On Field Constraint Analysis

Thomas Wies[1], Viktor Kuncak[2],
Patrick Lam[2], Andreas Podelski[1], and Martin Rinard[2]

[1] Max-Planck-Institut für Informatik, Saarbrücken, Germany
{wies,podelski}@mpi-inf.mpg.de
[2] MIT Computer Science and Artificial Intelligence Lab, Cambridge, USA
{vkuncak,plam,rinard}@csail.mit.edu

Abstract. We introduce *field constraint analysis*, a new technique for verifying data structure invariants. A field constraint for a field is a formula specifying a set of objects to which the field can point. Field constraints enable the application of decidable logics to data structures which were originally beyond the scope of these logics, by verifying the backbone of the data structure and then verifying constraints on fields that cross-cut the backbone in arbitrary ways. Previously, such cross-cutting fields could only be verified when they were uniquely determined by the backbone, which significantly limited the range of analyzable data structures.

Our field constraint analysis permits *non-deterministic* field constraints on cross-cutting fields, which allows to verify invariants of data structures such as skip lists. Non-deterministic field constraints also enable the verification of invariants between data structures, yielding an expressive generalization of static type declarations.

The generality of our field constraints requires new techniques, which are orthogonal to the traditional use of structure simulation. We present one such technique and prove its soundness. We have implemented this technique as part of a symbolic shape analysis deployed in the context of the Hob system for verifying data structure consistency. Using this implementation we were able to verify data structures that were previously beyond the reach of similar techniques.

1 Introduction

The goal of shape analysis [29, Chapter 4], [2, 4–6, 24, 27, 28, 35] is to verify complex consistency properties of linked data structures. The verification of such properties is important in itself, because the correct execution of the program often requires data structure consistency. In addition, the information computed by shape analysis is important for verifying other program properties in programs with dynamic memory allocation.

Shape analyses based on expressive decidable logics [12, 14, 28] are interesting for several reasons. First, the correctness of such analyses is easier to establish than for approaches based on ad-hoc representations; the use of a decidable logic separates the problem of generating constraints that imply program properties from the problem of solving these constraints. Next, such analyses can be used in the context of assume-guarantee reasoning because logics provide a language for specifying the behaviors of code fragments. Finally, the decidability of logics leads to completeness properties for these analyses, eliminating false alarms and making the analyses easier to interact with. We were able to confirm these observations in the context of Hob system [16, 22] for analyzing data structure consistency, where we have integrated one such tool [28] with other analyses, allowing us to use shape analysis in the context of larger programs: in particular, Hob enabled us to leverage the power of shape analysis, while avoiding the associated performance penalty, by applying shape analysis only to those parts of the program where its extreme precision is necessary.

Our experience with such analyses has also taught us that some of the techniques that make these analyses predictable also make them inapplicable to many useful data structures. Among the most striking examples is the restriction on pointer fields in the Pointer Assertion Logic Engine [28]. This restriction states that all fields of the data structure that are not part of the data structure's tree backbone must be functionally determined by the backbone; that is, such fields must be specified by a formula that uniquely determines where they point to. Formally, we have

$$\forall x\ y.\ f(x){=}y \ \leftrightarrow\ F(x,y) \tag{1}$$

where f is a function representing the field, and F is the defining formula for f. The restriction that F is functional means that, although data structures such as doubly linked lists with backward pointers can be verified, many other data structures remain beyond the scope of the analysis. This includes data structures where the exact value of pointer fields depends on the history of data structure operations, and data structures that use randomness to achieve good average-case performance, such as skip lists [33]. In such cases, the invariant on the pointer field does not uniquely determine where the field points to, but merely gives a constraint on the field, of the form

$$\forall x\ y.\ f(x){=}y \ \rightarrow\ F(x,y) \tag{2}$$

This constraint is equivalent to $\forall x.\ F(x, f(x))$, which states that the function f is a solution of a given binary predicate. The motivation for this paper is to find a technique that supports reasoning about constraints of this, more general, form. In a search for existing approaches, we have considered structure simulation [9, 11], which, intuitively, allows richer logics to be embedded into existing logics that are known to be decidable, and of which [28] can be viewed as a specific instance. Unfortunately, even the general structure simulation requires definitions of the form $\forall x\ y.\ r(x,y) \ \leftrightarrow\ F(x,y)$ where $r(x,y)$ is the relation being simulated. When the relation $r(x,y)$ is a function, which is the case with most reference fields in programming languages, structure simulation implies the same restriction on the functionality of the defining relation. To handle the general case, an alternative approach therefore appears to be necessary.

Field constraint analysis. This paper presents field constraint analysis, our approach for analyzing fields with general constraints of the form (2). Field constraint analysis is a proper generalization of the existing approach and reduces to it when the constraint formula F is functional. It is based on approximating the occurrences of f with F, taking into account the polarity of f, and is always sound. It is expressive enough to verify constraints on pointers in data structures such as two-level skip lists. The applicability of our field constraint analysis to non-deterministic field constraints is important because many complex properties have useful non-deterministic approximations. Yet despite this fundamentally approximate nature of field constraints, we were able to prove its completeness for some important special cases. Field constraint analysis naturally combines with structure simulation, as well as with a symbolic approach to shape analysis [32, 36]. Our presentation and current implementation are in the context of the monadic second-order logic (MSOL) of trees [13], but our results extend to other logics. We therefore view field constraint analysis as a useful component of shape analysis approaches that makes shape analysis applicable to a wider range of data structures.

Contributions. This paper makes the following contributions:

- We introduce an **algorithm** (Figure 12) that uses field constraints to eliminate derived fields from verification conditions.
- We prove that the algorithm is both **sound** (Theorem 1) and, in certain cases, **complete**. The completeness applies not only to deterministic fields (Theorem 2), but also to the preservation of field constraints themselves over loop-free code (Theorem 3). The last result implies a complete technique for checking that field constraints hold, if the programmer adheres to a discipline of maintaining them e.g. at the beginning of each loop.
- We describe how to combine our algorithm with symbolic shape analysis [36] to **infer loop invariants**.
- We describe an **implementation** and experience in the context of the Hob system for verifying data structure consistency.

The implementation of field constraint analysis as part of the Hob system [16,22] allows us to apply the analysis to modules of larger applications, with other modules analyzed by more scalable analyses, such as typestate analysis [21].

2 Examples

We next explain our field constraint analysis with a set of examples. The doubly-linked list example shows that our analysis handles, as a special case, the ubiquitous back pointers of data structures. The skip list example shows how field constraint analysis handles non-deterministic field constraints on derived fields, and how it can infer loop invariants. Finally, the students example illustrates inter-data-structure constraints, which are simple but useful for high-level application properties.

2.1 Doubly-Linked List with an Iterator

This section presents a doubly-linked list with a built-in iterator. It illustrates the usefulness of field constraints for specifying pointers that form doubly-linked structures, and introduces the language we use for writing implementations and specifications in the Hob system [22, 23].

```
impl module DLLIter {
  format Entry { next : Entry; prev : Entry; }

  var root, current : Entry;
  proc remove(n : Entry) {
    if (n==current) { current = current.next; }
    if (n==root) { root = root.next; }
    if (n.prev != null) { n.prev.next = n.next; }
    if (n.next != null) { n.next.prev = n.prev; }
    n.next = null; n.prev = null;
  }
}
```

Fig. 1. Iterable list implementation section, containing standard imperative code

Our doubly-linked list implementation is a global data structure with operations `add` and `remove` that insert and remove elements from the list, as well as the operations `initIter`, `nextIter`, and `lastIter` for manipulating the iterator built into the list. We have verified all these operations using our system; we here present only the `remove` operation. Our list data structure is implemented in the form of a Hob module. A module consists of an implementation section, which suffices to execute the module (Figure 1), a specification section, which suffices for abstract reasoning about the behavior of the module (Figures 2), and an abstraction section, which connects implementations and specifications by defining the abstraction function and representation invariants (Figure 3). As Figure 1 shows, we implement the doubly-linked list with two private fields, `next` and `prev` that apply to type (format) `Entry`, the private `root` variable of the doubly-linked list, and the private `current` variable that indicates the position of the iterator in the list. The specification section in Figure 2 specifies the behavior of the operation `remove` using two sets: `Content`, which contains the set of elements in the list, and `Iter`, which specifies the set of elements that remain to be iterated over. These two sets abstractly characterize the behavior of operations, allowing the clients to soundly reason about the hidden implementation of the list. This reasoning is sound because our analysis verifies that the implementation conforms to the specification, using the definitions of sets `Content` and `Iter` in Figure 3. These definitions are expressed in a subset of Isabelle [31] formulas that can be translated into monadic second-order logic [13]. The module defines `Content` as the set of all objects reachable from `root` and `Iter` as the set of all objects reachable from `current`. Function `rtrancl` is a higher-order function that accepts a binary predicate on objects and returns the reflexive transitive closure of this predicate. The abstraction section in Figure 3 also contains module representation invariants; our system ensures that these invariants are maintained by each operation in the module. The first invariant is a global invariant saying that no `next` fields point to the root of the list. The second invariant is recognized by our analysis as a field constraint on the field `prev`. This invariant indicates to the system that `prev` is a derived field. The `next` field has no field constraints, so our system treats it as a backbone field.

```
spec module DLLIter {
  format Entry;
  specvar Content, Iter : Entry set;

  invariant Iter in Content;

  proc remove(n : Entry)
    requires card(n)=1 & (n in Content);
    modifies Content, Iter
    ensures (Content' = Content - n) &
            (Iter' = Iter - n);
}
```

Fig. 2. Iterable list specification section, containing procedure interfaces

Our system verifies that the `remove` procedure implementation in Figure 1 conforms to its procedure contract in Figure 2 as follows. The system expands the modifies clause into a frame condition, which it conjoins with the ensures clause. Next, it conjoins the public set-based invariant $\texttt{Iter} \subseteq \texttt{Content}$ to both the requires and ensures clause. The resulting pre- and postcondition are expressed in terms of the sets `Content` and `Iter`, so the system applies the definitions of the sets in Figure 3 to obtain pre and postcondition expressed in terms of `next` and `prev`, and conjoins the first invariant from Figure 3 to both pre and postcondition. It then uses standard weakest precondition computation [1] to generate a verification condition that captures the correctness of `remove`.

To decide the resulting verification condition, our system treats the field constraint invariant specially: it exploits the fact that `next` is a backbone field and `prev` is a field given by a field constraint to reduce the verification condition to one expressible using only the `next` field. (This elimination is given by the algorithm in Figure 12.) Because `next` fields form a tree, the system can decide the verification condition using monadic second-order logic on trees [13]. To ensure the soundness of this approach, the system also verifies that the structure remains a tree after each operation.

We note that our first implementation of the doubly-linked list with an iterator was verified using a Hob plugin [20] that relies on Pointer Assertion Logic Engine tool [28]. While verifying the initial version of the list module, we discovered an error in `remove`: the first line of `remove` procedure in Figure 1 was not present, resulting in violation of the specification of `remove` in the special case when the element being removed is the next element to iterate over. What distinguishes our system from the previous Hob analysis based on PALE is the ability to handle the cases where the field constraints are non-deterministic. We illustrate such cases in the examples that follow. Additionally, we show how our new analysis synthesizes loop invariants using symbolic shape analysis [36].

```
abst module DLLIter {
  use plugin "Bohne decaf";

  Content = {x : Node | "rtrancl (% v1 v2 . next v1 = v2) root x"};
  Iter = {x : Node | "rtrancl (% v1 v2 . next v1 = v2) current x"};

  invariant "ALL x . root ~= null --> next x ~= root";
  invariant "ALL x y. prev x = y -->
    (x = root  --> y = null) &
    (x ~= root & (rtrancl (% v1 v2. next v1 = v2) root x)) --> next y = x";
}
```

Fig. 3. Iterable list abstraction section, containing abstraction function and invariants

2.2 Skip List

We next present the analysis of a two-level skip list. Skip lists [33] support logarithmic average-time access to elements by augmenting a linked list with sublists that skip over some of the elements in the list. The two-level skip list is a simplified implementation of a skip list, which has only two levels: the list containing all elements, and a sublist of this list. Figure 4 presents an example two-level skip list. Our implementation uses the `next` field to represent the main list, which forms the backbone of the data structure, and uses the derived `nextSub` field to represent a sublist of the main list. We focus on the `add` procedure, which inserts an element into an appropriate position in the skip list. Figure 5 presents the implementation of `add`, which first searches through `nextSub` links to get an estimate of the position of the entry, then finds the entry by searching through `next` links, and inserts the element into the main `next`-linked list. Optionally, the procedure also inserts the element into `nextSub` list, which is modelled using a non-deterministic choice in our language and is an abstraction of the insertion with certain probability in the original implementation. Figure 6 presents a specification for add, which indicates that add always inserts the element into the set of elements stored in the list. Figure 7 presents the abstraction section for the two-level skip list. This section defines the abstract set S as the set of nodes reachable from root.next, indicating that root is used as a header node. The abstraction section contains three invariants. The first invariant is the field constraint on the field nextSub, which defines it as a derived field.

Note that the constraint for this derived field is non-deterministic, because it only states that if x.nextSub==y, then there exists a path of length at least one from x to y along next fields, without indicating where nextSub points. Indeed, the simplicity of the skip list implementation stems from the fact that the position of nextSub is not uniquely given by next; it depends not only on the history of invocations, but also on the random number generator used to decide when to introduce new nextSub links. The ability to support such non-deterministic constraints is what distinguishes our approach from approaches that can only handle deterministic fields.

The last two invariants indicate that root is never null (assuming, for simplicity of the example, that the state is initialized), and that all objects not reachable from root are isolated: they have no incoming or outgoing next pointers. These two invariants allow the analysis to conclude that the object referenced by e in add(e) is not referenced by any node, which, together with the precondition not(e in S), allows our analysis to prove that objects remain in an acyclic list along the next field.[3]

Our analysis successfully verifies that add preserves all invariants, including the non-deterministic field constraint on nextSub. While doing so, the analysis takes advantage of these invariants as well, as is usual in assume/guarantee reasoning. In this example, the analysis is able to infer the loop invariants in add. The analysis constructs these loop invariants as disjunctions of universally quantified boolean combinations of the unary predicates that correspond to the sets of elements supplied for add in the abstraction section, using symbolic shape analysis [32, 36].

2.3 Students and Schools

Our next example illustrates the power of non-deterministic field constraints. This example contains two linked lists: one containing students and one containing schools. Each Elem object may represent either a student or a school; students have a pointer to the school which they attend. Both students and schools use the next backbone pointer to indicate the next student or school in the relevant linked list.

Figures 8 and 10 present the interface and implementation of our students example. The addStudent procedure adds a student to the student list and associates it with a school that is supposed to be already contained in the school data structure. The procedure may assume that the relevant data structure invariants (described below) hold upon entry, but must guarantee that they hold upon exit, if the stated postcondition is to make any sense at all.

Figure 9 presents the abstraction section for our module. ST denotes all students, that is, all Elem objects reachable from the root students reference through next

[3] The analysis still needs to know that e is not identical to the header node. In this example we have used an explicit (assume "e $\neq$ root") statement to supply this information. Such assume statements can be automatically generated if the developer specifies the set of representation objects of a data structure, but this is orthogonal to field constraint analysis itself.

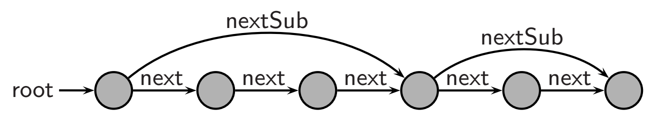

Fig. 4. An instance of a two-level skip list

```
impl module Skiplist {
  format Entry {
    v : int;
    next, nextSub : Entry;
  }
  var root : Entry;

  proc add(e:Entry) {
    assume "e ~= root";
    int v = e.v;
    Entry sprev = root, scurrent = root.nextSub;
    while ((scurrent != null) && (scurrent.v < v)) {
      sprev = scurrent; scurrent = scurrent.nextSub;
    }
    Entry prev = sprev, current = sprev.next;
    while ((current != scurrent) && (current.v < v)) {
      prev = current; current = current.next;
    }
    e.next = current; prev.next = e;
    choice { sprev.nextSub = e; e.nextSub = scurrent; }
         | { e.nextSub = null; }
}
```

Fig. 5. Skip list implementation

```
spec module Skiplist {
  format Entry;
  specvar S : Entry set;

  proc add(e:Entry)
    requires card(e) = 1 & not (e in S)
    modifies S
    ensures S' = S + e';
}
```

Fig. 6. Skip list specification

```
abst module Skiplist {
  use plugin "Bohne";

  S = {x : Entry | "rtrancl (% v1 v2. next v1 = v2) (next root) x"};
  invariant "ALL x y. (nextSub x = y) --> ((x = null --> y = null) &
                  (x ~= null --> rtrancl (% v1 v2. next v1 = v2) (next x) y))";
  invariant "root ~= null";
  invariant "ALL x. x ~= null &
                ~(rtrancl (% v1 v2. next v1 = v2) root x) -->
                ~(EX y. y ~= null & next y = x) & (next x = null)";
  proc add {
    has_pred = {x : Entry | "EX y. next y = x"};
    r_current = {x : Entry | "rtrancl (% v1 v2. next v1 = v2) current x"};
    r_scurrent = {x : Entry | "rtrancl (% v1 v2. next v1 = v2) scurrent x"};
    r_sprev = {x : Entry | "rtrancl (% v1 v2. next v1 = v2) sprev x"};
    next_null = {x : Entry | "next x = null"};
    sprev_nextSub = {x : Entry | "nextSub sprev = scurrent"};
    prev_next = {x : Entry | "next prev = current"};
  }
}
```

Fig. 7. Skip list abstraction (including invariants)

fields. SC denotes all schools, that is, all Elem objects reachable from schools. The abstraction section then gives three module invariants. The first two module invariants state disjointness properties: no objects are shared between ST and SC (if an object is reachable from schools through next fields, then it is not reachable from students through next fields, and vice-versa). The third module invariant states that if an object x is not in either ST or SC, then its next field is set to null, and no object points to x. Combined, these invariants guarantee the well-formedness of the schools and students linked lists.

The abstraction section also gives a field constraint on the attends field. Section 3 describes how we verify the validity of the non-deterministic constraint on the attends field. In particular, our analysis can successfully verify the property that for any student, attends points to some (undetermined) element of the SC set of schools. Note that this goes beyond the power of previous analyses, which required the identity of the school pointed to by the student be functionally determined by the identity of the student. The example therefore illustrates how our analysis eliminates a key restriction of previous approaches—certain data structures exhibit properties that the logics in previous approaches were not expressive enough to capture. In general, previous approaches could express and verify properties that were, in some sense, more restrictive than the properties of many data structures that we would like to implement. Because our analysis supports properties that express the correct level of partial information (for example, that a field points to some undetermined object within a set of objects), it is able to successfully analyze these kinds of data structures.

```
spec module Students {
  format Elem;
  specvar ST : Elem set;
  specvar SC : Elem set;

  proc addStudent(st:Elem; sc:Elem)
    requires card(st)=1 & card(sc)=1 & (sc in SC) &
             (not (st in ST)) & (not (st in SC))
    modifies ST
    ensures ST' = ST + st;
}
```

Fig. 8. Specification for students example

```
abst module Students {
  use plugin "Bohne decaf";

  ST = { x : Elem | "rtrancl (% v1 v2. next v1 = v2) students x" };
  SC = { x : Elem | "rtrancl (% v1 v2. next v1 = v2) schools x" };

  ...

  invariant "ALL x y. (attends x = y) -->
              (x ~= null -->
              ((~(rtrancl (% v1 v2. next v1 = v2) students x) --> y = null) &
               ((rtrancl (% v1 v2. next v1 = v2) students x) -->
                (rtrancl (% v1 v2. next v1 = v2) schools y))))";
}
```

Fig. 9. Abstraction for students example

```
impl module Students {
  format Elem {
    attends : Elem;
    next : Elem;
  }
  var students : Elem;
  var schools  : Elem;

  proc addStudent(st:Elem; sc:Elem) {
    st.attends = sc;
    st.next = students;
    students = st;
  }
}
```

Fig. 10. Implementation for students example

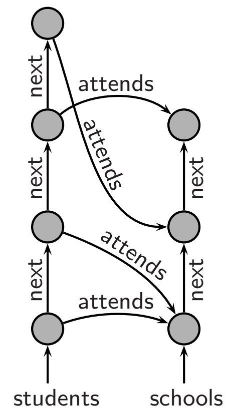

Fig. 11. Students data structure instance

3 Field Constraint Analysis

This section presents the field constraint analysis algorithm and proves its soundness as well as, for some important cases, completeness.

We consider a logic $\mathcal{L}$ over a signature Σ where Σ consists of unary function symbols $f \in \mathsf{Fld}$ corresponding to fields in data structures and constant symbols $c \in \mathsf{Var}$ corresponding to program variables. We use monadic second-order logic (MSOL) over trees as our working example, but in general we only require $\mathcal{L}$ to support conjunction, implication and equality reasoning.

A Σ-structure S is a first-order interpretation of symbols in Σ. For a formula F in $\mathcal{L}$, we denote by $\mathsf{Fields}(F) \subseteq \Sigma$ the set of all fields occurring in F.

We assume that $\mathcal{L}$ is decidable over some set of well-formed structures and we assume that this set of structures is expressible by a formula I in $\mathcal{L}$. We call I the *simulation invariant* [11]. For simplicity, we consider the simulation itself to be given by the restriction of a structure to the fields in $\mathsf{Fields}(I)$, i.e. we assume that there exists a decision procedure for checking validity of implications of the form $I \;\rightarrow\; F$ where F is a formula such that $\mathsf{Fields}(F) \subseteq \mathsf{Fields}(I)$. In our running example, MSOL, the simulation invariant I states that the fields in $\mathsf{Fields}(I)$ span a forest.

We call a field $f \in \mathsf{Fields}(I)$ a *backbone field*, and call a field $f \in \mathsf{Fld} \setminus \mathsf{Fields}(I)$ a *derived field*. We refer to the decision procedure for formulas with fields in $\mathsf{Fields}(I)$ over the set of structures defined by the simulation invariant I as *the underlying decision procedure*. Field constraint analysis enables the use of the underlying decision procedure to reason about non-deterministically constrained derived fields. We state invariants on the derived fields using field constraints.

Definition 1 (Field constraints on derived fields). *A field constraint D_f for a simulation invariant I and a derived field f is a formula of the form*

$$\mathsf{D}_f \;\equiv\; \forall x\, y.\; f(x) = y \;\rightarrow\; \mathsf{FC}_f(x, y)$$

where FC_f is a formula with two free variables such that (1) $\mathsf{Fields}(\mathsf{FC}_f) \subseteq \mathsf{Fields}(I)$, and (2) FC_f is total with respect to I, i.e. $I \models \forall x.\, \exists\, y\, .\, \mathsf{FC}_f(x, y)$.

We call the constraint D_f deterministic *if* FC_f *is deterministic with respect to* I*, i.e.*

$$I \models \forall x\, y\, z.\ \mathsf{FC}_f(x,y) \wedge \mathsf{FC}_f(x,z) \ \rightarrow\ y = z\ .$$

We write D *for the conjunction of* D_f *for all derived fields* f*.*

Note that Definition 1 covers arbitrary constraints on a field, because D_f is equivalent to $\forall x.\ \mathsf{FC}_f(x, f(x))$.

The totality condition (2) is not required for the soundness of our approach, only for its completeness, and rules out invariants equivalent to "false". The condition (2) does not involve derived fields and can therefore be checked automatically using a single call to the underlying decision procedure.

Our goal is to check validity of formulas of the form $I \wedge D \ \rightarrow\ G$, where G is a formula with possible occurrences of derived fields. If G does not contain any derived fields then there is nothing to do, because in that case checking validity immediately reduces to the validity problem without field constraints, as given by the following lemma.

Lemma 1. *Let* G *be a formula such that* $\mathsf{Fields}(G) \subseteq \mathsf{Fields}(I)$*.*
Then $I \models G$ *iff* $I \wedge D \models G$*.*

To check validity of $I \wedge D \ \rightarrow\ G$, we therefore proceed as follows. We first obtain a formula G' from G by eliminating all occurrences of derived fields in G. Next, we check validity of G' with respect to I. In the case of a derived field f that is defined by a deterministic field constraint, occurrences of f in G can be eliminated by flattening the formula and substituting each term $f(x) = y$ by $\mathsf{FC}_f(x, y)$. However, in the general case of non-deterministic field constraints such a substitution is only sound for negative occurrences of derived fields, since the field constraint gives an over-approximation of the derived field. Therefore, a more sophisticated elimination algorithm is needed.

Eliminating derived fields. Figure 12 presents our algorithm Elim for elimination of derived fields. Consider a derived field f and let $F \equiv \mathsf{FC}_f$. The basic idea of Elim is that we can replace an occurrence $G(f(x))$ of f by a new variable y that satisfies $F(x, y)$, yielding a stronger formula $\forall y.\ F(x,y) \ \rightarrow\ G(y)$. As an improvement, if G contains two occurrences $f(x_1)$ and $f(x_2)$, and if x_1 and x_2 evaluate to the same value, then we attempt to replace $f(x_1)$ and $f(x_2)$ with the same value. Elim implements this idea using the set K of triples (x, f, y) to record previously assigned values for $f(x)$. Elim runs in time $O(n^2)$ where n is the size of the formula and produces an at most quadratically larger formula. Elim accepts formulas in negation normal form, where all negation signs apply to atomic formulas (see Figure 16 in the Appendix for rules of transformation into negation normal form). We generally assume that each quantifier $Q\, z$ binds a variable z that is distinct from other bound variables and distinct from the free variables of the entire formula. The algorithm Elim is presented as acting on first-order formulas, but is also applicable to checking validity of quantifier-free formulas because it only introduces universal quantifiers which can be replaced by Skolem constants. The algorithm is also applicable to multisorted logics, and, by treating sets of elements as a new sort, to MSOL. To make the discussion simpler, we consider a deterministic version of Elim where the non-deterministic choices of variables and terms are resolved by some

$$
\begin{aligned}
S &- \text{a term or a formula} \\
\mathsf{Terms}(S) &- \text{terms occurring in } S \\
\mathsf{FV}(S) &- \text{variables free in } S \\
\mathsf{Ground}(S) &= \{t \in \mathsf{Terms}(S).\ \mathsf{FV}(t) \subseteq \mathsf{FV}(S)\} \\
\mathsf{Derived}(S) &- \text{derived function symbols in } S
\end{aligned}
$$

proc $\mathsf{Elim}(G) = \mathsf{elim}(G, \emptyset)$
proc $\mathsf{elim}(G$: formula in negation normal form;
    K : set of (variable,field,variable) triples)**:**
  let $T = \{f(t) \in \mathsf{Ground}(G).\ f \in \mathsf{Derived}(G) \wedge \mathsf{Derived}(t) = \emptyset\}$
  if $T \neq \emptyset$ **do**
    choose $f(t) \in T$
    choose x, y fresh first-order variables
    let $F = \mathsf{FC}_f$
    let $F_1 = F(x, y) \wedge \bigwedge_{(x_i, f, y_i) \in K} (x = x_i \ \rightarrow\ y = y_i)$
    let $G_1 = G[f(t) := y]$
    return $\forall x.\ x = t \ \rightarrow\ \forall y.\ (F_1 \ \rightarrow\ \mathsf{elim}(G_1, K \cup \{(x, f, y)\}))$
  else case G **of**
  | $Qx.\ G_1$ **where** $Q \in \{\forall, \exists\}$**:**
      return $Qx.\ \mathsf{elim}(G_1, K)$
  | G_1 *op* G_2 **where** *op* $\in \{\wedge, \vee\}$**:**
      return $\mathsf{elim}(G_1, K)$ *op* $\mathsf{elim}(G_2, K)$
  | **else return** G

Fig. 12. Derived-field elimination algorithm

arbitrary, but fixed, linear ordering on terms. We write $\mathsf{Elim}(G)$ to denote the result of applying Elim to a formula G.

The correctness of Elim is given by Theorem 1. The proof of Theorem 1 relies on the monotonicity of logical operations and quantifiers in negation normal form of a formula.

Theorem 1 (Soundness). *The algorithm* Elim *is sound: if* $I \wedge D \models \mathsf{Elim}(G)$*, then* $I \wedge D \models G$*. What is more,* $I \wedge D \wedge \mathsf{Elim}(G) \models G$*.*

Completeness. We now analyze the classes of formulas G for which Elim is *complete*.

Definition 2. *We say that* Elim *is complete for* (D, G) *iff* $I \wedge D \models G$ *implies* $I \wedge D \models \mathsf{Elim}(G)$*.*

Note that we cannot hope to achieve completeness for arbitrary constraints D. Indeed, if we let $D \equiv \mathsf{true}$, then D imposes no constraint whatsoever on the derived fields, and reasoning about the derived fields becomes reasoning about uninterpreted function symbols, that is, reasoning in unconstrained predicate logic. Such reasoning is undecidable not only for monadic second-order logic, but also for much weaker fragments of first-order logic [7]. Despite these general observations, we have identified two cases important in practice for which Elim is complete (Theorem 2 and Theorem 3).

Theorem 2 expresses the fact that, in the case where all field constraints are deterministic, Elim is complete (and then it reduces to previous approaches [11, 28] that are restricted to the deterministic case). The proof of Theorem 2 uses the assumption that

F is total and functional to conclude $\forall x\ y.\ F(x,y) \ \rightarrow\ f(x)=y$, and then uses an inductive argument similar to the proof of Theorem 1.

Theorem 2 (Completeness for deterministic fields). *Algorithm* Elim *is complete for* (D,G) *when each field constraint in* D *is deterministic.*
What is more, $I \wedge D \wedge G \models \mathsf{Elim}(G)$.

$$
\begin{array}{lll}
x \in \mathsf{Var} - \text{program variables} & & f \in \mathsf{Fld} - \text{pointer fields} \\
e \in \mathsf{Exp} ::= x \mid e.f & & F - \text{quantifier free formula} \\
c \in \mathsf{Com} ::= & e_1 := e_2 & \\
 & \mid\ \mathsf{havoc}(x) & \text{(non-deterministic assignment to } x\text{)} \\
 & \mid\ \mathsf{assume}(F) \mid \mathsf{assert}(F) & \\
 & \mid\ c_1\,\mathbf{;}\,c_2 & \text{(sequential composition)} \\
 & \mid\ c_1 \,\square\, c_2 & \text{(non-deterministic choice)}
\end{array}
$$

Fig. 13. Loop-free statements of a guarded command language (see e.g. [1])

We next turn to completeness in the cases that admit non-determinism of derived fields. Theorem 3 states that our algorithm is complete for derived fields introduced by the weakest precondition operator to a class of postconditions that includes field constraints. This result is very important in practice. For example, when we used a previous version of an elimination algorithm that was incomplete, we were not able to verify the skip list example in Section 2.2. To formalize our completeness result, we introduce two classes of well-behaved formulas: *nice formulas* and *pretty nice formulas*.

Definition 3 (Nice formulas). *A formula* G *is a* nice formula *if each occurrence of each derived field* f *in* G *is of the form* $f(t)$*, where* $t \in \mathsf{Ground}(G)$.

Nice formulas generalize the notion of quantifier-free formulas by disallowing quantifiers only for variables that are used as arguments to derived fields. Lemma 2 shows that the elimination of derived fields from nice formulas is complete. The intuition behind Lemma 2 is that if $I \wedge D \models G$, then for the choice of y_i such that $F(x_i, y_i)$ we can find an interpretation of the function symbol f such that $f(x_i) = y_i$, and $I \wedge D$ holds, so G holds as well, and $\mathsf{Elim}(G)$ evaluates to the same truth value as G.

Lemma 2. Elim *is complete for* (D,G) *if* G *is a nice formula.*

Definition 4 (Pretty nice formulas). *The set of* pretty nice formulas *is defined inductively by 1) a nice formula is pretty nice; 2) if* G_1 *and* G_2 *are pretty nice, then* $G_1 \wedge G_2$ *is pretty nice; 3) if* G *is pretty nice and* x *is a first-order variable, then* $\forall x.G$ *is pretty nice.*

Pretty nice formulas therefore additionally admit universal quantification over arguments of derived fields. Define function skolem as follows: 1) $\mathsf{skolem}(\forall x.G) = G$; 2) $\mathsf{skolem}(G_1 \wedge G_2) = \mathsf{skolem}(G_1) \wedge \mathsf{skolem}(G_2)$; and 3) $\mathsf{skolem}(G) = G$ if G is not of the form $\forall x.G$ or $G_1 \wedge G_2$.

Lemma 3. *The following observations hold:*

1. *each field constraint* D_f *is a pretty nice formula;*
2. *if* G *is a pretty nice formula, then* $\mathsf{skolem}(G)$ *is a nice formula and* $H \models G$ *iff* $H \models \mathsf{skolem}(G)$ *for any set of formulas* H.

The next Lemma 4 shows that pretty nice formulas are closed under wlp; the lemma follows from the conjunctivity of the weakest precondition operator.

Lemma 4. *Let c be a guarded command of the language in Figure 13. If G is a nice formula, then $\mathsf{wlp}(c, G)$ is a nice formula. If G is a pretty nice formula, then $\mathsf{wlp}(c, G)$ is equivalent to a pretty nice formula.*

Lemmas 4, 3, 2, and 1 imply our main theorem, Theorem 3. Theorem 3 implies that Elim is a complete technique for checking preservation (over straight-line code) of field constraints, even if they are conjoined with additional pretty nice formulas. Elimination is also complete for data structure operations with loops as long as the necessary loop invariants are pretty nice.

Theorem 3 (Completeness for preservation of field constraints). *Let G be a pretty nice formula, D a conjunction of field constraints, and c a guarded command (Figure 13). Then*

$$I \wedge D \models \mathsf{wlp}(c, G \wedge D) \quad \textit{iff} \quad I \models \mathsf{Elim}(\mathsf{wlp}(c, \mathsf{skolem}(G \wedge D))).$$

Example 1. The example in Figure 14 demonstrates the elimination of derived fields using algorithm Elim. It is inspired by the skip list module from Section 2.

$$\mathsf{D}_{nextSub} \equiv \forall v_1\, v_2.\ nextSub(v_1) = v_2 \rightarrow next^+(v_1, v_2)$$

$$\begin{aligned} G &\equiv \mathsf{wlp}((e.nextSub := root.nextSub\, \mathbf{;}\ e.next := root), \mathsf{D}_{nextSub}) \\ &\equiv \forall v_1\, v_2.\ nextSub[e := nextSub(root)](v_1) = v_2 \rightarrow (next[e := root])^+(v_1, v_2) \end{aligned}$$

$$\begin{aligned} G' \equiv\ & \mathsf{skolem}(\mathsf{Elim}(G)) \equiv \\ & x_1 = root \rightarrow next^+(x_1, y_1) \rightarrow \\ & \quad x_2 = v_1 \rightarrow next^+[e := y_1](x_2, y_2) \wedge (x_2 = x_1 \rightarrow y_2 = y_1) \rightarrow \\ & \qquad y_2 = v_2 \rightarrow (next[e := root])^+(v_1, v_2) \end{aligned}$$

Fig. 14. Elimination of derived fields from a pretty nice formula. The notation $next^+$ denotes the irreflexive transitive closure of predicate $next(x) = y$.

The formula G expresses the preservation of field constraint $\mathsf{D}_{nextSub}$ for updates of fields $next$ and $nextSub$ that insert e in front of $root$. This formula is valid under the assumption that $\forall x.\ next(x) \neq e$ holds. The algorithm Elim first replaces the inner occurrence $nextSub(root)$ and then the outer occurrence of $nextSub$. Theorem 3 implies that the resulting formula $\mathsf{skolem}(\mathsf{Elim}(G))$ is valid under the same assumption as the original formula G.

Limits of completeness. In our implementation, we have successfully used Elim in the context of MSOL, where we encode transitive closure using second-order quantification. Unfortunately, formulas that contain transitive closure of derived fields are often not pretty nice, leading to false alarms after the application of Elim. This behavior is to be expected due to the undecidability of transitive closure logics over general graphs [10]. On the other hand, unlike approaches based on axiomatizations of transitive closure in first-order logic, our use of MSOL enables complete reasoning about reachability over the backbone fields. It is therefore useful to be able to consider a field as part of a backbone whenever possible. For this purpose, it can be helpful to verify conjunctions of constraints using different backbone for different conjuncts.

Verifying conjunctions of constraints. In our skip list example, the field `nextSub` forms an acyclic (sub-)list. It is therefore possible to verify the conjunction of constraints independently, with `nextSub` a derived field in the first conjunct (as in Section 2.2) but a backbone field in the second conjunct. Therefore, although the reasoning about transitive closure is incomplete in the first conjunct, it is complete in the second conjunct.

Verifying programs with loop invariants. The technique described so far supports the following approach for verifying programs annotated with loop invariants:

1. generate verification conditions using loop invariants, pre-, and postconditions;
2. eliminate derived fields from verification conditions using Elim (and skolem);
3. decide the resulting formula using a decision procedure such as MONA [13].

Field constraints specific to program point. Our completeness results also apply when, instead of having one global field constraint, we introduce different field constraints for each program point. This allows the developer to refine data structure invariants with the information about the data structure specific to particular program points.

Field constraint analysis and loop invariant inference. Field constraint analysis is not limited to verification in the presence of loop invariants. In combination with abstract interpretation [3] it can be used to infer loop invariants automatically. Our implementation combines field constraint analysis with symbolic shape analysis based on Boolean heaps [32, 36] to infer loop invariants that are universally quantified Boolean combinations of unary predicates over heap objects.

Symbolic shape analysis casts the idea of three-valued shape analysis [35] in the framework of predicate abstraction. It uses the machinery of predicate abstraction to automatically construct the abstract post operator and this construction solely goes by deductive reasoning. In fact, the computation of the abstraction amounts to checking validity of entailments that are of the form: $\Gamma \wedge C \quad \rightarrow \quad \mathsf{wlp}(c, p)$. Here Γ is an over-approximation of the reachable states, C is a conjunction of abstraction predicates and p is a single abstraction predicate. We use field constraint analysis to check validity of these formulas by augmenting them with the appropriate simulation invariant I and field constraints D that specify the data structure invariants we want to preserve: $I \wedge D \wedge \Gamma \wedge C \quad \rightarrow \quad \mathsf{wlp}(c, p)$. The only problem arises from the fact that these additional invariants may be temporarily violated during program execution. To ensure applicability of the analysis, we abstract complete loop free paths in the control flow graph of the program at once. That means we only require that simulation invariants are valid at loop cut points and hence part of the loop invariants. This supports the programming model where violations of data structure invariants are confined to the interior of basic blocks [28].

Amortizing invariant checking in loop invariant inference. A straightforward approach to combine field constraint analysis with abstract interpretation would do a well-formedness check for global invariants and field constraints at every step of the fixed-point computation, invoking a decision procedure at iteration of the fixed-point computation. The following insight allows us to use a single well-formedness check per basic block: *the loop invariant synthesized in the presence of well-formedness is identical to the loop invariant synthesized by ignoring the well-formedness check*. We therefore speculatively compute the abstraction of the system under the assumption that both the simulation invariant and the field constraints are preserved. After the least fixed-point $\mathsf{lfp}^{\#}$ of the abstract system has been computed, we generate for every loop free path c with start point ℓ_c a verification condition: $I \wedge D \wedge \mathsf{lfp}^{\#}_{\ell_c} \;\rightarrow\; \mathsf{wlp}(c, I \wedge D)$ where $\mathsf{lfp}^{\#}_{\ell_c}$ is the projection of $\mathsf{lfp}^{\#}$ to program location ℓ_c. We then use again our Elim algorithm to eliminate derived fields and check the validity of these verification conditions. If they are all valid then the analysis is sound and the data structure invariants are preserved. Note that this approach succeeds whenever the straightforward approach would have succeeded, so it improves analysis performance without degrading the precision. Moreover, when the analysis detects an error, it repeats the fixed-point computation with the simple approach to obtain an indication of the error trace.

4 Deployment as Modular Analysis Plugin

We have implemented our field constraint analysis and deployed it as the "Bohne" analysis plugin of our Hob framework [16, 22]. We have successfully verified singly-linked lists, doubly-linked lists with and without iterators and header nodes (Section 2.1), two-level skip lists (Section 2.2), and our students example from Section 2. When the developer supplies loop invariants, these benchmarks, including skip list, verify in 1.7 seconds (for the doubly-linked list) to 8 seconds (for insertion into a tree). Bohne automatically infers loop invariants for insertion and lookup in the two-level skip list in 30 minutes total. We believe the running time for loop invariant inference can be reduced using ideas such as lazy predicate abstraction [8].

Because we have integrated Bohne into the Hob framework, we were able to verify just the parts of programs which require the power of field constraint analysis with the Bohne plugin, while using less detailed analyses for the remainder of the program. We have used the list data structures verified with Bohne as modules of larger examples, such as the 900-line Minesweeper benchmark and the 1200-line web server benchmark. Hob's pluggable analysis approach allowed us to use the typestate plugin [21] and loop invariant inference techniques to efficiently verify client code, while reserving shape analysis for the container data structures.

5 Further Related Work

We are not aware of any previous work that provides completeness guarantees for analyzing tree-like data structures with non-deterministic cross-cutting fields for expressive constraints such as MSOL. TVLA [26, 35] was initially designed as an analysis framework with user-supplied transfer functions; subsequent work addresses synthesis of transfer functions using finite differencing [34], which is not guaranteed to be complete. Decision procedures [18, 27] are effective at reasoning about local properties, but are not complete for reasoning about reachability. Promising, although still incomplete, approaches include [25] as well as [19, 30]. Some reachability properties can be reduced to first-order properties using hints in the form of ghost fields [15, 27]. Completeness of analysis can be achieved by representing loop invariants or candidate loop invariants by formulas in a logic that supports transitive closure [17, 28, 32, 36–39]. These approaches treat decision procedure as a black box and, when applied to MSOL, inherit the limitations of structure simulation [11]. Our work can be viewed as a technique for lifting existing decision procedures into decision procedures that are applicable to a larger class of structures. Therefore, it can be incorporated into all of these previous approaches.

6 Conclusion

Shape analysis is one of the most challenging problems in the field of program analysis; its central relevance stems from the fact that it addresses key data structure consistency properties that are 1) important in and of themselves 2) critical for the further verification of other program properties.

Historically, the primary challenge in shape analysis was seen to be dealing effectively with the extremely precise and detailed consistency properties that characterize many (but by no means all) data structures. Perhaps for this reason, many formalisms were built on logics that supported *only* data structures with very precisely defined referencing relationships. This paper presents an analysis that supports both the extreme precision of previous approaches and the controlled reduction in the precision required to support a more general class of data structures whose referencing relationships may be random, depend on the history of the data structure, or vary for some other reason that places the referencing relationships inherently beyond the ability of previous logics and analyses to characterize. We have deployed this analysis in the context of the Hob program analysis and verification system; our results show that it is effective at 1) analyzing individual data structures to 2) verify interfaces that allow other, more scalable analyses to verify larger-grain data structure consistency properties whose scope spans larger regions of the program.

In a broader context, we view our result as taking an important step towards the practical application of shape analysis. By supporting data structures whose backbone functionally determines the referencing relationships as well as data structures with inherently less structured referencing relationships, it promises to be able to successfully analyze the broad range of data structures that arise in practice. Its integration within the Hob program analysis and verification framework shows how to leverage this analysis capability to obtain more scalable analyses that build on the results of the shape analysis to verify important properties that involve larger regions of the program. Ideally, this research will significantly increase our ability to effectively deploy shape analysis and other subsequently enabled analyses on important programs of interest to the practicing software engineer.

References

1. R.-J. Back and J. von Wright. *Refinement Calculus*. Springer-Verlag, 1998.
2. I. Balaban, A. Pnueli, and L. Zuck. Shape analysis by predicate abstraction. In *VMCAI'05*, 2005.
3. P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Proc. 6th POPL*, pages 269–282, San Antonio, Texas, 1979. ACM Press, New York, NY.

4. D. Dams and K. S. Namjoshi. Shape analysis through predicate abstraction and model checking. In *Proc. 4th International Conference on Verification, Model Checking and Abstract Interpretation*, volume 2575 of *LNCS*, pages 310–323, 2003.
5. P. Fradet and D. L. Métayer. Shape types. In *Proc. 24th ACM POPL*, 1997.
6. R. Ghiya and L. Hendren. Is it a tree, a DAG, or a cyclic graph? In *Proc. 23rd ACM POPL*, 1996.
7. E. Grädel. 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.
8. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 58–70, New York, NY, USA, 2002. ACM Press.
9. N. Immerman. *Descriptive Complexity*. Springer-Verlag, 1998.
10. N. Immerman, A. M. Rabinovich, T. W. Reps, S. Sagiv, and G. Yorsh. The boundary between decidability and undecidability for transitive-closure logics. In *Computer Science Logic (CSL)*, pages 160–174, 2004.
11. N. Immerman, A. M. Rabinovich, T. W. Reps, S. Sagiv, and G. Yorsh. Verification via structure simulation. In *CAV*, pages 281–294, 2004.
12. J. L. Jensen, 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.
13. N. Klarlund, A. Møller, and M. I. Schwartzbach. MONA implementation secrets. In *Proc. 5th International Conference on Implementation and Application of Automata*. LNCS, 2000.
14. N. Klarlund and M. I. Schwartzbach. Graph types. In *Proc. 20th ACM POPL*, Charleston, SC, 1993.
15. V. Kuncak, P. Lam, and M. Rinard. Role analysis. In *Proc. 29th POPL*, 2002.
16. V. Kuncak, P. Lam, K. Zee, and M. Rinard. Implications of a data structure consistency checking system. In *International conference on Verified Software: Theories, Tools, Experiments (VSTTE, IFIP Working Group 2.3 Conference)*, Zürich, Switzerland, 10–13th October 2005.
17. V. Kuncak and M. Rinard. Boolean algebra of shape analysis constraints. In *Proc. 5th International Conference on Verification, Model Checking and Abstract Interpretation*, 2004.
18. V. Kuncak and M. Rinard. Decision procedures for set-valued fields. In *1st International Workshop on Abstract Interpretation of Object-Oriented Languages (AIOOL 2005)*, 2005.
19. S. K. Lahiri and S. Qadeer. Verifying properties of well-founded linked lists. In *POPL'06*, 2006.
20. P. Lam, V. Kuncak, and M. Rinard. On our experience with modular pluggable analyses. Technical Report 965, MIT CSAIL, September 2004.
21. P. Lam, V. Kuncak, and M. Rinard. Generalized typestate checking for data structure consistency. In *6th International Conference on Verification, Model Checking and Abstract Interpretation*, 2005.
22. P. Lam, V. Kuncak, and M. Rinard. Hob: A tool for verifying data structure consistency. In *14th International Conference on Compiler Construction (tool demo)*, April 2005.
23. P. Lam, V. Kuncak, K. Zee, and M. Rinard. The Hob project web page. http://hob.csail.mit.edu, 2004.
24. O. Lee, H. Yang, and K. Yi. Automatic verification of pointer programs using grammar-based shape analysis. In *ESOP*, 2005.
25. T. Lev-Ami, N. Immerman, T. Reps, M. Sagiv, S. Srivastava, and G. Yorsh. Simulating reachability using first-order logic with applications to verification of linked data structures. In *CADE-20*, 2005.

26. T. Lev-Ami, T. Reps, M. Sagiv, and R. Wilhelm. Putting static analysis to work for verification: A case study. In *International Symposium on Software Testing and Analysis*, 2000.
27. S. McPeak and G. C. Necula. Data structure specifications via local equality axioms. In *CAV*, pages 476–490, 2005.
28. A. Møller and M. I. Schwartzbach. The Pointer Assertion Logic Engine. In *Programming Language Design and Implementation*, 2001.
29. S. S. Muchnick and N. D. Jones, editors. *Program Flow Analysis: Theory and Applications*. Prentice-Hall, Inc., 1981.
30. G. Nelson. Verifying reachability invariants of linked structures. In *Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 38–47. ACM Press, 1983.
31. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer-Verlag, 2002.
32. A. Podelski and T. Wies. Boolean heaps. In *SAS*, 2005.
33. W. Pugh. Skip lists: A probabilistic alternative to balanced trees. In *Communications of the ACM 33(6):668–676*, 1990.
34. T. Reps, M. Sagiv, and A. Loginov. Finite differencing of logical formulas for static analysis. In *Proc. 12th ESOP*, 2003.
35. M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM TOPLAS*, 24(3):217–298, 2002.
36. T. Wies. Symbolic shape analysis. Master's thesis, Universität des Saarlandes, Saarbrücken, Germany, Sep 2004.
37. G. Yorsh, T. Reps, and M. Sagiv. Symbolically computing most-precise abstract operations for shape analysis. In *10th TACAS*, 2004.
38. G. Yorsh, T. Reps, M. Sagiv, and R. Wilhelm. Logical characterizations of heap abstractions. *TOCL*, 2005. (to appear).
39. G. Yorsh, A. Skidanov, T. Reps, and M. Sagiv. Automatic assume/guarantee reasoning for heap-manupilating programs (ongoing work). In *1st AIOOL Workshop on Abstract Interpretation of Object-Oriented Programs*, 2005.

A Semantics of Guarded-Command Language

To make the completeness statement for our guarded command language precise, we present in Figure 15 the weakest precondition semantics for the language presented in Figure 13.

$$\begin{aligned}
\mathsf{wlp}(x := e, G) &\stackrel{\mathit{def}}{=} G[x := e] \\
\mathsf{wlp}(e_1.f := e_2, G) &\stackrel{\mathit{def}}{=} G[f := \lambda\, v\,.\, \text{if } v = e_1 \text{ then } e_2 \text{ else } f(v)] \\
\mathsf{wlp}(\mathsf{havoc}(x), G) &\stackrel{\mathit{def}}{=} G[x := x_0] \quad \text{with } x_0 \text{ a fresh constant symbol} \\
\mathsf{wlp}(\mathsf{assert}(F), G) &\stackrel{\mathit{def}}{=} F \wedge G \\
\mathsf{wlp}(\mathsf{assume}(F), G) &\stackrel{\mathit{def}}{=} \neg F \vee G \\
\mathsf{wlp}(c_1\, ;\, c_2, G) &\stackrel{\mathit{def}}{=} \mathsf{wlp}(c_1, \mathsf{wlp}(c_2, G)) \\
\mathsf{wlp}(c_1 \,\Box\, c_2, G) &\stackrel{\mathit{def}}{=} \mathsf{wlp}(c_1, G) \;\wedge\; \mathsf{wlp}(c_2, G)
\end{aligned}$$

Fig. 15. Weakest Precondition Semantics

B Negation Normal Form

To avoid any ambiguity, Figure 16 presents rules for transforming a formula into negation normal form. This transformation ensures that all occurrences of field constraint formulas introduced by Elim are negative in the top-level formula.

proc $\mathsf{NegationNormalForm}(G$: formula with connectives $\wedge, \vee, \neg)$**:**
apply the following rewrite rules:

$$\begin{array}{rcl} \neg(\forall x.G) & \rightarrow & \exists x.\neg G \\ \neg(\exists x.G) & \rightarrow & \forall x.\neg G \\ \neg\neg G & \rightarrow & G \\ \neg(G_1 \wedge G_2) & \rightarrow & (\neg G_1) \vee (\neg G_2) \\ \neg(G_1 \vee G_2) & \rightarrow & (\neg G_1) \wedge (\neg G_2) \end{array}$$

Fig. 16. Negation Normal Form

C Proofs

Proof of Lemma 1. The left-to-right direction follows immediately. For the right-to-left direction assume that $I \wedge D \;\rightarrow\; G$ is valid. Let S be a structure such that $S \models I$. By totality of all field constraints in D there exists a structure S' such that $S' \models I \wedge D$ and S' differs from S only in the interpretation of derived fields. Since $\mathsf{Fields}(G) \subseteq \mathsf{Fields}(I)$ and I contains no derived fields we have that $S' \models G$ implies $S \models G$. ■

Proof of Theorem 1. By induction on the first argument G of elim we prove that, for all finite K,

$$I \wedge D \wedge \mathsf{elim}(G, K) \wedge \bigwedge_{(x_i, f_i, y_i) \in K} \mathsf{FC}_{f_i}(x_i, y_i) \models G$$

For $K = \emptyset$ we obtain $I \wedge D \wedge \mathsf{Elim}(G) \models G$, as desired. In the inductive proof, the cases when $T = \emptyset$ are straightforward. The case $f(t) \in T$ uses the fact that if $M \models G[f(t) := y]$ and $M \models f(t) = y$, then $M \models G$. ■

Proof of Theorem 2. Consider a field constraint $F \equiv \mathsf{FC}_f$ and let $\bar{x}$ and $\bar{y}$ be such that $F(\bar{x}, \bar{y})$. Because $F(\bar{x}, f(\bar{x}))$ and F is deterministic by assumption, we have $\bar{y} = f(\bar{x})$. It follows that $I \wedge D \wedge F(x, y) \models f(x) = y$. We then prove by induction on the argument G of elim that, for all finite K,

$$I \wedge D \wedge G \wedge \bigwedge_{(x_i, f_i, y_i) \in K} f_i(x_i) = y_i \models \mathsf{elim}(G, K)$$

For $K = \emptyset$ we obtain $I \wedge D \wedge G \models \mathsf{Elim}(G)$, as desired. The inductive proof is similar to the proof of Theorem 1. In the case $f(t) \in T$, we consider a model M such that $M \models I \wedge D \wedge G \wedge \bigwedge_{(x_i, f_i, y_i) \in K} f_i(x_i) = y_i$. Consider any $\bar{x}, \bar{y}$ such that: 1) $M \models x = t$, 2) $M \models F(x, y)$ and 3) $M \models x = x_i \;\rightarrow\; y = y_i$ for all $(x_i, f, y_i) \in K$. To show $M \models \mathsf{elim}(G_1, K \cup \{(x, f, y)\})$, we consider a modified model $M_1 = M[f(\bar{x}) := \bar{y}]$ which is like M except that the interpretation of f at $\bar{x}$ is $\bar{y}$. By $M \models F(x, y)$ we conclude $M_1 \models I \wedge D$. By $M \models x = x_i \;\rightarrow\; y = y_i$, we conclude $M_1 \models \bigwedge_{(x_i, f_i, y_i) \in K} f_i(x_i) = y_i$ as well. Because $I \wedge D \wedge F(x, y) \models f(x) = y$, we conclude $M_1 \models f(x) = y$. Because $M \models x = t$ and $\mathsf{Derived}(t) = \emptyset$, we have $M_1 \models x = t$ so from $M \models G$ we conclude $M_1 \models G_1$ where $G_1 = G[f(t) := y]$. By induction hypothesis we then conclude $M_1 \models \mathsf{elim}(G_1, K \cup \{(x, f, y)\}$. Then also $M \models \mathsf{elim}(G_1, K \cup \{(x, f, y)\}$ because the result of elim does not contain f. Because $\bar{x}, \bar{y}$ were arbitrary, we conclude $M \models \mathsf{elim}(G, K)$. ■

Proof of Lemma 2. Let G be a nice formula. To show that $I \wedge D \models G$ implies $I \wedge D \models \mathsf{Elim}(G)$, let $I \wedge D \models G$ and let $f_1(t_1), \ldots, f_n(t_n)$ be the occurrences of derived fields in G. By assumption, $t_1, \ldots, t_n \in \mathsf{Ground}(G)$ and $\mathsf{Elim}(G)$ is of the form

$$\begin{array}{l}
\forall x_1\, y_1.\, x_1 = t_1 \;\rightarrow\; (F_1^1 \wedge \\
\quad \forall x_2\, y_2.\, x_2 = t_2' \;\rightarrow\; (F_1^2 \wedge \\
\quad\quad \ldots \\
\quad\quad\quad \forall x_n, y_n.\, x_n = t_n' \;\rightarrow\; (F_1^n \wedge G_0) \ldots))
\end{array}$$

where t'_i differs from t_i in that some of its subterms may be replaced by variables y_j for $j < i$. Here $F^i = \mathsf{FC}_{f_i}$ and

$$F_1^i = F^i(x_i, y_i) \wedge \bigwedge_{j<i, f_j=f_i} (x_i = x_j \ \rightarrow \ y_i = y_j).$$

Consider a model M of $I \wedge D$, we show M is a model for $\mathsf{Elim}(G)$. Consider any assignment $\bar{x}_i, \bar{y}_i$ to variables x_i, y_i for $1 \leq i \leq n$. If any of the conditions $x_i = t_i$ or F_1^i are false for this assignment, then $\mathsf{Elim}(G)$ is true because these conditions are on the left-hand side of an implication. Otherwise, conditions $F_1^i(x_i, y_i)$ hold, so by definition of F_1^i, if $\bar{x}_i = \bar{x}_j$, then $\bar{y}_i = \bar{y}_j$. Therefore, for each distinct function symbol f_j there exist a function $\bar{f}_j$ such that $\bar{f}(x_i) = \bar{y}_i$ for $f_j = f_i$. Because $F^i(x_i, y_i)$ holds and each FC_f is total, we can define such $\bar{f}_j$ so that D holds. Let $M' = M[f_j \mapsto \bar{f}_j]_j$ be a model that differs from M only in that f_j are interpreted as $\bar{f}_j$. Then $M' \models I$ because I does not mention derived fields and $M' \models D$ by construction. We therefore conclude $M' \models G$. If $\bar{t}_i$ is the value of t_i in M', then $\bar{x}_i = \bar{t}_i$ because $M \models x_i = t_i$ and $\mathsf{Derived}(t_i) = \emptyset$. Using this fact, as well as $\bar{f}_j(\bar{x}_i) = \bar{y}_i$, by induction on subformulas of G_0 we conclude that G_0 has the same truth value as G in M', so $M' \models G_0$. Because G_0 does not contain derived function symbols, $M \models G_0$ as well. Because $\bar{x}_i$ and $\bar{y}_i$ were arbitrary, we conclude $M \models \mathsf{Elim}(G)$. This completes the proof.

Remark. Note that it is not the case that a stronger statement $I \wedge D \wedge G \models \mathsf{Elim}(G)$ holds. For example, take $D \equiv \mathsf{true}$, and $G \equiv f(a) = b$. Then $\mathsf{Elim}(G)$ is equivalent to $\forall y. y = b$ and it is not the case that $I \wedge f(a) = b \models \forall y. y = b$. ■

Proof of Lemma 4. Using the conjunctivity properties of wlp:

$$\mathsf{wlp}(c, \forall x. G) \ \leftrightarrow \ \forall x. \mathsf{wlp}(c, G)$$

and

$$\mathsf{wlp}(c, G_1 \wedge G_2) \ \leftrightarrow \ \mathsf{wlp}(c, G_1) \wedge \mathsf{wlp}(c, G_2)$$

the problem reduces to proving the lemma for the case of nice formulas.

Since we defined wlp recursively on the structure of commands, we prove the statement by structural induction on command c. For $c = (e_1 := e_2)$ and $c = \mathsf{havoc}(x)$ we have that wlp replaces ground terms by ground terms, i.e. in particular all introduced occurrences of derived fields are ground. For $c = \mathsf{assume}(F)$ and $c = \mathsf{assert}(F)$ every occurrence of a derived field introduced by wlp comes from F. Since F is quantifier free, all such occurrences are ground. The remaining cases follow from the induction hypothesis for component commands. ■

Proof of Theorem 3. Let G be a quite nice formula, D a conjunction of field constraints, and c a guarded command. Since $\mathsf{skolem}(G \wedge D)$ is a nice formula, Lemma 4 implies that $\mathsf{wlp}(c, \mathsf{skolem}(G \wedge D))$ is a nice formula, so we have

$$\begin{array}{ll}
I \wedge D \models \mathsf{wlp}(c, G \wedge D) & \\
I \wedge D \models \mathsf{wlp}(c, \mathsf{skolem}(G \wedge D)) & \text{(by Lemma 3)} \\
I \wedge D \models \mathsf{Elim}(\mathsf{wlp}(c, \mathsf{skolem}(G \wedge D))) & \text{(by Lemma 2)} \\
I \models \mathsf{Elim}(\mathsf{wlp}(c, \mathsf{skolem}(G \wedge D))) & \text{(by Lemma 1)}
\end{array}$$

■

D Specifying Bohne Analysis Tasks

In this appendix, we expand on Section 4 and describe how a developer actually uses Bohne to verify program parts using shape analysis. When developing programs with the Hob framework, the developer divides the program into a set of modules. For each module, the developer must provide module implementations (in a standard programming language) and specifications (in a set-based specification language) for program modules. To make sense of the set specifications, an analysis clearly needs to know what each set means. Hob enables developers to supply set definitions using customized abstraction function languages: each analysis plugin can verify that a module's implementations conforms to its specification using the module's abstraction section.

We next describe the contents of Bohne abstraction modules; these abstraction modules express set definitions and invariants using first-order formulas with reflexive transitive closure, thereby enabling the Bohne plugin to verify that a module implementation conforms to its specification. Abstract sets in procedure preconditions and postconditions are translated using the set definitions in the abstraction modules. Invariants ensure that the set definitions are always meaningful by constraining the concrete program state. They prohibit backbone fields from forming non-tree data structures and give field constraints for derived fields. Invariants are always assumed upon entry to a procedure and verified upon exit from a procedure; they may temporarily be violated within procedures. Given module implementations, specifications, invariants, and set definitions, the Bohne plugin emits and approximates verification conditions using the techniques described in Section 3 and checks them using the MONA decision procedure.

Specifying heap predicates. The abstraction function used in the analysis of the Bohne plugin is induced by a set of unary heap predicates. Heap predicates are specified by the developer in terms of sets. These sets are defined by using formulas in first-order logic with reflexive transitive closure. In particular, the developer must provide the definitions of all abstract sets used in the specification section of the module. Furthermore, additional heap predicates are often needed for Bohne to successfully infer loop invariants; the proc construct allows the developer to define these heap predicates.

In addition to user-provided heap predicates, the plugin automatically introduces heap predicates for every global and local object-typed variable of the analyzed procedure and the `null` object. Moreover, for every unprimed abstract set S that occurs in a post condition of the analyzed procedure, a *tick* predicate $'S$ is introduced. The Bohne plugin uses these tick predicates to compute a procedure summary that allows the verification of the post condition.

Specifying representation invariants. The developer specifies the representation invariants for the Bohne plugin using invariant declarations in the abstraction section, as previously illustrated, for instance, in Figure 7. The Bohne plugin supports two kinds of representation invariants:

- *field constraints*, given by formulas D_f of the form

$$\forall x\ y.\ f(x) = y \ \rightarrow\ F(x, y)$$

- and *state invariants*, given by any formula which is not a field constraint.

A field constraint describes a field f in terms of a formula $F \equiv \mathsf{D}_f$. An example of such a derived field—that is, a field specified using only field constraints—in Figure 7 is the field `prev`. Field constraints impose additional implicit well-formedness constraints on the heap: all fields without a field invariant are considered to span a forest. The field constraints themselves and the treeness property for the non-derived fields may be violated within the procedure, with the exception of loop cut points and exit points of the procedure. This means, in particular, that the field constraints and the treeness property are part of all loop invariants.

State invariants may be violated at any point within the procedure, as long as they are reestablished by the end of the procedure. An example of a state invariant is the invariant given in Figure 7 which says that, if `root` is not pointing to `null`, then it has no incoming `next` edges,

The analysis restricts the heap to the part visible from program variables in the analyzed procedure. Moreover, all constraints apply to the projection of the heap onto the fields declared in the currently analyzed module. In keeping with the Hob philosophy of modular analysis, field and treeness constraints do not apply to fields declared in other modules, which enables objects to participate in multiple data structures and makes the Bohne plugin applicable to more general program components.