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 limits the range of analyzable data structures.

Field constraint analysis permits *non-deterministic* field constraints on cross-cutting fields, which allows the verificiation of invariants for 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. 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 [27, Chapter 4], [2, 4–6, 22, 25, 26, 32] 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, 26] 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, 21] for analyzing data structure consistency, where we have integrated one such shape analysis [26] 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 [26]. 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 relationship (1) 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 [30]. 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 [26] 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. To handle the general case (2), 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 [29,33]. 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 9) 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). Theorem 3 implies a complete technique for checking that field constraints hold, if the programmer adheres to a discipline of maintaining them, for instance at the beginning of each loop.
- We describe how to combine our algorithm with symbolic shape analysis [33] 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, 21] allows us to apply the analysis to modules of larger applications, with other modules analyzed by more scalable analyses, such as typestate analysis [20].

Additional details (including proofs of theorems) are in [34].

2 Examples

We next explain our field constraint analysis with a set of examples. Note that our analysis handles, as a special case, data structures that have back pointers constrained by deterministic constraints. Such data structures (for instance, doubly linked lists and trees with parent pointers [34]) have also been analyzed by previous approaches [26]. To illustrate the additional power of our analysis, we first present an example illustrating inter-data-structure constraints, which are simple and useful for high-level application properties, but are often nondeterministic. We then present a skip list example, which shows how non-deterministic field constraints arise within data structures, and illustrates how our analysis can synthesize loop invariants.

2.1 Students and Schools

The data structure in our first example contains two linked lists: one containing students and one containing schools (Figure 2). 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. An invariant of the data structure is that, if an object is in the list of students, then its `attends` field points to an object in the schools list; that is, it cannot be null and it cannot point to an object outside the list of schools. This invariant is an example of a non-deterministic field constraint: the `attends` field has a non-trivial constraint, but the target of the field is not uniquely defined in terms of existing fields; instead, this field carries important new information about the school that each student attends.

We implement our example as a module in the Hob system [21], which allows us to specify and, using field constraint analysis, verify the desired data structure invariants and interfaces of data structure operations. In general, a module in Hob consists of three sections: 1) an implementation section (Figure 1) containing declarations of memory cell formats (in this case `Elem`) and executable code for data structure operations

```
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. 1. Implementation for students example

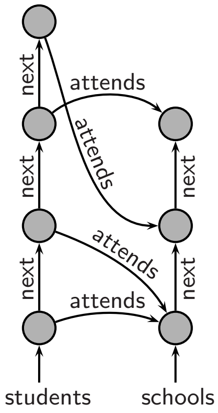

Fig. 2. Students data structure instance

```
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. 3. 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))))";

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

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

Fig. 4. Abstraction for students example

(such as addStudent); 2) a specification section (Figure 3) containing declarations of abstract sets of objects (such as ST for the set of students and SC for the set of schools in the data structure) and interfaces of data structure operations expressed in terms of these abstract sets; and 3) the abstraction section, which gives the abstraction function specifying the definition of sets (SC and ST) and specifies the representation invariants of the data structure, including field constraints (in this case, on the field attends).

The implementation in Figure 1 states that the addStudent procedure adds a student st to the student list and associates it (via the attends field) with an existing school sc, which is expected to be already in the list of schools. Figure 3 presents the set interface for the addStudents procedure, consisting of a precondition (requires clause), frame condition (modifies clause), and postcondition (ensures clause). The precondition states that st must not already be in the list of students ST, and that sc must be in the list of schools. We represent parameters as sets of cardinality at most one (the null object is represented as an empty set). Therefore, the conjuncts card(st)=1 and card(sc)=1 in the precondition indicate that the parameters st and sc are not null. The modifies clause indicates that only the set of students ST and not the set of schools SC is modified. The postcondition describes the effect of the procedure: it states that the set of students ST' after procedure execution is equal to the union (denoted +) of the set ST of student objects before procedure execution, and (the singleton containing) the given student object st.

Our analysis automatically verifies that the data structure operation addStudent conforms to its interface expressed in terms of abstract sets. Proving the conformance of a procedure to such a set interface is useful for several reasons. First, the preconditions indicate to data structure clients the conditions under which it is possible to invoke operations. These preconditions are necessary to prove that the field constraint is maintained: if it was not the case that the school parameter sc belonged to the set SC of schools, the insertion would violate the representation invariant. Similarly, if it was the case that the student object st was a member of the student list, insertion would introduce cycles in the list and violate the implicit acyclicity invariant of the data structure. Also, the postcondition of addStudents communicates the fact that st is in the list after the insertion, preventing clients from executing duplicate calls to addStudents with the same student object. Finally, the set interface expresses an important partial correctness property for the addStudent procedure, so that the verification of the set interface indicates that the procedure is correctly inserting an object into the set of students.

Note that the interface of the procedure does not reveal the details of procedure implementation, thanks to the use of abstract set variables. Since the set variables in the specification are abstract, any verification of a concrete implementation's conformance to the set interface requires concrete definitions for the abstract variables. The abstraction section in Figure 4 contains this information. First, the abstraction section indicates which analysis (in this case, Bohne decaf, which implements field constraint analysis) is to be used to analyze the module. Next, the abstraction section contains definitions for abstract variables: namely, ST is defined as the set of Elem objects reachable from the root students through next fields, and SC is the set of Elem objects reachable from schools. (The function rtrancl is a higher-order function that accepts a binary predicate on objects and returns the reflexive transitive closure of the predicate.) The abstraction section also specifies data structure invariants, including field constraints. Field constraints are invariants with syntactic form `ALL x y. (f x = y) -->` $\cdots$. A field `f` for which there is no field constraint invariant in the abstraction section is considered to be part of the data structure *backbone*, which has an implicit invariant that it is a union of trees. Finally, the abstraction section may contain additional invariants; our example contains invariants stating disjointness of the lists rooted at `students` and `schools`.

Our Bohne analysis verifies the conformance of a procedure to its specification as follows. It first desugars the modifies clauses into a frame formula and conjoins it with the ensures clause, then replaces abstract sets in preconditions and postconditions with their definitions from the abstraction section, obtaining a procedure contract in terms of the concrete state variables (`next` and `attends`). It then conjoins representation invariants of the data structure to preconditions and postconditions. For a loop-free procedure such as `addStudents`, the analysis can then generate a verification condition whose validity implies that the procedure conforms to its interface.

The generated verification condition for our example cannot directly be solved using decision procedures such as MONA: it contains the function symbol `attends` that violates the tree invariant required by MONA. Section 3 describes how our analysis uses field constraints in the verification condition to verify the validity of such verification conditions. 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 that 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.

2.2 Skip List

We next present the analysis of a two-level skip list. Skip lists [30] 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 with only two levels: the list containing all elements, and a sublist of this list. Figure 5 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 6 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 7 presents a specification for `add`, which indicates that `add` always inserts the element into the set of elements stored in the list. Figure 8 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 unary predicates over heap objects, using symbolic shape analysis [29,33]. These unary predicates correspond to the sets that are supplied in the abstraction section using the `proc` keyword.

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) of 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 of trees, the simulation invariant I states that the fields in $\mathsf{Fields}(I)$ span a forest.

[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.

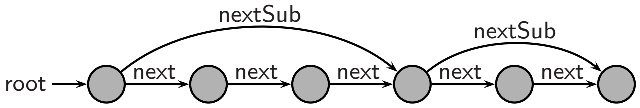

Fig. 5. An instance of a two-level skip list

```
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. 6. 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. 7. 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. 8. Skip list abstraction (including invariants)

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 we can answer the query using the underlying decision procedure. 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 9 presents our algorithm Elim for elimination of derived fields. Consider a derived field 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 $\mathsf{FC}_f(x, y)$, yielding a stronger formula $\forall y.\, \mathsf{FC}_f(x, y) \;\rightarrow\; G(y)$. 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. 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 arbitrary, but fixed, linear ordering on terms. We write $\mathsf{Elim}(G)$ to denote the result of applying Elim to a formula G.

```
                 $S$ – a term or a formula
         $\mathsf{Terms}(S)$ – terms occurring in $S$
            $\mathsf{FV}(S)$ – variables free in $S$
        $\mathsf{Ground}(S) = \{t \in \mathsf{Terms}(S).\ \mathsf{FV}(t) \subseteq \mathsf{FV}(S)\}$
       $\mathsf{Derived}(S)$ – derived function symbols in $S$

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 $T = \{f(t) \in \mathsf{Ground}(G).\ f \in \mathsf{Derived}(G) \land \mathsf{Derived}(t) = \emptyset\}$
  if $T \neq \emptyset$ do
    choose $f(t) \in T$
    choose $x, y$ fresh first-order variables
    let $F_1 = \mathsf{FC}_f(x, y) \land \bigwedge_{(x_i, f, y_i) \in K} (x = x_i \ \rightarrow\ y = y_i)$
    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 $op \in \{\land, \lor\}$:
      return $\mathsf{elim}(G_1, K)\ op\ \mathsf{elim}(G_2, K)$
  | else return $G$
```

Fig. 9. Derived-field elimination algorithm

The correctness of Elim is given by Theorem 1. The proof of Theorem 1 relies on monotonicity of logical operations and quantifiers in negation normal form of a formula. (Proofs for the theorems stated here can be found in [34]).

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

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

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

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, 26] that are restricted to the deterministic case). The proof of Theorem 2 uses the assumption that F is total and functional to conclude $\forall x\ y.\ \mathsf{FC}_f(x, y) \ \rightarrow \ f(x) = y$, and then uses an inductive argument similar to the proof of Theorem 1.

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

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 important in practice: a previous, incomplete, version of our elimination algorithm was 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. We can show that the elimination of derived fields from nice formulas is complete. The intuition behind this result is that if $I \wedge D \models G$, then for the choice of y_i such that $\mathsf{FC}_f(x_i, y_i)$ 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.

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* $G_1 \wedge G_2$ *and* $G_1 \vee G_2$ *are 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. We define the function skolem, which strips (top-level) universal quantifiers, as follows: 1) $\mathsf{skolem}(G_1\ op\ G_2) = \mathsf{skolem}(G_1)\ op\ \mathsf{skolem}(G_2)$ where $op \in \{\vee, \wedge\}$; 2) $\mathsf{skolem}(\forall x.G) = G$; and 3) $\mathsf{skolem}(G) = G$, otherwise. Note that pretty nice formulas are closed under wlp (up to formula equivalence); the closure property follows from the conjunctivity of the weakest precondition operator.

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 10). Then*

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

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

Fig. 10. Loop-free statements of a guarded command language (see e.g. [1])

Example 1. The example in Figure 11 demonstrates the elimination of derived fields using algorithm Elim. It is inspired by the skip list module from Section 2.

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

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

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

Fig. 11. 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 $root$. The formula G is valid under the assumption that $\forall x.\ next(x) \neq e$. 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 assumptions 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 backbones 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 points. 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 information 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 [29, 33] to infer loop invariants that are disjunctions of universally quantified Boolean combinations of unary predicates over heap objects.

Symbolic shape analysis casts the idea of three-valued shape analysis [32] in the framework of predicate abstraction. It uses the machinery of predicate abstraction to automatically construct the abstract post operator; this construction proceeds solely by deductive reasoning. 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. This means that we only require that simulation invariants and field constraints are valid at loop cut points; effectively, these invariants are implicit conjuncts in each loop invariant. This approach supports the programming model where violations of invariants are confined to the interior of basic blocks [26].

Amortizing invariant checking in loop invariant inference. A straightforward approach for combining 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 each iteration step. 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 check 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: $I \wedge D \wedge \mathsf{lfp}^{\#}_{\ell_c} \quad \rightarrow \quad \mathsf{wlp}(c, I \wedge D)$ 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 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* and *Bohne decaf* [4] analysis plugins of our Hob framework [16, 21]. We have successfully verified singly-linked lists, doubly-linked lists with and without iterators and header nodes, insertion into a tree with parent pointers, 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 [20] 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 [24, 32] was initially designed as an analysis framework with user-supplied transfer functions; subsequent work addresses synthesis of transfer functions using finite differencing [31], which is not guaranteed to be complete. Decision procedures [18, 25] are effective at reasoning about local properties, but are not complete for reasoning about reachability. Promising, although still incomplete, approaches include [23] as well as [19, 28]. Some reachability properties can be reduced to first-order properties using hints in the form of ghost fields [15, 25]. Completeness of analysis can be achieved by representing loop invariants or candidate loop invariants by formulas in a logic that supports transitive closure [17, 26, 29, 33, 35–37]. 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.

[4] Bohne decaf is a simpler version of Bohne that does not do loop invariant inference.

6 Conclusion

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 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.

Acknowledgements. We thank Patrick Maier, Alexandru Salcianu, and anonymous referees for comments on the presentation of the paper.

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 *POPL*, pages 269–282, 1979.
4. D. Dams and K. S. Namjoshi. Shape analysis through predicate abstraction and model checking. In *VMCAI'03*, 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*, 2002.
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 *Int. conf. on Verified Software: Theories, Tools, Experiments (VSTTE, IFIP Working Group 2.3 Conference)*, Zürich, 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. Generalized typestate checking for data structure consistency. In *6th International Conference on Verification, Model Checking and Abstract Interpretation*, 2005.

21. 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.

22. O. Lee, H. Yang, and K. Yi. Automatic verification of pointer programs using grammar-based shape analysis. In *ESOP*, 2005.

23. 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.

24. 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.

25. S. McPeak and G. C. Necula. Data structure specifications via local equality axioms. In *CAV*, pages 476–490, 2005.

26. A. Møller and M. I. Schwartzbach. The Pointer Assertion Logic Engine. In *Programming Language Design and Implementation*, 2001.

27. S. S. Muchnick and N. D. Jones, editors. *Program Flow Analysis: Theory and Applications*. Prentice-Hall, Inc., 1981.

28. G. Nelson. Verifying reachability invariants of linked structures. In *POPL*, 1983.

29. A. Podelski and T. Wies. Boolean heaps. In *SAS*, 2005.

30. W. Pugh. Skip lists: A probabilistic alternative to balanced trees. In *Communications of the ACM 33(6):668–676*, 1990.

31. T. Reps, M. Sagiv, and A. Loginov. Finite differencing of logical formulas for static analysis. In *Proc. 12th ESOP*, 2003.

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

33. T. Wies. Symbolic shape analysis. Master's thesis, Universität des Saarlandes, Saarbrücken, Germany, Sep 2004.

34. T. Wies, V. Kuncak, P. Lam, A. Podelski, and M. Rinard. On field constraint analysis. Technical Report MIT-CSAIL-TR-2005-072, MIT-LCS-TR-1010, MIT CSAIL, November 2005.

35. G. Yorsh, T. Reps, and M. Sagiv. Symbolically computing most-precise abstract operations for shape analysis. In *10th TACAS*, 2004.

36. G. Yorsh, T. Reps, M. Sagiv, and R. Wilhelm. Logical characterizations of heap abstractions. *TOCL*, 2005. (to appear).

37. G. Yorsh, A. Skidanov, T. Reps, and M. Sagiv. Automatic assume/guarantee reasoning for heap-manupilating programs. In *1st AIOOL Workshop*, 2005.