Modular Construction of Shape-Numeric Analyzers

Bor-Yuh Evan Chang
University of Colorado Boulder
bec@cs.colorado.edu

Xavier Rival
INRIA, ENS, and CNRS
rival@di.ens.fr

The aim of static analysis is to infer invariants about programs that are precise enough to establish semantic properties, such as the absence of run-time errors. Broadly speaking, there are two major branches of static analysis for imperative programs. Pointer and *shape* analyses focus on inferring properties of pointers, dynamically-allocated memory, and recursive data structures, while *numeric* analyses seek to derive invariants on numeric values. Although simultaneous inference of shape-numeric invariants is often needed, this case is especially challenging and is not particularly well explored. Notably, simultaneous shape-numeric inference raises complex issues in the design of the static analyzer itself.

In this paper, we study the construction of such shape-numeric, static analyzers. We set up an abstract interpretation framework that allows us to reason about simultaneous shape-numeric properties by combining shape and numeric abstractions into a modular, expressive abstract domain. Such a modular structure is highly desirable to make its formalization and implementation easier to do and get correct. To achieve this, we choose a concrete semantics that can be abstracted step-by-step, while preserving a high level of expressiveness. The structure of abstract operations (i.e., transfer, join, and comparison) follows the structure of this semantics. The advantage of this construction is to divide the analyzer in modules and functors that implement abstractions of distinct features.

1 Introduction

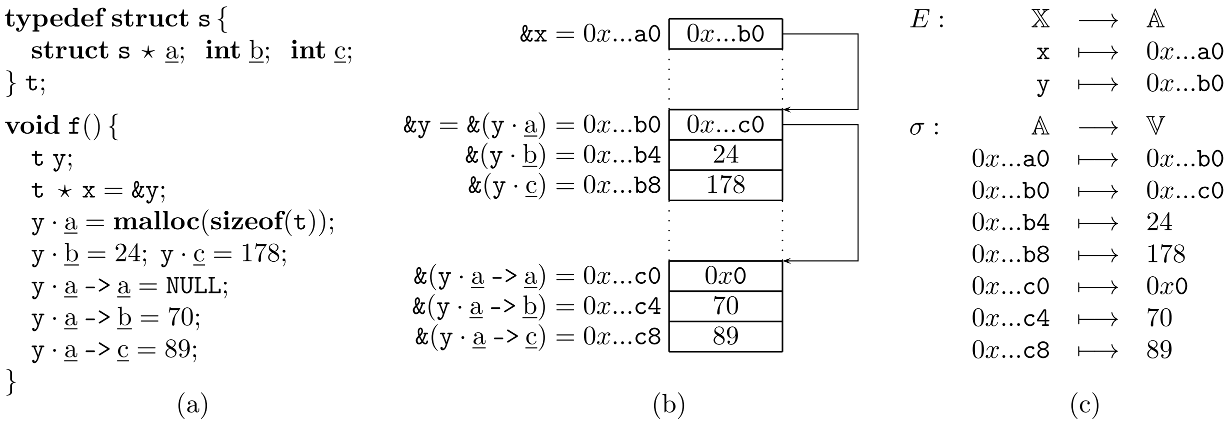
The static analysis of programs written in real-world imperative languages like C or Java are challenging because of the mix of programming features that the analyzer must handle effectively. On one hand, there are pointer values (i.e., memory addresses) that can be used to create dynamically-allocated recursive data structures. On the other hand, there are numeric data values (e.g., integer and floating-point values) that are integral to the behavior of the program. While it is desirable to use distinct abstract domains to handle such different families of properties, precise analyses require these abstract domains to *exchange* information because the pointer and numeric values are often interdependent. Setting up the structure of the implementation of such a shape-numeric analyzer can be quite difficult. While maintaining separate modules with clearly defined interfaces is a cornerstone of software engineering, such boundaries also impede the easy exchange of semantic information.

In this manuscript, we contribute a modular construction of an abstract domain [10] that layers a numeric abstraction on a shape abstraction of memory. The construction that we present is parametric in the numeric abstraction, as well as the shape abstraction. For example, the numeric abstraction may be instantiated with an abstract domain such such as polyhedra [12] or octagons [27], while the shape abstraction may be instantiated with domains such as Xisa [5,7] or TVLA [31]. Note that the focus of this paper is on describing the formalization and construction of the abstract domain. Empirical evaluation of implementations based on this construction are given elsewhere [5,7,8,22,29,36,37].

We describe our construction in four steps:

1. We define a concrete program semantics for a generic imperative programming language focusing on the concrete model of mutable memory (Section 2).

David A. Schmidt's 60th Birthday Festschrift
EPTCS 129, 2013



```
typedef struct s {
  struct s ⋆ a;  int b;  int c;
} t;
void f() {
  t y;
  t ⋆ x = &y;
  y · a = malloc(sizeof(t));
  y · b = 24;  y · c = 178;
  y · a -> a = NULL;
  y · a -> b = 70;
  y · a -> c = 89;
}
```

(a)

(b)

$$
\begin{array}{lrcl}
E: & \mathbb{X} & \longrightarrow & \mathbb{A} \\
 & \mathtt{x} & \longmapsto & 0x...\mathtt{a0} \\
 & \mathtt{y} & \longmapsto & 0x...\mathtt{b0} \\
\sigma: & \mathbb{A} & \longrightarrow & \mathbb{V} \\
 & 0x...\mathtt{a0} & \longmapsto & 0x...\mathtt{b0} \\
 & 0x...\mathtt{b0} & \longmapsto & 0x...\mathtt{c0} \\
 & 0x...\mathtt{b4} & \longmapsto & 24 \\
 & 0x...\mathtt{b8} & \longmapsto & 178 \\
 & 0x...\mathtt{c0} & \longmapsto & 0x0 \\
 & 0x...\mathtt{c4} & \longmapsto & 70 \\
 & 0x...\mathtt{c8} & \longmapsto & 89
\end{array}
$$

(c)

Figure 1: A concrete memory state consists of an environment E and a store σ shown in (c). This example state corresponds to the informal box diagram shown in (b) and a state at the return point of the C-procedure `f` in (a).

2. We describe a step-by-step abstraction of program states as a cofibered construction of a numeric abstraction layer on top of a shape abstraction layer (Section 3). In particular, we characterize a shape abstraction as a combination of an *exact* abstraction of memory cells along with a *summarization* operation. Then, we describe how a value abstraction can be applied both globally on *materialized* memory locations and locally within summarized regions.
3. We detail the abstract operators necessary to implement an abstract program semantics in terms of interfaces that a shape abstraction and a value abstraction must implement (Section 4).
4. We overview a modular construction of a shape-numeric static analyzer based on our abstract operators (Section 5).

2 A concrete semantics

We first define a concrete program semantics for a generic imperative programming language.

2.1 Concrete memory states

We define a "bare metal" model of machine memory. A *concrete store* is a partial function $\sigma \in \mathbb{H} = \mathbb{A} \rightharpoonup_{\text{fin}} \mathbb{V}$ from addresses to values. An address $a \in \mathbb{A}$ is also considered a value $v \in \mathbb{V}$, that is, we assume that $\mathbb{A} \subseteq \mathbb{V}$. For simplicity, we assume that all cells of any store σ have the same size (i.e., word-sized) and that all addresses are aligned (i.e., word-aligned). For example, we can imagine a standard 32-bit architecture where all values are 4-bytes and all addresses are 4-byte–aligned. We write for $\mathbf{dom}(\sigma)$ the set of addresses at which σ is defined, and we let $\sigma[a \leftarrow v]$ denote the heap obtained after updating the cell at address a with value v. A *concrete environment* $E \in \mathbb{E} = \mathbb{X} \rightarrow \mathbb{A}$ maps program variables to their addresses. That is, we consider all program variables as mutable cells in the concrete store—the concrete environment E indicates where each variable is allocated. A *concrete memory state* m simply pairs a concrete environment and a concrete store: (E, σ). Thus, the set of memory states $\mathbb{M} = \mathbb{E} \times \mathbb{H}$ is the product of the set of concrete environments and the set of concrete stores.

Figure 1(c) shows an example concrete memory state at the return point of the procedure `f` in (a). The environment E has two bindings for the variables `x` and `y` that are in scope. For concreteness, we show the concrete store for this example laid out using 32-bit addresses and a C-style layout for **struct** `s`. The figure shown in (b) shows the concrete store as an informal box diagram.

Related work and discussion. Observe that we do not make the distinction between stack and heap space in a concrete store σ (as in a C-style model), nor have we partitioned a heap on field names (as in Java-style model). We have intentionally chosen this rather low-level definition of concrete memory states—essentially an assembly-level model of memory—and leave any abstraction to the definition of abstract memory states. An advantage of this approach is the ability to use a common concrete model for combining abstractions that make different choices about the details they wish to expose or hide [22]. For example, Laviron et al. [22] defines an abstract domain that treats precisely C-style aggregates: both **struct**s and **union**s with sized-fields and pointer arithmetic. Another abstract domain [36] abstracts nested structures using a hierarchical abstraction. Rival and Chang [29] defines an abstraction that simultaneously summarizes the stack of activation records and the heap data structures (with a slightly extended notion of concrete environments), which is useful for analyzing recursive procedures.

2.2 Concrete program semantics

For the most part, we can be agnostic about the particulars of the imperative programming language of interest. To separate concerns between abstracting memory and control points on which abstract interpretation collects, all we assume is that a *concrete execution state* consists of a *control state* and a concrete memory state. A shape-numeric abstract domain as we define in Section 3 abstracts the concrete memory state component.

Definition 1 (Execution states)**.** An *execution state* $s \in \mathbb{S}$ consists of a triple (ℓ, E, σ) where $\ell \in \mathbb{L}$ is a control state, $E \in \mathbb{E}$ is an concrete environment, and $\sigma \in \mathbb{H}$ is a concrete store. The memory component of an execution state is the pair $(E, \sigma) \in \mathbb{M}$.

Thus, the set of execution states $\mathbb{S} = \mathbb{L} \times \mathbb{E} \times \mathbb{H} \equiv \mathbb{L} \times \mathbb{M}$. A *program execution* is described by a *finite trace*, that is, a finite sequence of states $\langle s_0, \ldots, s_n \rangle$. We let $\mathbb{T} = \mathbb{S}^\star$ denote the set of finite traces over $\mathbb{S}$.

To make our examples more concrete, we consider a C-like programming language whose syntax is shown in Figure 2. A location expression *loc* names a memory cell, which can be a program variable x, a field offset from another memory location $loc_1 \cdot \underline{\mathrm{f}}$, or the memory location named by a pointer value $\star exp$. We write $\underline{\mathrm{f}} \in \mathbb{F}$ for a field name and implicitly read any field as an offset, that is, we write $a + \underline{\mathrm{f}}$ for the address $a' \in \mathbb{A}$ obtained by offsetting an address a with field $\underline{\mathrm{f}}$. To emphasize that we mean C-style field offset as opposed to Java-style field dereference, we write $\mathtt{x} \cdot \underline{\mathrm{f}}$ for what is normally written as $\mathtt{x}.\underline{\mathrm{f}}$ in C. As in C, we write *exp* `->`$\underline{\mathrm{f}}$ for Java-style field dereference, which is a shorthand for $(\star exp) \cdot \underline{\mathrm{f}}$. An expression *exp* can be a memory location expression *loc*, an address of a memory location `&`*loc*, or any value literal v, some other n-ary operator $\oplus(\overline{exp})$. Like in C, a memory location expression *loc* used as an expression (i.e., "r-value") refers to the contents of the named memory cell, while the `&`*loc* converts the location's address (i.e., "l-value") into a pointer "r-value." We leave the value literals v (e.g., `1`) and expression operators $\oplus$ (e.g., !, $+$, $==$) unspecified.

An operational semantics: Given a program p, we assume its execution is described by a transition relation $\rightarrow_p \subseteq \mathbb{S} \times \mathbb{S}$. This relation defines a small-step operational semantics, which can be defined as a structured operational semantics judgment $s \rightarrow_p s'$. Such a definition is completely standard for our language, so we do not detail it here.

$$\begin{array}{rcll}
loc\ (\in \mathscr{L}_{\mathbb{X}}) & ::= & x & (x \in \mathbb{X}) \\
 & | & loc_1 \cdot \underline{\mathrm{f}} & (loc_1 \in \mathscr{L}_{\mathbb{X}}; \underline{\mathrm{f}} \in \mathbb{F}) \\
 & | & \star exp & (exp \in \mathscr{E}_{\mathbb{X}})
\end{array}
\qquad
\begin{array}{rcll}
exp\ (\in \mathscr{E}_{\mathbb{X}}) & ::= & loc & (loc \in \mathscr{L}_{\mathbb{X}}) \\
 & | & \&loc & (loc \in \mathscr{L}_{\mathbb{X}}) \\
 & | & v & (v \in \mathbb{V}) \\
 & | & \oplus(\overline{exp}) & (\overline{exp} \in \mathscr{E}_{\mathbb{X}}) \\
\oplus & ::= & \cdots &
\end{array}$$

$$\begin{array}{rclll}
p\ (\in \mathscr{P}_{\mathbb{X}}) & ::= & loc = exp & (loc \in \mathscr{L}_{\mathbb{X}}; exp \in \mathscr{E}_{\mathbb{X}}) & \text{assignment} \\
 & | & loc = \mathbf{malloc}(\{\underline{\mathrm{f}}_1, \ldots, \underline{\mathrm{f}}_n\}) & (loc \in \mathscr{L}_{\mathbb{X}}; [\underline{\mathrm{f}}_1, \ldots, \underline{\mathrm{f}}_n] \in \mathbb{F}^*) & \text{memory allocation} \\
 & | & \mathbf{free}(loc) & (loc \in \mathscr{L}_{\mathbb{X}}) & \text{memory deallocation} \\
 & | & p_1 ; p_2 & (p_1, p_2 \in \mathscr{P}_{\mathbb{X}}) & \text{sequence} \\
 & | & \mathbf{if}\ (exp)\ p_1\ \mathbf{else}\ p_1 & (exp \in \mathscr{E}_{\mathbb{X}}; p_1, p_2 \in \mathscr{P}_{\mathbb{X}}) & \text{condition test} \\
 & | & \mathbf{while}\ (exp)\ p_1 & (exp \in \mathscr{E}_{\mathbb{X}}; p_1 \in \mathscr{P}_{\mathbb{X}}) & \text{loop}
\end{array}$$

Figure 2: Abstract syntax for a C-like imperative programming language. A program p consists of assignment, dynamic memory allocation and deallocation, sequences, condition tests, and loops. An assignment is specified by a location expression *loc* that names a memory cell to update and an expression *exp* that is evaluated to yield the new contents for the cell. For simplicity, we specify allocation with a list of field names (i.e., $\mathbf{malloc}(\{\underline{\mathrm{f}}_1, \ldots, \underline{\mathrm{f}}_n\})$).

DoAssignment

$$\frac{}{(\ell_{\mathrm{pre}}, E, \sigma) \to_{loc=exp} (\ell_{\mathrm{post}}, E, \sigma[\mathscr{L}[\![loc]\!](E,\sigma) \leftarrow \mathscr{E}[\![exp]\!](E,\sigma)])}$$

$$\begin{array}{rclrcl}
\mathscr{L}[\![x]\!](E,\sigma) & \stackrel{\text{def}}{=} & E(x) & \mathscr{L}[\![\star exp]\!] & \stackrel{\text{def}}{=} & \mathscr{E}[\![exp]\!] \\
\mathscr{E}[\![loc]\!](E,\sigma) & \stackrel{\text{def}}{=} & \sigma \circ \mathscr{L}[\![loc]\!](E,\sigma) & \mathscr{E}[\![\&loc]\!] & \stackrel{\text{def}}{=} & \mathscr{L}[\![loc]\!]
\end{array}$$

$$\mathscr{L}[\![loc \cdot \underline{\mathrm{f}}]\!](E,\sigma) \stackrel{\text{def}}{=} \mathscr{L}[\![loc]\!](E,\sigma) + \underline{\mathrm{f}}$$

Figure 3: A small-step operational semantics for programs.

As an example rule, consider the case for an assignment $loc = exp$ where ℓ_{pre} and ℓ_{post} are the control points before and after the assignment, respectively. We assume that the semantics of a location expression $\mathscr{L}[\![loc]\!]$ is a function from memory states to addresses $\mathbb{M} \to \mathbb{A}$ and that the semantics of an expression $\mathscr{E}[\![exp]\!]$ is a function from memory states to values $\mathbb{M} \to \mathbb{V}$. Then, the transition relation for assignment simply updates the input store σ at the address given by *loc* with the value given by *exp* as shown in Figure 3. The evaluation of locations *loc* and expressions *exp*, that is, $\mathscr{L}[\![loc]\!](E,\sigma)$ and $\mathscr{E}[\![exp]\!](E,\sigma)$, respectively, can be defined by induction on their structure. The environment E is used to lookup the allocated address for program variables in $\mathscr{L}[\![x]\!]$. The value for a memory location $\mathscr{E}[\![loc]\!]$ is obtained by looking up the contents in the store σ. Dereference $\star exp$ and $\&loc$ mediate between address and value evaluation, while field offset $loc \cdot \underline{\mathrm{f}}$ is simply an address computation. The evaluation of the remaining expression forms is completely standard.

Example 1 (Evaluating an assignment)**.** Using the concrete memory state (E,σ) shown in Figure 1, the evaluation of the assignment $\mathtt{x} \,\texttt{->}\, \underline{\mathtt{a}} \,\texttt{->}\, \underline{\mathtt{b}} = \mathtt{y} \cdot \underline{\mathtt{c}}$ proceeds as follows. First, the right-hand side gets evaluated by noting that $E(\mathtt{y}) = 0x...\mathtt{b0}$ and following

$$\mathscr{E}[\![\mathtt{y} \cdot \underline{\mathtt{c}}]\!](E,\sigma) = \sigma(\mathscr{L}[\![\mathtt{y} \cdot \underline{\mathtt{c}}]\!](E,\sigma)) = \sigma(\mathscr{L}[\![\mathtt{y}]\!](E,\sigma) + \underline{\mathtt{c}}) = \sigma(E(\mathtt{y}) + \underline{\mathtt{c}}) = \sigma(0x...\mathtt{b8}) = 178\ .$$

Second, the left-hand side gets evaluated by noting that $E(\mathtt{x}) = 0x...\mathtt{a0}$ and then following the location evaluation $\mathscr{L}[\![\mathtt{x} \,\texttt{->}\, \underline{\mathtt{a}} \,\texttt{->}\, \underline{\mathtt{b}}]\!](E,\sigma) = \sigma(\sigma(0x...\mathtt{a0}) + \underline{\mathtt{a}}) + \underline{\mathtt{b}}) = \sigma(0x...\mathtt{b0} + \underline{\mathtt{a}}) + \underline{\mathtt{b}} = 0x...\mathtt{c0} + \underline{\mathtt{b}} = 0x...\mathtt{c4}$. Finally, the store is updated at address $0x...\mathtt{c4}$ with the value 178 with $\sigma[0x...\mathtt{c4} \leftarrow 178]$.

Concrete program semantical definitions: Several notions of program semantics can be used as a basis for static analysis, which each depend on the desired properties and the kinds of invariants needed to establish them. A semantical definition expressed as the least fixed-point of a continuous function F over a concrete, complete lattice is particularly well-suited to the design of abstract interpreters [10]. Following this analysis design methodology, an abstract interpretation consists of (1) choosing an abstraction of the concrete lattice (Section 3), (2) designing abstract operators that over-approximate the effect of the transition relation $\rightarrow_p$ and concrete joins $\cup$ (Section 4), and (3) applying abstract operators to over-approximate F using widening (Section 5).

Definition 2 (A concrete domain)**.** Let us fix a form for our concrete domains $\mathbb{D}$ to be the powerset of some set of concrete objects $\mathbb{O}$, that is, let $\mathbb{D} = \mathscr{P}(\mathbb{O})$. Domain $\mathbb{D}$ form a complete lattice with subset containment $\subseteq$ as the partial order. Hence, concrete joins are simply set union $\cup$.

For a program p, let ℓ_{pre} be its entry point (i.e., its initial control state). A standard definition of interest is the set of reachable states, which is sufficient for reasoning about safety properties.

Example 2 (Reachable states)**.** We write $[\![p]\!]_{\mathbf{r}}$ for the set of reachable states of program p, that is,

$$[\![p]\!]_{\mathbf{r}} \stackrel{\text{def}}{=} \{s \mid (\ell_{\mathrm{pre}}, E, \sigma) \rightarrow_p^\star s \text{ for some } E \in \mathbb{E} \text{ and } \sigma \in \mathbb{H}\}$$

where $\rightarrow_p^\star$ is the reflexive-transitive closure of the single-step transition relation $\rightarrow$. Alternatively, $[\![p]\!]_{\mathbf{r}}$ can be defined as $\mathbf{lfp}\, F_{\mathbf{r}}$, the least-fixed point of $F_{\mathbf{r}}$, where $F_{\mathbf{r}} : \mathscr{P}(\mathbb{S}) \rightarrow \mathscr{P}(\mathbb{S})$ is as follows:

$$F_{\mathbf{r}}(S) \stackrel{\text{def}}{=} \{(\ell_{\mathrm{pre}}, E, \sigma) \mid E \in \mathbb{E} \text{ and } \sigma \in \mathbb{H}\} \cup \{s' \mid s \in S \text{ and } s \rightarrow_p s' \text{ for some } s' \in \mathbb{S}\}\ .$$

Note that we have let the concrete objects $\mathbb{O}$ be the execution states $\mathbb{S}$ in this example.

We can also describe the reachable states *denotationally* [34]—$[\![p]\!]_{\mathbf{d}}(E,\sigma) \stackrel{\text{def}}{=} \{s \mid (\ell_{\mathrm{pre}}, E, \sigma) \rightarrow_p^\star s\}$—that enables a compositional way to reason about programs. Here, we let the set of concrete objects be functions from memory states to sets of states (i.e., $\mathbb{M} \rightarrow \mathscr{P}(\mathbb{S})$).

Related work and discussion. For additional precision or for richer properties, it may be critical to retain some information about the history of program executions (i.e., how a state can be reached) [30]. In this case, we might choose a *trace semantics* as a concrete semantics where the concrete objects $\mathbb{O}$ are chosen to be traces $\mathbb{T}$. For instance, the finite prefix traces semantics is defined by $[\![p]\!]_{\mathbf{t}} \stackrel{\text{def}}{=} \{\langle s_0, \ldots, s_n\rangle \mid s_0 \colon (\ell_{\mathrm{pre}}, E_0, \sigma_0) \text{ and } s_i \rightarrow_p s_{i+1} \text{ for some } E_0 \in \mathbb{E}, \sigma_0 \in \mathbb{H} \text{ and for all } 0 \leq i < n\}$. Or we may to choose to define a trace semantics denotationally $[\![p]\!]_{\mathbf{dh}} : \mathbb{M} \rightarrow \mathscr{P}(\mathbb{T})$ that maps input memory states into traces starting from them.

In this section, we have left the definition of a control state essentially abstract. A control state is simply a member of a set of labels on which an interpreter visits. In the intraprocedural setting, the control state is usually a point in the program text corresponding to a program counter. Since the set of program points is finite, the control state can be left unabstracted yielding a flow-sensitive analysis. Meanwhile, richer notions of control states are often needed for interprocedural analysis [26, 35].

3 Abstraction of memory states

In this section, we discuss the abstraction of memory states, including environments and stores, as well as the values stored in them. A shape abstraction typically abstracts entire stores but only the pointer values (i.e., addresses) in them. In contrast, a numeric abstraction is typically applied only to the data values stored in program variables (i.e., the part of the store containing the global and local variables). We defer the abstraction of program executions to Section 5.

Following the abstract interpretation framework [10], an *abstraction* or *abstract domain* is a set of abstract properties $\mathbb{D}^\sharp$ together with a concretization function and sound abstract operators.

Definition 3 (Concretization)**.** A *concretization function* $\gamma : \mathbb{D}^\sharp \to \mathbb{D}$ defines the meaning of $\mathbb{D}^\sharp$ in terms of a concrete domain $\mathbb{D} = \mathscr{P}(\mathbb{O})$ for some set of concrete objects $\mathbb{O}$. An abstract inclusion $d_1^\sharp \sqsubseteq d_2^\sharp$ for abstract elements $d_1^\sharp, d_2^\sharp \in \mathbb{D}^\sharp$ should be sound with respect to concrete inclusion: $\gamma(d_1^\sharp) \subseteq \gamma(d_2^\sharp)$, and γ should be monotone. For each concrete operation f, we expect a sound abstract counterpart $f^\sharp$; for example, an abstract operation $f^\sharp : \mathbb{D}^\sharp \to \mathbb{D}^\sharp$ is sound with respect to a concrete operation $f : \mathbb{D} \to \mathbb{D}$ if and only if $\gamma \circ f(d^\sharp) \subseteq \gamma \circ f^\sharp(d^\sharp)$ for all $d^\sharp \in \mathbb{D}^\sharp$.

In this section, we focus on the abstract domains and concretization functions, while the construction of abstract operations are detailed in Section 4.

3.1 An exact store abstraction based on separating shape graphs

An *abstract heap* $\sigma^\sharp \in \mathbb{H}^\sharp$ should over-approximate a set of concrete heaps with a compact representation. This set of abstract heaps $\mathbb{H}^\sharp$ form the *domain of abstract heaps* (or the *shape abstract domain*). For simplicity, we first consider an *exact abstraction* of heaps with no unbounded dynamic data structures. That is, such an abstraction explicitly enumerates a finite number of memory cells and performs no summarization. Summarization is considered in Section 3.3.

A heap can be viewed as a set of *disjoint* cells (cf., Figure 1). At the abstract level, it is convenient to make disjointness explicit and describe disjoint cells independently. Thus, we write $\sigma_0^\sharp * \sigma_1^\sharp$ for the abstract heap element that denotes all that can be partitioned into a sub-heap satisfying $\sigma_0^\sharp$ and another disjoint sub-heap satisfying $\sigma_1^\sharp$. This observation about disjointness underlies separation logic [28] and thus we borrow the separating conjunction operator $*$ from there. An individual cell is described by an *exact points-to* predicate of the form $\alpha \cdot \underline{f} \mapsto \beta$ where α, β are symbolic variables (or, abstract values) drawn from a set $\mathbb{V}^\sharp$. The symbolic variable α denotes an address, while β represents the contents at the memory cell with address $\alpha \cdot \underline{f}$ (i.e., α offset by a field $\underline{f}$). An exact heap abstraction is thus a separating conjunction of a set of exact points-to predicates.

Such abstract heap predicates can be represented using *separating shape graphs* [8,22] where nodes are symbolic variables and edges represent heap predicates. An exact points-to predicate $\alpha \cdot \underline{f} \mapsto \beta$ is denoted by an edge from node α to node β with a label for the field offset $\underline{f}$. For example, $\beta_{\underline{a}}$ denotes the *value* corresponding to the C expression $\mathrm{y} \cdot \underline{a}$.

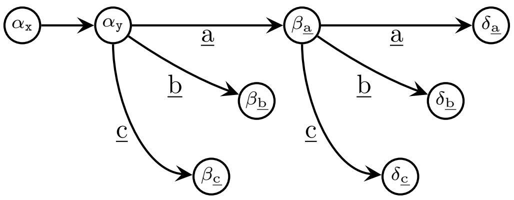
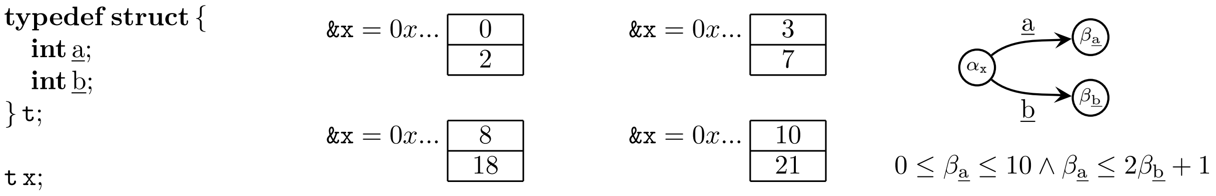


Figure 4: separating shape graph abstraction of σ in Figure 1. Symbolics α_x and α_y denote the *address* of x and y, respectively.

The concretization $\gamma_{\mathbb{H}}$ of a separating shape graph must account for symbolic variables that denote some concrete values, so it also must yield an *instantiation* or a *valuation* $\nu : \mathbb{V}^\sharp \to \mathbb{V}$. Thus, this concretization has type $\gamma_{\mathbb{H}} : \mathbb{H}^\sharp \to \mathscr{P}(\mathbb{H} \times (\mathbb{V}^\sharp \to \mathbb{V}))$ and is defined as follows (by induction on the structure $\sigma^\sharp$):

$$\gamma_{\mathbb{H}}(\alpha \cdot \underline{f} \mapsto \beta) \stackrel{\text{def}}{=} \{([\nu(\alpha) + \underline{f} \mapsto \nu(\beta)], \nu) \mid \nu \in \mathbb{V}^\sharp \to \mathbb{V}\}$$
$$\gamma_{\mathbb{H}}(\sigma_0^\sharp * \sigma_1^\sharp) \stackrel{\text{def}}{=} \{(\sigma_0 \uplus \sigma_1, \nu) \mid (\sigma_0, \nu) \in \gamma_{\mathbb{H}}(\sigma_0^\sharp) \text{ and } (\sigma_1, \nu) \in \gamma_{\mathbb{H}}(\sigma_1^\sharp) \text{ and } \mathbf{dom}(\sigma_0) \cap \mathbf{dom}(\sigma_1) = \emptyset\}\ .$$

That is, an exact points-to predicate corresponds to a single cell concrete store under a valuation ν, and a separating conjunction of abstract heaps is a concrete store composed of disjoint sub-stores that



```
typedef struct {
  int a;
  int b;
} t;

t x;
```

&x = 0x... 0 2
&x = 0x... 3 7
&x = 0x... 8 18
&x = 0x... 10 21
a
b
α_x
β_a
β_b
$0 \leq \beta_{\underline{a}} \leq 10 \wedge \beta_{\underline{a}} \leq 2\beta_{\underline{b}} + 1$

Figure 5: An example separating shape graph enriched with a numeric constraint (right) with four concrete instances (center) for the C type declaration (left).

are individually abstracted by the conjuncts under the same instantiation (as in separation logic [28]). Symbolic variables can be viewed as existentially-quantified variables that are bound at the top-level of the abstraction. The valuation makes this explicit and thus is a bit similar to a concrete environment E.

Related work and discussion. Separating conjunction manifests itself in separating shape graphs as simply distinct edges. In other words, distinct edges denote disjoint heap regions. Separating shape graphs are visually quite similar to classical shape and points-to graphs [9, 31] but are actually quite different semantically. In classical shape and points-to graphs, the nodes represent memory cells, and typically, a node corresponds to one-or-more concrete cells. Distinct nodes represent disjoint memory memory regions, and edges express variants of may or must points-to relations between two sets of cells. In contrast, it is the edges in separating shape graphs that correspond to disjoint memory cells, while the nodes simply represent values. We have found two main advantages of this approach. First, because there is no *a priori* requirement that two nodes be distinct values, we do not need to case split simply to speak about the contents of cells (e.g., consider two pointer variables x and y and representing to which objects they point; a classic shape graph must consider two cases where x and y are aliases or not, while a separating shape graph does not). Limiting case splits is critical to getting good analysis performance [5]. Second, a separating shape graph is agnostic to the type of values that nodes represent. Nodes may represent addresses, but they can just as easily represent non-address values, such as integer, Boolean, or floating-point values. We take advantage of this observation to interface with numeric abstract domains [7], which we discuss further next in Section 3.2.

3.2 Enriching shapes with a numeric abstraction

From Section 3.1, we have an exact heap abstraction based on a separating shape graph with a finite number of exact points-to edges. Intuitively, this abstraction is quite weak, as we have simply enumerated the memory cells of interest. We have, however, given names to all values—both addresses and contents—of potential interest. Here, we enrich the abstraction with information about the values contained in data structures, not just the pointer shape. We focus on *scalar* numeric values, such as integers or floating-point values, but other types of values could be handled similarly. A separating shape graph defines a set of symbolic variables corresponding to values, so we can abstract the values those symbolic variables represent. First, we consider a simple example, shown in Figure 5. In Figure 5, we show four concrete stores such that $0 \leq \mathtt{x} \cdot \underline{a} \leq 10$ and $\mathtt{x} \cdot \underline{a} \leq 2(\mathtt{x} \cdot \underline{b}) + 1$. The separating shape graph on the right clearly abstracts the shape of the four stores (i.e., two fields $\underline{a}$ and $\underline{b}$ off a **struct** at variable x). The symbolic variables $\beta_{\underline{a}}$ and $\beta_{\underline{b}}$ represent the contents of cells $x \cdot \underline{a}$ and $x \cdot \underline{b}$, respectively, so the numeric property specified above can expressed simply by using a logical formula involving $\beta_{\underline{a}}$ and $\beta_{\underline{b}}$ (as shown).

In general, a separating shape graph $\sigma^\sharp$ is defined over a set of symbolic variables $\mathbb{V}^\sharp[\sigma^\sharp]$ where $\mathbb{V}^\sharp[\sigma^\sharp] \subseteq \mathbb{V}^\sharp$. The properties of the values stored in heaps described by $\sigma^\sharp$ can be characterized by

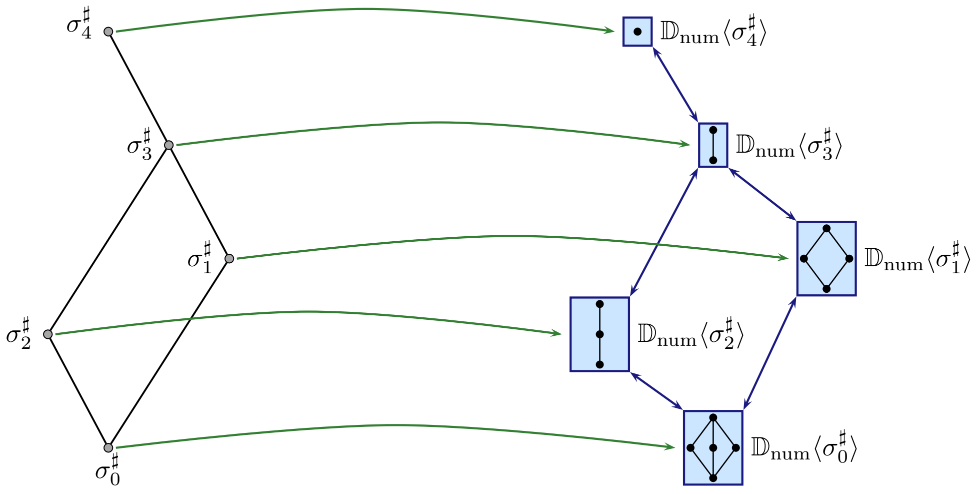

$\sigma_4^\sharp$
$\sigma_3^\sharp$
$\sigma_1^\sharp$
$\sigma_2^\sharp$
$\sigma_0^\sharp$
$\mathbb{D}_{\mathrm{num}}\langle\sigma_4^\sharp\rangle$
$\mathbb{D}_{\mathrm{num}}\langle\sigma_3^\sharp\rangle$
$\mathbb{D}_{\mathrm{num}}\langle\sigma_1^\sharp\rangle$
$\mathbb{D}_{\mathrm{num}}\langle\sigma_2^\sharp\rangle$
$\mathbb{D}_{\mathrm{num}}\langle\sigma_0^\sharp\rangle$

Figure 6: The combined shape-numeric abstract domain is a cofibered layering of a numeric abstract domain on a shape abstract domain.

logical formulas over $\mathbb{V}^\sharp[\sigma^\sharp]$. Such logical formulas expressing numeric properties can be represented using a numeric abstract domain $\mathbb{D}_{\mathrm{num}}\langle\mathbb{V}^\sharp[\sigma^\sharp]\rangle$ that abstracts functions from $\mathbb{V}^\sharp[\sigma^\sharp]$ to $\mathbb{V}$, that is, it comes with concretization function parametrized by a set of symbolic values $\mathbb{V}^\sharp[\sigma^\sharp]$ of the following type: $\gamma_{\mathrm{num}}\langle\mathbb{V}^\sharp[\sigma^\sharp]\rangle : \mathbb{D}_{\mathrm{num}}\langle\mathbb{V}^\sharp[\sigma^\sharp]\rangle \to \mathscr{P}(\mathbb{V}^\sharp[\sigma^\sharp] \to \mathbb{V})$. For example, the numeric property mentioned in Figure 5 could be expressed using the convex polyhedra abstract domain [12]. As a shape graph concretizes into a set of pairs composed of a heap σ and a valuation $\nu : \mathbb{V}^\sharp[\sigma^\sharp] \to \mathbb{V}$, such numeric constraints simply restrict the set of admissible valuations.

The need to combine a shape graph with a numeric constraint suggests using a product abstraction [11] of a shape abstract domain $\mathbb{H}^\sharp$ and a numeric abstract domain $\mathbb{D}_{\mathrm{num}}\langle-\rangle$. However, note that the numeric abstract domain that needs to be used depends on the separating shape graph, as the set of dimensions is equal to the set of nodes in the separating shape graph. Therefore, the conventional notion of a *symmetric* reduced product does not apply here. Instead, we use a different construction known as a *cofibered abstract domain* [38] (in reference with the categorical notion underlying this construction).

Definition 4 (Combined shape-numeric abstract domain)**.** Given a shape domain $\mathbb{H}^\sharp$ and a numeric domain $\mathbb{D}_{\mathrm{num}}\langle-\rangle$ parametrized by a set of symbolic variables. We let $\mathbb{N}^\sharp$ denote the set of numeric abstract values corresponding to any shape graph (i.e., $\mathbb{N}^\sharp \stackrel{\mathrm{def}}{=} \bigcup\{\mathbb{D}_{\mathrm{num}}\langle V\rangle \mid V \subseteq \mathbb{V}^\sharp\}$), and we define the *combined shape-numeric abstract domain* $\mathbb{H}^\sharp \rightrightarrows \mathbb{N}^\sharp$ and its concretization $\gamma_{\mathbb{H}^\sharp\rightrightarrows\mathbb{N}^\sharp} : (\mathbb{H}^\sharp \rightrightarrows \mathbb{N}^\sharp) \to \mathscr{P}(\mathbb{H} \times (\mathbb{V}^\sharp \to \mathbb{V}))$ as follows:

$$\begin{array}{rcl}
\mathbb{H}^\sharp \rightrightarrows \mathbb{N}^\sharp & \stackrel{\mathrm{def}}{=} & \{(\sigma^\sharp, \nu^\sharp) \mid \sigma^\sharp \in \mathbb{H}^\sharp \text{ and } \nu^\sharp \in \mathbb{D}_{\mathrm{num}}\langle\mathbb{V}^\sharp[\sigma^\sharp]\rangle\} \\
\gamma_{\mathbb{H}^\sharp\rightrightarrows\mathbb{N}^\sharp}(\sigma^\sharp, \nu^\sharp) & \stackrel{\mathrm{def}}{=} & \{(\sigma, \nu) \mid (\sigma, \nu) \in \gamma_{\mathbb{H}}(\sigma^\sharp) \text{ and } \nu \in \gamma_{\mathrm{num}}\langle\mathbb{V}^\sharp[\sigma^\sharp]\rangle(\nu^\sharp)\}
\end{array}$$

This product is clearly *asymmetric*, as the left member defines the abstract lattice to which the right member belongs. We illustrate this structure in Figure 6. The left part depicts the lattice of abstract heaps, while the right part illustrates a lattice of numeric lattices. Each element of the lattice of lattices is an instance of the numeric abstract domain over the symbolic variables defined by the abstract heap, that is, it is the image of the function $\sigma^\sharp \mapsto \mathbb{D}_{\mathrm{num}}\langle\mathbb{V}^\sharp[\sigma^\sharp]\rangle$.

This dependence is not simply theoretical but has practical implications on both the representation of abstract values and the design of abstract operations in the combined abstract domain. For instance,

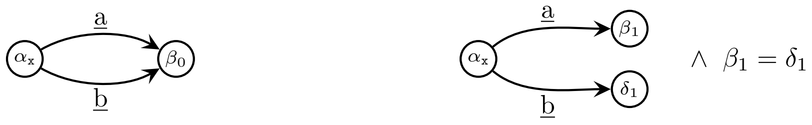


Figure 7: Two abstractions drawn from the combined abstract domain $\mathbb{H}^\sharp \rightrightarrows \mathbb{N}^\sharp$ that have equivalent concretizations but with non-isomorphic sets of symbolic variables.

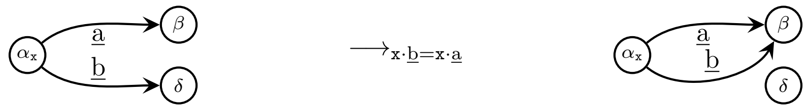

α_x
a
b
β
δ
$\longrightarrow_{\mathtt{x}\cdot\underline{\mathrm{b}}=\mathtt{x}\cdot\underline{\mathrm{a}}}$
α_x
a
b
β
δ

Figure 8: Applying the transfer function for an assignment on a separating shape graph that changes the set of "live" symbolic variables.

Figure 7 shows two separating shape graphs together with numerical invariants that represent the same set of concrete stores even though they use two different sets of symbolic variables (even up to α-renaming). Both of these combined shape-numeric abstract domain elements represent a store with two fields $\mathtt{x} \cdot \underline{\mathrm{a}}$ and $\mathtt{x} \cdot \underline{\mathrm{b}}$ such that $\mathtt{x} \cdot \underline{\mathrm{a}} = \mathtt{x} \cdot \underline{\mathrm{b}}$. In the right abstract domain element, the contents of both fields are associated with distinct nodes, and the values denoted by those nodes are constrained to be equal by the numeric domain. In the left graph, the contents of both fields are associated to the same node, which implies that they must be equal (without any constraint in the numeric domain).

Now, with respect to the design of abstract operations in the combined abstract domain, the set of nodes in the shape graph will in general change during the course of the analysis. For instance, the analysis of an assignment of the value contained into field $\underline{\mathrm{a}}$ to field $\underline{\mathrm{b}}$ from the abstract state shown in the left produces the one in the right in Figure 8. After this transformation takes place, node δ becomes "garbage" or irrelevant, as it is not linked anywhere in the shape graph, and no numeric property is attached to it. This symbolic variable δ should thus be removed or projected from the numeric abstract domain. Other operations can cause new symbolic variables to be added, and this issue is only magnified with summaries (cf., Section 3.3). Thus, the combined abstract domain must take great care in ensuring the consistency of the numeric abstract values with the shape graphs, as well as dealing with graphs with different sets of nodes. Considering again the diagram in Figure 6, whenever two shape graphs are ordered $\sigma_0^\sharp \sqsubseteq \sigma_1^\sharp$, there exists a *symbolic variable renaming function* $\Phi\langle\sigma_0^\sharp, \sigma_1^\sharp\rangle : \mathbb{V}^\sharp[\sigma_1^\sharp] \to \mathbb{V}^\sharp[\sigma_0^\sharp]$ that expresses a renaming of the symbolic variables from the weaker shape graph $\sigma_1^\sharp$ to the stronger one $\sigma_0^\sharp$. For example, the symbolic renaming function Φ for the shape graphs shown in Figure 7 is $[\alpha_x \mapsto \alpha_x, \beta_1 \mapsto \beta_0, \delta_1 \mapsto \beta_0]$.

Related work and discussion. In practice, the implementation of the shape abstract domain takes the form of a functor (in the ML programming sense) that takes as input a module implementing a numeric domain interface (e.g., a wrapper on top of the APRON library [20]) and outputs another module that implements the memory abstract domain interface. The construction that we have shown in this section is general to analyses where the set of symbolic variables is dynamic during the course of the analysis and where the inference of this set is bound to the inference of cell contents. In other words, it is well-suited to applying shape analyses for summarizing memory cells and then reasoning about their contents with another domain. This construction has been used not only in Xisa [7] but also in a TVLA-based setup [25] and one based on a history of heap updates [6].

Another approach that avoids this construction by performing a sequence of analyses: first, a shape analysis infers the set of symbolic variables; then, a numeric static analysis relies on this set [23, 24]. While less involved, this approach prevents the exchange of information between both analyses, which is often required to achieve a satisfactory level of precision [7]. This sequencing of heap analysis followed by value analysis is similar to the application of a pre-pass pointer analysis followed by model checking over a Boolean abstraction exemplified in SLAM [1] and BLAST [18]

3.3 Enhancing store abstractions with summaries

So far, we have considered very simple abstract heaps described by separating shape graphs where all concrete memory cells are abstracted by exact points-to edges. To support abstracting a potentially unbounded number of concrete memory cells via dynamic memory allocation, we must extend abstract heaps with *summarization*, that is, a way of providing a compact abstraction for possibly unbounded, possibly non-contiguous memory regions.

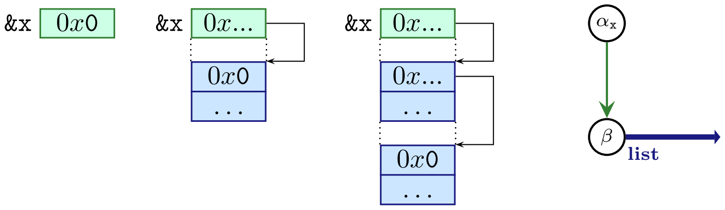


Figure 9: Summarizing linked lists with inductive predicate edges in separating shape graphs.

As an example, consider the concrete stores shown in the left part of Figure 9 consisting of a series of linked lists with 0, 1, and 2 elements. These concrete stores are just instances among infinitely many ones where `x` stores a reference to a list of arbitrary length. Each of these instances consist of two regions: the cell corresponding to variable `x` (green) and the list elements (blue). To abstract all of these stores in a compact and precise manner, we need to summarize the second region with a predicate. We can define such a predicate for summarizing such a region using an inductive definition **list** following the structure of lists: $\alpha \cdot \mathbf{list} := (\mathbf{emp} \wedge \alpha = 0x0) \vee (\alpha \cdot \underline{\mathrm{a}} \mapsto \beta_0 * \alpha \cdot \underline{\mathrm{b}} \mapsto \beta_1 * \beta_0 \cdot \mathbf{list} \wedge \alpha \neq 0x0)$. This definition notation is slightly non-standard to match the graphical notation: the predicate name is **list** and α is the formal induction parameter. A **list** memory region is empty if the root pointer α of the list is null, or otherwise, there is a head list element with two fields $\underline{\mathrm{a}}$ and $\underline{\mathrm{b}}$ such that the contents of cell $\alpha \cdot \underline{\mathrm{a}}$ called β_0 is itself a pointer to a list. Then, in Figure 9, if variable `x` contains a pointer value denoted by β, the second region can be summarized by the inductive predicate instance $\beta \cdot \mathbf{list}$. Furthermore, the three concrete stores are abstracted by the abstract heap $\alpha_{\mathrm{x}} \mapsto \beta * \beta \cdot \mathbf{list}$ (drawn as a graph to the right). The inductive predicate $\beta \cdot \mathbf{list}$ is drawn as the bold, thick edge from node β.

Materialization: The analyzer must be able to apply transfer functions on summarized regions. However, designing precise transfer functions on arbitrary summaries is extremely difficult. An effective approach is to define direct transfer functions only on exact predicates and then define transfer functions on summaries indirectly via *materialization* [32] of exact predicates from them. In the following, we focus on the case where summaries are derived from inductive predicates [8] and thus call the materialization operation *unfolding*. In practice, unfolding should be guided by a specification of the summarized region where the analyzer needs to perform local reasoning on materialized cells (see Section 4.2). However, from the theoretical point of view, we can let an unfolding operator be defined as some function that replaces one abstract $(\sigma^\sharp, \nu^\sharp)$ with a *finite set* of abstract elements $(\sigma_0^\sharp, \nu_0^\sharp), \ldots, (\sigma_{n-1}^\sharp, \nu_{n-1}^\sharp)$.

Definition 5 (Materialization)**.** Let us write $\rightsquigarrow \subseteq (\mathbb{H}^\sharp \rightrightarrows \mathbb{N}^\sharp) \times \mathscr{P}_{\mathrm{fin}}(\mathbb{H}^\sharp \rightrightarrows \mathbb{N}^\sharp)$ for the unfolding relation.

Then, any unfolding of an abstract element should be sound with respect to concretization:

$$\text{If } (\sigma^\sharp, \nu^\sharp) \rightsquigarrow (\sigma^\sharp_0, \nu^\sharp_0), \ldots, (\sigma^\sharp_{n-1}, \nu^\sharp_{n-1}) \text{ ,then } \gamma_{\mathbb{H}^\sharp \rightrightarrows \mathbb{N}^\sharp}(\sigma^\sharp, \nu^\sharp) \subseteq \bigcup_{0 \leq i < n} \gamma_{\mathbb{H}^\sharp \rightrightarrows \mathbb{N}^\sharp}(\sigma^\sharp_i, \nu^\sharp_i) \ .$$

As seen above, the finite set of abstract elements that results from materialization represents a disjunction of abstract elements (i.e., materialization is a form of case analysis). For precision, we typically want an equality instead of inclusion in the conclusion, which motivates a need to represent a disjunction of abstract elements (cf., Section 3.4).

Example 3 (Unfolding an inductively-defined list)**.** For instance, the abstract element from $\mathbb{H}^\sharp \rightrightarrows \mathbb{N}^\sharp$ depicted in Figure 9 can be unfolded to two elements:

$$(\alpha_{\mathtt{x}} \mapsto \beta * \beta \cdot \mathbf{list}, \top) \rightsquigarrow (\alpha_{\mathtt{x}} \mapsto \beta, \beta = 0x0), (\alpha_{\mathtt{x}} \mapsto \beta * \beta \cdot \underline{\mathrm{a}} \mapsto \beta_0 * \beta \cdot \underline{\mathrm{b}} \mapsto \beta_1 * \beta_0 \cdot \mathbf{list}, \beta \neq 0x0) \ ,$$

which means that the list pointer β is either a null pointer or points to a list element whose $\underline{\mathrm{a}}$ field contains a pointer to another list.

Related work and discussion. Historically, the idea of using compact summaries for an unbounded number of concrete memory cells goes back to at least Jones and Muchnick [21], though the set of abstract locations was fixed *a priori* before the analysis. Chase et al. [9] considered dynamic summarization during analysis, while Sagiv et al. [32] introduced materialization. We make note of existing analysis algorithms that make use of summarization-materialization. *TVLA summary nodes* [31] represent unbounded sets of concrete memory cells with predicates that express universal properties of all the concrete cells they denote. The use of three-valued logic enables abstraction beyond a set of exact points-to constraints (i.e., the separating shape graphs in Section 3.1 are akin to two-valued structures in TVLA), and summarization is controlled by instrumentation predicates that limits the compaction done by canonical abstraction. Fixed *list segment predicates* [2, 14] characterize consecutive chains of list elements by its first and last pointers. Thus, a predicate of the form $\mathtt{ls}(\alpha, \alpha')$ denotes all chains of list elements (of any length) starting at α and ending at α'. Then, an abstract heap consists of a separating conjunction of points-to predicates (Section 3.1) and list segments. These predicates can be generalized to other structure segments. *Inductive predicates* [7, 8] generalize the list segment predicates in several ways. First, the abstract domain may be parametrized by a set of user-supplied inductive definitions. Note that as parameters to the abstract domain and thus the analyzer, the inductive definitions specify possible templates for summarization. A sound analysis can only infer a summary predicate essentially if it exhibits an exact instance of the summary. The “correctness” of such inductive definitions are not assumed, but rather a disconnect between the user’s intent and the meaning an inductive predicate could lead to unexpected results. Second, inductive predicates can correspond to complete structures (e.g., a tree that is completely summarized into a single abstract predicate), whereas segments correspond to incomplete structures characterized by a missing sub-structure. Inductive predicates can be generically lifted to unmaterializable segment summaries [8] or materializable ones [7]. Independently, *array region predicates* [15] have been used to describe the contents of zones in arrays. Some analyses on arrays and containers have used index variables into summaries instead of explicit materialization operations [13, 16, 17].

3.4 Lifting store abstractions to disjunctive memory state abstractions

At this point, we have described an abstraction framework for concrete stores σ. To complete an abstraction for memory states $m : (E, \sigma)$, we need two things: (1) an abstract counterpart to E and (2) a disjunctive abstraction for when a single abstract heap $\sigma^\sharp$ is insufficient for precisely abstracting the set of possible concrete stores.

Abstract environments: Since the abstract counterpart for addresses are symbolic variables (or nodes) in shape graphs, an *abstract environment* $E^\sharp$ can simply be a function mapping program variables to nodes, that is, $E^\sharp \in \mathbb{E}^\sharp = \mathbb{X} \to \mathbb{V}^\sharp$. Now, the *memory abstract domain* $\mathbb{M}^\sharp$ is defined by $\mathbb{M}^\sharp = \mathbb{E}^\sharp \times (\mathbb{H}^\sharp \rightrightarrows \mathbb{N}^\sharp)$, and its concretization $\gamma_{\mathbb{M}} : \mathbb{M}^\sharp \to \mathscr{P}(\mathbb{E} \times \mathbb{H})$ can be defined as follows:

$$\gamma_{\mathbb{M}}(E^\sharp, (\sigma^\sharp, \nu^\sharp)) \stackrel{\text{def}}{=} \{(\nu \circ E^\sharp, \sigma) \mid (\sigma, \nu) \in \gamma_{\mathbb{H}}(\sigma^\sharp) \text{ and } \nu \in \gamma_{\text{num}}\langle \mathbb{V}^\sharp[\sigma^\sharp]\rangle(\nu^\sharp)\} \ .$$

Note that in an abstract memory state $m^\sharp : (E^\sharp, \sigma^\sharp)$, the abstract environment $E^\sharp$ simply gives the symbolic address of program variables, while the abstract heap $\sigma^\sharp$ abstracts all memory cells—just like the concrete model in Section 2.2.

We let the abstract environment be depicted by node labels in the graphical representation of abstract heaps. For instance, the concrete memory state shown in Figure 1 can be described by the diagram in Figure 10.

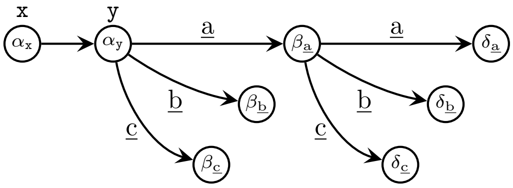


Figure 10: Depicting a memory abstraction including the abstract heap from Figure 4 and an abstract environment.

Disjunctive abstraction: Recall that the unfolding operation from Section 3.3 generates a finite disjunction of abstract facts—specifically, combined shape-numeric abstract elements $\{\ldots, (\sigma_i^\sharp, \nu_i^\sharp), \ldots\} \subseteq \mathbb{H}^\sharp \rightrightarrows \mathbb{N}^\sharp$. Thus, a disjunctive abstraction layer is required regardless of other analysis reasons (e.g., path-sensitivity). We assume the *disjunctive abstraction* is defined by an abstract domain $\mathbb{M}_\vee^\sharp$ and a concretization function $\gamma_\vee : \mathbb{M}_\vee^\sharp \to \mathscr{P}(\mathbb{M})$. We do not prescribe any specific disjunctive abstraction. A simple choice is to apply a disjunctive completion [11], but further innovations might be possible by taking advantage of being specific to memory.

Example 4 (Disjunctive completion)**.** For a memory abstract domain $\mathbb{M}^\sharp$, its disjunctive completion $\mathbb{M}_\vee^\sharp$ is defined as follows:

$$\mathbb{M}_\vee^\sharp \stackrel{\text{def}}{=} \mathscr{P}_{\text{fin}}(\mathbb{M}^\sharp) \qquad\qquad \gamma_\vee(s^\sharp) \stackrel{\text{def}}{=} \bigcup\{\gamma_{\mathbb{M}}(m^\sharp) \mid m^\sharp \in s^\sharp\} \ .$$

In Figure 11, we sum up the structure of the abstract domain for abstracting memory states $\mathbb{M}$ as a stack of layers, which are typically implemented as ML-style functors. Each layer corresponds to the abstraction of a different form of concrete semantics (as shown in the diagram).

Related work and discussion. Trace partitioning [30] relies on control-flow history to manage disjunctions, which could be used as an alternative to disjunctive completion. However, it is a rather general construction and can be instantiated in multiple ways with a large effect on precision and performance.

4 Static analysis operations

In this section, we describe the main abstract operations on the memory abstract domain $\mathbb{M}^\sharp$ and demonstrate how they are computed through the composition of abstract domains discussed in Section 3. Our presentation describes each kind of operation (i.e., transfer functions for commands like assignment, abstract comparison, and abstract join) one by one and shows how unfolding and folding operations are triggered by their application. The end result of this discussion is a description of how these domains implement the interfaces shown in Figure 12. For these interfaces, we let $\mathbb{B}$ denote the set of booleans $\{\textbf{true}, \textbf{false}\}$ and $\mho$ denote an undefined value for some functions that may fail to produce a result. We write $X_\mho$ for $X \uplus \{\mho\}$ for any set X (i.e., an option type).

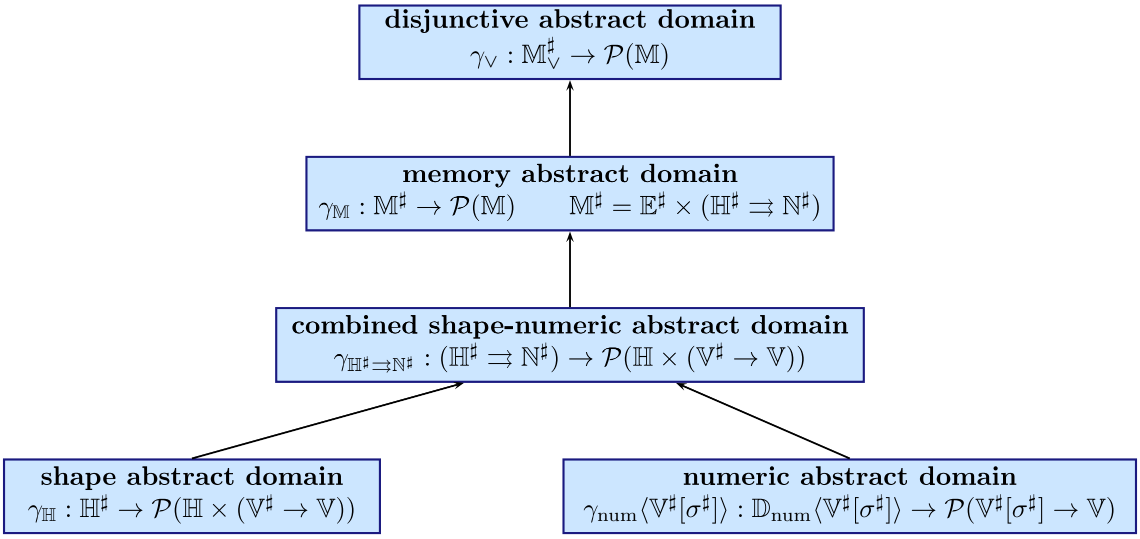

disjunctive abstract domain
$\gamma_\vee : \mathbb{M}^\sharp_\vee \to \mathcal{P}(\mathbb{M})$
memory abstract domain
$\gamma_\mathbb{M} : \mathbb{M}^\sharp \to \mathcal{P}(\mathbb{M})$ $\mathbb{M}^\sharp = \mathbb{E}^\sharp \times (\mathbb{H}^\sharp \rightrightarrows \mathbb{N}^\sharp)$
combined shape-numeric abstract domain
$\gamma_{\mathbb{H}^\sharp \rightrightarrows \mathbb{N}^\sharp} : (\mathbb{H}^\sharp \rightrightarrows \mathbb{N}^\sharp) \to \mathcal{P}(\mathbb{H} \times (\mathbb{V}^\sharp \to \mathbb{V}))$
shape abstract domain
$\gamma_\mathbb{H} : \mathbb{H}^\sharp \to \mathcal{P}(\mathbb{H} \times (\mathbb{V}^\sharp \to \mathbb{V}))$
numeric abstract domain
$\gamma_{\mathrm{num}}\langle \mathbb{V}^\sharp[\sigma^\sharp]\rangle : \mathbb{D}_{\mathrm{num}}\langle \mathbb{V}^\sharp[\sigma^\sharp]\rangle \to \mathcal{P}(\mathbb{V}^\sharp[\sigma^\sharp] \to \mathbb{V})$

Figure 11: Layers of abstract domains to yield a disjunctive memory state abstraction. From an implementation perspective, the edges correspond to inputs for ML-style functor instantiations.

4.1 Assignment over materialized cells

First, we consider the transfer function for assignment. In this subsection, for simplicity, we focus on the case where *none* of the locations that appear in either side of the assignment are summarized, and we defer the case of transfer functions over summarized graph regions to Section 4.2. Because of this simplification, the types of the abstract operators mentioned will not exactly match those given in Figure 12. At the same time, this transfer function captures the essence of the shape-numeric combination.

Recall that $loc \in \mathscr{L}_\mathbb{X}$ and $exp \in \mathscr{E}_\mathbb{X}$ are location and value expressions, respectively, in our programming language (cf., Figure 2). The transfer function $\mathfrak{assign}_{\mathrm{mem}} : \mathscr{L}_\mathbb{X} \times \mathscr{E}_\mathbb{X} \times \mathbb{M}^\sharp \to \mathbb{M}^\sharp$ should compute a sound post-condition for the assignment command $loc = exp$ stated as follows:

Condition 1 (Soundness of $\mathfrak{assign}_{\mathrm{mem}}$)**.** If $(E,\sigma) \in \gamma_\mathbb{M}(m^\sharp)$, then

$$(E,\sigma[\mathscr{L}[\![loc]\!](E,\sigma) \leftarrow \mathscr{E}[\![exp]\!](E,\sigma)]) \in \gamma_\mathbb{M}(\mathfrak{assign}_{\mathrm{mem}}(loc,exp,m^\sharp))\ .$$

Assignments of the form $loc = loc'$**.** Let us first assume that right hand side of the assignment is a location expression. As an example, consider the assignment shown in Figure 13 and applying $\mathfrak{assign}_{\mathrm{mem}}$ to the pre-condition on the left to yield the post-condition on the right. The essence is that *loc* dictates an edge that should be updated to point to the node specified by loc'.

To compute a post-condition in this case, $\mathfrak{assign}_{\mathrm{mem}}$ should update the abstract heap, that is, the preheap $\sigma^\sharp \in \mathbb{H}^\sharp$. An $\mathfrak{assign}_{\mathrm{mem}}$ call should eventually forward the assignment to the heap abstract domain via the $\mathfrak{eval}[\mathfrak{l}]_{\mathrm{shape}}$ operation that evaluates a location expression *loc* to an edge, $\mathfrak{eval}[\mathfrak{e}]_{\mathrm{shape}}$ that evaluates a value expression *exp* to a node, and $\mathfrak{mutate}_{\mathrm{shape}}$ that swings a points-to edge.

The base of a sequence of pointer dereferences is given by a program variable, so the first step consists of replacing the program variables in the assignment with the symbolic names corresponding to their addresses using the abstract environment $E^\sharp$. For our example, this results in the call to $\mathfrak{assign}_{\mathrm{comb}}(\alpha_0\ \texttt{->}\ \underline{\mathrm{a}}\cdot\underline{\mathrm{b}}, \alpha_0\cdot\underline{\mathrm{b}}, (\sigma^\sharp,\nu^\sharp))$ at the combined shape-numeric layer, which should satisfy a soundness condition similar to that of $\mathfrak{assign}_{\mathrm{mem}}$ (Condition 1). The next step consists of traversing the abstract heap $\sigma^\sharp$

- A shape abstract domain $\mathbb{H}^\sharp$

$$\begin{array}{rlcl}
\mathfrak{eval}[\mathfrak{l}]_{\text{shape}}: & \mathscr{L}_{\mathbb{V}^\sharp} \times \mathbb{H}^\sharp & \longrightarrow & (\mathbb{V}^\sharp \times \mathbb{F} \times \mathbb{H}^\sharp)_{\mathfrak{U}} \\
\mathfrak{eval}[\mathfrak{e}]_{\text{shape}}: & \mathscr{E}_{\mathbb{V}^\sharp} \times \mathbb{H}^\sharp & \longrightarrow & (\mathbb{V}^\sharp \times \mathbb{H}^\sharp)_{\mathfrak{U}} \\
\mathfrak{mutate}_{\text{shape}}: & \mathbb{V}^\sharp \times \mathbb{F} \times \mathbb{V}^\sharp \times \mathbb{H}^\sharp & \longrightarrow & \mathbb{H}^\sharp_{\mathfrak{U}} \\
\mathfrak{unfold}_{\text{shape}}: & (\mathscr{L}_{\mathbb{V}^\sharp} \times \mathbb{F}) \times \mathbb{H}^\sharp & \longrightarrow & \mathscr{P}_{\text{fin}}(\mathbb{H}^\sharp \times \mathscr{E}_{\mathbb{V}^\sharp}) \\
\mathfrak{new}_{\text{shape}}: & \mathbb{V}^\sharp \times \mathbb{H}^\sharp & \longrightarrow & \mathbb{H}^\sharp \\
\mathfrak{delete}[\mathfrak{n}]_{\text{shape}}: & \mathbb{V}^\sharp \times \mathbb{H}^\sharp & \longrightarrow & \mathbb{H}^\sharp \\
\mathfrak{delete}[\mathfrak{e}]_{\text{shape}}: & \mathbb{V}^\sharp \times \mathbb{F} \times \mathbb{H}^\sharp & \longrightarrow & \mathbb{H}^\sharp \\
\mathfrak{compare}_{\text{shape}}: & (\mathbb{V}^\sharp \to \mathbb{V}^\sharp) \times \mathbb{H}^\sharp \times \mathbb{H}^\sharp & \longrightarrow & \{\textbf{false}\} \uplus \{\textbf{true}\} \times (\mathbb{V}^\sharp \to \mathbb{V}^\sharp) \\
\mathfrak{join}_{\text{shape}}: & ((\mathbb{V}^\sharp)^2 \to \mathbb{V}^\sharp) \times \mathbb{H}^\sharp \times \mathbb{H}^\sharp & \longrightarrow & \mathbb{H}^\sharp \times (\mathbb{V}^\sharp \to \mathbb{V}^\sharp)^2 \\
\mathfrak{widen}_{\text{shape}}: & ((\mathbb{V}^\sharp)^2 \to \mathbb{V}^\sharp) \times \mathbb{H}^\sharp \times \mathbb{H}^\sharp & \longrightarrow & \mathbb{H}^\sharp \times (\mathbb{V}^\sharp \to \mathbb{V}^\sharp)^2
\end{array}$$

- A numeric abstract domain over symbolic variables $\mathbb{N}^\sharp$

$$\begin{array}{rlcl}
\mathfrak{assign}_{\text{num}}: & \mathscr{L}_{\mathbb{V}^\sharp} \times \mathscr{E}_{\mathbb{V}^\sharp} \times \mathbb{N}^\sharp & \longrightarrow & \mathbb{N}^\sharp \\
\mathfrak{guard}_{\text{num}}: & \mathscr{E}_{\mathbb{V}^\sharp} \times \mathbb{N}^\sharp & \longrightarrow & \mathbb{N}^\sharp \\
\mathfrak{new}_{\text{num}}: & \mathbb{V}^\sharp \times \mathbb{N}^\sharp & \longrightarrow & \mathbb{N}^\sharp \\
\mathfrak{delete}[\mathfrak{n}]_{\text{num}}: & \mathbb{V}^\sharp \times \mathbb{N}^\sharp & \longrightarrow & \mathbb{N}^\sharp \\
\mathfrak{rename}_{\text{num}}: & (\mathbb{V}^\sharp \to \mathbb{V}^\sharp) \times \mathbb{N}^\sharp & \longrightarrow & \mathbb{N}^\sharp \\
\mathfrak{compare}_{\text{num}}: & \mathbb{N}^\sharp \times \mathbb{N}^\sharp & \longrightarrow & \mathbb{B} \\
\mathfrak{join}_{\text{num}}: & \mathbb{N}^\sharp \times \mathbb{N}^\sharp & \longrightarrow & \mathbb{N}^\sharp \\
\mathfrak{widen}_{\text{num}}: & \mathbb{N}^\sharp \times \mathbb{N}^\sharp & \longrightarrow & \mathbb{N}^\sharp
\end{array}$$

- A combined shape-numeric abstract domain $\mathbb{H}^\sharp \rightrightarrows \mathbb{N}^\sharp$

$$\begin{array}{rlcl}
\mathfrak{assign}_{\text{comb}}: & \mathscr{L}_{\mathbb{V}^\sharp} \times \mathscr{E}_{\mathbb{V}^\sharp} \times \mathbb{H}^\sharp \rightrightarrows \mathbb{N}^\sharp & \longrightarrow & \mathscr{P}_{\text{fin}}(\mathbb{H}^\sharp \rightrightarrows \mathbb{N}^\sharp)_{\mathfrak{U}} \\
\mathfrak{guard}_{\text{comb}}: & \mathscr{E}_{\mathbb{V}^\sharp} \times (\mathbb{H}^\sharp \rightrightarrows \mathbb{N}^\sharp) & \longrightarrow & \mathscr{P}_{\text{fin}}(\mathbb{H}^\sharp \rightrightarrows \mathbb{N}^\sharp)_{\mathfrak{U}} \\
\mathfrak{unfold}_{\text{comb}}: & \mathscr{L}_{\mathbb{V}^\sharp} \times \mathbb{H}^\sharp \rightrightarrows \mathbb{N}^\sharp & \longrightarrow & \mathscr{P}_{\text{fin}}(\mathbb{H}^\sharp \rightrightarrows \mathbb{N}^\sharp)_{\mathfrak{U}} \\
\mathfrak{alloc}_{\text{comb}}: & \mathscr{L}_{\mathbb{V}^\sharp} \times \mathbb{F}^* \times (\mathbb{H}^\sharp \rightrightarrows \mathbb{N}^\sharp) & \longrightarrow & \mathscr{P}_{\text{fin}}(\mathbb{H}^\sharp \rightrightarrows \mathbb{N}^\sharp)_{\mathfrak{U}} \\
\mathfrak{free}_{\text{comb}}: & \mathscr{L}_{\mathbb{V}^\sharp} \times \mathbb{F}^* \times (\mathbb{H}^\sharp \rightrightarrows \mathbb{N}^\sharp) & \longrightarrow & \mathscr{P}_{\text{fin}}(\mathbb{H}^\sharp \rightrightarrows \mathbb{N}^\sharp)_{\mathfrak{U}} \\
\mathfrak{compare}_{\text{comb}}: & (\mathbb{V}^\sharp \to \mathbb{V}^\sharp) \times (\mathbb{H}^\sharp \rightrightarrows \mathbb{N}^\sharp) \times (\mathbb{H}^\sharp \rightrightarrows \mathbb{N}^\sharp) & \longrightarrow & \{\textbf{false}\} \uplus \{\textbf{true}\} \times (\mathbb{V}^\sharp \to \mathbb{V}^\sharp) \\
\mathfrak{join}_{\text{comb}}: & ((\mathbb{V}^\sharp)^2 \to \mathbb{V}^\sharp) \times (\mathbb{H}^\sharp \rightrightarrows \mathbb{N}^\sharp) \times (\mathbb{H}^\sharp \rightrightarrows \mathbb{N}^\sharp) & \longrightarrow & (\mathbb{H}^\sharp \rightrightarrows \mathbb{N}^\sharp) \\
\mathfrak{widen}_{\text{comb}}: & ((\mathbb{V}^\sharp)^2 \to \mathbb{V}^\sharp) \times (\mathbb{H}^\sharp \rightrightarrows \mathbb{N}^\sharp) \times (\mathbb{H}^\sharp \rightrightarrows \mathbb{N}^\sharp) & \longrightarrow & (\mathbb{H}^\sharp \rightrightarrows \mathbb{N}^\sharp)
\end{array}$$

- A memory abstract domain $\mathbb{M}^\sharp$

$$\begin{array}{rlcl}
\mathfrak{assign}_{\text{mem}}: & \mathscr{L}_{\mathbb{X}} \times \mathscr{E}_{\mathbb{X}} \times \mathbb{M}^\sharp & \longrightarrow & \mathscr{P}_{\text{fin}}(\mathbb{M}^\sharp)_{\mathfrak{U}} \\
\mathfrak{guard}_{\text{mem}}: & \mathscr{E}_{\mathbb{X}} \times \mathbb{M}^\sharp & \longrightarrow & \mathscr{P}_{\text{fin}}(\mathbb{M}^\sharp)_{\mathfrak{U}} \\
\mathfrak{alloc}_{\text{mem}}: & \mathscr{L}_{\mathbb{X}} \times \mathbb{F}^* \times \mathbb{M}^\sharp & \longrightarrow & \mathscr{P}_{\text{fin}}(\mathbb{M}^\sharp)_{\mathfrak{U}} \\
\mathfrak{free}_{\text{mem}}: & \mathscr{L}_{\mathbb{X}} \times \mathbb{F}^* \times \mathbb{M}^\sharp & \longrightarrow & \mathscr{P}_{\text{fin}}(\mathbb{M}^\sharp)_{\mathfrak{U}} \\
\mathfrak{compare}_{\text{mem}}: & \mathbb{M}^\sharp \times \mathbb{M}^\sharp & \longrightarrow & \mathbb{B} \\
\mathfrak{join}_{\text{mem}}: & \mathbb{M}^\sharp \times \mathbb{M}^\sharp & \longrightarrow & \mathbb{M}^\sharp \\
\mathfrak{widen}_{\text{mem}}: & \mathbb{M}^\sharp \times \mathbb{M}^\sharp & \longrightarrow & \mathbb{M}^\sharp
\end{array}$$

Figure 12: Interfaces for the abstract domain layers shown in Figure 11 (except the disjunctive abstraction layer).

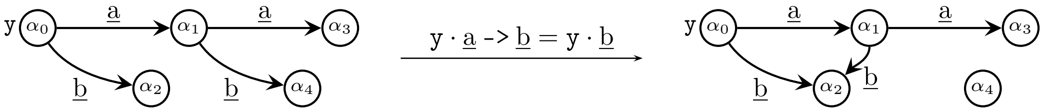


Figure 13: Applying $\mathfrak{assign}_{\mathrm{mem}}$ to an example assignment of the form $loc = loc'$.

$$\textsc{LocAddress}\ \frac{}{\mathfrak{eval}[\mathfrak{l}]_{\mathrm{shape}}(\alpha,\sigma^{\sharp}) = (\alpha,\underline{\emptyset})} \qquad \textsc{LocField}\ \frac{\mathfrak{eval}[\mathfrak{l}]_{\mathrm{shape}}(loc,\sigma^{\sharp}) = (\alpha,\underline{\mathrm{f}})}{\mathfrak{eval}[\mathfrak{l}]_{\mathrm{shape}}(loc\cdot\underline{\mathrm{g}},\sigma^{\sharp}) = (\alpha,\underline{\mathrm{f}}+\underline{\mathrm{g}})} \qquad \textsc{LocVal}\ \frac{\mathfrak{eval}[\mathfrak{e}]_{\mathrm{shape}}(exp,\sigma^{\sharp}) = \alpha}{\mathfrak{eval}[\mathfrak{l}]_{\mathrm{shape}}(\star exp,\sigma^{\sharp}) = (\alpha,\underline{\emptyset})}$$

$$\textsc{ValDereference}\ \frac{\mathfrak{eval}[\mathfrak{l}]_{\mathrm{shape}}(loc,\sigma^{\sharp}) = (\alpha,\underline{\mathrm{f}}) \qquad \sigma^{\sharp} = \sigma_0^{\sharp} * \alpha\cdot\underline{\mathrm{f}} \mapsto \beta}{\mathfrak{eval}[\mathfrak{e}]_{\mathrm{shape}}(loc,\sigma^{\sharp}) = \beta} \qquad \textsc{ValLoc}\ \frac{\mathfrak{eval}[\mathfrak{l}]_{\mathrm{shape}}(loc,\sigma^{\sharp}) = (\alpha,\underline{\emptyset})}{\mathfrak{eval}[\mathfrak{e}]_{\mathrm{shape}}(\&loc,\sigma^{\sharp}) = \alpha}$$

Figure 14: Evaluating dereferences in an abstract heap.

according to the location expression and the value expression of the assignment. As mentioned above, this evaluation is performed using the location evaluation function $\mathfrak{eval}[\mathfrak{l}]_{\mathrm{shape}}$ that yields an edge and the value expression evaluation function $\mathfrak{eval}[\mathfrak{e}]_{\mathrm{shape}}$ that yields a node.

Condition 2 (Soundness of $\mathfrak{eval}[\mathfrak{l}]_{\mathrm{shape}}$ and $\mathfrak{eval}[\mathfrak{e}]_{\mathrm{shape}}$)**.** Let $(\sigma,\nu) \in \gamma_{\mathbb{H}}(\sigma^{\sharp})$. Then,

$$\begin{array}{l}\text{If } \mathfrak{eval}[\mathfrak{l}]_{\mathrm{shape}}(loc,\sigma^{\sharp}) = (\alpha,\underline{\mathrm{f}}) \text{ ,then } \mathscr{L}[\![loc]\!](\sigma) = \nu(\alpha) + \underline{\mathrm{f}}\ . \\ \text{If } \mathfrak{eval}[\mathfrak{e}]_{\mathrm{shape}}(loc,\sigma^{\sharp}) = \beta \text{ ,then } \mathscr{E}[\![loc]\!](\sigma) = \nu(\beta)\ .\end{array}$$

In Figure 14, we define $\mathfrak{eval}[\mathfrak{l}]_{\mathrm{shape}}$ and $\mathfrak{eval}[\mathfrak{e}]_{\mathrm{shape}}$ following the syntax of location and value expressions (over symbolic variables). We write $\underline{\emptyset}$ for a designated 0-offset field. This abstract evaluation corresponds directly to the concrete evaluation defined in Figure 3. Note that abstract evaluation is not necessarily defined for all expressions. For example, an points-to edge may simply not exist for the computed address in ValDereference. The edge may need to be *materialized* by unfolding (cf., Section 4.2) or otherwise is a potential memory error.

Returning to the example in Figure 13, we get $\mathfrak{eval}[\mathfrak{l}]_{\mathrm{shape}}(\alpha_0 \text{ -> } \underline{\mathrm{a}}\cdot\underline{\mathrm{b}},\sigma^{\sharp}) = (\alpha_1,\underline{\mathrm{b}})$—the cell being assigned-to corresponds to the exact points-to edge $\alpha_1\cdot\underline{\mathrm{b}} \mapsto \alpha_4$—and $\mathfrak{eval}[\mathfrak{e}]_{\mathrm{shape}}(\alpha_0\cdot\underline{\mathrm{b}}) = \alpha_2$—the value to assign is abstracted by α_2. The abstract post-condition returned by $\mathfrak{assign}_{\mathrm{comb}}$ should reflect the swinging of that edge in the shape graph, which is accomplished by the $\mathfrak{mutate}_{\mathrm{shape}}$ function:

$$\mathfrak{mutate}_{\mathrm{shape}}(\alpha,\underline{\mathrm{f}},\beta,(\alpha\cdot\underline{\mathrm{f}} \mapsto \delta) * \sigma^{\sharp}) = (\alpha\cdot\underline{\mathrm{f}} \mapsto \beta) * \sigma^{\sharp}\ .$$

This function simply replaces a points-to edge named by the address α and field $\underline{\mathrm{f}}$ with a new one for the updated contents (and fails if such a points-to edge does not exist in the abstract heap $\sigma^{\sharp}$). The effect of this assignment can be completely reflected in the abstract heap since the cell corresponding to the assignment is abstracted by exactly one points-to edge and the new value to store in that cell is also exactly abstracted by one node. We note that node α_4 is no longer reachable in the shape graph, and thus the value that this node denotes is no longer relevant when concretizing the abstract state. As a consequence, it can be safely removed both in $\mathbb{H}^{\sharp}$ (using function $\mathfrak{delete}[\mathfrak{n}]_{\mathrm{shape}}$) and in $\mathbb{N}^{\sharp}$ (using function

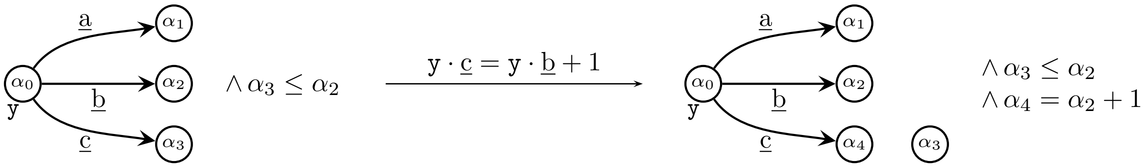

a
b
c
y
α_0
α_1
α_2
α_3
$\wedge\,\alpha_3 \le \alpha_2$
$\mathtt{y}\cdot\underline{\mathtt{c}} = \mathtt{y}\cdot\underline{\mathtt{b}} + 1$
a
b
c
y
α_0
α_1
α_2
α_4
α_3
$\wedge\,\alpha_3 \le \alpha_2$
$\wedge\,\alpha_4 = \alpha_2 + 1$

Figure 15: Applying $\mathfrak{assign}_{\text{mem}}$ to an example assignment of the form $loc = exp$.

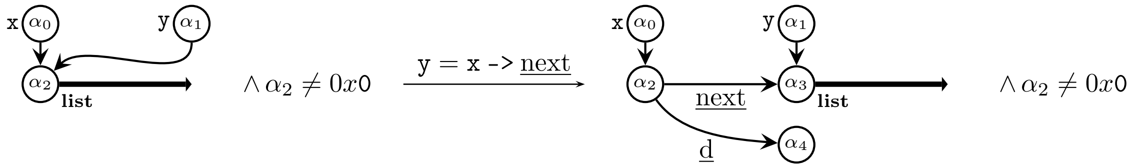


Figure 16: Applying $\mathfrak{assign}_{\text{mem}}$ to an example that affects the summarized region $\alpha_2 \cdot \textbf{list}$.

$\mathfrak{delete}[\mathfrak{n}]_{\text{num}}$). Such an existential projection or "garbage collection" step may be viewed as a conversion operation in the cofibered lattice structure shown in Figure 6.

Assignments of the form $loc = exp$**.** In general, the right-hand side of an assignment is not necessarily a location expression. The evaluation of left-hand side *loc* proceeds as above, but the evaluation of the right-hand side expression *exp* is extended. As an example, consider the assignment shown in Figure 15.

The evaluation of the location expression down to the abstract heap level works as before where we find that $\mathfrak{eval}[\mathfrak{l}]_{\text{shape}}(\alpha_0 \cdot \underline{\mathrm{c}}, \sigma^\sharp) = (\alpha_0, \underline{\mathrm{c}})$. For the right-hand–side expression, it is not obvious what $\mathfrak{eval}[\mathfrak{e}]_{\text{shape}}(\alpha_0 \cdot \underline{\mathrm{b}} + 1, \sigma^\sharp)$ should return, as no symbolic node is equal to that value in the concretization of all elements of $\sigma^\sharp$. It is possible to evaluate sub-expression $\alpha_0 \cdot \underline{\mathrm{b}}$ to α_2, but then $\mathfrak{eval}[\mathfrak{e}]_{\text{shape}}(\alpha_2 + 1, \sigma^\sharp)$ cannot be evaluated any further. The solution is to create a new symbolic variable and constrain it to represent the value of the right-hand–side expression. Therefore, the evaluation of $\mathfrak{assign}_{\text{comb}}$ proceeds as follows: (1) generate a fresh node α_4; (2) add α_4 to the abstract heap $\sigma^\sharp$ and the numeric abstract value $\nu^\sharp$ using the function $\mathfrak{new}_{\text{shape}}$ and $\mathfrak{new}_{\text{num}}$, respectively; (3) update the numeric abstract value $\nu^\sharp$ using $\mathfrak{assign}_{\text{num}}(\alpha_4, \alpha_2 + 1, \nu^\sharp)$, which over-approximates constraining $\alpha_4 = \alpha_2 + 1$; and (4) mutate with $\mathfrak{mutate}_{\text{shape}}$ with the new node α_4 (i.e., $\mathfrak{mutate}_{\text{shape}}(\alpha_0, \underline{\mathrm{c}}, \alpha_4, \sigma^\sharp)$).

4.2 Unfolding and assignment over summarized cells

We now consider $\mathfrak{assign}_{\text{mem}}$ in the presence of summary predicates, which intuitively "get in the way" of evaluating location and value expressions in a shape graph. For instance, consider trying to apply the assignment shown in Figure 16. On the left, we have a separating shape graph where α_2 is a list described by the inductive definition shown inset. For clarity, we also show the C-style **struct** definition that corresponds to the layout of each list element. In applying the assignment, the evaluation of the right-hand–side expression x -> $\underline{\text{next}}$ fails. While x evaluates to node α_2, there is no points-to edge from α_2. Thus, $\mathfrak{eval}[\mathfrak{e}]_{\text{shape}}(\alpha_0 \text{ -> } \underline{\text{next}})$ fails. It is clear that the reason for this failure is that the memory cell corresponding to the right-hand–side expression is *summarized* as part of the $\alpha_2 \cdot \textbf{list}$ predicate. To materialize this cell, this predicate

struct list { **struct list** $\star$ $\underline{\text{next}}$; **int** $\underline{\text{d}}$; };
$\alpha \cdot \textbf{list} := (\textbf{emp} \wedge \alpha = 0x0)$
$\quad\vee\; (\alpha \cdot \underline{\text{next}} \mapsto \beta_0 * \alpha \cdot \underline{\text{d}} \mapsto \beta_1 * \beta_0 \cdot \textbf{list} \wedge \alpha \neq 0x0)$

should be *unfolded*; then, the assignment can proceed as in the previous section (Section 4.1). We can now describe the transfer function for assignment $\mathfrak{assign}_{\mathrm{mem}}(loc, exp, (\sigma^\sharp, \nu^\sharp))$ in general:

1. It should call the underlying $\mathfrak{assign}_{\mathrm{comb}}$ and follow the process described previously in Section 4.1. If evaluation via $\mathfrak{eval}[\mathfrak{l}]_{\mathrm{shape}}$ or $\mathfrak{eval}[\mathfrak{e}]_{\mathrm{shape}}$ fail, then they should return a failure address, which consists of a pair $(\beta, \underline{\mathrm{f}})$ corresponding to the node and field offset that does not have a materialized points-to edge. In the example in Figure 16, the failure address is $(\alpha_2, \underline{\mathrm{next}})$. Note that the interface for evaluation shown in Figure 16 does not show the contents of the failure case for simplicity.
2. Then, $\mathfrak{assign}_{\mathrm{comb}}$ in the combined domain performs an unfolding of the abstract heap by calling a function $\mathfrak{unfold}_{\mathrm{shape}}$ that implements the unfolding relation $\rightsquigarrow$ with the target points-to edge to materialize $(\beta, \underline{\mathrm{f}})$.

   Condition 3 (Soundness of $\mathfrak{unfold}_{\mathrm{shape}}$)**.**

   $$\gamma_{\mathbb{H}}(\sigma^\sharp) \subseteq \bigcup\{(\sigma, \nu) \in \gamma_{\mathbb{H}}(\sigma^\sharp_{\mathrm{u}}) \mid (\sigma^\sharp_{\mathrm{u}}, exp_{\mathrm{u}}) \in \mathfrak{unfold}_{\mathrm{shape}}((\beta, \underline{\mathrm{f}}), \sigma^\sharp) \text{ and } [\![exp_{\mathrm{u}}]\!](\nu) = \mathbf{true}\}\,.$$

   Note that unfolding of an abstract heap returns pairs consisting of an unfolded abstract heap and a numeric constraint as an expression $exp_{\mathrm{u}} \in \mathscr{E}_{\mathbb{V}^\sharp[\sigma^\sharp_{\mathrm{u}}]}$ over the symbolic variables of the unfolded abstract heap. This expression allows a summary to contain constraints not expressible in a shape graph itself. For instance, in the **list** inductive definition, each case comes with a nullness or non-nullness condition on the head pointer. Or more interestingly, we can imagine an orderedness constraint for an inductive definition describing an ordered list. For the example from Figure 16, unfolding the shape graph at $(\alpha_2, \underline{\mathrm{next}})$ generates two disjuncts, but the one corresponding to the empty list can be eliminated due to the constraint that α_2 has to be non-null.
3. The numeric constraints should be evaluated in the numeric abstract domain using a condition test operator $\mathfrak{guard}_{\mathrm{num}}$.

   Condition 4 (Soundness of $\mathfrak{guard}_{\mathrm{num}}$)**.** Let $V \subseteq \mathbb{V}^\sharp$, $\nu^\sharp \in \mathbb{D}_{\mathrm{num}}\langle V\rangle$, and $\nu \in \gamma_{\mathrm{num}}\langle V\rangle(\nu^\sharp)$. Then,

   $$\text{If } [\![exp]\!](\nu) = \mathbf{true}\text{ , then } \nu \in \gamma_{\mathrm{num}}\langle V\rangle(\mathfrak{guard}_{\mathrm{num}}(exp, \nu^\sharp))\,.$$

   Thus, the initial abstract state in the combined domain $(\sigma^\sharp, \nu^\sharp) \in \mathbb{H}^\sharp \rightrightarrows \mathbb{N}^\sharp$ can be over-approximated by the following finite set of abstract states:

   $$\mathfrak{unfold}_{\mathrm{comb}}(loc, (\sigma^\sharp, \nu^\sharp)) \stackrel{\text{def}}{=} \{(\sigma^\sharp_{\mathrm{u}}, \mathfrak{guard}_{\mathrm{num}}(exp_{\mathrm{u}}, \nu^\sharp)) \mid (\sigma^\sharp_{\mathrm{u}}, exp_{\mathrm{u}}) \in \mathfrak{unfold}_{\mathrm{shape}}((\beta, \underline{\mathrm{f}}), \sigma^\sharp)\}$$
4. Finally, $\mathfrak{assign}_{\mathrm{comb}}$ should perform the same set of operations as described in Section 4.1 to reflect the assignment on *each* unfolded heap. The $\mathfrak{assign}_{\mathrm{comb}}$ returns a *finite set* of elements because of potential unfolding (and similarly for $\mathfrak{assign}_{\mathrm{mem}}$). The soundness condition for $\mathfrak{assign}_{\mathrm{mem}}$ is therefore as follows.

   Condition 5 (Soundness of $\mathfrak{assign}_{\mathrm{mem}}$)**.** Let $(E, \sigma) \in \gamma_{\mathbb{M}}(m^\sharp)$. Then,

   $$(E, \sigma[\mathscr{L}[\![loc]\!](E, \sigma) \leftarrow \mathscr{E}[\![exp]\!](E, \sigma)]) \in \bigcup\{\gamma_{\mathbb{M}}(m^\sharp_{\mathrm{u}}) \mid m^\sharp_{\mathrm{u}} \in \mathfrak{assign}_{\mathrm{mem}}(loc, exp, m^\sharp)\}\,.$$

   A very similar soundness condition applies to $\mathfrak{assign}_{\mathrm{comb}}$.

Figure 16 shows the resulting abstract state for the assignment after unfolding and mutation on the right. In certain cases, the unfolding process may have to be performed multiple times due to repeated failures of calling $\mathfrak{eval}[\mathfrak{l}]_{\mathrm{shape}}$ and $\mathfrak{eval}[\mathfrak{e}]_{\mathrm{shape}}$ as shown in Chang and Rival [7]. This behavior is expected, as unfolding may fail to materialize the correct region, and thus, termination should be enforced with a bound on the number of unfolding steps.

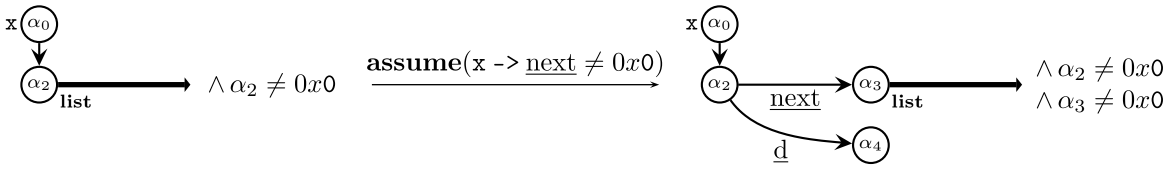

x
α_0
α_2
list
$\wedge\, \alpha_2 \neq 0x0$
assume(x -> next $\neq 0x0$)
x
α_0
α_2
next
α_3
list
d
α_4
$\wedge\, \alpha_2 \neq 0x0$
$\wedge\, \alpha_3 \neq 0x0$

Figure 17: Applying the condition test $\mathfrak{guard}_{\text{mem}}$ to an example that affects a summarized region $\alpha_2 \cdot \textbf{list}$.

4.3 Other transfer functions

Unfolding is also the basis for most other transfer functions. Once the points-to edges in question are materialized, their definition is straightforward as it was for assignment (cf., Section 4.1).

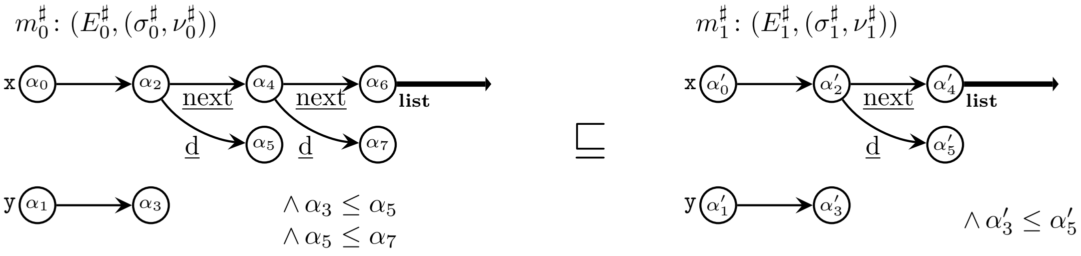
- **Condition test.** The abstract domain $\mathbb{M}^\sharp$ should define an operator $\mathfrak{guard}_{\text{mem}}$ that takes an expression (of Boolean type) and an abstract value and then returns an abstract value that has taken into account the effect of the guard expression. Just like with assignment, this function may need to perform an unfolding and thus returns in general a *finite set* of abstract states.
  Condition 6 (Soundness of $\mathfrak{guard}_{\text{mem}}$)**.** Let $m \in \gamma_{\mathbb{M}}(m^\sharp)$. Then,

  $$\text{If } [\![exp]\!](m) = \textbf{true} \text{ ,then } m \in \bigcup\{\gamma_{\mathbb{H}}(\sigma_{\text{u}}^\sharp) \mid \sigma_{\text{u}}^\sharp \in \gamma_{\mathbb{M}}(\mathfrak{guard}_{\text{mem}}(exp, m^\sharp))\} \ .$$

  It applies the transfer function $\mathfrak{assign}_{\text{num}}$ provided by $\mathbb{N}^\sharp$ satisfying a similar soundness condition, which is fairly standard (e.g., the APRON library provides such a function).
- **Memory allocation.** Transfer function $\mathfrak{alloc}_{\text{mem}}$ accounts for the allocation of a fresh memory block, and the assignment of the address of this block to a given location. Given abstract pre-condition $\sigma^\sharp$, the abstract allocation function $\mathfrak{alloc}_{\text{mem}}(loc, [\underline{\text{f}}_1, \ldots, \underline{\text{f}}_n], \sigma^\sharp)$ returns a sound abstract post-condition for the statement $loc = \textbf{malloc}(\{\underline{\text{f}}_1, \ldots, \underline{\text{f}}_n\})$.
- **Memory deallocation.** Similarly, transfer function $\mathfrak{free}_{\text{mem}}$ accounts for freeing the block pointed to by an instruction such as **free**. It takes as argument a location pointing to the block being freed, a list of fields, and the abstract pre-condition. It may also need to perform unfolding to materialize the location. It calls $\mathfrak{free}_{\text{comb}}$ in the $\mathbb{H}^\sharp \rightrightarrows \mathbb{N}^\sharp$ level, which then materializes points-to edges corresponding to the block to remove and deletes them from the graph using function $\mathfrak{delete}[\mathfrak{e}]_{\text{shape}}$ defined by $\mathfrak{delete}[\mathfrak{e}]_{\text{shape}}(\alpha, \underline{\text{f}}, \alpha \cdot \underline{\text{f}} \mapsto \beta * \sigma_0^\sharp) = \sigma_0^\sharp$. After removing these edges, some symbolic nodes may become unreachable in the graph and should be removed using $\mathfrak{delete}[\mathfrak{n}]_{\text{shape}}$ and $\mathfrak{delete}[\mathfrak{n}]_{\text{num}}$.

The analysis of a more full featured programming language would require additional classical transfer functions, such as support for variable creation and deletion, though this can be supported completely at the memory abstract domain $\mathbb{M}^\sharp$ layer with the abstract environment $E^\sharp$.

As an example of a condition test, consider applying $\mathfrak{guard}_{\text{mem}}$ in Figure 17. In the same way as for the example assignment of Figure 16, the first attempt to compute $\mathfrak{guard}_{\text{comb}}(\alpha_2 \text{ -> } \underline{\text{next}} \neq 0x0, \sigma^\sharp)$ fails, as there is no points-to edge labeled with next starting from node α_2. Thus $\mathfrak{guard}_{\text{comb}}$ must first call $\mathfrak{unfold}_{\text{comb}}$. The unfolding returns a pair of abstract elements, yet the one corresponding to the case where the list is empty does not need to be considered any further due to the numerical constraint $\alpha_2 \neq 0x0$. Therefore, only the second abstract elements remains, which corresponds to a list with the first element materialized. At this stage, expression α_2 -> next can be evaluated. Finally, the condition test is reflected by applying $\mathfrak{guard}_{\text{num}}$ in the numerical abstract domain $\mathbb{N}^\sharp$.



$m_0^\sharp : (E_0^\sharp, (\sigma_0^\sharp, \nu_0^\sharp))$
x
α_0
α_2
next
α_4
next
α_6
list
d
α_5
d
α_7
y
α_1
α_3
$\wedge\, \alpha_3 \leq \alpha_5$
$\wedge\, \alpha_5 \leq \alpha_7$
$\sqsubseteq$
$m_1^\sharp : (E_1^\sharp, (\sigma_1^\sharp, \nu_1^\sharp))$
x
α_0'
α_2'
next
α_4'
list
d
α_5'
y
α_1'
α_3'
$\wedge\, \alpha_3' \leq \alpha_5'$

Figure 18: An abstract inclusion that holds and shows the need for a node relation Φ. In both abstract heaps, variable x points to a list and y points to a number. On the left, the abstract heap describes a list with at least two elements, while on the right, it describes one with at least one element. The number pointed to by y is less than or equal to the data field d of the first element in both abstract heaps. The data field of the first element is less than or equal to the data field of the second in the left abstract heap.

4.4 Abstract comparison

Abstract interpreters make use of inclusion testing operations in many situations, such as checking that an abstract post-fixed point has been reached in a loop invariant computation or that some, for example, user-supplied post-condition can be verified with the analysis results. As inclusion is often not decidable, the comparison function is not required to be complete but should meet a soundness condition:

Condition 7 (Soundness of $\mathfrak{compare}_{\mathrm{mem}}$)**.** If $\mathfrak{compare}_{\mathrm{mem}}(m_0^\sharp, m_1^\sharp) = \mathbf{true}$, then $\gamma_{\mathbb{M}}(m_0^\sharp) \subseteq \gamma_{\mathbb{M}}(m_1^\sharp)$

The implementation of such an operator is complicated by the fact that the underlying abstract heaps may have *distinct* sets of symbolic nodes. This issue is a manifestation of the the cofibered abstract domain construction (Section 3.2). The concretizations of all abstract domains below $\mathbb{H}^\sharp \rightrightarrows \mathbb{N}^\sharp$ make use of *valuations*, and thus the inclusion checking operator needs to account for a relation between the symbolic nodes of the graphs. This relation between nodes in two graphs Φ is computed step-by-step during the course of the inclusion checking.

The example in Figure 18 illustrates these difficulties. It is quite intuitive that any state in the concretization of $m_0^\sharp$ is also in the concretization of $m_1^\sharp$. To see the role of the node relation Φ, let us consider concretizations of $m_0^\sharp$ and $m_1^\sharp$. Clearly, if concrete state (E, σ) is in the concretization of $m_0^\sharp$ and in the concretization of $m_1^\sharp$, then node α_0 in $m_0^\sharp$ and α_0' denotes the address of x. Thus α_0 and α_0' denote the *same* value, that is, valuations used as part of the concretization should map those two nodes to the same value. The Φ should relate these two nodes akin to a unification substitution. Similarly, α_2 and α_2' both denote the value stored in variable x, thus should be related in Φ. On the other hand, node α_6 of abstract state $m_0^\sharp$ has no counterpart in $m_1^\sharp$—it corresponds to a null or non-null address in the region *summarized* by the inductive edge.

We notice Φ can be viewed as a map from nodes in $m_1^\sharp$ to nodes of $m_0^\sharp$ and in this example, defined by $\Phi(\alpha_i') = \alpha_i$ for $0 \leq i \leq 5$. Also, we notice that mapping Φ can be derived step-by-step, starting from the abstract environments. Thus, $\mathfrak{compare}_{\mathrm{shape}}$ and $\mathfrak{compare}_{\mathrm{comb}}$ each take as a parameter a set of pairs of symbolic nodes that should be related in Φ. We call this initial set the *roots*, as they are used as a starting point in the computation of Φ.

We can now describe the steps of computing $\mathfrak{compare}_{\mathrm{mem}}(m_0^\sharp : (E_0^\sharp, (\sigma_0^\sharp, \nu_0^\sharp)), m_1^\sharp : (E_1^\sharp, (\sigma_1^\sharp, \nu_1^\sharp)))$:

1. First, an initial node mapping $\Phi : \mathbb{V}^\sharp[\sigma_1^\sharp] \to \mathbb{V}^\sharp[\sigma_0^\sharp]$ is derived from the abstract environments: $\Phi \stackrel{\text{def}}{=} E_0^\sharp \circ (E_1^\sharp)^{-1}$. This definition states that the addresses of the program variables in $m_1^\sharp$ correspond to the respective addresses of the program variables in $m_0^\sharp$. It is well-defined, as two distinct

variables cannot be allocated at the same physical address.

2. Then, it calls $\mathfrak{compare}_{\text{comb}}(\Phi, (\sigma_0^\sharp, \nu_0^\sharp), (\sigma_1^\sharp, \nu_1^\sharp))$ that forwards to a call of $\mathfrak{compare}_{\text{shape}}(\Phi, \sigma_0^\sharp, \sigma_1^\sharp)$.
3. The abstract heap comparison function $\mathfrak{compare}_{\text{shape}}$ attempts to match $\sigma_0^\sharp$ and $\sigma_1^\sharp$ region-by-region using a set of *local rules*:
   - **(Decomposition)** Suppose $\sigma_0^\sharp$ and $\sigma_1^\sharp$ can be decomposed as $\sigma_0^\sharp = \sigma_{0,0}^\sharp * \sigma_{0,1}^\sharp$ and $\sigma_1^\sharp = \sigma_{1,0}^\sharp * \sigma_{1,1}^\sharp$. And if the corresponding sub-regions can be shown to satisfy the inclusions

     $$\mathfrak{compare}_{\text{shape}}(\Phi, \sigma_{0,0}^\sharp, \sigma_{1,0}^\sharp) = (\textbf{true}, \Phi') \quad \text{and} \quad \mathfrak{compare}_{\text{shape}}(\Phi', \sigma_{0,1}^\sharp, \sigma_{1,0}^\sharp) = (\textbf{true}, \Phi'') \ ,$$

     then the overall inclusion holds—$\mathfrak{compare}_{\text{shape}}(\Phi, \sigma_0^\sharp, \sigma_1^\sharp)$ returns $(\textbf{true}, \Phi'')$;
   - **(Points-to edges)** If $\sigma_0^\sharp = \alpha_0 \cdot \underline{\mathrm{f}} \mapsto \beta_0 * \sigma_{0,r}^\sharp$, $\sigma_1^\sharp = \alpha_1 \cdot \underline{\mathrm{f}} \mapsto \beta_1 * \sigma_{1,r}^\sharp$ and $\Phi(\alpha_1) = \alpha_0$, then we can conclude inclusion holds locally and extend Φ with $\Phi(\beta_1) = \beta_0$;
   - **(Unfolding)** If there is an unfolding of $\sigma_1^\sharp$ called $\sigma_{1,u}^\sharp$ such that $\mathfrak{compare}_{\text{shape}}(\Phi, \sigma_0^\sharp, \sigma_{1,u}^\sharp) = (\textbf{true}, \Phi')$, then $\mathfrak{compare}_{\text{shape}}(\Phi, \sigma_0^\sharp, \sigma_1^\sharp) = (\textbf{true}, \Phi')$.
4. When $\mathfrak{compare}_{\text{shape}}(\Phi, \sigma_0^\sharp, \sigma_1^\sharp)$ succeeds and returns $(\textbf{true}, \Phi')$, it means the inclusion holds with respect to the shape. We, however, still need to check for inclusion with respect to the numeric properties. Recall that the base numeric domain elements $\nu_0^\sharp \in \mathbb{D}_{\text{num}}\langle \mathbb{V}^\sharp[\sigma_0^\sharp]\rangle$ and $\nu_1^\sharp \in \mathbb{D}_{\text{num}}\langle \mathbb{V}^\sharp[\sigma_1^\sharp]\rangle$ have incomparable sets of symbolic variables. An inclusion check in the base numeric domain can only be performed after renaming symbolic names so that they are consistent. The node mapping Φ' computed by the above is precisely the renaming that is needed. Thus, the last step to perform to decide inclusion is to compute $\mathfrak{compare}_{\text{num}}(\nu_0^\sharp, \mathfrak{rename}_{\text{num}}(\Phi', \nu_1^\sharp))$ and return it as a result for $\mathfrak{compare}_{\text{comb}}(\Phi, (\sigma_0^\sharp, \nu_0^\sharp), (\sigma_1^\sharp, \nu_1^\sharp))$. Note that function $\mathfrak{rename}_{\text{num}}$ should be sound in the following sense:

   $$\forall \nu^\sharp \in \mathbb{D}_{\text{num}}\langle V\rangle,\ \forall \nu \in \gamma_{\text{num}}\langle V\rangle(\nu^\sharp),\ (\nu \circ \Phi) \in \gamma_{\text{num}}\langle \Phi(V)\rangle(\mathfrak{rename}_{\text{num}}(\Phi, \nu^\sharp))$$

   where $\Phi(V)$ is the set of symbolic variables obtained by applying Φ to set V.
5. If any of the above steps fail, $\mathfrak{compare}_{\text{mem}}$ returns **false**.

To summarize, the soundness conditions of the inclusion tests for the lower-level domains on which $\mathfrak{compare}_{\text{mem}}$ relies are as follows:

Condition 8 (Soundness of inclusion tests)**.**

1. If $\mathfrak{compare}_{\text{num}}(\nu_0^\sharp, \nu_1^\sharp) = \textbf{true}$,then $\gamma_{\text{num}}\langle V\rangle(\nu_0^\sharp) \subseteq \gamma_{\text{num}}\langle V\rangle(\nu_1^\sharp)$.
2. If $\mathfrak{compare}_{\text{shape}}(\Phi, \sigma_0^\sharp, \sigma_1^\sharp) = (\textbf{true}, \Phi')$,then $(\sigma, \nu \circ \Phi') \in \gamma_{\mathbb{H}}(\sigma_1^\sharp)$ for all $(\sigma, \nu) \in \gamma_{\mathbb{H}}(\sigma_0^\sharp)$.
3. If $\mathfrak{compare}_{\text{comb}}(\Phi, (\sigma_0^\sharp, \nu_0^\sharp), (\sigma_1^\sharp, \nu_1^\sharp)) = (\textbf{true}, \Phi')$,then $(\sigma, \nu \circ \Phi') \in \gamma_{\mathbb{H}^\sharp \rightrightarrows \mathbb{N}^\sharp}(\sigma_1^\sharp, \nu_1^\sharp)$ for all $(\sigma, \nu) \in \gamma_{\mathbb{H}}(\sigma_0^\sharp, \nu_0^\sharp)$.

Returning to the example in Figure 18, after starting with $\Phi = [\alpha_0' \mapsto \alpha_0, \alpha_1' \mapsto \alpha_1]$, the $\mathfrak{compare}_{\text{shape}}$ operation consumes the points-to edges one-by-one extending Φ incrementally, unfolding the inductive edges in the right argument before concluding that inclusion holds in the shape domain. With the final mapping $\Phi'(\alpha_i') = \alpha_i$ for all i, the numeric inclusion simply needs to check that $\mathfrak{compare}_{\text{num}}(\alpha_3 \leq \alpha_5 \wedge \alpha_5 \leq \alpha_7, \mathfrak{rename}_{\text{num}}(\Phi', \alpha_3' \leq \alpha_5')) = \mathfrak{compare}_{\text{num}}(\alpha_3 \leq \alpha_5 \wedge \alpha_5 \leq \alpha_7, \alpha_3 \leq \alpha_5) = \textbf{true}$.

4.5 Join and widening

As is standard, the $\mathfrak{join}_{\text{mem}}$ operation should satisfy the following:

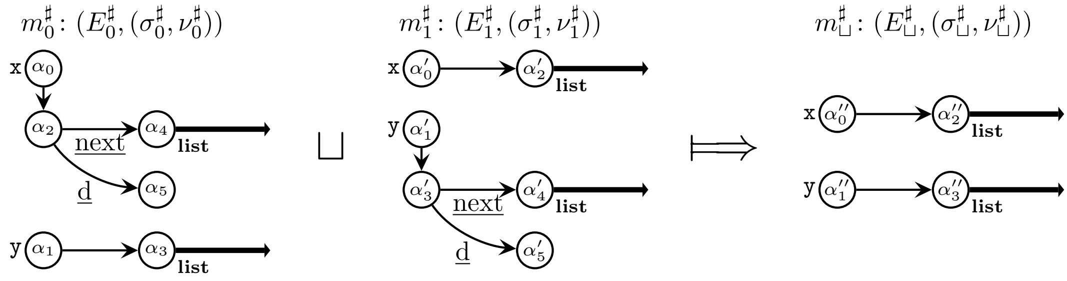


Figure 19: An abstract join showing the need for different sets of symbolic variables for each of the inputs and the result. The inputs are the two possible abstract heaps where a possibly-empty and a non-empty list are pointed to by two non-aliased program variables x and y, so the most precise over-approximation is the abstract heap where both x and y point to two possibly-empty lists.

Condition 9 (Soundness of $\mathfrak{join}_{\mathrm{mem}}$). For all $m_0^\sharp$ and $m_1^\sharp$, $\gamma_{\mathbb{M}}(m_0^\sharp) \cup \gamma_{\mathbb{M}}(m_1^\sharp) \subseteq \gamma_{\mathbb{M}}(\mathfrak{join}_{\mathrm{mem}}(m_0^\sharp, m_1^\sharp))$.

Like the comparison operator, the join operator takes two abstract heaps that have distinct sets of symbolic variables as input. Additionally, it generates a new abstract heap, which requires another set of symbolic variables, as it may not be possible to use the same set as either input. The example shown in Figure 19 illustrates this situation. In left input $m_0^\sharp$, variable x points to a non-empty list and y points to a possibly empty list, whereas right input $m_1^\sharp$ describes the opposite. The most precise over-approximation of $m_0^\sharp$ and $m_1^\sharp$ corresponds to the case where both x and y point to lists of any length (as shown on the right side of the figure). These three elements all have distinct sets of nodes (that cannot be put in a bijection). Thus, the join algorithm uses a slightly different notion of symbolic node mapping Ψ that binds three-tuples of nodes consisting of one node from each parameter and one node in the output abstract heap. Conceptually, the output abstract heap is a kind of product construction, so it is composed of new symbolic variables corresponding to pairs of nodes with one from each input.

Overall, the join algorithm proceeds in a similar way as the inclusion test: the abstract heap join produces a mapping relating symbolic variables along with a new abstract heap. This mapping is then used to rename symbolic variables in the base numeric domain elements consistently to then apply the join in the base domain. Similar to the inclusion test, an initial mapping Ψ is constructed using the abstract environment at the $\mathbb{M}^\sharp$ level and then extended step-by-step at the $\mathbb{H}^\sharp$ level. For instance, in Figure 19, the initial mapping is $\{(\alpha_0, \alpha'_0, \alpha''_0), (\alpha_1, \alpha'_1, \alpha''_1)\}$, and then pairs $(\alpha_2, \alpha'_2, \alpha''_2)$ and $(\alpha_3, \alpha'_3, \alpha''_3)$ are added by $\mathfrak{join}_{\mathrm{shape}}$. Note that nodes $\alpha_3, \alpha_4, \alpha'_3, \alpha'_4$ have no counterpart in the result.

The local rules abstract heap join rules used in $\mathfrak{join}_{\mathrm{shape}}$ belong to two main categories:

- **(Bijection)** When two fragments of each input are isomorphic modulo Ψ, they can be joined into another such fragment. In the example, the points-to edges $\alpha_0 \mapsto \alpha_2$ and $\alpha'_0 \mapsto \alpha'_2$ can both be over-approximated by $\alpha''_0 \mapsto \alpha''_2$. Applying this rule adds the triple $(\alpha_2, \alpha'_2, \alpha''_2)$ to the mapping Ψ.
- **(Weakening)** When a heap fragment can be shown to be included in a more simple, summary fragment (in terms of their concretizations), we can over-approximate the original fragment with the summary. For instance, fragment $\alpha_2 \cdot \underline{\mathrm{next}} \mapsto \alpha_3 * \alpha_2 \cdot \underline{\mathrm{d}} \mapsto \alpha_4 * \alpha_3 \cdot \mathbf{list}$ can be shown to be included in $\alpha_2 \cdot \mathbf{list}$. The other input can be an effective means for directing the choice of possible summary fragments [7, 8].

The widening operator $\mathfrak{widen}_{\mathrm{mem}}$ can be defined similarly to $\mathfrak{join}_{\mathrm{mem}}$. If the heap join rules enforce termination (i.e., $\mathfrak{join}_{\mathrm{shape}}$ can be used as a widening) and $\mathfrak{join}_{\mathrm{num}}$ is replaced with a widening operator $\mathfrak{widen}_{\mathrm{num}}$, the cofibered domain definition guarantees the resulting operator enforces termination [38].

4.6 Disjunctive abstract domain interface

Recall from Sections 3.3 and 4.2 that unfolding returns a finite set of abstract elements interpreted disjunctively and thus justifies the need for a disjunctive abstraction layer—independent of other possible reasons like a desire for path-sensitivity. In this subsection, we describe the interface for a disjunctive abstraction layer $\mathbb{M}^\sharp_\vee$ shown in Figure 20 that sits above the memory layer $\mathbb{M}^\sharp$. The following discussion completes the picture of the abstract domain interfaces (cf., Figure 11). There are two main differences in the interface as compared to the one for $\mathbb{M}^\sharp$. First, the disjunctive abstract domain should provide two additional operations $\mathfrak{partition}_\vee$ and $\mathfrak{collapse}_\vee$ that create and collapse partitions, respectively. A partition represents a disjunctive set of base domain elements. Second, the transfer functions take an additional context information parameter $c \in \mathbb{C}$ that can be used in $\mathbb{M}^\sharp_\vee$ to tag each disjunct with how it arose in the course of the abstract interpretation.

$$\begin{array}{rlcl}
\mathfrak{partition}_\vee : & \mathbb{C} \times \mathscr{P}_{\text{fin}}(\mathbb{M}^\sharp_\vee) & \longrightarrow & \mathbb{M}^\sharp_\vee \\
\mathfrak{collapse}_\vee : & \mathbb{C} \times \mathbb{M}^\sharp_\vee & \longrightarrow & \mathbb{M}^\sharp_\vee \\
\mathfrak{assign}_\vee : & \mathbb{C} \times \mathscr{L}_\mathbb{X} \times \mathscr{E}_\mathbb{X} \times \mathbb{M}^\sharp_\vee & \longrightarrow & \mathbb{M}^\sharp_\vee \\
\mathfrak{guard}_\vee : & \mathbb{C} \times \mathscr{E}_\mathbb{X} \times \mathbb{M}^\sharp_\vee & \longrightarrow & \mathbb{M}^\sharp_\vee \\
\mathfrak{alloc}_\vee : & \mathbb{C} \times \mathscr{L}_\mathbb{X} \times \mathbb{N} \times \mathbb{M}^\sharp_\vee & \longrightarrow & \mathbb{M}^\sharp_\vee \\
\mathfrak{free}_\vee : & \mathbb{C} \times \mathscr{L}_\mathbb{X} \times \mathbb{N} \times \mathbb{M}^\sharp_\vee & \longrightarrow & \mathbb{M}^\sharp_\vee \\
\mathfrak{compare}_\vee : & \mathbb{M}^\sharp_\vee \times \mathbb{M}^\sharp_\vee & \longrightarrow & \mathbb{B} \\
\mathfrak{join}_\vee : & \mathbb{M}^\sharp_\vee \times \mathbb{M}^\sharp_\vee & \longrightarrow & \mathbb{M}^\sharp_\vee \\
\mathfrak{widen}_\vee : & \mathbb{M}^\sharp_\vee \times \mathbb{M}^\sharp_\vee & \longrightarrow & \mathbb{M}^\sharp_\vee
\end{array}$$

Figure 20: Disjunctive abstraction interface.

Condition 10 (Soundness of $\mathfrak{partition}_\vee$ and $\mathfrak{collapse}_\vee$)**.** Let $s^\sharp : \mathbb{M}^\sharp_\vee$ and $S^\sharp : \mathscr{P}_{\text{fin}}(\mathbb{M}^\sharp_\vee)$.

$$\bigcup\{\gamma_\vee(s^\sharp) \mid s^\sharp \in S^\sharp\} \subseteq \gamma_\vee(\mathfrak{partition}_\vee(c, S^\sharp)) \qquad \gamma_\vee(s^\sharp) \subseteq \gamma_\vee(\mathfrak{collapse}_\vee(c, s^\sharp))$$

Note that contexts play no role in the concretization, but operations can use them, for example, to decide which disjuncts to merge using $\mathfrak{join}_{\text{mem}}$ and which disjuncts to preserve.

Transfer functions $\mathfrak{assign}_\vee$, $\mathfrak{guard}_\vee$, $\mathfrak{alloc}_\vee$, and $\mathfrak{free}_\vee$ all follow the same structure. They first call the underlying operation on the memory abstract domain $\mathbb{M}^\sharp$ and then apply the $\mathfrak{partition}_\vee$ partition on the output. For instance, $\mathfrak{assign}_\vee$ is defined as follows while satisfying the expected soundness condition:

$$\begin{array}{ll}
\mathfrak{assign}_\vee(c, loc, exp) \overset{\text{def}}{=} \mathfrak{partition}_\vee(c, \{\mathfrak{assign}_{\text{mem}}(loc, exp, m^\sharp) \mid m^\sharp \in s^\sharp\}) & \text{(definition)} \\
(E, \sigma[\mathscr{L}\llbracket loc \rrbracket(E,\sigma) \leftarrow \mathscr{E}\llbracket exp \rrbracket(E,\sigma)]) \in \gamma_\vee(\mathfrak{assign}_\vee(c, loc, exp, s^\sharp)) & \text{(soundness)}
\end{array}$$

Inclusion ($\mathfrak{compare}_\vee$), join, and widening operations should satisfy the usual soundness conditions. The $\mathfrak{collapse}_\vee$ operator may be used to avoid generating too many disjuncts (and termination of the analysis).

5 A compositional abstract interpreter

In this section, we assemble an abstract interpreter for the language defined in Section 2 using the abstraction set up in Section 3 and the interface of abstract operations described in Section 4.

The abstract semantics of a program p is a function $\llbracket p \rrbracket^\sharp : \mathbb{M}^\sharp_\vee \to \mathbb{M}^\sharp_\vee$, which takes an abstract precondition as input and produces an abstract post-condition as output. Based on an abstract interpretation of the denotational semantics of programs [33, 34], we can define the abstract semantics by induction over the syntax of programs as shown in Figure 21 in a completely standard manner. We let $\mathscr{C}[\ldots]$ stand for computing some context information based on, for example, the control state ℓ and/or the branch taken. This context information may be used, for instance, by the disjunctive domain $\mathbb{M}^\sharp_\vee$ to guide trace partitioning [30]. The abstract transitions for sequencing, assignment, dynamic memory allocation, and deallocation are straightforward with the latter three calling the corresponding transfer function in the top-layer abstract domain $\mathbb{M}^\sharp_\vee$. For **if**, the pre-condition is first constrained by the guard condition via

$$
\begin{array}{rcl}
[\![p_0; p_1]\!]^\sharp(s^\sharp) \stackrel{\text{def}}{=} [\![p_1]\!]^\sharp \circ [\![p_0]\!]^\sharp(s^\sharp) & & [\![\ell : loc = \mathbf{malloc}(n)]\!]^\sharp(s^\sharp) \stackrel{\text{def}}{=} \mathfrak{alloc}_{\mathbb{V}}(\mathscr{C}[\ell], loc, n, s^\sharp) \\
[\![\ell : loc = exp]\!]^\sharp(s^\sharp) \stackrel{\text{def}}{=} \mathfrak{assign}_{\mathbb{V}}(\mathscr{C}[\ell], loc, exp, s^\sharp) & & [\![\ell : \mathbf{free}(loc)]\!]^\sharp(s^\sharp) \stackrel{\text{def}}{=} \mathfrak{free}_{\mathbb{V}}(\mathscr{C}[\ell], loc, n, s^\sharp)
\end{array}
$$

$$
\begin{array}{rcl}
[\![\ell : \mathbf{if}\ (exp)\ p_{\mathrm{t}}\ \mathbf{else}\ p_{\mathrm{f}}]\!]^\sharp(s^\sharp) & \stackrel{\text{def}}{=} & \mathfrak{join}_{\mathbb{V}}([\![p_{\mathrm{t}}]\!]^\sharp(\mathfrak{guard}_{\mathbb{V}}(\mathscr{C}[\ell, \mathbf{true}], exp, s^\sharp)), \\
& & \quad [\![p_{\mathrm{f}}]\!]^\sharp(\mathfrak{guard}_{\mathbb{V}}(\mathscr{C}[\ell, \mathbf{false}], exp = \mathbf{false}, s^\sharp))) \\
[\![\ell : \mathbf{while}\ (exp)\ p]\!]^\sharp(s^\sharp) & \stackrel{\text{def}}{=} & \mathfrak{guard}_{\mathbb{V}}(\mathscr{C}[\ell, \mathbf{false}], exp = \mathbf{false}, \mathbf{lfp}^\sharp_{s^\sharp} F^\sharp) \\
& & \text{where } F^\sharp : \begin{array}{ccl} \mathbb{M}^\sharp_{\mathbb{V}} & \longrightarrow & \mathbb{M}^\sharp_{\mathbb{V}} \\ s^\sharp_0 & \longmapsto & [\![p]\!]^\sharp(\mathfrak{guard}_{\mathbb{V}}(\mathscr{C}[\ell, \mathbf{true}], exp, s^\sharp_0)) \end{array}
\end{array}
$$

Figure 21: A denotational-style abstract interpreter for the programming language defined in Section 2.2.

$\mathfrak{guard}_{\mathbb{V}}$ to interpret the two branches and then the resulting states are joined via $\mathfrak{join}_{\mathbb{V}}$. For **while**, we write $\mathbf{lfp}^\sharp$ for an abstract post–fixed-point operator. The $\mathbf{lfp}^\sharp$ operator relies on $\mathfrak{widen}_{\mathbb{V}}$ to terminate and on $\mathfrak{compare}_{\mathbb{V}}$ to verify the stability of the abstract post-fixed point. It may also use $\mathfrak{join}_{\mathbb{V}}$ to increase the level of precision when computing the first iterations. We omit a full definition of $\mathbf{lfp}^\sharp$ as there are many well-known ways to obtain such an operator. The most simple one consists of applying only $\mathfrak{widen}_{\mathbb{V}}$ until stabilization can be shown by $\mathfrak{compare}_{\mathbb{V}}$. We simply state its soundness condition:

Condition 11 (Soundness of $\mathbf{lfp}^\sharp$)**.** For all concrete transformers $F : \mathscr{P}(\mathbb{S}) \to \mathscr{P}(\mathbb{S})$ monotone, all abstract transformers $F^\sharp : \mathbb{M}^\sharp_{\mathbb{V}} \to \mathbb{M}^\sharp_{\mathbb{V}}$, and all abstract states $s^\sharp \in \mathbb{M}^\sharp_{\mathbb{V}}$,

$$\text{if } F \circ \gamma_{\mathbb{V}} \subseteq \gamma_{\mathbb{V}} \circ F^\sharp \text{ ,then } \mathbf{lfp}_{\gamma_{\mathbb{V}}(s^\sharp)} F \subseteq \gamma_{\mathbb{V}}(\mathbf{lfp}^\sharp_{s^\sharp} F^\sharp) \ .$$

We write $\mathbf{lfp}_S$ for the least post–fixed point that is at least S and similarly for $\mathbf{lfp}^\sharp_{s^\sharp}$. Finally, the static analysis is sound in the following sense:

Theorem 1 (Soundness of the analysis)**.** *Let p be a program, and let $s^\sharp \in \mathbb{M}^\sharp_{\mathbb{V}}$ be an abstract precondition. Then, the result of the analysis is sound:*

$$\forall s \in \gamma_{\mathbb{V}}(s^\sharp),\ [\![p]\!]_{\mathbf{d}}(s) \subseteq \gamma_{\mathbb{V}}([\![p]\!]^\sharp(s^\sharp)) \ .$$

Soundness can be proven by induction over the syntax of programs and by composing the local soundness conditions of all abstract operators.

Related work and discussion. An advantage of this iteration strategy, is that it leads to an intuitive order of application of the abstract equations corresponding to the program [10], eliminating complex iteration strategies [19]. It also simplifies the choice of widening points [4], as it applies widening naturally, at loop heads, though it also allows one to make different choices in strategy by, for example, modifying $\mathbf{lfp}^\sharp$ to unroll loop iterations [3].

6 Conclusion

We have presented a modular construction of a static analysis that is able to reason both about the shape of data structures and their numeric contents simultaneously. Our construction is parametric in the desired numeric abstraction, as well as the shape abstraction, making it possible to continuously substitute improvements for each component or with variants targeted at different classes of programs or even different programming languages. The main advantage of a modular construction is that it allows one to design, prove, and implement each component of the analysis independently. Modular construction is a cornerstone of quality software engineering, and our experience has been that this nice property becomes even more important when dealing with the complexity of creating a static analysis that simultaneously reasons about shape and numeric properties.

Acknowledgments We are inspired by Dave Schmidt's continual effort in formalizing the foundations of static analysis, abstract interpretation, and the semantics of programming languages. We would also wish to thank Dave for being a pillar in the static analysis community and supporting the research community with often thankless, behind-the-scenes work.

This work has been supported in part by the European Research Council under the FP7 grant agreement 278673, Project MemCAD, from the ARTEMIS Joint Undertaking under agreement no 269335 (ARTEMIS Project MBAT) (See Article II.9 of the Joint Undertaking Agreement) and the United States National Science Foundation under grant CCF-1055066.

References

[1] T. Ball, R. Majumdar, T. D. Millstein & S. K. Rajamani (2001): *Automatic Predicate Abstraction of C Programs*. In: *PLDI'01*, pp. 203–213, doi:10.1145/378795.378846.

[2] J. Berdine, C. Calcagno, B. Cook, D. Distefano, P. O'Hearn, T. Wies & H. Yang (2007): *Shape Analysis for Composite Data Structures*. In: *CAV'07*, pp. 178–192, doi:10.1007/978-3-540-73368-3_22.

[3] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux & X. Rival (2003): *A static analyzer for large safety-critical software*. In: *PLDI'03*, pp. 196–207, doi:10.1145/781131.781153.

[4] F. Bourdoncle (1992): *Abstract Interpretation by Dynamic Partitioning*. Journal of Functional Programming 2(4), pp. 407–423, doi:10.1017/S0956796800000496.

[5] B.-Y. E. Chang (2008): *End-User Program Analysis*. Ph.D. thesis, University of California, Berkeley.

[6] B.-Y. E. Chang & R. Leino (2005): *Abstract Interpretation with Alien Expressions and Heap Structures*. In: *VMCAI'05*, pp. 147–163, doi:10.1007/978-3-540-30579-8_11.

[7] B.-Y. E. Chang & X. Rival (2008): *Relational inductive shape analysis*. In: *POPL'08*, pp. 247–260, doi:10.1145/1328438.1328469.

[8] B.-Y. E. Chang, X. Rival & G. Necula (2007): *Shape Analysis with Structural Invariant Checkers*. In: *SAS'07*, pp. 384–401, doi:10.1007/978-3-540-74061-2_24.

[9] D. R. Chase, M. Wegman & F. K. Zadeck (1990): *Analysis of Pointers and Structures*. In: *PLDI'90*, pp. 296–310, doi:10.1145/93542.93585.

[10] P. Cousot & R. Cousot (1977): *Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints*. In: *POPL'77*, pp. 238–252, doi:10.1145/512950.512973.

[11] P. Cousot & R. Cousot (1979): *Systematic Design of Program Analysis Frameworks*. In: *POPL'79*, pp. 269–282, doi:10.1145/567752.567778.

[12] P. Cousot & N. Halbwachs (1978): *Automatic discovery of linear restraints among variables of a program*. In: *POPL'78*, pp. 84–97, doi:10.1145/512760.512770.

[13] I. Dillig, T. Dillig & A. Aiken (2011): *Precise reasoning for programs using containers*. In: *POPL'11*, pp. 187–200, doi:10.1145/1926385.1926407.

[14] D. Distefano, P. O'Hearn & H. Yang (2006): *A Local Shape Analysis Based on Separation Logic*. In: *TACAS'06*, pp. 287–302, doi:10.1007/11691372_19.

[15] D. Gopan, T. W. Reps & M. Sagiv (2005): *A framework for numeric analysis of array operations*. In: *POPL'05*, pp. 338–350, doi:10.1145/1040305.1040333.

[16] S. Gulwani, B. McCloskey & A. Tiwari (2008): *Lifting abstract interpreters to quantified logical domains*. In: *POPL'08*, pp. 235–246, doi:10.1145/1328438.1328468.

[17] N. Halbwachs & M. Péron (2008): *Discovering properties about arrays in simple programs*. In: *PLDI'08*, pp. 339–348, doi:10.1145/1375581.1375623.

[18] T. A. Henzinger, R. Jhala, R. Majumdar & G. Sutre (2002): *Lazy abstraction*. In: *POPL'02*, pp. 58–70, doi:10.1145/503272.503279.

[19] S. Horwitz, A. J. Demers & T. Teitelbaum (1987): *An Efficient General Iterative Algorithm for Dataflow Analysis*. *Acta Informatica* 24(6), pp. 679–694, doi:10.1007/BF00282621. Springer.

[20] B. Jeannet & A. Miné (2009): *Apron: A Library of Numerical Abstract Domains for Static Analysis*. In: *CAV'09*, pp. 661–667, doi:10.1007/978-3-642-02658-4_52.

[21] N. Jones & S. Muchnick (1981): *Flow Analysis and Optimization of LISP-like Structures*. In: *Program Flow Analysis: Theory and Applications*, chapter 4, Prentice-Hall, pp. 102–131.

[22] V. Laviron, B.-Y. E. Chang & X. Rival (2010): *Separating Shape Graphs*. In: *ESOP'10*, pp. 387–406, doi:10.1007/978-3-642-11957-6_21.

[23] S. Magill, J. Berdine, E. Clarke & B. Cook (2007): *Arithmetic Strengthening for Shape Analysis*. In: *SAS'07*, pp. 419–436, doi:10.1007/978-3-540-74061-2_26.

[24] S. Magill, M.-H. Tsai, P. Lee & Y.-K. Tsay (2010): *Automatic numeric abstractions for heap-manipulating programs*. In: *POPL'10*, pp. 211–222, doi:10.1145/1706299.1706326.

[25] B. McCloskey, T. Reps & M. Sagiv (2010): *Statically Inferring Complex Heap, Array, and Numeric Invariants*. In: *SAS'10*, pp. 71–99, doi:10.1007/978-3-642-15769-1_6.

[26] A. Milanova, A. Rountev & B. G. Ryder (2005): *Parameterized object sensitivity for points-to analysis for Java*. *TOSEM* 14(1), pp. 1–41, doi:10.1145/1044834.1044835.

[27] A. Miné (2006): *The octagon abstract domain*. *HOSC* 19(1), pp. 31–100, doi:10.1007/s10990-006-8609-1.

[28] J. Reynolds (2002): *Separation Logic: A Logic for Shared Mutable Data Structures*. In: *LICS'02*, pp. 55–74, doi:10.1109/LICS.2002.1029817.

[29] X. Rival & B.-Y. E. Chang (2011): *Calling context abstraction with shapes*. In: *POPL'11*, pp. 173–186, doi:10.1145/1926385.1926406.

[30] X. Rival & L. Mauborgne (2007): *The trace partitioning abstract domain*. *ACM TOPLAS* 29(5), pp. 26–69, doi:10.1145/1275497.1275501.

[31] M. Sagiv, T. Reps & R. Wilhelm (2002): *Parametric shape analysis via 3-valued logic*. *ACM TOPLAS* 24(3), pp. 217–298, doi:10.1145/514188.514190.

[32] M. Sagiv, T. W. Reps & R. Wilhelm (1998): *Solving Shape-Analysis Problems in Languages with Destructive Updating*. *ACM TOPLAS* 20(1), pp. 1–50, doi:10.1145/271510.271517.

[33] D. A. Schmidt (1986): *Denotational semantics: a methodology for language development*. William C. Brown Publishers, Dubuque, IA, USA.

[34] D. A. Schmidt (2009): *Abstract Interpretation From a Denotational-semantics Perspective*. *ENTCS* 249, pp. 19–37, doi:10.1016/j.entcs.2009.07.082.

[35] M. Sharir & A. Pnueli (1981): *Two approaches to interprocedural data flow analysis*. In: *Program Flow Analysis: Theory and Applications*, chapter 7, Prentice-Hall, pp. 189–233.

[36] P. Sotin & X. Rival (2012): *Hierarchical Shape Abstraction of Dynamic Structures in Static Blocks*. In: *APLAS'12*, pp. 131–147, doi:10.1007/978-3-642-35182-2_10.

[37] A. Toubhans, B.-Y. E. Chang & X. Rival (2013): *Reduced Product Combination of Abstract Domains for Shapes*. In: *VMCAI'13*, pp. 375–395, doi:10.1007/978-3-642-35873-9_23.

[38] A. Venet (1996): *Abstract Cofibered Domains: Application to the Alias Analysis of Untyped Programs.* In: *SAS'96*, pp. 366–382, doi:10.1007/3-540-61739-6_53.