MIT CSAIL Technical Report No 975, November 2004

On Decision Procedures for Set-Valued Fields

Viktor Kuncak [1]   Martin Rinard [2]

MIT Computer Science and Artificial Intelligence Laboratory
Cambridge, Massachusetts, USA

Abstract

An important feature of object-oriented programming languages is the ability to dynamically instantiate user-defined container data structures such as lists, trees, and hash tables. Programs implement such data structures using references to dynamically allocated objects, which allows data structures to store unbounded numbers of objects, but makes reasoning about programs more difficult. Reasoning about object-oriented programs with complex data structures is simplified if data structure operations are specified in terms of abstract sets of objects associated with each data structure. For example, an insertion into a data structure in this approach becomes simply an insertion into a dynamically changing set-valued field of an object, as opposed to a manipulation of a dynamically linked structure linked to the object.

In this paper we explore reasoning techniques for programs that manipulate data structures specified using set-valued abstract fields associated with container objects. We compare the expressive power and the complexity of specification languages based on 1) decidable prefix vocabulary classes of first-order logic, 2) two-variable logic with counting, and 3) Nelson-Oppen combinations of multisorted theories. Such specification logics can be used for verification of object-oriented programs with supplied invariants. Moreover, by selecting an appropriate subset of properties expressible in such logic, the decision procedures for these logics yield automated computation of lattice operations in abstract interpretation domain, as well as automated computation of abstract program semantics.

1 Introduction

Analysis and verification of modern object-oriented programming languages poses unique challenges [50, 34, 44, 20]. In this paper we study a feature that we consider essential for object-oriented programming: the ability to introduce user-defined abstract data types, and create an unbounded number of instances of these data types during program execution. Particular difficulties

[1] Email: vkuncak@csail.mit.edu
[2] Email: rinard@csail.mit.edu

arise when each data type instance is itself implemented using multiple dynamically allocated objects that form a linked data structure. Our approach for analyzing such programs is to use abstract set-valued fields as specification variables that describe operations of an abstract data type, and separate the analysis of the program into verifying the correctness of the implementation of the abstract data type with respect to the set specification and verifying the correctness of the rest of the program where linked data structure is replaced by abstract set-valued fields. We next give some more context of our work.

Global abstract data types. An important feature of modern programming languages is the ability to introduce user-defined abstract data types; such data types allow the developers to build applications on top of concepts that are most appropriate for the applications, as opposed to relying only on the concepts built into the language. Modules have been used successfully as a language mechanism for implementing abstract data types [33, 54, 39, 36], and are an effective way of specifying abstract data types if there is only one instance of the abstract data type in the program, or if the instances are implemented without using linked data structures.

Linked data structures. Containers that store objects form a large class of user-defined data types. Such containers are often implemented as linked data structures (such as lists, trees, and hash tables) that use references to cells dynamically allocated on the heap. Reasoning about programs containing linked data structures is difficult because there is no compile-time bound on the size and the complexity of the linking structure that can be created. Sophisticated shape analyses have been developed to statically analyze sets of possible linking structures created by programs [37, 23, 15, 47, 16, 11]. Shape analyses are generally effective with analyzing individual data structures, but often have difficulties scaling to larger programs.

Hob project. One of the main design principles behind the Hob project [30, 31, 58, 29] is that reasoning about programs with complex data structures becomes simpler if data structure operations are specified in terms of abstract sets of objects associated with each data structure. For example, an insertion into a data structure in this approach becomes simply an insertion into a dynamically changing sets of objects, as opposed to a manipulation of a dynamically linked data structure. Hob splits the verification of programs with such data structures into two tasks: 1) using shape analysis to verify that data structure implementation conforms to the specification given in terms of the abstract set variables, and 2) using only the abstract set variables in the rest of the program to reason about the behavior of the data structure. The use of different analysis techniques is possible because Hob architecture supports the combination of heterogeneous analysis plugins while analyzing a single program. So far, we have used Hob to verify implementations of global data structures, which are instantiated at compile time into a finite number of instances. The focus on global data structures allowed us to use static module mechanism to encapsulate fields of objects and prevent representation exposure, as well as to use the decidable theory of Boolean algebras [24, 48] to reason about the finite number of abstract sets that specify data structures. Our goal is to make Hob applicable to dynamically instantiated data structures as well.

Dynamic instantiation of linked data structures. Dynamic instantiation of abstract data types is one of the key features of object-oriented programming languages. Dynamic instantiation is typically achieved by associating abstract data type instance with an object, and using a field to attach the underlying linked data structure to the object. We are currently extending Hob to verify programs that use linked data structures that can be dynamically instantiated. In this approach, we specify a linked data structure attached to an object using a finite number of set-valued fields of an object. The result of abstracting the content of data structures in a program using this technique is a program that manipulates objects connected using relations. A relation in the resulting program can be either a function (whose value for a given object is the object referenced by an object-valued field), or a general relation (whose value for a given object is the set of objects stored in the data structure associated with the object).

The generalization to dynamic instantiation of data structures in Hob requires extensions to both phases of verification: 1) verification that linked data structure conforms to the set interface given by values of object fields and 2) verification of the resulting program that uses objects with set-valued fields. To address the first problem, we are extending the existing technique in Hob with the techniques for specifying representations of individual objects [42, 8, 6, 2, 3]; these extensions are necessary to ensure that the analysis of one instance remains valid in the presence of other instances in the heap.

The topic of this paper is the second problem: verification of programs that manipulate objects with set-valued fields. Like [45], we are concerned with verification of clients of abstract data types, but we focus on specifications expressed in terms of set-valued fields and derive a complete decision procedure for the constraints in our class. Our approach uses assume/guarantee reasoning with user-supplied annotations to completely separate the analysis of the implementation of the class from the analysis of the context; other approaches attempt to automatically infer both the approximation the context and the approximation of class implementation [34], potentially using a global fixpoint analysis.

Decision procedures for set-valued fields. To study the automation of reasoning about programs with set-valued fields, we explore decision procedures for constraints on such fields. Our constraints can express relationships between sets associated with the same object, the aliasing between object references, as well as the relationships between sets associated with different objects. By annotating programs with such constraints and using a verification-condition generator [58], developers can verify a range of invariants of object-oriented programs. Moreover, by selecting an appropriate subset of properties expressible using such constraints, a decision procedure for these constraints yields automated computation of lattice operations in abstract interpretation domain, as well as automated computation of abstract program semantics (transfer functions) for the analysis [9].

```
assume $x \neq \mathsf{null} \land x \in \mathsf{alloc}$;
oldxc := $x$.c;
new $y$;
while  $[x \neq \mathsf{null} \land y \neq \mathsf{null} \land x \neq y \land x.\mathsf{c} \cup y.\mathsf{c} = \mathsf{oldxc}]$
       $(x.\mathsf{c} \neq \emptyset)$
{
    $e$ := removeFirst($x$);
    // process($e$);
    insert($y, e$);
}
assert $y$.c = oldxc;
```

Fig. 1. An example program fragment that manipulates set-valued fields. Here $z.\mathsf{c}$ denotes the value of the set associated with object denoted by z.

```
$e$ := removeFirst($x$) :
    havoc $e$;
    assume $e \in x.\mathsf{c}$;
    $x.\mathsf{c} := x.\mathsf{c} \setminus \{e\}$
insert($y, e$) :
    $y.\mathsf{c} := y.\mathsf{c} \cup \{e\}$
```

Fig. 2. Specifications of procedure calls from Figure 1

Contributions and overview. To motivate the constraints studied in this paper, we present an example in Section 2. We present our formal setup in Section 3. As the main result of this paper, we explore reasoning techniques for programs that use set-valued abstract fields by comparing the expressive power and the complexity of specification languages based on decidable prefix classes of first-order logic (Section 5), two-variable logic with counting (Section 6), and Nelson-Oppen combinations of multisorted theories (Section 7). We observe that both the decidable prefix class $[\exists^*\forall^*]_=$ and Nelson-Oppen combination yield optimal NP algorithms for deciding an interesting class of constraints. On the other hand, the use of two-variable logic with counting allows more expressive constraints (such as the constraint that a field is never null), but requires an NEXPTIME decision procedure in general. We present our preliminary conclusions in Section 8 and discuss related work throughout the paper.

2 Example

Figure 1 presents an example program fragment containing a precondition (expressed using an assume statement), a loop invariant (expressed using $[\ldots]$ brackets just before the condition of the while loop), and a postcondition (expressed using an assert statement). The program fragment empties the set $x.\mathsf{c}$ and copies its content into the set $y.\mathsf{c}$ (one could imagine some processing of primitive fields of e being performed in each loop iteration, but this is of no relevance to our example). The property that we wish to verify is that loop invariant initially holds:

$$
\begin{aligned}
&x \neq \mathsf{null} \ \wedge \ x \in \mathsf{alloc} \ \Rightarrow \\
&\quad y \notin \mathsf{alloc} \ \wedge \ y \neq \mathsf{null} \ \wedge \ y.\mathsf{c} = \emptyset \ \Rightarrow \\
&\qquad x \neq y \ \wedge \ x \neq \mathsf{null} \ \wedge \ y \neq \mathsf{null} \ \wedge \ x.\mathsf{c} \cup y.\mathsf{c} = x.\mathsf{c}
\end{aligned}
$$

loop invariant is preserved:

$$
\begin{aligned}
&x \neq y \ \wedge \ x \neq \mathsf{null} \ \wedge \ y \neq \mathsf{null} \ \wedge \ x.\mathsf{c} \cup y.\mathsf{c} = \mathsf{oldxc} \ \Rightarrow \\
&\quad e \in x.\mathsf{c} \Rightarrow x \neq y \ \wedge \ (x.\mathsf{c} \setminus \{e\}) \cup (y.\mathsf{c} \cup \{e\}) = \mathsf{oldxc}
\end{aligned}
$$

loop invariant implies postcondition:

$$
\begin{aligned}
&x \neq y \ \wedge \ x \neq \mathsf{null} \ \wedge \ y \neq \mathsf{null} \ \wedge \ x.\mathsf{c} \cup y.\mathsf{c} = \mathsf{oldxc} \ \wedge \ x.\mathsf{c} = \emptyset \ \Rightarrow \\
&\quad y.\mathsf{c} = \mathsf{oldxc}
\end{aligned}
$$

Fig. 3. Verification conditions for Figure 1

$$
\begin{aligned}
O &::= V_O \mid \mathsf{null} \mid O.f_O \\
S &::= V_S \mid S_1 \cup S_2 \mid S_1 \cap S_2 \mid S_1 \setminus S_2 \mid \{O_1, \ldots, O_n\} \mid O.f_S \\
f &::= V_f \mid f_O[O_1 \mapsto O_2] \mid f_S[O \mapsto S] \\
A &::= O_1 = O_2 \mid O \in S \mid S_1 = S_2 \mid \mathsf{card}(S) \leq k \mid f_1 = f_2 \\
F &::= A \mid F_1 \wedge F_2 \mid F_1 \vee F_2 \mid \neg F
\end{aligned}
$$

Fig. 4. Syntax of expressions and formulas

the content of the set $y.\mathsf{c}$ at the end of the program fragment is equal to the original content of the set $x.\mathsf{c}$, which is stored in the auxiliary set-valued local variable oldxc. The property is true, because procedure call removeFirst removes an element from $x.\mathsf{c}$ and returns it in e, and then insert inserts the same element into $y.\mathsf{c}$. Figure 2 shows guarded-command specifications of procedure calls that we use to reason about the effects of procedures; our system verifies separately that procedures conform to their specifications.

Given the precondition, loop invariant and the postcondition for the program fragment in Figure 1, we can generate verification conditions that imply that the program postconditions will hold. Figure 3 shows these verification conditions for the program fragment. Note that the resulting constraints require not only reasoning about the content of individual sets (as in the semantics of insert), but also reasoning about aliasing of references to objects (as in the conjunct $x \neq y$) and reasoning about the relations between sets associated with distinct objects (as in the conjunct $x.\mathsf{c} \cup y.\mathsf{c} = \mathsf{oldxc}$).

In Section 3 we define a class of constraints on objects with set-valued fields, and introduce a guarded command language whose verification conditions belong to this class. In the rest of this paper we study the validity and the satisfiability problem for constraints in this class.

3 Specification Language

We next introduce a specification language for expressing constraints on objects with set-valued fields. The syntax of this language is in Figure 4. Our specification language is typed (multisorted); we are only concerned with well-typed formulas. The nonterminal O denotes objects, which can be potentially

$$\begin{aligned}
\mathsf{wp}(x := E, P) &= P[x := E] \ \wedge\ \mathsf{WD}(E) \\
\mathsf{wp}(\mathsf{havoc}\ x, P) &= \forall x.P \\
\mathsf{wp}(\mathsf{assert}\ Q, P) &= Q \wedge P \\
\mathsf{wp}(\mathsf{assume}\ Q, P) &= Q \Rightarrow P \\
\mathsf{wp}(s_1 \,[]\, s_2, P) &= \mathsf{wp}(s_1, P) \wedge \mathsf{wp}(s_2, P) \\
\mathsf{wp}(s_1 \,;\, s_2, P) &= \mathsf{wp}(s_1, \mathsf{wp}(s_2, P))
\end{aligned}$$

Fig. 5. Weakest preconditions of guarded commands

$$\begin{aligned}
x.f := E \ &\equiv\ f := f[x \mapsto E] \\
\mathsf{new}\ y \ &\equiv\ \mathsf{havoc}\ y; \\
&\ \ \mathsf{assume}\ y \notin \mathsf{alloc} \wedge y \neq \mathsf{null} \wedge y.\mathsf{c} = \emptyset; \\
&\ \ \mathsf{alloc} := \mathsf{alloc} \cup \{y\}
\end{aligned}$$

Fig. 6. Desugaring of some commands

$$\begin{aligned}
\mathsf{WD}(x.f) &= x \neq \mathsf{null} \ \wedge\ \mathsf{WD}(x) \ \wedge\ \mathsf{WD}(f) \\
\mathsf{WD}(f[x \mapsto E]) &= x \neq \mathsf{null} \wedge \mathsf{WD}(f) \wedge \mathsf{WD}(x) \wedge \mathsf{WD}(E) \\
\mathsf{WD}(S_1 \cup S_2) &= \mathsf{WD}(S_1) \wedge \mathsf{WD}(S_2) \\
\mathsf{WD}(\neg F) &= \mathsf{WD}(F)
\end{aligned}$$

Fig. 7. Key clauses in well-definedness of expressions

null, S denotes sets of non-null objects, and f denotes fields. Fields can map objects to objects (then they are denoted f_O) or they can map objects to sets (then they are denoted f_S). We use formulas (the non-terminal F) as part of assume and assert statements, the conditions of while loops, and if statements (which can be represented using assume and $[]$). We use the object-valued and set-valued terms of this language (the non-terminals O and S in Figure 4) on the right-hand side of the assignment statements.

The meaning of constructs in this language is straightforward. Notation $x.f$ denotes a dereference of a field f of objects x, which can be thought of as a function application that signals an error if the object x is null. Notation $f_O[o_1 \mapsto o_2]$ denotes an update of an object-valued field f_O so that $o_1.f_O = o_2$ and the value of the same field for all other objects is the same; such update operation corresponds array update if we view the field f as an array of objects indexed by objects. Set operations in our language have standard meaning. In the expression $\mathsf{card}(S) \leq k$, notation $\mathsf{card}(S)$ denotes the number of elements (cardinality) of the set S, and k denotes a non-negative integer constant. For complexity considerations, note that we represent integer constants in unary notation, so a constant k has the length k as opposed to $\log k$.

This paper considers decision procedures for the validity of formulas whose syntax is given by non-terminal F in Figure 4. The validity of such formulas can be used to show the validity of verification conditions in a programming language. To illustrate this claim, we sketch a weakest-precondition semantics of a guarded-command language in Figure 5. The language contains no procedure calls or loops; these constructs can be transformed into loop-free guarded commands using supplied loop invariants, procedure preconditions and procedure postconditions (see, for example, [58,14,32]). Figure 6 presents the desugaring of field assignment as well as the desugaring of new statement using a global variable alloc denoting the set of currently allocated objects (the desugaring of new assumes that c is the only field of y). Figure 7 shows some key clauses for computing the well-definedness condition of an expression; this condition ensures there are no null dereferences while computing the expression. We present the key cases that check for null on field update and field dereference; the remaining cases simply take the conjunction of the conditions for constituent subexpressions, as illustrated in Figure 7 for the case of $\cup$ operation.

4 Preliminary Observations

We first make several observations on deciding the validity of formulas in the language of Figure 4.

Boolean closure and satisfiability. First note that the language is closed under all boolean operations, which is a desirable property for program specifications [26]. An important consequence of the boolean closure is that the validity problem for our constraints reduces to the satisfiability problem. In the sequel, we will therefore only consider the satisfiability problem.

Propositional structure of constraints. Note further that our constraints are quantifier-free. By a transformation into disjunctive normal form, the satisfiability of constraints reduces to satisfiability of conjunctions of literals A and $\neg A$ where A is given by Figure 4. Algorithmically it is better to avoid the transformation into disjunctive normal form, and view the satisfiability algorithm as a non-deterministic procedure that selects a satisfying assignment to atoms of the quantifier-free formula, and checks that the satisfying assignment corresponds to a satisfiable conjunction of literals [13]. In any case, we reduce satisfiability of constraints to satisfiability of conjunctions of literals.

Translation to unnested form. Note finally that we can transform every conjunction of literals into an equisatisfiable unnested form which contains no nested terms. We transform a formula into unnested form by introducing fresh variables; these fresh variables become existentially quantified, because we are looking at satisfiability. In the resulting unnested form, each atomic formula is of one of the following syntactic forms: $V_O^1 = V_O^2.f_O$, $V_S = V_S^1 \cup V_S^2$, $V_S = V_S^1 \cap V_S^2$, $V_S = V_S^1 \setminus V_S^2$, $V_S = \{V_O^1, \ldots, V_O^n\}$, $V_S = V_O.f_S$, $V_f^1 = V_f^2[V_O^1 \mapsto V_O^2]$, $V_f^1 = V_f^2[V_O \mapsto V_S]$, $V_O^1 = V_O^2$, $V_O \in V_S$, $V_S^1 = V_S^2$, $\mathsf{card}(V_S) \leq k$, $V_f^1 = V_f^2$.

In the sequel we outline decidability of conjunctions of such unnested formulas and their negations. We consider three different methods. We pay most attention to the first method (Section 5). This method is based on a previously well-studied class of formulas; what we found interesting is that this class is applicable to such an expressive constraint language, and that it yields the optimal complexity bound for this class, namely NP. It is interesting to mention the use of Nelson-Oppen combination of theories because it shows that our problem can be naturally decomposed into individual problems each of which can be solved using previously identified theories. Our result also implies that each of these individual theories can be showed decidable using the result of Section 5. Finally, the use of two-variable logic with counting is interesting because it shows how to express some additional constraints that go beyond the language in Figure 4.

5 A Classical Prefix-Vocabulary Class

In this section we outline our first technique for checking satisfiability of conjunctions of unnested literals. This technique is based on the class of universal formulas in first-order logic with a relational signature without function symbols of non-zero arity. We translate conjunctions of literals into equisatisfiable formulas in this class while introducing a constant number of universal quantifiers.

The class $[\exists^*\forall^*]_=$. Define the class $[\exists^*\forall^q]_=$ as the set of all formulas of the form $\exists x_1, \ldots, x_p. \forall y_1, \ldots, y_q.F$ where $p \geq 0$ and F is quantifier-free formula of first-order logic with equality without function symbols. Let $[\exists^*\forall^*]_=$ be the set of formulas $\bigcup_{q \geq 0}[\exists^*\forall^q]_=$. We then have the following two results [4, Page 258].

Fact 5.1 *For any fixed q, satisfiability for $[\exists^*\forall^q]_=$ is in NP. The satisfiability for $[\exists^*\forall^*]_=$ is in NEXPTIME.*

The decision procedure for $[\exists^*\forall^*]_=$ can be based on the small model property and amounts to generating models with at most one element for each existentially quantified variable of a formula and evaluating the formula on those models.

The idea of the translation. The translation of the language in Figure 4 into $[\exists^*\forall^*]_=$ class can be summarized as follows: 1) use unary relations to represent sets, 2) use binary relations to represent object-valued and set-valued fields, and 3) use universal quantifiers to represent set operations. To make this approach work, we need to properly represent null references, eliminate array updates by case analysis, and carefully translate cardinality constraints to avoid introducing an unbounded number of quantifiers.

Axioms for fields. We represent both object-valued and set-valued fields using binary relations. To ensure that object-valued fields are not assigned multiple values simultaneously, for each object-valued field f_O we introduce a conjunct $\forall x, y, z.\ f_O(x, y) \wedge f_O(x, z) \Rightarrow y = z$.

Representing null values. We identify two approaches for representing variables denoting references that can be potentially null. The first approach represents references as sets of cardinality of at most one. In this approach the null reference is therefore an empty set. This approach is used in [47] as well as in the typestate flag analysis plugin of the Hob system [28]. The disadvantage of this approach is that reasoning about sets and relations is generally more difficult than reasoning about elements.

In this paper we examine an alternative approach that retains the dis-

F	$[\![F]\!]$
$V_O^1 = V_O^2.f_O$	$f_O(V_O^1, V_O^2)$
$V_S = V_S^1 \cup V_S^2$	$\forall^+ x.\ V_S(x) \iff V_S^1(x) \lor V_S^2(x)$
$V_S = V_S^1 \cap V_S^2$	$\forall^+ x.\ V_S(x) \iff V_S^1(x) \land V_S^2(x)$
$V_S = V_S^1 \setminus V_S^2$	$\forall^+ x.\ V_S(x) \iff V_S^1(x) \land \neg V_S^2(x)$
$V_S = \{V_O^1, \ldots, V_O^n\}$	$\forall^+ x.\ V_S(x) \iff x = V_O^1 \lor \ldots \lor x = V_O^n$
$V_S = V_O.f_S$	$\forall^+ x.\ V_S(x) \iff f_S(V_O, x)$
$V_O^1 = V_O^2$	$V_O^1 = V_O^2$
$V_O \in V_S$	$V_S(V_O)$
$V_S^1 = V_S^2$	$\forall^+ x.\ V_S^1(x) \iff V_S^2(x)$
$V_f^1 = V_f^2$	$\forall^+ x, y.\ V_f^1(x,y) \iff V_f^2(x,y)$
$V_f^1 = V_f^2[V_O^1 \mapsto V_O^2]$	$\forall^+ x, y.\ V_f^1(x,y) \iff ((x = V_O^1 \land y = V_O^2) \lor$ $(x \neq V_O^1 \land V_f^2(x,y)))$
$V_f^1 = V_f^2[V_O \mapsto V_S]$	$\forall^+ x, y.\ V_f^1(x,y) \iff ((x = V_O \land V_S(y)) \lor$ $(x \neq V_O \land V_S(x,y)))$

Fig. 8. Rules for transforming positive literals into $[\exists^*\forall^*]_=$ fragment

tinction between sets and elements, and uses the constant null to represent null references so that each field is a total function that may potentially have null value. To make the decision procedure for $[\exists^*\forall^*]_=$ applicable to such encoding, we need to overcome the difficulty that is fundamental to the small model property of the $[\exists^*\forall^*]_=$ class: although it is possible to write axioms that constrain binary predicates to have *at most one* value for each argument, it is not possible to constrain them to have *exactly one* value. As a result, the satisfiability will consider models where some object-valued fields are not total. We next make sure that such models do not pose a problem: we transform the formula so that the following holds: if there is a model where some object-valued fields are partial, then there is a *completed* model where these fields have the value null. A completed model replaces the interpretation $[\![f]\!]$ with the interpretation $[\![f]\!] \cup \{(x, \mathsf{null}) \mid \neg\exists y.(x,y) \in [\![f]\!]\}$. Here f denotes an object-valued field, and $[\![f]\!]$ denotes the interpretation of relation symbol f. We first make sure to use only quantifiers that range over non-null objects. We write $\forall^+ x.F$ as a shorthand for $\forall x. x \neq \mathsf{null} \Rightarrow F$. As a result, when x and y are universally quantified, the truth value of an atomic formula $f(x,y)$ is not affected by completion. To ensure that a similar claim holds when one of x, y is an existentially quantified variable, we ensure that for each variable appearing in a binary relation symbol, the conjunction of literals always contains either $x \neq \mathsf{null}$ or $x = \mathsf{null}$. It then suffices to consider the case of literals $f(x,y)$ and $\neg f(x,y)$ such that $y = \mathsf{null}$ occurs as one of the literals. Note that, if $f(x,y)$ holds in the original model, then we know that completed model will still satisfy $f(x,y)$. Therefore, the only problem is the case of literals $\neg f(x,y) \land y = \mathsf{null}$. We therefore transform such conjunction into $f(x,z) \land z \neq \mathsf{null}$ for a fresh (existentially quantified) variable z. The result of the transformation is equivalent on models where f is total. We also introduce the universal axiom $\forall x. \neg f(\mathsf{null}, x)$ to simplify the structure of possible models.

Translating positive literals. Modulo the treatment of null references discussed above, the expected semantics of operations in our language yields translation rules. Figure 8 shows the translation of positive atomic formulas.

Translating negative literals. To translate a negative literal, negate the translation of the underlying atomic formula as in Figure 8 by replacing universal quantifiers with existential quantifiers. Because we are looking at satisfiability, we drop existential quantifiers while making sure that the newly introduced variables are fresh.

Translating cardinality constraints. We translate positive cardinality constraint $\mathsf{card}(V_S) \leq k$ by introducing k fresh constants $a_1, \ldots, a_k$, replacing the constraint with $V_S = \{a_1, \ldots, a_k\}$, and then translating the result as in Figure 8. We translate the negative cardinality constraint $\neg(\mathsf{card}(V_S) \leq k)$, which is equivalent to $\mathsf{card}(V_S) \geq k+1$, by introducing fresh constants $a_1, \ldots, a_k, a_{k+1}$, and replacing the constraint with

$$\bigwedge_{1 \leq i \leq k+1} V_S(a_i) \;\wedge\; \bigwedge_{1 \leq i < j \leq k+1} a_i \neq a_j.$$

Complexity. We next show that satisfiability of formulas in Figure 4 is NP complete. We have carefully constructed our translation so that it introduces a bounded number of quantifiers. Indeed, each conjunct introduces at most three universal quantifiers. By moving these quantifiers to prenex position using the transformation

$$(\forall x, y, z. F_1(x, y, z)) \wedge (\forall x, y, z. F_2(x, y, z)) \;\rightsquigarrow\; \forall x, y, z. (F_1(x, y, z) \wedge F_2(x, y, z))$$

we can write the formula in prenex form $[\exists^*\forall^3]_=$. Because the size of the generated formula is polynomial in the size of the original formula and the time to generate it is polynomial, by Fact 5.1 we conclude that checking the satisfiability of one assignment to unnested atomic formulas is in NP. Unnested form is polynomial in the size of conjunction of literals that specifies an assignment to atomic formulas of a formula F in Figure 4, and picking an assignment to atomic formulas can be done in NP. By composing these two non-deterministic choices, we obtain an NP decision procedure for satisfiability of expressions. NP-hardness follows trivially because our language subsumes propositional logic. We conclude that the satisfiability of formulas in Figure 4 is NP-complete.

Remarks on related work. Fragments of first-order logics based on quantifier prefixes are systematized in [4, 17] where the $[\exists^*\forall^*]_=$ class is described as Bernays-Schönfinkel-Ramsey class. Finite model finding tools such as Alloy [22], MACE [35] and Paradox [7] can therefore be used to check satisfiability of such formulas. Resolution techniques [46] are also complete for this class because the term model is finite.

Symbolic formulations of shape analysis such as [56, 55, 26, 53] can be adapted to work with our specification language, so our decision procedure can be used a component of a shape analysis. The idea of exploiting the parameterized complexity of $[\exists^*\forall^p]_=$ seems more generally useful. For example, we can use it to obtain the fact that Boolean shape analysis constraints [26] are NP-complete: it suffices to non-deterministically pick a satisfiable quantified formula, and then observe that each of the quantified conjuncts is in $[\exists^*\forall^2]_=$. [26] was influenced by [56], which points out the importance of $[\exists^*\forall^*]_=$ fragment itself [56, Section 3.4, Page 20]. [21] studies the extensions of the $[\exists^*\forall^*]_=$ fragment with transitive closure and shows several decidability and undecidability results that delineate the boundary between decidable and undecidable extensions of $[\exists^*\forall^*]_=$. Decidable extensions of $[\exists^*\forall^*]_=$ fragment are useful for shape analysis of recursive structures. Nevertheless, by encapsulating recursive data structures and specifying them using sets, even logics without transitive closure can be useful in establishing high-level properties of programs [25, 30, 27], following the idea that different levels of abstraction require different reasoning techniques [29, 19].

6 Two-Variable Logics

In this section we show that two-variable logic with counting, denoted C^2, can be used to decide constraints in Figure 4, as well as some useful extensions of these constraints. We consider the satisfiability problem and use a language containing any number of constants, unary relation symbols, and binary relation symbols.

Two-variable logics. The logic C^2 is a first-order logic 1) extended with counting quantifiers $\exists^{\geq k}x.F(x)$, saying that there are at least k elements x satisfying formula $F(x)$ for some constant k, and 2) restricted to allow only two variable names x, y in formulas. Note that the variables x and y may be reused via quantifier nesting, and that formulas of the form $\exists^{=k}x.\, F(x)$ and $\exists^{\leq k}x.\, F(x)$ are expressible as boolean combination of formulas of the form $\exists^{\geq k}x.\, F(x)$. The logic C^2 was shown decidable in [18] and the complexity for the C^2_1 fragment of C^2 (with counting up to one) was established in [43]. Two-variable logic without counting L^2 was known to be decidable previously due to a finite model property [38], in fact there is a doubly exponential bound on the size of the finite model. On the other hand, two-variable logic with counting does not have a finite model property [18]. The usefulness of two-variable logic with counting for reasoning about relations between objects was identified in [25, 27] and its use for encoding description logics can be found in e.g. [5, 1].

Encoding into two-variable logic with counting. We next explain how to encode the constraints in Figure 4 into two-variable logic with counting. It turns out that most of the ideas of the encoding in Section 5 apply to encoding using two-variable logic as well, because we only use at most two universal quantifiers in Figure 8, and the existentially quantified variables simply become constants in the language. To avoid using three variables to express the fact that some relations are functions, we replace $\forall x, y, z.\ f_O(x, y) \land f_O(x, z) \Rightarrow y = z$ with $\forall x.\exists^{\leq 1}y.f(x, y)$. Finally, we can even express the cardinality constraints directly by replacing $\mathsf{card}(S) \leq k$ with $\exists^{\leq k}x.S(x)$.

The additional expressive power of two-variable logic comes from expressing the constructs of the form $\forall x.\exists y.f(x, y)$. Such constructs allow us to state non-null properties of objects, which are important for reasoning about initialization of objects in object-oriented programming languages [12]. Moreover, counting quantifiers can naturally express high-level application constraints identified in the database community and object-oriented modelling community as referential integrity, cardinality constraints, as well as role constraints [27].

7 Nelson-Oppen Combination

We next note that satisfiability of formulas in Figure 4 can be decided using a multi-sorted Nelson-Oppen decision procedure that combines three individual decision procedures 1) two-level syllogistic expressed as a component Nelson-Oppen procedure as discussed in [57] 2) uninterpreted function symbols [41] in multisorted language with function symbols whose result sort can be a set sort, and 3) extensional theory of arrays [40, 49]. Because an equivalence class on shared variables in Nelson-Oppen procedure can be guessed using a nondeterministic polynomial algorithm, and each individual decision procedure is in NP, we conclude that a Nelson-Oppen combination decision procedure for our language is also in NP.

Note that we can use Nelson-Oppen combination in conjunction with the techniques presented in Section 5 and Section 6 because Nelson-Oppen method allows quantifier-free combinations of formulas that themselves need not be quantifier-free. The approach based on decomposing the language of Figure 4 into smaller Nelson-Oppen theories has the advantage of using previously understood and efficient decision procedures that may be useful in other contexts. Moreover, no special encodings are necessary because the use of sorts naturally decomposes constraints into the constraints of individual decidable theories.

8 Conclusions

We have outlined a range of possible techniques for solving constraints on set-valued fields: the use of $[\exists^*\forall^*]_=$ class of first-order logic, the use of two-variable logic with counting, and the use of Nelson-Oppen combination of decision procedures. In addition to these techniques, it may be possible to use general-purpose theorem provers on formulas that are of interest to us [10, 51, 52]. We are currently examining the structure of constraints generated by the Hob system [29] to evaluate the effectiveness of different decision procedures.

Acknowledgements. We thank Darko Marinov for useful comments on an earlier version of this paper and useful discussions about the use of sets in symbolic execution.

References

[1] Baader, F., D. Calvanese, D. McGuinness, D. Nardi and P. Patel-Schneider, editors, "The Description Logic Handbook: Theory, Implementation and Applications," Cambridge University Press, 2003.

[2] Barnett, M., R. DeLine, M. Fähndrich, K. R. M. Leino and W. Schulte, *Verification of object-oriented programs with invariants*, Journal of Object Technology **3** (2004), pp. 27–56.

[3] Barnett, M., K. R. M. Leino and W. Schulte, *The Spec# programming system: An overview*, in: *CASSIS 2004: International Workshop on Construction and Analysis of Safe, Secure and Interoperable Smart devices*, 2004.

[4] Börger, E., E. Grädel and Y. Gurevich, "The Classical Decision Problem," Springer-Verlag, 1997.

[5] Borgida, A., *Description logics in data management*, Artificial Intelligence **82** (1996), pp. 353–367.

[6] Boyapati, C., R. Lee and M. C. Rinard, *A type system for preventing data races and deadlocks*, in: *Proc. 17th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2002.

[7] Claessen, K. and N. Sörensson, *New techniques that improve MACE-style model finding*, in: *Model Computation*, 2003.

[8] Clarke, D. G., J. M. Potter and J. Noble, *Ownership types for flexible alias protection*, in: *Proc. 13th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 1998.

[9] Cousot, P., *Automatic verification by abstract interpretation, invited tutorial*, in: L. Zuck, P. Attie, A. Cortesi and S. Mukhopadhyay, editors, *Proc. 4th International Conference on Verification, Model Checking and Abstract Interpretation* (2003), pp. 20–24.

[10] Detlefs, D., G. Nelson and J. B. Saxe, *Simplify: A theorem prover for program checking*, Technical Report HPL-2003-148, HP Laboratories Palo Alto (2003).

[11] Deutsch, A., *Interprocedural may-alias analysis for pointers: Beyond k-limiting*, in: *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation* (1994), pp. 230–241.

[12] Fähndrich, M. and K. R. M. Leino, *Declaring and checking non-null types in an object-oriented language*, in: *OOPSLA'03*, 2003.

[13] Flanagan, C., R. Joshi, X. Ou and J. B. Saxe, *Theorem proving using lazy proof explication*, in: *CAV*, 2003, pp. 355–367.

[14] Flanagan, C. and J. B. Saxe, *Avoiding exponential explosion: Generating compact verification conditions*, in: *Proc. 28th ACM POPL*, 2001.

[15] Fradet, P. and D. L. Métayer, *Shape types*, in: *Proc. 24th ACM POPL*, 1997.

[16] Ghiya, R. and L. Hendren, *Is it a tree, a DAG, or a cyclic graph?*, in: *Proc. 23rd ACM POPL*, 1996.

[17] Grädel, E., *Decidable fragments of first-order and fixed-point logic. From prefix-vocabulary classes to guarded logics*, in: *Proceedings of Kalmár Workshop on Logic and Computer Science, Szeged*, 2003.

[18] Grädel, E., M. Otto and E. Rosen, *Two-variable logic with counting is decidable*, in: *Proceedings of 12th IEEE Symposium on Logic in Computer Science LICS '97, Warschau*, 1997.

[19] Guttag, J. V., *Abstract data types, then and now*, in: *Software Pioneers: Contributions to Software Engineering*, Springer, 2002 pp. 442–452.

[20] Hirzel, M., A. Diwan and M. Hind, *Pointer analysis in the presence of dynamic class loading*, in: *ECOOP*, 2004.

[21] Immerman, N., A. M. Rabinovich, T. W. Reps, S. Sagiv and G. Yorsh, *The boundary between decidability and undecidability for transitive-closure logics.*, in: *CSL*, 2004, pp. 160–174.

[22] Jackson, D., *Alloy: a lightweight object modelling notation*, ACM TOSEM **11** (2002), pp. 256–290.

[23] Jensen, J. L., M. E. Jørgensen, N. Klarlund and M. I. Schwartzbach, *Automatic verification of pointer programs using monadic second order logic*, in: *Proc. ACM PLDI*, Las Vegas, NV, 1997.

[24] Kozen, D., *Complexity of boolean algebras*, Theoretical Computer Science **10** (1980), pp. 221–247.

[25] Kuncak, V. and M. Rinard, *On role logic*, Technical Report 925, MIT CSAIL (2003).

[26] Kuncak, V. and M. Rinard, *Boolean algebra of shape analysis constraints*, in: *Proc. 5th International Conference on Verification, Model Checking and Abstract Interpretation*, 2004.

[27] Kuncak, V. and M. Rinard, *Generalized records and spatial conjunction in role logic*, in: *11th Annual International Static Analysis Symposium (SAS'04)*, Verona, Italy, 2004.

[28] Lam, P., V. Kuncak and M. Rinard, *Generalized typestate checking using set interfaces and pluggable analyses*, SIGPLAN Notices **39** (2004), pp. 46–55.

[29] Lam, P., V. Kuncak and M. Rinard, *On our experience with modular pluggable analyses*, Technical Report 965, MIT CSAIL (2004).

[30] Lam, P., V. Kuncak and M. Rinard, *Generalized typestate checking for data structure consistency*, in: *6th International Conference on Verification, Model Checking and Abstract Interpretation*, 2005.

[31] Lam, P., V. Kuncak, K. Zee and M. Rinard, *The Hob project web page*, http://catfish.csail.mit.edu/~plam/hob/ (2004).

[32] Leino, K. R. M., *Efficient weakest preconditions*, Information Processing Letters (to appear).

[33] Liskov, B., R. Atkinson, T. Bloom, E. Moss, C. Schaffert, B. Scheifler and A. Snyder, *Clu reference manual*, Technical Report 225, MIT LCS (1979).

[34] Logozzo, F., *Separate compositional analysis of class-based object-oriented languages*, in: *Proceedings of the 10th International Conference on Algebraic Methodology And Software Technology (AMAST'2004)*, Lectures Notes in Computer Science **3116** (2004), pp. 332–346.

[35] McCune, W., *MACE 2.0 Reference Manual and Guide*, ArXiv Computer Science e-prints (2001).

[36] Milner, R., M. Tofte, R. Harper and D. MacQueen, "The Definition of Standard ML (Revised)," The MIT Press, Cambridge, Mass., 1997.

[37] Møller, A. and M. I. Schwartzbach, *The Pointer Assertion Logic Engine*, in: *Proc. ACM PLDI*, 2001.

[38] Mortimer, M., *On languages with two variables*, Zeitschr. für math. Logik und Grundlagen der Math. **21** (1975), pp. 135–140.

[39] Nelson, G., editor, "Systems Programming with Modula-3," Prentice Hall, 1991.

[40] Nelson, G. and D. C. Oppen, *Simplification by cooperating decision procedures*, ACM Trans. Program. Lang. Syst. **1** (1979), pp. 245–257.

[41] Nelson, G. and D. C. Oppen, *Fast decision procedures based on congruence closure*, Journal of the ACM (JACM) **27** (1980), pp. 356–364.

[42] Noble, J., J. Vitek and J. Potter, *Flexible alias protection*, in: *Proc. 12th ECOOP*, 1998.

[43] Pacholski, L., W. Szwast and L. Tendera, *Complexity results for first-order two-variable logic with counting*, SIAM J. on Computing **29** (2000), pp. 1083–1117.

[44] Pollet, I., B. L. Charlier and A. Cortesi, *Distinctness and sharing domains for static analysis of java programs*, in: *ECOOP*, 2001.

[45] Ramalingam, G., A. Warshavsky, J. Field, D. Goyal and M. Sagiv, *Deriving specialized program analyses for certifying component-client conformance*, in: *Proceeding of the ACM SIGPLAN 2002 Conference on Programming language design and implementation* (2002), pp. 83–94.

[46] Robinson, A. and A. Voronkov, editors, "Handbook of Automated Reasoning (Volume 1)," Elsevier and The MIT Press, 2001.

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

[48] Skolem, T., *Untersuchungen über die Axiome des Klassenkalküls and über "Produktations- und Summationsprobleme", welche gewisse Klassen von Aussagen betreffen*, Skrifter utgit av Vidnskapsselskapet i Kristiania, I. klasse, no. 3, Oslo (1919).

[49] Stump, A., C. W. Barrett, D. L. Dill and J. R. Levitt, *A decision procedure for an extensional theory of arrays.*, in: *LICS*, 2001, pp. 29–37.

[50] Tip, F. and J. Palsberg, *Scalable propagation-based call graph construction algorithms*, in: *Proc. 15th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2000, pp. 281–293.

[51] Voronkov, A. et al., *The anatomy of Vampire (implementing bottom-up procedures with code trees)*, Journal of Automated Reasoning **15** (1995), pp. 237–265.

[52] Weidenbach, C., *Combining superposition, sorts and splitting*, , **II**, Elsevier Science, 2001 pp. 1965–2013.

[53] Wies, T., "Symbolic Shape Analysis," Master's thesis, Max-Planck Instutut für Informatik (2004).

[54] Wirth, N., "Programming in Modula-2," Springer, 1982.

[55] Yorsh, G., "Logical Characterizations of Heap Abstractions," Master's thesis, Tel-Aviv University (2003).

[56] Yorsh, G., T. Reps and M. Sagiv, *Symbolically computing most-precise abstract operations for shape analysis*, in: *10th TACAS*, 2004.

[57] Zarba, C. G., *Combining sets with elements*, in: N. Dershowitz, editor, *Verification: Theory and Practice*, Lecture Notes in Computer Science **2772** (2004), pp. 762–782.

[58] Zee, K., P. Lam, V. Kuncak and M. Rinard, *Combining theorem proving with static analysis for data structure consistency*, in: *International Workshop on Software Verification and Validation (SVV 2004)*, Seattle, 2004.