TVAL+ : TVLA and Value Analyses Together

Pietro Ferrara, Raphael Fuchs, and Uri Juhasz

ETH Zürich, Switzerland
{pietro.ferrara,uri.juhasz}@inf.ethz.ch, fuchsra@ethz.ch

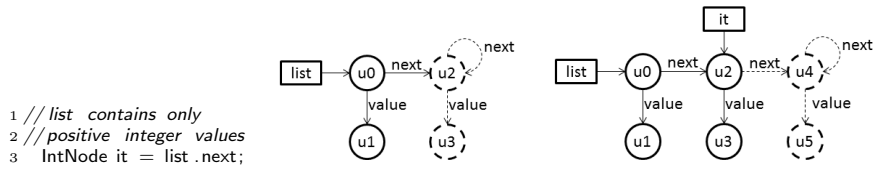
Abstract. Effective static analyses must precisely approximate both heap structure and information about values. During the last decade, shape analysis has obtained great achievements in the field of heap abstraction. Similarly, numerical and other value abstractions have made tremendous progress, and they are effectively applied to the analysis of industrial software. In addition, several generic static analyzers have been introduced. These compositional analyzers combine many types of abstraction into the same analysis to prove various properties. The main contribution of this paper is the combination of Sample, an existing generic analyzer, with a TVLA-based heap abstraction (TVAL+).

1 Introduction

During the last decades, heap analysis has been extensively, deeply and successfully studied. Its goal is to approximate all possible heap shapes in a finite way. This is particularly important when analyzing object-oriented programs, which heavily interact with dynamically allocated memory. Static analysis has been widely applied to the abstraction of numerical information as well. Numerical domains [8, 22] track static information at different levels of approximation. In addition, other approaches (e.g., string analyses) approximate other types of information over the values computed during the execution.

Usually, the combination of the heap abstraction with information about other values (called *value* domain) is necessary. For instance, consider a program that sum the values contained in the nodes of a list. Here we would like to prove that, at the end of the execution, the computed value of is the summation of all elements in the given list. For this reason, several recent approaches have combined heap and value abstractions. In this context, some heap analyses (e.g., TVLA) were extended with information about numerical values [21], or ad-hoc heap analyses were combined with some existing numerical domains [3].

Thanks to compositional analyses based on abstract interpretation [5], we can define a generic analyzer that combines various abstractions modularly and automatically. In this way, the implementation of different domains can be composed together without reimplementing the analysis. In addition, generic static analyzers take care of all aspects not strictly related to the abstract domain, e.g., the computation of a fixpoint. As far as we know, existing generic analyzers [9, 18, 24] apply a fixed heap analysis, while they let the user specify the value abstraction and the property of interest. Sample (Static Analyzer of Multiple



(a) Running example (b) At the beginning and at the end (c) At line 4

Fig. 1. The running example and the states obtained by TVLA when analyzing it

Programming LanguagEs) is a novel generic analyzer of object-oriented programs that is parametric not only in the value domain and in the property of interest, but in the heap abstraction as well.

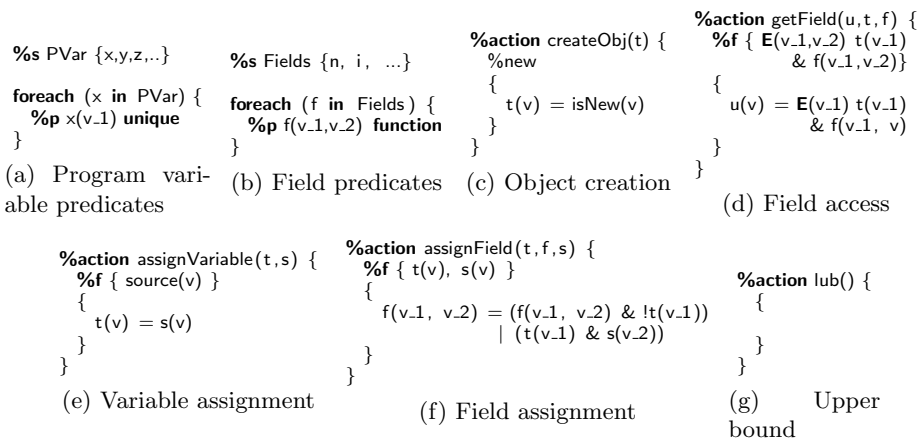
Contribution Given this context, the contribution of this work is the extension of Sample with a TVLA-based heap analysis (TVAL+). In particular, we formalize the structure of Sample, how we name nodes in TVLA states through name predicates, and how we communicate the modifications performed by TVLA on the heap structure to the value analysis. In this way, TVAL+ can be combined with any existing value abstraction in Sample. The combination between TVAL+ and value analyses is completely automatic.

Intuitevely, Sample computes an abstract state for each program point. We use TVLA as the engine to define the heap small-step semantics of our language, while the value analysis tracks information over the values contained in abstract heap nodes represented by heap identifiers. Since TVLA names nodes in a completely unpredictable and arbitrary way, and it does not provide any information from where nodes come from after the application of a TVLA action, we augment standard TVLA states with name predicates, and we normalize the exit states obtained by applying TVLA actions to keep the naming schema consistent. When we perform this normalization, we communicate the changes performed by TVLA on the heap structure to the value analysis. Our approach supports this normalization without requiring any additional feature to the value analysis, since it relies on standard semantic operators (namely, assignment and forgetting of identifiers). The TVLA state can contain any instrumentation predicates.

1.1 Running Example

Consider now the code in Figure 1a. This program assigns -1 to the value contained in the second node of a given list. Let us suppose that the list is acyclic, it contains at least two nodes, and that the initial TVLA state is the one depicted in Figure 1b. The node pointed by list.next is materialized when it is assigned to the iterator, while it.value is materialized when it is assigned at line 4. Therefore, after the analysis of line 4, TVLA infers the heap state depicted in Figure 1c[1]. Finally, when we assign null to it, TVLA removes this unary predicate

[1] TVLA would actually give more options, which we omit here.



(a) Program variable predicates (b) Field predicates (c) Object creation (d) Field access

(e) Variable assignment (f) Field assignment (g) Upper bound

Fig. 2. TVLA actions of the heap semantics

from the TVLA state, and this leads to summarize u2 with u4, and u3 with u5. This brings the analysis to the initial state depicted in Figure 1b.

2 Background

2.1 Sample

Sample (Static Analyzer of Multiple Programming LanguagEs) is a novel generic analyzer of object-oriented programs based on the abstract interpretation theory [6, 7]. Relying on compositional analyses, Sample can be instantiated with various heap abstractions and value domains. A state of the analysis is a pair composed by a state of the heap domain H and a state of the value domain V (formally, $\Sigma = \mathsf{H} \times \mathsf{V}$). Sample has been already applied to various value analyses [4, 10, 11, 25], and it supports some of the most common numerical analyses through Apron [16]. In addition, some rough heap analyses have been already developed in Sample. The analyzer works on an intermediate object-oriented language called Simple, and it supports the compilation of Scala and Java bytecode to this language. Simple is based on control flow graphs (cfg). Each block of the cfg contains a list of statements that may be $\mathsf{x} := \mathsf{y.f}$, $\mathsf{y.f} := \mathsf{x}$, or $\mathsf{x} := \mathsf{new\ T}$.

2.2 Shape Analysis

TVLA [17] is a framework for defining and implementing heap abstractions in 3-valued first order logic [23] with transitive closure (FOLTC). In this section, we sketch the standard TVLA features we adopt in TVAL+. For each analysis a FOL signature (predicates of arity up to 2) is defined. The predicates are divided into core (uninterpreted) predicates, and instrumentation predicates, which are defined by a FOLTC formula over the core predicates. The abstract domain

is composed of sets of structures of 3-valued FOLTC. A 3-valued structure is composed of normal and *summary* nodes. Normal nodes represent exactly one concrete node, while summary nodes may represent many concrete nodes.

The abstraction is defined by a set of unary predicates out of the signature, which can be core or instrumentation predicates (the *abstraction predicates AP*). In a normalized structure any two distinct nodes are differentiated by at least one abstraction predicate. For heaps, usually unary predicates represent local reference valued variables and binary predicates represent reference valued fields. The graphical representation, as in Figure 1b, uses circles for nodes (dashed for summary nodes), labeled originless arrows for local variables (again dashed for may point to), and labeled arrows between nodes to represent reference valued fields. For example, for a linked list as in Figure 1b, we could use as abstraction predicates with one free variable pointed-to-by-list (u0 in the example) $list(x)$, so that the first node is not summarized with the rest of the nodes.

In our example, we would need predicates to differentiate u1, u3 and u5. We can achieve that by using (i) value-of-node-pointed-to-by-list (u1), that is, $\exists y : list(y) \wedge value(y, x)$, (ii) value-of-node-pointed-to-by-it (u3 in 1c), that is, $\exists y : it(y) \wedge value(y, x)$, and (iii) $\exists y, z : it(y) \wedge next * (y, z) \wedge value(z, x)$ for differentiating the nodes coming before and after it.

Concrete transformers are represented by update formulae ($P'(\overline{x}) = \phi(\overline{x})$). Here P is a predicate symbol, ϕ a formula (evaluated in the pre-state), $\overline{x}$ are bound variables (implicitly universally quantified - exactly as many as the arity of P) and P' is P in the post state. For example, Figure 2f represents the assignment t.f = s. Here the new value of f (the field being written) is given by a formula on the old values of f, t and s (we omit here null checks).

TVLA works by applying a *semantic reduction* (called *focus*) before the abstract transformer. Given an abstract state, focus produces a set of abstract states with the same concrete representation, but ensuring some pairs of nodes are not merged, in addition to the separation enforced by the abstraction predicates. In the linked list example, before advancing to the next node, we would like to make sure it is not merged with any other node, so we would focus on it using the formula $\exists y : it(y) \wedge next(y, x)$. TVLA uses *widening* (called *blur*) to ensure termination. After applying the abstract transformer on the focused structures, nodes which are not separable by the abstraction predicates are merged, and the same happens for structures, ensuring a bound on the size of the abstract domain.

TVLA actions

Figure 2 reports all TVLA actions that are used in TVAL+. For every program variable x, a unary predicate $P_x(v)$ specifies the node that is pointed by the local variable (Figure 2a). Unique specifies that at most one node satisfies the predicate. Heap nodes are connected to each other when a field of an object references another object. For every possible field f, we introduce a binary predicate $P_f(v_1, v_2)$ that connects heap nodes (Figure 2b). For example, if field n of node a references node b, we have that $P_n(a, b) = 1$. In the definition, function means that the field relation is a (partial) function. Object creation relies on the TVLA

built-in predicate *isNew*. It creates a new (non-summary) node and assigns it to a temporary program variable temp (Figure 2c). When we access a field of an object, the node modelling the target object may have been summarized with other nodes. However, we would like to obtain a concrete (i.e., not summarized) node as the result of our access. With this purpose we add the focus formula $\exists(v_1, v_2) : P_{target}(v_1) \wedge P_f(v_1, v_2)$. As in the case of object creation, the result is assigned to a temporary program variable (Figure 2d). In the case of assignments, we assume there is always a unary predicate pointing to the source of the assignment. In the case of a variable, it is the program variable predicate, while in the case of a heap access or an object creation it is the variable temp which was created as explained above. Therefore, we simply copy the valuation of the unary predicate (Figure 2e). The treatment of the assignment to a field is similar to normal assignment. However, the translation to TVLA is different, as it involves a field predicate, and we need to access the target object whose field is assigned (Figure 2f). When we join two states (e.g., when computing the exit state of an if statement), we rely on the join performed automatically by TVLA on all input structures in the entry state. Therefore, we simply provide TVLA with the two states and an empty action, and we take the exit state as the result of the upper bound operator (Figure 2g).

3 Heap and Value Analyses in Sample

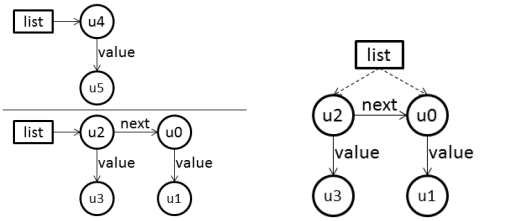
The state of the computation of an object oriented program can be defined as the combination of the heap structure with the values that can be contained in heap locations or local variables. Let Ref be the set of concrete references, and FieldName the set of field names. The heap may be defined by $\mathsf{Ref} \times \mathsf{FieldName} \rightarrow \mathsf{Ref}$ for heap locations, and by $\mathsf{VarId} \rightarrow \mathsf{Ref}$ (where VarId is the set of local variables) for the local variables. Let Val be the set of values (e.g., integers or strings). The runtime values can be represented by $\mathsf{Ref} \times \mathsf{FieldName} \rightarrow \mathsf{Val}$ for heap locations, and by $\mathsf{VarId} \rightarrow \mathsf{Val}$ for local variables.

When we reason about the abstraction of concrete states, often we would like to reason about heap structures and values separately. Therefore, we suppose that concrete references are abstracted by abstract heap identifiers (HId). Each heap analysis defines its own finite set of heap identifiers. A heap identifier could represent one or many concrete references. Let $\gamma_{\mathsf{HId}} : \mathsf{HId} \rightarrow \wp(\mathsf{Ref})$ be the concretization of abstract heap identifiers. We say that a heap identifier i represents a summary node if $|\gamma_{\mathsf{HId}}(\mathsf{i})| > 1$.

Since the heap analysis needs to abstract together many concrete heaps, a heap interaction (e.g., a field access) may provide many heap identifiers. Sometimes the heap analysis may not be able to establish one exact node for a heap access, and therefore it would return a *possible* set of heap identifiers. In other cases, the heap analysis could track disjunctive information through a set of heap states at a given program point, and therefore it would return a *definite* set of heap identifiers. Formally, we define set of heap identifiers $\mathsf{SHId} = \wp(\mathsf{HId}) \times \{\mathsf{true}, \mathsf{false}\}$. The boolean flag is true if the set is definite (if it represents *all* identifiers in

```
1  IntNode list = new IntNode();
2  if (...) {
3    IntNode newNode = new IntNode();
4    newNode.next = list;
5     list = newNode;
6  }
7  list .value = 1;
```

(a) Non-deterministic code



(b) Precise abstraction (c) Rough abstraction

Fig. 3. A non-deterministing program possible abstractions of the heap at line 7

the set), false if it is possible (if it represents *some* of them). Note that trace partitioning [20] is supported in Sample[12]. Therefore, if the heap domain uses a disjunction of heaps rather than a single 3-valued logical structure (as often happens in TVLA), we can use this feature to prove complex properties.

Example: Consider the code in Figure 3a. This program non-deterministically adds a node at the beginning of a list containing only one element. It then assigns 1 to field value of the node at the head of the list. Heap analyses could produce different states at line 7. A precise analysis like TVLA may track two distinct states (Figure 3b), while a rough analysis may abstract the two states into one (Figure 3c). What happens when we assign to list.value? In the first scenario, we have to perform a strong assignment to both the nodes pointed by list.value. This is represented by assigning to the definite set of heap identifiers $(\{\mathsf{u5}, \mathsf{u3}\}, \mathsf{true})$. In the second scenario, we can only perform a weak assignment, since we do not have two distinct states. This is represented by the possible set $(\{\mathsf{u1}, \mathsf{u3}\}, \mathsf{false})$.

3.1 Replacements

The application of semantic or lattice operators could affect the structure and the identifiers contained in the heap state. In particular, nodes could be materialized or merged. We must reflect these changes in the state of the value domain to preserve the soundness of the analysis. This information is passed by functions in $\mathsf{R} = \wp(\mathsf{HId}) \to \wp(\mathsf{HId})$ called *replacements*. Given a single relation in a replacement, its semantics is to assign the upper bound of the values of identifiers on the left side to all identifiers in the right side.

Running example: Consider the transition from Figure 1b to Figure 1c. The node u2 of the initial state is split into nodes u2 and u4 in the final state. This is represented by the relation $\{\mathsf{u2}\} \mapsto \{\mathsf{u2}, \mathsf{u4}\}$. The same happens on u3 when it is split to nodes u3 and u5. Therefore we obtain the replacement $[\{\mathsf{u2}\} \mapsto \{\mathsf{u2}, \mathsf{u4}\}, \{\mathsf{u3}\} \mapsto \{\mathsf{u3}, \mathsf{u5}\}]$. Consider now the transition obtained analyzing $\mathsf{it} = \mathsf{null}$, that is, the transition from Figure 1c to Figure 1b. u2 and u4 are summarized into u2. The same happens for u3 and u5 that are summarized to u3, obtaining the replacement $[\{\mathsf{u2}, \mathsf{u4}\} \mapsto \{\mathsf{u2}\}, \{\mathsf{u3}, \mathsf{u5}\} \mapsto \{\mathsf{u3}\}]$. □

$$assignVar_{\mathsf{V}} : (\mathsf{VarId} \times \mathsf{SHId} \times \mathsf{V}) \to \mathsf{V}$$
$$assignVar_{\mathsf{V}}(\mathsf{x}, (\mathsf{I}, \mathsf{b}), \mathsf{s}) = \begin{cases} \mathsf{s} & \text{if } \mathsf{I} = \emptyset \\ \bigsqcup_{\mathsf{i} \in \mathsf{I}} assignId_{\mathsf{V}}(\mathsf{x}, \mathsf{i}, \mathsf{s}) & \text{if } \mathsf{b} = \mathsf{false} \\ \bigsqcap_{\mathsf{i} \in \mathsf{I}} assignId_{\mathsf{V}}(\mathsf{x}, \mathsf{i}, \mathsf{s}) & \text{if } \mathsf{b} = \mathsf{true} \end{cases}$$

$$assignHIds_{\mathsf{V}} : (\mathsf{SHId} \times \mathsf{VarId} \times \mathsf{V}) \to \mathsf{V}$$
$$assignHIds_{\mathsf{V}}((\mathsf{I}, \mathsf{b}), \mathsf{x}, \mathsf{s}) = \begin{cases} \mathsf{s} & \text{if } \mathsf{I} = \emptyset \\ \bigsqcup_{\mathsf{i} \in \mathsf{I}} assignId_{\mathsf{V}}(\mathsf{i}, \mathsf{x}, \mathsf{s}) & \text{if } \mathsf{b} = \mathsf{false} \\ \bigsqcap_{\mathsf{i} \in \mathsf{I}} assignId_{\mathsf{V}}(\mathsf{i}, \mathsf{x}, \mathsf{s}) & \text{if } \mathsf{b} = \mathsf{true} \end{cases}$$

$$replace_{\mathsf{V}} : (\mathsf{V} \times \mathsf{R}) \to \mathsf{V}$$
$$replace_{\mathsf{V}}(\mathsf{s}, \emptyset) = \mathsf{s}$$
$$replace_{\mathsf{V}}(\mathsf{s}, r) = forgetAll_{\mathsf{V}}((\bigcup_{\mathsf{l} \in dom(\mathsf{r})} \mathsf{l}) \setminus (\bigcup_{\mathsf{l} \in dom(\mathsf{r})} \mathsf{r}(\mathsf{l})), \mathsf{s}_n)$$
$$\text{where } dom(r) = \{\mathsf{l}_1, \cdots, \mathsf{l}_n\} \wedge \mathsf{s}_0 = \mathsf{s} \wedge \forall i \in [1..n] :$$
$$\mathsf{s}'_i = assignVar_{\mathsf{V}}(\mathsf{temp}, (\mathsf{l}_i, \mathsf{false}), \mathsf{s}_{i-1}) \wedge$$
$$\mathsf{s}''_i = assignHIds_{\mathsf{V}}((\mathsf{r}(\mathsf{l}_i), \mathsf{true}), \mathsf{temp}, \mathsf{s}'_i) \wedge$$
$$\mathsf{s}_i = forget_{\mathsf{V}}(\mathsf{temp}, \mathsf{s}''_i)$$
and $forgetAll : (\wp(\mathsf{Id}) \times \mathsf{V}) \to \mathsf{V}$ is defined as follows:
$$forgetAll_{\mathsf{V}}(\{\mathsf{i}_1, \cdots, \mathsf{i}_n\}, \mathsf{s}) = \mathsf{s}_n \text{ where } \mathsf{s}_0 = \mathsf{s} \wedge \forall k \in [1..n] :$$
$$\mathsf{s}_k = forget_{\mathsf{V}}(\mathsf{i}_k, \mathsf{s}_{k-1})$$

Fig. 4. Definition of $replace_{\mathsf{V}}$

3.2 Heap Analysis

The semantic operators of the heap analysis are (i) $getFieldId_{\mathsf{H}} : (\mathsf{VarId} \times \mathsf{FieldName} \times \mathsf{H}) \to (\mathsf{SHId} \times \mathsf{H} \times \mathsf{R})$, (ii) $assignVar_{\mathsf{H}} : (\mathsf{VarId} \times \mathsf{SHId} \times \mathsf{H}) \to (\mathsf{H} \times \mathsf{R})$, (iii) $assignField_{\mathsf{H}} : (\mathsf{VarId} \times \mathsf{FieldName} \times \mathsf{VarId} \times \mathsf{H}) \to (\mathsf{H} \times \mathsf{R})$, and (iv) $createObject_{\mathsf{H}} : (\mathsf{C} \times \mathsf{H}) \to (\mathsf{SHId} \times \mathsf{H} \times \mathsf{R})$. C is the set of classes that can be instantiated. All these operators return a state of the heap, and a replacement to represent merges and materializations of heap nodes. In addition, $getFieldId_{\mathsf{H}}(\mathsf{x}, \mathsf{f}, \mathsf{h})$ returns a set of heap identifiers that could be pointed to by $\mathsf{x.f}$ in h. $assignVar_{\mathsf{H}}(\mathsf{x}, \mathsf{I}, \mathsf{h})$ assigns I to x, while $assignField_{\mathsf{H}}(\mathsf{x}, \mathsf{f}, \mathsf{I}, \mathsf{h})$ assigns I to $\mathsf{x.f}$. Finally, $createObject_{\mathsf{H}}(\mathsf{C}, \mathsf{h})$ creates an instance of class C returning the heap identifiers pointing to the fresh object as well.

3.3 Value Analysis

The value analysis treats variable and heap identifiers in the same way. Therefore we define identifiers (Id) as variable (VarId) or heap (HId) identifiers ($\mathsf{Id} = \mathsf{VarId} \cup \mathsf{HId}$). The semantic operators the value analysis has to provide are (i) $assignId_{\mathsf{V}} : (\mathsf{Id} \times \mathsf{Id} \times \mathsf{V}) \to \mathsf{V}$, and (ii) $forget_{\mathsf{V}} : (\mathsf{Id} \times \mathsf{V}) \to \mathsf{V}$. $assignId_{\mathsf{V}}(\mathsf{x}, \mathsf{y}, \mathsf{v})$ assigns the value of y to x in state v, while $forget_{\mathsf{V}}(\mathsf{x}, \mathsf{v})$ removes the value of x from state v. Usually these operators are already supported by existing value analyses. The only additional feature the value analysis has to take into account is when a single heap identifier represents a summary node performing weak updates.

Relying on these semantic operators, Figure 4 defines how replacements and assignments are computed by Sample in the value analysis. These operators will be used in the definition of the semantics of our language. $assignVar_{\mathsf{V}}(\mathsf{x}, \mathsf{I}, \mathsf{s})$ assigns the set of heap identifiers contained in I to variable x, while $assignHIds_{\mathsf{V}}(\mathsf{I}, \mathsf{x}, \mathsf{s})$

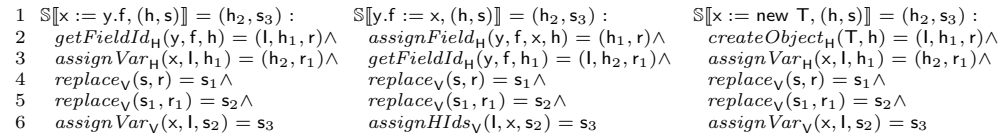


Fig. 5. Sample's semantics of statements

assigns variable x to the set of heap identifiers inside l. Both these functions behave in accordance with whether the set of heap identifiers is definite or possible. If we assign a definite set of heap identifiers to a variable, we assign the *greatest* lower bound of the values of all given heap identifiers (since we have to assign the intersection of their values). On the other hand, if we assign a possible set, we assign *one* of the values, and therefore we have to take the upper bound. Similarly, when we assign a variable to a possible set of heap identifiers, we are affecting only one of the heap identifiers in the set, and therefore we have to take the upper bound. Instead, when we are assigning to a definite set, we take the greatest lower bound of the assignments of all the heap identifiers.

$replace_{\mathsf{V}}(\mathsf{s}, \mathsf{r})$ applies the replacement r to s. For each relation $[\mathsf{D} \mapsto \mathsf{C}] \in \mathsf{r}$ it assigns the upper bound of the values of identifiers in D to each identifier in C. In its definition we denote by $\mathsf{temp} \in \mathsf{VarId}$ a variable identifier that does not appear in the program. This variable is used as a gateway to build up the abstract value represented by the variables in D, and to assign it to all identifiers in C. At the end we remove all identifiers that appear at least once on the left part of the replacement, and never on the right side. Intuitively, these identifiers are replaced by something else, and they are never used as target of other replaced variables. Therefore, they are not anymore used, and they can be safely removed.

Running example: We suppose that all nodes of the given list contain values greater or equal to zero at the beginning of the program in Figure 1a. Assuming we analyze the program using intervals (that is, tracking the interval of numerical values that each variable could have at a given program point), in the heap state of Figure 1b the value domain tracks that $[\mathsf{u1} \mapsto [0..\infty], \mathsf{u3} \mapsto [0..\infty]]$. After the first statement, the replacement we have to apply contains the relation $\{\mathsf{u3}\} \mapsto \{\mathsf{u3}, \mathsf{u5}\}$. Therefore, the application of this replacement results in the state $[\mathsf{u1} \mapsto [0..\infty], \mathsf{u3} \mapsto [0..\infty], \mathsf{u5} \mapsto [0..\infty]]$. The semantics of $\mathsf{it.value} = -1$ assigns $[-1..-1]$ to u3 obtaining the state $[\mathsf{u1} \mapsto [0..\infty], \mathsf{u3} \mapsto [-1..-1], \mathsf{u5} \mapsto [0..\infty]]$. When we finally apply the second replacement during the evaluation of $\mathsf{it} = \mathsf{null}$, the relation $\{\mathsf{u3}, \mathsf{u5}\} \mapsto \{\mathsf{u3}\}$ tells the analysis to (i) assign the upper bound of u3 and u5 (that is, $[-1..\infty]$) to u3, and (ii) remove u5. Therefore, the final state is $[\mathsf{u1} \mapsto [0..\infty], \mathsf{u3} \mapsto [-1..\infty]]$. □

3.4 Overall Semantics

Figure 5 defines the semantics of the language introduced in Section 2.1. When we assign y.f to x, we extract the identifiers pointed by y.f (line 2) and we assign

them to x in the state of the heap analysis (line 3). These two actions lead to two distinct replacements, which are passed to the value domain (line 4 and 5) before assigning the identifiers of y.f to x (line 6). Similarly, when we assign x to y.f, we perform the assignment on the heap state (line 2), and we query the heap analysis to obtain the identifiers pointed by y.f (line 3). The two actions produce two replacements that are passed to the value domain (line 4 and 5). At the end, x is assigned to the heap identifiers pointed by y.f in the value domain (line 6). When we assign new T to x, we create the object (line 2) and we assign it to x in the heap analysis (line 3) obtaining two replacements. After the application of these two replacements in the value domain (line 4 and 5), the heap identifiers of the created object is assigned to x in the value domain (line 6).

4 TVAL+

We need that each node is represented exactly by one heap identifier in a TVLA state, and each heap identifier points exactly to one node. Since TVLA names nodes in a unpredictable way, and the same node could have several canonical (that is, the evaluation of abstract predicates) names, we need to add some predicates in order to track node identity. In addition, TVLA does not provide any information about from where nodes come after an action, while these predicates track that. Note that so far we did not use the names given by TVLA in the running example, but we adopted a more predictable naming schema.

4.1 Name Predicates

We name nodes through unary non-abstraction predicates (called *name predicates*). Each time we run TVLA, the entry state contains a name predicate for each node. After running TVLA, name predicates tell us how nodes have been split or merged. We then *normalizes* the exit state to ensure that each node is pointed to exactly by one name predicate, and each name predicate points exactly to one node.

Usually unary predicates are used to distinguish between different structures when a join of heap states is performed. Since we only wish to track nodes, we do not want the name predicates to influence the abstraction. We achieve this behavior through non-abstraction predicates, since these allow nodes to be merged even though different non-abstraction predicates hold for them [17].

The naming schema defines how we name nodes. A naïve approach is to consecutively number all created heap nodes. The numbers are based on the pre-state, and not counted globally, since otherwise we could go on creating new names endlessly. In addition, we always need to obtain the same result when the same operation is performed on the same pre-state. If we used a global counter, this property would not be guaranteed. However, we would lose a lot of precision in the analysis with this approach. Consider for instance an if statement that allocates an object in both branches, but it assigns -1 to its value field in a branch, and 1 in the other. Suppose that the internal counter of the state of the

analysis is 0. The analysis would name the nodes created by the new statements in the two branches with the same name (that is, 1 for the created object, and then 2 for its field value). When the abstract states in the two branches are joined to compute the abstract state after the if statements, the values associated with the heap identifier 2 in the value domain are joined as well. This means that, regardless of the fact that the two nodes could be kept disjoint by the heap analysis, we merged the values of the two nodes in the value domain, introducing a sensible loss of precision. In the example above, we would not be able to distinguish that value 1 may have been assigned only to x.value, and -1 only to y.value after the if statement.

4.2 TVAL+ Naming Schema

This example shows that we need a more sophisticated naming schema. In particular, we have to take into account the context in which heap identifiers are created. Therefore, the name of a new node is based on the allocation site. In addition, since a given program point *pp* could create several nodes (e.g., inside a while loop), we have to count the number of times we are allocating (in the abstract) the node, and increment the counter at each iteration. Let PP be the set of program points. A *basic* heap identifier is a pair composed of a program point and a natural number (formally, $\mathsf{BHId}_{\mathsf{TVAL+}} = \mathsf{PP} \times \mathbb{N}$).

What happens when two nodes are merged into a summary node? Since TVLA could summarize nodes created at different program points, we have to extend the definitions above to precisely approximate this scenario. Therefore, a heap identifier is composed by a set of basic heap identifiers. When TVLA summarizes two nodes created at different program points (e.g., $(\mathsf{pp1}, n_1)$ and $(\mathsf{pp2}, n_2)$), the resulting name will be a set composed by both $(\{(\mathsf{pp1}, n_1), (\mathsf{pp2}, n_2)\})$.

Instead, when a node is materialized from a summary node, the output state of TVLA will contain two nodes pointed by the same name predicate, and we have to provide two different names when normalizing this state. To keep the precision of the analysis in this scenario, we add another counter to the whole heap identifier. Formally, the set of heap identifiers is defined by $\mathsf{HId}_{\mathsf{TVAL+}} = \wp(\mathsf{BHId}_{\mathsf{TVAL+}}) \times \mathbb{N}$.

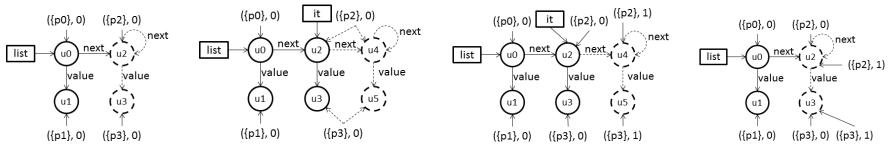
Normalization By normalized state we mean a state in which (i) each node is pointed only by one name predicate, and (ii) each name predicate points only to one node. After we run TVLA on a normalized state, we may obtain a state in which a name predicate points to many nodes, and a node is pointed by many name predicates. Figure 6 formalizes this normalization. We focus the formal definitions on the part of the TVLA state that deal with name predicates. Let $\mathsf{Nodes}_{\mathsf{TVAL+}}$ be the set of node identifiers given by TVLA. We define the TVAL+ state by a function that relates each node to the set of name predicates that point to it. Formally, $\Sigma_{\mathsf{TVAL+}} = \mathsf{Nodes}_{\mathsf{TVAL+}} \to \wp(\mathsf{HId}_{\mathsf{TVAL+}})$. Function π_{Id} returns the set of all heap identifiers contained in a given state, while *rev* returns a function relating each heap identifier to the set of nodes it may point to.

First of all, we define what it means to merge a set of name predicates when they point to the same node. *mergeHIds* takes the set union of all program points

$$
\begin{array}{l}
\pi_{\mathsf{Id}} : \Sigma_{\mathsf{TVAL}+} \to \wp(\mathsf{HId}_{\mathsf{TVAL}+}) \\
\pi_{\mathsf{Id}}(\mathsf{f}) = \bigcup_{\mathsf{n} \in dom(\mathsf{f})} \mathsf{f}(\mathsf{n}) \\
\\
rev : \Sigma_{\mathsf{TVAL}+} \to (\mathsf{HId}_{\mathsf{TVAL}+} \to \wp(\mathsf{Nodes}_{\mathsf{TVAL}+})) \\
rev(\mathsf{f}) = [\mathsf{id} \mapsto \mathsf{P} : \mathsf{id} \in \pi_{\mathsf{Id}}(\mathsf{f}) \wedge \mathsf{P} = \{\mathsf{p}' : \mathsf{id} \in \mathsf{f}(\mathsf{p}')\} \\
\\
mergeHIds : \wp(\mathsf{HId}_{\mathsf{TVAL}+}) \to \mathsf{HId}_{\mathsf{TVAL}+} \\
mergeHIds(\{(\mathsf{E}_i, u_i) : i \in [0..n]\}) = (\mathsf{E}', \min(\{u_i : i \in [0..n]\})) : \qquad (a) \\
\quad \mathsf{E}' = \{(\mathsf{pp}, c) : \exists (\mathsf{pp}, c') \in \bigcup_{i \in [0..n]} \mathsf{E}_i \wedge c = \min(\{c'' : (\mathsf{pp}, c'') \in \bigcup_{i \in [0..n]} \mathsf{E}_i\} \qquad (b) \\
\\
merge : \Sigma_{\mathsf{TVAL}+} \to (\Sigma_{\mathsf{TVAL}+} \times \mathsf{R}) \\
merge(\mathsf{f}) = (\mathsf{f}', \mathsf{r}) : \\
\mathsf{f}' = \left[\mathsf{n} \mapsto \begin{cases} \{mergeHIds(\mathsf{f}(\mathsf{n}))\} & \text{if } |\mathsf{f}(\mathsf{n})| > 1 \\ \mathsf{f}(\mathsf{n}) & \text{if } |\mathsf{f}(\mathsf{n})| == 1 \end{cases} : \mathsf{n} \in dom(\mathsf{f})\right] \wedge \\
\mathsf{r} = [\mathsf{f}(\mathsf{n}) \mapsto \{mergeHIds(\mathsf{f}(\mathsf{n}))\} : \exists \mathsf{n} \in dom(\mathsf{f}) : |\mathsf{f}(\mathsf{n})| > 1] \qquad (c) \\
\\
split : \Sigma_{\mathsf{TVAL}+} \to (\Sigma_{\mathsf{TVAL}+} \times \mathsf{R}) \\
split(\mathsf{f}) = (\mathsf{f}', \mathsf{r}) : \\
\mathsf{f}' = \left[\begin{array}{l} \mathsf{n} \mapsto \mathsf{P} : \mathsf{n} \in dom(\mathsf{f}) \wedge \\ \mathsf{P} = \begin{cases} \{(\mathsf{T}, |\{\mathsf{k} \in exclude((\mathsf{T}, \mathsf{i}), \mathsf{f}) : \mathsf{k} \leq \mathsf{i}\}| + in(\mathsf{n}, \mathsf{N}))\} & \text{if } \mathsf{f}(\mathsf{n}) = \{(\mathsf{T}, \mathsf{i})\} \wedge \\ & rev(\mathsf{f})(\mathsf{T}, \mathsf{i}) = \mathsf{N} \wedge |\mathsf{N}| > 1 \\ \mathsf{f}(\mathsf{n}) & \text{otherwise} \end{cases} \end{array}\right] \qquad (d) \\
\mathsf{r} = [\{(\mathsf{T}, \mathsf{i})\} \mapsto \{(\mathsf{T}, \mathsf{i}') : \exists \mathsf{n} : \mathsf{f}(\mathsf{n}) = \{(\mathsf{T}, \mathsf{i})\} \wedge rev(\mathsf{f})(\mathsf{T}, \mathsf{i}) = \mathsf{N} \ \wedge |\mathsf{N}| > 1 \wedge \\
\qquad \mathsf{i}' \in \bigcup_{\mathsf{n}' \in \mathsf{N}} \{exclude((\mathsf{T}, \mathsf{i}), \mathsf{f}) : \mathsf{k} \leq \mathsf{i}\}| + in(\mathsf{n}', \mathsf{N}))\} : |rev(\mathsf{f})(\mathsf{T}, \mathsf{i})| > 1] \\
\\
in : \mathsf{Nodes}_{\mathsf{TVAL}+} \times \wp(\mathsf{Nodes}_{\mathsf{TVAL}+}) \to \mathbb{N} \\
in(\mathsf{n}, \mathsf{N}) = |\{\mathsf{n}' \in \mathsf{N} : \mathsf{n}' <_{\mathsf{TVAL}+} \mathsf{n}\}| \\
\\
exclude : \mathsf{HId}_{\mathsf{TVAL}+} \times \Sigma_{\mathsf{TVAL}+} \to \wp(\mathbb{N}) \\
exclude((\mathsf{T}, \mathsf{i}), \mathsf{f}) = \{\mathsf{j} : \mathsf{j} \neq \mathsf{i} \wedge |rev(\mathsf{f})((\mathsf{T}, \mathsf{i}))| == 1\} \\
\\
normalize : \Sigma_{\mathsf{TVAL}+} \to (\Sigma_{\mathsf{TVAL}+} \times \mathsf{R}) \\
normalize(\mathsf{f}) = (\mathsf{f}', \mathsf{r}) : (\mathsf{f}_1, \mathsf{r}_1) = merge(\mathsf{f}) \wedge (\mathsf{f}', \mathsf{r}_2) = merge(\mathsf{f}_1) \wedge \mathsf{r} = combine(\mathsf{r}_1, \mathsf{r}_2)
\end{array}
$$

Fig. 6. Formal definition of the normalization of a TVLA state, where *combine*, given two replacements, builds up a replacement that is their concatenation

in the set of heap identifiers, and the minimum values for the counters (point b). The same approach is adopted for the global counter of the heap identifier (point a). *mergeHIds* is the basis to define the merge of a complete state performed by *merge*. This function applies *mergeHIds* to all nodes that are pointed by more than one name predicate, and it builds up a coherent replacement function (point c). After that, we have that each node is pointed to by one name predicate, but we do not have yet that each name predicate points only to a node. *split* ensures this. For each predicate name that points to (at least) two nodes, and for each pointed node by this predicate, it creates a unique predicate by modifying its counter (point d), and it builds up a coherent replacement. The modification of the counter relies on *in*, a function that, given a set of nodes, and a node in that set, returns its position inside that set. To achieve this, we suppose that a total order $<_{\mathsf{TVAL}+}$ over node identifiers is provided by TVLA. In addition, *exclude* provides the set of the counters already used in other name predicates that point only to one node, and therefore they will be in the normalized state. In this way, we cover possible holes in the counters related to a given set of basic heap identifiers, avoiding duplicates and ensuring that the set of heap identifiers



(a) Initial and final state (b) At line 4 before *normalize* (c) At line 4 after *normalize* (d) At the end before *normalize*

Fig. 7. TVLA states of the running example with name predicates

is bounded. Finally, *normalize* returns the normalized form of a given state by applying *merge* and *split* in sequence, and combining the two replacements returned by these two functions.

Running example: When we analyze the running example introduced in Section 1.1, we start from the TVLA state represented in Figure 7a. The name predicates contain four basic heap identifiers $\{\mathsf{p1}, \mathsf{p2}, \mathsf{p3}, \mathsf{p4}\}$ to name the nodes in the entry state. Their initial counter is always zero. After the analysis of line 4, we obtain the state depicted in Figure 7b. Here nodes u4 and u5 have been materialized from nodes u2 and u3 respectively, and the name predicates $(\{\mathsf{p2}\}, 0)$ and $(\{\mathsf{p3}\}, 0)$ both point to many (two) nodes. Therefore *normalize* introduces $(\{\mathsf{p2}\}, 1)$ and $(\{\mathsf{p3}\}, 1)$, and it sets them to point to one of the two nodes (Figure 7c). This is the entry state of the semantics of $\mathsf{it} = \mathsf{null}$. After the computation of the semantics of this statement, we obtain the state in Figure 7d. Here we have that $(\{\mathsf{p2}\}, 0)$ and $(\{\mathsf{p2}\}, 1)$ refer to the same node u2 (and the same happens with $(\{\mathsf{p3}\}, 0)$ and $(\{\mathsf{p3}\}, 1)$). Therefore, *normalize* merges these name predicates together into $(\{\mathsf{p2}\}, 0)$, obtaining the same state we had at the beginning of the method (Figure 7a). Note that the heap identifier $(\{\mathsf{p2}\}, 0)$ in the entry state represents something different in the exit state, since here it is the result of the merge of $\{(\{\mathsf{p2}\}, 0), (\{\mathsf{p2}\}, 1)\}$ as expressed by the replacement $\{(\{\mathsf{p2}\}, 0), (\{\mathsf{p2}\}, 1)\} \mapsto \{(\{\mathsf{p2}\}, 0)\}$. □

4.3 Abstract Semantics

The abstract semantics applies the TVLA actions introduced in Section 2.2 and normalizes the resulting states to define the semantic operators introduced in Figure 3.2. In particular, (i) $getFieldId_{\mathsf{H}}$ applies the TVLA action in Figure 2d, (ii) $assignVar_{\mathsf{H}}$ that in Figure 2e, (iii) $assignField_{\mathsf{H}}$ that in Figure 2f, and (iv) $createObject_{\mathsf{H}}$ that in Figure 2c. In all the cases, after the application of the TVLA action, the state is normalized through *normalize*. The same happens for the lattice operators. In addition, $createObject_{\mathsf{H}}$ and $getFieldId_{\mathsf{H}}$ return the heap identifier of the node pointed by temp.

5 Experimental Results

We ran TVAL+ on a set of case studies that represent a comprehensive set of common interactions with the heap and some representative examples of list

Program	#tvla	t (sec)	Program	#tvla	t (sec)
createObject	4	0,7	createObjectIfCondition	7	0,2
accessNullField	7	0,3	assignFieldSelf	7	0,3
assignNumericField	8	1,3	assignNextField	9	0,6
createAndOverWrite	9	0,7	assumeEqual	10	0,2
assumeUnequal2	10	0,2	overwriteField	10	0,4
assignAndAccessNumericField	10	1,0	conditionalAssignment	11	0,4
assignTwoFields	12	0,9	assumeUnequal	13	0,3
conditionalAssignmentVariant	14	0,6	appendByFieldAccessTwo	15	0,6
createSharedObject	19	0,8	buildList	20	0,5
appendByFieldAccessThree	22	0,9	createOneOrTwoNodes	22	1,8
accessNextSummarized	23	0,6	swapHeapObjectsOnce	24	1,8
createThreeElementList	30	2,0	linkObjects	32	1,4
createSummarizedIntList	32	2,0	swapLoop	34	0,9
linkAndTraverseObjects	38	1,9	createObjectWhile	39	1,1
assignFieldsAndSummarize	44	4,1	createPrependList	54	1,4
assignAndAddFields	55	8,3	appendByFieldAccessFour	72	4,7
traverseFixedShortList	93	3,3	createAppendList	103	3,3
initializeFixedList	109	4,5	createNumericalList	148	14,4
traverseSummarizedList	164	8,6	initializeAbstractedListFields	220	31,8
sumListElementsZero	609	86,7			

Table 1. Execution times

manipulation. We combined TVAL+ with an implementation of intervals as value domain. We ran the analysis on an Intel Core 2 Duo CPU at 2.53GHz with 4GB of RAM. We used the Java HotSpot 64-Bit Server VM included in Java SE Runtime Environment 1.6.0_26-b03. Table 1 depicts the experimental results. Column **#tvla** shows the number of invocations of TVLA performed during the analysis, while **t** reports the time of execution of the analysis.

The analysis is quite fast in many cases, since the execution rarely requires more than few seconds. Anyway, most of these case studies are composed of few sequential statements, and the examples whose analysis is slower are the ones that create a list and iterate over it. This means that the analysis has to compute a fixpoint, and this explains why TVLA is invoked many times.

For each of the case studies, we checked by hand if the heap abstraction produced by TVAL+ is what we expected, and if the information tracked by the value domain was sound and precise. In all examples, we obtained the expected results. For instance, in program sumListElementsZero we are able to (i) construct and precisely summarize a list whose nodes all contain 0 in field value, (ii) traverse the list computing the sum of the values, and (iii) prove that the sum of the elements is zero at the end of the program. This underlines that the combination of TVAL+ and the intervals domain fully benefits of the precision of TVLA, leading to really precise results on the value analysis as well.

6 Related Work

In this paper, we used an existing, well-established shape analysis to improve the results of other value analyses supported by Sample. McCloskey et al. [21] introduced a general way of integrating various analyses represented in FOLTC, combining different theories in a generic way. Their work allows the flow of information between all analyses concerned. To allow this flow, each analysis has to define classification and communication predicates, which are defined in terms of other predicates as well as core predicates that are interpreted only in that particular domain. Therefore, all analyses have to be represented in FOLTC.

In our work, we took a different approach. On the one hand, we propagate the information only *from* the heap *to* the value analysis. On the other hand, we completely automate the integration between the two analyses without enforcing any restrictions on the value domain, which could track information that is not represented in FOLTC, thus allowing easier use of existing analyses. In addition, since the information flows only in one direction, this leads to faster analyses, as we only need to propagate information once.

Gopan et al. [14] presented a framework to track numeric information on array elements. This work is specific for numeric analyses over arrays, while our approach targets any value analysis. A previous work [13] shows that its instantiation to a specific numeric domain is neither trivial nor automatic. The approach they adopted to reflect the modifications performed by the heap analysis is similar to ours. While they discharge the folding and unfolding of identifiers on the interface of the numerical analysis by adding some specific operators (namely, fold, expand, add, drop), our approach relies on assignment and forget.

Gulwani and Tiwari [15] combined analyzers represented in first order logic through an approach based on the Nelson-Oppen method. They propagate equalities both ways, but they place some restrictions on theories (e.g., convexity).

Magill et al. [19] adopted a shape analysis based on separation logic. Numerical domains are used to refine the heap analysis through counter-examples generated by the shape analysis. A potential error discovered by the shape analysis is translated into a counter-example program which is later reduced to a heapless arithmetic program. This program passes through the arithmetic analyzer in order to try and rule out the error by finding some arithmetic properties. This means they use arithmetic information only on demand to help to resolve potential heap errors, and not to prove arithmetic properties in general.

Beyer et al. [2] combined the model checker BLAST [1] with TVLA using Counter-Example Guided Abstraction Refinement for refining the shape analysis. Instead, we allow general abstract domains (not just predicate abstraction as in BLAST) at the price of having a fixed heap abstraction for the entire session.

Bouajjani et al. [3] developed a framework to statically infer properties over programs manipulating lists containing integer numerical data. This approach combines a specific heap analysis tracking information over lists with some existing numerical domains. Therefore, it cannot be automatically applied to other value analyses, or to analyze other heap structures, but it can prove properties that combine the content and the shape of lists.

Acknowledgments. Special thanks go to Roman Manevich for his support during the implementation of TVAL+. This work was partially supported by the SNF project "Verification-Driven Inference of Contracts".

References

1. D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar. The software model checker blast. *STTT*, 9(5-6):505–525, 2007.

2. D. Beyer, T. A. Henzinger, and G. Théoduloz. Lazy shape analysis. In *Proceedings of CAV '06*. ACM Press, 2006.
3. A. Bouajjani, C. Dragoi, C. Enea, and M. Sighireanu. Abstract domains for automated reasoning about list-manipulating programs with infinite data. In *Proceedings of VMCAI '12*, LNCS. Springer, 2012.
4. G. Costantini, P. Ferrara, and A. Cortesi. Static analysis of string values. In *Proceedings of ICFEM '11*, LNCS. Springer, 2011.
5. P. Cousot. The calculational design of a generic abstract interpreter. In *Calculational System Design*. NATO ASI Series F. IOS Press, Amsterdam, 1999.
6. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of POPL '77*. ACM Press, 1977.
7. P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Proceedings of POPL '79*. ACM Press, 1979.
8. P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of POPL '78*. ACM Press, 1978.
9. P. Ferrara. Checkmate: a generic static analyzer of java multithreaded programs. In *Proceedings of SEFM '09*. IEEE Computer Society, 2009.
10. P. Ferrara. Static type analysis of pattern matching by abstract interpretation. In *Proceedings of FORTE/FMOODS '09*, LNCS. Springer, 2010.
11. P. Ferrara and P. Müller. Automatic inference of access permissions. In *Proceedings of VMCAI '12*, LNCS. Springer, 2012.
12. D. Gabi. Disjunction on demand. Master thesis, ETH Zürich, 2011.
13. D. Gopan, F. DiMaio, N. Dor, T. W. Reps, and M. Sagiv. Numeric domains with summarized dimensions. In *Proceedings of TACAS '04*, LNCS. Springer, 2004.
14. D. Gopan, T. W. Reps, and M. Sagiv. A framework for numeric analysis of array operations. In *Proceedings of POPL '05*. ACM Press, 2005.
15. S. Gulwani and A. Tiwari. Combining abstract interpreters. In *Proceedings of PLDI '06*. ACM Press, 2006.
16. B. Jeannet and A. Miné. Apron: A library of numerical abstract domains for static analysis. In *Proceedings of CAV '09*, LNCS. Springer, 2009.
17. T. Lev-Ami and M. Sagiv. TVLA: A framework for kleene logic based static analyses. Master's thesis, Tel Aviv University, 2000.
18. F. Logozzo and M. Fähndrich. Static contract checking with abstract interpretation. In *Proceedings of FoVeOOS '10*, LNCS. Springer, 2010.
19. S. Magill, J. Berdine, E. M. Clarke, and B. Cook. Arithmetic strengthening for shape analysis. In *Proceedings of SAS '07*, LNCS. Springer, 2007.
20. L. Mauborgne and X. Rival. Trace partitioning in abstract interpretation based static analyzers. In *Proceedings of ESOP '05*, LNCS. Springer, 2005.
21. B. McCloskey, T. W. Reps, and M. Sagiv. Statically inferring complex heap, array, and numeric invariants. In Springer, editor, *Proceedings of SAS '10*, LNCS, 2010.
22. A. Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 2006.
23. M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *TOPLAS*, 24(3):217–298, May 2002.
24. F. Spoto. Julia: A Generic Static Analyser for the Java Bytecode. In *Proceedings of FTfJP '05*, 2005.
25. M. Zanioli, P. Ferrara, and A. Cortesi. SAILS: static analysis of information leakage with Sample. In *Proceedings of SAC '12*. ACM Press, 2012.