Automatic Inference of Heap Properties Exploiting Value Domains

Pietro Ferrara[1], Peter Müller[2], and Milos Novacek[2]

[1] IBM Thomas J. Watson Research Center, USA
pietroferrara@us.ibm.com
[2] Department of Computer Science, ETH Zurich, Switzerland
{peter.mueller,milos.novacek}@inf.ethz.ch

Abstract. Effective static analyses of heap-manipulating programs need to track precise information about the heap structures and the values computed by the program. Most existing heap analyses rely on manual annotations to precisely analyze general and, in particular, recursive, heap structures. Moreover, they either do not exploit value information to obtain more precise heap information or require more annotations for this purpose. In this paper, we present a combined heap and value analysis that infers complex invariants for recursive heap structures such as lists and trees, including relations between value fields of heap-allocated objects. Our analysis uses a novel notion of edge-local identifiers to track value information about the source and target of a pointer, even if these are summary nodes. With each potential pointer in the heap, our analysis associates value information that describes in which states the pointer may exist, and uses this information to improve the precision of the analysis by pruning infeasible heap structures. Our analysis has been implemented in the static analyzer Sample; experimental results show that it can automatically infer invariants for data structures, for which state-of-the-art analyses require manual annotations.

1 Introduction

Effective static analyses of heap-manipulating programs need to track precise information about the heap structures and the values computed by a program. Heap and value information is not independent: heap information determines which locations need to be tracked by a value analysis, and information about value fields may be useful to obtain more precise heap information, for instance, to rule out certain forms of aliasing. Moreover, many interesting invariants of heap-manipulating programs combine heap and value information such as the invariant that a heap structure is a sorted linked list.

Despite these connections, heap and value analyses have often been treated as orthogonal problems. Some existing heap analyses such as TVLA [18] rely on manual instrumentation to infer invariants that combine heap and value information. However, TVLA does not support general value domains, which limits, for instance, arithmetical reasoning. Recent work addresses this issue by combining TVLA with value domains, but still requires the user to provide predicates to track and exchange information between the heap and value domains [21], or is not able to track complex invariants over recursive data structures [14]. Chang and Rival [5] present an efficient inference for combined heap and value invariants, which also relies on user-provided predicates. Other analyses do not require manual annotations [2,3], but are specific to programs that manipulate certain data structures such as singly-linked lists.

In this paper, we present a combined heap and value analysis—expressed as an abstract interpretation [8]—that infers complex invariants of heap structures. It is automatic in the sense that it uses only the information included in the program, without relying on manual annotations. Our analysis uses a graph-based abstraction of heaps, where each edge in the graph represents a *potential* pointer in the concrete heap. Each edge is associated with an abstract value state that characterizes in which concrete states this pointer might actually exist. The value states on the edges allow our analysis to represent disjunctive information in a single heap graph (like the bracketing constraints in Dillig et al.'s Fluid Updates [10]). They are also used to improve the precision of the analysis when value information implies that certain pointer chains cannot exist in concrete heaps. Our analysis can be instantiated with different value domains to obtain different trade-offs between precision and efficiency.

Like many heap analyses, we use summary nodes to abstract over sets of concrete objects. A key innovation of our analysis is to introduce *edge-local identifiers* for the source and target of each edge in the heap graph. An edge-local identifier represents a field of one particular concrete object, even when the object is abstracted by a summary node. By having identifiers per edge, the value analysis may relate the fields of the source and the target of a concrete pointer and, thus, track inductive invariants such as the sortedness of a linked list.

Example. Method increasingList in Fig. 1 creates and returns a linked list. If parameter v is non-positive, the list is empty, that is, result is null (invariant I1). Otherwise, the list satisfies the following properties: it is non-empty, that is, the result is non-null (I2), the first node has value 0 (I3), the values of all other nodes are one larger than their predecessor's (I4), and the value of the last node is v − 1 (I5). Note that these invariants imply that the list is acyclic and has v nodes.

```
 1 Node increasingList (int v) {
 2   Node result = null;
 3   int i = v;
 4   while (i > 0) {
 5     Node p = new Node();
 6     p.next = result;
 7     p.val = i - 1;
 8     result = p;
 9     i = i - 1;
10   }
11   return result;
12 }
```

Fig. 1. Running example

Fig. 2 shows the abstract state that our analysis infers at the end of method increasingList. Here, we use a numerical domain such as Polyhedra [9] or Octagon [22] for the abstract states associated with each edge in the graph. The figure shows the relevant constraints from these states. They are expressed in terms of parameter v and the edge-local identifiers (Src, val) and (Trg, val), which refer to the val field of the source and target of a pointer, respectively.

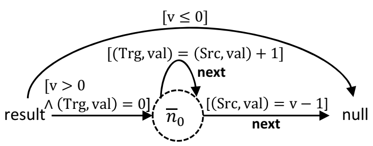


Fig. 2. The abstract heap state inferred at line 11 of Fig. 1

The abstract state reflects the five invariants stated above. Variable result is null if the constraints on the corresponding edge hold, that is, if v is non-positive (I1). Otherwise, result points to the summary node $\overline{n}_0$, which implies that it is non-null (I2). This example illustrates that our analysis represents disjunctive information in a single graph: both possible values of result are represented by the same graph, and we use value information to determine the states in which each pointer may exist. The constraint $(\mathsf{Trg}, \mathsf{val}) = 0$ on the edge from result to the summary node $\overline{n}_0$ expresses that the first list node has value 0 (I3). Note that the edge-local identifier allows us to express properties of a single object, even if it is abstracted by a summary node. The same feature is used in the constraint $(\mathsf{Trg}, \mathsf{val}) = (\mathsf{Src}, \mathsf{val}) + 1$ on the edge from $\overline{n}_0$ to itself to express invariant I4. Finally, the constraint $(\mathsf{Src}, \mathsf{val}) = \mathsf{v} - 1$ on the edge from $\overline{n}_0$ to null expresses that the last list node has value $\mathsf{v} - 1$ (I5). All five invariants are inferred automatically by our analysis without manual annotations.

Outline. Sec. 2 defines the language and the concrete domain. Sec. 3 formalizes the abstract domain, while Sec. 4 defines the abstract semantics. Sec. 5 reports the experimental results, Sec. 6 discusses related work, and Sec. 7 concludes.

2 Programming Language and Concrete Domain

We present our analysis for the small object-based language in Fig. 3. To simplify the formalization, we model local variables as fields of a special object $\Im$, that is, treat local variables as heap locations. We distinguish *reference field* and *value field* access expressions rAE and vAE, depending on the type of the accessed field. A reference expression rexp may be null, a reference field access expression, or an object creation. A value expression vexp may be a literal, a value field access expression, or a binary expression. Since the treatment of loops and conditionals is standard, the only relevant statements in ST are value and reference assignments.

$$\begin{aligned}
&\mathsf{rAE} ::= \Im.\mathsf{f_r} \mid \mathsf{rAE}.\mathsf{f_r}\\
&\mathsf{vAE} ::= \Im.\mathsf{f_v} \mid \mathsf{rAE}.\mathsf{f_v}\\
&\mathsf{rexp} ::= \mathsf{null} \mid \mathsf{rAE} \mid \mathsf{new\ C}\\
&\mathsf{vexp} ::= \mathsf{n} \mid \mathsf{vAE} \mid \mathsf{vexp}\ \langle \mathsf{op} \rangle\ \mathsf{vexp}\\
&\mathsf{op} ::= + \mid - \mid * \mid \cdots\\
&\mathsf{ST} ::= \mathsf{rAE} = \mathsf{rexp} \mid \mathsf{vAE} = \mathsf{vexp}
\end{aligned}$$

Fig. 3. Expressions and statements

In the concrete domain, we partition the content of heap locations into values and references. Let Ref be the set of concrete references (objects and null), with $\Im, \mathsf{null} \in \mathsf{Ref}$, and let Val be the set of values. Let $\mathsf{Field}_{\mathsf{Ref}}$ and $\mathsf{Field}_{\mathsf{Val}}$ be finite sets of reference and value fields, respectively. An execution state consists of a *value store* and a *reference store*. We model a value store as a partial map in $\mathsf{Store}_{\mathsf{Val}} = (\mathsf{Ref} \setminus \{\mathsf{null}\}) \times \mathsf{Field}_{\mathsf{Val}} \rightharpoonup \mathsf{Val}$ and a reference store as a partial map in $\mathsf{Store}_{\mathsf{Ref}} = (\mathsf{Ref} \setminus \{\mathsf{null}\}) \times \mathsf{Field}_{\mathsf{Ref}} \rightharpoonup \mathsf{Ref}$. For each reference in their domain, these maps contain an entry for every field. We will refer to entries in a reference store as *concrete edges*. We define the set of all concrete states (*concrete heaps*) as $\Sigma = \mathsf{Store}_{\mathsf{Ref}} \times \mathsf{Store}_{\mathsf{Val}}$.

3 Abstract Domain and Operators

In this section, we present the abstract domain, the concretization function, as well as join and widening operators.

3.1 Abstract Domain

Let $\overline{\mathsf{Ref}}$ be the set of *abstract references* (or *abstract nodes*) with $\Im, \mathsf{null} \in \overline{\mathsf{Ref}}$ (that is, we overload the symbols $\Im$ and null to denote both concrete and abstract references). Each abstract node $\overline{n} \in \overline{\mathsf{Ref}}$ represents either a single concrete non-null reference (*definite node*), or a non-empty set of concrete non-null references (*summary node*) with $\Im$ and null being definite nodes. The functions in $\mathsf{IsSummary} = \overline{\mathsf{Ref}} \rightarrow \{\mathsf{true}, \mathsf{false}\}$ define whether a node is a summary node.

An *abstract reference store* in $\overline{\mathsf{Store}_{\mathsf{Ref}}} = \mathcal{P}((\overline{\mathsf{Ref}} \setminus \{\mathsf{null}\}) \times \mathsf{Field}_{\mathsf{Ref}} \times \overline{\mathsf{Ref}})$ represents possible pointers between abstract nodes through reference fields. It can be interpreted as a directed graph where edges are labeled with a field name. Hence, we will refer to members of the abstract reference store as *abstract edges.* For an abstract edge $\overline{n}_1 \xrightarrow{f_r} \overline{n}_2$, we will refer to $\overline{n}_1$ as the *source* and to $\overline{n}_2$ as the *target* of the edge.

Our heap analysis is parameterized by an *abstract value domain* $\overline{\mathsf{V}}$, which tracks information about value fields, for instance, relations among numerical values. Each abstract edge is associated with an abstract value state (*abstract condition*) via a map in $\overline{\mathsf{Cond}} = (\overline{\mathsf{Ref}} \setminus \{\mathsf{null}\}) \times \mathsf{Field}_{\mathsf{Ref}} \times \overline{\mathsf{Ref}} \rightarrow \overline{\mathsf{V}}$. The abstract condition of an abstract edge approximates the concrete value stores in which the edge exists. That is, our abstract domain tracks disjunctive information by having several edges with the same source and field, and associating them with different abstract conditions.

Abstract value states in $\overline{\mathsf{V}}$ refer to memory locations via *abstract identifiers* $\overline{\mathsf{ID}} = \overline{\mathsf{Loc}} \cup \overline{\mathsf{EId}}$ where $\overline{\mathsf{Loc}} = (\overline{\mathsf{Ref}} \setminus \{\mathsf{null}\}) \times \mathsf{Field}_{\mathsf{Val}}$ and $\overline{\mathsf{EId}} = \{\mathsf{Src}, \mathsf{Trg}\} \times \mathsf{Field}_{\mathsf{Val}}$. An identifier $(\overline{n}, f_v) \in \overline{\mathsf{Loc}}$ represents the value field f_v of the concrete references abstracted by the node $\overline{n}$. *Edge-local identifiers* $(\mathsf{Src}, f_v), (\mathsf{Trg}, f_v) \in \overline{\mathsf{EId}}$ represent the value field f_v of the *single* concrete *source* or *target* reference of the concrete edges represented by an abstract edge. They track relations between the value fields of adjacent references in concrete heaps, which allows us to infer precise invariants on summary nodes. For instance, the constraint $(\mathsf{Src}, \mathsf{val}) \leq (\mathsf{Trg}, \mathsf{val})$ in the abstract condition of an abstract edge $\overline{n} \xrightarrow{\mathsf{next}} \overline{n}$ expresses sortedness of the concrete list that is abstracted by the summary node $\overline{n}$.

We define the set of all abstract states (*abstract heaps*) as $\overline{\Sigma} = \overline{\mathsf{Store}_{\mathsf{Ref}}} \times \overline{\mathsf{Cond}} \times \mathsf{IsSummary}$.

Example. The abstract heap in Fig. 4 depicts the loop invariant of the program in Fig. 1. Many of the constraints are similar to the constraints in Fig. 2. In particular, combining the abstract heap for the loop invariant with the negation of the loop guard (that is, $\mathsf{i} \leq 0$) yields the information reflected in Fig. 2, for instance, that result is null iff $\mathsf{v} \leq 0$ and that the first list node has value 0.

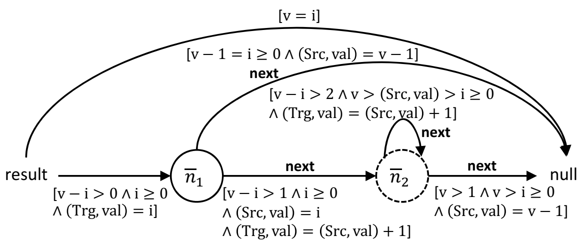


Fig. 4. The abstract heap representing the loop invariant at line 4 of the example in Fig. 1. Solid and dashed circles denote definite and summary nodes, respectively. Arrows depict abstract edges and are annotated with relevant constraints from their abstract conditions. To improve readability, we depict local reference variables as nodes and use local variables as identifiers in constraints, although the analysis models them as fields of $\Im$.

3.2 Concretization

In this section, we define the concretization function $\gamma : \overline{\Sigma} \to \mathcal{P}(\Sigma)$ that yields the set of concrete heaps represented by a given abstract heap.

We assume that our heap analysis is instantiated with a sound value analysis. Its concretization function $\gamma_{\overline{\mathsf{V}}} : \overline{\mathsf{V}} \to \mathcal{P}(\overline{\mathsf{ID}} \to \mathcal{P}(\mathsf{Val}))$ yields a set of maps from abstract identifiers to sets of concrete values. These maps yield sets of concrete values rather than single values since an abstract value state may contain identifiers for fields of summary nodes, and the value analysis alone cannot concretize them. Let *references*(st_{Ref}) be the set of concrete references of a given concrete reference store st_{Ref}, including $\Im$ and null. We define the concretization function γ of abstract heaps as:

$$(st_{\mathsf{Ref}}, st_{\mathsf{Val}}) \in \gamma(\overline{St}, \overline{con}, isSum) \Leftrightarrow \left(\begin{array}{l} \exists \alpha_{\mathsf{Ref}} \in (references(st_{\mathsf{Ref}}) \to \overline{\mathsf{Ref}}) \cdot \\ \quad GraphEmbed(\alpha_{\mathsf{Ref}}, st_{\mathsf{Ref}}, (\overline{St}, isSum)) \wedge \\ \quad ValueEmbed(\alpha_{\mathsf{Ref}}, (st_{\mathsf{Ref}}, st_{\mathsf{Val}}), \overline{con}) \end{array} \right)$$

That is, a concrete heap $(st_{\mathsf{Ref}}, st_{\mathsf{Val}})$ is in the concretization of an abstract heap $(\overline{St}, \overline{con}, isSum)$ iff there exists an embedding α_{Ref} (a function from concrete references to abstract nodes) such that the *shape* and the *values* of the concrete heap can be embedded into the abstract heap. These embeddings are expressed via the predicates *GraphEmbed* and *ValueEmbed*, which are defined as follows.

GraphEmbed holds if a given concrete reference store matches the shape of a given abstract heap, ignoring the value information. This is the case if $\Im$ and null are the only concrete references that are abstracted to the abstract $\Im$ and null (1), if, whenever multiple concrete references are abstracted to a single abstract reference, that abstract reference is a summary node (2), and if every concrete edge is represented by an abstract edge in the abstract heap (3). Note that this abstract edge is unique since α_{Ref} is a function:

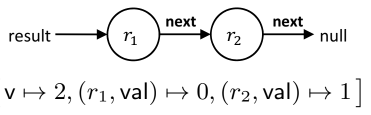
$$\begin{aligned}
&\mathit{GraphEmbed}(\alpha_{\mathsf{Ref}}, st_{\mathsf{Ref}}, (\overline{St}, \mathit{isSum})) \Leftrightarrow \\
&\quad \alpha_{\mathsf{Ref}}^{-1}(\Im) = \{\Im\} \;\wedge\; \alpha_{\mathsf{Ref}}^{-1}(\mathsf{null}) = \{\mathsf{null}\} \;\wedge && (1)\\
&\quad (\forall \overline{n} \in \mathit{img}(\alpha_{\mathsf{Ref}}) \cdot |\alpha_{\mathsf{Ref}}^{-1}(\overline{n})| > 1 \Rightarrow \mathit{isSum}(\overline{n})) \;\wedge && (2)\\
&\quad \forall r_1 \xrightarrow{f_r} r_2 \in st_{\mathsf{Ref}} \cdot \alpha_{\mathsf{Ref}}(r_1) \xrightarrow{f_r} \alpha_{\mathsf{Ref}}(r_2) \in \overline{St} && (3)
\end{aligned}$$

where $\alpha_{\mathsf{Ref}}^{-1}$ is the *preimage* of α_{Ref} (that is, it yields the set of concrete references abstracted by a given abstract reference).

ValueEmbed expresses that, for a given concrete reference store st_{Ref}, the value store st_{Val} matches all relevant abstract conditions in the abstract heap. Here, an abstract condition is *relevant* if it is associated with an abstract edge that corresponds to a concrete edge in st_{Ref}. In the definition below, we relate the concrete value store st_{Val} to each relevant abstract condition via a map s from abstract identifiers to sets of concrete values. For each concrete edge $r_1 \xrightarrow{f_r} r_2$ in the concrete reference store st_{Ref}, there is a map s in the concretization of the abstract condition of the corresponding abstract edge (4). The map s may constrain a concrete location (r, f_v) in three ways: via the abstract identifier $(\alpha_{\mathsf{Ref}}(r), f_v)$, via the edge-local identifier (Src, f_v) if r is the source of the concrete edge, that is, $r = r_1$, and via the edge-local identifier (Trg, f_v) if r is the target of the concrete edge, that is, $r = r_2$. In all three cases, the map s must yield a set that contains the value v stored in the concrete value store for (r, f_v) (5). Finally, any concrete value store matches the relevant abstract conditions only if the conditions do not contradict each other, even on abstract locations that are not included in a given concrete heap. To ensure there are no such contradictions, s must be in the concretization of *all* relevant conditions, ignoring edge-local identifiers, which may denote different locations for different abstract edges. We use the operator $\downarrow_{\overline{\mathsf{Loc}}}$ to project to the identifiers in $\overline{\mathsf{Loc}}$, that is, to remove edge-local identifiers (6).

$$\begin{aligned}
&\mathit{ValueEmbed}(\alpha_{\mathsf{Ref}}, (st_{\mathsf{Ref}}, st_{\mathsf{Val}}), \overline{con}) \Leftrightarrow \\
&\quad \forall r_1 \xrightarrow{f_r} r_2 \in st_{\mathsf{Ref}} \cdot \exists s \in \gamma_{\overline{\mathsf{V}}}(\overline{con}(\alpha_{\mathsf{Ref}}(r_1) \xrightarrow{f_r} \alpha_{\mathsf{Ref}}(r_2))) \cdot && (4)\\
&\quad\quad \forall ((r, f_v) \mapsto v) \in st_{\mathsf{Val}} \cdot \begin{pmatrix} v \in s(\alpha_{\mathsf{Ref}}(r), f_v) \wedge \\ r = r_1 \Rightarrow v \in s(\mathsf{Src}, f_v) \wedge \\ r = r_2 \Rightarrow v \in s(\mathsf{Trg}, f_v) \end{pmatrix} \wedge && (5)\\
&\quad\quad s \downarrow_{\overline{\mathsf{Loc}}} \in \gamma_{\overline{\mathsf{V}}} \left(\bigsqcap_{r_1' \xrightarrow{f_r'} r_2' \in st_{\mathsf{Ref}}} \left(\overline{con}(\alpha_{\mathsf{Ref}}(r_1') \xrightarrow{f_r'} \alpha_{\mathsf{Ref}}(r_2')) \downarrow_{\overline{\mathsf{Loc}}} \right) \right) && (6)
\end{aligned}$$

Example. Fig. 5 shows the reference and value stores of two concrete heaps. The heap of the left) is in the concretization of the abstract heap in Fig. 2. For the embedding $\alpha_{\mathsf{Ref}} = [\Im \mapsto \Im, \mathsf{null} \mapsto \mathsf{null}, r_1 \mapsto \overline{n}_0, r_2 \mapsto \overline{n}_0]$, *GraphEmbed* holds since $\overline{n}_0$ is a summary node and all three concrete edges have corresponding abstract edges. *ValueEmbed* also holds since the concrete value store satisfies


result
next
next
null
r_1
r_2
$[\mathsf{v} \mapsto 2, (r_1, \mathsf{val}) \mapsto 0, (r_2, \mathsf{val}) \mapsto 1]$

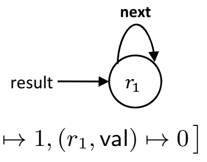


Fig. 5. Concrete heaps, consisting of a reference store, displayed on top, and a value store, displayed underneath. The heap on the left is in the concretization of the abstract heap in Fig. 2, whereas the heap on the right is not because it violates the condition that list nodes store increasing values.

the three relevant abstract conditions, and these conditions do not contradict each other.

In contrast, the heap on the right is *not* in the concretization of the abstract heap in Fig. 2. The graph embedding forces the embedding to be $\alpha_{\mathsf{Ref}} = [\Im \mapsto \Im, \mathsf{null} \mapsto \mathsf{null}, r_1 \mapsto \overline{n}_0]$. Therefore, both edge-local identifiers $(\mathsf{Src}, \mathsf{val})$ and $(\mathsf{Trg}, \mathsf{val})$ on the abstract edge from $\overline{n}_0$ to $\overline{n}_0$ correspond to (r_1, val), such that there is no value for (r_1, val) that satisfies the constraint $(\mathsf{Trg}, \mathsf{val}) = (\mathsf{Src}, \mathsf{val}) + 1$. In other words, any map s in the concretization of this constraint assigns different values to these edge-local identifiers and, thus, does not satisfy condition (5).

3.3 Join

The *join* operator $\sqcup_{\overline{\Sigma}}$ first computes an abstract reference store for the joined heaps and then the abstract conditions for the edges in this store.

Abstract Reference Store. An abstract heap can be viewed as a directed graph in which vertices are labeled as $\Im$, null, definite node other than $\Im$ and null, or summary node; edges are labeled with reference fields. The vertex labels are used to avoid matching nodes in two heaps that cannot correspond (for instance, a summary node and a definite node). A *labeled heap graph* is a triple $g = (V, E, \eta) \in \mathsf{Graph}$, where $V \subseteq \overline{\mathsf{Ref}}$ is a set of vertices, $E \subseteq V \times \mathsf{Field}_{\mathsf{Ref}} \times V$ is a set of edges labeled with a reference field, and $\eta : V \to \{\Im, \mathsf{Null}, \mathsf{Def}, \mathsf{Sum}\}$ is a labeling function on vertices. We assume a *strict total order* $<_{\mathsf{G}}$ on graphs that ensures in particular that $g_1 <_{\mathsf{G}} g_2$ if g_1 has fewer vertices than g_2 or the same number of vertices but fewer edges.

To improve performance, we define the join of two abstract reference stores such that it minimizes the size of the resulting store. Its structure is the minimum common supergraph of the two joined stores. Let g_1 and g_2 be graphs. Graph g is a *common supergraph* of g_1 and g_2 iff g_1 and g_2 are subgraph isomorphic to g with the isomorphisms $\mathcal{I}_1$ and $\mathcal{I}_2$, respectively. We call g the *minimum common supergraph* (MCS) of g_1 and g_2 if there exists no other common supergraph that is smaller in the ordering $<_{\mathsf{G}}$. The procedure $\mathit{MCS}(g_1, g_2)$ yields the (unique) minimum common supergraph g of g_1 and g_2 as well as the corresponding subgraph isomorphisms $\mathcal{I}_1$ and $\mathcal{I}_2$ between g_1 and g, and g_2 and g, respectively. The problem of computing MCS can be reduced to the well-studied problem

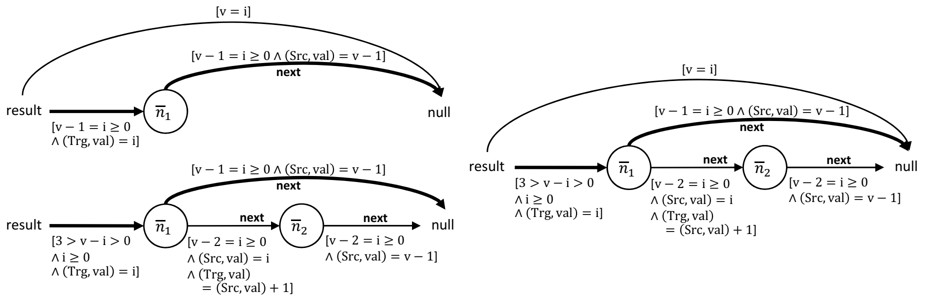

[v = i]
[v − 1 = i ≥ 0 ∧ (Src, val) = v − 1]
next
result
$\overline{n}_1$
null
[v − 1 = i ≥ 0 ∧ (Trg, val) = i]
[3 > v − i > 0 ∧ i ≥ 0 ∧ (Trg, val) = i]
$\overline{n}_2$
[v − 2 = i ≥ 0 ∧ (Src, val) = i ∧ (Trg, val) = (Src, val) + 1]
[v − 2 = i ≥ 0 ∧ (Src, val) = v − 1]

Fig. 6. The abstract heap graphs on the left occur before and after the second iteration of the fixed-point computation of the loop invariant (line 4) in Fig. 1. Joining them results in the heap on the right. Bold arrows indicate edges of the maximum common subgraph.

of finding the maximum common subgraph [4]. See the accompanying technical report [15] for the definitions of graph/subgraph isomorphism and maximum common subgraph. Intuitively, we can compute $(g, \mathcal{I}_1, \mathcal{I}_2) = MCS(g_1, g_2)$ by "gluing" to the maximum common subgraph of g_1 and g_2 those parts of g_1 and g_2 that are not in their maximum common subgraph.

Let $\Pi : \overline{\mathsf{Store}_{\mathsf{Ref}}} \times \mathsf{IsSummary} \to \mathsf{Graph}$ be a *bijective* function from abstract stores to heap graphs. The abstract store and the IsSummary function of the join of $(\overline{St}_1, \overline{con}_1, isSum_1)$ and $(\overline{St}_2, \overline{con}_2, isSum_2)$ are $(\overline{St}, isSum) = \Pi^{-1}(MCS(\Pi(\overline{St}_1, isSum_1), \Pi(\overline{St}_2, isSum_2)) \downarrow_1)$, where $\downarrow_1$ denotes projection of a tuple on the first component, that is, the graph returned by MCS. Note that $\Pi(\overline{St}, isSum)$ includes both $\Pi(\overline{St}_1, isSum_1)$ and $\Pi(\overline{St}_2, isSum_2)$. Hence, the abstract reference store $(\overline{St}, isSum)$ subsumes the abstract reference stores $(\overline{St}_1, isSum_1)$ and $(\overline{St}_2, isSum_2)$.

Example. Fig. 6 shows on the left the abstract heap graphs g_1 and g_2 before and after the second iteration of the fixed-point computation of the loop invariant (line 4) in Fig. 1, and their join g on the right. Besides the special nodes $\Im$ and null, the maximum common subgraph includes node $\overline{n}_1$ as well as the edges from result to $\overline{n}_1$ and from $\overline{n}_1$ to null. To this common subgraph, we add the remainder of g_1 (the edge from result to null) and the remainder of g_2 ($\overline{n}_2$ with its edges). Note that both g_1 and g_2 are subgraph isomorphic to g, where the isomorphism is the identity function.

Abstract Conditions. Consider an edge in the abstract store resulting from the join of two abstract heaps. We determine its abstract condition as follows. If the edge is in the maximum common subgraph of the joined heap graphs, its abstract condition is the join of the abstract conditions in the two heaps. Otherwise, the condition is the same as in the heap that contributed the edge, after applying the subgraph isomorphism.

As explained above, computing the minimum common supergraph $(g, \mathcal{I}_1, \mathcal{I}_2) = MCS(\Pi(\overline{St}_1, isSum_1), \Pi(\overline{St}_2, isSum_2))$ yields the subgraph isomorphisms $\mathcal{I}_1$ and $\mathcal{I}_2$ from $\Pi(\overline{St}_1, isSum_1)$ to g and from $\Pi(\overline{St}_2, isSum_2)$ to g, respectively.

We define the function $\mathit{rename}_{\mathsf{ISO}} : (\overline{\mathsf{Ref}} \to \overline{\mathsf{Ref}}) \times \overline{\mathsf{V}} \to \overline{\mathsf{V}}$ to rename the identifiers in $\overline{\mathsf{Loc}}$ of a given abstract value state according to an isomorphism. Using this renaming, we define the join operator $\sqcup_{\overline{\Sigma}} : \overline{\Sigma} \times \overline{\Sigma} \to \overline{\Sigma}$ as

$$(\overline{St}_1, \overline{con}_1, \mathit{isSum}_1) \sqcup_{\overline{\Sigma}} (\overline{St}_2, \overline{con}_2, \mathit{isSum}_2) = (\overline{St}, \overline{con}, \mathit{isSum})$$

where:

$$\begin{aligned}
&(g, \mathcal{I}_1, \mathcal{I}_2) = \mathit{MCS}(\Pi(\overline{St}_1, \mathit{isSum}_1), \Pi(\overline{St}_2, \mathit{isSum}_2)) \land \\
&(\overline{St}, \mathit{isSum}) = \Pi^{-1}(g) \land \\
&\overline{con} = \left[\overline{e} \mapsto \overline{s} \;\middle|\; \overline{e} \in \overline{St} \;\land\; \overline{s} = \bigsqcup \left\{ \overline{s}' \;\middle|\; \begin{array}{l} \exists\, i \in \{1,2\} \cdot \exists\, (\overline{n}_1, f_r, \overline{n}_2) \in \overline{St}_i \,\cdot \\ \overline{e} = (\mathcal{I}_i(\overline{n}_1), f_r, \mathcal{I}_i(\overline{n}_2)) \land \\ \overline{s}' = \mathit{rename}_{\mathsf{ISO}}(\mathcal{I}_i, \overline{con}_i(\overline{n}_1, f_r, \overline{n}_2)) \end{array} \right\} \right]
\end{aligned}$$

Computing the maximum common subgraph is NP-complete; however, most code fragments change only small portions of the abstract heap. Our implementation exploits this fact to compute the isomorphisms incrementally, usually in linear time.

Example. Consider the edge from result to $\overline{n}_1$ in the heap on the right of Fig. 6, which is in the maximum common subgraph of the heaps on the left. Hence, its abstract condition is the join of the conditions for those heaps (assuming a relational numerical domain). Since the constraint $\mathsf{v}-1 = \mathsf{i}$ in the top left abstract heap implies the constraint $3 > \mathsf{v} - \mathsf{i} > 0$ in the lower heap, the latter constraint is tracked by the result of the join operation; the other constraints of the joined conditions are identical and, thus, carried over to the result. Conversely, the edges from result to null, from $\overline{n}_1$ to $\overline{n}_2$, and from $\overline{n}_2$ to null are not in the maximum common subgraph; their conditions come from the heap contributing the edges.

3.4 Widening

The above join operator does not guarantee the convergence of the analysis. In fact, the size of the abstract heap may grow at each application of join, and the abstract conditions may not stabilize. Therefore, we define a *widening* operator $\nabla_{\overline{\Sigma}} : \overline{\Sigma} \times \overline{\Sigma} \to \overline{\Sigma}$ that guarantees that the analysis reaches a fixed point in finite time (that is, terminates). In order to do so, the widening operator must bound the size of the abstract heap, which means that it has to merge nodes into summary nodes. This merging is controlled via a finite set of field access expressions $\mathcal{W}$, which is a parameter of the analysis and denotes references that the analysis should track separately. By default, $\mathcal{W}$ is the set of local reference variables, but it can be extended to any set of field access expressions if desired. For all examples in our evaluation (Sec. 5), the analysis uses the default.

We perform widening in two steps. First, in both input heaps, we merge nodes that (i) are denoted by the same set of field access expressions from $\mathcal{W}$, *and*

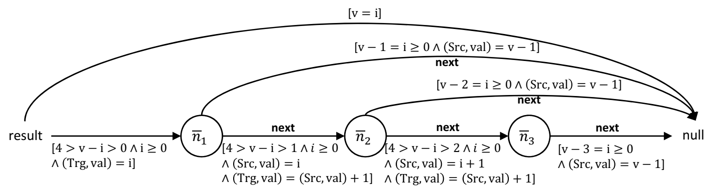

[v = i]
[v − 1 = i ≥ 0 ∧ (Src, val) = v − 1]
next
[v − 2 = i ≥ 0 ∧ (Src, val) = v − 1]
next
result
[4 > v − i > 0 ∧ i ≥ 0 ∧ (Trg, val) = i]
n̄1
next
[4 > v − i > 1 ∧ i ≥ 0 ∧ (Src, val) = i ∧ (Trg, val) = (Src, val) + 1]
n̄2
next
[4 > v − i > 2 ∧ i ≥ 0 ∧ (Src, val) = i + 1 ∧ (Trg, val) = (Src, val) + 1]
n̄3
next
[v − 3 = i ≥ 0 ∧ (Src, val) = v − 1]
null

Fig. 7. Heap before the fourth iteration of the fixed-point computation of the loop invariant in Fig. 1.

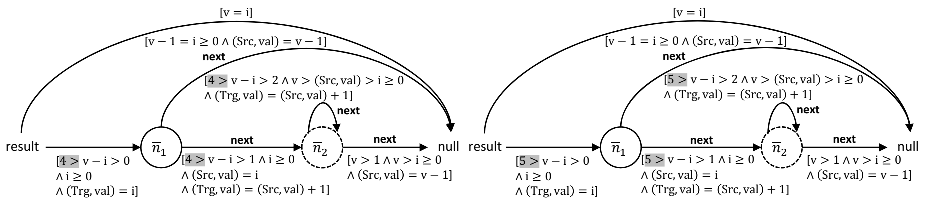

[v = i]
[v − 1 = i ≥ 0 ∧ (Src, val) = v − 1]
next
[4 > v − i > 2 ∧ v > (Src, val) > i ≥ 0 ∧ (Trg, val) = (Src, val) + 1]
next
result
[4 > v − i > 0 ∧ i ≥ 0 ∧ (Trg, val) = i]
n̄1
next
[4 > v − i > 1 ∧ i ≥ 0 ∧ (Src, val) = i ∧ (Trg, val) = (Src, val) + 1]
n̄2
next
[v > 1 ∧ v > i ≥ 0 ∧ (Src, val) = v − 1]
null
[v = i]
[v − 1 = i ≥ 0 ∧ (Src, val) = v − 1]
next
[5 > v − i > 2 ∧ v > (Src, val) > i ≥ 0 ∧ (Trg, val) = (Src, val) + 1]
next
result
[5 > v − i > 0 ∧ i ≥ 0 ∧ (Trg, val) = i]
n̄1
next
[5 > v − i > 1 ∧ i ≥ 0 ∧ (Src, val) = i ∧ (Trg, val) = (Src, val) + 1]
n̄2
next
[v > 1 ∧ v > i ≥ 0 ∧ (Src, val) = v − 1]
null

Fig. 8. Heaps with merged nodes before the fourth (left) and fifth (right) iteration of the fixed-point computation. The heaps differ only in the highlighted constraints.

(ii) are reachable (via some access path) from the same set of local variables. Second, if the two heaps are isomorphic, we apply edge-wise widening to the abstract conditions; otherwise, we join them. We refer the reader to our technical report [15] for more details and a complete formalization.

Example. Suppose we widen the abstract heap before the fourth iteration of the fixed-point computation for the loop in Fig. 1 with the heap before the fifth iteration. The abstract heap before the fourth iteration is displayed in Fig. 7; the heap before the fifth iteration looks similar, but has four definite nodes.

In the first step, widening merges nodes using the default $\mathcal{W} = \{\mathsf{result}\}$. In the heap from Fig. 7, we merge $\overline{n}_2$ and $\overline{n}_3$ into a single summary node $\overline{n}_2$ since they are (i) denoted by the same set of field access expressions from $\mathcal{W}$ (the empty set since result denotes neither $\overline{n}_2$ nor $\overline{n}_3$), and (ii) they are reachable from the same set of local variables ($\{\mathsf{result}\}$). However, $\overline{n}_1$ is denoted by a different set of field access expressions ($\{\mathsf{result}\}$), and therefore not merged with $\overline{n}_2$ and $\overline{n}_3$. The edges from $\overline{n}_2$ and $\overline{n}_3$ to null are also merged, and their conditions are joined. The resulting heap is shown on the left of Fig. 8. Merging nodes in the heap before the fifth iteration (not shown) results in the heap on the right of Fig. 8. Note that these heaps are isomorphic, that is, the heap shape has stabilized.

In the second step, since the heaps after merging are isomorphic, we apply edge-wise widening to the abstract conditions. This step removes the upper bound on $\mathsf{v} - \mathsf{i}$, but leaves all other constraints unaffected, that is, the abstract conditions have stabilized. The resulting heap is shown in Fig. 4; it represents the loop invariant of the program from Fig. 1.

4 Abstract Semantics

In this section, we formalize the semantics of reference and value assignments.

4.1 Reference Assignment

An abstract store includes disjunctive information. Therefore, for a reference assignment $\mathsf{p.f_r} = \mathsf{rhs}$, there may be several abstract references for the receiver p and the right-hand side rhs, which may be reached through different paths with different value conditions. The value states on the edges along each path specify the conditions under which p and rhs evaluate to a particular abstract reference. The abstract semantics for reference assignments adds an abstract edge for each possible combination of receiver p and right-hand side rhs, with an abstract condition that reflects when this combination exists.

The rule below formalizes reference assignments of the form $\mathsf{p.f_r} = \mathsf{rhs}$, where $\mathsf{p.f_r} \in \mathsf{rAE}$ and $\mathsf{rhs} \in \mathsf{rexp}$. Since we encode local variables as fields of a special reference $\Im$, the rule also covers assignments to those. It uses an auxiliary function $\overline{eval}_{\mathsf{rexp}}$ (defined in the technical report [15]), which takes a reference expression (or $\Im$) re and an abstract state $\overline{\sigma}$ and yields (a) a set NC of pairs, each consisting of an abstract reference to which re may evaluate in $\overline{\sigma}$ and the condition under which re may evaluate to this reference, and (b) a resulting abstract state, which is used to encode allocation, that is, when re contains new expressions.

$$\frac{\begin{array}{rr}
(NC_{\mathsf{rhs}}, (\overline{St}_{\mathsf{rhs}}, \overline{con}_{\mathsf{rhs}}, isSum_{\mathsf{rhs}})) = \overline{eval}_{\mathsf{rexp}}(\mathsf{rhs}, \overline{\sigma}) & (1) \\
(NC_{\mathsf{p}}, \_) = \overline{eval}_{\mathsf{rexp}}(\mathsf{p}, (\overline{St}_{\mathsf{rhs}}, \overline{con}_{\mathsf{rhs}}, isSum_{\mathsf{rhs}})) & (2) \\
strong \iff \exists\, \overline{n} \in \overline{\mathsf{Ref}} \cdot (NC_{\mathsf{p}} = \{(\overline{n}, \_)\} \ \wedge\ \neg isSum_r(\overline{n})) & (3) \\
strong \Rightarrow (\overline{St} = \{(\overline{n}_1, f, \overline{n}_2) \in \overline{St}_{\mathsf{rhs}} \mid (\overline{n}_1, \_) \notin NC_{\mathsf{p}} \ \vee\ \mathsf{f_r} \neq f\}) & (4) \\
(\neg strong) \Rightarrow (\overline{St} = \overline{St}_{\mathsf{rhs}}) & (5) \\
\overline{con}_{asg} = \left[(\overline{n}_{\mathsf{p}}, \mathsf{f_r}, \overline{n}_{\mathsf{rhs}}) \mapsto (\mathsf{Trg}\,To\mathsf{Src}(\overline{s}_{\mathsf{p}}) \sqcap \overline{s}_{\mathsf{rhs}}) \,\middle|\, \begin{array}{l} (\overline{n}_{\mathsf{p}}, \overline{s}_{\mathsf{p}}) \in NC_{\mathsf{p}} \wedge \\ (\overline{n}_{\mathsf{rhs}}, \overline{s}_{\mathsf{rhs}}) \in NC_{\mathsf{rhs}} \end{array} \right] & (6) \\
\overline{St}' = \overline{St} \cup dom(\overline{con}_{asg}) & (7) \\
\overline{con}' = \overline{con}_{\mathsf{rhs}} \left[\overline{e} \mapsto \overline{s} \,\middle|\, \begin{array}{l} \overline{e} \in dom(\overline{con}_{asg}) \ \wedge\ (\overline{e} \notin \overline{St} \Rightarrow \overline{s} = \overline{con}_{asg}(\overline{e})) \ \wedge \\ (\overline{e} \in \overline{St} \Rightarrow \overline{s} = \overline{con}_{asg}(\overline{e}) \sqcup \overline{con}_{\mathsf{rhs}}(\overline{e})) \end{array} \right] & (8)
\end{array}}{\langle \mathsf{p.f_r} = \mathsf{rhs}, \overline{\sigma} \rangle \rightarrow_{\overline{\Sigma}} (\overline{St}', \overline{con}', isSum_{\mathsf{rhs}})}$$

A reference assignment first evaluates rhs to obtain the possible abstract references for the right-hand side expression together with the corresponding conditions, as well as a successor state (1). The receiver p is evaluated in this successor state. Since it is side-effect free (see Fig. 3), we discard the state resulting from its evaluation (2). The analysis performs a strong update iff there is only one abstract reference $\overline{n}$ for the receiver, which is a definite node (3). In that case, the analysis removes all edges whose source is the receiver node and that are labeled with the assigned field $\mathsf{f_r}$ (4); otherwise, it performs a weak update, that is, retains all existing edges (5). To add the edges for all possible combinations

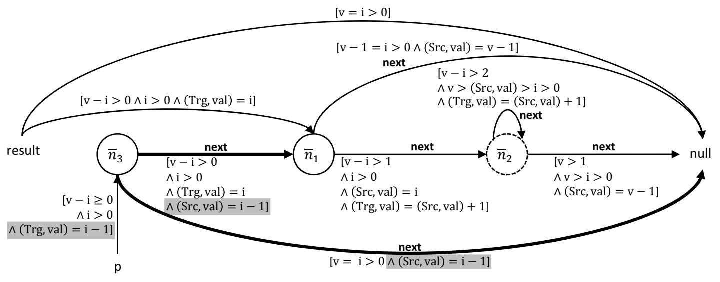


Fig. 9. The abstract heap after line 7 of the program in Fig. 1. The bold edges are added by the reference assignment in line 6; the highlighted constraints come from the value assignment in line 7.

of receivers and right-hand sides, we first create a map $\overline{con}_{asg}$ that maps each of the new edges to the abstract condition that describes when the particular combination exists, that is, the greatest lower bound of the conditions for choosing a particular abstract reference for the receiver and a particular abstract reference for the right-hand side, respectively (6). The only twist in this step is how to handle edge-local identifiers. The receiver is denoted by Trg in conditions on edges pointing to the receiver, but by Src in the new edges. Function Trg*To*Src performs this conversion. Since the map $\overline{con}_{asg}$ contains an entry for each new edge, we obtain the final abstract store by adding the domain of this map to the store constructed in step 4 or 5 (7). Finally, the abstract conditions are updated: For each new edge that is not present in the store before the reference assignment, we add the condition from $\overline{con}_{asg}$. For each edge that is already present (which may happen during a weak update), we join the condition from $\overline{con}_{asg}$ and the existing condition (8).

Example. Fig. 9 *without* the bold edges and the highlighted constraints, shows the abstract heap after line 5 in Fig. 1. It is obtained from the abstract heap in Fig. 4 (the loop invariant) by (i) assuming the loop guard ($\mathsf{i} > 0$) in all abstract conditions and (ii) applying the abstract semantics of the statement Node p = new Node() (line 5), which introduces the definite node $\overline{n}_3$. (Its next field initially points to null, which is not shown in the figure.) We will now illustrate the abstract semantics of the reference assignment p.next = result (line 6).

The right-hand side of the assignment, result, evaluates to null or to $\overline{n}_1$ (point (1) of of the rule above). Since the receiver p evaluates to a single definite node $\overline{n}_3$ (2), we perform a strong update (3). The strong update removes all out-edges of $\overline{n}_3$ labeled with next (4) and introduces new edges for all combinations of abstract references for the receiver and for the right-hand side, that is, edges from $\overline{n}_3$ to null and from $\overline{n}_3$ to $\overline{n}_1$ (7). These edges are shown in bold in Fig. 9. The former edge exists if the right-hand side (result) evaluates to null, that is, if $\mathsf{v} = \mathsf{i} > 0$; the latter exists if result evaluates to $\overline{n}_1$, that is, if $\mathsf{v} - \mathsf{i} > 0 \wedge \mathsf{i} > 0 \wedge (\mathsf{Trg}, \mathsf{val}) = \mathsf{i}$ (6). These constraints are the abstract conditions of the new edges (8), as shown in Fig. 9 (still ignoring the highlighted constraints, which will be discussed later).

4.2 Value Assignment

Like the semantics of reference assignments, the abstract semantics of a value assignment $\mathsf{p.f_v} = \mathsf{rhs}$ (where $\mathsf{p.f_v} \in \mathsf{vAE}$ and $\mathsf{rhs} \in \mathsf{vexp}$) needs to consider each possible combination of evaluations for the receiver p and for the right-hand side rhs. For each combination, it updates all abstract conditions in the abstract store to reflect the assignment and the conditions under which the combination exists. The following rule formalizes this intuition.

$$\frac{\begin{array}{rl} \overline{S} = \overline{eval}_{\mathsf{vexp}}(\mathsf{rhs}, (\overline{St}, \overline{con}, isSum)) & (1) \\ (NC_{\mathsf{p}}, \_) = \overline{eval}_{\mathsf{rexp}}(\mathsf{p}, (\overline{St}, \overline{con}, isSum)) & (2) \\ \overline{con}' = update_{\overline{\mathsf{Cond}}}(\mathsf{rhs}, \mathsf{f_v}, NC_{\mathsf{p}}, \overline{S}, \overline{con}) & (3) \end{array}}{\langle \mathsf{p.f_v} = \mathsf{rhs}, (\overline{St}, \overline{con}, isSum)\rangle \rightarrow_{\overline{\Sigma}} (\overline{St}, \overline{con}', isSum)}$$

Each way of evaluating the right-hand side expression rhs, if it contains a field access, chooses a path through the abstract store. The abstract conditions of the edges along a path describe when this path may be chosen. Function $\overline{eval}_{\mathsf{vexp}}$ (defined in our technical report [15]) yields the set $\overline{S}$ of conditions (value states) that describe each way of evaluating rhs (1). Analogously to step 2 of reference assignment, we evaluate the receiver expression p to obtain the possible receiver references, each with a condition under which p may evaluate to this reference (2). We use the function $update_{\overline{\mathsf{Cond}}}$ (defined in our technical report [15]) to reflect the value assignment in the value states of all edges in the abstract store (3). This function considers all possible combinations of receiver reference (obtained from NC_{p}) and value state for a particular way of evaluating the right-hand side expression (from $\overline{S}$). For each of them, it propagates the value information that has to hold when this combination is chosen to the conditions of each edge in the abstract store and applies the assignment operation of the value domain. The condition of each edge in the abstract store is then defined to be the join of the conditions obtained for all ways of executing the value assignment.

Example. Fig. 9 *without* the highlighted constraints shows the abstract heap after line 6 in Fig. 1. The highlighted constraints are introduced by the abstract semantics of the statement `p.val = i - 1` (line 7). There is only one way to evaluate the right-hand side expression. Therefore, $\overline{eval}_{\mathsf{vexp}}$ yields a singleton set (point (1) of the rule above). This set contains the condition $\mathsf{v} - \mathsf{i} \geq 0 \land \mathsf{i} > 0$, which holds in each concrete heap (otherwise there would be no value for result). Similarly, the receiver expression p evaluates to a single node, $\overline{n}_3$, under the same condition (2). This condition must be satisfied in order to be able to perform the assignment. Therefore, we conjoin it to each abstract condition in the store (which has no effect in this example), and then assign $\mathsf{i} - 1$ to $(\overline{n}_3, \mathsf{val})$ since p evaluates to $\overline{n}_3$ (3). Moreover, since $\overline{n}_3$ is the target of the edge from p to $\overline{n}_3$, we also add the constraint $(\mathsf{Trg}, \mathsf{val}) = \mathsf{i} - 1$ for the edge-local identifier to the condition on this edge, and analogously for $(\mathsf{Src}, \mathsf{val})$ on both out-edges of $\overline{n}_3$ (3).

Table 1. Analysis times (in seconds) of classes implementing different data structures when instantiating the analysis with the Octagon and Polyhedra value domains. For each class, we inferred an object invariant by computing a fixed point over all its methods.

Data Structure	Operations	Octa.	Poly.
SortedSLL	constructor insertKey deleteKey findKey deepCopy	1.24	1.82
SortedDLL	constructor insertKey deleteKey findKey deepCopy	1.91	2.83
BST	constructor insertKey findKey	1.96	2.43
NodeCachingSLL	constructor add remove findKey	0.87	1.04
PersonAndAccount	withdraw deposit changeInterest	0.38	0.43

5 Experimental Results

We implemented our analysis in the static analyzer Sample and applied it to Scala implementations of typical list and tree operations (some of which we took from the literature [5,7,14]), operations on nested recursive data structures (such as lists of lists), and a simple aggregate structure [12]. We performed the experiments on an Intel Core i7-Q820 CPU (1.73GHz, 8GB) running the 64-bit version of Ubuntu 14.04. We instantiated our analysis with the Octagon [22] and Polyhedra [9] value domains implemented in Apron [17]. We used the default widening parameter, that is, $\mathcal{W}$ is the set of local reference variables. There were no manual annotations for any of the benchmarks.

Inference of Object Invariants. Tab. 1 reports the analysis times (the average of 10 runs) for implementations of five different data structures. We instantiated Logozzo's framework [19] with our analysis to infer object invariants for each data structure by computing a fixed point over all its operations.

SortedSLL implements a sorted singly-linked list (SLL). The inferred object invariant expresses that the values stored in the list nodes are non-decreasing.

SortedDLL implements a sorted doubly-linked list (DLL). Our analysis infers sortedness in both directions, that is, via the next and prev fields. However, the analysis cannot infer the structural invariant of doubly-linked lists $n.\mathsf{next} \neq \mathsf{null} \Rightarrow n = n.\mathsf{next}.\mathsf{prev}$ because it has no way of relating the concrete references of the two edges $\overline{n} \xrightarrow{\mathsf{next}} \overline{n}$ and $\overline{n} \xrightarrow{\mathsf{prev}} \overline{n}$ for the summary node $\overline{n}$.

BST implements a binary search tree. The analysis infers both the value and the shape information of a BST data structure. Our implementation stores the infimum and supremum of all keys of a sub-tree in its root. This information allows our analysis to relate the value stored in the root to the values in the left and right sub-trees, and, thus, to infer that the shape is a tree, that is, loop-free and not a general DAG. We omitted method deleteKey because our analysis is not able to infer that replacing the deleted key with the next smallest key preserves sortedness; it does, however, infer that the tree shape is preserved.

Table 2. Analysis times (in seconds) for single operations on different data structures when instantiating the analysis with the Octagon and Polyhedra value domains. The first 4 operations work on singly-linked lists (SLL) and doubly-linked lists (DLL). The fifth operation works on lists of singly-linked lists that store values. The last operation transforms an SLL to a binary search tree.

Operation	Octa.	Poly.
insertionSort	0.43 - SLL 0.72 - DLL	0.48 - SLL 0.85 - DLL
partitionWithKey	0.32 - SLL 0.48 - DLL	0.34 - SLL 0.55 - DLL
createListOfZerosAndSum	0.22 - SLL 0.39 - DLL	0.23 - SLL 0.43 - DLL
increasingList	0.28 - SLL 0.41 - DLL	0.31 - SLL 0.50 - DLL
sortListOfListsOfValues	1.45	1.88
listToBST	1.03	1.21

NodeCachingSLL implements an acyclic SLL that maintains a cache of node objects to reduce object creation and garbage collection. The inferred object invariant expresses that the list and the cache are disjoint and that the size of the cache is between 0 and maximumCacheSize. Moreover, we inferred that the addKey method creates a new object only if the cache is empty. Every node of the list stores the length of the list rooted at the node. This information lets our analysis infer that the list and its cache are acyclic, which is needed to infer disjointness of the list and the cache. The latter step required materialization, that is, splitting a definite node off a summary node, which is supported by our implementation, but not explained in this paper.

Besides the object invariants for these four classes, our analysis infers that the result of method findKey is either null or has the value of the given key. This postcondition is inferred even if the result is represented by a summary node.

PersonAndAccount implements an aggregate data structure similar to the example from a paper on the verification of object invariants [12]. The analysis infers combined shape and value invariants, for instance, that Account and Person objects reference each other, the sum of the account balance and person's salary is positive, and the interest rate of the account is always non-negative.

Inference of Method Postconditions. Tab. 2 reports the analysis times of individual operations on different data structures. The initial abstract states and the abstract heaps that represent the arguments to the operations contain only information that is provided by the static types; no annotations were used. The first four operations manipulate singly and doubly-linked lists. insertionSort takes an unsorted list of values and sorts it. The analysis infers that the result is a sorted list. partitionWithKey takes a list of values and a key, and creates two new lists such that the keys in one are less than or equal to the given key, and the keys in the other are greater. The analysis infers this value property and that the resulting lists are disjoint. createListOfZerosAndSum creates a list of zeros and subsequently traverses the list and sums up the values. The analysis infers that the result is a list of zeros, and the sum of the values is zero. increasingList is the method from Fig. 1, with an analogous implementation for DLLs. The analysis infers the heap in Fig. 2 (and an analogous heap for DLLs).

The last two operations of Tab. 2 demonstrate that the analysis is able to infer non-trivial shape and value properties for programs manipulating nested recursive data structures or a combination of different data structures. sortListOfListsOfValues takes a singly-linked list of SLLs that store values, and sorts each of the lists. The analysis infers that the result is a list of sorted SLLs. listToBST takes an SLL of values and creates a binary search tree out of it, without using the methods of the BST class discussed above. The analysis infers that the result is a binary search tree.

Discussion. The analysis times in Tab. 1 and Tab. 2 demonstrate the efficiency of our analysis. For all our benchmark classes, the fixed point over all their methods was computed within 3 seconds when using the Polyhedra domain. When instantiated with a more efficient but less precise value domain, the efficiency of the analysis increases, as illustrated by the usage of the Octagon domain.

Our experiments demonstrate that our analysis can infer invariants that combine shape and value information in interesting ways, for instance, sortedness of lists and trees, or invariants that relate the states of different objects in an aggregate structure. Our analysis leverages data stored in value fields, such as the infimum and supremum in the BST class discussed above, to obtain more precise shape information. As future work, we plan to rely less on such fields by tracking additional abstract conditions (such as injectivity of references) on edges and by generalizing edge-local identifiers to reference fields.

6 Related Work

Dillig et al. [10,11] present a precise content analysis for arrays and containers, in which heap edges are qualified by logical constraints over indexes into a container. This idea inspired our approach of tracking disjunctive information via the value states associated with edges in the heap. Our analysis uses generic value domains instead of logical constraints and can therefore be instantiated with different levels of precision and efficiency. Moreover, it uses edge-local identifiers instead of indexes, which allows us to express constraints on arbitrary nodes (especially summary nodes) in the heap, not only on indexed structures such as arrays and containers. Whereas Dillig et al. concentrate on clients of arrays and containers, our analysis targets arbitrary heap-manipulating programs including implementations of containers.

Similarly to our work, Bouajjani et al. [2,3] introduce a static analysis that automatically infers combined shape and numerical invariants and is parametric in the underlying value domain. The main difference is that their technique is specific to programs that manipulate singly-linked lists of values. For such data structures Bouajjani et al.'s approach is more powerful since it can relate an arbitrary number of successive positions in a list. In contrast, the aim of our analysis is to be applicable to general heap-manipulating programs.

Sagiv et al. [23] introduce a shape analysis in which invariants are expressed in 3-valued first order logic with transitive closure (FOLTC). These invariants may combine shape and value constraints. The analysis requires user-supplied predicates, whereas our analysis does not need manual annotations; it represents a state by a set of logical structures, whereas our analysis maintains a single abstract heap, reducing the number of nodes and edges, and therefore the complexity of the overall analysis. The merging of nodes in our widening operator can be viewed as a special case of canonical abstractions.

McCloskey et al. [21] propose a framework for combining shape and numerical domains (encoded as predicates in FOLTC) in a generic way. However, users have to supply shared predicates via which the domains communicate and which usually resemble the properties one wants to prove. In contrast, our analysis can be parameterized by arbitrary value domains without any manual overhead.

Ferrara et al. [13,14] and Fu [16] combine different heap and value analyses. Whereas their work represents a state as a heap abstraction and a single value state, our analysis attaches a value state to each edge in the heap abstraction, allowing for a precise tracking of disjunctive information. Moreover, in the value states of Ferrara et al.'s and Fu's work, different heap identifiers represent disjoint portions of the heap. This is not the case for our edge-local identifiers, which refer to memory locations already represented by abstract identifiers and which enable a precise treatment of summary nodes.

Chang et al. [7] introduce a shape analysis based on user-supplied invariants that describe data structures such as lists and trees. These invariants are used to abstract over a potentially unbounded number of concrete references. Chang and Rival [5,6] extend this work and present a framework for combining shape and numeric abstractions into a single domain. Their approach enables the precise and modular analysis of heap and numeric invariants, but relies on user-supplied properties, whereas our analysis does not require manual annotations.

Abdulla et al. [1] introduce a fully automatic analysis of dynamically-allocated heap data structures. They abstract heaps as forest automata, extended by constraints on the values stored in heap nodes. While the analysis precisely tracks shape information, the value constraints can represent only a fixed set of ordering relations. For instance, they cannot express invariant I4 of our running example (see introduction). Moreover, our analysis can be parametrized with different value domains, allowing for different trade-offs between precision and efficiency.

Marron et al. [20] introduce heap abstractions that are similar to the graphs representing abstract heaps in our work. In fact, the formalization of the concretization function in Sec. 3.2 is inspired by their work. However, there are important technical differences. In particular, Marron et al.'s analysis maintains a normal form, which makes their lattice finite, but loses information when merging two heap graphs. In contrast, we deal with an infinite lattice, but preserve some of this information. Moreover, Marron et al.'s heap graphs track specific aliasing predicates (such as injectivity of fields or tree shapes), but no value information. Finally, the purpose of their work is to provide a high-level abstraction of *concrete* runtime heaps, whereas we propose an abstract domain and an abstract semantics for a static code analysis.

7 Conclusion

In this paper, we have presented a static analysis that infers complex invariants combining shape and value information. The analysis is parametric in the underlying value domain, allowing for different trade-offs between precision and efficiency. A key innovation of our analysis is the introduction of edge-local identifiers to track value information about the source and target of a pointer, which allows it to infer inductive invariants such as sortedness of a linked list. The analysis has been implemented in the static analyzer Sample. Our experiments demonstrate its effectiveness.

As future work, we plan to generalize the abstract conditions associated with abstract edges to track richer information. Supporting reference equalities and inequalities would allow our analysis to infer more structural invariants such as the invariant of a doubly-linked list. Supporting regular expressions over field names as additional abstract identifiers would allow the analysis to infer global properties.

Acknowledgments. We are grateful to Uri Juhasz and Alexander Summers for numerous discussions, to John Boyland for helpful comments on a draft of this paper, and to Severin Heiniger for his contributions to the implementation. We would like to thank the anonymous reviewers for the useful feedback.

References

1. Abdulla, P.A., Holík, L., Jonsson, B., Lengál, O., Trinh, C.Q., Vojnar, T.: Verification of heap manipulating programs with ordered data by extended forest automata. In: Van Hung, D., Ogawa, M. (eds.) ATVA 2013. LNCS, vol. 8172, pp. 224–239. Springer, Heidelberg (2013)
2. Bouajjani, A., Drăgoi, C., Enea, C., Sighireanu, M.: On inter-procedural analysis of programs with lists and data. In: PLDI. ACM (2011)
3. Bouajjani, A., Drăgoi, C., Enea, C., Sighireanu, M.: Abstract domains for automated reasoning about list-manipulating programs with infinite data. In: Kuncak, V., Rybalchenko, A. (eds.) VMCAI 2012. LNCS, vol. 7148, pp. 1–22. Springer, Heidelberg (2012)
4. Bunke, H., Jiang, X., Kandel, A.: On the minimum common supergraph of two graphs. Computing 65(1), 13–25 (2000)
5. Chang, B.-Y.E., Rival, X.: Relational inductive shape analysis. In: POPL. ACM (2008)
6. Chang, B.-Y.E., Rival, X.: Modular construction of shape-numeric analyzers. In: David A. Schmidt's 60th Birthday Festschrift. EPTCS (2013)
7. Chang, B.-Y.E., Rival, X., Necula, G.C.: Shape analysis with structural invariant checkers. In: Riis Nielson, H., Filé, G. (eds.) SAS 2007. LNCS, vol. 4634, pp. 384–401. Springer, Heidelberg (2007)

8. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL. ACM (1977)
9. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: POPL. ACM (1978)
10. Dillig, I., Dillig, T., Aiken, A.: Fluid updates: Beyond strong vs. Weak updates. In: Gordon, A.D. (ed.) ESOP 2010. LNCS, vol. 6012, pp. 246–266. Springer, Heidelberg (2010)
11. Dillig, I., Dillig, T., Aiken, A.: Precise reasoning for programs using containers. In: POPL. ACM (2011)
12. Drossopoulou, S., Francalanza, A., Müller, P., Summers, A.J.: A unified framework for verification techniques for object invariants. In: Vitek, J. (ed.) ECOOP 2008. LNCS, vol. 5142, pp. 412–437. Springer, Heidelberg (2008)
13. Ferrara, P.: Generic combination of heap and value analyses in abstract interpretation. In: McMillan, K.L., Rival, X. (eds.) VMCAI 2014. LNCS, vol. 8318, pp. 302–321. Springer, Heidelberg (2014)
14. Ferrara, P., Fuchs, R., Juhasz, U.: TVAL+: TVLA and value analyses together. In: Eleftherakis, G., Hinchey, M., Holcombe, M. (eds.) SEFM 2012. LNCS, vol. 7504, pp. 63–77. Springer, Heidelberg (2012)
15. Ferrara, P., Müller, P., Novacek, M.: Automatic inference of heap properties exploiting value domains. Technical Report 794, ETH Zurich (2013)
16. Fu, Z.: Modularly combining numeric abstract domains with points-to analysis, and a scalable static numeric analyzer for Java. In: McMillan, K.L., Rival, X. (eds.) VMCAI 2014. LNCS, vol. 8318, pp. 282–301. Springer, Heidelberg (2014)
17. Jeannet, B., Miné, A.: Apron: A library of numerical abstract domains for static analysis. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 661–667. Springer, Heidelberg (2009)
18. Lev-Ami, T., Sagiv, M.: TVLA: A system for implementing static analyses. In: Palsberg, J. (ed.) SAS 2000. LNCS, vol. 1824, pp. 280–302. Springer, Heidelberg (2000)
19. Logozzo, F.: Automatic inference of class invariants. In: Steffen, B., Levi, G. (eds.) VMCAI 2004. LNCS, vol. 2937, pp. 211–222. Springer, Heidelberg (2004)
20. Marron, M., Sánchez, C., Su, Z., Fähndrich, M.: Abstracting runtime heaps for program understanding. IEEE Trans. Software Eng. 39(6), 774–786 (2013)
21. McCloskey, B., Reps, T., Sagiv, M.: Statically inferring complex heap, array, and numeric invariants. In: Cousot, R., Martel, M. (eds.) SAS 2010. LNCS, vol. 6337, pp. 71–99. Springer, Heidelberg (2010)
22. Miné, A.: The octagon abstract domain. Higher Order Symbol. Comput. (2006)
23. Sagiv, M., Reps, T., Wilhelm, R.: Parametric shape analysis via 3–valued logic. In: POPL. ACM (1999)