Noname manuscript No.
(will be inserted by the editor)

A divide-and-conquer approach for analysing overlaid data structures

Oukseh Lee · Hongseok Yang · Rasmus Petersen

the date of receipt and acceptance should be inserted later

Abstract We present a static program analysis for overlaid data structures such that 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. 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. 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.

Keywords automatic program verification · memory safety · shape analysis · overlaid data structure

We want to thank Gilad Arnold, Patrick Cousot, Peter Hawkins, Peter O'Hearn, Martin Rinard, Noam Rinetzky, Xavier Rival, and John Wickerson for helpful comments. This work was supported by EPSRC, and Lee by the Engineering Research Center of Excellence Program of Korea Ministry of Education, Science and Technology (MEST) / National Research Foundation of Korea (NRF) (Grant 2011-0000968).

O. Lee
Dept. of CSE, Hanyang University, Sangnok-gu, Ansan, Gyeonggi 426-791, South Korea
E-mail: oukseh@hanyang.ac.kr

H. Yang
Department of Computer Science, University of Oxford, Oxford OX1 3QD, United Kingdom
E-mail: hongseok.yang@cs.ox.ac.uk

R. Petersen
Queen Mary University of London, London E1 4NS, United Kingdom
E-mail: rusmus@eecs.qmul.ac.uk

1 Introduction

Recent advances in verification research have resulted in successful industrial-strength software verifiers, such as Microsoft SDV [2] and Astrée [4]. 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 they 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 [5, 17], 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 have 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 objective here is not to verify toy problems of overlaid data structures, but to prove real-world examples. In fact, we created an analyser in 2008 that could show the memory safety of toy examples, but this analyser did not scale to verify real code like the deadline IO scheduler for several fundamental reasons (see Section 10). 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 examples [11, 9, 6, 16].

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:

1. Run multiple sub-analyses that track information about standard 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. This is reminiscent of Cartesian abstraction [3].
2. 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 [7]); the memory safety of the 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 at allowing only necessary information

to be transferred among sub-analyses and at as few program points as possible. To achieve the aim, the analysis uses ghost states, special instructions for modifying ghost states, and algorithms that insert those instructions before or during the main phase of the analysis.

The rest of the paper is organized as follows. Related works are given in Section 2. Section 3 gives an informal account of the analysis, and Section 4 details the ghost states that allow communication among sub-analyses. Section 5 describes the abstract domain of our analysis where abstract states are seen to be tuples of those of the sub-analyses. Section 6 describes the weak reduction operator that enables communication among the sub-analyses. We then move on to the formal account of the analysis itself: pre-analyses (Section 7 and 8) and a main analysis (Section 9). Section 10 shows some experimental results and Section 11 concludes.

An extended abstract [13] of this paper was presented at the CAV 2011 conference. Compared to the conference version, this paper expands on the pre-analysis phase of our program analyser. This phase modifies an input program by automatically inserting ghost instructions, which control information transfer among sub-analyses. It is one of the main reasons that our analysis is efficient.

2 Related work

In addition to the approaches cited already, we discuss three other related works. The first is the synthesis approach by Hawkins *et al.* [10], 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 at computing the exact meet, not an efficient over-approximation of the meet as in this paper. The last is the Hob system by Kuncak *et al.* [12], where one can apply different analysis plug-ins for different data structures, combine the analysis results, and verify that values stored in these data structures are properly related. This system regards an overlaid data structure as a single data structure, and requires a plug-in for its analysis. This requirement can be met by the analysis in this paper.

3 Informal description

We start with an informal description of our analysis using the baby IO scheduler in Fig. 1, which is modelled on the Linux deadline IO scheduler.

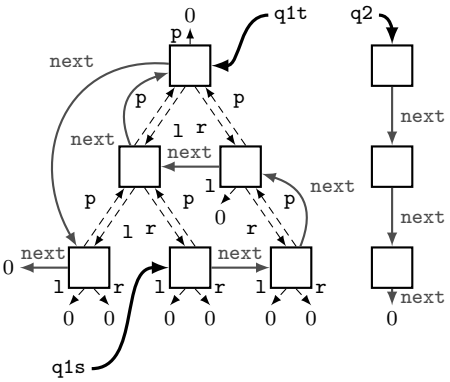
Our baby IO scheduler schedules IO requests using two disjoint queues. When a request arrives, it is stored in the first queue. Later the request is selected according to a scheduling policy, processed, and moved to the second queue. In order to help the performance of the scheduling, the first queue 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

Fig. 1 Baby IO scheduler and snapshot of the data structure used

```
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)
}
```



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 Fig. 1.

The `move_request` function in Fig. 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 exploit fully 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 φ describes the heap only in terms of list fields and ψ 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, φ and ψ use what we call region variables α, β, γ, which denote sets of memory addresses. Concretely, our analysis infers that the data structures of our IO scheduler normally satisfy the following assertion:

$$(\mathsf{ls}(\mathtt{q1s})_\alpha * \mathsf{ls}(\mathtt{q2})_\beta) \ \wedge \ (\mathsf{tr}(\mathtt{q1t})_\alpha * \mathsf{true}_\beta). \tag{1}$$

The predicate $\mathsf{ls}(x)$ means a singly-linked list starting from the address x, and $\mathsf{tr}(y)$ 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 α, and those of the nodes in the list `q2` the set β. 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 α, while the addresses of all the other nodes make the set β. Note that each conjunct has its own characterisations of α and β. To be consistent, both characterisations of α 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 the 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 Fig. 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 analysis 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 $\mathtt{trans}_{\mathrm{list}\to\mathrm{tree}}(\mathtt{c})$ and $\mathtt{move}(\mathtt{c}, \gamma)$ before and after `tree_remove`, as shown in Fig. 1. The instruction $\mathtt{trans}_{\mathrm{list}\to\mathrm{tree}}(\mathtt{c})$ 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 [7]. The instruction is inserted here because the pre-analysis conjectures that information about cell `c` at this program point will be necessary for verification. The second instruction $\mathtt{move}(\mathtt{c}, \gamma)$ tells the analysis to manage the values of region variables by moving the address `c` from its current region to the region γ. We defer the details of the pre-analysis to Section 7.

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

$$(\mathsf{ls}(\mathtt{q1s})_\alpha * \mathtt{c} \mapsto \{\}_\alpha * \mathsf{ls}(\mathtt{q2})_\beta) \ \wedge \ (\mathsf{tr}(\mathtt{q1t})_\alpha * \mathsf{true}_\beta) \tag{2}$$

for the true branch of the following conditional statement. Compared to the original in (1), the assertion has additionally $\mathtt{c} \mapsto \{\}_\alpha$ in the first conjunct, and this additional predicate describes the cell `c` removed from the list `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 α. The next instruction $\mathtt{trans}_{\mathrm{tree}\to\mathrm{list}}(\mathtt{c})$ makes the analysis transfer the information about cell `c` from the first to the second conjunct, which gives the assertion:

$$(\mathsf{ls}(\mathtt{q1s})_\alpha * \mathtt{c} \mapsto \{\}_\alpha * \mathsf{ls}(\mathtt{q2})_\beta) \ \wedge \ (\varphi(\mathtt{q1t}, \mathtt{c}, \alpha) * \mathsf{true}_\beta). \tag{3}$$

Here $\varphi(\mathtt{q1t}, \mathtt{c}, \alpha)$ is an assertion with free variables $\mathtt{q1t}, \mathtt{c}, \alpha$, and it describes a tree with root `q1t` and a normal node `c` such that all nodes of the tree form the set α.[1] 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 over-approximate the instruction's output states by the assertion:

$$(\mathsf{ls}(\mathtt{q1s})_\alpha * \mathtt{c} \mapsto \{\}_\alpha * \mathsf{ls}(\mathtt{q2})_\beta) \ \wedge \ (\mathsf{tr}(\mathtt{q1t})_\alpha * \mathtt{c} \mapsto \{\}_\alpha * \mathsf{true}_\beta). \tag{4}$$

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

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

$$(\mathsf{ls}(\mathtt{q1s})_\alpha * \mathtt{c} \mapsto \{\}_\gamma * \mathsf{ls}(\mathtt{q2})_\beta) \ \wedge \ (\mathsf{tr}(\mathtt{q1t})_\alpha * \mathtt{c} \mapsto \{\}_\gamma * \mathsf{true}_\beta). \tag{5}$$

Semantically, this change means that the allocated cell `c` is moved from the set α to the set γ, which only contains `c`. The decision for singling out `c` and putting it in a separate set γ is made because the pre-analysis detected a possibility of moving cell `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 `c` to the second queue `q2`. The analysis tracks the move of the cell, using its list sub-analysis, and transforms (5) to the assertion:

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

The variable a has the old value of `q2`, and b the old value of `c`.

Note that the sub-formula $\mathtt{q2} \mapsto \{\mathtt{next}{:}a\}_\gamma * \mathsf{ls}(a)_\beta$ in (6) describes a list starting from `q2` of length at least one (because of cell `q2`). The list sub-analysis decides

[1] Concretely, $\varphi(\mathtt{q1t}, \mathtt{c}, \alpha)$ is $\exists uvwxy.\ \mathsf{tseg}(\mathtt{q1t}, 0, \mathtt{c}, u)_\alpha * \mathtt{c} \mapsto \{\mathtt{p}{:}u, \mathtt{l}{:}v, \mathtt{r}{:}x\}_\alpha * \mathsf{tseg}(v, \mathtt{c}, 0, w)_\alpha * \mathsf{tseg}(x, \mathtt{c}, 0, y)_\alpha$ where `tseg` is a tree segment predicate explained in Section 5.

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 $\mathsf{ls}(\mathtt{q2})$. To do this, the analysis inserts the instruction $\mathtt{moveRgn}(\gamma, \beta)$ for moving all cells in γ to β, and analyzes the inserted instruction:

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

The reason for inserting the instruction $\mathtt{moveRgn}(\gamma, \beta)$ 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 $\mathtt{q2}$ and the tail a are in the same set β, so the abstraction applies and gives the final result:

$$(\mathsf{ls}(\mathtt{q1s})_\alpha * \mathsf{ls}(\mathtt{q2})_\beta) \ \wedge\ (\mathsf{tr}(\mathtt{q1t})_\alpha * \mathsf{true}_\beta). \qquad (8)$$

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

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

4 Formal setting for region variables

4.1 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 Addrs of addresses; a set Vars of program variables x, y, z; sets Fields and $\mathsf{Regions}$ that respectively contain field names $\mathtt{f}, \mathtt{g}$ of heap cells and region variables α, β, γ. We assume that a fixed constant $\mathtt{null}$ is not in Addrs. The storage model is defined by the following equations:

$$\mathsf{Vals} = \mathsf{Addrs} \cup \{\mathtt{null}\} \quad \mathsf{Stacks} = \mathsf{Vars} \to \mathsf{Vals} \quad \mathsf{Heaps} = \mathsf{Addrs} \rightharpoonup_{\mathit{fin}} (\mathsf{Fields} \rightharpoonup \mathsf{Vals})$$
$$\mathsf{Partitions} = \mathsf{Regions} \to \mathcal{P}(\mathsf{Addrs}) \quad \mathsf{States} = \mathsf{Stacks} \times \mathsf{Heaps} \times \mathsf{Partitions}$$

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

$$(\mathsf{dom}(h) = \cup_{\alpha \in \mathsf{Regions}} \eta(\alpha)) \ \wedge\ (\forall \alpha, \beta \in \mathsf{Regions}.\ \alpha \not\equiv \beta \implies \eta(\alpha) \cap \eta(\beta) = \emptyset).$$

A state (s, h, η) is *well-formed* when (h, η) 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 about an allocated address l can be approximated by the region variable α 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.

Fig. 2 Semantics of sample assertions. We assume a function $[\![e]\!]$ from Stacks to Vals that defines the meaning of an expression e, and a mapping $[\![p]\!]$ from tuples of values to heaps that specifies the semantics of a primitive predicate p.

$$
\begin{array}{lcl}
s,h,\eta \models \varphi_\alpha & \iff & s,h,\eta \models \varphi \text{ and } \mathsf{dom}(h) = \eta(\alpha) \\
s,h,\eta \models e \in \alpha & \iff & [\![e]\!]s \in \eta(\alpha) \\
s,h,\eta \models e \mapsto \{\mathtt{f} : \mathbf{e}\} & \iff & \mathsf{dom}(h) = \{[\![e]\!]s\} \text{ and } h([\![e]\!]s)\mathtt{f}_i = [\![e_i]\!]s \text{ for all } 1 \le i \le |\mathtt{f}| \\
s,h,\eta \models p(e_1,\ldots,e_n) & \iff & h \in [\![p]\!]([\![e_1]\!]s,\ldots,[\![e_n]\!]s) \\
s,h,\eta \models \mathsf{emp} & \iff & \mathsf{dom}(h) = \emptyset \text{ and } \eta(\alpha) = \emptyset \text{ for all } \alpha \\
s,h,\eta \models P * Q & \iff & \exists h_1,h_2,\eta_1,\eta_2.\ (h_1,\eta_1) \bullet (h_2,\eta_2) = (h,\eta) \\
 & & \qquad \text{and } s,h_1,\eta_1 \models P_1 \text{ and } s,h_2,\eta_2 \models P_2
\end{array}
$$

4.2 Assertions

Assertions φ describe properties of states, and are defined as follows:

$$
\begin{array}{rcl}
e & ::= & x \mid \mathtt{null} \\
\varphi & ::= & \varphi_\alpha \mid e = e \mid e \in \alpha \mid e \mapsto \{\mathtt{f} : \mathbf{e}\} \mid p(\mathbf{e}) \\
 & \mid & \mathsf{emp} \mid \varphi * \varphi \mid \mathsf{true} \mid \varphi \wedge \varphi \mid \neg\varphi \mid \exists x.\, \varphi
\end{array}
$$

This is a variant of the assertion language from separation logic [15]. The first φ_α says that the heap satisfies φ and all the allocated addresses in the heap form the set α. 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 $x \mapsto \{\mathtt{f} : \mathbf{e}\}$ means a heap containing only one cell x that stores $\mathbf{e}$ in fields $\mathtt{f}$. This definition does not require that $\mathtt{f}$ be the only field in cell x. Hence, the cell x can have fields other than $\mathtt{f}$. The following case $p(e_1,\ldots,e_n)$ 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 φ and the other ψ. 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 Fig. 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:

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

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 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.

Fig. 3 Semantics of sample primitive instructions. We assume a function $[\![b]\!]$ from Stacks to $\{true, false\}$ that defines the meaning of a Boolean expression b.

$$
\begin{aligned}
[\![\texttt{assume}(b)]\!](s,h,\eta) &= \textbf{if } ([\![b]\!]s = true) \textbf{ then } \{(s,h,\eta)\} \textbf{ else } \emptyset \\
[\![x := \texttt{new}_{\alpha,F}()]\!](s,h,\eta) &= \{(s[x \mapsto l], h[l \mapsto v], \eta[\alpha \mapsto \eta(\alpha) \cup \{l\}]) \mid \\
&\qquad l \in \mathsf{Addrs} \setminus \mathsf{dom}(h) \text{ and } v \text{ is a function from } F \text{ to Vals}\} \\
[\![\texttt{move}(e,\alpha)]\!](s,h,\eta) &= \textbf{if } \neg(\exists\beta.\, [\![e]\!]s \in \eta(\beta)) \textbf{ then } err \\
&\qquad \textbf{else } \{(s, h, \eta[\beta \mapsto \eta(\beta) \setminus \{[\![e]\!]s\}, \alpha \mapsto \eta(\alpha) \cup \{[\![e]\!]s\}])\} \\
[\![\texttt{moveRgn}(\alpha,\beta)]\!](s,h,\eta) &= \{(s, h, \eta[\alpha \mapsto \emptyset, \beta \mapsto \eta(\alpha) \cup \eta(\beta)])\}
\end{aligned}
$$

4.3 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 $\mathsf{entry}, \mathsf{exit} \in V$ 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:

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

Most cases are standard imperative operations. For instance, $\texttt{assume}(b)$ checks whether the input state satisfies a Boolean expression b. If so, it skips. Otherwise, it diverges. The only exceptions are the last three cases. The instruction $x := \texttt{new}_{\alpha,F}()$ allocates a new cell with fields F, and puts this cell into the region α. The fields of this new cell are uninitialized, and may contain any non-deterministically chosen values. The next two $\texttt{move}(e,\alpha)$ and $\texttt{moveRgn}(\alpha,\beta)$ are ghost instructions that mainly manipulate the region-map parts of states. When cell e is allocated in the input state, $\texttt{move}(e,\alpha)$ removes this cell from its current region, and puts it in the region α. The instruction $\texttt{moveRgn}(\alpha,\beta)$ moves all the cells in the region α to the region β. Hence, at the end of this instruction, α contains no cells, while β contains all cells that used to be in α. 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 to put each data structure, such as a list or a tree, in its own partition described by some region variable α because then knowing $e \in \alpha$ is sufficient to identify the data structure containing e.

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

5 Abstract states

Our abstract domain consists of assertions of the form:

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

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 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 parametrised 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:

$$
\begin{array}{lll}
\Pi & ::= & \mathsf{true} \mid e = e \mid e \neq e \mid \Pi \wedge \Pi & \textit{Pure formulae} \\
\Sigma & ::= & \mathsf{true}_\alpha \mid (e \mapsto \{\mathtt{f} : \mathbf{e}\})_\alpha \mid (p(\mathbf{e}))_\alpha \mid \mathsf{emp} \mid \Sigma * \Sigma & \textit{Spatial formulae} \\
H & ::= & \exists \mathbf{x}.\, \Pi \wedge \Sigma & \textit{Symbolic heaps}
\end{array}
$$

The Π part of a symbolic heap describes the information about variables, and the Σ 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 [8]. We write SH for the set of all symbolic heaps.

Second, we define a set of assertions used by each sub-analysis i. Let SH_i be the set of symbolic heaps H such that all points-to predicates $(e \mapsto \{\mathtt{f} : \mathbf{e}\})_\alpha$ in H mention only fields in F_i (i.e., $\mathtt{f} \subseteq F_i$), and all primitive predicates $(p(\mathbf{e}))_\alpha$ in H belong to F_i (i.e., $p \in F_i$). The domain for the sub-analysis i is $\mathcal{D}_i = \mathcal{P}_{\mathit{fin}}(\mathsf{SH}_i)$. The finite powerset operator is used here to express finite disjunction. For instance, the set $\{H_1, \ldots, H_m\} \in \mathcal{D}_i$ means the disjunction $H_1 \vee \ldots \vee H_m$.

Finally, the abstract domain $\mathcal{D}(\mathcal{F})$ is defined by $\mathcal{D} = \mathcal{D}_1 \times \ldots \times \mathcal{D}_n \cup \{\top\}$ 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 $(\{\varphi_{1,1}, \ldots, \varphi_{1,m_1}\}, \{\varphi_{2,1}, \ldots, \varphi_{2,m_2}\}, \ldots, \{\varphi_{n,1}, \ldots, \varphi_{n,m_n}\})$ 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 ith 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 point-wise. Then, the lattice operations of $\mathcal{D}(\mathcal{F})$ are obtained by extending corresponding operations on the $\mathcal{D}_i$'s point-wise. For instance, the join $d \sqcup d'$ is given by $(d_1 \sqcup d'_1, \ldots, d_n \sqcup d'_n)$.

Fig. 4 Subroutines getRegion and caseSH. 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.

$\mathsf{get}_{\Pi}(e,\ \mathsf{emp}) = \mathsf{NoInfo}$
$\mathsf{get}_{\Pi}(e,\ e'{\mapsto}\{\mathtt{f}{:}\mathbf{e''}\}_\alpha * \Sigma) = \textbf{if}\ (\Pi \vdash e = e')\ \textbf{then}\ \alpha\ \textbf{else}\ \mathsf{get}_{\Pi}(e,\ \Sigma)$
$\mathsf{get}_{\Pi}(e,\ p(\mathbf{e'})_\alpha * \Sigma) = \textbf{if}\ (\Pi \wedge p(\mathbf{e'}) \vdash e{\mapsto}\{\} * \mathsf{true})\ \textbf{then}\ \alpha\ \textbf{else}\ \mathsf{get}_{\Pi}(e,\ \Sigma)$
$\mathsf{getRegion}(e,\ H) = \textbf{let}\ (\exists \mathbf{x}.\ \Pi \wedge \Sigma) = H\ \textbf{in}\ \mathsf{get}_{\Pi}(e,\ \Sigma)$
$\mathsf{caseID}(e, \mathsf{tr}(e_0)_\alpha) =$
$\{(uvwxy,\ \mathsf{tseg}(e_0, 0, e, u)_\alpha * e \mapsto \{\mathtt{p}{:}u, \mathtt{l}{:}v, \mathtt{r}{:}w\}_\alpha * \mathsf{tseg}(v, e, 0, x)_\alpha * \mathsf{tseg}(w, e, 0, y)_\alpha)\}$
$\mathsf{caseID}(e, \mathsf{tseg}(e_0, e_1, e_2, e_3)_\alpha) =$
$\{(uvwx,\ \mathsf{tseg}(e_0, e_1, e, u)_\alpha * e \mapsto \{\mathtt{p}{:}u, \mathtt{l}{:}v, \mathtt{r}{:}w\}_\alpha * \mathsf{tseg}(v, e, e_2, e_3)_\alpha * \mathsf{tseg}(w, e, 0, x)_\alpha),$
$(uvwx,\ \mathsf{tseg}(e_0, e_1, e, u)_\alpha * e \mapsto \{\mathtt{p}{:}u, \mathtt{l}{:}v, \mathtt{r}{:}w\}_\alpha * \mathsf{tseg}(v, e, 0, x)_\alpha * \mathsf{tseg}(w, e, e_2, e_3)_\alpha)\}$
$\mathsf{case}_{(e,\alpha,\Pi)}(\Sigma,\ \mathsf{emp}) = \emptyset$
$\mathsf{case}_{(e,\alpha,\Pi)}(\Sigma,\ e'{\mapsto}\{\mathtt{f}{:}\mathbf{e''}\}_\beta * \Sigma') =$
$\quad \textbf{if}\ (\Pi \vdash e \neq e'\ \text{or}\ \alpha \not\equiv \beta)\ \textbf{then}\ \mathsf{case}_{(e,\alpha,\Pi)}(\Sigma * e'{\mapsto}\{\mathtt{f}{:}\mathbf{e''}\}_\beta,\ \Sigma')$
$\quad \textbf{else}\ \{([],\ e{=}e',\ \Sigma * e'{\mapsto}\{\mathtt{f}{:}\mathbf{e''}\}_\beta * \Sigma')\} \cup \mathsf{case}_{(e,\alpha,\Pi)}(\Sigma * e'{\mapsto}\{\mathtt{f}{:}\mathbf{e''}\}_\beta,\ \Sigma')$
$\mathsf{case}_{(e,\alpha,\Pi)}(\Sigma,\ p(\mathbf{e'})_\beta * \Sigma') =$
$\quad \textbf{if}\ (\alpha \not\equiv \beta)\ \textbf{then}\ \mathsf{case}_{(e,\alpha,\Pi)}(\Sigma * p(\mathbf{e'})_\beta,\ \Sigma')$
$\quad \textbf{else}\ \{(\mathbf{a}, \mathsf{true}, \Sigma * \Sigma'' * \Sigma')\ \mid\ (\mathbf{a}, \Sigma'') \in \mathsf{caseID}(e, p(\mathbf{e'}))\} \cup \mathsf{case}_{(e,\alpha,\Pi)}(\Sigma * p(\mathbf{e'})_\beta, \Sigma')$
$\mathsf{caseSH}(e, \alpha, H) =$
$\quad \textbf{let}\ (\exists \mathbf{x}.\ \Pi \wedge \Sigma) = H\ \textbf{in}\ \{\exists \mathbf{xa}.\ (\Pi \wedge \Pi') \wedge \Sigma'\ \mid\ (\mathbf{a},\ \Pi',\ \Sigma') \in \mathsf{case}_{(e,\alpha,\Pi)}(\mathsf{emp},\ \Sigma)\}$

6 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 α containing this cell. For instance, consider the abstract state:

$$\big(\mathtt{x} \mapsto \{\mathtt{next}{:}0\}_\alpha\ \vee\ (\exists a.\, \mathtt{x} \mapsto \{\mathtt{next}{:}a\}_\alpha * \mathsf{ls}(a)_\alpha)\big)\ \ \wedge\ \ \mathsf{tr}(\mathtt{y})_\alpha$$

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

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

Here the predicate $\mathsf{tseg}(a, b, c, d)$ 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 tseg when we do not want to specify the parameter.[2] Note that the second conjunct now talks about the allocatedness of cell x and its membership of α.

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

$\mathsf{trans}_i(e)(H : \mathsf{SH}, H' : \mathsf{SH}_i) : \mathcal{D}_i =$
$\quad \textbf{let}\ R = \mathsf{getRegion}(e, H)\ \textbf{in}\ \textbf{if}\ (R = \mathsf{NoInfo})\ \textbf{then}\ \{H'\}\ \textbf{else}\ \mathsf{caseSH}(e, R, H').$

[2] Formally, $\varphi * \mathsf{tseg}(a, b, c, \_)$ is an abbreviation for $\exists d.\varphi * \mathsf{tseg}(a, b, c, d)$ for a fresh d.

The operator $\mathsf{trans}_i(e)(H, H')$ 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 $\mathsf{getRegion}(e, H)$, which has two possible outcomes. The first outcome is 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 trans. The second outcome is a region variable α satisfying the entailment $H \vdash e \in \alpha$, which means that according to H, the region variable α 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 SH_i while ensuring the soundness condition expressed below:

$$\mathcal{H} = \mathsf{caseSH}(e, \alpha, H') \implies (e \in \alpha \wedge H') \vdash \bigvee \mathcal{H}.$$

One implementation of $\mathsf{getRegion}$ and caseSH is given in Fig. 4.

For sub-analysis ids i, j and an expression e, we define our weak reduction operator $\mathsf{trans}_{i\to j}(e) : \mathcal{D} \to \mathcal{D}$ by $\mathsf{trans}_{i\to j}(e)(\top) = \top$ and

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

This operator applies trans_j to all possible symbolic-heap combinations from the i and jth components of d, and uses the result to update the jth 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 a 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.

7 Inserting the weak reduction instructions

The first step of our analysis is to insert `trans` instructions that apply weak reduction into the input program. Intuitively, the pre-analysis inserts $\texttt{trans}_{i\to j}(e)$ 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 j is *unlikely* to infer that e is allocated or `null`.
3. But the sub-analysis i is *likely* to infer that e is allocated or `null`.

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.

7.1 Semantics of component-wise execution with symbolic stacks

In order to make our intuition clear, we start from a modified semantics that explicitly describes the information about the stack and the heap tracked by each

sub-analysis. The input program is a control flow graph $G = (V, E, \mathsf{entry}, \mathsf{exit}, L)$. Since G represents a normal C program, its labelling L does not use our ghost instructions or weak reduction operator, and `new` instructions do not have region variables. We assume that the standard semantics $[\![c]\!](s, h)$ is given.

A symbolic stack σ is an extension of pure formulae with expressions involving addresses, disjunction and existential quantifiers. Using a symbol $l \in \mathsf{Addrs}$, we describe the syntax of symbolic stacks as follows:

$$E ::= x \mid \mathtt{null} \mid l \qquad \sigma ::= E = E \mid E \neq E \mid \sigma \wedge \sigma \mid \sigma \vee \sigma \mid \exists x.\sigma \quad \textit{Symbolic stacks}$$

Symbolic stacks have the standard meaning similar to that for pure formulae, which can be formalised by the satisfaction relation $s \models \sigma$.

We consider the following operations on symbolic stacks:

$$\begin{aligned}
\sigma(x) \quad &= \textbf{if } ((\exists s.s \models \sigma) \wedge (\forall s.s \models \sigma \Longrightarrow s(x) = v)) \text{ for a unique } v \textbf{ then } v \\
&\quad\ \textbf{else } \text{undefined} \\
\sigma[x \mapsto a] &= \exists x'.(\sigma[x'/x] \wedge x = a[x'/x]) \text{ when } a \text{ is a value or an expression}
\end{aligned}$$

Note that $\sigma(x)$ is defined only when there exists at least one stack that satisfies σ, and all stacks that satisfy σ give the same value for variable x. These operations let us reuse the definition of $[\![c]\!](s, h)$ and specify $[\![c]\!](\sigma, h)$ simply by replacing the stack lookup and update operations with the corresponding ones above.

The semantics of this section is called component-wise semantics because a state in this new semantics consists of multiple components, each of which is a pair of a symbolic stack and a heap. The number of components is $|\mathcal{F}|$, that of sub-analyses, and the heap of the ith component tracks the values of only those fields in F_i. Such a multiple-component state $\rho = ((\sigma_1, h_1), \ldots, (\sigma_n, h_n))$ is *well-formed* if and only if

1. there exists a stack s such that $s \models \sigma_1 \wedge \ldots \wedge \sigma_n$;
2. for all program variables x, $(\sigma_1 \wedge \ldots \wedge \sigma_n)(x)$ is defined;
3. for all $1 \leq i, j \leq n$, $\mathsf{dom}(h_i) = \mathsf{dom}(h_j)$; and
4. for all $1 \leq i \leq n$ and for all locations $l \in \mathsf{dom}(h_i)$, $\mathsf{dom}(h_i(l)) \subseteq F_i$.

We will consider well-formed states only in this paper.

The execution of an instruction c on the multiple-component states is based on the separate execution $[\![c]\!]_i$ of the instruction for each component i (Fig. 5), which accesses the symbolic stack and the heap of the ith component only. Concretely, $[\![x := \mathtt{new}_F()]\!]_i$ allocates a cell with only those fields associated with the component i, i.e., the fields in F_i. The heap update $[\![e.\mathtt{f} := e']\!]_i$ changes the heap cell only when `f` belongs to F_i. Otherwise, this instruction is skip. Similarly, $[\![x := e.\mathtt{f}]\!]_i$ does the usual lookup of the heap and the update on the symbolic stack only if the field `f` is associated with the component i. If this is not the case, the semantics sets x to undefined because it is impossible to get the value of $e.\mathtt{f}$ from the heap of the ith component. The assignment $[\![x := e]\!]_i$ does not depend on i, and it always updates the symbolic stack of the ith component such that $x = e$ holds. Similarly, $[\![\mathtt{assume}(b)]\!]_i$ is independent of i, and it conjoins the predicate b with the symbolic stack of the ith component. If this conjunction leads to inconsistency, the resulting state is ill-formed, and it gets filtered out. The execution of the remaining instructions is standard.

Fig. 5 Semantics of component-wise execution. $[\![c]\!]_{\mathsf{CE}}$ denotes the semantics of component-wise execution, $[\![c]\!]_i$ denotes the semantics for executing the ith component, and $[\![c]\!]$ denotes the original semantics.

$$
\begin{aligned}
[\![c]\!]_{\mathsf{CE}}((\sigma_1,h_1),\ldots,(\sigma_n,h_n)) &= \{\rho \mid \rho \in [\![c]\!]_1(\sigma'_1,h_1) \times \ldots \times [\![c]\!]_n(\sigma'_n,h_n) \text{ and } \rho \text{ is well-formed}\} \\
&\quad \text{where } \sigma'_i = \sigma_i \wedge \mathsf{reveal}(i,c,\sigma_1 \wedge \ldots \wedge \sigma_n) \text{ for all } 1 \le i \le n \\
[\![x := \mathtt{new}_F()]\!]_i(\sigma,h) &= [\![x := \mathtt{new}_{F \cap F_i}()]\!](\sigma,h) \\
[\![x := e.\mathtt{f}]\!]_i(\sigma,h) &= \textbf{if } (\mathtt{f} \in F_i) \textbf{ then } [\![x := e.\mathtt{f}]\!](\sigma,h) \textbf{ else } \{(\exists x.\sigma,h)\} \\
[\![e.\mathtt{f} := e']\!]_i(\sigma,h) &= \textbf{if } (\mathtt{f} \in F_i) \textbf{ then } [\![e.\mathtt{f} := e']\!](\sigma,h) \textbf{ else } \{(\sigma,h)\} \\
[\![x := e]\!]_i(\sigma,h) &= \{(\sigma[x \mapsto e],h)\} \\
[\![\mathtt{assume}(b)]\!]_i(\sigma,h) &= \textbf{if } (\exists s.\, s \models \sigma \wedge b) \textbf{ then } \{(\sigma \wedge b,h)\} \textbf{ else } \emptyset \\
[\![c]\!]_i(\sigma,h) &= [\![c]\!](\sigma,h) \quad \text{(for other instructions)}
\end{aligned}
$$

$\mathsf{reveal}(i,c,\sigma) = \bigwedge \{x = \sigma(x) \mid (i,x) \in \mathsf{need}(c)\}$ with need below :

instr c	$\mathsf{need}(c)$
$x := e.\mathtt{f}$	$\{(i,e) \mid \mathtt{f} \in F_i\}$
$e.\mathtt{f} := e'$	$\{(i,e),(i,e') \mid \mathtt{f} \in F_i\}$
$\mathtt{free}(e)$	$\{(i,e) \mid 1 \le i \le n\}$
other instructions	$\emptyset$

where the notation $\{(i,e),(i,e') \mid \mathtt{f} \in F_i\}$ means $\{(i,e) \mid \mathtt{f} \in F_i\} \cup \{(i,e') \mid \mathtt{f} \in F_i\}$

The definition of $[\![c]\!]_{\mathsf{CE}}$ in Fig. 5 describes how an instruction c transforms a given multiple-component state $((\sigma_1,h_1),\ldots,(\sigma_n,h_n))$. It first strengthens symbolic stacks $\sigma_1 \ldots \sigma_n$ to $\sigma'_1 \ldots \sigma'_n$, and then transforms each (σ'_i,h_i) using $[\![c]\!]_i$. To see the need of the strengthening here, consider the sequential composition of two instructions $x := y.\mathtt{f}; z := x.\mathtt{g}$, where $\mathtt{f}$ and $\mathtt{g}$ belong to different components. Suppose that the first instruction was run on each component using $[\![x := y.\mathtt{f}]\!]_i$. After this execution, most components lose track of the value of x, but the component associated with $\mathtt{f}$ still keeps the updated value of x. However, if all the components are immediately updated again by $[\![z := x.\mathtt{g}]\!]_i$ of the following instruction, we cannot avoid the generation of error because the component associated with $\mathtt{g}$ is different from the one for $\mathtt{f}$ and it is ignorant about the dereferenced variable x. The information about x is kept in the component for $\mathtt{f}$, and to avoid this error, this information should be transferred from the $\mathtt{f}$ component to the $\mathtt{g}$ component. This transfer is implemented by the strengthening of σ_i to σ'_i in the definition of $[\![c]\!]_{\mathsf{CE}}$, and the strengthening itself is done by the subroutine reveal. The subroutine uses $\mathsf{need}(c)$ to find out which component will need information about which variables. Then, it gathers information about these variables from the combined stack σ, and transfers the information to the stacks of appropriate components.

This component-wise semantics is the basis of our analysis. The ith sub-analysis over-estimates all the reachable stack-heap pairs of the ith component, and regions are maintained across the sub-analyses in order to accurately capture the behaviours of reveal.

7.2 Insertion of the trans instruction

We interpret $\texttt{trans}_{i\to j}(x)$ in the component-wise semantics as follows:

$$[\![\texttt{trans}_{i\to j}(x)]\!]_{\mathsf{CE}}((\sigma_1,h_1),\ldots,(\sigma_n,h_n)) = \\ \{((\sigma_1,h_1),\ldots,(\sigma_j \wedge x=\sigma_i(x),h_j),\ldots,(\sigma_n,h_n))\}$$

This definition means that $\texttt{trans}_{i\to j}(x)$ transfers information about the cell x from the ith component to the jth component. While the reveal operation is implicit about the transferred information, $\texttt{trans}_{i\to j}(x)$ is explicit about the originator i and the recipient j of the information as well as the variable x in concern.

The information transferred by the reveal operation is useful when it is new to the recipient. Our pre-analysis captures all such cases in a given program, and inserts appropriate trans instructions. For each component in our component-wise semantics, the pre-analysis under-approximates the set of program variables that are tracked by the component at each program point. Then, it concludes that the ith component may need information about a variable x at a program point v, if x is not one of those definitely tracked variables by the component but it is dereferenced at the point v.

The under-estimation of tracked variables is done by the data-flow analysis for *Avail* in Fig. 6. The domain of this analysis is $\mathcal{D}_{pre} = \mathcal{P}(\mathsf{Comps}\times\mathsf{Exp})$ where Comps is the set of component ids $\{1,\ldots,n\}$ and Exp is the set of expressions in the given program. The data-flow analysis computes a map *Avail* from program points to $\mathcal{D}_{pre}$ by repeatedly applying the first three equations in Fig. 6 until $Avail_{k+1} = Avail_k$ for some k. If $(i,e) \in Avail(v)$ at a program point v, the expression e must be `null` or its value can be obtained just using the stack of the ith component. Our data-flow equation differs from usual ones in having $\mathsf{need}(c)$ in the equation, which reflects the effect of reveal in the semantics.

By using the analysis result *Avail*, we insert `trans` instructions as follows: for each edge (v_0,v_1) with label c, and for all $(j,x) \in \mathsf{need}(c)$, if $(j,x) \notin Avail(v_0)$ but $(i,x) \in Avail(v_0)$ for some i, an edge with label $\texttt{trans}_{i\to j}(x)$ is inserted right before the edge (v_0,v_1).

7.3 Inter-procedural setting

A straightforward extension to inter-procedural analysis could miss the insertion of trans in a desired place. Recall the function `tree_remove(t)` in Fig. 1, which separates the node `t` from its enclosing tree. Consider a call `tree_remove(x)` where the tree component has no information about the node `x`. Then our transformation algorithm in the previous section would insert `trans` for `t` inside the procedure body, since the tree fields of t are accessed there but the node `t` is unknown to the tree component. This insertion is not ideal. To see this, consider another call `tree_remove(y)` where the tree component has the information about the node `y` this time. In this case, the inserted `trans` inside the body of `tree_remove` will make the sub-analyses communicate, even when this communication is not necessary. Inserting trans right before the particular call `tree_remove(x)` is better than doing it inside the body of `tree_remove`.

Fig. 6 The pre-analyses for discovering `trans` instructions. $d_{init} \in D$ is the initial abstract state given as the input to the whole analysis.

$$
\begin{aligned}
Avail_0(v) &= \mathsf{Comps} \times \mathsf{Exp} \text{ (for all } v \in V) \\
Avail_{k+1}(\mathsf{entry}) &= \{(i,e) \mid e \text{ is } \mathtt{null} \text{ or appears in all } H \text{ in } (d_{init})_i\} \\
Avail_{k+1}(v) &= \textstyle\bigcap_{(v',v)\in E} (\!|L(v',v)|\!)^\sharp_{\mathsf{A}}(Avail_k(v')) \text{ (for } v \in V \setminus \{\mathsf{entry}\}) \\
&\quad \text{where } (\!|c|\!)^\sharp_{\mathsf{A}}(X) = (X \cup \mathsf{need}(c)) \setminus \mathsf{kill}(c) \cup \mathsf{gen}_{\mathsf{A}}(c, X) \\
Live_0(v) &= \emptyset \text{ (for all } v \in V) \\
Live_{k+1}(\mathsf{exit}) &= \mathsf{Comps} \times \{\mathtt{null}\} \\
Live_{k+1}(v) &= \textstyle\bigcup_{(v,v')\in E} (\!|L(v,v')|\!)^\sharp_{\mathsf{L}}(Live_k(v')) \text{ (for } v \in V \setminus \{\mathsf{exit}\}) \\
&\quad \text{where } (\!|c|\!)^\sharp_{\mathsf{L}}(X) = X \setminus \mathsf{kill}(c) \cup \mathsf{gen}_{\mathsf{L}}(c, X)
\end{aligned}
$$

instr c	$\mathsf{need}(c)$	$\mathsf{kill}(c)$	$\mathsf{gen}_{\mathsf{A}}(c,X)$	$\mathsf{gen}_{\mathsf{L}}(c,X)$
$\mathtt{assume}(b)$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
$x := e$	$\emptyset$	$\mathsf{Comps} \times \{x\}$	$\{(i,x) \mid (i,e) \in X\}$	$\{(i,e) \mid (i,x) \in X\}$
$x := e.\mathtt{f}$	$\{(i,e) \mid \mathtt{f} \in F_i\}$	$\mathsf{Comps} \times \{x\}$	$\{(i,x) \mid \mathtt{f} \in F_i\}$	$\{(i,e) \mid \mathtt{f} \in F_i\}$
$e.\mathtt{f} := e'$	$\{(i,e),(i,e') \mid \mathtt{f} \in F_i\}$	$\emptyset$	$\emptyset$	$\{(i,e),(i,e') \mid \mathtt{f} \in F_i\}$
$\mathtt{free}(e)$	$\mathsf{Comps} \times \{e\}$	$\mathsf{Comps} \times \{e\}$	$\emptyset$	$\mathsf{Comps} \times \{e\}$
$x := \mathtt{new}_F()$	$\emptyset$	$\mathsf{Comps} \times \{x\}$	$\mathsf{Comps} \times \{x\}$	$\emptyset$

To enable the insertion of trans at call sites, we infer which fields of parameters are accessed in the body of each procedure using a data-flow analysis. This data-flow analysis computes a map *Live* from program points to $\mathcal{D}_{pre}$ as shown in Fig. 6. Our intention is that if $(i,e) \in Live(v)$, the value of e should be obtainable from the stack of the ith component, so that the ith component does not need the help from the reveal operation. The equation in the figure is similar to that of the standard liveness analysis except in the case $x := e$ of $\mathsf{gen}_{\mathsf{L}}$. For $x := e$, the liveness of x is propagated to e.

For interprocedural analysis of *Avail*, we use the analysis result *Live*. For the entry point v of procedure $p(t)$, $Avail_{k+1}(v) = Live(v)$ for all $k \geq 0$, and $\mathsf{need}(x := p(e)) = Live(v)[e/t]$ for all procedure calls $p(e)$.

8 Region inference

The next step is to find `move` and `moveRgn` instructions which give us a *fine-grained* region scheme. Note that there is no correct region scheme; it is not wrong to assign two separate data structures to one region. However, a coarse region scheme will sometimes fail to verify program correctness. We found it desirable to assign as many different regions to separate overlaid data structures as possible.

8.1 Introducing regions

We use `new` and `move` instructions to introduce new regions at a cell's allocation and at the separation of a cell from its data structure, respectively. When a cell is newly allocated, it is natural to introduce a new region for the cell. So, we transform $x := \mathtt{new}_F()$ to $x := \mathtt{new}_{\alpha,F}()$ with fresh α. Also, we introduce a new region when a cell is detached from *all* the components of an overlaid data structure because, as shown in Section 3, this separation is possibly the first step of moving the cell across

Fig. 7 The pre-analysis for discovering `move` instructions.

$$
\begin{aligned}
\mathit{Alloc}_0(v) &= \mathsf{Comps} \times (\mathsf{Exp} \setminus \{\mathtt{null}\}) \text{ (for all } v \in V) \\
\mathit{Alloc}_{k+1}(\mathsf{entry}) &= \emptyset \\
\mathit{Alloc}_{k+1}(v) &= \bigcap_{(v',v)\in E} (\!|L(v',v)|\!)^\sharp_{\mathsf{O}}(\mathit{Alloc}_k(v')) \text{ (for } v \in V \setminus \{\mathsf{entry}\}) \\
&\quad \text{where } (\!|c|\!)^\sharp_{\mathsf{O}}(X) = (X \cup \mathsf{nonnull}(c)) \setminus \mathsf{kill}(c) \cup \mathsf{gen}_{\mathsf{O}}(c, X)
\end{aligned}
$$

instr c	$\mathsf{nonnull}(c)$	$\mathsf{kill}(c)$	$\mathsf{gen}_{\mathsf{O}}(c, X)$
$\mathtt{assume}(b)$	$\emptyset$	$\emptyset$	$\emptyset$
$x := e$	$\emptyset$	$\mathsf{Comps} \times \{x\}$	$\{(i, x) \mid (i, e) \in X\}$
$x := e.\mathtt{f}$	$\{(i, e) \mid \mathtt{f} \in F_i\}$	$\mathsf{Comps} \times \{x\}$	$\emptyset$
$e.\mathtt{f} := e'$	$\{(i, e) \mid \mathtt{f} \in F_i\}$	$\emptyset$	$\emptyset$
$\mathtt{free}(e)$	$\mathsf{Comps} \times \{e\}$	$\mathsf{Comps} \times \{e\}$	$\emptyset$
$x := \mathtt{new}_F()$	$\emptyset$	$\mathsf{Comps} \times \{x\}$	$\mathsf{Comps} \times \{x\}$

overlaid data structures. Note that we are not interested in a cell's separation from only *some* components. Such a separation cannot be the start of a move across overlaid data structures; at most within one overlaid data structure. This might be valuable to be captured for verification but it was not in our experiment.

By tracking of a cell's allocatedness in the semantics, we can capture the cell's separation from an overlaid data structure. For cutting the pointers of all the components in the cell x, the cell should be looked up for all the components; that is, the value of x should be available as a location for all the stack components in the component-wise semantics.

Such program points can be statically estimated by a data-flow analysis similar to Fig. 6. *Avail* shows which variables must be defined, but here it is necessary to figure out which variables must have locations. We need a map *Alloc* from program points to $\mathcal{D}_{pre}$ such that if $(i, e) \in \mathit{Alloc}(v)$, e should have a location in the ith stack component at program point v. The map can be computed by the data-flow analysis in Fig. 7. The equation is similar to that of *Avail* in Fig. 6 but they are different in the cases of $x := e.\mathtt{f}$ and $e.\mathtt{f} := e'$. For *Avail*, it is guaranteed that x and e' are defined in `f`'s component by reveal, but for *Alloc*, it is not guaranteed that they are not `null`.

By using the analysis result *Alloc*, we insert `move` instructions as follows: for all $(v', v) \in E$ and for all variables x, if there exists i such that $(i, x) \notin \mathit{Alloc}(v')$ and $\{(j, x) \mid 1 \le j \le n\} \subseteq (\!|L(v', v)|\!)^\sharp_{\mathsf{O}}(\mathit{Alloc}(v'))$, $\mathtt{move}(x, \alpha)$ is inserted with fresh α right after this edge (v', v).

8.2 Merging regions

We use `moveRgn` instructions to combine two regions when the cells in two regions are connected as one overlaid data structure. This behaviour cannot be easily captured in the semantics but can be detected by our main shape analysis. The function abs used in our main shape analysis detects several assertions connected as one data structure and converts them to one assertion. This is exactly what we want capture. We assume that for all $1 \le i \le n$, the function abs_i of each ith component is given. Additionally, we assume that abs_i combines only assertions in the same region, so that abs_i cannot abstract assertions in different regions. The abstraction function abs on our abstract domain $\mathcal{D}$ is defined by component-wise

Fig. 8 Abstract transfer functions. The abstract value d below is not $\top$. When $\top$ is the input, $[\![c]\!]^\sharp\top = \top$. We assume that $[\![c]\!]^\sharp_i$'s are given for other instructions c.

$[\![c]\!]^\sharp(d) =$ **if** $(\exists i.\, [\![c]\!]^\sharp_i(d_i) = \top)$ **then** $\top$ **else** $([\![c]\!]^\sharp_1(d_1), \ldots, [\![c]\!]^\sharp_n(d_n))$
$[\![\mathtt{trans}_{i\to j}(e)]\!]^\sharp(d) = \mathsf{trans}_{i\to j}(e)(d)$
$[\![\mathtt{moveRgn}(\alpha,\beta)]\!]^\sharp(d) = d[\beta/\alpha]$
$[\![\mathtt{move}(e,\alpha)]\!]^\sharp(d) =$
  let $\mathsf{check}(i, H) =$
    (1) Find finitely many H_k's in SH_i such that
      $H \vdash \bigvee_{k\in K} H_k$ and H_k has the form $\exists \mathbf{x_k}.\Pi_k \wedge e{\mapsto}\{\mathbf{f_k}{:}\mathbf{e'_k}\}_{\beta_k} * \Sigma_k$.
    (2) If cannot find, return $\{\top\}$. Otherwise, for the found H_k's, rename the
      subscript β_k of $e{\mapsto}\{...\}_{\beta_k}$ by α and return $\{\exists \mathbf{x_k}.\Pi_k \wedge e{\mapsto}\{\mathbf{f_k}{:}\mathbf{e'_k}\}_\alpha * \Sigma_k\}_{k\in K}$.
    and $r_i = \bigcup\{\mathsf{check}(i, H) \mid H \in d_i\}$ for all i
  in if $(\exists i$ s.t. $\top \in r_i)$ **then** $\top$ **else** $(r_1, \ldots, r_n)$

applications of abs_i's:

$$\mathsf{abs}(\top) = \top \qquad \mathsf{abs}(d) = (\mathsf{abs}_1(d_1), \ldots, \mathsf{abs}_n(d_n)).$$

When merging two regions enables further abstraction, we insert `moveRgn` instructions. Let d be an abstract state and $d[\beta/\alpha]$ be the same one but such that every region variable α is replaced by β. Suppose that $\mathsf{abs}(d) \neq \mathsf{abs}(d[\beta/\alpha])$. It means that some cells in two regions α and β are connected as one data structure in at least one component of d. In this case, we insert $\mathtt{moveRgn}(\alpha, \beta)$. Here we just define which `moveRgn` instructions are discovered for an abstract state d:

$$\mathsf{enableAbs}(d) = \{\mathtt{moveRgn}(\alpha,\beta) \mid \mathsf{abs}(d) \neq \mathsf{abs}(d[\beta/\alpha])\}\,.$$

This function will be used in our main shape analysis.

9 Main analysis: 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 $G = (V, E, \mathsf{entry}, \mathsf{exit}, L)$ that can include ghost instructions $\mathtt{move}(e, \alpha)$ and reduction operators $\mathsf{trans}_{i\to j}(e)$ discovered by our pre-analysis (but not $\mathtt{moveRgn}(\alpha, \beta)$). Given this input, the engine computes two maps I and A from program points, the first I to abstract states and the second A to sets of ghost instructions of the form `moveRgn`:

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

Here $\mathsf{Regions}(d_{init}, L)$ is the set of region variables appearing in d_{init} or some instruction in the range of L. Note that since $\mathsf{Regions}(d_{init}, L)$ 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 that for all normal or ghost instructions or our weak reduction operator c, we are given the transfer function $[\![c]\!]^\sharp : \mathcal{D} \to \mathcal{D}$. Now, for (finite) subsets M_0 of M, define $[\![M_0]\!]^\sharp(d) = ([\![c_n]\!]^\sharp \circ \ldots \circ [\![c_1]\!]^\sharp)(d)$, where $c_1, \ldots, c_n$ 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:

$$\begin{array}{ll} I_n(\mathsf{entry}) = d_{init}, & I_{n+1}(v) = \bigsqcup_{(v',v)\in E}(\mathsf{abs} \circ [\![A_n(v)]\!]^\sharp \circ [\![L(v',v)]\!]^\sharp)(I_n(v')), \\ A_n(\mathsf{entry}) = \emptyset, & A_{n+1}(v) = A_n(v) \cup \mathsf{enableAbs}(I_{n+1}(v)). \end{array}$$

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 abstract transfer function $[\![c]\!]^\sharp$. For normal instructions c, we define $[\![c]\!]^\sharp$ by applying component-wise the sub-analyses' transfer functions $[\![c]\!]^\sharp_i : \mathcal{D}_i \cup \{\top\} \to \mathcal{D}_i \cup \{\top\}$. The details are given in Fig. 8. The figure also shows that $[\![\mathtt{trans}_{i\to j}(e)]\!]^\sharp$ is implemented by the reduction operator with the same name $\mathsf{trans}_{i\to j}(e)$ defined in Section 6, $[\![\mathtt{moveRgn}(\alpha,\beta)]\!]^\sharp$ by the substitution of the source region variable α by the target β, and $[\![\mathtt{move}(e,\alpha)]\!]^\sharp$ by the exposure of a points-to fact $e \mapsto \{\ldots\}_{\beta_k}$ from a symbolic heap followed by the renaming of its subscript β_k by α.

10 Experiments

We have implemented an interprocedural version of the analysis (based on the RHS algorithm [14]), and applied it to verify the memory safety of two types 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 Fig. 9.

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 $\mathcal{P}(\mathsf{SH}_1 \times \ldots \times \mathsf{SH}_n) \cup \{\top\}$), 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 $\mathtt{trans}_{i\to j}(e)$ 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.

- `list-dio` is an abstract version of the deadline IO scheduler. It uses two doubly-linked lists instead of a list and a 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.

Fig. 9 Experimental result obtained using Intel Core i7 2.66GHz with 8GB memory.

filename	# of lines[a]	analysis time (sec) (A) old	(B) new	speedup (A/B)	# of `trans` inserted
`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

[a] Only relevant lines of the preprocessed source files are counted.

- `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 a high analysis cost.
- `fs/afs/server.c` has an overlaid data structure of three components: two doubly-linked lists and one red-black tree. The components of a list and a tree are for maintaining a list of servers, and the other doubly-linked list is for removing servers: Servers to be removed are additionally connected to the `graveyard` list.

11 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 into its components, and to track components using sub-analyses as independently as possible, while allowing communication among them using ghost instructions. Besides the progress in verifying more chal-

lenging data structures, we hope that our work has provided further evidence that with a 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. Arnold, G., Manevich, R., Sagiv, M., Shaham, R.: Combining shape analyses by intersecting abstractions. In: Proc. of the International Conference on Verification, Model Checking, and Abstract Interpretation, pp. 33–48 (2006)
2. Ball, T., Levin, V., Rajamani, S.K.: A decade of software model checking with SLAM. Communications of the ACM **54**(7), 68–76 (2011)
3. Ball, T., Podelski, A., Rajamani, S.: Boolean and cartesian abstraction for model checking C programs. In: In Proc. of the Tools and Algorithms for the Construction and Analysis of Systems, pp. 268–283 (2001)
4. Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: A static analyzer for large safety-critical software. In: Proc. of the ACM Conference on Programming Language Design and Implementation, pp. 196–207 (2003)
5. Calcagno, C., Distefano, D., O'Hearn, P., Yang, H.: Compositional shape analysis by means of bi-abduction. In: Proc. of the ACM Symposium on Principles of Programming Languages, pp. 289–300 (2009)
6. Cherini, R., Rearte, L., Blanco, J.: A shape analysis for non-linear data structures. In: Proc. of the International Static Analysis Symposium, pp. 201–217 (2010)
7. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: Proc. of the ACM Symposium on Principles of Programming Languages, pp. 269–282 (1979)
8. Distefano, D., O'Hearn, P., Yang, H.: A local shape analysis based on separation logic. In: In Proc. of the Tools and Algorithms for the Construction and Analysis of Systems, pp. 287–302 (2006)
9. Hawkins, P., Aiken, A., Fisher, K.: Reasoning about shared mutable data structures (2010). Manuscript
10. Hawkins, P., Aiken, A., Fisher, K., Rinard, M., Sagiv, M.: Data structure fusion. In: Proc. of the Asian Symposium on Programming Languages and Systems, pp. 204–221 (2010)
11. Kreiker, J., Seidl, H., Vojdani, V.: Shape analysis of low-level C with overlapping structures. In: Proc. of the International Conference on Verification, Model Checking, and Abstract Interpretation, pp. 214–230 (2010)
12. Kuncak, V., Lam, P., Zee, K., Rinard, M.: Modular pluggable analyses for data structure consistency. IEEE Transactions on Software Engineering **32**(12), 988–1005 (2006)
13. Lee, O., Yang, H., Petersen, R.: Program analysis for overlaid data structures. In: Proc. of the International Conference on Computer Aided Verification, pp. 592–608 (2011)
14. Reps, T., Horwitz, S., Sagiv, S.: Precise interprocedural dataflow analysis via graph reachability. In: Proc. of the ACM Symposium on Principles of Programming Languages, pp. 49–61 (1995)
15. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: Proc. of the IEEE Symposium on Logic in Computer Science, pp. 55–74 (2002)
16. Sagiv, M., Reps, T., Wilhelm, R.: Parametric shape analysis via 3-valued logic. ACM Transactions on Programming Languages and Systems **24**(3), 217–298 (2002)
17. Yang, H., Lee, O., Berdine, J., Calcagno, C., Cook, B., Distefano, D., O'Hearn, P.: Scalable shape analysis for systems code. In: Proc. of the International Conference on Computer Aided Verification, pp. 285–398 (2008)