Reduced Product Combination of Abstract Domains for Shapes *

Antoine Toubhans[1], Bor-Yuh Evan Chang[2], and Xavier Rival[1]

[1] INRIA, ENS, CNRS, Paris, France
[2] University of Colorado, Boulder, Colorado, USA
toubhans@di.ens.fr, bec@cs.colorado.edu, rival@di.ens.fr

Abstract. Real-world data structures are often enhanced with additional pointers capturing alternative paths through a basic inductive skeleton (e.g., back pointers, head pointers). From the static analysis point of view, we must obtain several interlocking shape invariants. At the same time, it is well understood in abstract interpretation design that supporting a separation of concerns is critically important to designing powerful static analyses. Such a separation of concerns is often obtained via a reduced product on a case-by-case basis. In this paper, we lift this idea to abstract domains for shape analyses, introducing a domain combination operator for memory abstractions. As an example, we present *simultaneous separating shape graphs*, a product construction that combines instances of separation logic-based shape domains. The key enabler for this construction is a static analysis on inductive data structure definitions to derive relations between the skeleton and the alternative paths. From the engineering standpoint, this construction allows each component to reason independently about different aspects of the data structure invariant and then separately exchange information via a reduction operator. From the usability standpoint, we enable describing a data structure invariant in terms of several inductive definitions that hold simultaneously.

1 Introduction

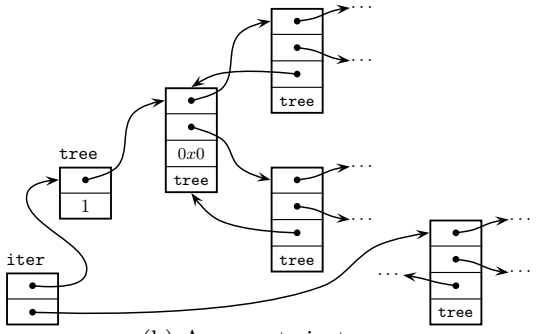
Shape analyses aim at inferring precise and sound invariants about programs manipulating complex data structures so as to prove safety and functional properties [20,10,2,5]. Such data structures typically mix several independent characteristics, which makes abstraction hard. For instance, an extreme case would be that of a binary tree with parent pointers, satisfying a balancing property and sortedness of data fields, and the property that all nodes contain a pointer to some shared record. Some shape analysis tools rely on a hard-coded abstraction [10,2], while others require some specialization of a generic abstract

* The research leading to these results has received funding from the European Research Council under the FP7 grant agreement 278673, Project MemCAD and the United States National Science Foundation under grant CCF-1055066.

VMCAI 2013.

```
typedef struct node {
    struct node * l;
    struct node * r;
    struct node * p;
    struct tree * h;
} * node;
typedef struct tree {
    struct node * r;
    int i;
} * tree;
typedef struct iterator {
    struct tree * t;
    struct node * c;
} * iter;
```

(a) Type definitions



(b) A concrete instance

Fig. 1. A complex tree structure

domain, either via user-supplied instrumentation predicates or inductive definitions [20,5]. In either case, when the data structures of interest feature many independent characteristics as in the aforementioned example, the user-supplied specifications or hard-coded abstractions should ideally reflect some level of *separation of concerns*, for example, between balancing properties, parent pointers, and properties over keys.

Intuitively, a separation of concerns in a shape analysis means that it should be possible for conceptually independent attributes of a data structure to be understood and treated separately by both the tool and the tool user. As an example, Fig. 1 shows an iterator over a binary tree with parent pointers. The `node` data-type (Fig. 1(a)) features left (`l`), right (`r`), and parent (`p`) pointers, as well as a special field `h` pointing to an enclosing record; the purpose of this record is to keep track of the root node of the whole structure (field `r`) and the number of active iterators (field `i`). Iterator `iter` encloses pointers to the structure being iterated over (field `t`) and to the current position (field `c`). All nodes satisfy two independent invariants: (1) `p` fields point to their parent and (2) `h` fields point to the enclosing record `tree`.

Reduced product [8] of abstract domains [7] is a framework achieving a separation of concerns in a static analyzer by combining several abstractions $\mathbb{D}_0, \ldots, \mathbb{D}_k$ and expressing conjunctions of abstract properties of the form $p_0 \wedge \ldots \wedge p_k$, where p_i is an abstract element of abstract domain $\mathbb{D}_i$. This construction has been abundantly used to combine *numerical* abstract domains into more expressive ones without having to design a monolithic domain that would be overly complex to set up and implement. For example, the Astrée analyzer [3] has been built as such a combination of numeric and symbolic abstract domains [9]; reduced product is implemented as a generic operation over abstract domains. While shape analyses often decompose abstract properties using *separating* conjunction [10,2,5] introduced in [19], few analyses explicitly introduce *non-separating* conjunction, as this operation is viewed as more complex: for instance, updates need be analyzed for all conjunction elements. Non-separating conjunctions tend to be used in a local manner in order to capture co-existing views over small

blocks [18,14]. The work presented in [15] uses global conjunctions (in addition to other refinements), yet does not turn it into a general abstract domain operation.

In this paper, we present the following contributions:

- We set up a framework for reduced product of memory abstractions in Sect. 3.
- We instantiate our framework to separating shape graphs in Sect. 4.
- We extend this instantiation to cope with user-supplied inductive definitions in Sect. 5; in that case, the reduction functions use information computed by static analysis of the inductive definitions.

Moreover, the reduced product combinator was implemented in the MemCAD analyzer (http://www.di.ens.fr/~rival/memcad.html), and experiments on reduction strategies are reported in Sect. 6.

2 Analysis of an iterator over a tree with parent pointers

In this section, we overview the abstraction of the structure shown in Fig. 1 using conjunctions of simpler properties and how an iteration procedure, with imperative update, can be analyzed. For simplicity in presentation, we assume variables `tree` and `iter` have global scope. Variable `tree` points to the header structure of type `tree`, while `iter` points to an iterator of type `iterator`. The tree structure can be captured by an inductive definition that can be given as a parameter to a parametric abstract domain such as that of Xisa [5,6]. For instance, the inductive definition below captures both the tree structure, the consistency of the parent pointers, and the fact that each node is separated, as materialized by the separating conjunction operator applied between fields and inductive calls:

$$\begin{array}{rll} \alpha \cdot \iota_{\mathrm{p}}(\beta) ::= & \alpha = 0 \wedge \mathbf{emp} \\ & \bigvee \alpha \neq 0 \wedge \left(\begin{array}{c} \alpha \cdot \mathtt{l} \mapsto \delta_l * \alpha \cdot \mathtt{r} \mapsto \delta_r * \alpha \cdot \mathtt{p} \mapsto \beta * \alpha \cdot \mathtt{h} \mapsto \_ \\ * \, \delta_l \cdot \iota_{\mathrm{p}}(\alpha) * \delta_r \cdot \iota_{\mathrm{p}}(\alpha) \end{array} \right) \end{array}$$

The parameter β captures the address to which the `p` field of the α parameter should point. However, field `h` is not constrained, and the property that all `h` fields should point to enclosing `tree` record is not captured by this definition. The inductive definition below captures the property that all `h` should point to some given address represented by ϵ (whereas it ignores the parent pointer constraint on `p` fields):

$$\begin{array}{rll} \alpha \cdot \iota_{\mathrm{h}}(\epsilon) ::= & \alpha = 0 \wedge \mathbf{emp} \\ & \bigvee \alpha \neq 0 \wedge \left(\begin{array}{c} \alpha \cdot \mathtt{l} \mapsto \delta_l * \alpha \cdot \mathtt{r} \mapsto \delta_r * \alpha \cdot \mathtt{p} \mapsto \_ * \alpha \cdot \mathtt{h} \mapsto \epsilon \\ * \, \delta_l \cdot \iota_{\mathrm{h}}(\epsilon) * \delta_r \cdot \iota_{\mathrm{h}}(\epsilon) \end{array} \right) \end{array}$$

We can express the fact that α represents the address of a pointer to a node of the structure of Fig. 1 by $\alpha \cdot \iota_{\mathrm{p}}(\beta) \wedge \alpha \cdot \iota_{\mathrm{h}}(\epsilon)$ where β and ϵ represent the address of the parent pointer and of the enclosing record, respectively. We shall express such conjunctions as pairs of shape graphs, using a *product* [8] abstraction.

```
void next(){
  if (iter → c has a child) {
    iter → c = left child
  } else {
    go up to the first node with
      a not yet visited right child
    iter → c = that right child
  }
}
```

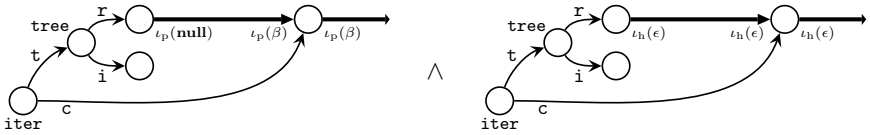
```
void replace(node n) {
  n → l = iter → c → l;
  n → r = iter → c → r;
  n → p = iter → c → p;
  n → h = iter → c → h;
  if (iter → c has a parent){
    update it to point n where iter → c is
  } else { iter → t → r = n; }
}
```

```
void main() {
  node n = malloc(sizeof(struct node));
  while (iter → c ≠ null && ...) { next(); }
  if(iter → t → i == 1) { replace(n); }
}
```

Fig. 2. An iteration using an iterator followed by a node replacement.

In the following, we consider the C program shown in Fig. 2. The code is simplified by summarizing some parts with pseudo-code. Function `main` uses the iterator to walk up to a randomly defined point of the structure, by making a series of left, right, and up steps determined by the tree structure. When no other iterator is active on `iter` → `t`, then the replacement can be performed safely, by updating fields of the node being inserted in the structure, and also fields in the parent/enclosing record depending on the position of the iterator. Note that this code reads and updates both `h` and `p` fields.

An invariant at the exit of the loop in function `main` can be represented by a conjunction of shape graphs as shown in the figure below:



The left and right shape graphs can be expressed in the abstract domain of [5], individually using only ι_p and ι_h as domain parameters, respectively. Edges abstract pairwise disjoint memory regions. Points-to edges (marked as thin edges) describe individual cells Thick edges inductively abstract (potentially empty) summarized memory regions, which could be either full structures or structure *segments*: for instance, the left shape graph expresses that field `iter`→ `c` points to a node of the tree pointed to by `tree` → `r`.

While both components of that invariant can be expressed as a shape graph in the abstract domain of [5], it is not possible to *infer* either without reasoning about parent pointers, as function `next` may follow unbounded upward paths in the tree. Similarly, the preservation of structural invariants in `replace` requires reasoning about both `p` and `h`. However, ι_p ignores information about `h` and vice versa for ι_h; thus, neither component can perform all those steps on its own. Therefore the product analysis must organize *information exchange* among both components, which corresponds to a *reduction* operation. For instance, we

consider assignment $\texttt{iter} \to \texttt{c} = \texttt{iter} \to \texttt{c} \to \texttt{p}$ in `next`. In the right component (ι_h), no information about `p` is known so the analysis of this operation would lose all precision when it fails to materialize this field, whereas the left component (ι_p) will materialize `p` with no loss in precision. The left component will also come up with the property that when $\texttt{iter} \to \texttt{c}$ has a parent, then either $\texttt{iter} \to \texttt{c} \to \texttt{p} \to \texttt{l}$ or $\texttt{iter} \to \texttt{c} \to \texttt{p} \to \texttt{r}$ is equal to $\texttt{iter} \to \texttt{c}$ (i.e., the current node is a child of its parent) that would allow a precise materialization in the right component. Similar cases occur in the analysis of `replace`. To conclude, we need to set up a language to express such simple constraints, to design operators to extract them, and to constrain abstract values with them, and to identify when such information exchange should be performed.

3 Interfaces for memory abstractions

In this section, we layout our framework for defining reduced products of memory abstractions. Our goal is to define a flexible abstract domain combination operator, so we begin by defining a generic interface that we expect memory abstractions to implement.

Concrete memories. We use a direct model of concrete memories m, which consist of an environment and a heap (shown inset). A *concrete environment* e is a finite map from program variables to values. A *concrete heap* h is as a finite map from addresses to values. We assume that the set of addresses is a subset of the set of values (i.e., $\mathbb{A} \subseteq \mathbb{V}$). Fields $\texttt{f}, \texttt{g}, \ldots$ are treated as numerical offsets where we write $a + \texttt{f}$ for the address that is an offset `f` from base address a (i.e., $(a + \texttt{f}) \in \mathbb{A}$).

memories	$\mathbb{M} \ni m ::= (e, h)$
environments	$\mathbb{E} \ni e : \mathbb{X} \rightharpoonup_\text{fin} \mathbb{V}$
heaps	$\mathbb{H} \ni h : \mathbb{A} \rightharpoonup_\text{fin} \mathbb{V}$
variables	$x \in \mathbb{X}$
values	$v \in \mathbb{V}$
addresses	$a \in \mathbb{A}$
fields	$\texttt{f}, \texttt{g}, \ldots \in \mathbb{F}$

3.1 Memory abstract domains

An *abstract memory state* $m^\sharp$ describes a set of concrete memory states (shown inset). As such, it should abstract both heap addresses along with stored values. To abstract addresses and values, we let $\mathbb{V}^\sharp = \{\alpha, \beta, \ldots\}$ be a set of symbolic variables. An *abstract environment* $e^\sharp \in \mathbb{E}^\sharp = \mathbb{X} \to \mathbb{V}^\sharp$ then maps each variable into an abstraction of its address. To express the consistency between an abstract environment and a concrete environment, we need a *valuation* ν that relates an abstract address to a concrete one. An abstract heap

memories	$m^\sharp ::= (e^\sharp, h^\sharp)$
environments	$e^\sharp : \mathbb{X} \rightharpoonup_\text{fin} \mathbb{V}^\sharp$
heaps	$\mathbb{D}_\mathbf{s} \ni h^\sharp ::= \bot \mid \cdots$
concretization	$\gamma_\mathbf{s} : \mathbb{D}_\mathbf{s} \to \mathcal{P}(\mathbb{H} \times \mathcal{V})$
valuations	$\nu \in \mathcal{V}$
assignment	$\mathfrak{assign} : \mathbf{lvals} \times \mathbf{exprs} \times \mathbb{D}_\mathbf{s} \to \mathbb{D}_\mathbf{s}$
conditional	$\mathfrak{guard} : \mathbf{exprs} \times \mathbb{D}_\mathbf{s} \to \mathbb{D}_\mathbf{s}$
widening	$\triangledown : \mathbb{D}_\mathbf{s} \times \mathbb{D}_\mathbf{s} \to \mathbb{D}_\mathbf{s}$

$h^\sharp ::=$		*abstract heaps*	$\gamma_{\mathbf{s}}(h^\sharp)$
	emp	empty heap	$\{\,([\,], \nu)\,\}$
$\mid$	$\alpha \cdot \mathtt{f} \mapsto \beta$	single cell	$\{\,([\nu(\alpha) + \mathtt{f} \mapsto \nu(\beta)], \nu)\,\}$
$\mid$	$h_0^\sharp * h_1^\sharp$	disjoint regions	$\{\,(h_0 \uplus h_1, \nu) \mid (h_0, \nu) \in \gamma_{\mathbf{s}}(h_0^\sharp)$ and $(h_1, \nu) \in \gamma_{\mathbf{s}}(h_1^\sharp)\,\}$
$\mid$	$h^\sharp \wedge P$	with constraint	$\{\,(h, \nu) \mid (h, \nu) \in \gamma_{\mathbf{s}}(h^\sharp)$ and $\nu \models P\,\}$
$P ::=$		*pure predicates*	$\nu \models P$
	$\alpha = 0 \mid \alpha \neq 0 \mid P_0 \wedge P_1 \mid \ldots$		

Fig. 3. A shape abstract domain based on exact separating shape graphs.

$h^\sharp ::=$	*abstract heaps*	$\gamma_{\mathbf{s}}(h^\sharp)$
$\{\,\alpha \cdot \mathtt{f} \Rrightarrow \beta, \ldots\,\}$	set of may points-to	$\{\,(h, \nu) \mid a + f \mapsto v \in h$ implies $\alpha \cdot f \Rrightarrow \beta \in h^\sharp$ and $a \in \nu(\alpha)$ and $v \in \nu(\beta)\,\}$

Fig. 4. A shape abstract domain based on points-to graphs.

$h^\sharp$ expresses pointer relationships between abstract addresses, so it abstracts a set of concrete-heap–valuation pairs. A *shape abstract domain* is a set $\mathbb{D}_{\mathbf{s}}$ of abstract heaps, together with a concretization function $\gamma_{\mathbf{s}}$ and sound abstract operators. Among the abstract operators, we include operators to compute abstract post-conditions, such as $\mathfrak{assign}$ for assignments and $\mathfrak{guard}$ for conditional guards. We also include a widening operator $\triangledown$ that joins abstract states while enforcing termination of abstract iteration [7]. All of these operators are required to satisfy the usual soundness conditions—that is, they ensure that no concrete behavior will be lost in the abstract interpretation. For example, the widening operator $\triangledown$ should soundly over-approximate unions of concrete states (i.e., $\gamma_{\mathbf{s}}(h_0^\sharp) \cup \gamma_{\mathbf{s}}(h_1^\sharp) \subseteq \gamma_{\mathbf{s}}(h_0^\sharp \triangledown h_1^\sharp)$ for all abstract heaps $h_0^\sharp$ and $h_1^\sharp$). We also let $\bot$ denote the least element of the memory abstraction, which should have the empty concretization (i.e., $\gamma_{\mathbf{s}}(\bot) = \emptyset$).

Example 1 (Exact separating shape graphs). In Fig. 3, we describe exact separating shape graphs, which is a memory abstraction with no summaries. We consider separating shape graphs with inductive summaries in Sect. 5. An abstract heap $h^\sharp$ is a formula syntactically formed according to the given grammar. We define the concretization of $h^\sharp$ inductively on the formula structure in the rightmost column. Intuitively, an abstract heap is simply a finite separating conjunction [19] of must points-to predicates along with pure constraints over heap values. The formula **emp** is the abstract heap corresponding to the concrete empty heap $[\,]$. Thus, notice $\gamma_{\mathbf{s}}(\mathbf{emp})$ is independent of the choice of valuation ν. A must points-to $\alpha \cdot \mathtt{f} \mapsto \beta$ corresponds to a singleton heap whose address and contents are given by the valuation ν. Here, we let valuations be functions from symbolic variables to concrete values (i.e., $\nu : \mathbb{V}^\sharp \to \mathbb{V}$). As usual, the formula $h_0^\sharp * h_1^\sharp$ joins disjoint abstract sub-heaps. In the concrete, we write $h_0 \uplus h_1$ to join sub-heaps if their *domains* are disjoint (and undefined otherwise). Finally,

the formula $h^\sharp \wedge P$ conjoins a pure constraint; we write $\nu \models P$ to say valuation ν semantically entails pure predicate P.

Example 2 (May points-to graphs). As another example, we formalize a memory abstraction based on points-to graphs as one obtains from Andersen's analysis [1] in our framework (Fig. 4). An abstract heap $h^\sharp$ is a set of may points-to edges of the form $\alpha \cdot f \Rightarrow \beta$. For this abstraction, an abstract location α corresponds to a set of concrete addresses (thus, $\nu : \mathbb{V}^\sharp \to \mathcal{P}(\mathbb{V})$). A concrete heap h is in the concretization of an abstract heap $h^\sharp$ if and only if for all concrete cells $a+f \mapsto v$ in h, there exists a corresponding may points-to edge in the abstract heap $h^\sharp$ as given by the valuation ν. In the literature, there are usually some additional constraints placed on abstract locations. These restrictions on abstract locations can be reflected as constraints on valuations ν, such as non-empty corresponding concrete addresses (i.e., $|\nu(\alpha)| \geq 1$ for all $\alpha \in \mathrm{dom}(\nu)$) and disjoint abstract locations (i.e., $\nu(\alpha) \cap \nu(\beta) = \emptyset$ for all $\alpha, \beta \in \mathrm{dom}(\nu)$). Sometimes, abstract locations are also classified as *non-summary* versus *summary*. A non-summary abstract location α means we restrict ν such that $|\nu(\alpha)| = 1$. In contrast to exact separating shape graphs, may points-to graphs can never give precise information about the *presence* of a cell. For example, observe that the empty concrete heap $[\,]$ is in the concretization of all may points-to graphs.

3.2 Products of memory abstractions

Shape domains implementing the interface described in Sect. 3.1 can be combined into *product abstractions* in a straightforward manner. Let us assume two shape abstract domains $\mathbb{D}_0$ and $\mathbb{D}_1$ are given. Then, $\mathbb{D}_\times \stackrel{\text{def}}{=} \mathbb{D}_0 \times \mathbb{D}_1$ has a concretization function $\gamma_\times : \mathbb{D}_\times \to \mathcal{P}(\mathbb{H}^\sharp \times \mathcal{V})$ defined by $(h^\sharp, \nu) \in \gamma_\times(h_0^\sharp, h_1^\sharp) \iff (h^\sharp, \nu) \in \gamma_0(h_0^\sharp) \wedge (h^\sharp, \nu) \in \gamma_1(h_1^\sharp)$. This amounts to expressing *non-separating conjunctions* of abstract predicates of $\mathbb{D}_0$ and $\mathbb{D}_1$.

A direct implementation of the abstract operators ($\mathfrak{assign}, \mathfrak{guard}, \triangledown, \ldots$) can be obtained by composing the underlying operators pair-wise. However, the resulting analysis will not take advantage of the information available into one abstract domain in order to refine the facts in the other domain.

To overcome that limitation, we now propose to extend the product abstract domain with a *reduction operation*. A classical (and trivial) reduction operator would map, for example, $(\bot, h_1^\sharp)$ into $(\bot, \bot)$. In this paper, we describe much more powerful reduction operators for memory abstractions that allow us to transfer non-trivial information from one shape abstract domain to another (and vice versa).

To extend domain $\mathbb{D}_0 \times \mathbb{D}_1$ into a reduced product domain $\mathbb{D}_\bowtie \stackrel{\text{def}}{=} \mathbb{D}_0 \bowtie \mathbb{D}_1$, we need to augment it with a *reduction operator* $\pi : \mathbb{D}_\bowtie \to \mathbb{D}_\bowtie$, which satisfies the following soundness condition: $\forall h_0^\sharp \in \mathbb{D}_0,\ h_1^\sharp \in \mathbb{D}_1,\ \gamma_\times(h_0^\sharp, h_1^\sharp) \subseteq \gamma_\times(\pi(h_0^\sharp, h_1^\sharp))$. To implement a reduction operator π, we need be able to extract information from one domain and forward that information as a constraint into

constraints	$\mathcal{F} \subseteq \mathbb{P}_f$
extract	$\mathfrak{extract} : \mathbb{D}_\mathbf{s} \to \mathcal{P}(\mathbb{P}_f)$
constrain	$\mathfrak{constrain} : \mathbb{D}_\mathbf{s} \times \mathcal{P}(\mathbb{P}_f) \to \mathbb{D}_\mathbf{s}$

the other. Thus, we need to set up a language of constraints $\mathbb{P}_f$ and to extend the abstract domain interface with operators $\mathfrak{extract}$ to extract constraints from an abstract value and $\mathfrak{constrain}$ to constrain an abstract value with a constraint. The language $\mathbb{P}_f$ must be the same for *every* abstract shape domains. In practice, the user has to provide an implementation of those two operators in order to be able to use our framework. Given such operators, a reduction from the left domain into the right domain, for example, is defined as follows:

$$\pi_{0\to 1}(h_0^\sharp, h_1^\sharp) \stackrel{\text{def}}{=} (h_0^\sharp, \mathfrak{constrain}_1(h_1^\sharp, \mathfrak{extract}_0(h_0^\sharp))) \ .$$

We subscript the operators to make explicit the domain to which they belong. To specify the soundness requirements, we need a concretization relation for constraints. We write $h, \nu \models \mathcal{F}$ when the pair (h, ν) satisfies all the constraints $\mathcal{F}$ (i.e., we interpret a set of constraints conjunctively). We can now specify the soundness conditions for the domain operators $\mathfrak{extract}$ and $\mathfrak{constrain}$: for all $h^\sharp \in \mathbb{D}_\mathbf{s}$ and all $\mathcal{F} \in \mathcal{P}(\mathbb{P}_f)$,

$$\forall (h, \nu) \in \gamma_\mathbf{s}(h^\sharp), \quad h, \nu \models \mathfrak{extract}(h^\sharp) \tag{1}$$

$$\forall (h, \nu) \in \gamma_\mathbf{s}(h^\sharp), \quad h, \nu \models \mathcal{F} \implies (h, \nu) \in \gamma_\mathbf{s}(\mathfrak{constrain}(h^\sharp, \mathcal{F})) \tag{2}$$

Under these soundness conditions, the operator $\pi_{0\to 1}$ defined above is sound (and similarly for the analogous operator $\pi_{0\leftarrow 1}$). In the above, we have focused on the soundness requirements of these operators and set up a framework for discussing them. While any sequence of $\mathfrak{extract}$ and $\mathfrak{constrain}$ satisfying these properties would yield a sound result, we have not yet discussed how to do so efficiently. And in practice, these operations must be carefully crafted to transfer just the necessary information, and we must apply them parsimoniously or on-demand to avoid making the analysis overly expensive and cluttering abstract values with facts that are not actually useful for the analysis [9]. To do so requires considering specific instantiations of this framework.

4 Instantiation to separating shape graph abstractions

In this section, we consider a first instantiation of our framework for defining reduced products of memory abstract domains. We focus on a product of two instances of $\mathbb{D}_\mathbf{g}$. While this example may look overly simple at first, it illustrates a large part of the issues brought up by the analysis discussed in Sect. 2. For the moment, inductive predicates are fully unfolded and thus not present (issues specific to inductive predicates are discussed in Sect. 5). Let us consider the abstract conjunction shown in Fig. 5. While expression $\mathtt{x}\to\mathtt{l}\to\mathtt{p}\to\mathtt{h}\to\mathtt{i}$ cannot be evaluated in either component of such an abstract state, all concrete memories corresponding to their conjunction would allow that expression to evaluate, as all such concrete memories would let α and δ represent the *same* address. In this section, we show how reduction allows us to perform such reasoning and to strengthen the abstract heaps of Fig. 5.

Fig. 5. A simple reduction example

$p ::=$		paths
	$\mid \emptyset$	empty path
	$\mid \mathtt{f} \quad (\in \mathbb{F})$	single field
	$\mid p \cdot p$	concatenation
$a ::=$		formulas ($a \in \mathbb{P}_f$)
	$\mid \alpha \cdot p \triangleright \beta$	path equality
	$\mid \alpha \cdot p \triangleright \mathbf{null}$	path to null

(a) Syntax

$$
\begin{array}{l}
h, \nu \models \alpha \cdot \emptyset \triangleright \mathbf{null} \iff \nu(\alpha) = 0 \\
h, \nu \models \alpha \cdot \emptyset \triangleright \beta \iff \nu(\alpha) = \nu(\beta) \\
h, \nu \models \alpha \cdot \mathtt{f} \triangleright \mathbf{null} \iff h(\nu(\alpha) + \mathtt{f}) = 0 \\
h, \nu \models \alpha \cdot \mathtt{f} \triangleright \beta \iff h(\nu(\alpha) + \mathtt{f}) = \nu(\beta) \\
h, \nu \models \alpha \cdot p_0 \cdot p_1 \triangleright \bar{\beta} \text{ (where } \bar{\beta} \in \{\mathbf{null}\} \cup \mathbb{V}^\sharp) \\
\quad \iff \exists \delta \in \mathbb{V}^\sharp, \begin{cases} h, \nu \models \alpha \cdot p_0 \triangleright \delta \\ \wedge\ h, \nu \models \delta \cdot p_1 \triangleright \bar{\beta} \end{cases}
\end{array}
$$

(b) Semantics

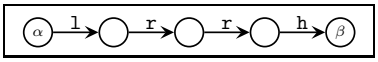
Fig. 6. Language of path constraints for reducing between separating shape graphs

4.1 A language of constraints based on path predicates

First of all, we notice that if (h, ν_0) belongs to the concretization of the left component, then $h(h(\nu_0(\alpha) + \mathtt{l}) + \mathtt{p}) = \nu_0(\alpha)$, or equivalently, that dereferencing `l` and then `p` from address α gets us back to α. In the right component, the same sequence of dereferences yields δ: if (h, ν_1) belongs to the concretization of the right component, then $h(h(\nu_1(\alpha) + \mathtt{l}) + \mathtt{p}) = \nu_1(\delta)$. This suggests that enforcing a simple path reachability equation computed in the left component into the right component will allow us to conclude the equality of α and δ, that is, that they represent the same concrete values and can be merged.

Therefore, we include in the language of constraints to be used for reduction a way to express path predicates of the form $\alpha \cdot \mathtt{f}_0 \cdot \ldots \cdot \mathtt{f}_n \equiv \beta$ (which we write down as $\alpha \cdot \mathtt{f}_0 \cdot \ldots \cdot \mathtt{f}_n \triangleright \beta$). A *path* p is defined by a possibly empty sequence of fields, denoting field dereferences from a node, as shown in Fig. 6(a). A *path formula* a expresses that a series of dereferences described by a path will lead to read either the value denoted by some node β ($\alpha \cdot p \triangleright \beta$) or the null value ($\alpha \cdot p \triangleright \mathbf{null}$). Thus, we obtain the set $\mathbb{P}_f$ defined in Fig. 6(a). Their concretization is defined in Fig. 6(b).

As an example, path formula $\alpha \cdot \mathtt{l} \cdot \mathtt{r} \cdot \mathtt{r} \cdot \mathtt{h} \triangleright \beta$ expresses that β can be reached from α after dereferencing fields `l`, `r`, `r`, and `h` in that order, as shown in the inset figure (not all fields are represented). Intuitively path formulas of the form $\alpha \cdot p \triangleright \beta$ allow us to express *reachability* properties, as commonly used in, for example, TVLA [20].



4.2 Reduction operators for shape graphs

At this stage, we need to set up operators $\mathfrak{extract}$ and $\mathfrak{constrain}$ for the shape graph abstract domain mentioned in Sect. 3.1 using the path language of Sect. 4.1.

Triggering of the reduction process. As discussed earlier, reduction should be performed only when needed [9]. In the case of the product of shape abstract domains, information may need to be sent from one domain to another when a memory cell update or read cannot be analyzed due to a lack of pointer information:

- to read l-value $\alpha \cdot \mathtt{f}$, we need to find an edge of the form $\alpha \cdot \mathtt{f} \mapsto \beta$ in *at least one* of the components of the product; then this domain ensures the existence of the cell (even if the others fail to materialize such an edge);
- to update l-value $\alpha \cdot \mathtt{f}$, we need to find such a points-to edge in *all* components of the product; indeed, if it was not possible to do so in $\mathbb{D}_i$, then, conservative analysis of the update would require dropping *all* the information available in $\mathbb{D}_i$ since any cell may be modified from $\mathbb{D}_i$'s point of view; this would be an unacceptable loss in precision.

Therefore, a on-demand reduction strategy is to trigger when either *one* of the components fails to materialize an edge for an update or when *all* components fail to materialize an edge for a read (in the same way as for unfolding in [6]). While this is the basis of a *minimal* reduction strategy, more aggressive strategies can be used such as:

- an *on-read* strategy which attempts to reduce path constraints about any node involved in a read operation;
- a *maximal* strategy which attempts to reduce all available path constraints about all nodes, at all times.

An empirical evaluation of these strategies will be presented in Sect. 6.

Computation of path information. Operator $\mathfrak{extract} : \mathbb{D}_{\mathbf{s}} \longrightarrow \mathcal{P}(\mathbb{P}_f)$ should extract *sound* sets of constraints, satisfying soundness property (1). In the case of abstract domain $\mathbb{D}_{\mathbf{g}}$, $\mathfrak{extract}(h^\sharp)$ could simply collect all predicates of the form $\alpha \cdot \mathtt{f} \triangleright \beta$ where predicate $\alpha \cdot \mathtt{f} \mapsto \beta$ appears in $h^\sharp$:

$$\mathfrak{extract}(\alpha_0 \cdot \mathtt{f}_0 \mapsto \beta_0 * \ldots * \alpha_q \cdot \mathtt{f}_q \mapsto \beta_q) = \{\alpha_i \cdot \mathtt{f}_i \triangleright \beta_i \mid 0 \leq i \leq q\}$$

Collecting all such constraints would be almost certainly too costly, as it leads to exporting all information available in $h^\sharp$. Since our reduction reduction is triggered by a materialization operation, we only care about finding some field from some node α in either component of the product. Only constraints locally around this node are relevant, which restricts what needs to be exported. Thus, in practice, $\mathfrak{extract}$ and $\mathfrak{constrain}$ are computed locally. In the example of Fig. 5, constraint $\alpha \cdot \mathtt{l} \cdot \mathtt{p} \triangleright \alpha$ can be extracted from the left conjunct.

Enforcing path information. If $\alpha \cdot p \triangleright \beta$ and $\alpha \cdot p \triangleright \gamma$, then β and γ denote the same concrete value, hence these two nodes can be merged. This rule ($\mathbf{R}_{\triangleright\equiv}$) forms the basis of an operator $\mathfrak{constrain}$ satisfying soundness property (2). In the example of Fig. 5, this operator allows us to merge nodes α and δ in the right component, thanks to constraint $\alpha \cdot \mathtt{l} \cdot \mathtt{p} \triangleright \alpha$ inferred by $\mathfrak{extract}$ in the left component, as shown in the previous paragraph. This reduction allows us to materialize $\mathtt{x} \to \mathtt{l} \to \mathtt{p} \to \mathtt{h} \to \mathtt{i}$.

$h^\sharp_\iota := h^\sharp$ (Fig. 3)	points-to, **emp**	• if $h^\sharp \in \mathtt{D}_{\mathbf{g}}$, $\gamma_{\langle\iota\rangle}(h^\sharp) = \gamma_{\mathbf{g}}(h^\sharp)$
$\mid\ h^\sharp_\iota * h^\sharp_\iota$	separating conj.	• if $(h,\nu) \in \gamma_{\langle\iota\rangle}(h^\sharp_u)$
$\mid\ \alpha \cdot \iota(\beta)$	inductive predicate	and $h^\sharp \rightsquigarrow h^\sharp_u$
$\mid\ \alpha \cdot \iota(\beta) \mathrel{-\!\!*\!\!=} \alpha' \cdot \iota(\beta')$	segment predicate	then $(h,\nu) \in \gamma_{\langle\iota\rangle}(h^\sharp)$
(a) Abstract heaps in $\mathtt{D}_{\langle\iota\rangle}$		(b) Concretization $\gamma_{\langle\iota\rangle}$

Fig. 7. A shape abstract domain based on separating shape graphs

5 Instantiation to separating shape graphs with inductive summaries

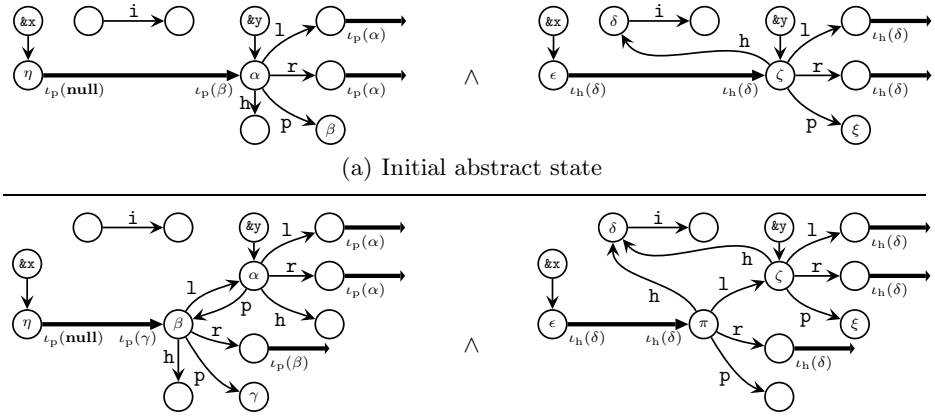
In this section, we consider a more powerful memory abstract domain, with inductive summaries for unbounded memory regions [5,6].

5.1 A memory abstraction with inductive summaries

As noticed in Sect. 2, unbounded heap regions can be summarized using inductive predicates. Given an inductive definition ι, inductive predicate $\alpha \cdot \iota(\beta)$ [6], expresses that α points to a heap region containing an inductive data structure expressed by ι. Similarly, $\alpha \cdot \iota(\beta) \mathrel{-\!\!*\!\!=} \alpha' \cdot \iota(\beta')$ [6] abstracts segments of such data-structures, that is, heap regions containing a data structure with a hole. Inductive and segment predicates can be unfolded using a syntactic unfolding relation $\rightsquigarrow$, which basically unrolls inductive definitions. Therefore, assuming ι is an inductive definition, the abstract values of domain $\mathbb{D}_{\langle\iota\rangle}$ are defined as a superset of those of $\mathbb{D}_{\mathbf{g}}$, shown in Fig. 7(a). Concretization $\gamma_{\langle\iota\rangle}$ also extends $\gamma_{\mathbf{g}}$, and relies on the unfolding relation to unfold arbitrarily many times all inductive predicates into an element with no inductive predicates, which can be concretized using $\gamma_{\mathbf{g}}$.

5.2 An extended language of constraints

In the following, we consider the abstract state shown in Fig. 8, which can be observed at the beginning of the `replace` function of Sect. 2. We let `y` (resp., `x`) be a shortcut for $\mathtt{iter} \to \mathtt{c}$ (resp., $\mathtt{tree} \to \mathtt{r}$). At this point, the analysis should update field $\mathtt{y} \to \mathtt{p} \to \mathtt{l}$, so this field should be materialized in all elements of the product. Fig. 8(a) shows the abstract state before the analysis of this operation. At this stage, we notice that field $\mathtt{y} \to \mathtt{p}$ is materialized as a points-to predicate in both sides of the conjunction, but $\mathtt{y} \to \mathtt{p} \to \mathtt{l}$ is materialized in neither. To obtain this materialization, the analysis first needs to *unfold* the segment backwards (i.e., at its destination node's site) from the node corresponding to the value of `y`. The triggering of this unfolding in $\mathbb{D}_{\langle\iota_{\mathtt{p}}\rangle}$ relies on the typing of inductive parameters proposed in [6] (left conjunct in Fig. 8). That unfolding actually generates three cases: either the segment is empty, or node α is the left child of its parent, or α is the right child of its parent. As the empty segment case is trivial (it would entail that α and η are equal so that both `x` and `y` hold



(a) Initial abstract state

(b) Abstract state after backward unfolding

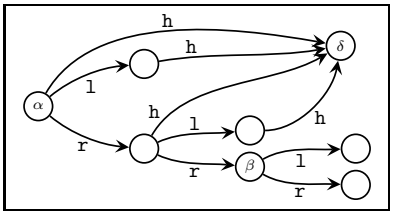
Fig. 8. Reduction example

the same value), and the two latter cases are similar, we focus on the second one (that of a left parent). This unfolding, however, cannot be triggered in $\mathbb{D}_{\langle \iota_h \rangle}$, as this domain does not capture the back-pointer tree invariant; thus the $\mathbb{D}_{\langle \iota_h \rangle}$ components needs some guidance from $\mathbb{D}_{\langle \iota_p \rangle}$ before it can proceed with the unfolding. Intuitively, $\mathbb{D}_{\langle \iota_p \rangle}$ carries the information that any node χ in the tree that has a left child is such that $\chi \cdot \mathtt{l} \cdot \mathtt{p} \equiv \chi$. Applying this principle to node π shows that the following steps should be carried out, in the right conjunct:

- the segment of the right conjunct that ends in ζ also needs to be unfolded since $\zeta \cdot \mathtt{p} \triangleright \xi$ (as shown in the right conjunct in Fig. 8);
- after unfolding, $\pi \cdot \mathtt{l} \cdot \mathtt{p} \triangleright \xi$, thus node ξ is actually equal to node π.

This example shows that the language of constraints introduced in Sect. 4.1 needs to be extended with *universally quantified predicates* over summarized regions. As such predicates may be expressed over unbounded paths, we use general regular expressions over the set of fields (Fig. 9(a)) instead of only sequences of offsets. Moreover, the quantification may also need be done on *segments*. Therefore, we augment $\mathbb{P}_f$ with the $\mathcal{S}_\forall[\cdot,\cdot](\cdot,\cdot)$ predicates shown in Fig. 9(a): intuitively, $\mathcal{S}_\forall[p, a[X]](\alpha, S)$ means that $a[\chi]$ holds for any node χ that can be reached from α following path expression p (i.e., $\alpha \cdot p \triangleright \delta$ holds) but cannot be reached from any node $\beta \in S$ following path expression p (i.e. $\beta \cdot p \triangleright \delta$ does not hold). The variable X is a bound variable whose scope is the disjunctive path formula a. Thus, such predicates allow us to express not only properties about inductively-summarized structures but also about segments of such structures [5,6]. The semantics of that construction is shown in Fig. 9(b), extending the definition of Fig. 6(b).

For instance, the path quantification formula $\mathcal{S}_\forall[(\mathtt{l}+\mathtt{r})^\star, X \cdot \mathtt{h} \triangleright \delta](\alpha, \{\beta\})$ holds true for the element of $\mathbb{D}_{\mathbf{g}}$ below and expresses that any node in the tree stored at address



$p ::=$		paths
	$\mid \emptyset$	empty path
	$\mid \mathtt{f} \quad (\in \mathtt{F})$	single field
	$\mid p \cdot p$	concatenation
	$\mid p + p$	disjunction
	$\mid p^\star$	sequences
$a ::=$	$\ldots$	
	$\mid a \vee a$	disjunction
	$\mid \mathcal{S}_\forall[p, a[X]](\alpha, S)$	quantification

(a) Extended syntax

$$
\begin{array}{l}
h, \nu \models \alpha \cdot p_0 + p_1 \triangleright \beta \\
\quad \iff h, \nu \models \alpha \cdot p_0 \triangleright \beta \vee h, \nu \models \alpha \cdot p_1 \triangleright \beta \\
h, \nu \models \alpha \cdot p^\star \triangleright \beta \\
\quad \iff \exists n \in \mathtt{N},\ h, \nu \models \alpha \cdot p^n \triangleright \beta \\
h, \nu \models a_0 \vee a_1 \\
\quad \iff h, \nu \models a_0 \text{ or } h, \nu \models a_1 \\
h, \nu \models \mathcal{S}_\forall[p, a[X]](\alpha, S) \\
\quad \iff \forall \delta \in \mathbb{V}^\sharp, \begin{cases} \alpha \cdot p \triangleright \delta \Rightarrow a[\delta] \\ \vee\ \exists \beta \in S,\ \beta \cdot p \triangleright \delta \end{cases}
\end{array}
$$

(b) Extended semantics

Fig. 9. Extended language of path constraints

α and that is not in the sub-tree of address β for some $\beta \in S$ has an $\mathtt{h}$ field pointing to address δ, as shown in the inset figure.

5.3 Extraction of path predicates from inductive definitions

We now need to extend operators $\mathfrak{extract}$ and $\mathfrak{constrain}$ so as to allow communication from and to $\mathbb{D}_{\langle \iota \rangle}$. Compared to $\mathbb{D}_{\mathbf{g}}$, the main issue is that path information now has to be computed about inductive summaries; thus this computation logically requires inductive reasoning.

For instance, the property that any node α of a tree with parent pointers that has a left child is such that $\alpha \cdot \mathtt{l} \cdot \mathtt{p} \equiv \alpha$ needs be computed by induction over the ι_{p} inductive predicate describing that tree. This inductive reasoning can actually be done once and for all about inductive ι_{p} so that it can then be applied to any of its occurrences. Therefore, operator $\mathfrak{extract}$ should rely on the results of a pre-analysis of inductive definition ι_{p} that is computed *before* the static analysis of the program (i.e., it may be performed only once per inductive definition used in the analyzer). Moreover, we remark that such properties can be derived by induction over the inductive definition, which suggests a fixed-point computation, that is, an abstract interpretation based static analysis of ι_{p}—a parameter to the abstract domain used to static analyze the program.

In the following, we label inductive predicates with a bound on the induction depth so as to express the soundness of the inductive definition analysis. For instance, the inductive definition ι_{p} becomes the following:

$$
\begin{array}{rcl}
\alpha \cdot \iota_{\mathrm{p}}^0(\beta) & \rightsquigarrow & \alpha = 0 \wedge \mathbf{emp} \\
\alpha \cdot \iota_{\mathrm{p}}^{i+1}(\beta) & \rightsquigarrow & \alpha = 0 \wedge \mathbf{emp} \\
& & \bigvee \alpha \neq 0 \wedge \left(\begin{array}{c} \alpha \cdot \mathtt{l} \mapsto \delta_l * \alpha \cdot \mathtt{r} \mapsto \delta_r * \alpha \cdot \mathtt{p} \mapsto \beta * \alpha \cdot \mathtt{h} \mapsto \_ \\ * \ \delta_l \cdot \iota_{\mathrm{p}}^i(\alpha) * \delta_r \cdot \iota_{\mathrm{p}}^i(\alpha) \end{array} \right)
\end{array}
$$

Then, the analysis should compute a sequence of sets of path predicates $(\mathcal{A}_i)_{i \in \mathbb{N}}$ such that at any rank i, if $(h, \nu) \in \gamma_{\mathbf{s}}(\alpha \cdot \iota_{\mathrm{p}}^i(\beta))$, then $h, \nu \models \mathcal{A}_i$.

Analysis algorithm. It proceeds by a classical abstract interpretation over the inductive structure of ι, using an abstract domain of path predicates. More precisely, given $\mathcal{A}_i$, the computation of $\mathcal{A}_{i+1}$ involves the following steps:

– *Base rules* (with no inductive call) can be handled using the $\mathfrak{extract}$ function shown in Sect. 4.2. For instance, considering ι we get $\mathcal{A}_0 = \{\alpha \cdot \emptyset \triangleright \mathbf{null}\}$.

– *Inductive rule* $\alpha \cdot \iota^{i+1}(\beta) \rightsquigarrow M * \pi_1 \cdot \iota^i(\rho_1) * \ldots * \pi_k \cdot \iota^i(\rho_k)$ (where M contains no inductive predicate) requires the following steps to be performed:

1. by instantiation of $\mathcal{A}_i$ and application of $\mathfrak{extract}$ to M, the analysis generates $\mathfrak{extract}(M) \cup \mathcal{A}^i[\pi_1, \rho_1/\pi, \rho] \cup \ldots \cup \mathcal{A}^i[\pi_r, \rho_r/\pi, \rho]$;
2. then, the analysis introduces quantified path predicates using the following principle: if $\alpha \cdot \mathtt{f}_1 \triangleright \beta_1, \ldots, \alpha \cdot \mathtt{f}_p \triangleright \beta_p$ and $a[X/\alpha]$ holds, then so does
$$\mathcal{S}_\forall[(\mathtt{f}_1 + \ldots + \mathtt{f}_p)^\star, a[X]](\alpha, \{\beta_1, \ldots, \beta_p\});$$
3. last, the analysis eliminates intermediate variables $(\pi_1, \ldots, \pi_p, \beta_1, \ldots, \beta_p)$ after applying transitivity principles to the set of path predicates:
$$\begin{array}{rl} \alpha \cdot p \triangleright \beta \wedge \beta \cdot q \triangleright \delta & \Longrightarrow \alpha \cdot p \cdot q \triangleright \delta \\ \mathcal{S}_\forall[p, a](\beta, S) \wedge \mathcal{S}_\forall[p, a](\alpha, S') & \Longrightarrow \mathcal{S}_\forall[p, a](\beta, (S \setminus \{\alpha\}) \cup S') \quad \text{if } \alpha \in S. \end{array}$$

The resulting set of path predicates $\mathcal{B}_{i+1}$ collects sound path constraints over induction depths $1, \ldots, i+1$.

– *Over-approximation* of the resulting path predicates and of those in the previous iterate $\mathcal{A}_i$, using an abstract join over sets of path predicates, defined using a rewrite relation which maps pairs of path formulas into weaker path formulas, that is, such that $\forall j \in \{0, 1\},\ a_0, a_1 \overset{\sqcup}{\mapsto} a_\sqcup \wedge h, \nu \models a_j \Longrightarrow h, \nu \models a_\sqcup$. This property is guaranteed using rules such as:

$$\begin{array}{rll} \alpha \cdot \mathtt{p} \triangleright \beta,\ \alpha \cdot \mathtt{q} \triangleright \beta & \overset{\sqcup}{\mapsto} \alpha \cdot \mathtt{p} + \mathtt{q} \triangleright \beta & \\ \mathcal{S}_\forall[p, a](\alpha, S),\ \alpha \cdot \emptyset \triangleright \beta & \overset{\sqcup}{\mapsto} \mathcal{S}_\forall[p, a](\alpha, S) & \text{if } \beta \in S \\ \mathcal{S}_\forall[p, a](\alpha, S),\ \mathcal{S}_\forall[p, a'](\alpha, S') & \overset{\sqcup}{\mapsto} \mathcal{S}_\forall[p, b](\alpha, S \cup S') & \text{if } a, a' \overset{\sqcup}{\mapsto} b \end{array}$$

Then, $\mathcal{A}_{i+1}$ is defined as $\{a \mid \exists (a_i, a_{i+1}) \in \mathcal{A}_i \times \mathcal{B}_{i+1},\ a_i, a_{i+1} \overset{\sqcup}{\mapsto} a\}$.

Furthermore, at each step, regular expressions may be simplified. Termination is ensured by the definition of a $\overset{\sqcup}{\mapsto}$ operator that avoids generating too large formulas and hence is a widening. The inductive definitions analysis returns a sound result, as all steps are sound, that is, they preserve the aforementioned invariant property:

Theorem 1 (soundness). *This analysis algorithm is sound:*
– *For all* $i \in \mathbb{N}$ *and for all* $(h, \nu) \in \gamma_{\mathbf{s}}(\alpha \cdot \iota^i(\beta))$, $h, \nu \models \mathcal{A}_i$ *holds.*
– *Thus, after convergence to* $\mathcal{A}_\infty$, *we have:* $\forall (h, \nu) \in \gamma_{\mathbf{s}}(\alpha \cdot \iota(\beta)),\ h, \nu \models \mathcal{A}_\infty$.

Path predicates derived from segments predicates. Segment predicates may be considered inductive predicates with a slightly different set of base rules for the end-point [6]. Therefore, the same inductive analysis applies to segments as well.

In particular, if we consider segment $\pi \cdot \iota_{\mathrm{p}}(\rho) \mathrel{-\!\!\!*\!\!=} \eta \cdot \iota_{\mathrm{p}}(\varepsilon)$, the analysis computes the iterates below:

iteration i	iterate $\mathcal{A}_i$
0	$\{\pi \cdot \emptyset \triangleright \eta, \rho \cdot \emptyset \triangleright \varepsilon\}$
1	$\left\{\begin{array}{c} \pi \cdot (\mathtt{l}+\mathtt{r})^\star \triangleright \eta \ , \ \varepsilon \cdot \mathtt{p}^\star \triangleright \rho \\ \pi \cdot \emptyset \triangleright \eta \ \vee \ \pi \cdot \mathtt{p} \triangleright \rho \ , \ \varepsilon \cdot \emptyset \triangleright \rho \ \vee \ \varepsilon \cdot (\mathtt{l}+\mathtt{r}) \triangleright \eta \\ \mathcal{S}_\forall[(\mathtt{l}+\mathtt{r})^\star, X \cdot \mathtt{l} \cdot \mathtt{p} \triangleright X \vee X \cdot \mathtt{l} \triangleright 0 \vee X \cdot \mathtt{l} \triangleright \eta](\pi, \{\eta, 0\}) \\ \mathcal{S}_\forall[(\mathtt{l}+\mathtt{r})^\star, X \cdot \mathtt{r} \cdot \mathtt{p} \triangleright X \vee X \cdot \mathtt{r} \triangleright 0 \vee X \cdot \mathtt{r} \triangleright \eta](\pi, \{\eta, 0\}) \\ \mathcal{S}_\forall[\mathtt{p}^\star, X \cdot \mathtt{p} \cdot (\mathtt{l}+\mathtt{r}) \triangleright X \vee X \cdot \mathtt{p} \triangleright \rho](\varepsilon, \{\rho\}) \\ \mathcal{S}_\forall[\mathtt{p}^\star, X \cdot (\mathtt{l}+\mathtt{r}) \triangleright \eta](\varepsilon, \{\rho\}) \quad ... \end{array}\right\}$
2	$\left\{\begin{array}{c} \pi \cdot (\mathtt{l}+\mathtt{r})^\star \triangleright \eta \ , \ \varepsilon \cdot \mathtt{p}^\star \triangleright \rho \\ \pi \cdot \emptyset \triangleright \eta \ \vee \ \pi \cdot \mathtt{p} \triangleright \rho \ , \ \varepsilon \cdot \emptyset \triangleright \rho \ \vee \ \varepsilon \cdot (\mathtt{l}+\mathtt{r}) \triangleright \eta \\ \mathcal{S}_\forall[(\mathtt{l}+\mathtt{r})^\star, X \cdot \mathtt{l} \cdot \mathtt{p} \triangleright X \vee X \cdot \mathtt{l} \triangleright 0 \vee X \cdot \mathtt{l} \triangleright \eta](\pi, \{\eta, 0\}) \\ \mathcal{S}_\forall[(\mathtt{l}+\mathtt{r})^\star, X \cdot \mathtt{r} \cdot \mathtt{p} \triangleright X \vee X \cdot \mathtt{r} \triangleright 0 \vee X \cdot \mathtt{r} \triangleright \eta](\pi, \{\eta, 0\}) \\ \mathcal{S}_\forall[\mathtt{p}^\star, X \cdot \mathtt{p} \cdot (\mathtt{l}+\mathtt{r}) \triangleright X \vee X \cdot \mathtt{p} \triangleright \rho](\varepsilon, \{\rho\}) \end{array}\right\}$
3	$\mathcal{A}_2$ fixpoint reached

5.4 Reduction operators in the presence of inductive predicates

First, we note that the triggering and computation of reduction based on rule ($\mathbf{R}_{\triangleright\equiv}$) given in Sect. 4.2 still apply, with one minor caveat: when $\alpha \cdot p \triangleright \beta$ and $\alpha \cdot p \triangleright \gamma$, then β and γ can be merged only if path p is rigid, that is, involves no disjunction or $.^\star$ unbounded sequence. A supplementary rule ($\mathbf{R}_{\mathcal{S}\forall}$) is needed to treat universally quantified path predicates.

Triggering of the reduction process. The triggering condition for such a reduction is actually similar to before: when a field fails to be materialized at node α in the right conjunct, the reduced product searches for universal properties of the nodes of the structure to which α belongs. Thus it searches for universally quantified path properties from nodes corresponding to "ancestors" of α. In the example of Fig. 8, we notice that we need information about node ξ, which is part of the structure pointed to by ϵ; as ϵ and η both correspond to the value stored in $\mathtt{x}$, thus denote equal values, information should be read in $\mathbb{D}_{\langle \iota_{\mathrm{p}} \rangle}$ from η.

Computation of universally quantified path information. Operator $\mathfrak{extract}$ should simply instantiate the results of the pre-analysis of inductive definitions shown in Section 5.3. In the case of the example shown in Fig. 8, node η satisfies:

$$\mathcal{S}_\forall[(\mathtt{l}+\mathtt{r})^\star, X \cdot \mathtt{l} \cdot \mathtt{p} \triangleright X \vee X \cdot \mathtt{l} \triangleright \mathbf{null}](\eta, \{\mathbf{null}\})$$

Enforcing universally quantified path information. Reduction rule ($\mathbf{R}_{\mathcal{S}\forall}$) boils down to the following principle: if $\alpha \cdot p \triangleright \delta$ and $\mathcal{S}_\forall[p, a[X]](\alpha, S)$, then either $a[\delta]$ holds or there exists $\beta \in S$ such that $\beta \cdot p \triangleright \delta$. Besides, when $S = \{\mathbf{null}\}$, we directly derive $a[\delta]$. In the right conjunct of the example of Fig. 8, as π is reachable from ϵ following a path of the form $(\mathtt{l}+\mathtt{r})^\star$, we derive $\pi \cdot \mathtt{l} \cdot \mathtt{p} \triangleright \pi \vee \pi \cdot \mathtt{l} \triangleright \mathbf{null}$ from the above $\mathcal{S}_\forall[.,.](.,.)$ constraint. Clearly the left child of π is not null, so $\pi \cdot \mathtt{l} \cdot \mathtt{p} \triangleright \pi$. However, $\pi \cdot \mathtt{l} \cdot \mathtt{p} \triangleright \xi$ also holds. Thus, ($\mathbf{R}_{\triangleright\equiv}$) now applies and

$\xi = \pi$. This allows us to materialize field $\mathtt{y} \to \mathtt{p} \to \mathtt{l}$ in $\mathbb{D}_{\langle \iota_{\mathrm{h}} \rangle}$. as well as in $\mathbb{D}_{\langle \iota_{\mathrm{h}} \rangle}$. The reduction allows the analysis to continue while retaining a high level of precision (note that we needed to materialize that field in both conjuncts to analyze the update precisely). Note that cases where node α would be the left child in one conjunct and the right one in the other conjunct are ruled out. Indeed, as soon as the parent nodes are identified as equal during the reduction process, the `constrain` operator will deduce that its left and right children are equal, which would violate the separation property. A reduction in the opposite direction would need be performed for the analysis of an assignment to $\mathtt{y} \to \mathtt{h} \to \mathtt{i}$. More generally, sequences of accesses to `p` and `h` fields will generate cascades of reductions. This example shows that the reduced product analysis decomposes reasoning over `p` and `h` fields in both domains and organizes the exchange of information to help materialize points-to predicates across the product.

6 Implementation

We have implemented the memory reduced product combinator into the MemCAD analyzer (*Memory Compositional Abstract Domain*) as an ML functor taking two memory abstract domains as arguments and returning a new one and the inductive definition pre-analysis described in Sect. 5.3. Reduction strategy can be selected among those presented in Sect. 4.2 (*on-read* comes as default whereas *minimal* and *maximal* can be activated as options). The analysis is fully automatic and takes as input C code and inductive definitions such as those shown in Sect. 2. It computes a finite disjunction of abstract states for each program point. It was run on a set of over 30 micro-benchmarks, as well as medium-sized ones such as the iterator described in Sect. 2. Fig 10 presents selected analysis results (timings were measured on a 2.2 Ghz Intel Core i7 with 8 GB of RAM) that highlight the impact of the reduced product and the reduction strategies. In particular, multiple other list and trees algorithms gave similar results and are not presented here. For each analysis, the table shows the number of LOCs, the mode (analysis with no reduced product —no r.p.—, using a monolithic domain, with a single inductive definition, or with a reduced product and minimal, on-read or maximal reduction mode), analysis time in seconds, number of calls to reduction operations in col. **(a)**, number of path predicates computed by reduction rules ($\mathbf{R}_{\triangleright\equiv}$) and ($\mathbf{R}_{\mathcal{S}\forall}$) in col. **(b)** (comprising all steps to perform reduction operations), number of node merges performed as part of reductions in col. **(c)**, number of reduction proving a disjunct has an empty concretization (hence, can be pruned) in col. **(d)**, average number of disjuncts per program point, and timing ratio compared with the analysis time with no reduction product. Reduction may prove abstract elements have an empty concretization, for example, when it infers that both $\alpha \cdot \mathtt{f} \triangleright \beta$ and $\alpha \cdot \emptyset \triangleright \mathbf{null}$ hold, or when it discovers equalities that would violate separation.

The comparison with monolithic analyses involving a single, more complex inductive definition shows those are faster than analyses with a product domain, which is not surprising, as the product analyses induces an overhead of duplicate

Filename & Description	LOCs	Reduction mode	Time (s)	(a)	(b)	(c)	(d)	Avg. disjs	Speed down
structure: *doubly linked list with shared record*									
`insert_list.c`	35	no r.p.	0.018	-	-	-	-	1.33	1
		on read	0.042	2	40	1	1	2.07	2.33
		maximal	0.056	26	417	8	4	1.96	3.11
structure: *tree with parent pointers and pointers to static record*									
`read_tree.c`	42	no r.p.	0.028	-	-	-	-	1.43	1
(random traversal		minimal	0.120	4	118	0	0	3.07	4.28
then read data field)		on read	0.086	9	391	8	0	1.87	3.07
		maximal	0.095	32	919	18	4	1.73	3.39
`insert_tree.c`	47	no r.p.	0.031	-	-	-	-	1.56	1
(random traversal		minimal	Fails						
then insert element)		on read	0.080	9	379	8	0	1.86	2.58
		maximal	0.090	43	1089	18	4	1.73	2.90
`rotate_tree.c`	47	no r.p.	0.031	-	-	-	-	1.56	1
(random traversal		on read	0.086	8	350	8	0	2.03	2.77
then rotate)		maximal	0.098	44	1201	22	4	1.92	3.16
structure: *tree with parent pointers and pointers to static record, and iterator*									
`iter_00.c`	171	no r.p.	0.278	-	-	-	-	8.74	1
(random traversal)		minimal	Fails						
		on read	0.701	64	2578	30	28	10.22	2.52
		on r & u	0.689	66	2635	28	30	9.68	2.47
		maximal	1.807	854	19714	28	30	10.06	6.5
`iter_01.c`	181	no r.p.	0.353	-	-	-	-	7.27	1
(random traversal)		on read	0.907	70	2902	34	32	7.99	2.56
		on r & u	0.871	80	3287	46	34	7.53	2.46
		maximal	2.263	978	24865	41	34	7.81	6.41

Fig. 10. Implementation Results.

domain operations and reduction operations. However, as regards the analysis with the "on-read" and "on r & u" strategies, the timing difference does not seem so dramatic (between 2.33X and 3.07X) and interestingly tends to reduce on larger examples). Besides, applying the product analysis *only* to the part of the heap where the composite structure lies, using a *separating product* domain combinator would cut that cost down further (though, is not part of the scope of this work).

The key part of our empirical evaluation is to assess strategies. While an overly aggressive strategy is likely to slow down the analysis by performing useless reductions, a too passive strategy may cause a loss in precision. This is why, in some cases, the minimal reduction strategy does not allow the analysis to succeed, as it tends to perform reduction too late, at a point where a stronger `constrain` operator would be needed to fully exploit predicates brought up by `extract` (this occurs for `insert_tree.c`, `rotate_tree.c` and both analyses with the iterator structure). More eager strategies such as on-read and maximal do not suffer from this issue. The on-read strategy analyzes all tests precisely. Moreover, while the slow down of maximal over on-read is low for small programs, it tends to increase more than linearly in the analysis time and reach 6.5X on the larger examples (while the on-read strategy is around 2.5X slower),

which suggests it is not likely to scale. This suggests the on-read strategy is a good balance.

Curiously, we also noticed that more aggressive strategies reduce the number of average disjuncts. Upon review, we discovered that this is due to some disjuncts being pruned by reduction earlier in the analysis and often right after unfolding points. Following this practical observation, we extended the on-read strategy so as to also perform reduction right after unfolding. The results obtained with this new strategy, called "on r & u" in the table, validate this hypothesis, as it reduces numbers of disjuncts and analysis time compared to the on-read strategy. It overall appears to be an efficient strategy.

7 Related works

Reduced product construction has been widely studied as a mathematical lattice operator [8,11] as well as a way to combine abstract domains so as to improve the precision of static analyses [8,3,9,13,4]. However, it is notoriously hard to design a general notion of reduced product: first, optimal reduction is either not computable or too costly to compute in general (so that all reduced product implementations should instead try to achieve a compromise between the cost of reduction and precision); second, exchanging information between domains requires them to support reduction primitives using a common language that may be hard to choose depending on the application domain. We believe this is the reason why no general form of such construction has been set up for memory abstractions thus far.

The most closely related work to ours is that of [15]. In that work, the authors consider a hybrid data-structure which contains a list structure laid over a tree structure. To abstract such memory states, the authors use non-separating conjunctions over *zones*. Our construction presents some similarities to theirs: we also use non-separating conjunction. Their technique and ours are quite complementary. Whereas our product construction focuses on the situations where precise path information must be transferred between components, theirs looks at the case where the only needed information is that two structures share the same nodes. In terms of an implementation strategy, their reduction relies on an instrumentation of the program to analyze, using ghost statements, whereas our implementation uses a *semantic* triggering of reduction, when one component fails.

Some analyses inferring properties of memory states utilize non-separating conjunctions in a local manner [18,14] in order to account for co-existing views on contiguous blocks corresponding to values of union types, or in order to handle casts of pointers to structures. Those analyses exploit conjunctions in a very local way and are unable to propagate global shape properties across the conjunctions.

Other authors have proposed to do a product of shape abstraction with a numerical domain [6,12,17]. These works are very different in that they do not combine two views of memory properties. Instead, they usually use numerical

abstract values to characterize the contents of memory cells the structure of which is accounted for in the shape abstraction: thus, those are usually *asymmetric* constructions, using a form of a co-fibered domain [21], that is, where the shape abstraction "controls" the memory abstraction. Other works have examined decomposing analyses of programs with numerical and memory properties into a sequence of analyses [16]. Compared to the above works, this approach does not allow information flow between analyses into both directions.

Last, we remark that the language of constraints used for the reduction conveys reachability information, which are at the foundation of TVLA shape analyses [20]. We found it interesting to note that such predicates are effective at providing a low level view of memory properties, that is, a kind of assembly language used between shape abstractions.

8 Conclusion

In this paper, we have proposed a reduced product combinator for memory abstract domains, with a general interface, allowing a modular abstract domain design. We have shown that this product can be used with existing shape abstractions based on separation logic and inductive definitions. Moreover, we have implemented the resulting framework inside the MemCAD analyzer and shown the impact of reduction strategies on analysis results and efficiency.

A first direction for future work consists of integrating other memory abstractions into our framework, so as to benefit from the reduced product combinator with expressive abstractions, such as, a domain based on three-valued logic [20]. A second direction for future work is to design and implement a combinator for memory abstraction based on *separating* conjunction, which would enable one to apply entirely different abstractions to cope with data structures stored in different memory regions, while keeping the interactions between those abstractions minimal. Together with our reduced product combinator, this combinator would enable one to derive analyses such as that of [15] as instances of our framework, among others, while retaining the advantages of a modular abstract domain.

References

1. L. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, 1994.
2. J. Berdine, C. Calcagno, B. Cook, D. Distefano, P. O'Hearn, T. Wies, and H. Yang. Shape analysis for composite data structures. In *CAV*, pages 178–192, 2007.
3. B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *PLDI*, pages 196–207, 2003.
4. B.-Y. E. Chang and R. Leino. Abstract interpretation with alien expressions and heap structures. In *VMCAI*, pages 147–163, 2005.
5. B.-Y. E. Chang, X. Rival, and G. Necula. Shape analysis with structural invariant checkers. In *SAS*, pages 384–401, 2007.

6. Bor-Yuh Evan Chang and Xavier Rival. Relational inductive shape analysis. In *POPL*, pages 247–260, 2008.
7. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252, 1977.
8. P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *POPL*, pages 269–282, 1979.
9. P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. Combination of abstractions in the astrée static analyzer. In *ASIAN*, pages 272–300, 2006.
10. D. Distefano, P. O'Hearn, and H. Yang. A local shape analysis based on separation logic. In *TACAS*, pages 287–302, 2006.
11. R. Giacobazzi and I. Mastroeni. Domain compression for complete abstractions. In *VMCAI*, pages 146–160, 2003.
12. S. Gulwani, T. Lev-Ami, and M. Sagiv. A combination framework for tracking partition sizes. In *POPL*, pages 239–251, 2009.
13. S. Gulwani and A. Tiwari. Combining abstract interpreters. In *PLDI*, pages 376–386, 2006.
14. V. Laviron, B.-Y. E. Chang, and X. Rival. Separating shape graphs. In *ESOP*, pages 387–406, 2010.
15. O. Lee, H. Yang, and R. Petersen. Program analysis for overlaid data structures. In *CAV*, pages 592–608, 2011.
16. S. Magill, J. Berdine, E. Clarke, and B. Cook. Arithmetic strengthening for shape analysis. In *SAS*, volume 4634 of *LNCS*, pages 419–436. Springer, 2007.
17. B. McCloskey, T. Reps, and M. Sagiv. Statically inferring complex heap, array, and numeric invariants. In *SAS*, pages 71–99, 2010.
18. A. Miné. Field-sensitive value analysis of embedded C programs with union types and pointer arithmetics. In *LCTES*, pages 54–63, 2006.
19. J. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, pages 55–74, 2002.
20. M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. In *POPL*, pages 105–118, 1999.
21. A. Venet. Abstract cofibered domains: Application to the alias analysis of untyped programs. In *SAS*, pages 366–382, 1996.