Generic Combination of Heap and Value Analyses in Abstract Interpretation

Pietro Ferrara[1,2]

[1] IBM Thomas J. Watson Research Center, USA
[2] ETH Zurich, Switzerland
pietroferrara@us.ibm.com

Abstract. Abstract interpretation has been widely applied to approximate data structures and (usually numerical) value information. One needs to combine them to effectively apply static analysis to real software. Nevertheless, they have been studied mainly as orthogonal problems so far. In this context, we introduce a generic framework that, given a heap and a value analysis, combines them, and we formally prove its soundness. The heap analysis approximates concrete locations with heap identifiers, that can be materialized or merged. Meanwhile, the value analysis tracks information both on variable and heap identifiers, taking into account when heap identifiers are merged or materialized. We show how existing pointer and shape analyses, as well as numerical domains, can be plugged in our framework. As far as we know, this is the first sound generic automatic framework combining heap and value analyses that allows to freely manage heap identifiers.

1 Introduction

Two major fields of static program analysis have been heap and (usually numerical) value abstractions. Venet states that "If one wants to use static analysis to support or achieve verification of *real* programs, we believe that symbolic (i.e., heap) and numerical static analysis must be tightly integrated" [33]. Nevertheless, "symbolic and numerical static analysis are commonly regarded as entirely orthogonal problems".

Object-oriented programming languages are currently mainstream in software development, and many analyzers targeting these languages have been developed. Two main lines appeared in this context: (i) analyzers focused on value information that preprocess the program applying a specific heap analysis, and replace heap accesses with symbolic variables (e.g., Clousot [23]), and (ii) heap abstractions (e.g., TVLA [22]) that do not track value information, or that have to be manually extended (e.g., with specific predicates) to track a particular type of value information [24,25]. As far as we know, existing analyzers that combine heap and value analyses are not both generic (that is, they are specific on a particular heap and/or value analysis) and automatic (that is, they require to provide some annotation, like instrumentation predicates).

K.L. McMillan and X. Rival (Eds.): VMCAI 2014, LNCS 8318, pp. 302–321, 2014.

Motivating Example. Consider the motivating example in Figure 1. Class `ListInt` represents a list of integers, with an integer field `f` (containing the value of an element) and a `ListInt next` field (pointing to the next element of the list, or to `null` if we are at the end). Method `absSum(l)` computes the sum of the absolute values of the elements in the list. Imagine that two clients call this method. `client1` passes the list [1; 2][1] to `absSum`, where the two elements are allocated at different program points (`p1` and `p2`). Instead, `client2` calls `absSum` with a list of `n` positive elements, where `n` is an input of the program. There are various properties and invariants we would like to prove and infer on such program. First of all, we would like to prove that we do not have any `NullPointerException` (property `P1`). In addition, we could discover that the value returned by `sumAbs` is positive (`P2`), or that it is greater or equal than all the elements in the list pointed by `l` (`P3`). These properties require to combine different heap and value analyses. `P1` does not require any particular numerical analysis, and for both the clients a simple and efficient heap analysis based on the allocation sites [29] would be precise enough. Instead, `P2` requires at least a numerical domain that tracks the sign of numerical variables, while `P3` requires a relational domain like Octagons [28]. In addition, for `client1` the allocation site-based heap abstraction would be precise enough both for `P2` and `P3`. Instead, on `client2` this abstraction would approximate all the nodes of the list with a unique summary node, and it would not be able to discover that the value added to `sum` is positive, since it cannot track precise information on the Boolean condition of the `if` statement. Therefore, we need a more precise heap abstraction that materializes the node pointed by `it` (e.g., shape analysis [30]).

```
int absSum(ListInt l) {
    int sum = 0;
    ListInt it = l;
    while(it != null) {
        if(it.f < 0) sum = sum − it.f;
        else sum = sum + it.f;
        it=it.next;
    }
    return sum;
}
```

Fig. 1. The motivating example

Contribution. The contribution of this work is the formalization of a sound generic analysis that allows to combine various heap and value abstractions automatically. The heap analysis approximates concrete locations through *heap identifiers*, while the value analysis tracks information on these identifiers. In addition, our framework allows the heap analysis to *freely* manage heap identifiers, and in particular to merge and materialize them. These modifications are represented by *substitutions*, and they are propagated to the value analysis. For the most part, our approach relies on standard components of abstract interpretation-based sound static analyses, and we formally define and prove the soundness of their combination. In addition, we show how to instantiate our framework with a pointer and a shape analyses, as well as with numerical

[1] [1; 2] is a shortcut to denote a list of two elements, with value `1` stored in the field `f` of the first element list, and `2` in the second one.

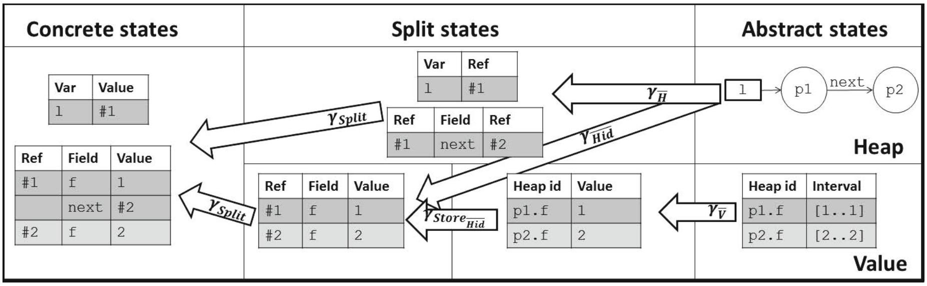

Fig. 2. The architecture of the domains in our approach

domains. This proves that our framework is expressive enough to be applied to the most common heap and value analyses.

1.1 Overview of the Framework

Domains. Figure 2 depicts the overall structure of our approach. On the left, we have standard object-oriented states composed by an environment and a store. On the right, we have our target abstract domain composed by a heap and a value abstract state. Here we represent the state of `client1` when calling method `absSum` in our motivating example. We adopt an allocation site-based heap abstraction [1] and the Interval domain [9]. Therefore, the heap analysis abstracts the list with two abstract nodes named `p1` and `p2`, while the value analysis tracks that field `f` of `p1` is [1..1], and field `f` of `p2` is [2..2].

The heap analysis concretizes to a set of environments and stores representing information only about references. This is represented in Figure 2 in the upper central box of *split* states, and it is obtained through the heap concretization $\gamma_{\overline{\mathsf{H}}}$. Similarly, the value analysis concretizes to environments and stores representing information only about values through $\gamma_{\overline{\mathsf{V}}}$. The value analysis contains information about heap identifiers `p1.f` and `p2.f`, and it needs the concretization of heap identifiers $\gamma_{\overline{\mathsf{HId}}}$ provided by $\gamma_{\overline{\mathsf{H}}}$ to produce concrete states. For instance, in Figure 2 we assume that one possible $\gamma_{\overline{\mathsf{HId}}}$ concretizes `p1.f` and `p2.f` to $(\#1, \texttt{f})$ and $(\#2, \texttt{f})$, respectively.

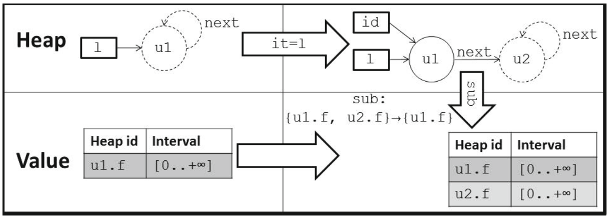

Fig. 3. The semantics' architecture of our approach

Semantics. The heap semantics may need to materialize or merge heap identifiers. The information about how the heap identifiers are modified is communicated through *substitutions*. A substitution is a function that tells the value analysis from which heap identifiers of the pre-state the identifiers of the post-state come. In this way, the value analysis can preserve the soundness of the information tracked on heap identifiers modified by the heap analysis.

For instance, consider the semantics step depicted in Figure 3. Suppose to analyze our motivating example combining a simple shape analysis [30] with the Interval domain. We analyze method `absSum` when it is called by `client2`. The abstract state on the left of Figure 3 is produced before computing the abstract semantics of line `3` in Figure 1[2]. Supposing that the node pointed by `it` has to be always definite, when we assign `l` to `it`, a definite node is materialized from `u1`. The value analysis was already tracking information about `u1.f` before this step, and it has to propagate this information to the materialized identifier `u2.f`. The heap analysis communicates a substitution `sub` to the value analysis, telling that `u1.f` and `u2.f` in the post-state derive from `u1.f` in the pre-state. Then the value analysis is updated reflecting the semantics of this substitution, that is, assigning the value tracked on `u1.f` in the pre-state to `u1.f` and `u2.f` in the post-state.

This paper formalizes this generic combination of heap and value analyses. First of all, Section 2 introduces the minimal object-oriented language we deal with. Then we formalize a standard concrete and a split domain and semantics in Section 3. Section 4 formalizes and proves the soundness of the abstract domain and semantics. Section 5 shows how to plug in our framework pointer and shape analyses, as well as numerical domains. In this way, we prove that our approach is generic enough to support some of the most common heap and value analyses. Finally, Section 6 discusses the related work, while Section 7 concludes.

Notation: In this paper, we will denote by $\rightarrow_{\mathsf{A}}$ a small-step transition semantics on the domain A, and by $\rightarrow_{\wp(\mathsf{A})}$ the pointwise application of $\rightarrow_{\mathsf{A}}$ to set of states in A. Formally, $\langle \mathtt{st}, \mathsf{A}_1 \rangle \rightarrow_{\wp(\mathsf{A})} \{\mathsf{a}' : \exists \mathsf{a} \in \mathsf{A}_1 : \langle \mathtt{st}, \mathsf{a} \rangle \rightarrow_{\mathsf{A}} \mathsf{a}'\}$ where $\mathsf{A}_1 \subseteq \mathsf{A}$. With an abuse of notation, we will denote by π_n the projection of a set of tuples (with at least n components) to the set containing the n-th component of each single tuple in the given set. Our approach is based on the abstract interpretation theory [9,10]. We will denote concrete sets and elements by C and c, respectively, and abstract sets and elements by $\overline{\mathsf{A}}$ and $\bar{\mathsf{a}}$, respectively. In addition, $\gamma_{\overline{\mathsf{A}}}$ will denote the concretization function of the abstract domain $\overline{\mathsf{A}}$.

2 Language

A program consists of a control flow graph of basic blocks. Each basic block consists of a sequence of statements. Different blocks are connected through edges that optionally contain a Boolean condition to represent conditional jumps.

[2] For the sake of simplicity, we assume that `l` is acyclic containing at least two elements.

$$\frac{\mathsf{e}' = \mathsf{e}[\mathtt{x} \mapsto \mathsf{s}(\mathsf{e}(\mathtt{y}), \mathtt{f})]}{\langle \mathtt{x} = \mathtt{y.f}, (\mathsf{e}, \mathsf{s})\rangle \rightarrow_{\Sigma} (\mathsf{e}', \mathsf{s})} \qquad \frac{\mathsf{e}' = \mathsf{e}[\mathtt{x} \mapsto \mathsf{e}(\mathtt{y})]}{\langle \mathtt{x} = \mathtt{y}, (\mathsf{e}, \mathsf{s})\rangle \rightarrow_{\Sigma} (\mathsf{e}', \mathsf{s})}$$

$$\frac{\mathsf{e}' = \mathsf{e}[\mathtt{x} \mapsto \mathit{alloc}(\mathtt{C}, (\mathsf{e}, \mathsf{s}))]}{\langle \mathtt{x} = \mathtt{new\ C}, (\mathsf{e}, \mathsf{s})\rangle \rightarrow_{\Sigma} (\mathsf{e}', \mathsf{s})} \qquad \frac{\mathsf{s}' = \mathsf{s}[(\mathsf{e}(\mathtt{x}), \mathtt{f}) \mapsto \mathsf{e}(\mathtt{y})]}{\langle \mathtt{x.f} = \mathtt{y}, (\mathsf{e}, \mathsf{s})\rangle \rightarrow_{\Sigma} (\mathsf{e}, \mathsf{s}')}$$

$$\frac{\mathsf{s}' = \mathsf{s}[(\mathsf{e}(\mathtt{x}), \mathtt{f}) \mapsto \mathit{eval}(\mathtt{vexp}, (\mathsf{e}, \mathsf{s}))]}{\langle \mathtt{x.f} = \mathtt{vexp}, (\mathsf{e}, \mathsf{s})\rangle \rightarrow_{\Sigma} (\mathsf{e}, \mathsf{s}')} \qquad \frac{\mathsf{e}' = \mathsf{e}[\mathtt{x} \mapsto \mathit{eval}(\mathtt{vexp}, (\mathsf{e}, \mathsf{s}))]}{\langle \mathtt{x} = \mathtt{vexp}, (\mathsf{e}, \mathsf{s})\rangle \rightarrow_{\Sigma} (\mathsf{e}', \mathsf{s})}$$

Fig. 4. The concrete semantics $\rightarrow_{\Sigma}$

For the sake of simplicity, we focus our attention on the statements of the minimal object-oriented language defined in Table 1. This supports assignments to variables and fields, and it distinguishes among value and reference expressions. Reference expressions can be variable identifiers, field accesses, and object creations. Value expressions can be variables, field accesses, or binary combinations of value expressions through an (e.g., arithmetic) operator. Therefore, we assume that we can distinguish between value (that is, `vexp` returning values of native types like `int` and `double` in Java) and reference expressions (that is, `rexp`). Note that in Table 1 `y` represents a variable of reference type, and $\mathtt{C} \in \mathsf{Class}$ where Class denotes the set of the classes of the object-oriented program that can be instantiated.

Table 1. Expressions and statements

$$\begin{aligned}
&\mathtt{rexp} ::= \mathtt{x} \mid \mathtt{x.f} \mid \mathtt{new\ C}\\
&\mathtt{vexp} ::= \mathtt{x} \mid \mathtt{x.f} \mid \mathtt{vexp1} < \mathtt{op} > \mathtt{vexp2}\\
&\mathtt{op} ::= + \mid - \mid * \mid \cdots\\
&\mathtt{st} ::= \mathtt{x} = \mathtt{rexp} \mid \mathtt{x.f} = \mathtt{y} \mid\\
&\qquad \mid \mathtt{x} = \mathtt{vexp} \mid \mathtt{x.f} = \mathtt{vexp}
\end{aligned}$$

3 Concrete Domain and Semantics

In this section we first introduce a standard domain (Σ) and semantics ($\rightarrow_{\Sigma}$) of object-oriented programs, and we abstract it with a split domain (Σ_{Split}) and semantics ($\rightarrow_{\mathsf{Split}}$) proving the soundness of our approach.

3.1 Standard Domain and Semantics

First of all, we partition the content of variables and heap locations into values (Val) and references (Ref). As usual in object-oriented programming languages, a state of the execution is composed by an environment (that relates local variables to references or values, $\mathsf{Env} : \mathsf{Var} \rightarrow (\mathsf{Ref} \cup \mathsf{Val})$) and a store (that relates locations to references or values, $\mathsf{Store} : (\mathsf{Ref} \times \mathsf{Field}) \rightarrow (\mathsf{Ref} \cup \mathsf{Val})$). A concrete state is defined by $\Sigma = \mathsf{Env} \times \mathsf{Store}$. The lattice structure is given by $\langle \wp(\Sigma), \subseteq \rangle$.

Semantics. Figure 4 defines a standard concrete small step semantics $\rightarrow_{\Sigma}$, while Figure 5 defines a standard concrete evaluation of value expressions. We assume that a function *alloc* : $(\mathsf{Class} \times \Sigma) \rightarrow \mathsf{Ref}$ is given. This allocates an object instance of the given class, and returns the reference pointing to it.

$$\begin{array}{l}
eval : (\mathtt{vexp} \times \Sigma) \rightarrow \mathsf{Val} \\
eval(\mathtt{x}, (\mathsf{e}, \mathsf{s})) = \mathsf{e}(\mathtt{x}) \\
eval(\mathtt{x.f}, (\mathsf{e}, \mathsf{s})) = \mathsf{s}(\mathsf{e}(\mathtt{x}), \mathtt{f}) \\
eval(\mathtt{vexp1} < \mathtt{op} > \mathtt{vexp2}, (\mathsf{e}, \mathsf{s})) = eval(\mathtt{vexp1}, (\mathsf{e}, \mathsf{s})) < \mathtt{op} > eval(\mathtt{vexp2}, (\mathsf{e}, \mathsf{s}))
\end{array}$$

Fig. 5. The concrete expression evaluation

$$\begin{array}{l}
eval_{\mathsf{Split}} : (\mathtt{vexp}' \times \Sigma_{\mathsf{Val}}) \rightarrow \mathsf{Val} \\
eval_{\mathsf{Split}}(\mathtt{x}, (\mathsf{e}_{\mathsf{Val}}, \mathsf{s}_{\mathsf{Val}})) = \mathsf{e}_{\mathsf{Val}}(\mathtt{x}) \\
eval_{\mathsf{Split}}(< \mathtt{r} > .\mathtt{f}, (\mathsf{e}_{\mathsf{Val}}, \mathsf{s}_{\mathsf{Val}})) = \mathsf{s}_{\mathsf{Val}}(\mathsf{r}, \mathtt{f}) \\
eval_{\mathsf{Split}}(\mathtt{vexp1} < \mathtt{op} > \mathtt{vexp2}, (\mathsf{e}_{\mathsf{Val}}, \mathsf{s}_{\mathsf{Val}})) = \\
\qquad\qquad = eval_{\mathsf{Split}}(\mathtt{vexp1}, (\mathsf{e}_{\mathsf{Val}}, \mathsf{s}_{\mathsf{Val}})) < \mathtt{op} > eval_{\mathsf{Split}}(\mathtt{vexp2}, (\mathsf{e}_{\mathsf{Val}}, \mathsf{s}_{\mathsf{Val}}))
\end{array}$$

Fig. 6. The split expression evaluation

3.2 Split Domain

We split the concrete domain and semantics between the portion dealing with values ($\mathsf{Env}_{\mathsf{Val}} : \mathsf{Var} \rightarrow \mathsf{Val}$, $\mathsf{Store}_{\mathsf{Val}} : (\mathsf{Ref} \times \mathsf{Field}) \rightarrow \mathsf{Val}$, and $\Sigma_{\mathsf{Val}} = \mathsf{Env}_{\mathsf{Val}} \times \mathsf{Store}_{\mathsf{Val}}$), and the portion dealing with references ($\mathsf{Env}_{\mathsf{Ref}} : \mathsf{Var} \rightarrow \mathsf{Ref}$, $\mathsf{Store}_{\mathsf{Ref}} : (\mathsf{Ref} \times \mathsf{Field}) \rightarrow \mathsf{Ref}$, and $\Sigma_{\mathsf{Ref}} = \mathsf{Env}_{\mathsf{Ref}} \times \mathsf{Store}_{\mathsf{Ref}}$). A state is then the Cartesian product of these two components ($\Sigma_{\mathsf{Split}} = \Sigma_{\mathsf{Ref}} \times \Sigma_{\mathsf{Val}}$). Like the concrete domain, the lattice structure is given by set of elements, that is, $\langle \wp(\Sigma_{\mathsf{Split}}), \subseteq \rangle$.

Soundness. To prove the soundness of $\langle \wp(\Sigma_{\mathsf{Split}}), \subseteq \rangle$ with respect to $\langle \wp(\Sigma), \subseteq \rangle$, we have to formalize the concretization $\gamma_{\mathsf{Split}} : \wp(\Sigma_{\mathsf{Split}}) \rightarrow \wp(\Sigma)$ that defines how states in Σ_{Split} are mapped into states in Σ. Intuitively, this consists in the pointwise set union of the two parts of split states. Formally, $\gamma_{\mathsf{Split}}(\mathsf{T}) = \{(\mathsf{e}_v \cup \mathsf{e}_h, \mathsf{s}_v \cup \mathsf{s}_h) : ((\mathsf{e}_h, \mathsf{s}_h), (\mathsf{e}_v, \mathsf{s}_v)) \in \mathsf{T}\}$. Note that, since in the definition of the language in Section 2 we assumed that we can distinguish among value and reference expressions (and in particular local variables and field accesses), the domains of e_v and e_h do not overlap. The same considerations apply to s_v and s_h.

Then, we have that $\langle \wp(\Sigma_{\mathsf{Split}}), \subseteq \rangle$ is a sound approximation of $\langle \wp(\Sigma), \subseteq \rangle$, that is, they form a Galois connection.

Semantics. Figure 6 defines the evaluation of expressions in $\mathtt{vexp}'$, where $\mathtt{vexp}' ::= \mathtt{x} \mid < \mathtt{r} > .\mathtt{f} \mid \mathtt{vexp1}' < \mathtt{op} > \mathtt{vexp2}'$ with $\mathtt{r} \in \mathsf{Ref}$. The main difference w.r.t. the concrete expression evaluation defined by Figure 5 is that it deals only with the value portion of the heap state. This is possible since $\mathtt{vexp}'$ contains

$$\frac{\mathsf{s}_{\mathsf{Val}}^{'} = \mathsf{s}_{\mathsf{Val}}[(\mathtt{r}, \mathtt{f}) \mapsto eval_{\mathsf{Split}}(\mathtt{vexp}', (\mathsf{e}_{\mathsf{Val}}, \mathsf{s}_{\mathsf{Val}}))]}{\langle < \mathtt{r} > .\mathtt{f} = \mathtt{vexp}^{'}, (\mathsf{e}_{\mathsf{Val}}, \mathsf{s}_{\mathsf{Val}}) \rangle \rightarrow_{\mathsf{Val}} (\mathsf{e}_{\mathsf{Val}}, \mathsf{s}_{\mathsf{Val}}^{'})} \quad \frac{\mathsf{e}_{\mathsf{Val}}^{'} = \mathsf{e}_{\mathsf{Val}}[\mathtt{x} \mapsto eval_{\mathsf{Split}}(\mathtt{vexp}', (\mathsf{e}_{\mathsf{Val}}, \mathsf{s}_{\mathsf{Val}}))]}{\langle \mathtt{x} = \mathtt{vexp}^{'}, (\mathsf{e}_{\mathsf{Val}}, \mathsf{s}_{\mathsf{Val}}) \rangle \rightarrow_{\mathsf{Val}} (\mathsf{e}_{\mathsf{Val}}^{'}, \mathsf{s}_{\mathsf{Val}})}$$

Fig. 7. The semantics of the value part

$$\frac{\mathsf{e}'_{\mathsf{Ref}} = \mathsf{e}_{\mathsf{Ref}}[\mathtt{x} \mapsto \mathit{alloc}(\mathtt{C}, (\mathsf{e}_{\mathsf{Ref}}, \mathsf{s}_{\mathsf{Ref}}))]}{\langle \mathtt{x} = \mathtt{new}\ \mathtt{C}, (\mathsf{e}_{\mathsf{Ref}}, \mathsf{s}_{\mathsf{Ref}})\rangle \rightarrow_{\mathsf{Ref}} (\mathsf{e}'_{\mathsf{Ref}}, \mathsf{s}_{\mathsf{Ref}})} \qquad \frac{\mathsf{e}'_{\mathsf{Ref}} = \mathsf{e}_{\mathsf{Ref}}[\mathtt{x} \mapsto \mathsf{s}_{\mathsf{Ref}}(\mathsf{e}_{\mathsf{Ref}}(\mathtt{y}), \mathtt{f})]}{\langle \mathtt{x} = \mathtt{y.f}, (\mathsf{e}_{\mathsf{Ref}}, \mathsf{s}_{\mathsf{Ref}})\rangle \rightarrow_{\mathsf{Ref}} (\mathsf{e}'_{\mathsf{Ref}}, \mathsf{s}_{\mathsf{Ref}})}$$

$$\frac{\mathsf{s}'_{\mathsf{Ref}} = \mathsf{s}_{\mathsf{Ref}}[(\mathsf{e}_{\mathsf{Ref}}(\mathtt{x}), \mathtt{f}) \mapsto \mathsf{e}_{\mathsf{Ref}}(\mathtt{y})]}{\langle \mathtt{x.f} = \mathtt{y}, (\mathsf{e}_{\mathsf{Ref}}, \mathsf{s}_{\mathsf{Ref}})\rangle \rightarrow_{\mathsf{Ref}} (\mathsf{e}_{\mathsf{Ref}}, \mathsf{s}'_{\mathsf{Ref}})} \qquad \frac{\mathsf{e}'_{\mathsf{Ref}} = \mathsf{e}_{\mathsf{Ref}}[\mathtt{x} \mapsto \mathsf{e}_{\mathsf{Ref}}(\mathtt{y})]}{\langle \mathtt{x} = \mathtt{y}, (\mathsf{e}_{\mathsf{Ref}}, \mathsf{s}_{\mathsf{Ref}})\rangle \rightarrow_{\mathsf{Ref}} (\mathsf{e}'_{\mathsf{Ref}}, \mathsf{s}_{\mathsf{Ref}})}$$

Fig. 8. The semantics of the heap part

$$\frac{\langle \mathtt{x} = \mathtt{rexp}, \sigma_{\mathsf{Ref}}\rangle \rightarrow_{\mathsf{Ref}} \sigma'_{\mathsf{Ref}}}{\langle \mathtt{x} = \mathtt{rexp}, (\sigma_{\mathsf{Ref}}, \sigma_{\mathsf{Val}})\rangle \rightarrow_{\mathsf{Split}} (\sigma'_{\mathsf{Ref}}, \sigma_{\mathsf{Val}})} \qquad \frac{\langle \mathtt{x.f} = \mathtt{y}, \sigma_{\mathsf{Ref}}\rangle \rightarrow_{\mathsf{Ref}} \sigma'_{\mathsf{Ref}}}{\langle \mathtt{x.f} = \mathtt{y}, (\sigma_{\mathsf{Ref}}, \sigma_{\mathsf{Val}})\rangle \rightarrow_{\mathsf{Split}} (\sigma'_{\mathsf{Ref}}, \sigma_{\mathsf{Val}})}$$

$$\frac{\langle \mathbb{R}[\![\mathtt{x.f}, \sigma_{\mathsf{Ref}}]\!] = \mathbb{R}[\![\mathtt{vexp}, \sigma_{\mathsf{Ref}}]\!], \sigma_{\mathsf{Val}}\rangle \rightarrow_{\mathsf{Val}} \sigma'_{\mathsf{Val}}}{\langle \mathtt{x.f} = \mathtt{vexp}, (\sigma_{\mathsf{Ref}}, \sigma_{\mathsf{Val}})\rangle \rightarrow_{\mathsf{Split}} (\sigma_{\mathsf{Ref}}, \sigma'_{\mathsf{Val}})} \qquad \frac{\langle \mathtt{x} = \mathbb{R}[\![\mathtt{vexp}, \sigma_{\mathsf{Ref}}]\!], \sigma_{\mathsf{Val}}\rangle \rightarrow_{\mathsf{Val}} \sigma'_{\mathsf{Val}}}{\langle \mathtt{x} = \mathtt{vexp}, (\sigma_{\mathsf{Ref}}, \sigma_{\mathsf{Val}})\rangle \rightarrow_{\mathsf{Split}} (\sigma_{\mathsf{Ref}}, \sigma'_{\mathsf{Val}})}$$

Fig. 9. The semantics of the split domain

a reference instead of a local variable when accessing a field with statement $< \mathtt{r} >$ .f. Then, we have that $\rightarrow_{\mathsf{Split}}$ is a sound approximation of $\rightarrow_{\Sigma}$.

The small-step semantics $\rightarrow_{\mathsf{Split}}$ over Σ_{Split} is formalized by Figure 9. It mainly applies the proper semantics of the value (Figure 7) or the heap (Figure 8) part of the state by looking to the statement. The only noticeable difference appears when we deal with statements requiring both value and heap state (namely, $\mathtt{x} = \mathtt{vexp}$ and $\mathtt{x.f} = \mathtt{vexp}$). In these cases, we preprocess vexp and x.f with the following function $\mathbb{R}$:

$\mathbb{R} : (\mathtt{vexp} \times \Sigma_{\mathsf{Ref}}) \rightarrow \mathtt{vexp}'$
$\mathbb{R}[\![\mathtt{x}, (\mathsf{e}_{\mathsf{Ref}}, \mathsf{s}_{\mathsf{Ref}})]\!] = \mathtt{x}$
$\mathbb{R}[\![\mathtt{x.f}, (\mathsf{e}_{\mathsf{Ref}}, \mathsf{s}_{\mathsf{Ref}})]\!] = <\mathsf{e}_{\mathsf{Ref}}(\mathtt{x})> .\mathtt{f}$
$\mathbb{R}[\![\mathtt{vexp1} < \mathtt{op} > \mathtt{vexp2}, (\mathsf{e}_{\mathsf{Ref}}, \mathsf{s}_{\mathsf{Ref}})]\!] =$
$= \mathbb{R}[\![\mathtt{vexp1}, (\mathsf{e}_{\mathsf{Ref}}, \mathsf{s}_{\mathsf{Ref}})]\!] < \mathtt{op} > \mathbb{R}[\![\mathtt{vexp2}, (\mathsf{e}_{\mathsf{Ref}}, \mathsf{s}_{\mathsf{Ref}})]\!]$

This function replaces in a value expression the local variable x in a field access x.f with the reference r pointed by x in the reference environment. This step is necessary to allow the evaluation of value expressions to perform without any knowledge of the actual state of the reference part.

4 Abstract Domain and Semantics

The reference (Σ_{Ref}) and the value (Σ_{Val}) part of the split domain are approximated by a given heap ($\overline{\mathsf{H}}$) and value ($\overline{\mathsf{V}}$) analysis, respectively. In addition, the heap analysis defines a set of heap identifiers $\overline{\mathsf{HId}}$ which aims at abstracting concrete locations, and the value analysis tracks information over these identifiers like it does over variable identifiers. Since we want to allow the heap analysis to merge and materialize heap identifiers, we have to communicate these changes to the value analysis through substitutions. In this Section, we formalize and prove the soundness of this framework.

4.1 Abstract Domain

We assume that a value analysis $\overline{\mathsf{V}}$ and a heap analysis $\overline{\mathsf{H}}$ are provided with lattice operators ($\langle \overline{\mathsf{V}}, \sqsubseteq_{\overline{\mathsf{V}}}, \sqcup_{\overline{\mathsf{V}}}, \sqcap_{\overline{\mathsf{V}}} \rangle$ and $\langle \overline{\mathsf{H}}, \sqsubseteq_{\overline{\mathsf{H}}} \sqcup_{\overline{\mathsf{H}}}, \sqcap_{\overline{\mathsf{H}}} \rangle$, respectively). In addition, they provide the widening operators $\nabla_{\overline{\mathsf{V}}}$ and $\nabla_{\overline{\mathsf{H}}}$, respectively. The states $\overline{\Sigma}$ of our abstract domain are composed by a heap and a value state ($\overline{\Sigma} = \overline{\mathsf{H}} \times \overline{\mathsf{V}}$). Then a state of our analysis is the Cartesian product of $\overline{\mathsf{H}}$ and $\overline{\mathsf{V}}$ denoted by $\langle \overline{\Sigma}, \sqsubseteq_{\overline{\Sigma}}, \sqcup_{\overline{\Sigma}}, \sqcap_{\overline{\Sigma}} \rangle$.

4.2 Concretization Function

We assume that the heap analysis defines a finite set of heap identifiers $\overline{\mathsf{HId}}$. In addition, it provides a function $\overline{\mathit{heapId}} : \overline{\mathsf{H}} \rightarrow \wp(\overline{\mathsf{HId}})$ that returns the set of heap identifiers contained in a given abstract heap. The value analysis tracks information on variables as well as heap identifiers. Therefore, the concretization of the value analysis produces (i) environments in $\mathsf{Env}_{\mathsf{Val}}$, and (ii) stores with abstract heap identifiers instead of concrete locations since the value analysis alone cannot concretize heap identifiers ($\mathsf{Store}_{\overline{\mathsf{HId}}} : \overline{\mathsf{HId}} \rightarrow \wp(\mathsf{Val})$). Note that we have a set of concrete values as codomain since a single heap identifier, representing a summary node, may concretize into many concrete references that could have different values in the same store's concretization. For instance, a list of positive values may be approximated by a single heap identifier `u`. Imagine that we are dealing with a particular concretization of this list containing two elements. These two elements are not necessarily equal, so for instance the value analysis could tell us that `u.f` concretizes to $\{1, 2\}$ (e.g., to represent a list $[1; 2]$).

To concretize a store in $\mathsf{Store}_{\overline{\mathsf{HId}}}$ to Store, we need that the heap analysis provides the concretization of heap identifiers. Therefore, the heap concretization has to provide a function $\gamma_{\overline{\mathsf{HId}}}$ that relates each heap identifier to a set of concrete locations ($\overline{\mathsf{HId}} \rightarrow \wp(\mathsf{Ref} \times \mathsf{Field})$). Also in this case, we need to have a set of concrete locations as codomain in order to support summary nodes.

In this way, the heap analysis can track the shape of the heap, and represent symbolically nodes using heap identifiers. When it concretizes, the concrete values of references in one concrete store transform the shape into a concrete memory state. In this scenario, the heap identifiers' concretization is the component that tells us how we go from the shape to the concrete references. Note that a single shape could concretize to a (possibly infinite) set of concrete stores, since there are infinitely many possible reference values for the heap identifiers, and the heap identifiers' concretization is specific for one concrete store.

Formally, the heap concretization $\gamma_{\overline{\mathsf{H}}}$ returns a set of pairs containing a concrete store, and a concretization of heap identifiers ($\gamma_{\overline{\mathsf{H}}} : \overline{\mathsf{H}} \rightarrow \wp(\Sigma_{\mathsf{Ref}} \times (\overline{\mathsf{HId}} \rightarrow \wp(\mathsf{Ref} \times \mathsf{Field}))))$. The mapping of heap identifiers to sets of concrete locations is necessary to concretize later value stores.

We assume that the heap and value analyses are sound, that is, they form a Galois connection. Formally, $\langle \wp(\mathsf{Env}_{\mathsf{Val}} \times \mathsf{Store}_{\overline{\mathsf{HId}}}), \subseteq\rangle \underset{\alpha_{\overline{\mathsf{V}}}}{\overset{\gamma_{\overline{\mathsf{V}}}}{\leftrightarrows}} \langle \overline{\mathsf{V}}, \sqsubseteq_{\overline{\mathsf{V}}}\rangle$ and $\langle \wp(\Sigma_{\mathsf{Ref}}), \subseteq\rangle \underset{\alpha_{\overline{\mathsf{H}}}}{\overset{\pi_1 \circ \gamma_{\overline{\mathsf{H}}}}{\leftrightarrows}} \langle \overline{\mathsf{H}}, \sqsubseteq_{\overline{\mathsf{H}}}\rangle$. In addition, we assume that $\gamma_{\overline{\mathsf{V}}}$ and $\pi_1(\gamma_{\overline{\mathsf{H}}})$ are complete meet-morphisms.

We are now in position to combine the heap and value concretizations to concretize abstract states in $\overline{\Sigma}$ to $\wp(\Sigma_{\mathsf{Split}})$. What is still missing is the concretization of $\mathsf{Store}_{\overline{\mathsf{HId}}}$ to Store. Intuitively, the resulting stores should relate a location $(\mathsf{r}, \mathtt{f})$ with the values related to a heap identifier that is concretized into $(\mathsf{r}, \mathtt{f})$. Formally, we define $\gamma_{\mathsf{Store}_{\overline{\mathsf{HId}}}} : (\mathsf{Store}_{\overline{\mathsf{HId}}} \times (\overline{\mathsf{HId}} \to \wp(\mathsf{Ref} \times \mathsf{Field}))) \to \wp(\mathsf{Store}_{\mathsf{Val}})$ as follows:

$$\gamma_{\mathsf{Store}_{\overline{\mathsf{HId}}}}(\mathsf{s}, \gamma_{\overline{\mathsf{HId}}}) = \{[(\mathsf{r}, \mathtt{f}) \mapsto \mathsf{v}] : \mathsf{i} \in \mathit{dom}(\gamma_{\overline{\mathsf{HId}}}) \wedge (\mathsf{r}, \mathtt{f}) \in \gamma_{\overline{\mathsf{HId}}}(\mathsf{i}) \wedge \mathsf{v} \in \mathsf{s}(\mathsf{i})\}$$

Finally, the concretization of abstract states $\gamma_{\overline{\Sigma}} : \overline{\Sigma} \to \wp(\Sigma_{\mathsf{Split}})$ is defined by:

$$\gamma_{\overline{\Sigma}}(\overline{\mathsf{v}}, \overline{\mathsf{h}}) = \{(\sigma_H, (\mathsf{e}_v, \mathsf{s}'_v)) : (\mathsf{e}_v, \mathsf{s}_v) \in \gamma_{\overline{\mathsf{V}}}(\overline{\mathsf{v}}) \wedge (\sigma_H, \gamma_{\overline{\mathsf{HId}}}) \in \gamma_{\overline{\mathsf{H}}}(\overline{\mathsf{h}}) \wedge \\ \mathsf{s}'_v \in \gamma_{\mathsf{Store}_{\overline{\mathsf{HId}}}}(\mathsf{s}_v, \gamma_{\overline{\mathsf{HId}}})\}$$

Running Example. Consider now the motivating example of Figure 1 with the list passed by `client1`, and the abstract state depicted in the right part of Figure 2. The numerical domain concretizes the store (s_v in the definition of $\gamma_{\overline{\Sigma}}$) to $\{[(\mathtt{p1}, \mathtt{f}) \mapsto \{1\}, (\mathtt{p2}, \mathtt{f}) \mapsto \{2\}]\}$ while the value environment e_v is empty since there is no local value variable. Instead, $\gamma_{\overline{\mathsf{H}}}$ may concretize to many heaps with different $\gamma_{\overline{\mathsf{HId}}}$. For the sake of simplicity, let us focus on the case in which $\gamma_{\overline{\mathsf{HId}}} = [(\mathtt{p1}, \mathtt{f}) \mapsto \{(\#1, \mathtt{f})\}, (\mathtt{p2}, \mathtt{f}) \mapsto \{(\#2, \mathtt{f})\}]$. Then $\gamma_{\mathsf{Store}_{\overline{\mathsf{HId}}}}$ returns the numerical store $s'_v = [(\#1, \mathtt{f}) \mapsto 1, (\#2, \mathtt{f}) \mapsto 2]$, that is, it substitutes $(\mathtt{p1}, \mathtt{f})$ and $(\mathtt{p2}, \mathtt{f})$ with $(\#1, \mathtt{f})$ and $(\#2, \mathtt{f})$ in s_v, respectively.

Soundness. We need to assume some conditions on how heap identifiers are concretized in order to prove the soundness of the analysis.

C1 $\gamma_{\overline{\mathsf{HId}}}$ has to define the concretization of all the heap identifiers contained in the concretized heap. Formally, $\forall \gamma_{\overline{\mathsf{HId}}} \in \pi_2(\gamma_{\overline{\mathsf{H}}}(\overline{\mathsf{h}})) : \mathit{dom}(\gamma_{\overline{\mathsf{HId}}}) = \overline{\mathit{heapId}}(\overline{\mathsf{h}})$.

C2 If a heap identifier is not in all the states we are intersecting, then it will not be part of the results of the intersection, since through the greatest lower bound we are taking only the part that is common among all the states we are intersecting. Formally, $\forall \overline{\mathsf{H}}_1 \subseteq \overline{\mathsf{H}} : \overline{\mathit{heapId}}(\bigsqcap_{\overline{\mathsf{h}} \in \overline{\mathsf{H}}_1} \overline{\mathsf{h}}) = \bigcap_{\overline{\mathsf{h}} \in \overline{\mathsf{H}}_1} \overline{\mathit{heapId}}(\overline{\mathsf{h}})$.

C3 Different heap identifiers represent different portions of the heap. Formally, $\forall \mathsf{i}_1, \mathsf{i}_2 \in \mathit{dom}(\gamma_{\overline{\mathsf{HId}}}) : \mathsf{i}_1 \neq \mathsf{i}_2 \Rightarrow \gamma_{\overline{\mathsf{HId}}}(\mathsf{i}_1) \cap \gamma_{\overline{\mathsf{HId}}}(\mathsf{i}_2) = \emptyset$.

C4 When we intersect a set of abstract heaps, the heap identifiers' concretization is the pointwise intersection of the heap identifiers' concretization of all the intersected states. Formally, $\forall (\sigma_H, \gamma_{\overline{\mathsf{HId}}}) \in \gamma_{\overline{\mathsf{H}}}(\bigsqcap_{\overline{\mathsf{h}} \in \overline{\mathsf{H}}_1} \overline{\mathsf{h}}), \forall (\sigma_H, \gamma^i_{\overline{\mathsf{HId}}}) \in \gamma_{\overline{\mathsf{H}}}(\overline{\mathsf{h}}_i) : \overline{\mathsf{h}}_i \in \overline{\mathsf{H}}_1 \Rightarrow \gamma_{\overline{\mathsf{HId}}} = \lambda \mathsf{x}. \bigcap_{\overline{\mathsf{h}}_i \in \overline{\mathsf{H}}_1} \gamma^i_{\overline{\mathsf{HId}}}(\mathsf{x})$.

Condition **C1** is necessary since otherwise the value analysis could track information on heap identifiers that it does not know how to concretize. Condition **C3** is a rather standard assumption over abstract nodes in heap analysis (e.g., in shape analysis [30]), and it states that different identifiers represents different portions of the heap. In this way, the assignment of an identifier is guaranteed not to affect other identifiers. Condition **C2** and **C4** are both necessary to prove that $\gamma_{\overline{\Sigma}}$ is meet-preserving. This is a fundamental property for Galois connections induced by a glb-preserving concretization function (e.g., see Proposition 7 of [11]). Intuitively, they correspond to the requirement that $\pi_1(\gamma_{\overline{\mathsf{H}}})$ and $\gamma_{\overline{\mathsf{V}}}$ are complete meet-morphism applied to $\pi_2(\gamma_{\overline{\mathsf{H}}})$.

Theorem 1 ($\langle\overline{\Sigma}, \sqsubseteq\rangle$ is a sound approximation of $\langle\wp(\Sigma_{\mathsf{Split}}), \subseteq\rangle$) $\langle\wp(\Sigma_{\mathsf{Split}}), \subseteq\rangle \overset{\gamma_{\overline{\Sigma}}}{\underset{\alpha_{\overline{\Sigma}}}{\leftrightarrows}} \langle\overline{\Sigma}, \sqsubseteq\rangle$ *where* $\alpha_{\overline{\Sigma}} = \lambda \mathsf{X}.\, \sqcap_{\overline{\Sigma}} \{\overline{\mathsf{x}} : \mathsf{X} \subseteq \gamma_{\overline{\Sigma}}(\overline{\mathsf{x}})\}$.

Since Galois connections compose [9], we have that $\langle\wp(\Sigma), \subseteq\rangle \overset{\gamma_{\wp(\mathsf{Split})} \circ \gamma_{\overline{\Sigma}}}{\underset{\alpha_{\overline{\Sigma}} \circ \alpha_{\wp(\mathsf{Split})}}{\leftrightarrows}}$ $\langle\overline{\Sigma}, \sqsubseteq\rangle$ that is, $\overline{\Sigma}$ is sound with respect to the standard concrete domain $\wp(\Sigma)$.

4.3 Substitutions

Substitutions allow the heap analysis to freely manage heap identifiers when applying semantic operators. They are defined by $\overline{\mathsf{Sub}} : \wp(\overline{\mathsf{HId}}) \to \wp(\overline{\mathsf{HId}})$. The meaning of a relation in a substitution is that the identifiers in the domain are in the post-state, and they derive from the identifiers in the pre-state they are in relation with. For instance, $[\{\mathsf{id1}, \mathsf{id2}\} \mapsto \{\mathsf{id3}\}]$ represents that $\mathsf{id1}$ and $\mathsf{id2}$ are materialized from $\mathsf{id3}$, while $[\{\mathsf{id1}\} \mapsto \{\mathsf{id2}, \mathsf{id3}\}]$ means that $\mathsf{id2}$ and $\mathsf{id3}$ are merged into $\mathsf{id1}$.

In our notation, we will represent by $\to_{\overline{\mathsf{H}}}^{\mathsf{sub}}$ that the heap semantic operator $\to_{\overline{\mathsf{H}}}$ produced the substitution sub. Function $\overline{applySub} : (\overline{\mathsf{V}} \times \overline{\mathsf{Sub}}) \to \overline{\mathsf{V}}$ applies a substitution to a state of the value analysis, and it is defined by:

$$\begin{aligned}\overline{applySub}(\overline{\mathsf{v}}, \mathsf{sub}) = \overline{\mathsf{v}}_n \text{ where } \mathsf{sub} = [\mathsf{l}_1 \mapsto \mathsf{l}'_1, ..., \mathsf{l}_n \mapsto \mathsf{l}'_n], \overline{\mathsf{v}}_0 = \overline{\mathsf{v}} \wedge \\ \forall j \in [1..n] : \overline{\mathsf{v}}_j = \sqcup_{\mathsf{i}' \in \mathsf{l}'_j} \{\overline{\mathsf{v}}'_n : \mathsf{l}_j = \{\mathsf{i}_1, \cdots, \mathsf{i}_n\}, \overline{\mathsf{v}}'_0 = \overline{\mathsf{v}}_{j-1}, \\ \forall \mathsf{k} \in [1..n] : \langle \mathsf{i}_{\mathsf{k}} = \mathsf{i}', \overline{\mathsf{v}}'_{k-1}\rangle \to_{\overline{\mathsf{V}}} \overline{\mathsf{v}}'_k\}\end{aligned}$$

The intuition behind $\overline{applySub}$ is that, given a substitution sub, each single *replacement* $\mathsf{l}_j \mapsto \mathsf{l}'_j \in \mathsf{sub}$ represents that the identifiers in l'_j are substituted by l_j. Therefore, each identifier $\mathsf{i}' \in \mathsf{l}'_j$ is assigned to each identifier $\mathsf{i} \in \mathsf{l}_j$.

Soundness. We expect that the substitution produced by a heap semantic operator is coherent with respect to the modifications of the heap identifiers that have been induced by such operator. This means that, if $\langle \mathtt{st}, \overline{\mathsf{h}}\rangle \to_{\overline{\mathsf{H}}}^{\mathsf{sub}} \overline{\mathsf{h}}'$ and $[\mathsf{l} \mapsto \mathsf{l}'] \in \mathsf{sub}$, the concrete locations represented by l' in the pre-state corresponds to what is represented by l in the post-state. This correspondence is bound to heap concretization that are related through the concrete heap semantics $\to_{\mathsf{Ref}}$. This concept is formalized by the following proposition.

Proposition 1 (Soundness of the substitution). *Let* $\overline{\mathsf{h}} \in \overline{\mathsf{H}}$ *be a state of the heap analysis such that* $\langle \mathtt{st}, \overline{\mathsf{h}}\rangle \to_{\overline{\mathsf{H}}}^{\mathsf{sub}} \overline{\mathsf{h}}'$.

A substitution is sound iff $\forall(\mathsf{h}, \gamma_{\overline{\mathsf{HId}}}) \in \gamma_{\overline{\mathsf{H}}}(\overline{\mathsf{h}}) : \langle \mathtt{st}, \mathsf{h}\rangle \to_{\mathsf{Ref}} \mathsf{h}', (\mathsf{h}', \gamma'_{\overline{\mathsf{HId}}}) \in \gamma_{\overline{\mathsf{H}}}(\overline{\mathsf{h}}')$ *we have that* $\gamma'_{\overline{\mathsf{HId}}} = \gamma_{\overline{\mathsf{HId}}}[\mathsf{i} \mapsto \mathsf{l}' : \mathsf{l}' \subseteq \bigcup_{\mathsf{i}_1 \in \mathsf{sub}(\mathsf{l})} \gamma_{\overline{\mathsf{HId}}}(\mathsf{i}_1) \wedge \exists \mathsf{l} \in dom(\mathsf{sub}) : \mathsf{i} \in \mathsf{l}]$ *and* $\forall \mathsf{l} \in dom(\mathsf{sub}) : \bigcup_{\mathsf{i} \in \mathsf{l}} \gamma'_{\overline{\mathsf{HId}}}(\mathsf{i}) = \bigcup_{\mathsf{i}' \in \mathsf{sub}(\mathsf{l})} \gamma_{\overline{\mathsf{HId}}}(\mathsf{i}')$.

Intuitively, the substitution univocally establishes how heap identifiers' concretization is affected by the heap semantic operator. In addition, since a substitution represents how heap identifiers are modified in a single step, we do not want that different replacements in the same substitution overlap. The intuition is that a set of heap identifiers can be substituted by another set, but during one single substitution the same heap identifiers cannot be in many replacements. For instance, imagine that we have the substitution $[\{\mathtt{id1}\} \mapsto \{\mathtt{id2}\}, \{\mathtt{id2}\} \mapsto \{\mathtt{id3}\}]$. In this case, it is not clear what is represented. In particular, we could have that (i) `id1` is replaced by `id2` that is then replaced by `id3`, or (ii) `id2` is replaced by `id3` and `id1` is replaced by `id2`. To avoid this ambiguity, we do not allow this scenario. Nevertheless, the effects of overlapping substitutions (like the one we sketched) can be obtained by a sequence of non-overlapping substitutions that disambiguate the semantics. For instance, (i) corresponds to $[\{\mathtt{id1}\} \mapsto \{\mathtt{id2}\}]$ followed by $[\{\mathtt{id2}\} \mapsto \{\mathtt{id3}\}]$, while (ii) corresponds to $[\{\mathtt{id2}\} \mapsto \{\mathtt{id3}\}]$ followed by $[\{\mathtt{id1}\} \mapsto \{\mathtt{id2}\}]$.

Proposition 2 (Non-overlapping replacements). *We assume that single replacements in the same substitution do not overlap. Formally,* $\forall \mathsf{l} \in \mathit{dom}(\mathsf{sub})$, $\forall \mathsf{l}' \in \mathit{dom}(\mathsf{sub}) \setminus \{\mathsf{l}\} : \mathsf{l} \cap \mathsf{l}' = \emptyset \wedge \mathsf{sub}(\mathsf{l}) \cap \mathsf{sub}(\mathsf{l}') = \emptyset$.

Running Example. Consider again the program in Figure 1, and suppose to have a transition like the one depicted in Figure 3. The materialization of `u2` produces the replacement $[\{\mathtt{u1.f}, \mathtt{u2.f}\} \mapsto \{\mathtt{u1.f}\}]$, and its application to the numerical state produces $[\mathtt{u1.f} \mapsto [0..+\infty], \mathtt{u2.f} \mapsto [0..+\infty]]$.

This replacement satisfies the soundness conditions of the substitution. Intuitively, the two heap identifiers `u1.f` and `u2.f` in the post state corresponds to `u1.f` in the heap state. Therefore, given a particular concrete state, the concretization of the heap identifiers of `u1.f` and `u2.f` in the post-state corresponds to the concretization of `u1.f` in the pre-state, that is exactly what is stated by Proposition 1. In addition, the substitution does not overlap, and therefore it satisfies Proposition 2.

4.4 Semantics

In our split domain, we assumed that the value part Σ_{Val} takes care of statements about values, while the heap part Σ_{Ref} defines the semantics of statements dealing with the heap. Nevertheless, we needed the heap state to replace field accesses with a reference and the accessed field in value expressions, and when assigning a value to a location. Similarly, we will have to replace field accesses with heap identifiers when defining the abstract semantics.

Let us define $\overline{\mathsf{vexp}} ::= \mathtt{x} \,|\, \mathtt{i} \,|\, \mathtt{vexp1} < \mathtt{op} > \mathtt{vexp2}$ where $\mathsf{i} \in \overline{\mathsf{HId}}$. $\overline{\mathsf{vexp}}$ is the abstract counterpart of $\mathtt{vexp}'$. We assume that $\overline{\mathsf{V}}$ provides the semantics of value assignment $\langle \mathtt{i} = \overline{\mathsf{vexp}}, \overline{\mathsf{v}} \rangle \rightarrow_{\overline{\mathsf{V}}} \overline{\mathsf{v}}'$, and that $\overline{\mathsf{H}}$ provides (i) the semantics of field access $\langle \mathtt{x.f}, \overline{\mathsf{h}} \rangle \rightarrow_{\overline{\mathsf{H}}} \bar{\mathsf{I}}$ (where $\bar{\mathsf{I}} \subseteq \overline{\mathsf{HId}}$ is the set of heap identifiers obtained by

$$\frac{\langle \mathtt{x.f} = \mathtt{y}, \overline{\mathsf{h}}\rangle \rightarrow^{\mathsf{sub}}_{\overline{\mathsf{H}}} \overline{\mathsf{h}'} \wedge \overline{applySub}(\overline{\mathsf{v}}, \mathsf{sub}) = \overline{\mathsf{v}}'}{\langle \mathtt{x.f} = \mathtt{y}, (\overline{\mathsf{h}}, \overline{\mathsf{v}})\rangle \rightarrow_{\overline{\Sigma}} (\overline{\mathsf{h}}', \overline{\mathsf{v}}')} \qquad \frac{\overline{\mathsf{v}}' = \bigsqcup_{\substack{\mathtt{i} \in \overline{\mathbb{R}}[\![\mathtt{x.f}, \overline{\mathsf{h}}]\!], \\ \overline{\mathsf{vexp}} \in \overline{\mathbb{R}}[\![\mathsf{vexp}, \overline{\mathsf{h}}]\!]}} \overline{\mathsf{v}}_1 : \langle \mathtt{i} = \overline{\mathsf{vexp}}, \overline{\mathsf{v}}\rangle \rightarrow_{\overline{\mathsf{V}}} \overline{\mathsf{v}}_1}{\langle \mathtt{x.f} = \mathsf{vexp}, (\overline{\mathsf{h}}, \overline{\mathsf{v}})\rangle \rightarrow_{\overline{\Sigma}} (\overline{\mathsf{h}}, \overline{\mathsf{v}}')}$$

$$\frac{\langle \mathtt{x} = \mathtt{rexp}, \overline{\mathsf{h}}\rangle \rightarrow^{\mathsf{sub}}_{\overline{\mathsf{H}}} \overline{\mathsf{h}'} \wedge \overline{applySub}(\overline{\mathsf{v}}, \mathsf{sub}) = \overline{\mathsf{v}}'}{\langle \mathtt{x} = \mathtt{rexp}, (\overline{\mathsf{h}}, \overline{\mathsf{v}})\rangle \rightarrow_{\overline{\Sigma}} (\overline{\mathsf{h}}', \overline{\mathsf{v}}')} \qquad \frac{\overline{\mathsf{v}}' = \bigsqcup_{\overline{\mathsf{vexp}} \in \overline{\mathbb{R}}[\![\mathsf{vexp}, \overline{\mathsf{h}}]\!]} \overline{\mathsf{v}}_1 : \langle \mathtt{x} = \overline{\mathsf{vexp}}, \overline{\mathsf{v}}\rangle \rightarrow_{\overline{\mathsf{V}}} \overline{\mathsf{v}}_1}{\langle \mathtt{x} = \mathsf{vexp}, (\overline{\mathsf{h}}, \overline{\mathsf{v}})\rangle \rightarrow_{\overline{\Sigma}} (\overline{\mathsf{h}}, \overline{\mathsf{v}}')}$$

Fig. 10. The abstract semantics $\rightarrow_{\overline{\Sigma}}$

$$\frac{}{\langle \mathtt{i} = \overline{\mathsf{vexp}}, (e_{\mathsf{Val}}, s_{\overline{\mathsf{HId}}})\rangle \rightarrow_{\mathsf{Val}'} (e_{\mathsf{Val}}, s_{\overline{\mathsf{HId}}}[\mathtt{i} \mapsto eval'(\overline{\mathsf{vexp}}, (e_{\mathsf{Val}}, s_{\overline{\mathsf{HId}}}))])}$$

$$\frac{}{\langle \mathtt{x} = \overline{\mathsf{vexp}}, (e_{\mathsf{Val}}, s_{\overline{\mathsf{HId}}})\rangle \rightarrow_{\mathsf{Val}'} (e_{\mathsf{Val}}[\mathtt{x} \mapsto eval'(\overline{\mathsf{vexp}}, (e_{\mathsf{Val}}, s_{\overline{\mathsf{HId}}}))], s_{\overline{\mathsf{HId}}})}$$

Fig. 11. The semantics $\rightarrow_{\mathsf{Val}'}$

accessing $\mathtt{x.f}$)[3], (ii) the semantics of local variable assignment $\langle \mathtt{x} = \mathtt{rexp}, \overline{\mathsf{h}}\rangle \rightarrow^{\mathsf{sub}}_{\overline{\mathsf{H}}} \overline{\mathsf{h}}'$, and (iii) the semantics of field assignment $\langle \mathtt{x.f} = \mathtt{y}, \overline{\mathsf{h}}\rangle \rightarrow^{\mathsf{sub}}_{\overline{\mathsf{H}}} \overline{\mathsf{h}}'$

The abstract semantics is defined by Figure 10. It relies on function $\overline{\mathbb{R}}$:

$\overline{\mathbb{R}} : (\mathsf{vexp} \times \overline{\mathsf{H}}) \rightarrow \wp(\overline{\mathsf{vexp}})$

$\overline{\mathbb{R}}[\![\mathtt{x}, \overline{\mathsf{h}}]\!] = \{\mathtt{x}\}$

$\overline{\mathbb{R}}[\![\mathtt{x.f}, \overline{\mathsf{h}}]\!] = \mathsf{I}$ where $\langle \mathtt{x.f}, \overline{\mathsf{h}}\rangle \rightarrow_{\overline{\mathsf{H}}} \mathsf{I}$

$$\overline{\mathbb{R}}[\![\mathsf{vexp1} < \mathsf{op} > \mathsf{vexp2}, (\overline{\mathsf{h}}, \overline{\mathsf{v}})]\!] = \bigcup_{\substack{\overline{\mathsf{vexp1}} \in \overline{\mathbb{R}}[\![\mathsf{vexp1}, \overline{\mathsf{h}}]\!] \\ \overline{\mathsf{vexp2}} \in \overline{\mathbb{R}}[\![\mathsf{vexp2}, \overline{\mathsf{h}}]\!]}} \overline{\mathsf{vexp1}} < \mathsf{op} > \overline{\mathsf{vexp2}}$$

Similarly to $\mathbb{R}$, this function replaces each field access $\mathtt{x.f}$ with the heap identifier i that represents such field access in the current heap state. Since the heap analysis may return a set of heap identifiers when accessing a field (e.g., because it may track that a local variable could point to two different abstract heap nodes), $\overline{\mathbb{R}}$ returns a set of possible value expressions in $\overline{\mathsf{vexp}}$.

Running Example. Suppose to analyze the statement $\mathtt{it} = \mathtt{l}$ at line 3 of the motivating example of Figure 1 obtaining a transition as depicted in Figure 3. The analysis materializes the node pointed by $\mathtt{it}$, and it produces a substitution as discussed in Section 4.3. The definition of the semantics $\rightarrow_{\overline{\Sigma}}$ simply applies this substitution to the value analysis after the heap semantics.

[3] For the sake of simplicity, we assume that field accesses do not produce any substitution nor they modify the abstract heap state. Anyway, this does not limit the expressiveness of our approach, since we may obtain this substitution and a new heap state by simulating this statement by assigning a field access to a local variable, and then by replacing the field access with this local variable in the expression or assignment containing $\mathtt{x.f}$.

$$
\begin{aligned}
&\mathit{eval}' : (\overline{\mathsf{vexp}} \times (\mathsf{Env}_{\mathsf{Val}} \times \mathsf{Store}_{\overline{\mathsf{HId}}})) \rightarrow \mathsf{Val}\\
&\mathit{eval}'(\mathtt{x}, (\mathsf{e}_{\mathsf{Val}}, \mathsf{s}_{\overline{\mathsf{HId}}})) = \mathsf{e}_{\mathsf{Val}}(\mathtt{x})\\
&\mathit{eval}'(\mathtt{i}, (\mathsf{e}_{\mathsf{Val}}, \mathsf{s}_{\overline{\mathsf{HId}}})) = \mathsf{s}_{\overline{\mathsf{HId}}}(\mathtt{i})\\
&\mathit{eval}'(\overline{\mathsf{vexp1}} < \mathsf{op} > \overline{\mathsf{vexp2}}, (\mathsf{e}_{\mathsf{Val}}, \mathsf{s}_{\overline{\mathsf{HId}}})) =\\
&\qquad\qquad = \mathit{eval}'(\overline{\mathtt{vexp1}}, (\mathsf{e}_{\mathsf{Val}}, \mathsf{s}_{\overline{\mathsf{HId}}})) < \mathsf{op} > \mathit{eval}'(\overline{\mathtt{vexp2}}, (\mathsf{e}_{\mathsf{Val}}, \mathsf{s}_{\overline{\mathsf{HId}}}))
\end{aligned}
$$

Fig. 12. The expression evaluation *eval′*

Soundness. Before establishing the soundness conditions of the heap and the value analyses, we need to introduce the semantics $\rightarrow_{\mathsf{Val}'}$ that defines the semantics of statements dealing with $\overline{\mathtt{vexp}}$ on $\mathsf{Env}_{\mathsf{Val}} \times \mathsf{Store}_{\overline{\mathsf{HId}}}$. This is formalized by Figure 11 and 12. Then, we assume that the semantics of the value and the heap analysis are both sound.

Proposition 3 (Soundness of the value semantics). *We assume that the semantic operator provided by the value analysis is sound. Formally,*
$\forall \overline{\mathsf{v}} \in \overline{\mathsf{V}}, \langle \mathtt{st}, \overline{\mathsf{v}} \rangle \rightarrow_{\overline{\mathsf{V}}} \overline{\mathsf{v}}', \langle \mathtt{st}, \gamma_{\overline{\mathsf{V}}}(\overline{\mathsf{v}}) \rangle \rightarrow_{\wp(\mathsf{Val}')} \mathsf{V}' \Rightarrow \mathsf{V}' \subseteq \gamma_{\overline{\mathsf{V}}}(\overline{\mathsf{v}}')$
where $\mathtt{st} \in \{\mathtt{x} = \mathtt{vexp}, \mathtt{x.f} = \mathtt{vexp}\}$.

Proposition 4 (Soundness of the heap semantics). *We assume that the semantic operators provided by the heap analysis are sound. Formally,*

- $\forall \overline{\mathsf{h}} \in \overline{\mathsf{H}}, \langle \mathtt{st}, \overline{\mathsf{h}} \rangle \rightarrow^{\mathsf{sub}}_{\overline{\mathsf{H}}} \overline{\mathsf{h}}', \langle \mathtt{st}, \pi_1(\gamma_{\overline{\mathsf{H}}}(\overline{\mathsf{h}})) \rangle \rightarrow_{\wp(\mathsf{Ref})} \mathsf{H}' \Rightarrow \mathsf{H}' \subseteq \pi_1(\gamma_{\overline{\mathsf{H}}}(\overline{\mathsf{h}}'))$ *where* $\mathtt{st} \in \{\mathtt{x} = \mathtt{rexp}, \mathtt{x.f} = \mathtt{y}\}$, *and*
- $\forall \overline{\mathsf{h}} \in \overline{\mathsf{H}}, \langle \mathtt{x.f}, \overline{\mathsf{h}} \rangle \rightarrow_{\overline{\mathsf{H}}} \mathsf{I}, \forall ((\mathsf{e}_{\mathsf{Ref}}, \mathsf{s}_{\mathsf{Ref}}), \gamma_{\overline{\mathsf{HId}}}) \in \gamma_{\overline{\mathsf{H}}}(\overline{\mathsf{h}}) \Rightarrow (\mathsf{e}_{\mathsf{Ref}}(\mathtt{x}), \mathtt{f}) \in \bigcup_{\mathsf{i} \in \mathsf{I}} \gamma_{\overline{\mathsf{HId}}}(\mathsf{i})$.

Theorem 2 (Soundness of $\rightarrow_{\overline{\Sigma}}$). *Let* $(\overline{\mathsf{h}}, \overline{\mathsf{v}}) \in \overline{\Sigma}$ *and* $\mathtt{st} \in \mathsf{St}$ *be a set of initial states and a statement, respectively. Then*
$\langle \mathtt{st}, (\overline{\mathsf{h}}, \overline{\mathsf{v}}) \rangle \rightarrow_{\overline{\Sigma}} (\overline{\mathsf{h}}', \overline{\mathsf{v}}'), \langle \mathtt{st}, \gamma_{\overline{\Sigma}}(\overline{\mathsf{h}}, \overline{\mathsf{v}}) \rangle \rightarrow_{\wp(\mathsf{Split})} \mathsf{S} \Rightarrow \mathsf{S} \subseteq \gamma_{\overline{\Sigma}}(\overline{\mathsf{h}}', \overline{\mathsf{v}}')$

4.5 Reduction

In abstract interpretation, the reduced product [10] allows two analyses to exchange information through a reduce operator. This operator is represented by a function $\rho_{\overline{\Sigma}} : \overline{\Sigma} \rightarrow \overline{\Sigma}$ such that (i) $\rho_{\overline{\Sigma}}(\overline{\sigma}) \sqsubseteq_{\overline{\Sigma}} \overline{\sigma}$, and (ii) $\gamma_{\overline{\Sigma}}(\rho_{\overline{\Sigma}}(\overline{\sigma})) = \gamma_{\overline{\Sigma}}(\overline{\sigma})$. Since the reduce operator may change what is represented by heap identifiers, it may produce a substitution whose effects are propagated to $\overline{\mathsf{V}}$ through $\overline{\mathit{applySub}}$.

For instance, a numerical analysis could discover that a list contains 2 elements, while the heap analysis was unable to track this information (e.g., it tracks the shape depicted on the left part of Figure 3). The reduce operator may refine the heap state leading to a shape similar to the one depicted in the upper right part of Figure 2.

Thanks to the genericness of the approach we adopted, we allow one to arbitrarily refine the heap state with the information tracked by the value analysis. Nevertheless, this refinement has to be defined on specific instances of value and heap analyses.

4.6 Interface of the Value and the Heap Analysis

We now summarize the interface of the value and the heap analysis. First of all, we have some standard assumptions on sound abstract domains. In particular, both the analyses form a lattice $(\langle \overline{\mathsf{V}}, \sqsubseteq_{\overline{\mathsf{V}}}, \sqcup_{\overline{\mathsf{V}}}, \sqcap_{\overline{\mathsf{V}}} \rangle$ and $\langle \overline{\mathsf{H}}, \sqsubseteq_{\overline{\mathsf{H}}} \sqcup_{\overline{\mathsf{H}}}, \sqcap_{\overline{\mathsf{H}}} \rangle)$ and a Galois connection with the concrete domain. In addition, they provide sound semantic operators to assign and read heap locations.

We have then some specific requirements on the heap analysis. In particular, $\overline{\mathsf{H}}$ provides (i) a finite set of heap identifiers $\overline{\mathsf{HId}}$, (ii) a function $\overline{heapId} : \overline{\mathsf{H}} \rightarrow \wp(\overline{\mathsf{HId}})$ that, given a state, returns the set of heap identifiers contained in that state, (iii) a coherent concretization of them (Conditions **C1-4**), and (iv) coherent substitutions of heap identifiers (Proposition 1 and 2).

These components are necessary to allow our framework to combine the heap and the value analyses, and to formally prove its soundness.

5 Instances

In this Section, we show how to plug two heap (namely, pointer and shape) analyses, and existing numerical domains in our framework. In this way, we prove that our framework is expressive enough to be applied to the most common heap and value analyses.

5.1 Pointer Analysis PA

Pointer analysis [20] has been extensively studied. One of the most known results in this field is Andersen's flow-insensitive analysis[1]. This analysis has been extended in various ways [31]. In this Section, we propose a slight modification of Might *et al.*'s analysis [26]. In particular, our analysis is flow-sensitive, field-sensitive, and it does not deal with context-sensitivity since we did not support method calls in our language. Nevertheless, we expect that our approach can be straightforwardly extended to such scenario. We adopt a standard allocation site abstraction [1,12,13,29] to approximate dynamic locations in a finite way.

Domain. Heap identifiers are represented by pairs made by (i) program labels in Lab of the `new` statements that allocate memory, and (ii) field names ($\overline{\mathsf{HId}}_{\mathsf{PA}} = \mathsf{Lab} \times \mathsf{Field}$). Since the sets of program labels and of field names are both finite, $\overline{\mathsf{HId}}_{\mathsf{PA}}$ is finite as well. The abstract environment relates each variable to a set of program labels ($\overline{\mathsf{Env}}_{\mathsf{PA}} : \mathsf{Var} \rightarrow \wp(\mathsf{Lab})$). We need a set of program labels as codomain since statically a variable could be related to references allocated at different program labels. Similarly, an abstract store relates a pair composed by a program label and a field name to a set of program labels ($\overline{\mathsf{Store}}_{\mathsf{PA}} : (\mathsf{Lab} \times \mathsf{Field}) \rightarrow \wp(\mathsf{Lab})$). Finally, abstract states are the Cartesian product of abstract environments and stores ($\overline{\Sigma}_{\mathsf{PA}} = \overline{\mathsf{Env}}_{\mathsf{PA}} \times \overline{\mathsf{Store}}_{\mathsf{PA}}$). The function $\overline{heapId}_{\mathsf{PA}} : \overline{\Sigma}_{\mathsf{PA}} \rightarrow \wp(\overline{\mathsf{HId}}_{\mathsf{PA}})$ returns the set of all the abstract memory locations in the environment and in the store. First of all, we

$$\begin{aligned}
&\mathbb{E}_{\mathsf{PA}} : (\mathtt{rexp} \times \overline{\Sigma}_{\mathsf{PA}}) \rightarrow \wp(\mathsf{Lab})\\
&\mathbb{E}_{\mathsf{PA}}[\![\mathtt{x}, (\overline{\mathsf{e}}, \overline{\mathsf{s}})]\!] = \overline{\mathsf{e}}(\mathtt{x})\\
&\mathbb{E}_{\mathsf{PA}}[\![\mathtt{x.f}, (\overline{\mathsf{e}}, \overline{\mathsf{s}})]\!] = \bigcup_{\mathsf{l}\in\overline{\mathsf{e}}(\mathtt{x})} \overline{\mathsf{s}}(\mathsf{l}, \mathtt{f})\\
&\mathbb{E}_{\mathsf{PA}}[\![\mathtt{new}\ \mathsf{C}, (\overline{\mathsf{e}}, \overline{\mathsf{s}})]\!] = \{\mathit{label}(\mathtt{new}\ \mathsf{C})\}
\end{aligned}$$

(a) The expression semantics of PA, where *label* given a statement returns its program label

$$\frac{\overline{\mathsf{s}}' = \bigsqcup_{\mathsf{l}\in\overline{\mathsf{e}}(\mathtt{x})} \overline{\mathsf{s}}[(\mathsf{l}, \mathtt{f}) \mapsto \overline{\mathsf{e}}(\mathtt{y})]}{\langle \mathtt{x.f} = \mathtt{y}, (\overline{\mathsf{e}}, \overline{\mathsf{s}})\rangle \rightarrow^{\emptyset}_{\mathsf{PA}} (\overline{\mathsf{e}}, \overline{\mathsf{s}}')}$$

$$\frac{\overline{\mathsf{e}}' = \overline{\mathsf{e}}[\mathtt{x} \mapsto \mathbb{E}_{\mathsf{PA}}[\![\mathtt{rexp}, (\overline{\mathsf{e}}, \overline{\mathsf{s}})]\!]]}{\langle \mathtt{x} = \mathtt{rexp}, (\overline{\mathsf{e}}, \overline{\mathsf{s}})\rangle \rightarrow^{\emptyset}_{\mathsf{PA}} (\overline{\mathsf{e}}', \overline{\mathsf{s}})}$$

$$\frac{}{\langle \mathtt{x.f}, (\overline{\mathsf{e}}, \overline{\mathsf{s}})\rangle \rightarrow_{\mathsf{PA}} \bigcup_{\mathsf{l}\in\overline{\mathsf{e}}(\mathtt{x})} \overline{\mathsf{s}}(\mathsf{l}, \mathtt{f})}$$

(b) The statement semantics of PA

Fig. 13. PA definitions

collect all the labels that are actually stored in the abstract state. Formally, $\overline{\mathit{label}}(\overline{\mathsf{e}}_{\mathsf{PA}}, \overline{\mathsf{s}}_{\mathsf{PA}}) = \bigcup_{\mathtt{x}\in \mathit{dom}(\overline{\mathsf{e}}_{\mathsf{PA}})} \overline{\mathsf{e}}_{\mathsf{PA}}(\mathtt{x}) \bigcup_{(\mathsf{l},\mathtt{f})\in \mathit{dom}(\overline{\mathsf{s}}_{\mathsf{PA}})} \{\mathsf{l}\} \cup \overline{\mathsf{s}}_{\mathsf{PA}}(\mathsf{l}, \mathtt{f})$. Then $\overline{\mathit{heapId}}_{\mathsf{PA}}$ is defined as follows: $\overline{\mathit{heapId}}_{\mathsf{PA}}(\overline{\mathsf{e}}_{\mathsf{PA}}, \overline{\mathsf{s}}_{\mathsf{PA}}) = \bigcup_{\mathsf{l}\in\overline{\mathit{label}}(\overline{\mathsf{e}}_{\mathsf{PA}},\overline{\mathsf{s}}_{\mathsf{PA}}), \mathtt{f}\in\overline{\mathit{fieldVal}}_{\mathsf{PA}}(\mathsf{l})} (\mathsf{l}, \mathtt{f})$ where $\overline{\mathit{fieldVal}}_{\mathsf{PA}} : \mathsf{Lab} \rightarrow \mathsf{Field}$ is a function that, given a program point that contains the assignment of a `new` statement, returns all the possible value field names of the instantiated object.

The lattice structure relies on the pointwise application of set operators on $\overline{\mathsf{Env}}_{\mathsf{PA}}$ and $\overline{\mathsf{Store}}_{\mathsf{PA}}$, and the widening operator corresponds to the least upper bound operator since the set of program labels is finite. The concretization γ_{PA} first concretizes each label to a set of concrete references that could have been created at that program label, and then builds up the environments and stores in which abstract references are replaced by the corresponding references. Formally,

$$\begin{aligned}
&\gamma_{\mathsf{PA}}(\overline{\mathsf{e}}_{\mathsf{PA}}, \overline{\mathsf{s}}_{\mathsf{PA}}) = \{((\mathsf{e}, \mathsf{s}), \gamma_{\overline{\mathsf{HId}}})\} :\\
&\qquad \gamma_{\overline{\mathsf{HId}}} \in \{[(\mathsf{l}, \mathtt{f}) \mapsto (\mathsf{r}, \mathtt{f}) : (\mathsf{l}, \mathtt{f}) \in \overline{\mathit{heapId}}_{\mathsf{PA}}(\overline{\mathsf{e}}_{\mathsf{PA}}, \overline{\mathsf{s}}_{\mathsf{PA}}) \wedge \mathsf{r} \in \mathit{allocatedRef}(\mathsf{l})]\}\\
&\qquad \mathsf{e} \in \{[\mathtt{x} \mapsto \mathsf{r} : \mathtt{x} \in \mathit{dom}(\overline{\mathsf{e}}) \wedge \mathsf{r} \in \textstyle\bigcup_{\mathsf{l}\in\overline{\mathsf{e}}(\mathtt{x})} \gamma_{\overline{\mathsf{HId}}}(\mathit{allocatedRef}(\mathsf{l}))]\}\\
&\qquad \mathsf{s} \in \{[(\mathsf{r}_1, \mathtt{f}) \mapsto \mathsf{r}_2 : (\mathsf{l}_1, \mathtt{f}) \in \mathit{dom}(\overline{\mathsf{s}}) \wedge \mathsf{r}_1 \in \mathit{allocatedRef}(\mathsf{l}_1) \wedge\\
&\qquad\qquad \mathsf{r}_2 \in \textstyle\bigcup_{\mathsf{l}_2\in\overline{\mathsf{s}}(\mathsf{l}_1,\mathtt{f})} \mathit{allocatedRef}(\mathsf{l}_2)]\}
\end{aligned}$$

where $\mathit{allocatedRef} : \mathsf{Lab} \rightarrow \wp(\mathsf{Ref})$ is a function that returns all the references allocated by a given program label.

These definitions satisfy the soundness conditions of heap identifier concretization, and in particular **C1** since $\gamma_{\overline{\mathsf{HId}}}$ always concretizes all the heap identifiers in the state, **C2** since by definition $\sqcap_{\mathsf{PA}}$ is the pointwise application of the set intersection $\cap$, **C3** since what is allocated by a label is disjoint from what can be allocated by other labels, and **C4** since what is represented by a label never changes during the computation of the abstract semantics.

Semantics. Figure 13a formalizes the abstract evaluation of expressions, while Figure 13b deals with the semantics of statements. Both these semantics are quite standard. The evaluation of expressions simply enquires the environment or the store to know the abstract references pointed by a variable or a field access, respectively. Instead, when we create a new object, this returns a singleton containing the label of the statement. Similarly, the abstract semantics of statements assigns the set of labels returned by the evaluation of expressions to the assigned variable or abstract location. The semantics always creates an empty substitution, since statements do not change how we concretize the heap identifiers, because each heap identifier represents all the concrete locations allocated by a given label. This is not touched by the abstract semantics, and empty substitutions always satisfy Proposition 1. In addition, since only empty substitutions are produced, Proposition 2 trivially holds.

5.2 TVLA-Based Shape Analysis

Shape analysis [30] is an approach to heap analysis that achieved an impressive amount of research results, and it was used to define quite precise abstractions. TVLA [22] is the first and one of the most popular shape analysis engines. Ferrara *et al.* [15] combined TVLA and value analyses in a generic way relying on substitutions. A further work [16] has plugged this combination in the framework we introduced in this paper.

Intuitively, the concrete structure of the heap is represented by shapes defined by 2-valued logic structures. These are then approximated by 3-valued logic structures. At this level, *maybe* nodes represent summary node in the heap graph. Unfortunately, TVLA names nodes in a completely arbitrary and unpredictable way. Therefore, TVAL+ augmented states with unary name predicates. Condition **C3** imposes that different names point to different nodes. Therefore, each name predicate can point only to one node, and each node has to be pointed by one name predicate. The states satisfying this property are called *normalized*. When computing the TVLA semantics, the exit state may not be normalized. [15] then defines a normalization algorithm, and [16] proves that the substitutions it produces satisfy Propositions 1 and 2.

5.3 Numerical Domains

Numerical domains are by far the most studied value abstraction, and usually they track information on local variables. Our approach introduces heap identifiers in addition to variables. On the one hand, if a heap identifier represents a definite node (that is, it is always concretized to a single concrete reference by $\gamma_{\overline{\mathsf{HId}}}$), then the value domain can treat it exactly as a variable identifier. On the other hand, if it is a summary node (that is, it concretizes to many concrete references), the value analysis has to take into account this fact to preserve the soundness of the whole analysis.

There are three major issues in this scenario. First of all, when performing an assignment to a heap identifier representing a summary node, the value analysis has to perform a weak update, that is, it has to compute the least upper bound between the state before the assignment and after it. A similar issue arises when reading from a summary node. Imagine to analyze $\texttt{a} = \texttt{l.f}; \texttt{b} = \texttt{l.next.f};$ with the linear equalities domain [21] on the heap state depicted in the upper-left corner of Figure 3. The heap analysis would evaluate both `l` and `l.next` with `u1`. Therefore, after the computation of the semantics of $\texttt{a} = \texttt{u1.f}; \texttt{b} = \texttt{u1.f};$ we would infer that $\texttt{a} == \texttt{u1.f} \wedge \texttt{b} == \texttt{u1.f}$, and then that $\texttt{a} == \texttt{b}$, that is unsound. For this reason, when considering expressions containing summary nodes, the value domain has to consider that the same heap identifiers may represent different concrete heap locations. Finally, numerical domains usually deal with *all* the program variables. Instead, in our framework we cannot know a priori the heap identifiers produced by the heap approximation during the analysis. For instance, this would lead to situations in which we have to join value states defined on different environments.

All these issues are already well-known, and Gopan *et al.* [18] extended existing numerical domains (dealing only with local variables) to *summarized dimensions* in a generic way. In particular, they require four operators from a numerical domain (add, drop, fold, and expand). Using these operators, they define a sound semantics dealing with summary nodes. Their main insight is to (i) materialize one node from the summary node, (ii) perform the abstract evaluation or assignment on this node, and (iii) merge this node with the summary node where it comes from. If on the one hand this approach is quite precise, on the other hand it could introduce several identifiers when computing the semantics, slowing down the analysis.

6 Related Work

In this Section, we briefly discuss some previous work dealing with the combination of (usually shape) heap and (usually numerical) value analyses.

McCloskey et al. [25] proposed a generic way of combining heap and numerical domains. Similarly to our work, the heap analysis splits the heap into classes of disjoint portions of the heap as we did with heap identifiers (in particular with Condition **C3**). They assume that "the set of individuals belonging to a class is not affected by an assignment", that in our framework roughly means that what is represented by heap identifiers is not modified by assignments. Instead, one of the main focuses of our approach was to allow this scenario, and substitutions are the component used to communicate to the value analysis how heap identifiers are modified. In addition, they adopt first order logic formula to allow the analyses to communicate, and they require that the user of the analysis provides the predicates that are shared among the analyses. Instead, we automatically combined heap and value analysis, while we rely on reduce operators to communicate information from the value to the heap analysis. Similarly, Gulwani and Tiwari [19] rely on the Nelson-Oppen method to combine analyzers represented in first order logic, while our approach relies on abstract interpretation-based domains. Chang and Leino [6] relied on heap analyses based on equalities to allow the numerical domain to track information over heap locations. They extended the variable identifiers usually adopted by numerical domains with aliens expressions to track information over heap locations. Intuitively, this corresponds to our notion of heap identifiers.

There are only few previous works that combined generically heap and numerical domains based on abstract interpretation. Miné's memory abstraction [27], that is part of ASTRÉE [4], is parametrized on a numerical domain, but it does not support neither summary nodes, nor dynamic allocation. Abstract cofibered domains [32] (and in a more generic way the reduced cardinal power [10]) takes as argument a numerical domain, and they could be instantiated with various heap analyses. This framework requires to manually define the functor to glue the two domains, while our work is aimed at building a generic framework that *automatically* combines the two domains, and that relies on few assumptions to ensure the soundness of the whole analysis. Recently, Chang and Rival [7] introduced a modular combination of shape and numerical abstract domains. The shape analysis relies on points-to predicate, while the numerical domain tracks information on a symbolic representation of values stored in the heap. This is slightly different from our concept of heap identifiers, that are aimed at abstracting memory locations. In addition, this work targets shape analyses based on summarization and materialization of nodes. This implies that when a node is materialized, the shape analysis needs to track a disjunctive abstraction made by a set of shapes.

Several works dealt with refining the results of a specific heap analysis with some numerical information inferred by another analysis. In this context, Magill et al. [24] refine a heap analysis based on separation logic with some numerical domains through counter-examples generated by the shape analysis. Similarly, Beyer et al. [3] combined the model checker BLAST [2] with TVLA using Counter-Example Guided Abstraction Refinement for refining the shape analysis. Bouajjani et al. [5] developed a framework to statically infer properties over programs manipulating lists containing integer numerical data. Instead, our approach is generic both on the heap and value analysis, and the information tracked by the heap analysis could be refined by the value analysis through a reduce operator as described in Section 4.5.

7 Conclusion

In this paper we presented a sound generic framework to combine heap and value analyses automatically. Our framework relies on standard operators of static analyses based on abstract interpretation. In addition, it requires that the heap analysis provides a set of heap identifiers, how these identifiers are concretized into references, and some additional soundness conditions. As far as we know, this is the first generic combination that allows the heap analysis to merge and materialize heap identifiers. We instantiated our framework to a standard pointer and shape analyses as well as to numerical domains, thus proving empirically the expressiveness of our approach.

7.1 Future Work

The most part of the theoretical ideas contained in this paper came from some practical experience the authors get with `Sample` [8,14,17,34], a generic static analyzer that combines different heap and value analyses. We are currently extending this analyzer with all the results of this paper, and to provide an interface to plug implementation of heap and numerical analyses to external users.

Acknowledgments. This work was partially supported by the SNF project "Verification-Driven Inference of Contracts".

References

1. Andersen, L.O.: Program Analysis and Specialization for the C Programming Language. PhD thesis, DIKU, University of Copenhagen (1994)
2. Beyer, D., Henzinger, T.A., Jhala, R., Majumdar, R.: The software model checker blast. STTT 9(5-6), 505–525 (2007)
3. Beyer, D., Henzinger, T.A., Théoduloz, G.: Lazy shape analysis. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 532–546. Springer, Heidelberg (2006)
4. Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: A static analyzer for large safety-critical software. In: Proceedings of PLDI 2003. ACM (2003)
5. 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)
6. Chang, B.-Y.E., Leino, K.R.M.: Abstract interpretation with alien expressions and heap structures. In: Cousot, R. (ed.) VMCAI 2005. LNCS, vol. 3385, pp. 147–163. Springer, Heidelberg (2005)
7. Chang, B.-Y.E., Rival, X.: Modular construction of shape-numeric analyzers. In: Festschrift for Dave Schmidt, EPTCS (2013)
8. Costantini, G., Ferrara, P., Cortesi, A.: Static analysis of string values. In: Qin, S., Qiu, Z. (eds.) ICFEM 2011. LNCS, vol. 6991, pp. 505–521. Springer, Heidelberg (2011)
9. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proceedings of POPL 1977. ACM (1977)
10. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: Proceedings of POPL 1979. ACM (1979)
11. Cousot, P., Cousot, R.: Abstract interpretation and application to logic programs. Journal of Logic Programming 13, 103–179 (1992)
12. Ferrara, P.: JAIL: Firewall analysis of java card by abstract interpretation. In: Proceedings of EAAI 2006 (2006)
13. Ferrara, P.: A fast and precise analysis for data race detection. In: Bytecode 2008 (2008)
14. Ferrara, P.: Static type analysis of pattern matching by abstract interpretation. In: Hatcliff, J., Zucca, E. (eds.) FMOODS/FORTE 2010, Part II. LNCS, vol. 6117, pp. 186–200. Springer, Heidelberg (2010)

15. 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)
16. Ferrara, P., Fuchs, R., Juhasz, U.: Tval+: A sound and generic combination of tvla and value analyses. Technical report, ETH Zurich (November 2013)
17. Ferrara, P., Müller, P.: Automatic inference of access permissions. In: Kuncak, V., Rybalchenko, A. (eds.) VMCAI 2012. LNCS, vol. 7148, pp. 202–218. Springer, Heidelberg (2012)
18. Gopan, D., DiMaio, F., Dor, N., Reps, T., Sagiv, M.: Numeric domains with summarized dimensions. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 512–529. Springer, Heidelberg (2004)
19. Gulwani, S., Tiwari, A.: Combining abstract interpreters. In: Proceedings of PLDI 2006. ACM (2006)
20. Hind, M.: Pointer analysis: haven't we solved this problem yet? In: Proceedings of PASTE 2001. ACM (2001)
21. Karr, M.: On affine relationships among variables of a program. Acta Informatica 6(2), 133–151 (1976)
22. 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)
23. Fähndrich, M., Logozzo, F.: Static contract checking with abstract interpretation. In: Beckert, B., Marché, C. (eds.) FoVeOOS 2010. LNCS, vol. 6528, pp. 10–30. Springer, Heidelberg (2011)
24. Magill, S., Berdine, J., Clarke, E., Cook, B.: Arithmetic strengthening for shape analysis. In: Riis Nielson, H., Filé, G. (eds.) SAS 2007. LNCS, vol. 4634, pp. 419–436. Springer, Heidelberg (2007)
25. 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)
26. Might, M., Smaragdakis, Y., Van Horn, D.: Resolving and exploiting the k-cfa paradox: illuminating functional vs. object-oriented program analysis. In: Proceedings of PLDI 2010. ACM (2010)
27. Miné, A.: Field-sensitive value analysis of embedded c programs with union types and pointer arithmetics. In: Proceedings of LCTES 2006. ACM (2006)
28. Miné, A.: The octagon abstract domain. Higher-Order and Symbolic Computation (2006)
29. Robert, V., Leroy, X.: A formally-verified alias analysis. In: Hawblitzel, C., Miller, D. (eds.) CPP 2012. LNCS, vol. 7679, pp. 11–26. Springer, Heidelberg (2012)
30. Sagiv, M., Reps, T., Wilhelm, R.: Parametric shape analysis via 3-valued logic. ACM Transactions on Programming Languages and Systems 24(3), 217–298 (2002)
31. Sridharan, M., Chandra, S., Dolby, J., Fink, S.J., Yahav, E.: Alias analysis for object-oriented programs. In: Clarke, D., Noble, J., Wrigstad, T. (eds.) Aliasing in Object-Oriented Programming. LNCS, vol. 7850, pp. 196–232. Springer, Heidelberg (2013)
32. Venet, A.: Abstract cofibered domains: Application to the alias analysis of untyped programs. In: Cousot, R., Schmidt, D.A. (eds.) SAS 1996. LNCS, vol. 1145, pp. 366–382. Springer, Heidelberg (1996)
33. Venet, A.: Towards the integration of symbolic and numerical static analysis. In: Meyer, B., Woodcock, J. (eds.) VSTTE 2005. LNCS, vol. 4171, pp. 227–236. Springer, Heidelberg (2008)
34. Zanioli, M., Ferrara, P., Cortesi, A.: SAILS: static analysis of information leakage with Sample. In: Proceedings of SAC 2012. ACM (2012)