Program Analysis With Write Invariant Properties

Mark Marron[1], Manuel Hermenegildo[1], and Ondřej Lhoták[2]

[1]IMDEA-Software, {mark.marron, manuel.hermenegildo}@imdea.org
[2]University of Waterloo, olhotak@uwaterloo.ca

Abstract. This paper introduces a general purpose method, *write invariant properties*, for improving the precision of heap analysis techniques at a minimal computational cost. This method is specifically focused on eliminating the imprecision introduced when program states from multiple call paths are merged at call sites when using partially call-context sensitive interprocedural analysis techniques. The concept of *write invariant properties* allows the recovery of many important classes of information such as collection sizes, null pointer properties and object allocation sites.
The concept of *write invariant properties* is based on the identification of heap object properties that are invariant during a method call provided certain parts of various objects are unmodified. By using a heap domain that can track this write information during the analysis we can extract the information for a given write invariant property at call entry and then, at the return of the call, we can assert that these properties must still hold (provided the required parts of the object are not modified). This paper presents a definition for *write invariant properties* in the concrete heap, translates this definition in a form usable in the abstract heap domain and integrates this into a basic partially call-sensitive analysis framework.

1 Introduction

Recent work on shape analysis has explored tradeoffs between precision and computational tractability that arise when employing various approaches to analyzing method calls. Since using a fully call-context insensitive analysis or a fully call-context sensitive analysis results in an imprecise or computationally intractable shape analysis [4,14] there has been significant interest in developing techniques that balance the precision needs of the shape analysis with the associated computational costs. This has led to the development of various heuristics, both for how calls are handled and how the abstract heap model is constructed. This includes work on partially call-context sensitive analysis techniques [7,9,13,15,19] (which selectively merge contexts for different call paths to improve performance), specialized join definitions that preserve important information [10,19] (by reducing the information loss when merging contexts in a partially call-context sensitive analysis) and the use of project and extend operations to minimize the number of call contexts [5,9,15,17,18] (these operations also reduce the information loss due to the merging of contexts in a partially call-context sensitive analysis). This paper proposes a new concept, *write invariant properties*, which can be used in conjunction with these techniques to further improve the precision of various heap analysis techniques while imposing a negligible increase on the computational cost of the analysis. The concept of *write invariant properties* allows us to recover, for important classes of information such as null pointers or collection sizes, much of the precision that is lost when using partially call-context sensitive analysis techniques [9, 13–15].

The concept of *write invariant properties* is an application of the basic concept that if a local procedure does not modify a given argument (or memory location) then properties that were true for the argument before the method call must also be true after the method return. This idea is the basis for *project/extend* operations [9, 17, 18], and the common heuristic of treating *pure methods* as *no-ops* [1, 6]. However, instead of working at the fairly coarse levels of entirely *pure methods* or visibility (for the project/extend operations) our definition of *write invariant properties* uses a more precise, object and field sensitive, notion of which parts of the heap must not be modified during a method call. By precisely tracking the objects and fields that are not modified by a given method we can eliminate imprecision by asserting that properties of certain objects that held at call entry also hold at the return.

The *write invariant properties* presented in this paper have a complementary relationship to much of the work on partially call-context sensitive style analysis [7, 9, 13–15]. The use of various forms of partial context sensitivity trade precision for improvements in analysis scalability and performance. These techniques use various approaches for merging the abstract models from several call sites before analyzing a call, to reduce the number of times each method in the program needs to be analyzed, and merging the models that result from the call at return, to reduce the cardinalities of the resulting disjunctive model sets that the analysis must manage. Since these approaches merge program states from many different flow paths and then propagate these states back there can be a substantial amount of imprecision introduced by the cross propagation of infeasible program states. The assertion of invariant properties allows the elimination of these infeasible states for several common cases.

We begin with a motivating example and a brief introduction of the parametric labeled storage shape graph (*lssg*) model, Section 3, that we use to illustrate the main contributions of this paper. These contributions are contained in the definition of *write invariant properties*, (Definitions 1 and 3) the generic formulation of how this concept can be applied to a range of heap properties and the incorporation of *write invariant* property information into a partially call-context sensitive analysis (Section 4).

2 Motivating Example

Consider the example in Figure 1. If we perform a disjunctive analysis using a simplified version of the storage shape graph model described in Section 3, then after the loop initialization the analysis is able to determine that program is in a state described by the abstract graph in Figure 2a or the abstract graph in Figure 2b.

The analysis we use tracks the *emptiness* of allocated collections (classes which inherit from `java.util.AbstractCollection`) via a simple three element domain. This domain consists of the elements *e* (which indicates empty collections), *!e* (which indicates non-empty collections), and *?e* (which indicates that the collection could be empty or non-empty). Thus after analyzing the loop a disjunctive analysis will produce the two abstract models shown in Figures 2a and 2b. Figure 2a shows the model that represents the special corner case where both collections are empty (the *e*

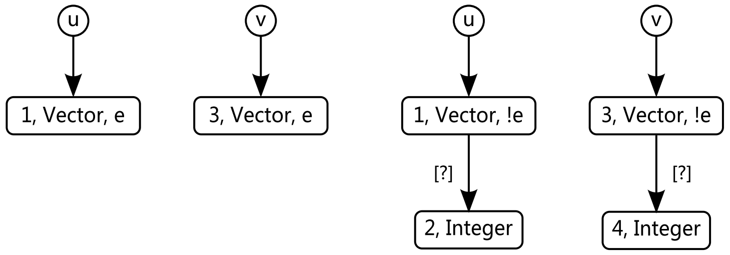
```
public static void main() {
  Random r = new Random();
  Vector<Integer> u = new Vector<Integer>(r.nextInt());
  Vector<Integer> v = new Vector<Integer>(u.size());
  for(int i = 0; i < u.size(); ++i) {
    int k = r.nextInt();
    u.set(i, new Integer(k));
    v.set(i, new Integer(k));
  }

  bubbleSort(v);
}

public static void bubbleSort(Vector<Integer> vv) {
  for(int i = 0; i < vv.size(); ++i) {
    ...
  }
}
```

Fig. 1: Running Example

labels in the nodes that represent the `Vector` objects). In this case both collections are known to be empty and thus do not contain any pointers to `Integer` objects. The other model, Figure 2b, represents the other cases where the collections are non-empty (the *!e* labels in the nodes that represent the `Vector` objects). In this figure both collections are known to be non-empty and both collections contain pointers (represented by the edges with the label `?` which represents all the pointers stored in the collection) to some `Integer` objects. To simplify the discussion of the examples and to support later labeling each node is also given a unique name from the set $\mathbb{N}$.

At the entry to the `bubbleSort` method the analysis will project out the part of the heap that is accessed by the called method [9, 15, 17, 19] and join some, or for the purposes of our example, all of these heap models into a single model [9]. While this is critical to ensuring good scalability this also results in the loss of the relational information that was preserved by using the disjunctive set of models. The result of this join is shown in Figure 3a. This figure shows that now the argument `Vector` *may* be either empty or non-empty indicated by the ?*e* label in the node representing the `Vector`. This imprecision has a minimal effect on the accuracy of the analysis of the `bubbleSort` method which is written in a general way and tests all sizes that are needed and since both empty and non-empty collections are passed into `bubbleSort` there is no loss (in this case) in the precision of the results produced for the `bubbleSort` method.

However, the join at the entry to the method can introduce significant imprecision on return when the results of analyzing the callee body are merged with the heap that is not visible to the callee method. This precision loss is due to the loss of the relational information implicit in the disjunctive set which has been mixed by the join at call entry/exit. In our example after the return and the recombination of the callee reachable and callee unreachable sections of the heap a number of infeasible states have been

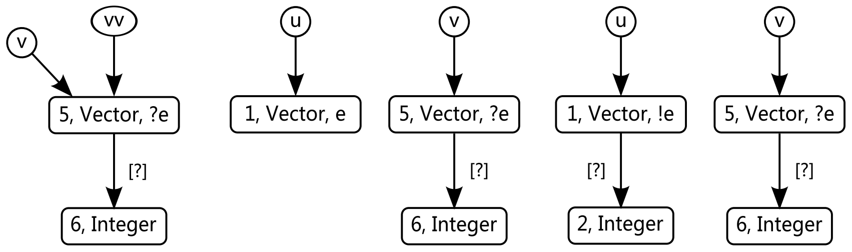


(a) Abstract Model Empty (b) Abstract Model Non-Empty

Fig. 2: Disjunctive Model after Loop

introduced and as a result the analysis has lost a substantial amount of information on the relations between the emptiness of the `Vector` collections.

Figure 3 shows an example of the introduction of this imprecision. Via a quick examination of the source it is obvious that if the `Vector` pointed to by variable `u` is empty then the `Vector` pointed to by variable `v` must also be empty and similarly if `v` is empty then `u` is as well (a fact which the analysis has discovered holds before the `bubbleSort` method call). However, after this method call the partial call-context sensitivity has mixed information from multiple models in the disjunctive set together introducing significant imprecision. In one of the resulting models, Figure 3b, although the `Vector` pointed to by `u` is still known to be empty the `Vector` pointed to by variable `v` *may* be non-empty and *may* contain references to some data objects, which is clearly infeasible. Similarly in Figure 3c the `Vector` pointed to by `v` *may* be empty even though the `Vector` pointed to by `u` is known to be non-empty. This introduction of infeasible states can have a significant impact on the precision of later analysis steps, and on the accuracy of the final analysis results, as the analysis will continue to produce infeasible program states based on the imprecision introduced in the join.



(a) `sort` Result (b) Return Orig. Empty (c) Return Orig. Non-Empty

Fig. 3: Info Loss at Return

To minimize the imprecision introduced by the joins at call sites and returns many analyses extend the domain to track relational information in addition to summary properties [4, 14]. In our example this could be done by extending the domain to track equalities on collection emptiness or size. However, these types of extensions are non-trivial to implement, increase the computational cost of the analysis, and must be designed and implemented for each property. As an alternative we propose the use of *write invariant properties* which are properties that, provided certain heap locations are not written, *must* be preserved across calls. Using properties that are *write invariant* we can, on call return, assert that certain properties which held at call entry *must* hold at call return. Thus, we can eliminate much of the imprecision caused by the call/return join operations. In our example we know that since the size of `Vector v` is not be changed by the call to `bubbleSort` then at call return the size must be the same as at call entry and we can assert this invariance in the analysis.

3 Concrete Heap and Labeled Storage Shape Graph

To analyze a program we first transform (via a modified compiler frontend) the Java 1.5 source into a semantically equivalent program in a simpler to analyze core language called MIL (Mid-level Intermediate Language). The MIL language is statically typed, has method invocations, conditional constructs, exception handling and the standard looping statements. The state modification and expressions cover the standard range of program operations: load, store, and assignment along with logical, arithmetic, and comparison operators. We associate with each statement and each control flow structure a program location $\ell \in \mathbb{N}$. During the transformation to MIL we also load in specialized standard library implementation stubs, so we can analyze programs that use classes from `java.util`, `java.lang` or `java.io`.

3.1 Concrete Memory Model

The semantics of memory are defined using an environment, mapping variables into values, and a store, mapping addresses into values. We refer to the environment and the store together as the concrete heap, which is treated as a labeled, directed multi-graph (V,O,R) where each $v \in V$ is a variable, each $o \in O$ is an object on the heap and each $r \in R$ is a reference. The set of references $R \subseteq (V \cup O) \times O \times L$ where L is the set of storage location identifiers (a variable name in the environment, a field identifier for references stored in objects, or an integer offset for references stored in arrays/collections). We use the terminology, *region* of memory $\mathfrak{R}$, to denote a subset of objects in the heap O.

Definition 1 (Concrete Write Invariant Property). *A predicate P is* write invariant *if: For all methods m and for all possible calls to m, $\exists$ a storage location $\lambda \in L$ s.t. $\forall$ objects o in the region of the heap that are visible in m if location specified by λ is not written in o then $P^m_{exit}(o) = P^m_{entry}(o)$.*

A number of important properties that are of interest in program analysis satisfy this property such as: collection emptiness, values of scalar fields and allocation site information. This technique can be generalized (Section 5) to handle properties that are parametrically invariant based on a particular field such as nullity of fields or that involve multiple nodes such as aliasing, reachability and shape.

3.2 Abstract Labeled Storage Shape Graph

Our abstract heap domain is based on the *storage shape graph* [2, 3, 10] approach. An *abstract storage graph* is a tuple of the form $(\hat{V}, \hat{N}, \hat{E})$, where $\hat{V}$ is a set of abstract nodes representing the variables, $\hat{N}$ is a set of abstract nodes (each of which abstracts a region $\Re$ of the heap), and $\hat{E} \subseteq (\hat{V} \cup \hat{N}) \times \hat{N} \times \hat{L}$ are the graph edges, each of which abstracts a set of references, and $\hat{L}$ is a set of abstract storage *offsets* (variable names, field offsets, or the special offset *?* for references stored in collections). Each node is also given a unique name (within the graph) from the set $\mathbb{N}$ to simplify labeling and discussion. We extend this definition with a set of additional relations, $\hat{U}$, that must hold for the given concrete heap h to be a valid concretization of the given abstract graph g. The *labeled storage shape graphs (lssg)*, which we refer to simply as *abstract graphs*, are tuples of the form $(\hat{V}, \hat{N}, \hat{E}, \hat{U})$.

Definition 2 (Valid Concretization of a *lssg*). *A given concrete heap h is a* valid concretization *of a* labeled storage shape graph *g if there are functions Π_v, Π_o, Π_r such that the following hold:*

- *$\Pi_v : V \mapsto \hat{V}$, $\Pi_o : O \mapsto \hat{N}$ and $\Pi_r : R \mapsto \hat{E}$ are functions (and Π_v is 1-1).*
- *h, Π_v, Π_o, and Π_r satisfy all the relations in $\hat{U}$.*
- *h, Π_v, Π_o, and Π_r are* connectively consistent *with g.*

Where h, Π_v, Π_o, Π_r are connectively consistent *with g if:*

- *$\forall\, o_1, o_2 \in O$ s.t. $(o_1, o_2, p) \in R$, $\exists\, e \in \hat{E}$ s.t. $e = \Pi_r((o_1, o_2, p))$, e starts at $\Pi_o(o_1)$, ends at $\Pi_o(o_2)$, and $e.\mathrm{offset} = \alpha_{offset}(p)$.*
- *$\forall\, v \in V$, $o \in O$ s.t. $(v, o, v) \in R$, $\exists\, e \in \hat{E}$ s.t. $e = \Pi_r((v, o, v))$, e starts at $\Pi_v(v)$, ends at $\Pi_o(o)$, and $e.\mathrm{offset} = v$.*

In Section 2 we informally introduced the *type* relation on the nodes in the abstract graph which restricts the types of the objects that each node abstracts. To formally define how this (and other relations) restrict the set of concrete heaps that a given abstract graph g represents we need to describe the relations between a given concrete heap h and the abstract graph g. In particular we need to look at the pre-images of the nodes and edges from g in h.

Notation 1 (Pre-Images of Abstract Graph in Concrete Heap) *Given an abstract graph $g = (\hat{V}, \hat{N}, \hat{E}, \hat{U})$, a concrete heap $h = (V, O, R)$, and the functions Π_v, Π_o, Π_r we introduce the following notation for the image of an edge or node from the abstract graph in the given concrete heap:*

1. *The notation $h \downarrow_g e$ is used to denote the set of references in the concrete heap h that are in the pre-image of e under the functions, $\{r \in R \mid \Pi_r(r) = e\}$.*
2. *The notation $h \downarrow_g n$ is used to denote the concrete region ($\Re$) that is the pre-image of n under the functions, where $\Re = \{o \in O \mid \Pi_o(o) = n\}$.*

3.3 Abstract Relations (in $\hat{U}$)

In previous work [8, 10, 11] this model is used to analyze a range of shape, sharing and heap region properties. While the properties discussed therein are critical to precisely analyzing these programs, we do not need all of this information in order to apply the *write invariant* technique described in this paper. Thus, to simplify the discussion and to focus on the novel concepts we only introduce the *identity* and read-write properties which are needed by our formulation of *write invariant properties* along with *type* and *collection size* properties as examples. We refer the interested reader to [12, 16] for a more extensive discussion on techniques for modeling the heap-carried data dependence information.

Region Types. The region *type* relation is used to track the types of all the objects that may appear in a given region. For each node we track a set of concrete types $\{\tau_1, \ldots, \tau_k\}$. A concrete heap h is valid concretization of the abstract graph g with respect to this relation if $\forall$ nodes n, where $h \downarrow_g n = \{\texttt{typeof}(o) \mid \text{object } o \in \Re\} \subseteq n.type$.

Collection Emptiness. The sizes of the collections are tracked with a relation on the nodes that abstract objects which subclass `java.util.AbstractCollection`, which we assume has a field `size` that is written by any method that uses or updates the cardinality of the collection object. For each node that may represent a collection we associate an *emptiness* value from the set $\{e, !e, ?e\}$ with the node. Given a node n where $h \downarrow_g n = \Re$ then:

$$\forall \text{ collection objects } o_c \in \Re, o_c.\texttt{empty()} \in \begin{cases} \{true\} & \text{if } n.empty = e \\ \{false\} & \text{if } n.empty = !e \\ \{true, false\} & \text{if } n.empty = ?e \end{cases}$$

Node Identity. In [12] we used the notion of *identity* tags to track how objects in the heap are rearranged inside a given method call. The nodes in the abstract graph that represents the state of the program at call entry implicitly define partitions ($\Pi_\iota : O \mapsto \mathbb{N}$) of the objects in the concrete heaps represented by this graph. We can use this initial partition as a baseline partition and at any later point in the method we can examine how the objects in the heap have been rearranged with respect to this baseline partition.

For each node we track a set of identity tags *identity* which contains the baseline identity tags of all the partitions that any of the objects abstracted by this node may belong to. Using the initial partition of the concrete heap Π_ι, a given concrete heap h is a valid concretization of the abstract graph g with respect to the *identity* tag set if $\forall$ nodes n, where $h \downarrow_g n = \Re$: $\{\Pi_\iota(o) \mid \text{object } o \in \Re\} \subseteq n.identity$.

Read-Write Locations. Each node may represent a number of objects of different classes ($\tau_1 \ldots \tau_m$) and each class may have many fields ($f^1_{\tau_i} \ldots f^n_{\tau_i}$). We define two relations from the *identity tags* to a list of *lastWrite* and a list of *lastRead* locations for each locally written/read field. These lists contain pairs of the form (f, ℓ) where f is a field and ℓ is the most recent (with respect to the order given by the abstract syntax tree representation) statement the field *may* have been written/read. A given concrete heap h is a valid concretization of the abstract graph g with respect to the *read-write* set if $\forall$ nodes n, where $h \downarrow_g n = \Re$:

– $\forall o \in \Re$ ($\forall$ fields $f \in o$, $(f$ is not written locally$) \vee (f$ was last written at line $\ell \wedge \exists$ $(f, \ell') \in n.lastWrite$ s.t. $\ell \leq \ell'))$.

– $\forall o \in \Re\ (\forall$ fields $f \in o, (f$ is not read locally$) \vee (f$ was last read at line $\ell \wedge \exists\,(f, \ell')$ $\in n.\mathit{lastRead}$ s.t. $\ell \leq \ell'))$.

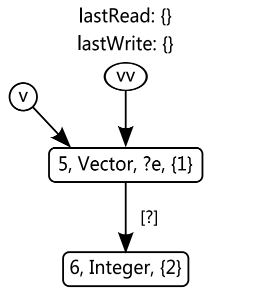
In our extended domain each variable node in the heap is represented with just the variable name and each region node is represented as a record $[\mathit{label}, \mathit{type}, \mathit{empty}, \mathit{identity}]$. The *lastRead* entry is a set of $(\mathit{label}: f_1\text{-}\ell_1 \ldots f_k\text{-}\ell_k)$ tuples, for the fields in each node that have been read in the local scope, and *lastWrite* is a set of $(\mathit{label}: f_1\text{-}\ell_1 \ldots f_k\text{-}\ell_k)$, for the fields in each node that have been written in the local scope.

3.4 Example: Entry and Exit State of `bubbleSort`

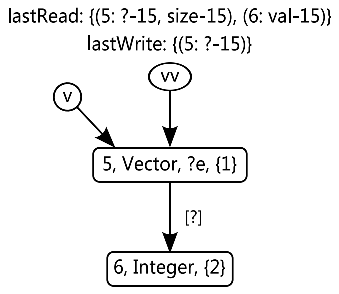
In Figure 4 we show the abstract graph at the entry to the `bubbleSort` method and the abstract graph produced by the analysis at the exit of the method.

Figure 4a is the abstract graph representing the state of the program at the entry to `bubblesort`. We have added the fresh *identity* tag, 1, to the node representing the `Vector` object and the fresh *identity* tag 2 to the node representing the `Integer` objects. Since this abstract graph represents the state at the entry of the method, no fields have been read or written, thus the *lastRead* and *lastWrite* relations are empty.

Figure 4b shows the abstract graph that represents the program state at the exit of the method. Since the pointer writes in the sort loop only reorder the elements in the `Vector` the graph connectivity is unchanged. As the method reads the size of the `Vector` and the val fields of the `Integer` objects in the sorting loop the analysis has updated the *lastRead* relation to indicate that these reads may have occurred in the body of the loop on line 15, the *size-15* entry in the list for node 5 and the *val-15* entry in the list for node 6 respectively. As the contents of the `Vector` may be both read and written during the sort loop the analysis has added the entry *?-15* for node 5 in both the *lastRead* and *lastWrite* relations (the special offset *?* is used to represent all of the indices in the `Vector`).



(a) Entry to `bubbleSort`



(b) Exit of `bubbleSort`

Fig. 4: Entry and Exit Model for `bubbleSort`

4 Write Invariant Properties

Given the ability to track heap locations that may be written in the abstract graph model we can now translate the definition of *write invariant properties* (Definition 1) in the concrete heap to a definition that we can use during the static analysis.

Definition 3 (Abstract Write Invariant Property). *An abstract relation $u \in \hat{U}$ is* write invariant *if: For all methods m and for all possible calls to m, $\exists$ a* dependent location *$\lambda \in \hat{L}$ s.t. $\forall$ nodes $n \in g_{exit}$ at the exit of m if location specified by λ is not written in n then $u^m_{exit}(n) \leq \bigsqcup\{u^m_{entry}(n') \mid n' \in g_{entry} \wedge n'.\text{identity} \cap n.\text{identity} \neq \emptyset\}$ where g_{entry} is the abstract graph at method entry.*

The first part of the definition is a straight forward translation of the *concrete write invariance* definition (Definition 1) into the abstract domain. In the abstract definition the relation between the value of the node property at the entry and the exit is a function of how object identity is tracked in the abstract graphs. The analysis uses the partition of the heap given by the abstract graph g_{entry} representing the state of the program at the entry to a given method call as a canonical partition. At the exit of the method it is possible that the abstract graph has been reorganized so that a node *n* represents objects from several of these canonical partitions and thus the property value for *n* must safely approximate this possibility (which is done by taking the upper bound of the property values for the canonical nodes n' that may have been split/merged in the method body to produce *n*).

Using this definition in the running example, the *empty* property of a given `Vector` object is invariant as long as the *size* field is not written. Thus if we can determine that a node represents `Vector` objects and that the `size` fields of all the objects abstracted by this node are not written then we can assert that the *empty* property of the node is unchanged during the method call.

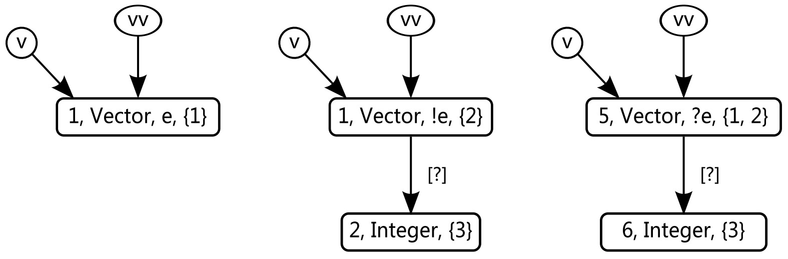
4.1 Extracting Write Invariant Properties

Once we have selected the properties $\{P_1, \ldots, P_k\}$ that we are interested in preserving, and have determined which abstract relations in $\hat{U}$ they correspond to and the abstract storage locations they depend on, we need to extract the information for each node at call site entry. This is done via a property specific computation provided by the domain implementation (*propertyExtract*) and then associating this property with the *dependent location* λ that the node/property depends on. The general method for extracting this information is shown in Algorithm 1.

In Algorithm 1 we iterate over each node extracting the value of each property of interest at call entry (via the user supplied *propertyExtract* method) and the field information that this property is dependent on. This information is then grouped together with the unique *identity* tag for the node (the identity tag must be unique since at call entry each node is given a fresh tag). This map now contains all the identity tags that are associated with a *single* abstract graph (so we can later remove any tags that are spuriously mixed in from other abstract graphs during the joins at entry/exit) and has the property/dependence information for the objects abstracted by each node (and has

Algorithm 1: extractWriteInvProperties

```
input  : graph $g$, method $propertyExtract$: $\hat{N} \mapsto D_u$ (set of property values)
output: map $M_I : \mathbb{N} \mapsto D_u$
$M_I \leftarrow$ new map;
foreach node $n \in g$ do
    $prop \leftarrow propertyExtract(n)$;
    $M_I$.addEntry(uniqueElement($n.identity$), $prop$);
return $M_I$;
```



(a) Empty Model (b) Non-Empty Model (c) Join Model

Fig. 5: Projected Model and Extracted Info with Join

represented them in a way such that regardless of structural transformations in the callee they can be precisely associated with the resulting model).

As the only required extension to the abstract model is the definition of the *propertyExtract* method, which in this case is simply $propertyExtract(n) \doteq n.size$, this approach is both simple to implement and computationally inexpensive when compared to the complexities and costs of extending the existing domain to track relational properties between the emptiness of collections.

Given the heap in our running example we begin by applying the project operation to each abstract model in the disjunctive set and giving fresh *identity* tags to each of the projected sections resulting in the abstract models shown in Figures 5a and 5b. Figure 5a shows the result of projecting the abstract graph from Figure 2a and giving the projected section of the model fresh *identity* tags. The application of the *extractWriteInvProperties* algorithm to this model gives the map of one element $\{1 \mapsto e\}$, which tells us that for this particular model as long as the *size* field of objects abstracted by the node with the identity tag 1 is not written then at exit these objects will only contain empty `Vectors`. Similarly, the abstract graph shown in Figure 5b is the result of projecting the abstract graph from Figure 2b and giving the projected section of the model fresh *identity* tags. The application of the *extractWriteInvProperties* algorithm to this model gives the map of one element $\{2 \mapsto !e\}$.

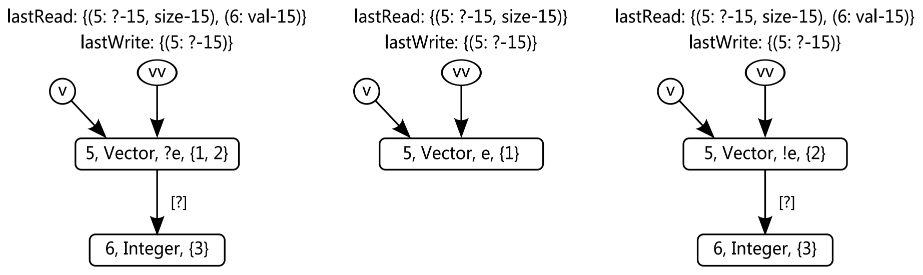
4.2 Restoring Write Invariant Properties

After the extraction of the invariant properties we can analyze the callee body as usual. On return we then create a copy of the return model for each abstract model and apply the appropriate invariant map to assert the more precise invariant results where possible. The method for doing this is shown in Algorithm 2. The algorithm first restricts the identity set to tags which are associated with the abstract model that this projected model and map are associated with. If the *dependent location* has not been written in the node then the algorithm computes and asserts the upper bound property value based on the possible identity classes that the node may represent (as given by the set of identity tags). The actual assertion is done via the domain specific *assertPropUpperBound* method, which for the emptiness property is $assertPropUpperBound(n, d_u) \doteq n.size = max(n.size, d_u)$.

Algorithm 2: assertInvProperties

```
input  : graph $g$, dependent location $\lambda$, map $M_I : \mathbb{N} \times D_u$, method assertPropUpperBound
output: graph $g'$
foreach node $n \in g$ do
    $n.identity \leftarrow n.identity \cap domain(M_I)$;
    ok $\leftarrow$ the $\lambda$ location was not modified in $n$ during the call;
    if ok then
        $d_u \leftarrow \bigsqcup\{M_I(\iota) \mid \iota \in n.identity\}$;
        assertPropUpperBound($n$, $d_u$);
return $g$;
```

Figure 6a shows the abstract model that the analysis produces for the `bubbleSort` method when given the abstract model in Figure 5c. To return this information back into the caller scope we create two copies of this model (one for each of the models in the disjunctive set at the call site) and apply the two maps that we produced with the *extractWriteInvProperties* method at the entry to the call.



(a) Analysis Result of `sort` (b) Assert Empty (c) Assert Non-Empty

Fig. 6: Result and Assertion of Invariant Properties

Figure 6b shows the result of applying the map $\{1 \mapsto e\}$ that we extracted for the state in Figure 5a. In this figure we have removed all the identity tags that are not in the domain of the map (since they came from other models that were merged in the partially call-context sensitive analysis algorithm). According to the *lastWrite* information, the node with label 5 has not had the *size* offset written in the callee method (`bubbleSort`) and the only value in the *identity* set is 1, so we can assert that the node only abstracts empty `Vectors`. Thus in the figure we have replaced the *?e* property *value* with the *e* value and since we know an empty collection does not contain any pointers we can remove the out edge (and the node with the label 6 since it is unreachable). Similarly we can apply the map $\{3 \mapsto !e\}$ to the abstract model from Figure 6a to get the abstract model shown in Figure 6c, allowing us to improve the precision of the model by asserting that the `Vector` is non-empty.

4.3 Full Call/Return Algorithm

With the supporting write invariance methods defined we can describe the complete call/return analysis Algorithm 3. The algorithm is based on the method for analyzing call/returns in [9, 15]. The base algorithm begins by separating the sections of the heap that are reachable/unreachable in the callee method [5, 9, 17] via the *extractReachUnreach* method. After this separation the reachable portions of each model are merged into a single context (thus method call/return is partially insensitive) and on return the single result of analyzing the callee method is extended back, via the *mergeReachUnreach* method, into the unreachable portions of each of the abstract graphs that we obtained in the earlier project operation.

To provide support for the write invariant properties we add two steps into the process. The first is immediately after the *extractReachUnreach* method where for the reachable portion of each graph model we extract the invariant properties using the *propertyExtract* method. The second is after the analysis of the callee method where instead of merging in the result (which is an approximation of the result for all the reachable sections) we assert the invariant properties that correspond to the unreachable section we are about to merge with via the *assertInvProperties* (thus reducing the spurious imprecision introduced by the partially insensitive manner in which the elements in the disjunctive set at call entry/exit are handled).

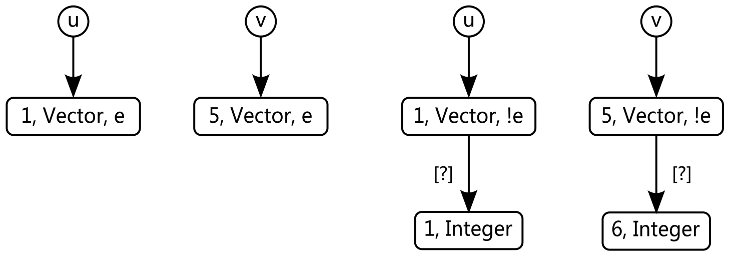
We note that the additional computational cost of performing the extraction and assertion of the *write invariant* properties is the cost of applying the *propertyExtract* method to each node in the graph (linear in the size of the model) and the cost of applying the *assertInvProperties* method to each node in the graph (also linear in the size of the model). Both of these operations represent negligible costs in comparison to the project/extend and join operations in the call which, depending on the exact model used, are usually both super linear.

In our running example the application of the *assertInvProperties* function results in the two abstract graphs in Figures 6b and 6c which are combined with the sections of the heap from the caller scope that cannot be accessed by the callee via the *mergeReachUnreach* method. The resulting abstract graphs are shown in Figures 7a and 7b. These figures show that the assertion of the invariance of the `Vector` *emptiness* properties has allowed the analysis to completely recover the relational information present in the disjunctive set after the analysis of the `bubbleSort` call. As desired the analysis has determined that for the case where the `Vector` is empty/non-empty before the

Algorithm 3: analyzeCall

```
input  : call $m$, abstract state $\sigma = [g_1 \ldots g_k]$, *dependent location* $\lambda$, method
         *propertyExtract*: $\hat{N} \mapsto D_u$, method *assertInvProperties*: graph $\mapsto$ graph
output: abstract state $(g'_1 \ldots g'_k)$
unreach $\leftarrow []$;
inv $\leftarrow []$;
$g_{\text{call}} \leftarrow \bot$;
foreach $i \in [1,k]$ do
    $(g_{\text{reach}}, g_{\text{unreach}}) \leftarrow$ extractReachUnreach($\sigma[i]$, $m$);
    unreach += $g_{\text{unreach}}$;
    inv += extractWriteInvProperties($g_{\text{reach}}$, propertyExtract);
    $g_{\text{call}} \leftarrow g_{\text{call}} \sqcup g_{\text{reach}}$;
$g_{\text{res}} \leftarrow$ analyze(m.body, $g_{\text{call}}$);
out $\leftarrow []$;
foreach $i \in [1,k]$ do
    $g_{\text{out}} \leftarrow$ assertInvProperties(copy($g_{\text{res}}$), $\lambda$, inv[$i$]);
    out += mergeReachUnreach($g_{\text{out}}$, unreach[$i$]);
return out
```

call then it is empty/non-empty after the call and it has done this without the introduction of explicit relational information on the emptiness of collection objects or passing disjunctive sets of models to/from method calls.



(a) Result Orig. Empty (b) Result Orig. Non-Empty

Fig. 7: Result After `bubbleSort` with Invariant Properties

In this case we note that the results are precise in the sense that the use of write invariant properties results in no information loss from merging the graphs at call entry/return. While this is not true in general (suppose if instead of `bubbleSort` we had called a method that modified the `Vector` size) the approach provides a simple way to improve the accuracy of partially call-context sensitive shape analysis techniques with a minimal cost. Thus this technique is not suitable for properties where the preservation of relational information about these properties across calls is critical but for many properties such as collection size, pointer must nullity, and allocation site tracking the use of write invariant information is an easy to implement and computationally inexpensive way (as opposed to adding/implementing specialized relational domains and associated project/extend support) to improve analysis precision.

5 Extensions

To generalize the range of properties that write invariance can be applied to and to simplify the definition of certain write invariant properties we can define a few extensions to the formulations presented in this paper.

Parametric Dependent Locations. The first extension is useful for supporting the definition of properties such as null-pointer invariance. If we use the formulation as defined previously we would need to define the nullity of each field as a distinct write invariant property. However if we allow the definition to be parametric in both the *dependent field* λ and the domain value (i.e. the method *propertyExtract* now returns a domain property in D_u as well as a dependent field), the nullity of all fields can be defined as single operation.

Multi-Object Invariant Properties. The second extension allows us to track additional properties such as aliasing which involve multiple objects by extending the *propertyExtract* method and the *assertPropUpperBound* method to extract sets of *dependent fields* and modifying the extraction and assertion methods to process groups of nodes. As this requires the processing of sets of nodes in the graph it can be more computationally expensive than the basic formulation of *write invariant properties* thus we want to limit the size of the node sets that we allow to be processed (e.g. only processing sets of size 2 or 3).

6 Case Studies

Em3d. The Em3d program from our modified version of the JOlden suite[1] computes electro-magnetic field values in a 3–dimensional space by constructing a `Vector` of `ENode` objects, each representing an electric field value and a second `Vector` of `ENode` objects, each of which represents a magnetic field value. To compute how the electric/magnetic field value for a given `ENode` object is updated at each time step the `computeNewValue` method uses an array of `ENode` objects from the opposite field and performs a convolution of these field values and a scaling vector, updating the current field value with the result.

The construction and initialization of the heap structures in this program contains a situation similar to the motivating example in this paper. In particular when computing a subset of `ENode` objects in the opposite field for use in the update of a given field value the partially call-context analysis will generate a number of infeasible states such as: a `Vector` that was empty before the subset computation may be non-empty after the subset computation and a state where a non-empty subset was computed from an empty `Vector`. Both of these infeasible states can be eliminated by the use of the *write invariant* emptiness property.

List Copy. Our basic `LinkedList` test code includes a simple list copy routine. If we are checking for null pointer violations the analysis produces a set of several abstract graphs when analyzing the method call `makeListSize(k)`. These include a list of size greater than 2 where the pointer `this.next` *must* be non-null and a list of size 1

[1] See `www.software.imdea.org/~marron/software/software.html` for benchmark code, examples of the analysis results, and an executable analysis demo.

where the `this.next` pointer *must* be null. Later calls is to the `listCopy` routine may then merge these models mixing them into a single state where the nullity of the `this.next` pointer is unknown. At return this results in the infeasible states where a call to `listCopy` of a list of size 1 results in a list of size larger than 1 and a list of size larger than 2 results in a list of size 1. Again the use of a pointer nullity as a *write invariant* property prevents the creation and propagation of these infeasible states.

7 Related Work

There has been a substantial amount of work on how to balance precision and computational costs when performing interprocedural shape analyses [5, 7, 9, 13, 15, 17, 19]. This work approaches the problem in a number of different ways. Most commonly the work builds on some form of partially call-context sensitive heuristics [6,7,9,13,15,19] which selectively merge call contexts to reduce the computational cost of the analysis while still maintaining an acceptable level of precision. A common technique to further improve both the performance and precision of these partially call-context sensitive approaches is to use projection/extension operators to restrict the portion of the heap that is passed to/from the callee method to only what is visible in the callee method [5, 9, 15, 17, 18]. This reduces the initial imprecision introduced by using a partially context sensitive analysis and the loss of information going into the callee. The concept of *write invariant properties* addresses the imprecision introduced by passing these merged states back to the caller scope.

The project/extend techniques along with the idea of treating *pure* methods as special cases in the analysis [1, 6] can be conceptually viewed as coarse special case applications of the *write invariant* properties introduced in this paper as they use the fact that there are no modifications to entire sections of the heap to assert that the heap is unchanged between call entry and exit. As all of these techniques play complementary roles in managing the cost of a heap analysis while ensuring that the results are still precise enough to be useful we see the concept of *write invariant* properties as being an additional tool that can be integrated easily into existing analyses to further improve their performance.

8 Conclusion

This paper introduces the concept of *write invariant* properties which we use to minimize the imprecision that results from the merging of program states from different call paths in partially call-context sensitive heap analysis algorithms. This concept can be used in conjunction with existing techniques (project/extend operations, special treatment of pure methods, and the extension of analysis domains with relational information) to further improve the precision of various heap analysis techniques and to do so in a manner that has a negligible impact on the computational cost of the analysis (the required operations are linear in the size of the model). We believe that this technique is a useful step in the development of robust and scalable analysis techniques and as our case studies indicate that this approach can be very useful for certain properties (such as null pointers and the collection emptiness example).

References

1. B.-Y. E. Chang and X. Rival. Relational inductive shape analysis. In *POPL*, 2008.
2. D. R. Chase, M. N. Wegman, and F. K. Zadeck. Analysis of pointers and structures. In *PLDI*, 1990.
3. S. Chong and R. Rugina. Static analysis of accessed regions in recursive data structures. In *SAS*, 2003.
4. P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *POPL*, 1979.
5. A. Gotsman, J. Berdine, and B. Cook. Interprocedural shape analysis with separated heap abstractions. In *SAS*, 2006.
6. C. Lattner, A. Lenharth, and V. Adve. Making context-sensitive points-to analysis with heap cloning practical for the real world. In *PLDI*, 2007.
7. O. Lhoták and L. Hendren. Evaluating the benefits of context-sensitive points-to analysis using a bdd-based implementation. *ACM Trans. Softw. Eng. Methodol.*, 2008.
8. D. K. Mark Marron and M. Hermenegildo. Identification of logically related heap regions. In *Submission*, 2009.
9. M. Marron, M. Hermenegildo, D. Stefanovic, and D. Kapur. Efficient context-sensitive shape analysis with graph based heap models. In *CC*, 2008.
10. M. Marron, D. Kapur, D. Stefanovic, and M. Hermenegildo. A static heap analysis for shape and connectivity. In *LCPC*, 2006.
11. M. Marron, M. Méndez-Lojo, M. Hermenegildo, D. Stefanovic, and D. Kapur. Sharing analysis of arrays, collections, and recursive structures. In *PASTE*, 2008.
12. M. Marron, D. Stefanovic, D. Kapur, and M. Hermenegildo. Identification of heap-carried data dependence via explicit store heap models. In *LCPC*, 2008.
13. A. Milanova, A. Rountev, and B. Ryder. Parameterized object sensitivity for points-to analysis for java. *ACM Trans. Softw. Eng. Methodol.*, 2005.
14. F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.
15. G. Puebla, J. Correas, M. Hermenegildo, F. Bueno, M. G. de la Banda, K. Marriott, and P. J. Stuckey. A generic framework for context-sensitive analysis of modular programs. In *Program Development in Computational Logic, A Decade of Research Advances in Logic-Based Program Development*, LNCS. Springer-Verlag, Heidelberg, Germany, 2004.
16. M. Raza, C. Calcagno, and P. Gardner. Automatic parallelization with separation logic. In *ESOP (To Appear)*, 2009.
17. N. Rinetzky, J. Bauer, T. W. Reps, S. Sagiv, and R. Wilhelm. A semantics for procedure local heaps and its abstractions. In *POPL*, 2005.
18. R. P. Wilson and M. S. Lam. Efficient context-sensitive pointer analysis for C programs. In *PLDI*, 1995.
19. H. Yang, O. Lee, J. Berdine, C. Calcagno, B. Cook, D. Distefano, and P. OHearn. Scalable shape analysis for systems code. In *CAV*, 2008.