

# Program analysis for overlaid data structures

Oukseh Lee<sup>1,2</sup>, Hongseok Yang<sup>2</sup>, and Rasmus Petersen<sup>2</sup>

Hanyang University<sup>1</sup> and Queen Mary University of London<sup>2</sup>

**Abstract.** We call a data structure overlaid, if a node in the structure includes links for multiple data structures and these links are intended to be used at the same time. In this paper, we present a static program analysis for overlaid data structures. Our analysis implements two main ideas. The first is to run multiple sub-analyses that track information about non-overlaid data structures, such as lists. Each sub-analysis infers shape properties of only one component of an overlaid data structure, but the results of these sub-analyses are later combined to derive the desired safety properties about the whole overlaid data structure. The second idea is to control the communication among the sub-analyses using ghost states and ghost instructions. The purpose of this control is to achieve a high level of efficiency by allowing only necessary information to be transferred among sub-analyses and at as few program points as possible. Our analysis has been successfully applied to prove the memory safety of the Linux deadline IO scheduler and AFS server.

## 1 Introduction

Recent advances in verification research have resulted in successful industrial-strength software verifiers, such as Microsoft SDV and ASTREE. These tools do verification-by-static-analysis, where the tools work fully automatically without asking the user to insert loop invariants or procedure specifications. But these tools cannot approach many parts of operating systems, because of their inaccurate or unsound treatment of the heap. In fact, the heap is one of the outstanding problems holding back verification-by-static-analysis (or software model checking). Although there have been works approaching verification of the heap in real-world systems programs [3, 13], fundamental problems remain, and one of the most fundamental is the presence of nontrivial, but not unrestricted, sharing. The not unrestricted aspect gives some hope that techniques might be found that do not immediately run into an efficiency brick wall.

In this paper, we consider the automatic verification of overlaid data structures, which show such nontrivial but not unrestricted sharing. We call a data structure overlaid, if a node in the structure includes links for multiple data structures and these links are intended to be used at the same time. These overlaid data structures are frequently used in systems code in order to impose multiple types of indexing structures over the same set of nodes. For instance, the deadline IO scheduler of Linux has a queue whose nodes has links for a doubly-linked list as well as links for a red-black tree. The linked list is used to

record the order in which nodes are inserted in the queue, and the red-black tree provides an efficient indexing structure on the sector fields of the nodes.

Our goal is to build an efficient yet precise program analysis for overlaid data structures, capable of verifying the memory safety or shape properties of real-world programs. The issue here is not to verify toy problems of overlaid data structures, but to verify real-world examples. In fact, we created an analyser in 2008 that could prove the memory safety of toy examples, but this analyser could not scale to verify real code like the deadline IO scheduler for several fundamental reasons (see Section 7). Also, there have been other papers that take on toy programs using overlaid data structures or graphs, but they are all too imprecise or too expensive to verify serious programs [9, 7, 4, 12].

In this paper, we present a new program analysis for overlaid data structures, which can verify the memory safety and shape properties of medium sized real-world examples from Linux. Our analysis implements two main ideas. The first is to run multiple sub-analyses that track information about standard data structures, such as lists and trees. Each sub-analysis infers shape properties of only one component of an overlaid data structure, but the results of these sub-analyses are later combined to derive the desired safety properties about the whole overlaid data structure. This is reminiscent of cartesian abstraction [2].

The second idea is to control the communication among the sub-analyses using ghost states and ghost instructions. We found that to prove the memory safety of programs using overlaid data structures, the sub-analyses need to transfer information among themselves (using a form of reduction operator [5]); the memory safety of those programs often relies on the fact that components of an overlaid data structure use the same set of nodes. Our analysis controls this information transfer in order to achieve a high level of efficiency. It aims for allowing only necessary information to be transferred among sub-analyses and only at as few program points as possible. To achieve this aim, the analysis uses ghost states, special instructions for modifying these ghost states, and algorithms that insert those instructions before or during the main phase of the analysis.

**Related work** We discuss two further related works here. The first is the synthesis approach by Hawkins *et al.* [8], where a programmer specifies an overlaid data structure using a high level specification in the style of a relational database. This approach focuses on generating new correct programs using overlaid data structures, and it is complementary to the results of this paper. The second is the general meet algorithm [1] for finding intersections of heap abstractions in TVLA. The algorithm is related to our operator for transferring information among sub-analyses, but it aims for computing the exact meet, not an efficient overapproximation of the meet as in this paper.

## 2 Informal description

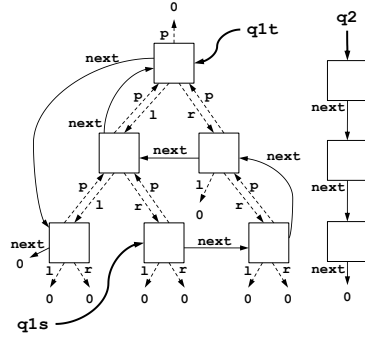
We start with an informal description of our analysis using the baby IO scheduler in Figure 1(a), which is modelled after the Linux deadline IO scheduler.

**Fig. 1** Baby IO scheduler

```
struct node { struct node *next;
             struct node *p,*l,*r;
             int key; };
struct node *q1s, *q1t, *q2;

void move_request() {
  struct node *c;
  c = list_remove_first(&q1s);
  if(c==0) return; //trans(list->tree)(c)
  tree_remove(c); //move(c,gamma)
  list_add_first(&q2,c);
  c = 0; //moveRgn(gamma,beta)
}
```

(a) C code



(b) a snapshot of data structure

Our baby IO scheduler schedules IO requests using two disjoint queues. When a request arrives, it is stored in the first queue. Later the request is selected according to a scheduling policy, processed, and moved to the second queue. In order to help the performance of the scheduling, the first queue uses an overlaid data structure with list and tree components. The list component is a singly-linked list starting from `q1s`, and it keeps requests in FIFO order. The tree component is a binary search tree with parent pointers. The address of the root of the tree is stored in `q1t`, and the tree provides an efficient search mechanism on the `key` field of requests. The second queue is, on the other hand, a simple linked list from `q2`, storing processed requests in FIFO order. A concrete example of both queues is shown in Figure 1(b).

The `move_request` function in Figure 1 shows a typical example of exploiting both components of an overlaid data structure. This function removes the first node of the list component `q1s` of the overlaid data structure. Then, it switches to the tree component, removes the node from the tree, and adds it to `q2`. One important aspect is that the removal from the tree exploits the correlation between components of the overlaid data structure—both the list `q1s` and the tree `q1t` use the same set of nodes. Although the node `c` is found using the list part, the correlation ensures that the node is in the tree as well. Hence, the removal from the tree can be performed safely without traversing the tree.

The main challenge for automatically proving the memory safety or shape properties of the baby IO scheduler is to find a good representation of the overlaid data structure (`q1s, q1t`), which enables the design of an efficient yet precise program analysis. Although nodes in this data structure are highly shared, this sharing has a pattern, i.e., it is generated by the overlay of a list and a tree. Furthermore, our baby scheduler, like the original Linux IO scheduler, relies only on the correlation between the list and tree components found in the `move_request` function—both components are formed using exactly the same set of nodes. We would like the representation to fully exploit the pattern of (`q1s, q1t`), and to express only this relatively weak correlation of its two components.

Our solution is to use the conjunction of two types of assertions  $\varphi \wedge \psi$ , where  $\varphi$  describes the heap only in terms of list fields and  $\psi$  does the same but using only the fields from the tree (including `key`). To express that the components of an overlaid data structure use the same set of nodes,  $\varphi$  and  $\psi$  use what we call region variables  $\alpha, \beta, \gamma$ , which denote sets of memory addresses. Concretely, our analysis infers that the data structures of our IO scheduler normally satisfy the following assertion:

$$(\text{ls}(\mathbf{q1s})_\alpha * \text{ls}(\mathbf{q2})_\beta) \wedge (\text{tr}(\mathbf{q1t})_\alpha * \text{true}_\beta). \quad (1)$$

The predicate  $\text{ls}(x)$  means a singly-linked list starting from the address  $x$ , and  $\text{tr}(y)$  a tree rooted at  $y$ . The separating conjunction  $P * Q$  means that the heap consists of two disjoint sub-heaps described by  $P$  and  $Q$ .

The first conjunct in (1) says that the heap contains two disjoint singly-linked lists `q1s` and `q2`. Using the subscripts  $-\alpha$  and  $-\beta$ , it also states that the addresses of the nodes in the list `q1s` form the set  $\alpha$ , and those of the nodes in the list `q2` the set  $\beta$ . The second conjunct, on the other hand, talks about tree-related properties of the heap. According to this conjunct, the heap contains a tree with root address `q1t`. Furthermore, the addresses of nodes in the tree form the set  $\alpha$ , while the addresses of all the other nodes make the set  $\beta$ . Note that each conjunct has its own characterisations of  $\alpha$  and  $\beta$ . To be consistent, both characterisations of  $\alpha$  should mean the same, which implies that the list and the tree use the same set of nodes. This is exactly the type of correlation that we want to express for the overlaid data structure (`q1s, q1t`).

This representation enables an interesting strategy for analyzing a client program of an overlaid data structure. The strategy is to run multiple sub-analyses that are designed for tracking information about standard non-overlaid data structures, such as lists and trees. Each of these sub-analyses infers shape properties of only one component of the overlaid data structure, hence handling only one conjunct in our representation. The desired memory properties of the program are then proved by combining the results of sub-analyses.

Our analysis implements a real-world adjustment of this strategy. Note that in our example, the sub-analyses cannot be completely independent. They need to communicate during (not after) analysis, because of the above mentioned correlation among components of an overlaid data structure; in the function `move_request`, the removal of `c` from the tree cannot be inferred to be safe without looking at the list. To address this concern while keeping the communication cost of the sub-analyses low, our analysis uses ghost instructions for region variables. It runs the sub-analyses independently most of the time, except at a few program points where the memory safety proof demands communication among the sub-analyses. At these program points, the analysis inserts ghost instructions that initiate communication among sub-analyses. Furthermore, even in those communication points, the analysis tries to keep the communicated information as simple as possible, using region variables.

We illustrate the analysis using the `move_request` function in Figure 1. In this case, our analysis runs the list and tree sub-analyses, which update the conjunct for list and that for tree, respectively. The first step of our analy-

sis is a pre-analysis that inserts ghost instructions for changing the values of region variables or for transferring information between the list and tree sub-analyses. For our `move_request` example, the pre-analysis inserts  $\text{trans}_{\text{list} \rightarrow \text{tree}}(c)$  and  $\text{move}(c, \gamma)$  before and after `tree_remove`, as shown in Figure 1. The first instruction  $\text{trans}_{\text{list} \rightarrow \text{tree}}(c)$  tells the tree sub-analysis to get information about cell  $c$  from the list sub-analysis, and it is a so-called reduction operator in program analysis [5]. The instruction is inserted here, because the pre-analysis conjectures that information transfer about cell  $c$  at this program point will be necessary for verification. The second instruction  $\text{move}(c, \gamma)$  tells the analysis to manage the values of region variables by moving the address  $c$  from its current region to the region  $\gamma$ . We defer the details of the pre-analysis to Section 5.

The second step is to run the `move_request` function symbolically starting from the assertion in (1), while abstracting away unnecessary information from time to time. This symbolic abstract execution is done by invoking the corresponding routines of the sub-analyses. The command `list_remove_first(&q1s)` is run first in this manner, and results in the assertion

$$(\text{ls}(\mathbf{q1s})_\alpha * c \mapsto \{\}_\alpha * \text{ls}(\mathbf{q2})_\beta) \wedge (\text{tr}(\mathbf{q1t})_\alpha * \text{true}_\beta) \quad (2)$$

for the true branch of the following conditional statement. Compared to the original in (1), the assertion has additionally  $c \mapsto \{\}_\alpha$  in the first conjunct, and this additional predicate describes the cell  $c$  removed from the list  $\mathbf{q1s}$ . In this abstract execution, our analysis runs only the list sub-analysis not the tree one, because it detects that `list_remove_first(&q1s)` is equivalent to skip as far as the tree sub-analysis is concerned.

Note that only the first conjunct of (2) knows the allocatedness of cell  $c$  in  $\alpha$ . The next instruction  $\text{trans}_{\text{tree} \rightarrow \text{list}}(c)$  makes the analysis transfer the information about cell  $c$  from the first to the second conjunct, which gives the assertion:

$$(\text{ls}(\mathbf{q1s})_\alpha * c \mapsto \{\}_\alpha * \text{ls}(\mathbf{q2})_\beta) \wedge (\varphi(\mathbf{q1t}, c, \alpha) * \text{true}_\beta). \quad (3)$$

Here  $\varphi(\mathbf{q1t}, c, \alpha)$  is an assertion with free variables  $\mathbf{q1t}, c, \alpha$ , and it describes a tree with root  $\mathbf{q1t}$  and a normal node  $c$  such that all nodes of the tree form the set  $\alpha$ .<sup>1</sup> This refinement of assertions is how our analysis enables the communication between sub-analyses, this time from the list to the tree sub-analysis. The transferred information allows the analysis to prove the memory safety of the following instruction `tree_remove(c)`, which is handled by the tree sub-analysis only, and to overapproximate the instruction's output states by the assertion:

$$(\text{ls}(\mathbf{q1s})_\alpha * c \mapsto \{\}_\alpha * \text{ls}(\mathbf{q2})_\beta) \wedge (\text{tr}(\mathbf{q1t})_\alpha * c \mapsto \{\}_\alpha * \text{true}_\beta). \quad (4)$$

This assertion has  $c \mapsto \{\}_\alpha$  in both conjuncts, hence confirming that the node  $c$  is indeed removed from both the list  $\mathbf{q1s}$  and the tree  $\mathbf{q1t}$ .

The next instruction is the ghost instruction  $\text{move}(c, \gamma)$  inserted by the pre-analysis. This instruction simply changes the subscript of  $c \mapsto \{\}$  from  $\alpha$  to  $\gamma$ :

<sup>1</sup> Concretely,  $\varphi(\mathbf{q1t}, c, \alpha)$  is  $\exists uvwxy. \text{tseg}(\mathbf{q1t}, 0, c, u)_\alpha * c \mapsto \{p:u, l:v, r:x\}_\alpha * \text{tseg}(v, c, 0, w)_\alpha * \text{tseg}(x, c, 0, y)_\alpha$ . Here  $\text{tseg}(a, b, c, d)$  is a tree segment predicate, and is explained in Section 4.

$$(\text{ls}(\mathbf{q1s})_\alpha * \mathbf{c} \mapsto \{\}_\gamma * \text{ls}(\mathbf{q2})_\beta) \wedge (\text{tr}(\mathbf{q1t})_\alpha * \mathbf{c} \mapsto \{\}_\gamma * \text{true}_\beta). \quad (5)$$

Semantically, this change means that the allocated cell  $\mathbf{c}$  is moved from the set  $\alpha$  to the set  $\gamma$ , which only contains  $\mathbf{c}$ . The decision for singling out  $\mathbf{c}$  and putting it in a separate set  $\gamma$  is made because the pre-analysis detected a possibility of moving cell  $\mathbf{c}$  between two different data structures. This possibility is indeed realized in the program, because the following two instructions `list_add_first(&q2, c)` and `c = 0` move the cell  $\mathbf{c}$  to the second queue  $\mathbf{q2}$ . The analysis tracks the move of the cell, using its list sub-analysis, and transforms (5) to the assertion:

$$(\exists a. \text{ls}(\mathbf{q1s})_\alpha * \mathbf{q2} \mapsto \{\text{next}:a\}_\gamma * \text{ls}(a)_\beta) \wedge (\exists b. \text{tr}(\mathbf{q1t})_\alpha * b \mapsto \{\}_\gamma * \text{true}_\beta). \quad (6)$$

The existentially quantified variable  $a$  has the old value of  $\mathbf{q2}$ , and  $b$  the old value of  $\mathbf{c}$ .

Note that the sub-formula  $\mathbf{q2} \mapsto \{\text{next}:a\}_\gamma * \text{ls}(a)_\beta$  in (6) describes a list starting from  $\mathbf{q2}$  of length at least one (because of cell  $\mathbf{q2}$ ). The list sub-analysis decides that this length information is not necessary for verifying the memory safety of the program, and it plans to drop the information by replacing the sub-formula by  $\text{ls}(\mathbf{q2})$ . To do this, the analysis inserts the instruction `moveRgn( $\gamma, \beta$ )` for moving all cells in  $\gamma$  to  $\beta$ , and analyses the inserted instruction:

$$(\exists a. \text{ls}(\mathbf{q1s})_\alpha * \mathbf{q2} \mapsto \{\text{next}:a\}_\beta * \text{ls}(a)_\beta) \wedge (\exists b. \text{tr}(\mathbf{q1t})_\alpha * b \mapsto \{\}_\beta * \text{true}_\beta). \quad (7)$$

The reason for inserting the instruction `moveRgn( $\gamma, \beta$ )` is to make sure that the changes in the values of region variables happen consistently for both conjuncts (i.e., both sub-analyses), although the changes are initiated by the need for abstracting a part of the first conjunct. Now, both the head  $\mathbf{q2}$  and the tail  $a$  are in the same set  $\beta$ , so the abstraction applies and gives the final result:

$$(\text{ls}(\mathbf{q1s})_\alpha * \text{ls}(\mathbf{q2})_\beta) \wedge (\text{tr}(\mathbf{q1t})_\alpha * \text{true}_\beta). \quad (8)$$

Here  $b \mapsto \{\}_\beta * \text{true}_\beta$  is also abstracted to  $\text{true}_\beta$  by the tree sub-analysis. This amounts to forgetting the fact that  $\beta$  contains at least one cell.

Our formalization of the ideas described so far will form the rest of the paper.

### 3 Formal setting for region variables

**Instrumented storage model** We use a storage model where a state consists of three components. The first two are the usual ones, namely, the stack for program variables and the heap for dynamically allocated cells. The third one is, however, unusual, and it defines the values of region variables.

To give a formal definition of our model, we need four disjoint countable sets: a set `Addr`s of addresses; a set `Vars` of normal variables  $x, y, z$ ; sets `Fields` and `Regions` that respectively contain field names  $\mathbf{f}, \mathbf{g}$  of heap cells and region variables  $\alpha, \beta, \gamma$ . We assume that a fixed constant `null` is not in `Addr`s. The storage model is defined by the following equations:

$$\begin{aligned} \text{Vals} &= \text{Addr} \cup \{\text{null}\} & \text{Stacks} &= \text{Vars} \rightarrow \text{Vals} & \text{Heaps} &= \text{Addr} \xrightarrow{\text{fin}} (\text{Fields} \rightarrow \text{Vals}) \\ \text{Partitions} &= \text{Regions} \rightarrow \mathcal{P}(\text{Addr}) & \text{States} &= \text{Stacks} \times \text{Heaps} \times \text{Partitions} \end{aligned}$$

Note that a state has three components  $(s, h, \eta) \in \mathbf{States}$ , where  $s$  defines the values of stack variables,  $h$  specifies the contents of allocated cells, and  $\eta$  maps region variables to address sets. We call a pair  $(h, \eta)$  **well-formed** if the mapping  $\eta$  defines a partition of allocated cells, that is, the following holds:

$$(\text{dom}(h) = \bigcup_{\alpha \in \mathbf{Regions}} \eta(\alpha)) \wedge (\forall \alpha, \beta \in \mathbf{Regions}. \alpha \neq \beta \implies \eta(\alpha) \cap \eta(\beta) = \emptyset).$$

A state  $(s, h, \eta)$  is **well-formed** when  $(h, \eta)$  is well-formed. In the rest of this paper, we consider only well-formed states and pairs of heaps and region-maps.

Note that in a well-formed state, every allocated address belongs to a unique region. As a result, a fact on an allocated address  $l$  can be approximated by the region variable  $\alpha$  containing  $l$ . For instance, when the variable  $x$  contains the address  $l$  of an allocated cell (i.e.,  $s(x) = l$ ), we can approximate this information by  $s(x) \in \eta(\alpha)$ . Our analysis uses this approximation to form the lightweight information to be passed among the sub-analyses.

**Assertions** Assertions  $\varphi$  describe properties of states, and are defined as follows:

$$e ::= x \mid \mathbf{null} \quad \varphi ::= \varphi_\alpha \mid e = e \mid e \in \alpha \mid e \mapsto \{\vec{f} : \vec{e}\} \mid p(\vec{e}) \\ \mid \mathbf{emp} \mid \varphi * \psi \mid \mathbf{true} \mid \varphi \wedge \psi \mid \neg \varphi \mid \exists x. \varphi$$

This is a variant of the assertion language from separation logic [11]. The first  $\varphi_\alpha$  says that the heap satisfies  $\varphi$  and all the allocated addresses in the heap form the set  $\alpha$ . This is the most unusual case of our assertion language, and it enables one to talk about the values of region variables, the new part of our storage model. The next two are the usual equalities on expressions and the membership of an expression to a region variable. The assertion  $x \mapsto \{\vec{f} : \vec{e}\}$  means a heap containing only one cell  $x$  that stores  $\vec{e}$  in fields  $\vec{f}$ . This definition does not require that  $\vec{f}$  be the only fields in cell  $x$ . Hence, the cell  $x$  can have fields other than  $\vec{f}$ . The following case  $p(\vec{e})$  is the application of a primitive predicate  $p$ , such as the tree or singly-linked list predicates, and it is mainly used to describe a recursive data structure. Our assertion language includes separating connectives—**emp** for the empty heap and the region variables all having the empty set, and  $\varphi * \psi$  for the splitting of both the heap and the region-variable map such that one pair satisfies  $\varphi$  and the other  $\psi$ . The remaining cases are the standard connectives from classical logic, and they have the usual meanings. We point out that other standard connectives from classical logic can be defined in a standard way.

The formal semantics is given by a satisfaction relation  $\models$  between well-formed states and assertions  $(s, h, \eta) \models \varphi$ , and sample clauses of the semantics appear in Figure 2. The clause for  $\varphi * \psi$  uses the following partial combining operator  $(h_1, \eta_1) \bullet (h_2, \eta_2)$  on well-formed pairs of heaps and region-maps:

$$(h, \eta) \bullet (h', \eta') = \begin{cases} (h \uplus h', \lambda \beta. \eta(\beta) \uplus \eta'(\beta)) & \text{if } \text{dom}(h) \cap \text{dom}(h') = \emptyset \text{ and} \\ & \forall \alpha. \text{dom}(\eta(\alpha)) \cap \text{dom}(\eta'(\alpha)) = \emptyset \\ \text{undefined} & \text{otherwise} \end{cases}$$

The operator merges two pairs of heaps and region-maps, when both components of the pairs do not overlap. The definition of  $\varphi * \psi$  uses this operator to express the

**Fig. 2** Semantics of sample assertions. We assume a function  $\llbracket e \rrbracket$  from **Stacks** to **Vals** that defines the meaning of expression  $e$ , and a mapping  $\llbracket p \rrbracket$  from value tuples to heaps that specifies the semantics of primitive predicate  $p$ .

---

$s, h, \eta \models \varphi_\alpha$	$\iff$	$s, h, \eta \models \varphi$ and $\text{dom}(h) = \eta(\alpha)$
$s, h, \eta \models e \in \alpha$	$\iff$	$\llbracket e \rrbracket s \in \eta(\alpha)$
$s, h, \eta \models e \mapsto \{\vec{f} : \vec{e}\}$	$\iff$	$\text{dom}(h) = \{\llbracket e \rrbracket s\}$ and $h(\llbracket e \rrbracket s)\mathbf{f}_i = \llbracket e_i \rrbracket$ for all $1 \leq i \leq  \vec{f} $
$s, h, \eta \models p(\vec{e})$	$\iff$	$h \in \llbracket p \rrbracket(\llbracket \vec{e} \rrbracket s)$
$s, h, \eta \models \mathbf{emp}$	$\iff$	$\text{dom}(h) = \emptyset$ and $\eta(\alpha) = \emptyset$ for all $\alpha$
$s, h, \eta \models P * Q$	$\iff$	$\exists h_1, h_2, \eta_1, \eta_2. (h_1, \eta_1) \bullet (h_2, \eta_2) = (h, \eta)$ and $s, h_1, \eta_1 \models P_1$ and $s, h_2, \eta_2 \models P_2$

---

splitting of the heap and region-map components. Also note that the semantics of **emp** says that both the heap and the region map are empty.

**Syntax and semantics of programs** We consider simple imperative programs specified in terms of standard control flow graphs. These programs are directed graphs  $(V, E)$  with two distinguished vertices **entry**, **exit**  $\in V$  and a labeling function  $L$  from  $E$  to primitive instructions. The vertex **entry** is required to have no incoming edges and **exit** no outgoing edges.

The syntax of primitive instructions  $c$  are given by the following grammar:

$$\begin{aligned}
e &::= x \mid \mathbf{null} & b &::= e = e \mid e \neq e \mid b \wedge b \mid b \vee b \\
c &::= \mathbf{assume}(b) \mid x := e \mid x := e.\mathbf{f} \mid e.\mathbf{f} := e \mid \mathbf{free}(e) \\
& \mid x := \mathbf{new}_{\alpha, F}() \quad (\text{where } F \subseteq \mathbf{Fields}) \mid \mathbf{move}(e, \alpha) \mid \mathbf{moveRgn}(\alpha, \beta)
\end{aligned}$$

Most cases are standard imperative operations. For instance, **assume**( $b$ ) checks whether the input state satisfies  $b$ . If so, it skips. Otherwise, it diverges. The only exceptions are the last three cases. The instruction  $x := \mathbf{new}_{\alpha, F}()$  allocates a new cell with fields  $F$ , and puts this cell into the region  $\alpha$ . The fields of this new cell are uninitialized. The next two **move**( $e, \alpha$ ) and **moveRgn**( $\alpha, \beta$ ) are ghost instructions that mainly manipulate the region-map parts of states. When cell  $e$  is allocated in the input state, **move**( $e, \alpha$ ) removes this cell from its current region, and puts it in the region  $\alpha$ . The instruction **moveRgn**( $\alpha, \beta$ ) moves all the cells in the region  $\alpha$  to the region  $\beta$ . Hence, at the end of this instruction,  $\alpha$  contains no cells, while  $\beta$  contains all cells that used to be in  $\alpha$ . The meaning of both instructions is not ambiguous, because we assume that the input states are well-formed and so all allocated addresses belong to only one region variable.

Our analysis uses **move** and **moveRgn** to ensure that the region-map part of a state carries useful information about heap data structures. In particular, it aims for putting each data structure, such as a list or a tree, in its own partition described by some region variable  $\alpha$ , because then knowing  $e \in \alpha$  is sufficient to identify the data structure containing  $e$ .

The formal meaning of primitive instructions are given in terms of functions from **States** to  $\mathcal{P}(\mathbf{States}) \cup \{\mathit{err}\}$ , where  $\mathit{err}$  models a memory error. Sample cases of the semantics appear in Figure 3.

**Fig. 3** Semantics of sample primitive instructions. We assume a function  $\llbracket b \rrbracket$  from **Stacks** to  $\{true, false\}$  that defines the meaning of boolean  $b$ .

---

$\llbracket \text{assume}(b) \rrbracket(s, h, \eta) = \text{if } (\llbracket b \rrbracket s = \text{true}) \text{ then } \{(s, h, \eta)\} \text{ else } \emptyset$
$\llbracket x := \text{new}_{\alpha, F}() \rrbracket(s, h, \eta) = \{(s[x \mapsto l], h[l \mapsto v], \eta[\alpha \mapsto \eta(\alpha) \cup \{l\}]) \mid$ $l \in \text{Addr} \setminus \text{dom}(h) \text{ and } v \text{ is a function from } F \text{ to Vals}\}$
$\llbracket \text{move}(e, \alpha) \rrbracket(s, h, \eta) = \text{if } \neg(\exists \beta. \llbracket e \rrbracket s \in \eta(\beta)) \text{ then } \text{err}$ $\text{else } \{(s, h, \eta[\beta \mapsto \eta(\beta) \setminus \{\llbracket e \rrbracket s\}, \alpha \mapsto \eta(\alpha) \cup \{\llbracket e \rrbracket s\}])\}$
$\llbracket \text{moveRgn}(\alpha, \beta) \rrbracket(s, h, \eta) = \{(s, h, \eta[\alpha \mapsto \emptyset, \beta \mapsto \eta(\alpha) \cup \eta(\beta)])\}$

---

## 4 Abstract states

Our abstract domain consists of assertions of the form:

$$(\varphi_{1,1} \vee \dots \vee \varphi_{1,m_1}) \wedge (\varphi_{2,1} \vee \dots \vee \varphi_{2,m_2}) \wedge \dots \wedge (\varphi_{n,1} \vee \dots \vee \varphi_{n,m_n}). \quad (9)$$

Each conjunct here records the current analysis result of one sub-analysis. For instance, the first conjunct could express the findings of the list analysis, and say how fields for singly-linked lists are connected in the heap. The second conjunct could, on the other hand, be concerned with the result of the tree analysis, and describe the connection of tree-related fields. Notice that a disjunction appears right under the conjunction. This disjunction is used by a sub-analysis to keep track of various correlations of stack variables and heap data structures explicitly. We point out that this is the only disjunction explicitly appearing in the abstract state;  $\varphi_{i,j}$  does not contain any disjuncts inside.

Formally, our domain is parameterized by a finite collection  $\mathcal{F} = \{F_i\}_{1 \leq i \leq n}$  of sets of fields and primitive predicates  $p$ . The intention is that  $n$  specifies the number of sub-analyses, and that each  $F_i$  describes the fields and primitive predicates that the sub-analysis  $i$  cares about.

Once a parameter  $\mathcal{F}$  is given, we can construct our abstract domain  $\mathcal{D}(\mathcal{F})$  in three steps. First, we define special forms of assertions, called symbolic heaps:

$$\begin{array}{ll} \Pi ::= \text{true} \mid e = e \mid e \neq e \mid \Pi \wedge \Pi & \text{Pure formulae} \\ \Sigma ::= \text{true}_{\alpha} \mid (e \mapsto \{\vec{f} : \vec{e}\})_{\alpha} \mid (p(\vec{e}))_{\alpha} \mid \text{emp} \mid \Sigma * \Sigma & \text{Spatial formulae} \\ H ::= \exists \vec{x}. \Pi \wedge \Sigma & \text{Symbolic heaps} \end{array}$$

The  $\Pi$  part of a symbolic heap describes the information about variables, and the  $\Sigma$  part expresses a property on the heap and region-map components of states. Note that in a symbolic heap, the region subscript  $-\alpha$  is used only in limited places with three basic predicates. Furthermore, the pure part of a symbolic heap does not contain membership expressions  $e \in \alpha$ ; all memberships are implicitly expressed using the subscript formulae  $-\alpha$ . These and other syntactic restrictions in symbolic heaps (such as the absence of disjunction and negation) are imposed so that we can reuse the core components of existing separation-logic based shape analyses, such as abstraction algorithms and transfer functions [6]. We write **SH** for the set of all symbolic heaps.

Second, we define a set of assertions used by each sub-analysis  $i$ . Let  $\text{SH}_i$  be the set of symbolic heaps  $H$  such that all pointsto predicates  $(e \mapsto \{\vec{f} : \vec{e}\})_{\alpha}$  in  $H$  mention only fields in  $\mathcal{F}_i$  (i.e.,  $\vec{f} \subseteq \mathcal{F}_i$ ), and all primitive predicates  $(p(\vec{e}))_{\alpha}$

in  $H$  belong to  $\mathcal{F}_i$  (i.e.,  $p \in \mathcal{F}_i$ ). The domain for the sub-analysis  $i$  is  $\mathcal{D}_i = \mathcal{P}_{fin}(\text{SH}_i)$ . The finite powerset operator is used here to express finite disjunction. For instance, the set  $\{H_1, \dots, H_m\} \in \mathcal{D}_i$  means the disjunction  $H_1 \vee \dots \vee H_m$ .

Finally, the abstract domain  $\mathcal{D}(\mathcal{F})$  is defined by  $\mathcal{D} = \mathcal{D}_1 \times \dots \times \mathcal{D}_n \cup \{\top\}$  for  $n = |\mathcal{F}|$ . The cartesian product means the conjunction of assertions. For instance, the assertion (9) in the beginning of this section is formally represented by the tuple  $(\{\varphi_{1,1}, \dots, \varphi_{1,m_1}\}, \{\varphi_{2,1}, \dots, \varphi_{2,m_2}\}, \dots, \{\varphi_{n,1}, \dots, \varphi_{n,m_n}\})$  in this domain. The element  $\top$  means the possibility of error. We will use  $d$  to denote a non- $\top$  element in  $\mathcal{D}$ , and  $d_i$  to mean the  $i$ -th component of  $d$ .

The domain  $\mathcal{D}(\mathcal{F})$  is a lattice, when  $\top$  is considered the largest element and the non- $\top$  elements are ordered pointwise. Then, the lattice operations of  $\mathcal{D}(\mathcal{F})$  are obtained by extending corresponding operations on the  $\mathcal{D}_i$ 's pointwise. For instance, the join  $d \sqcup d'$  is given by  $(d_1 \sqcup d'_1, \dots, d_n \sqcup d'_n)$ .

**Weak reduction operator** One important operator of our domain is a weak reduction operator that transfers information among components of abstract states. The transferred information is about the allocatedness of a cell and a region variable  $\alpha$  containing this cell. For instance, consider the abstract state:

$$(\mathbf{x} \mapsto \{\text{next}:0\}_\alpha \vee (\exists a. \mathbf{x} \mapsto \{\text{next}:a\}_\alpha * \text{ls}(a)_\alpha)) \wedge \text{tr}(\mathbf{y})_\alpha$$

where only the first conjunct says that cell  $\mathbf{x}$  is allocated and belongs to the set  $\alpha$ . Using our reduction operator, we can transfer this information about cell  $\mathbf{x}$  from the first to the second conjunct. Given appropriate parameters, the operator transforms this abstract state to the one below:

$$(\mathbf{x} \mapsto \{\text{next}:0\}_\alpha \vee (\exists a. \mathbf{x} \mapsto \{\text{next}:a\}_\alpha * \text{ls}(a)_\alpha)) \\ \wedge \exists uvw. \text{tseg}(\mathbf{y}, 0, \mathbf{x}, u)_\alpha * \mathbf{x} \mapsto \{\mathbf{p}:u, \mathbf{l}:v, \mathbf{r}:w\}_\alpha * \text{tseg}(v, \mathbf{x}, 0, \_)_\alpha * \text{tseg}(w, \mathbf{x}, 0, \_)_\alpha.$$

Here the predicate  $\text{tseg}(a, b, c, d)$  describes a rooted tree segment with one hole. The root is  $a$  and its parent pointer points to  $b$ . The hole of the segment is an outgoing pointer from the tree, going from address  $d$  to address  $c$ . The source  $d$  belongs to the segment, but the target  $c$  does not. We write  $\_$  in the parameter of  $\text{tseg}$  when we do not want to specify the parameter.<sup>2</sup> Note that the second conjunct now talks about the allocatedness of cell  $\mathbf{x}$  and its membership of  $\alpha$ .

Our operator is defined by lifting a similar reduction operator on symbolic heaps to abstract states. We first describe this original unlifted operator, denoted  $\text{trans}$ . Let  $i$  be a sub-analysis id and  $e$  an expression.

$$\text{trans}_i(e)(H : \text{SH}, H' : \text{SH}_i) : \mathcal{D}_i =$$

**let**  $R = \text{getRegion}(e, H)$  **in if**  $(R = \text{NoInfo})$  **then**  $\{H'\}$  **else**  $\text{caseSH}_i(e, R, H')$ .

The operator  $\text{trans}_i(e)(H, H')$  transfers information about cell  $e$  from  $H$  to  $H'$ , and the transferred information talks about the allocatedness of  $e$  and a region variable that contains  $e$ . The operator starts by calling the subroutine  $\text{getRegion}(e, H)$ , which has two possible outcomes. The first outcome is  $\text{NoInfo}$  indicating that  $H$  does not have any information on cell  $e$ . In this case, the input  $H'$  gets no information from  $H$ , and it becomes the output of  $\text{trans}$ . The second

<sup>2</sup> Formally,  $\varphi * \text{tseg}(a, b, c, \_)$  is an abbreviation for  $\exists d. \varphi * \text{tseg}(a, b, c, d)$  for a fresh  $d$ .

**Fig. 4** Subroutines `getRegion` and `caseSHi`. The function `caseID` below is a parameter provided for each primitive predicate  $p$ . In the figure, we give an example of `caseID` for `tr` and `tseg`.

---

```

getΠ(e, emp) = NolInfo
getΠ(e, e' ↦ {f̄:e''}α * Σ) = if (Π ⊢ e = e') then α else getΠ(e, Σ)
getΠ(e, p(e')α * Σ) = if (Π ∧ p(e') ⊢ e ↦ {} * true) then α else getΠ(e, Σ)
getRegion(e, H) = let (∃x̄. Π ∧ Σ) = H in getΠ(e, Σ)
caseID(e, tr(e0)α) =
  {(uvwxy, tseg(e0, 0, e, u)α * e ↦ {p:u, l:v, r:w}α * tseg(v, e, 0, x)α * tseg(w, e, 0, y)α)}
caseID(e, tseg(e0, e1, e2, e3)α) =
  {(uvwxx, tseg(e0, e1, e, u)α * e ↦ {p:u, l:v, r:w}α * tseg(v, e, e2, e3)α * tseg(w, e, 0, x)α},
  (uvwxx, tseg(e0, e1, e, u)α * e ↦ {p:u, l:v, r:w}α * tseg(v, e, 0, x)α * tseg(w, e, e2, e3)α)}
case(e,α,Π)(Σ, emp) = ∅
case(e,α,Π)(Σ, e' ↦ {f̄:e''}β * Σ') =
  if (Π ⊢ e ≠ e' or α ≠ β) then case(e,α,Π)(Σ * e' ↦ {f̄:e''}β, Σ')
  else {(∅, e=e', Σ * e' ↦ {f̄:e''}β * Σ') ∪ case(e,α,Π)(Σ * e' ↦ {f̄:e''}β, Σ')}
case(e,α,Π)(Σ, p(e')β * Σ') =
  if (α ≠ β) then case(e,α,Π)(Σ * p(e')β, Σ')
  else {(ā, true, Σ * Σ'' * Σ') | (ā, Σ'') ∈ caseID(e, p(e'))} ∪ case(e,α,Π)(Σ * p(e')β, Σ')
caseSH(e, α, H) =
  let (∃x̄. Π ∧ Σ) = H in {∃x̄ā. (Π ∧ Π') ∧ Σ' | (ā, Π', Σ') ∈ case(e,α,Π)(emp, Σ)}

```

---

outcome is a region variable  $\alpha$  satisfying the entailment  $H \vdash e \in \alpha$ , which means that according to  $H$ , the region variable  $\alpha$  contains cell  $e$ . Given this outcome, the operator `trans` conjoins the membership information  $e \in \alpha$  with  $H'$ , and calls a case-analysis routine that transforms the assertion back into a set of symbolic heaps in `SHi`, while ensuring the soundness condition expressed below:

$$\mathcal{H} = \text{caseSH}_i(e, \alpha, H') \implies (e \in \alpha \wedge H') \vdash \bigvee_{H_k \in \mathcal{H}} H_k.$$

One implementation of `getRegion` and `caseSH` is given in Figure 4.

For sub-analysis ids  $i, j$  and an expression  $e$ , we define our weak reduction operator `transi→j(e) : D → D` by `transi→j(e)(T) = T` and

$$\text{trans}_{i \rightarrow j}(e)(d) = \text{let } \mathcal{H} = \bigcup \{ \text{trans}_j(H, H') \mid (H, H') \in d_i \times d_j \} \text{ in } d[j \mapsto \mathcal{H}].$$

This operator applies `transj` to all possible symbolic-heap combinations from the  $i$  and  $j$ -th components of  $d$ , and uses the result to update the  $j$ -th component.

Note that the parameters  $i, j, e$  control the transferred information by our reduction operator. It restricts the source to only one component of an abstract state, and does a similar restriction on the target. Furthermore, it transfers information only about cell  $e$ , with respect to its membership to one region variable. This fine-grained control is essential for the performance of our analysis. Based on the results of a pre-analysis, our analysis does only necessary information transfer among component sub-analyses, by using our reduction operator with carefully chosen parameters and only at necessary program points.

## 5 Pre-analysis

The input to our analysis is a control flow graph  $G = (V, E, \text{entry}, \text{exit}, L)$  and an initial abstract state  $d_{\text{init}} \in \mathcal{D} \setminus \{\top\}$ . Since  $G$  represents a normal C program, its labelling  $L$  does not use our ghost instructions or weak reduction operator.

Given this input, the analysis first runs a pre-analysis, which changes  $G$  by inserting ghost instructions of the form  $\text{move}(e, \alpha)$  or our weak reduction operator  $\text{trans}_{i \rightarrow j}(e)$ . Intuitively, the pre-analysis finds a program point  $v$  such that if cell  $e$  is allocated at  $v$  in the concrete semantics, all sub-analyses are likely to infer the allocatedness of  $e$ . Then, it picks a fresh region variable  $\alpha$ , and inserts  $\text{move}(e, \alpha)$  after  $v$ . For  $\text{trans}_{i \rightarrow j}(e)$ , the pre-analysis inserts this instruction before a program point  $v'$ , if it makes the following three conclusions at  $v'$ :

1. The cell  $e$  will be dereferenced by the sub-analysis  $j$ .
2. The sub-analysis is *unlikely* to infer that all reachable cells from  $e$  by  $F_j$ -fields are allocated or `null`, while this allocation property indeed holds in the concrete semantics.
3. But the sub-analysis  $i$  is *likely* to infer the same type of information about reachable cells from  $e$  by  $F_i$ -fields.

Here  $\{F_i\}_{1 \leq i \leq n}$  is a parameter to our analysis. The first can be detected easily by a simple syntactic check, but the other two require more sophisticated reasoning. In the remainder of this section, we focus on this reasoning as well as the one used for inserting  $\text{move}(e, \alpha)$ .

Both types of reasoning are based on data-flow analyses. Let  $\text{Subanalyses}$  be the set of sub-analysis ids  $\{1, \dots, n\}$ , and  $\text{Exp}$  the set of expressions in the input control flow graph  $G$ . Define the domain  $\mathcal{D}_{\text{pre}}$  by  $\mathcal{D}_{\text{pre}} = \mathcal{P}(\text{Subanalyses} \times \text{Exp})$ . Two data-flow analyses compute maps from program points to  $\mathcal{D}_{\text{pre}}$ , denoted  $R^r$  and  $R^p$ , by repeatedly applying the following equations:

$$\text{for } k \in \{r, p\} \text{ and } v \in V \setminus \{\text{entry}\}, \quad R_{n+1}^k(v) = \bigcap_{(v', v) \in E} \llbracket c \rrbracket_k^\#(R_n^k(v)).$$

This commonality leaves only  $R^r(\text{entry})$ ,  $R^p(\text{entry})$ ,  $\llbracket c \rrbracket_r^\#$ , and  $\llbracket c \rrbracket_p^\#$  unspecified. We give this missing information about the two data-flow analyses in Figure 5.

Our intention is that if  $(i, e) \in R^r(v)$ , then sub-analysis  $i$  likely infers that at program point  $v$ , all reachable cells from  $e$  by  $F_i$ -fields are allocated or `null`, if this property holds in the concrete semantics. Also, by  $(i, e) \in R^p(v)$ , we intend that the sub-analysis  $i$  knows enough to prove the allocatedness of  $e$  at program point  $v$ , if  $e$  is indeed allocated in the concrete semantics. Hence, if our intentions are properly implemented, the reasoning steps necessary for inserting  $\text{trans}$  and  $\text{move}(e, \alpha)$  can be done using  $R^r$  and  $R^p$ .<sup>3</sup>

The definitions of  $\llbracket c \rrbracket_r^\#$  and  $\llbracket c \rrbracket_p^\#$  in Figure 5 follow our intentions. The only exception is in the use of  $\text{deref}$  and  $\text{gen}_r(x := e.f, X)$ . Here the definitions assume that after the dereference operation  $e.f$ , all sub-analyses caring about  $\mathbf{f}$

<sup>3</sup>  $\text{move}(e, \alpha)$  is inserted after a program point  $v$ , if  $\text{Subanalyses} \times \{e\} \subseteq R^p(v)$ . Our second and third conditions for inserting  $\text{trans}_{i \rightarrow j}(e)$  are  $(j, e) \notin R^r(v)$  and  $(i, e) \in R^r(v)$ .

**Fig. 5** Subroutines used by the pre-analysis. The abstract state  $d_{init} \in \mathcal{D}$  is the initial abstract state given as the input to the whole analysis.

$R_n^r(\text{entry}) = \{(i, e) \mid e \text{ is null or appears in all } H \text{ in } (d_{init})_i\}$ ,  $R_n^p(\text{entry}) = \emptyset$   
For  $k \in \{r, p\}$ ,  $\llbracket c \rrbracket_k^{\#}(X) = (X \cup \text{deref}(c)) \setminus \text{kill}(c) \cup \text{gen}_k(c, X)$  (with  $\text{deref}$ ,  $\text{kill}$ ,  $\text{gen}_k$  below)

instr $c$	$\text{deref}(c)$	$\text{kill}(c)$	$\text{gen}_r(c, X)$	$\text{gen}_p(c, X)$
<b>assume</b> ( $b$ )	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
$x := e$	$\emptyset$	$\text{Subanalyses} \times \{x\}$	$\{(i, x) \mid (i, e) \in X\}$	$\{(i, x) \mid (i, e) \in X\}$
$x := e.f$	$\{(i, e) \mid f \in F_i\}$	$\text{Subanalyses} \times \{x\}$	$\{(i, x) \mid f \in F_i\}$	$\emptyset$
$e.f := e'$	$\{(i, e) \mid f \in F_i\}$	$\emptyset$	$\emptyset$	$\emptyset$
<b>free</b> ( $e$ )	$\emptyset$	$\text{Subanalyses} \times \{e\}$	$\emptyset$	$\emptyset$
$x := \text{new}_{\alpha, F}()$	$\emptyset$	$\text{Subanalyses} \times \{x\}$	$\text{Subanalyses} \times \{x\}$	$\text{Subanalyses} \times \{x\}$

know both types of reachability and allocatedness information regarding  $e$ . Intuitively, this assumption amounts to hypothesizing that our pre-analysis inserts the reduction operator **trans** in all the necessary places, so that by the time that the field  $f$  is dereferenced, every sub-analysis interested in the field knows the necessary information.

Finally, the definitions of  $R^p(\text{entry})$  and  $R^r(\text{entry})$  in Figure 5 reflect another assumption of ours: That the initial abstract state  $d_{init}$  does not imply allocatedness, but it contains expression  $e$  in the  $i$ -th conjunct only when the **null** status or the allocatedness of reachable cells from  $e$  is known to sub-analysis  $i$ .

## 6 Invariant inference

Next, our analysis runs its main invariant inference engine, which computes an invariant at each program point. Our invariant inference engine takes an initial abstract state  $d_{init}$  and the output of our pre-analysis, which is a control flow graph  $G = (V, E, \text{entry}, \text{exit}, L)$  that can include ghost instructions  $\text{move}(e, \alpha)$  and the reduction operator  $\text{trans}_{i \rightarrow j}(e)$  (but not  $\text{moveRgn}(\alpha, \beta)$ ). Given this input, the engine computes two maps  $I$  and  $A$  from program points, the first  $I$  to abstract states and the next  $A$  to sets of ghost instructions of the form  $\text{moveRgn}$ :

$$M = \{\text{moveRgn}(\alpha, \beta) \mid \alpha, \beta \in \text{Regions}(d_{init}, L)\}, \quad I : V \rightarrow \mathcal{D}, \quad A : V \rightarrow \mathcal{P}(M).$$

Here  $\text{Regions}(d_{init}, L)$  is the set of region variables appearing in  $d_{init}$  or some instruction in the range of  $L$ . Note that since  $\text{Regions}(d_{init}, L)$  is finite, so are  $M$ , its subsets and the collection  $\mathcal{P}(M)$ . The first map  $I$  is the usual result of a program analysis, and keeps an invariant at each program point. The second map  $A$  records the ghost instructions dynamically discovered and then executed during the invariant inference. These instructions move cells from one region variable to another, and they are added and executed so as to maintain the relationship between region variables and data structures in the heap.

Our analysis uses the standard fixpoint algorithm for control flow graphs, with one interesting twist regarding the map  $A$  for ghost instructions. Assume

**Fig. 6** Abstract transfer functions. The abstract value  $d$  below is not  $\top$ . When  $\top$  is the input,  $\text{abs}(\top) = \llbracket c \rrbracket^\sharp \top = \top$ . We assume that  $\llbracket c \rrbracket_i^\sharp$  and  $\text{abs}_i$  are given.

---

$\llbracket c \rrbracket^\sharp(d) = \text{if } (\exists i. \llbracket c \rrbracket_i^\sharp(d_i) = \top) \text{ then } \top \text{ else } (\llbracket c \rrbracket_1^\sharp(d_1), \dots, \llbracket c \rrbracket_n^\sharp(d_n))$   
 $\text{abs}(d) = (\text{abs}_1(d_1), \dots, \text{abs}_n(d_n))$        $\llbracket \text{trans}_{i \rightarrow j}(e) \rrbracket^\sharp(d) = \text{trans}_{i \rightarrow j}(e)(d)$   
 $\llbracket \text{move}(e, \alpha) \rrbracket^\sharp(d) =$        $\llbracket \text{moveRgn}(\alpha, \beta) \rrbracket^\sharp(d) = d[\beta/\alpha]$   
**let**  $\text{check}(i, H) =$   
     (1) Find finitely many  $H_k$ 's in  $\text{SH}_i$  such that  
          $H \vdash \bigvee_{k \in K} H_k$  and  $H_k$  has the form  $\exists \vec{x}_k. \Pi_k \wedge e \mapsto \{\vec{f}_k : \vec{e}'_k\}_{\beta_k} * \Sigma_k$ .  
     (2) If cannot find, return  $\{\top\}$ . Otherwise, for the found  $H_k$ 's, rename the  
         subscript  $\beta_k$  of  $e \mapsto \{\dots\}_{\beta_k}$  by  $\alpha$  and return  $\{\exists \vec{x}_k. \Pi_k \wedge e \mapsto \{\vec{f}_k : \vec{e}'_k\}_\alpha * \Sigma_k\}_{k \in K}$ .  
     **and**  $r_i = \bigcup \{\text{check}(i, H) \mid H \in d_i\}$  for all  $i$   
**in if**  $(\exists i \text{ s.t. } \top \in r_i) \text{ then } \top \text{ else } (r_1, \dots, r_n)$

---

that for all normal or ghost instructions or our weak reduction operator,  $c$ , we are given the transfer function for  $c$ , and the below two functions:

$$\llbracket c \rrbracket^\sharp : \mathcal{D} \rightarrow \mathcal{D}, \quad \text{abs} : \mathcal{D} \rightarrow \mathcal{D}, \quad \text{enableAbs} : \mathcal{D} \rightarrow \mathcal{P}(M). \quad (10)$$

Here  $\text{abs}(d)$  abstracts assertions in  $d$ , and  $\text{enableAbs}(d)$  returns ghost instructions in  $M$  that will enable further abstraction of  $d$ . Now, for (finite) subsets  $M_0$  of  $M$ , define  $\llbracket M_0 \rrbracket^\sharp(d) = (\llbracket c_n \rrbracket^\sharp \circ \dots \circ \llbracket c_1 \rrbracket^\sharp)(d)$ , where  $c_1, \dots, c_n$  is one enumeration of  $M_0$  according to a fixed scheme. (In our analysis, this choice does not matter, because the transfer functions of any two instructions in  $M$  commute.) Using what we have assumed or defined, we define the main fixpoint algorithm below:

$$I_n(\text{entry}) = d_{\text{init}}, \quad I_{n+1}(v) = \bigsqcup_{(v', v) \in E} (\text{abs} \circ \llbracket A_n(v) \rrbracket^\sharp \circ \llbracket L(v', v) \rrbracket^\sharp)(I_n(v')),$$

$$A_n(\text{entry}) = \emptyset, \quad A_{n+1}(v) = A_n(v) \cup \text{enableAbs}(I_{n+1}(v)).$$

Note that since  $\mathcal{P}(M)$  is finite, there are only finitely many values for  $A$ , and the fixpoint computation of  $A$  does not cause non-termination. In practice, we found that the analysis time is dominated by the fixpoint computation for  $I$ .

To complete the story, we need to discharge our assumption of the three functions in (10). For normal instructions  $c$ , we define  $\llbracket c \rrbracket^\sharp$  and  $\text{abs}$  by applying componentwise the sub-analyses' transfer functions  $\llbracket c \rrbracket_i^\sharp : \mathcal{D}_i \cup \{\top\} \rightarrow \mathcal{D}_i \cup \{\top\}$  and abstraction routines  $\text{abs}_i : \mathcal{D}_i \rightarrow \mathcal{D}_i$ . The details are given in Figure 6. The figure also shows that  $\llbracket \text{trans}_{i \rightarrow j}(e) \rrbracket^\sharp$  is implemented by the reduction operator with the same name,  $\llbracket \text{moveRgn}(\alpha, \beta) \rrbracket^\sharp$  by the substitution of the source region variable  $\alpha$  by the target  $\beta$ , and  $\llbracket \text{move}(e, \alpha) \rrbracket^\sharp$  by the exposal of a pointsto fact  $e \mapsto \{\dots\}_{\beta_k}$  from a symbolic heap followed by the renaming of its subscript  $\beta_k$  by  $\alpha$ . The remaining operator is  $\text{enableAbs}$ , which we define by  $\text{enableAbs}(d) = \{\text{moveRgn}(\alpha, \beta) \mid \text{abs}(\llbracket \text{moveRgn}(\alpha, \beta) \rrbracket^\sharp(d)) \neq d\}$ . This operator returns all the region movement operations that enable further abstraction of its input  $d$ .

## 7 Experiments

We have implemented an interprocedural version of the analysis (based on the RHS algorithm [10]), and applied it to verify the memory safety of two types

**Fig. 7** Experimental result. Used Intel Core i7 2.66GHz with 8GB memory.

filename	# of lines	analysis time (sec)		speedup (A/B)	# of trans discovered
		(A) old	(B) new		
list-dio-sim.c	110	3.12	1.56	2.0	2
list-dio.c	134	–	3.95	–	4
many-keys-3.c	92	1.65	0.72	2.3	2
many-keys-4.c	98	8.16	1.22	6.7	3
many-lists-3.c	106	1.90	1.37	1.4	3
many-lists-4.c	124	12.53	3.05	4.1	4
cache-1.c	88	1.29	0.97	1.3	9
cache-2.c	93	14.70	1.81	7.8	11
linux/block/deadline-iosched-sim.c	1,941	237.67	32.76	7.3	4
linux/block/deadline-iosched-sim2.c	1,968	5,399.73	100.06	54.0	4
linux/block/deadline-iosched.c	2,131	–	364.45	–	5
linux/fs/afs/server-sim.c	712	705.67	22.61	31.2	9
linux/fs/afs/server.c	1,084	–	1,932.65	–	13

of programs. The first are toy examples of modest size and with just enough structure to warrant an overlaid analysis. The second are programs lifted from the Linux 2.6.37 code base. The results of our experiments appear in Figure 7.

The figure also includes the numbers obtained by applying our previous analysis built in 2008 to the same examples. This previous analysis couples the sub-analyses more tightly (using the abstract domain  $\mathcal{P}(\text{SH}_1 \times \dots \times \text{SH}_n) \cup \{\top\}$ ), it does not use ghost instructions, and it transfers information among sub-analyses more frequently than our current analysis. The figure shows that the current implementation performs better, and the performance gain becomes more significant, when a program becomes bigger or more complicated. There are also a number of programs that cannot be analysed at all by the old analysis.

The right-most column records the number of  $\text{trans}_{i \rightarrow j}(e)$  inserted by the pre-analysis of our current implementation. It shows that very little communication is happening. We consider this a primary factor of the efficiency of the analysis.

The benchmark set consists of the following programs, and can be found at <https://sites.google.com/site/overlaiddata/>

- `list-dio` is an abstract version of the deadline IO scheduler. It uses two doubly-linked lists instead of list and tree. The `sim` version skips the request-move routine, which cannot be verified by the old analysis.
- `many-keys` has an overlaid data structure of doubly-linked lists that are ordered by different keys. The number of lists is annotated in the filename.
- `many-lists` uses multiple doubly-linked lists implemented by different fields. These lists do not share nodes, so they do not form an overlaid data structure. However, our analysis can analyse each list separately, using a distinct conjunct for each list. The number of lists is annotated.
- `cache` has one doubly-linked list and pointers to cells in the list that were recently accessed. We can separately analyse the list and pointers by using our technique. The number of cache pointers is annotated.
- `block/deadline-iosched.c` has an overlaid data structure of a doubly-linked list and a red-black tree to maintain a list of requests. The original

source was modified as follows: irrelevant fields and procedures such as ones for locks, and language constructors such as arrays that our analyser does not support, were removed, and assumptions were inserted to tree operations to compensate for our inaccurate tree abstraction. The `sim/sim2` version skips procedures that the old analyser cannot verify due to its imprecision, as well as procedures of high analysis cost.

- `fs/afs/server.c` also has a similar data structure to maintain servers but it has one more component of doubly-linked list for removing servers: Servers to be removed are additionally connected to the `graveyard` list. So, in this case, the overlaid data structure consists of three components.

**Conclusion** In this paper, we have presented a static analysis for overlaid data structures, capable of verifying memory safety of real world programs. Our insight is to decompose an overlaid data structure to its components, and to track components using sub-analyses as independently as possible, while allowing communications among them using ghost instructions. Besides the progress in verifying more challenging data structures, we hope that our work has provided a further evidence that with proper understanding of more programming patterns in systems code, together with specialized abstractions, one can design effective automatic verifiers for ever-larger classes of real-world systems programs.

## References

1. G. Arnold, R. Manevich, M. Sagiv, and R. Shaham. Combining shape analyses by intersecting abstractions. In *VMCAI*, 2006.
2. T. Ball, A. Podelski, and S. Rajamani. Boolean and cartesian abstraction for model checking c programs. In *TACAS*, 2001.
3. C. Calcagno, D. Distefano, P. O’Hearn, and H. Yang. Compositional shape analysis by means of bi-abduction. In *POPL*, 2009.
4. R. Cherini, L. Rearte, and J. Blanco. A shape analysis for non-linear data structures. In *SAS*, 2010.
5. P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *POPL*, 1979.
6. D. Distefano, P. O’Hearn, and H. Yang. A local shape analysis based on separation logic. In *TACAS*, 2006.
7. P. Hawkins, A. Aiken, and K. Fisher. Reasoning about shared mutable data structures. Manuscript, 2010.
8. P. Hawkins, A. Aiken, K. Fisher, M. Rinard, and M. Sagiv. Data structure fusion. In *APLAS*, 2010.
9. J. Kreiker, H. Seidl, and V. Vojdani. Shape analysis of low-level c with overlapping structures. In *VMCAI*, 2010.
10. T. Reps, S. Horwitz, and S. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL*, 1995.
11. J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, 2002.
12. M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM TOPLAS*, 24(3):217–298, 2002.
13. H. Yang, O. Lee, J. Berdine, C. Calcagno, B. Cook, D. Distefano, and P. O’Hearn. Scalable shape analysis for systems code. In *CAV*, 2008.