerman et. al [5], with significant negative decidability results. Our technique is to approximate reachability using transitivity axioms, giving up some shape precision in exchange for more precision with respect to scalar values.

The shape analysis algorithm of Hackett and Rugina [4] partitions the heap into regions and infers points-to relationships among them. Its shape descriptions are less precise; it can describe singly-linked lists but not doubly-linked lists.

Roles [8] characterize an object by the types it points to, and the types that point at it. Role specifications are similar to our injectivity axioms. The role analysis in [8] provides greater automation but it can express fewer shapes.

Our work uses methods most similar to the Extended Static Checker [2] and Boogie/Spec# [10]. However, while we suspect that specifications similar to elements described here have been written before while verifying programs, we are unaware of any attempts to explore their expressiveness or decidability.

8 Conclusion

We have presented a language for describing data structure shapes along with scalar field relationships. The language is relatively simple because it can only talk about local properties of a neighborhood of nodes, yet is capable of expressing a wide range of global shape constraints sufficient to imply the kind of must-not-alias information needed for strong update reasoning. Furthermore, it is sufficiently tractable to admit a simple decision procedure, extensible with arbitrary decidable scalar predicates. Using this language we have verified a number of small example programs, of both theoretical and practical interest.

References

1. D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: A theorem prover for program checking. Technical Report HPL-2003-148, HP Labs, July 2003.
2. D. L. Detlefs, K. R. M. Leino, G. Nelson, and J. B. Saxe. Extended static checking. Technical Report SRC-159, COMPAQ SRC, Palo Alto, USA, Dec. 1998.
3. P. Fradet and D. L. Métayer. Shape types. In *POPL*, pages 27–39, 1997.
4. B. Hackett and R. Rugina. Region-based shape analysis with tracked locations. In *POPL*, pages 310–323, 2005.
5. N. Immerman, A. Rabinovich, T. Reps, M. Sagiv, and G. Yorsh. The boundary between decidability and undecidability for transitive-closure logics. In *CSL*, 2004.
6. N. Klarlund and M. Schwartzbach. Graph types. In *POPL*, pages 196–205, 1993.
7. D. E. Knuth and P. B. Bendix. Simple word problems in universal algebra. In J. Leech, editor, *Computational Problems in Abstract Algebras*, pages 263–297. Pergamon Press, 1970.
8. V. Kuncak, P. Lam, and M. Rinard. Role analysis. In *POPL*, pages 17–32, 2002.
9. V. Kuncak and M. Rinard. Existential heap abstraction entailment is undecidable. In *SAS*, pages 418–438, 2003.
10. K. R. M. Leino and P. Müller. Object invariants in dynamic contexts. In *European Conference on Object-Oriented Programming (ECOOP)*, 2004.
11. T. Lev-Ami and S. Sagiv. TVLA: A system for implementing static analyses. In *Static Analysis Symposium*, pages 280–301, 2000.
12. A. Möller and M. Schwartzbach. The pointer assertion logic engine. In *PLDI*, pages 221–231, 2001.
13. G. Nelson. Techniques for program verification. Technical Report CSL-81-10, Xerox Palo Alto Research Center, 1981.
14. G. Nelson. Verifying reachability invariants of linked structures. In *POPL*, pages 38–47, 1983.

15. G. Nelson and D. C. Oppen. Fast decision procedures based on congruence closure. *JACM*, 27(2):356–364, Apr. 1980.
16. S. Sagiv, T. W. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *TOPLAS*, 24(3):217–298, 2002.