Interprocedural Shape Analysis Using Separation Logic-based Transformer Summaries

Hugo Illous, Matthieu Lemerre, Xavier Rival

Interprocedural Shape Analysis Using Separation Logic-based Transformer Summaries

Hugo Illous[1,2], Matthieu Lemerre[1], and Xavier Rival[2]

[1] Université Paris-Saclay, CEA, List, F-91120, Palaiseau, France
[2] INRIA Paris/CNRS/École Normale Supérieure/PSL Research University

Abstract. Shape analyses aim at inferring semantic invariants related to the data-structures that programs manipulate. To achieve that, they typically abstract the set of reachable states. By contrast, abstractions for transformation relations between input states and output states not only provide a finer description of program executions but also enable the composition of the effect of program fragments so as to make the analysis modular. However, few logics can efficiently capture such transformation relations. In this paper, we propose to use connectors inspired by separation logic to describe memory state transformations and to represent procedure summaries. Based on this abstraction, we design a top-down interprocedural analysis using shape transformation relations as procedure summaries. Finally, we report on implementation and evaluation.

1 Introduction

Static analyses based on abstractions of sets of states (or for short, *state analyses*) compute an over-approximation of the states that a program may reach, so as to answer questions related, e.g., to safety (absence of errors or structural invariant violations). By contrast, one may also design static analyses that discover relations between program initial states and output states. In this paper, we refer to such static analyses as *transformation analyses*. A transformation relation between the initial state and the output state of a given execution can provide an answer to questions related to the functional correctness of a program (i.e., does it compute a correct result when it does not crash and terminates). Another application of such a transformation relation is to let the analysis reuse multiple times the result obtained for a given code fragment (e.g., a procedure), provided the analysis can compose transformation relations. The great advantage of this approach is to reduce the analysis cost, by avoiding to recalculate the effect, e.g., of a procedure in multiple calling contexts. This is known as the relational approach to interprocedural analysis [35].

However, a major difficulty is to find an accurate and lightweight representation of the input-output transformation relation of each procedure. A first solution is to resort to tables of abstract pre- and post-conditions that are all expressed in a given state abstract domain [10,2,15,22]. However, this generally makes composition imprecise unless very large tables can be computed. A second solution is to build upon a *relational abstract domain*, namely, an abstract domain that can capture

relations across distinct program variables. The transformation between states is then expressed using "primed" and "non-primed" variables where the former describe the input state and the latter the output state [24,26,27]. As an example, we consider a procedure that computes the absolute value of input x and stores it into y (for the sake of simplicity we assume mathematical integers):

- Using the intervals abstract domain [9], we can provide the table $[((x \in [-5,-1], y \in]-\infty,+\infty[) \mapsto (x \in [-5,-1], y \in [1,5])); ((x \in [-1,10], y \in]-\infty,+\infty[) \mapsto (x \in [-1,10], y \in [0,10]))]$ (this table is obviously not unique);
- Using the relational abstract domain of polyhedra [11], we can construct the transformation relation $(x' = x \wedge y' \geq x \wedge y' \geq -x)$.

We note that, while the expressiveness of the two is not comparable, the latter option is more adapted to compositional reasoning. For instance, given pre-condition $-10 \leq x \leq -5$, the analysis based on a table either returns a very imprecise answer or requires enriching the table whereas the analysis with a relational domain can immediately derive $x' = x \in [-10,-5]$ (x has not changed) and $y' \geq 5$.

Such reasoning becomes more complex when considering non-numerical facts, such as memory shape properties. Many works rely on the tabulation approach, using a conventional shape state abstraction [2,15,22]. In general, the tabulation approach restricts the ability to precisely relate procedure input and output states and may require large tables of pairs of formulas for a good level of precision. The approach based on a relational domain with primed and non-primed variables has been implemented by [19,18] in the TVLA shape analysis framework [33]. However, it is more difficult to extend shape analyses that are based on separation logic [28] since a separation logic formula describes a region of a given heap; thus, it does not naturally constrain fragments of two different states. To solve this issue, a first approach is to modify the semantics of separation logic connectors to pairs of states [34]. A more radical solution is to construct novel logical connectors over state transformation relations that are inspired by separation logic [17]. These transformations can describe the effect of a program and express facts such as "memory region A is left untouched whereas a single element is inserted into the list stored inside memory region B and the rest of that list is not modified". The analysis of [17] is designed as a forward abstract interpretation which produces abstract transformation relations. Therefore, it can describe tranfsormations precisely using separation logic predicates and without accumulating tables of input and output abstract states.

However, this analysis still lacks several important components to actually make interprocedural analysis modular. In particular, it lacks a *composition* algorithm over abstract transformation relations. Modular interprocedural analysis also needs to synchronize two distinct processes that respectively aim at *computing procedure summaries* and at *instantiating* them at a call-site. In this paper, we propose a top-down analysis based on shape summaries and make the following contributions:

- in Section 2, we demonstrate the use of abstract shape transformations;
- in Section 3 and Section 4, we formalize transformation summaries based on separation logic (intraprocedural analysis is presented in Section 5);
- in Section 6, we build a composition algorithm over transformation summaries;

```
1 typedef struct list { struct list * n; /* ... */ } list;
2 void append( list *l0, list *l1 ){
3   assume(l0 != NULL); list *c = l0;
4   while( c->n != NULL ){ c = c->n; }
5   c->n = l1;
6 }
7 void double_append( list *k0, list *k1; list *k2 ){
8   assume(k0 != NULL); append( k0, k1 ); append( k0, k2 );
9 }
```

Fig. 1. Simple and double list append procedures.

- in Section 7, we formalize a modular interprocedural analysis;
- in Section 8, we report on implementation and evaluation.

2 Overview

In this section, we study a restricted example to highlight some issues in interprocedural analysis. We consider a recursive implementation of C linked lists, with a couple of procedures shown in Figure 1. The function `append` takes two lists as arguments and assumes the first one non-empty, it traverses the first list, and mutates the pointer field of the last element. The function `double_append` takes three lists as arguments (the first one is assumed non-empty) and concatenates all three by calling `append`. The topic of our discussion is only the invariants underlying this code and their discovery, not the efficiency of the code itself.

State abstraction and analysis. We consider an abstraction based on separation logic [28], as shown, e.g., in [13,6]. To describe sets of states, we assume a predicate $\mathbf{lseg}(\alpha_0, \alpha_1)$ that represents heap regions that consist of a list segment starting at some address represented by the symbolic variable α_0 and such that the last element points to some address represented by α_1. Such a segment may be empty. For short, we note $\mathbf{list}(\alpha)$ for the instance $\mathbf{lseg}(\alpha, \mathbf{0x0})$, which denotes a complete singly-linked list (as the last element contains a null next link). A single list element writes down $\alpha_0 \cdot \underline{\mathtt{n}} \mapsto \alpha_1$ where $\underline{\mathtt{n}}$ denotes the next field offset (we elide other fields). More generally, $\alpha_0 \mapsto \alpha_1$ denotes a single memory cell of address α_0 and content α_1. Thus $\&\mathtt{x} \mapsto \alpha$ expresses that variable `x` stores a value α (which may be either a base value or the address of a memory cell). Predicates **lseg** and **list** are *summary* predicates as they describe unbounded memory regions; their denotation is naturally defined by induction. As usual, separating conjunction $*$ conjoins predicates over *disjoint* heap regions.

Assuming the abstract precondition $\&\mathtt{l0} \mapsto \alpha_0 * \mathbf{lseg}(\alpha_0, \alpha_1) * \alpha_1 \cdot \underline{\mathtt{n}} \mapsto \mathbf{0x0} * \&\mathtt{l1} \mapsto \alpha_2 * \mathbf{list}(\alpha_2)$, existing state shape analyses [13,6] can derive the post-condition $\&\mathtt{l0} \mapsto \alpha_0 * \mathbf{lseg}(\alpha_0, \alpha_1) * \alpha_1 \cdot \underline{\mathtt{n}} \mapsto \alpha_2 * \&\mathtt{l1} \mapsto \alpha_2 * \mathbf{list}(\alpha_2)$ by a standard forward abstract interpretation [9] of the body of `append`. The analysis proceeds by abstract interpretation of basic statements, unfolds summaries when

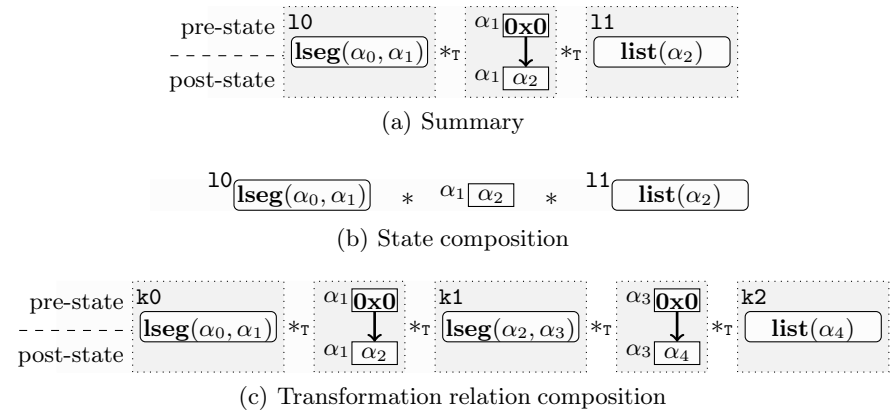

Fig. 2. A shape transformation procedure summary and composition.

analyzing reads or writes into the regions that they represent, and folds basic predicates into summaries at loop heads. The convergence of the loop analysis takes a few iterations and involves widening which is often a rather costly operation.

The analysis of `double_append` by inlining follows similar steps. One important note though is that it analyses the body of `append` twice, namely once per calling context, and thus also the loop inside `append`. In turn, if `double_append` is called in several contexts in a larger program, or inside a loop, its body will also be analyzed multiple times, which increases even more the overall analysis cost.

Transformation analysis. Unlike state analyses, transformation analyses compute abstractions of the input-output relation of a program fragment. As an example, given the above abstract pre-condition, the analysis of [17] infers relations as shown in Figure 2(a). This graphical view states that the procedure `append` keeps both summary predicates unchanged and only modifies the middle list element so as to append the two lists. This transformation is formally represented using three basic kinds of relational predicates that respectively state that one region is preserved, that one region is "transformed" into another one, and that two transformations are conjoined at the transformation level. To stress the difference, we write $*_{\mathtt{T}}$ for the latter, instead of just $*$. Although it uses transformation predicates, the analysis follows similar abstract interpretation steps as [13,6].

Towards modular analysis: composition of transformation abstractions. The first advantage of transformation predicates such as Figure 2(a) is that they can be *applied* to state predicates in the abstract, as a function would be in the concrete. Indeed, if we apply this abstract transformation to the abstract pre-condition given above, we can match each component of the abstract transformation of Figure 2(a), and derive its post-condition. In this example, the body of each summary is left unchanged, whereas the last element of the first list is updated. The result is shown in Figure 2(b) and it can be derived without reanalyzing the body of `append`.

While these steps produce a state analysis of the body of `double_append`, we may want to summarize the effect of this function too, as it may be called somewhere else in a larger program. To achieve this, we need not only to apply an abstract transformation to an abstract state, but to *compose* two abstract transformations together. Intuitively, this composition should proceed in a similar way as the application that we sketched above. In the case of `double_append`, the analysis requires case splits depending on whether `k1` is empty or not. For brevity, we show only the case where this list is non-empty in Figure 2(c). In general, the composition of two transformations requires to match the post-condition of the first and the pre-condition of the second, and to refine them, using some kind of intersection as shown in Section 6. Another important issue is the summary computation process. Bottom-up analyses strive for general summaries whatever the calling context. However, a procedure may behave differently when applied to other structures (e.g., a binary tree or a doubly linked list), thus the top-down approach which provides information about the calling contexts before they are analyzed seems more natural. However, this means that the analysis should infer summaries and apply them simultaneously, and that the discovery of new calling contexts may require for more general summaries. We describe this in Section 7.

3 Abstraction of sets of states and state transformations

In the following sections, we restrict ourselves to a small imperative language that is sufficient to study procedure transformation summaries, although our implementation described in Section 8 supports a larger fragment of C. We also only consider basic singly linked lists in the formalization although the implementation supports a large range of list or tree-like inductive data-structures.

Concrete states, programs, and semantics. We write $\mathbb{X}$ for the set of program variables and $\mathbb{V}$ for the set of values, which include memory addresses. The address of a variable $\mathtt{x}$ is assumed fixed and noted $\&\mathtt{x}$. Structure fields are viewed both as names and as offsets; they include the null offset (basic pointer) and $\underline{\mathtt{n}}$, the "next" field of a list element. We let a memory state $\sigma \in \mathbb{M}$ be a partial function from variable addresses and heap addresses to values. We write $\mathbf{dom}(\sigma)$ for the domain of σ, that is the set of elements for which it is defined. Additionally, if σ_0, σ_1 are such that $\mathbf{dom}(\sigma_0) \cap \mathbf{dom}(\sigma_1) = \emptyset$, we let $\sigma_0 \circledast \sigma_1$ be the memory state obtained by appending σ_1 to σ_0 (its domain is $\mathbf{dom}(\sigma_0) \cup \mathbf{dom}(\sigma_1)$). If a is an address and v a value, we write $[a \mapsto v]$ for the memory state σ such that $\mathbf{dom}(\sigma) = \{a\}$ and $\sigma(a) = v$. A command $\mathtt{C}$ is either an assignment, a local variable declaration or a loop (we omit tests and memory allocation out as our procedure summary analysis handles them in a very standard way). A procedure $\mathtt{P}$ is defined by a list of arguments and a body (we let function returns be encoded via parameter passing by reference). A program is made of a series of procedures including a `main`. All variables are local. The syntax of programs is defined by the grammar below:

$$\mathtt{C} ::= \mathtt{x} = \mathtt{y} \mid \mathtt{x} = v \mid \mathtt{x}\texttt{->}\underline{\mathtt{n}} = \mathtt{y} \mid \mathtt{x} = \mathtt{y}\texttt{->}\underline{\mathtt{n}} \mid \mathtt{C};\mathtt{C} \mid \mathbf{while}(\mathtt{x} \neq \mathbf{0x0})\{\mathtt{C}\} \mid \mathbf{decl}\,\mathtt{x}; \mid \mathtt{f}(\mathtt{x}_0, \ldots, \mathtt{x}_k)$$

$$\mathtt{P} ::= \mathbf{proc}\,\mathtt{f}(\mathtt{p}_0, \ldots, \mathtt{p}_k)\{\mathtt{C}\} \qquad \mathtt{R} ::= \mathtt{P}_0 \ \ldots \ \mathtt{P}_l \ \ \mathbf{proc}\,\mathtt{main}()\{\mathtt{C}\}$$

$$\mathrm{n}(\in \mathbb{N}) ::= \alpha \quad (\alpha \in \mathbb{A}) \mid \&\mathtt{x} \quad (\mathtt{x} \in \mathbb{X}) \qquad \mathrm{c}^\sharp ::= \mathrm{n} \odot \mathbf{0x0} \quad (\odot \in \{=, \neq\}) \mid \mathrm{n} = \mathrm{n}'$$
$$\mathrm{h}^\sharp(\in \mathbb{H}) ::= \mathbf{emp} \mid \mathrm{n} \cdot \underline{\mathtt{f}} \mapsto \mathrm{n}' \mid \mathbf{lseg}(\alpha, \alpha') \mid \mathbf{list}(\alpha) \mid \mathrm{h}^\sharp *_{\mathtt{s}} \mathrm{h}^\sharp \mid \mathrm{h}^\sharp \wedge \mathrm{c}^\sharp$$

(a) Abstract states syntax

$$\gamma_{\mathbb{H}}(\mathrm{n} \cdot \underline{\mathtt{f}} \mapsto \mathrm{n}') = \{([\nu(\mathrm{n}) + \underline{\mathtt{f}} \mapsto \nu(\mathrm{n}')], \nu)\} \qquad \gamma_{\mathbb{H}}(\mathbf{emp}) = \{([], \nu)\}$$
$$\gamma_{\mathbb{H}}(\mathrm{h}_0^\sharp *_{\mathtt{s}} \mathrm{h}_1^\sharp) = \{(\sigma_0 \circledast \sigma_1, \nu) \mid (\sigma_0, \nu) \in \gamma_{\mathbb{H}}(\mathrm{h}_0^\sharp) \wedge (\sigma_1, \nu) \in \gamma_{\mathbb{H}}(\mathrm{h}_1^\sharp)\}$$
$$\gamma_{\mathbb{H}}(\mathrm{h}^\sharp \wedge \mathrm{c}^\sharp) = \{(\sigma, \nu) \mid (\sigma, \nu) \in \gamma_{\mathbb{H}}(\mathrm{h}^\sharp) \wedge \nu \in \gamma_{\mathbb{C}}(\mathrm{c}^\sharp)\}$$

(b) Concretization of abstract states

Fig. 3. Syntax and concretization of the abstract states.

We let the semantics of a command $\mathtt{C}$ be a function $[\![\mathtt{C}]\!]_{\mathcal{T}} : \mathcal{P}(\mathbb{M}) \to \mathcal{P}(\mathbb{M})$ that maps a set of input states into a set of output states. We do not detail the definition of this semantics as it is standard. In the following, we formalize two abstractions, that respectively describe sets of states and relations over states.

Abstract states and transformations. The syntax of abstract heaps $\mathrm{h}^\sharp \in \mathbb{H}$ is shown in Figure 3(a). We let $\mathbb{A} = \{\alpha, \alpha', \ldots\}$ denote a set of *symbolic variables* that abstract heap addresses and values. A *symbolic name* $\mathrm{n} \in \mathbb{N}$ is either a variable address $\&\mathtt{x}$ or a symbolic value α. Numerical constraints $\mathrm{c}^\sharp$ describe predicates over symbolic names. An abstract heap (or state) $\mathrm{h}^\sharp \in \mathbb{H}$ is the (possibly) separating conjunction of region predicates that abstract *separate* regions [28], which consist either of an empty region **emp**, or of a basic memory block (described by a points-to predicate $\mathrm{n} \cdot \underline{\mathtt{f}} \mapsto \mathrm{n}'$), or inductive summaries, and may include some numerical constraints (that do not represent any memory region and only constrain symbolic names). We note $*_{\mathtt{s}}$ for separating conjunction over states. The abstract states defined in Figure 3(a) are of a comparable level of expressiveness as the abstractions used in common shape analysis tools such as [13,6,2,14] to verify properties such as the absence of memory errors or the preservation of structural invariants.

The concretization of abstract states is shown in Figure 3(b). It uses valuations to tie the abstract names and the value they denote. A valuation consists of a function $\nu : \mathbb{N} \to \mathbb{V}$. We assume the concretization $\gamma_{\mathbb{C}}(\mathrm{c}^\sharp)$ of a numeric constraint $\mathrm{c}^\sharp$ returns the set of valuations that meet this constraint. Abstract heaps are concretized into sets of pairs made of a heap and of the valuation that realizes this heap. The concretization of summary predicates **list** and **lseg** is defined recursively, by unfolding. Indeed, we let $\gamma_{\mathbb{H}}(\mathrm{h}_0^\sharp) = \bigcup\{\gamma_{\mathbb{H}}(\mathrm{h}_1^\sharp) \mid \mathrm{h}_0^\sharp \to_{\mathfrak{U}} \mathrm{h}_1^\sharp\}$, where $\to_{\mathfrak{U}}$ is defined by (cases for **list** are similar):

$$\mathbf{lseg}(\alpha_0, \alpha_1) \to_{\mathfrak{U}} \mathbf{emp} \wedge \alpha_0 = \alpha_1$$
$$\mathbf{lseg}(\alpha_0, \alpha_1) \to_{\mathfrak{U}} \alpha_0 \cdot \underline{\mathtt{n}} \mapsto \alpha_2 *_{\mathtt{s}} \mathbf{lseg}(\alpha_2, \alpha_1) \wedge \alpha_0 \neq \alpha_1$$

Example 1 (Abstract state). The abstract state in Figure 2(b) writes down as:

$$\&\mathtt{l0} \mapsto \alpha_0 *_{\mathtt{s}} \&\mathtt{l1} \mapsto \alpha_2 *_{\mathtt{s}} \mathbf{lseg}(\alpha_0, \alpha_1) *_{\mathtt{s}} \alpha_1 \cdot \underline{\mathtt{n}} \mapsto \alpha_2 *_{\mathtt{s}} \mathbf{list}(\alpha_2)$$

Assuming both $\mathbf{lseg}(\alpha_0, \alpha_1)$ and $\mathbf{list}(\alpha_2)$ unfold to structures of length one, it concretizes in the same way as:

$$\&\mathtt{l0} \mapsto \alpha_0 *_{\mathtt{s}} \&\mathtt{l1} \mapsto \alpha_2 *_{\mathtt{s}} \alpha_0 \cdot \underline{\mathtt{n}} \mapsto \alpha_1 *_{\mathtt{s}} \alpha_1 \cdot \underline{\mathtt{n}} \mapsto \alpha_2 *_{\mathtt{s}} \alpha_2 \cdot \underline{\mathtt{n}} \mapsto \alpha_3 \wedge \alpha_3 = 0$$

$$\mathrm{t}^\sharp (\in \mathbb{T}) ::= \mathtt{Id}(\mathrm{h}^\sharp) \mid [\mathrm{h}^\sharp \dashrightarrow \mathrm{h}^\sharp] \mid \mathrm{t}^\sharp *_{\mathtt{T}} \mathrm{t}^\sharp \mid \mathrm{t}^\sharp \wedge \mathrm{c}^\sharp$$

(a) Abstract transformations syntax

$$\begin{aligned}
\gamma_{\mathbb{T}}(\mathtt{Id}(\mathrm{h}^\sharp)) &= \{(\sigma, \sigma, \nu) \mid (\sigma, \nu) \in \gamma_{\mathbb{H}}(\mathrm{h}^\sharp)\} \\
\gamma_{\mathbb{T}}([\mathrm{h}_{\mathrm{i}}^\sharp \dashrightarrow \mathrm{h}_{\mathrm{o}}^\sharp]) &= \{(\sigma_{\mathrm{i}}, \sigma_{\mathrm{o}}, \nu) \mid (\sigma_{\mathrm{i}}, \nu) \in \gamma_{\mathbb{H}}(\mathrm{h}_{\mathrm{i}}^\sharp) \wedge (\sigma_{\mathrm{o}}, \nu) \in \gamma_{\mathbb{H}}(\mathrm{h}_{\mathrm{o}}^\sharp)\} \\
\gamma_{\mathbb{T}}(\mathrm{t}_0^\sharp *_{\mathtt{T}} \mathrm{t}_1^\sharp) &= \{(\sigma_{0,\mathrm{i}} \circledast \sigma_{1,\mathrm{i}}, \sigma_{0,\mathrm{o}} \circledast \sigma_{1,\mathrm{o}}, \nu) \mid \\
&\qquad (\sigma_{0,\mathrm{i}}, \sigma_{0,\mathrm{o}}, \nu) \in \gamma_{\mathbb{T}}(\mathrm{t}_0^\sharp) \wedge \mathbf{dom}(\sigma_{0,\mathrm{i}}) \cap \mathbf{dom}(\sigma_{1,\mathrm{o}}) = \emptyset \\
&\qquad \wedge (\sigma_{1,\mathrm{i}}, \sigma_{1,\mathrm{o}}, \nu) \in \gamma_{\mathbb{T}}(\mathrm{t}_1^\sharp) \wedge \mathbf{dom}(\sigma_{1,\mathrm{i}}) \cap \mathbf{dom}(\sigma_{0,\mathrm{o}}) = \emptyset\} \\
\gamma_{\mathbb{T}}(\mathrm{t}^\sharp \wedge \mathrm{c}^\sharp) &= \{(\sigma_{\mathrm{i}}, \sigma_{\mathrm{o}}, \nu) \in \gamma_{\mathbb{T}}(\mathrm{t}^\sharp) \mid \nu \in \gamma_{\mathbb{C}}(\mathrm{c}^\sharp)\}
\end{aligned}$$

(b) Concretization of abstract transformations

Fig. 4. Syntax and concretization of abstract transformations.

Abstract transformations. Abstract transformations are defined on top of abstract states and rely on specific logical connectors. Their syntax is defined in Figure 4(a). A heap transformation is either the identity $\mathtt{Id}(\mathrm{h}^\sharp)$, which denotes physical equality over pairs of states that are both described by $\mathrm{h}^\sharp$, a state transformation $[\mathrm{h}_{\mathrm{i}}^\sharp \dashrightarrow \mathrm{h}_{\mathrm{o}}^\sharp]$ which captures input/output pairs of states respectively defined by $\mathrm{h}_{\mathrm{i}}^\sharp$ and by $\mathrm{h}_{\mathrm{o}}^\sharp$, or a separating conjunction of transformations $\mathrm{t}_0^\sharp *_{\mathtt{T}} \mathrm{t}_1^\sharp$ (we write $*_{\mathtt{T}}$ to stress the distinction with the state separating conjunction $*_{\mathtt{S}}$). The concretization of transformations is shown in Figure 4(b). It is built upon the previously defined $\gamma_{\mathbb{H}}$ and also utilizes valuations. The most interesting case is that of $*_{\mathtt{T}}$: this connector imposes disjointness not only of the sub-heaps in both the pre- and post-state, but also across them. In this paper, we study only a basic form of the transformation predicate $*_{\mathtt{T}}$ although it may be strengthened with additional constraints [16], e.g., to assert that the footprint has not changed or that only specific fields may have been modified. We leave out such constraints as their goal is orthogonal to the focus of this paper. Finally, the analysis uses finite disjunctions of transformations.

Example 2 (Abstract transformation). The transformation informally described in Figure 2(a) is captured by the abstract transformation below:

$$\begin{aligned}
\mathrm{t}^\sharp = \mathtt{Id}\,(\&\mathtt{l0} \mapsto \alpha_0 *_{\mathtt{S}} \&\mathtt{l1} \mapsto \alpha_2 *_{\mathtt{S}} \mathbf{lseg}(\alpha_0, \alpha_1) *_{\mathtt{S}} \mathbf{list}(\alpha_2)) \\
*_{\mathtt{T}} [(\alpha_1 \cdot \underline{\mathtt{n}} \mapsto \mathbf{0x0}) \dashrightarrow (\alpha_1 \cdot \underline{\mathtt{n}} \mapsto \alpha_2)]
\end{aligned}$$

In the following, we write $\mathrm{h}_0^\sharp \rightarrow_{\mathfrak{U}} \mathrm{h}_1^\sharp$ when $\mathrm{h}_0^\sharp$ may be rewritten into $\mathrm{h}_1^\sharp$ by applying $\rightarrow_{\mathfrak{U}}$ to any of its sub-terms. We use this notation for both heaps and transformations. Last, we let $\rightarrow_{\mathfrak{U}[\alpha]}$ denote unfolding of a $\mathbf{list}(\alpha)$ or $\mathbf{lseg}(\alpha, \ldots)$ predicate.

4 Procedure summarization.

The semantics of a procedure boils down to a relation between its input states and its output states, thus our first attempt at summaries over-approximates the input-output relation of the procedure. To express this, we introduce the following notation. If $f : \mathcal{P}(A) \to \mathcal{P}(B)$ is a function and $R \subseteq A \times B$ is a relation, we note $f \Subset R$ if and only if $\forall X \subseteq A,\ X \times f(X) \subseteq R$.

Definition 1 (Global transformation summary). *A* sound global transformation summary *(or, for short,* global summary*) of procedure* **proc** $\mathtt{f}(\ldots)\{\mathtt{C}\}$ *is an abstract transformation* $\mathrm{t}^\sharp$ *that over-approximates* $[\![\mathtt{C}]\!]_\mathcal{T}$ *in the sense that:*

$$[\![\mathtt{C}]\!]_\mathcal{T} \Subset \{(\sigma_\mathrm{i}, \sigma_\mathrm{o}) \mid \exists \nu,\ (\sigma_\mathrm{i}, \sigma_\mathrm{o}, \nu) \in \gamma_\mathbb{T}(\mathrm{t}^\sharp)\}$$

For example, function `append` (Figure 1) can be described using a global procedure summary (noted $\mathrm{t}^\sharp$ in Example 2). While this notion of summary may account for the effect of a procedure, it is not adequate to describe intermediate analysis results. As an example, a procedure `f` is likely to be called in multiple contexts. In that case, when the analysis reaches a first context, it computes a summary $\mathrm{t}^\sharp$, that accounts for the effect of the procedure in that context, for a given set of procedure input states. When it reaches a second context, it should be able to decide whether $\mathrm{t}^\sharp$ also covers the states that reach the procedure in that second context. Observe that the pre-state of $\mathrm{t}^\sharp$ does not suffice since $\mathrm{t}^\sharp$ may have been computed for some very specific context. Moreover, the left projection of $\mathrm{t}^\sharp$ may not account for some call states encountered so far when these lead to non-termination or to an error in the body of `f`. To overcome this issue, an over-approximation of the procedure input states observed so far should be adjoined to the global summary:

Definition 2 (Context transformation summary). *A* sound context transformation summary *(or, for short,* context summary*) of procedure* **proc** $\mathtt{f}(\ldots)\{\mathtt{C}\}$ *is a pair* $(\mathrm{h}^\sharp_f, \mathrm{t}^\sharp_f)$ *such that the following holds:*

$$(\lambda(M \subseteq \{\sigma \mid \exists \nu, (\sigma, \nu) \in \gamma_\mathbb{H}(\mathrm{h}^\sharp_f)\}) \cdot [\![\mathtt{C}]\!]_\mathcal{T}(M)) \Subset \{(\sigma_\mathrm{i}, \sigma_\mathrm{o}) \mid \exists \nu, (\sigma_\mathrm{i}, \sigma_\mathrm{o}, \nu) \in \gamma_\mathbb{T}(\mathrm{t}^\sharp_f)\}$$

Intuitively, Definition 2 asserts that $(\mathrm{h}^\sharp_\mathtt{f}, \mathrm{t}^\sharp_\mathtt{f})$ captures all the functions such that their restriction to states in $\mathrm{h}^\sharp_\mathtt{f}$ can be over-approximated by relation $\mathrm{t}^\sharp_\mathtt{f}$. Although we do not follow this approach here, the $\mathrm{h}^\sharp_\mathtt{f}$ component may be used in order to augment summaries with context sensitivity. We note that $\mathrm{h}^\sharp_\mathtt{f}$ accounts for all states found to enter the body of `f` so far, even though they may lead to no output state in $\mathrm{t}^\sharp_\mathtt{f}$ (e.g., if the evaluation of the body of `f` from them does not terminate or fails due to an error, as shown in Example 4).

Example 3 (Context summary). We let $\mathrm{h}^\sharp = \&\mathtt{l0} \mapsto \alpha_0 \ *_\mathtt{s}\ \mathbf{lseg}(\alpha_0, \alpha_1) \ *_\mathtt{s}\ \alpha_1 \cdot \underline{\mathtt{n}} \mapsto \mathbf{0x0} \ *_\mathtt{s}\ \&\mathtt{l1} \mapsto \alpha_2 \ *_\mathtt{s}\ \mathbf{list}(\alpha_2)$ and assume that $\mathrm{t}^\sharp$ is defined as in Example 2. Then, $(\mathrm{h}^\sharp, \mathrm{t}^\sharp)$ defines a valid context summary for `append` (Figure 1).

Example 4 (Role of the pre-condition approximation in context summaries). We consider the function below and assume it is always called in a state where `l0` is a valid pointer and `l1` points to a well-formed, but possibly empty, singly-linked list:

```
1  void getnext( list **l0, list *l1 ){ *l0 = l1->n; }
```

Obviously, this function will crash when the list is empty, i.e., when `l0` is the null pointer. However, the pair $(\mathrm{h}^\sharp_\mathtt{f}, \mathrm{t}^\sharp_\mathtt{f})$ below defines a valid transformation summary

for this procedure:

$$\begin{aligned} \mathrm{h}^\sharp_{\mathtt{f}} &= \&\mathtt{l0} \mapsto \alpha_0 *_{\mathtt{S}} \alpha_0 \mapsto \alpha_1 *_{\mathtt{S}} \&\mathtt{l1} \mapsto \alpha_2 *_{\mathtt{S}} \mathbf{list}(\alpha_2) \\ \mathrm{t}^\sharp_{\mathtt{f}} &= \mathtt{Id}(\&\mathtt{l0} \mapsto \alpha_0 *_{\mathtt{S}} \&\mathtt{l1} \mapsto \alpha_2 *_{\mathtt{S}} \alpha_2 \cdot \underline{\mathtt{n}} \mapsto \alpha_3 *_{\mathtt{S}} \mathbf{list}(\alpha_3)) \\ &\quad *_{\mathtt{T}} [\alpha_0 \mapsto \alpha_1 \dashrightarrow \alpha_0 \mapsto \alpha_3] \end{aligned}$$

We observe that the $\mathrm{h}^\sharp_{\mathtt{f}}$ component describes not only states for which the procedure returns but also states for which it crashes since the list pointer `cl1` is null. The former are not part of the concretization of the transformation component $\mathrm{h}^\sharp_{\mathtt{f}}$.

The above example shows the importance of the first component of the transformation summary: it conveys the fact that a set of states have been considered by the analysis as a pre-condition for a program fragment, even when the program fragment may not produce any post-condition for these states hereby they can be omitted from the transformation part.

5 Intraprocedural analysis

The analysis performs a forward abstract interpretation [9] over abstract transformations (rather than on abstract states). More precisely, the abstract semantics $[\![\mathtt{C}]\!]^\sharp_{\mathbb{T}}$ of a command `C` inputs a transformation describing the entire computation so far, before executing `C`, and outputs a new transformation that reflects the effect of `C` on top of that computation. Intuitively, the input of $[\![\mathtt{C}]\!]^\sharp_{\mathbb{T}}$ may be viewed the dual of a continuation. Formally, the analysis is designed so as to meet the following soundness statement, for any transformation $\mathrm{t}^\sharp$:

$$\forall(\sigma_0, \sigma_1, \nu) \in \gamma_{\mathbb{T}}(\mathrm{t}^\sharp), \sigma_2 \in \mathbb{M},\ (\sigma_1, \sigma_2) \in [\![\mathtt{C}]\!]_{\mathcal{T}} \Longrightarrow (\sigma_0, \sigma_2, \nu) \in \gamma_{\mathbb{T}}([\![\mathtt{C}]\!]^\sharp_{\mathbb{T}}(\mathrm{t}^\sharp)) \quad (1)$$

The analysis of assignments and loops follows from [17]. It may require finite disjunctions of transformations although we do not formalize this aspect since it is orthogonal to the goal of this paper. We recall the main aspects of their algorithms in this section and refer the reader to [17] for a full description.

Post-conditions for assignment. We consider an assignment command $\mathtt{x} \texttt{->} \underline{\mathtt{n}} = \mathtt{y}$ (the analysis of other kinds of commands is similar), and a pre-transformation $\mathrm{t}^\sharp$, and we discuss the computation of $[\![\mathtt{x} \texttt{->} \underline{\mathtt{n}} = \mathtt{y}]\!]^\sharp_{\mathbb{T}}(\mathrm{t}^\sharp)$. To do this, the analysis should first localize both $\mathtt{x} \texttt{->} \underline{\mathtt{n}}$ and $\mathtt{y}$ in the post-state of $\mathrm{t}^\sharp$, by rewriting $\mathrm{t}^\sharp$ into an expression of the form $\mathtt{Id}(\&\mathtt{x} \mapsto \alpha_0 *_{\mathtt{S}} \&\mathtt{y} \mapsto \alpha_1) *_{\mathtt{T}} \mathrm{t}^\sharp_0$ or $[(\ldots) \dashrightarrow (\&\mathtt{x} \mapsto \alpha_0 *_{\mathtt{S}} \&\mathtt{y} \mapsto \alpha_1)] *_{\mathtt{T}} \mathrm{t}^\sharp_0$, and searching for α_0 in $\mathrm{t}^\sharp_0$. Two main cases may arise:

– if $\mathrm{t}^\sharp_0$ contains a term of the form $\mathtt{Id}(\alpha_0 \cdot \underline{\mathtt{n}} \mapsto \alpha_2)$ or $[(\ldots) \dashrightarrow (\alpha_0 \cdot \underline{\mathtt{n}} \mapsto \alpha_2)]$, the post-transformation is derived by a *mutation* over the pointer cell, which produces a term of the form $[(\ldots) \dashrightarrow (\alpha_0 \cdot \underline{\mathtt{n}} \mapsto \alpha_1)]$;

– if $\mathrm{t}^\sharp_0$ contains a term $\mathtt{Id}(\mathrm{h}^\sharp_0)$ or $[(\ldots) \dashrightarrow \mathrm{h}^\sharp_0]$ where $\mathrm{h}^\sharp_0$ is either $\mathbf{list}(\alpha_0)$ or $\mathbf{lseg}(\alpha_0, \ldots)$, the summary should be unfolded so that the modified cell can be resolved as in the previous case; this step relies on relation $\rightarrow_{\mathfrak{U}}$ (Section 3).

It is also possible that the localization of `x -> n` fails, which typically indicates that the program being analyzed may dereference an invalid pointer. Besides assignment, the analysis also supports other such operations; for instance, we write $\mathbf{newV}_{\mathbb{T}}{}^{\sharp}[\mathtt{x}_0, \ldots, \mathtt{x}_n]$ (resp., $\mathbf{delV}_{\mathbb{T}}{}^{\sharp}[\mathtt{x}_0, \ldots, \mathtt{x}_n]$) for the operation that adds (resp., removes) variables $\mathtt{x}_0, \ldots, \mathtt{x}_n$ to the output state of the current transformation. They over-approximate concrete operations noted **newV** and **delV**.

Weakening. The analysis of loop statements proceeds by abstract iteration over the loop body with widening. Intuitively, the widening $\mathrm{t}_0^\sharp \triangledown_{\mathbb{T}} \mathrm{t}_1^\sharp$ of transformations $\mathrm{t}_0^\sharp, \mathrm{t}_1^\sharp$ returns a new transformation $\mathrm{t}^\sharp$, such that $\gamma_{\mathbb{T}}(\mathrm{t}_i^\sharp) \subseteq \gamma_{\mathbb{T}}(\mathrm{t}^\sharp)$ for all $i \in \{0, 1\}$. In the state level, widening replaces basic blocks with summaries (effectively inversing $\rightarrow_{\mathfrak{U}}$). In the transformation level, widening commutes with `Id` and $*_{\mathtt{T}}$ whenever their arguments can be widened as above, and weakens them into $[\dashrightarrow]$ transformations otherwise. Furthermore, this transformation introduces summary predicates so as to ensure termination of all widening sequences [17]. Similarly, $\mathrm{t}_0^\sharp \sqsubseteq_{\mathbb{T}} \mathrm{t}_1^\sharp$ conservatively decides inclusion test (if $\mathrm{t}_0^\sharp \sqsubseteq_{\mathbb{T}} \mathrm{t}_1^\sharp$ holds, then $\gamma_{\mathbb{T}}(\mathrm{t}_0^\sharp) \subseteq \gamma_{\mathbb{T}}(\mathrm{t}_1^\sharp)$).

Example 5 (Analysis of the loop of `append`). In this example, we consider the loop at line 4 in the `append` function (Figure 1) and only present the part of the memory reachable from `c`. The analysis of the loop starts with the transformation $\mathtt{Id}(\&\mathtt{l0} \mapsto \alpha *_{\mathtt{S}} \&\mathtt{c} \mapsto \alpha *_{\mathtt{S}} \mathbf{list}(\alpha)) \wedge \alpha \neq 0$. The analysis of the assignment inside the loop body forces the unfolding of **list**, and produces $\mathtt{Id}(\&\mathtt{l0} \mapsto \alpha *_{\mathtt{S}} \alpha \cdot \mathtt{n} \mapsto \alpha' *_{\mathtt{S}} \mathbf{list}(\alpha')) *_{\mathtt{T}} [(\&\mathtt{c} \mapsto \alpha) \dashrightarrow (\&\mathtt{c} \mapsto \alpha')] \wedge \alpha \neq 0$. The widening of these two transformations produces $\mathtt{Id}(\&\mathtt{l0} \mapsto \alpha *_{\mathtt{S}} \mathbf{lseg}(\alpha, \alpha') *_{\mathtt{S}} \mathbf{list}(\alpha')) *_{\mathtt{T}} [(\&\mathtt{c} \mapsto \alpha) \dashrightarrow (\&\mathtt{c} \mapsto \alpha')] \wedge \alpha \neq 0$, which also turns out to be the loop invariant.

6 Abstract composition

In this section, we set up the abstract operations that are required to rely on transformations for modular analysis. In Section 2, we mentioned *application* and *composition*. We remark that the application of a transformation $\mathrm{t}^\sharp$ to an abstract state $\mathrm{h}^\sharp$ boils down to the abstract composition of $\mathrm{t}^\sharp$ with $\mathtt{Id}(\mathrm{h}^\sharp)$, since the latter represents exactly the set of pairs (σ, σ) where σ is described by $\mathrm{h}^\sharp$. Moreover, we observed in Section 2 that composition requires to reason over intersection of abstract states. Thus, we only define intersection and composition in this section.

Abstract intersection. In this paragraph, we set up an abstract operator $\mathbf{inter}^\sharp$, which inputs two abstract states and returns a disjunctive abstract state that over-approximates their intersection. The computation over abstract heaps is guided by a set of rewriting rules that are shown in Figure 5. The predicate $\mathrm{h}_0^\sharp \sqcap \mathrm{h}_1^\sharp \rightsquigarrow_{\sqcap} H$ means that the computation of the intersection of $\mathrm{h}_0^\sharp$ and $\mathrm{h}_1^\sharp$ may produce the disjunction of abstract heaps H (there may exist several solutions for a given pair of arguments). We remark that the definition of $\gamma_{\mathbb{H}}$ lets symbolic variables be existentially quantified, thus they may be renamed without changing the concretization, following usual α-equivalence. Therefore, the rules of Figure 5 assume that both

$$\frac{\mathrm{h}^\sharp_{0,0} \sqcap \mathrm{h}^\sharp_{1,0} \rightsquigarrow_\sqcap H_0 \qquad \mathrm{h}^\sharp_{0,1} \sqcap \mathrm{h}^\sharp_{1,1} \rightsquigarrow_\sqcap H_1}{(\mathrm{h}^\sharp_{0,0} *_\mathtt{s} \mathrm{h}^\sharp_{0,1}) \sqcap (\mathrm{h}^\sharp_{1,0} *_\mathtt{s} \mathrm{h}^\sharp_{1,1}) \rightsquigarrow_\sqcap \{\mathrm{h}^\sharp_0 *_\mathtt{s} \mathrm{h}^\sharp_1 \mid \mathrm{h}^\sharp_0 \in H_0 \wedge \mathrm{h}^\sharp_1 \in H_1\}} \sqcap_{*_\mathtt{s}}$$

$$\frac{\mathrm{h}^\sharp \text{ is either } \mathbf{emp},\ \text{or } \mathrm{n} \cdot \underline{\mathtt{f}} \mapsto \mathrm{n}',\ \text{or } \mathbf{list}(\alpha),\ \text{or } \mathbf{lseg}(\alpha, \alpha')}{\mathrm{h}^\sharp \sqcap \mathrm{h}^\sharp \rightsquigarrow_\sqcap \{\mathrm{h}^\sharp\}} \sqcap_=$$

$$\frac{\mathbf{list}(\alpha_1) \sqcap \mathrm{h}^\sharp_1 \rightsquigarrow_\sqcap H}{\mathbf{list}(\alpha_0) \sqcap (\mathbf{lseg}(\alpha_0, \alpha_1) *_\mathtt{s} \mathrm{h}^\sharp_1) \rightsquigarrow_\sqcap \{\mathbf{lseg}(\alpha_0, \alpha_1) *_\mathtt{s} \mathrm{h}^\sharp \mid \mathrm{h}^\sharp \in H\}} \sqcap[l, s]$$

$$\frac{\mathbf{lseg}(\alpha_1, \alpha_2) \sqcap \mathrm{h}^\sharp_1 \rightsquigarrow_\sqcap H \qquad \alpha_2 \neq \alpha_1}{\mathbf{lseg}(\alpha_0, \alpha_2) \sqcap (\mathbf{lseg}(\alpha_0, \alpha_1) *_\mathtt{s} \mathrm{h}^\sharp_1) \rightsquigarrow_\sqcap \{\mathbf{lseg}(\alpha_0, \alpha_1) *_\mathtt{s} \mathrm{h}^\sharp \mid \mathrm{h}^\sharp \in H\}} \sqcap[s, s]$$

$$\frac{\mathrm{h}^\sharp_0 \text{ contains a } \mathbf{list}(\alpha_0) \text{ or } \mathbf{lseg}(\alpha_0, \alpha_1) \text{ term} \qquad \alpha_0 \text{ carries no summary in } \mathrm{h}^\sharp_1}{\mathrm{h}^\sharp_0 \sqcap \mathrm{h}^\sharp_1 \rightsquigarrow_\sqcap \{\mathrm{h}^\sharp \mid \exists \mathrm{h}^\sharp_{0,\mathrm{u}}, H,\ (\mathrm{h}^\sharp_0 \rightarrow_{\mathfrak{U}[\alpha_0]} \mathrm{h}^\sharp_{0,\mathrm{u}}) \wedge (\mathrm{h}^\sharp_{0,\mathrm{u}} \sqcap \mathrm{h}^\sharp_1 \rightsquigarrow_\sqcap H) \wedge \mathrm{h}^\sharp \in H\}} \sqcap_u$$

Fig. 5. Abstract intersection rewriting rules.

arguments follow a consistent naming, although the implementation should perform α-equivalence whenever needed and maintain a correspondence of symbolic variables [5]. Rule $\sqcap_{*_\mathtt{s}}$ states that intersection can be computed locally. Rule $\sqcap_=$ expresses that intersection behaves like identity when both of its arguments are the same basic term. Rules $\sqcap[l, s]$ and $\sqcap[s, s]$ implement structural reasoning over summaries. Finally, rule $\sqcap_u$ unfolds one argument so as to consider all subsequent cases. The result may differ depending on the order of application or even on the way each rule is applied. As an example, $\sqcap_{*_\mathtt{s}}$ may produce different results depending on the way both arguments are split into $\mathrm{h}^\sharp_{i,0}$ and $\mathrm{h}^\sharp_{i,1}$, which may affect precision. Therefore, our implementation follows a carefully designed application strategy that attempts to maximize the use of $\sqcap_=$. We omit the numerical predicate intersection (handled by a numerical domain intersection operator). Given two abstract heaps $\mathrm{h}^\sharp_0, \mathrm{h}^\sharp_1$, the computation of $\mathbf{inter}^\sharp(\mathrm{h}^\sharp_0, \mathrm{h}^\sharp_1)$ proceeds by proof search following the rules of Figure 5 up-to commutativity (standard rule, not shown). In case this system fails to infer a solution, returning either argument provides a sound result.

Definition 3 (Abstract intersection algorithm). *The operator* $\mathbf{inter}^\sharp$ *is a partial function that inputs two abstract heaps* $\mathrm{h}^\sharp_0, \mathrm{h}^\sharp_1$ *and returns a disjunction of abstract heaps* H *such that* $\mathrm{h}^\sharp_0 \sqcap \mathrm{h}^\sharp_1 \rightsquigarrow_\sqcap H$ *following Figure 5.*

Theorem 1 (Soundness of abstract intersection). *Abstract intersection is sound in the sense that, for all* $\mathrm{h}^\sharp_0, \mathrm{h}^\sharp_1$, $\gamma_\mathbb{H}(\mathrm{h}^\sharp_0) \cap \gamma_\mathbb{H}(\mathrm{h}^\sharp_1) \subseteq \gamma_\mathbb{H}(\mathbf{inter}^\sharp(\mathrm{h}^\sharp_0, \mathrm{h}^\sharp_1))$.

Example 6 (Abstract intersection). Let us consider:

- $\mathrm{h}^\sharp_0 = \&\mathtt{x} \mapsto \alpha_0 *_\mathtt{s} \&\mathtt{y} \mapsto \alpha_2 *_\mathtt{s} \mathbf{lseg}(\alpha_0, \alpha_2) *_\mathtt{s} \alpha_2 \cdot \underline{\mathtt{n}} \mapsto \alpha_3 *_\mathtt{s} \mathbf{list}(\alpha_3)$ and
- $\mathrm{h}^\sharp_1 = \&\mathtt{x} \mapsto \alpha_0 *_\mathtt{s} \&\mathtt{y} \mapsto \alpha_2 *_\mathtt{s} \mathbf{lseg}(\alpha_0, \alpha_1) *_\mathtt{s} \alpha_1 \cdot \underline{\mathtt{n}} \mapsto \alpha_2 *_\mathtt{s} \mathbf{list}(\alpha_2)$.

Then, $\mathbf{inter}^\sharp(\mathrm{h}^\sharp_0, \mathrm{h}^\sharp_1)$ returns $\&\mathtt{x} \mapsto \alpha_0 *_\mathtt{s} \&\mathtt{y} \mapsto \alpha_2 *_\mathtt{s} \mathbf{lseg}(\alpha_0, \alpha_1) *_\mathtt{s} \alpha_1 \cdot \underline{\mathtt{n}} \mapsto \alpha_2 *_\mathtt{s} \alpha_2 \cdot \underline{\mathtt{n}} \mapsto \alpha_3 *_\mathtt{s} \mathbf{list}(\alpha_3)$. Note that the computation involves the structural

$$\frac{\mathrm{t}^\sharp_{0,0} \fatsemi \mathrm{t}^\sharp_{1,0} \rightsquigarrow_\fatsemi T_0 \qquad \mathrm{t}^\sharp_{0,1} \fatsemi \mathrm{t}^\sharp_{1,1} \rightsquigarrow_\fatsemi T_1}{(\mathrm{t}^\sharp_{0,0} *_\mathtt{T} \mathrm{t}^\sharp_{0,1}) \fatsemi (\mathrm{t}^\sharp_{1,0} *_\mathtt{T} \mathrm{t}^\sharp_{1,1}) \rightsquigarrow_\fatsemi \{\mathrm{t}^\sharp_0 *_\mathtt{T} \mathrm{t}^\sharp_1 \mid \mathrm{t}^\sharp_0 \in T_0 \wedge \mathrm{t}^\sharp_1 \in T_1\}} \fatsemi_{*_\mathtt{T}}$$

$$\frac{\mathrm{h}^\sharp_0 \sqcap \mathrm{h}^\sharp_1 \rightsquigarrow_\sqcap H}{\mathtt{Id}(\mathrm{h}^\sharp_0) \fatsemi \mathtt{Id}(\mathrm{h}^\sharp_1) \rightsquigarrow_\fatsemi \{\mathtt{Id}(\mathrm{h}^\sharp) \mid \mathrm{h}^\sharp \in H\}} \fatsemi_\mathtt{Id}$$

$$\frac{\mathrm{h}^\sharp_{0,\mathrm{o}} \sqcap \mathrm{h}^\sharp_{1,\mathrm{i}} \rightsquigarrow_\sqcap H \quad H \neq \emptyset}{[\mathrm{h}^\sharp_{0,\mathrm{i}} \dashrightarrow \mathrm{h}^\sharp_{0,\mathrm{o}}] \fatsemi [\mathrm{h}^\sharp_{1,\mathrm{i}} \dashrightarrow \mathrm{h}^\sharp_{1,\mathrm{o}}] \rightsquigarrow_\fatsemi [\mathrm{h}^\sharp_{0,\mathrm{i}} \dashrightarrow \mathrm{h}^\sharp_{1,\mathrm{o}}]} \fatsemi_\dashrightarrow$$

$$\frac{\mathrm{h}^\sharp_0 \sqcap \mathrm{h}^\sharp_{1,\mathrm{i}} \rightsquigarrow_\sqcap H}{\mathtt{Id}(\mathrm{h}^\sharp_0) \fatsemi [\mathrm{h}^\sharp_{1,\mathrm{i}} \dashrightarrow \mathrm{h}^\sharp_{1,\mathrm{o}}] \rightsquigarrow_\fatsemi \{[\mathrm{h}^\sharp_{\mathrm{i}} \dashrightarrow \mathrm{h}^\sharp_{1,\mathrm{o}}] \mid \mathrm{h}^\sharp_{\mathrm{i}} \in H\}} \fatsemi_{\mathtt{Id},\dashrightarrow,l}$$

$$\frac{[(\mathrm{h}^\sharp_0 *_\mathtt{S} \mathrm{h}^\sharp_{0,\mathrm{i}}) \dashrightarrow (\mathrm{h}^\sharp_0 *_\mathtt{S} \mathrm{h}^\sharp_{0,\mathrm{o}})] \fatsemi \mathrm{t}^\sharp_1 \rightsquigarrow_\fatsemi T}{(\mathtt{Id}(\mathrm{h}^\sharp_0) *_\mathtt{T} [\mathrm{h}^\sharp_{0,\mathrm{i}} \dashrightarrow \mathrm{h}^\sharp_{0,\mathrm{o}}]) \fatsemi \mathrm{t}^\sharp_1 \rightsquigarrow_\fatsemi T} \fatsemi_{\mathbf{weak},\mathtt{Id},l}$$

$$\frac{([(\mathrm{h}^\sharp_{0,\mathrm{i}} *_\mathtt{S} \mathrm{h}^\sharp_{1,\mathrm{i}}) \dashrightarrow (\mathrm{h}^\sharp_{0,\mathrm{o}} *_\mathtt{S} \mathrm{h}^\sharp_{1,\mathrm{o}})] *_\mathtt{T} \mathrm{t}^\sharp_2) \fatsemi \mathrm{t}^\sharp_3 \rightsquigarrow_\fatsemi T}{([\mathrm{h}^\sharp_{0,\mathrm{i}} \dashrightarrow \mathrm{h}^\sharp_{0,\mathrm{o}}] *_\mathtt{T} [\mathrm{h}^\sharp_{1,\mathrm{i}} \dashrightarrow \mathrm{h}^\sharp_{1,\mathrm{o}}] *_\mathtt{T} \mathrm{t}^\sharp_2) \fatsemi \mathrm{t}^\sharp_3 \rightsquigarrow_\fatsemi T} \fatsemi_{\mathbf{weak},*_\mathtt{T},l}$$

$$\frac{\mathrm{t}^\sharp_0 \text{ contains an } \mathbf{list}(\alpha_0) \text{ or } \mathbf{lseg}(\alpha_0, \alpha_1) \text{ term}}{\mathrm{t}^\sharp_0 \fatsemi \mathrm{t}^\sharp_1 \rightsquigarrow_\fatsemi \{\mathrm{t}^\sharp \mid \exists \mathrm{t}^\sharp_{0,\mathrm{u}},\ (\mathrm{t}^\sharp_0 \rightarrow_{\mathfrak{U}[\alpha_0]} \mathrm{t}^\sharp_{0,\mathrm{u}}) \wedge (\mathrm{t}^\sharp_{0,\mathrm{u}} \fatsemi \mathrm{t}^\sharp_1 \rightsquigarrow_\fatsemi T) \wedge \mathrm{t}^\sharp \in T\}} \fatsemi_{\mathbf{unf},l}$$

Fig. 6. Abstract composition rewriting rules (rules $\fatsemi_{\mathtt{Id},\dashrightarrow,r}$, $\fatsemi_{\mathbf{weak},\mathtt{Id},r}$, $\fatsemi_{\mathbf{weak},*_\mathtt{T},r}$, and $\fatsemi_{\mathbf{unf},r}$ which are right versions of rules $\fatsemi_{\mathtt{Id},\dashrightarrow,l}$, $\fatsemi_{\mathbf{weak},\mathtt{Id},l}$, $\fatsemi_{\mathbf{weak},*_\mathtt{T},l}$, and $\fatsemi_{\mathbf{unf},l}$, can be systematically derived by symmetry, and are omitted for the sake of brevity).

rule $\sqcap[s,s]$ and the unfolding rule to derive this result, where both the segment between `x` and `y` and the list pointed to by `y` are non-empty. This result is exact (no precision is lost) and the result is effectively more precise than both arguments.

Composition of abstract transformations. We now study the composition of abstract transformations. Again, the computation is based on a rewriting system, that gradually processes two input transformations into an abstraction of their composition. The rules are provided in Figure 6. The predicate $\mathrm{t}^\sharp_0 \fatsemi \mathrm{t}^\sharp_1 \rightsquigarrow_\fatsemi T$ means that the effect of applying transformation $\mathrm{t}^\sharp_0$ and then transformation $\mathrm{t}^\sharp_1$ can be described by the union of the transformations in T. Rule $\fatsemi_{*_\mathtt{T}}$ enables local reasoning over composition, at the transformation level. Rules $\fatsemi_\mathtt{Id}$ and $\fatsemi_\dashrightarrow$ respectively compose matching identity transformations and matching modifying transformations. Similarly, $\fatsemi_{\mathtt{Id},\dashrightarrow,l}$ composes an identity followed by a modifying transformation with a consistent support (this rule, as the following, has a corresponding right version that we omit for the sake of brevity). Rule $\fatsemi_{\mathbf{weak},\mathtt{Id},l}$ implements a weakening based on the inclusion $\gamma_\mathbb{T}(\mathtt{Id}(\mathrm{t}^\sharp)) \subseteq \gamma_\mathbb{T}([\mathrm{t}^\sharp \dashrightarrow \mathrm{t}^\sharp])$ (the inclusion is proved in [17]). Similarly, rule $\fatsemi_{\mathbf{weak},*_\mathtt{T},l}$ weakens $[\mathrm{h}^\sharp_{0,\mathrm{i}} \dashrightarrow \mathrm{h}^\sharp_{0,\mathrm{o}}] *_\mathtt{T} [\mathrm{h}^\sharp_{1,\mathrm{i}} \dashrightarrow \mathrm{h}^\sharp_{1,\mathrm{o}}]$ into $[(\mathrm{h}^\sharp_{0,\mathrm{i}} *_\mathtt{S} \mathrm{h}^\sharp_{1,\mathrm{i}}) \dashrightarrow (\mathrm{h}^\sharp_{0,\mathrm{o}} *_\mathtt{S} \mathrm{h}^\sharp_{1,\mathrm{o}})]$. Finally, rule $\fatsemi_{\mathbf{unf},l}$ unfolds a summary to enable composition. The abstract composition operation performs a proof search. Just as for intersection, the composition operator may produce different results depending on the application order; our implementation relies on a strategy designed to improve precision by maximizing the use of $\fatsemi_\mathtt{Id}$.

Definition 4 (Abstract composition algorithm). *The operator* $\mathbf{comp}^\sharp$ *is a partial function that inputs two abstract transformations* $\mathrm{t}_0^\sharp, \mathrm{t}_1^\sharp$ *and returns a set of abstract transformations* T *such that* $\mathrm{t}_0^\sharp \fatsemi \mathrm{t}_1^\sharp \rightsquigarrow_\fatsemi T$ *following Figure 6.*

Theorem 2 (Soundness of abstract composition). *Let* $\mathrm{t}_0^\sharp, \mathrm{t}_1^\sharp$ *be two transformations,* $\mathrm{h}_0^\sharp, \mathrm{h}_1^\sharp, \mathrm{h}_2^\sharp$ *be abstract heaps and* ν *be a valuation. We assume that* $\mathbf{comp}^\sharp(\mathrm{t}_0^\sharp, \mathrm{t}_1^\sharp)$ *evaluates to the set of transformations* T*. Then:*

$$(\sigma_0, \sigma_1, \nu) \in \gamma_{\mathbb{T}}(\mathrm{t}_0^\sharp) \wedge (\sigma_1, \sigma_2, \nu) \in \gamma_{\mathbb{T}}(\mathrm{t}_1^\sharp) \Longrightarrow \exists \mathrm{t}^\sharp \in T,\ (\sigma_0, \sigma_2, \nu) \in \gamma_{\mathbb{T}}(\mathrm{t}^\sharp)$$

Example 7 (Abstract composition and analysis compositionality). In this example, we study the classical case of an in-place list reverse code snippet:

```
1  // l points to a list
2  list *c = l; list *x = 0;
3  while( l != NULL ){ c = l->n; l->n = x; x = l; l = c; }
```

The effects of `c = l->n` and of `l->n = x` can be described by the abstract transformations $\mathrm{t}_0^\sharp$ and $\mathrm{t}_1^\sharp$:

$$\begin{aligned}
\mathrm{t}_0^\sharp &= \mathtt{Id}(\&\mathtt{l} \mapsto \alpha_0 *_{\mathtt{S}} \alpha_0 \cdot \underline{\mathtt{n}} \mapsto \alpha_1 *_{\mathtt{S}} \&\mathtt{x} \mapsto \alpha_2 *_{\mathtt{S}} \mathbf{list}(\alpha_1) *_{\mathtt{S}} \mathbf{list}(\alpha_2)) \\
&\quad *_{\mathtt{T}} [(\&\mathtt{c} \mapsto \alpha_3) \dashrightarrow (\&\mathtt{c} \mapsto \alpha_1)] \\
\mathrm{t}_1^\sharp &= \mathtt{Id}(\&\mathtt{l} \mapsto \alpha_0 *_{\mathtt{S}} \&\mathtt{x} \mapsto \alpha_2 *_{\mathtt{S}} \&\mathtt{c} \mapsto \alpha_1 *_{\mathtt{S}} \mathbf{list}(\alpha_1) *_{\mathtt{S}} \mathbf{list}(\alpha_2)) \\
&\quad *_{\mathtt{T}} [(\alpha_0 \cdot \underline{\mathtt{n}} \mapsto \alpha_1) \dashrightarrow (\alpha_0 \cdot \underline{\mathtt{n}} \mapsto \alpha_2)]
\end{aligned}$$

The composition of these two transformations needs to apply the weakening rules to match terms that are under the `Id` constructors. The result is the following transformation $\mathrm{t}_0^\sharp \fatsemi \mathrm{t}_1^\sharp$:

$$\begin{aligned}
&\mathtt{Id}(\&\mathtt{l} \mapsto \alpha_0 *_{\mathtt{S}} \&\mathtt{x} \mapsto \alpha_2 *_{\mathtt{S}} \mathbf{list}(\alpha_1) *_{\mathtt{S}} \mathbf{list}(\alpha_2)) \\
*_{\mathtt{T}} [(\&\mathtt{c} \mapsto \alpha_3) \dashrightarrow (\&\mathtt{c} \mapsto \alpha_1)] &*_{\mathtt{T}} [(\alpha_0 \cdot \underline{\mathtt{n}} \mapsto \alpha_1) \dashrightarrow (\alpha_0 \cdot \underline{\mathtt{n}} \mapsto \alpha_2)]
\end{aligned}$$

This description is actually a very precise account for the effect of the sequence of these two assignment commands. This example shows that composition may be used not only for interprocedural analysis (as we show in the next section), but also to supersede some operations of the intraprocedural analysis of Section 5.

Example 8 (Abstract application). As observed at the beginning of the section, composition may also be used as a means to analyze the application of a transformation to an abstract state. We consider the composition of the transformation $\mathrm{t}^\sharp$ corresponding to function `append` (shown in Example 2 and Figure 1) and the abstract pre-state $\mathrm{h}^\sharp = \&\mathtt{l0} \mapsto \alpha_0 *_{\mathtt{S}} \&\mathtt{l1} \mapsto \alpha_2 *_{\mathtt{S}} \mathbf{lseg}(\alpha_0, \alpha_1) *_{\mathtt{S}} \alpha_1 \cdot \underline{\mathtt{n}} \mapsto \alpha_3 *_{\mathtt{S}} \mathbf{list}(\alpha_2) \wedge \alpha_3 = \mathbf{0x0}$. Then, the composition $\mathtt{Id}(\mathrm{h}^\sharp) \fatsemi \mathrm{t}^\sharp$ returns:

$$\&\mathtt{l0} \mapsto \alpha_0 *_{\mathtt{S}} \&\mathtt{l1} \mapsto \alpha_2 *_{\mathtt{S}} \mathbf{lseg}(\alpha_0, \alpha_1) *_{\mathtt{S}} \alpha_1 \cdot \underline{\mathtt{n}} \mapsto \alpha_2 *_{\mathtt{S}} \mathbf{list}(\alpha_2)$$

7 Interprocedural analysis based on function summaries

In this section, we study two mutually dependent aspects: the application of summaries at call-sites and their inference by static analysis in a top down manner. We first focus on non-recursive calls and discuss recursive calls at the end of the section. The analysis maintains a context summary $(\mathrm{h}^\sharp_{\mathtt{f}}, \mathrm{t}^\sharp_{\mathtt{f}})$ for each procedure `f`. Initially, this context summary is set to $(\bot, \bot)$. When the analysis reaches a call to procedure `f` it should attempt to utilize the existing summary (Section 7.1). When the existing summary does not account for all the states that may reach the current context, a new summary needs to be computed first (Section 7.2).

7.1 Analysis of a call site using an existing summary

We assume a procedure **proc** $f(\mathtt{p}_0, \ldots, \mathtt{p}_k)\{\mathtt{C}\}$ and a sound context summary $(\mathrm{h}^\sharp_{\mathtt{f}}, \mathrm{t}^\sharp_{\mathtt{f}})$. We consider the analysis $[\![\mathtt{f}(\mathtt{x}_0, \ldots, \mathtt{x}_k)]\!]^\sharp_{\mathbb{T}}$ of a call to this procedure, with transformation $\mathrm{t}^\sharp_{\mathrm{pre}}$ as a pre-transformation. To analyze the call, the analysis should (1) process parameter passing, (2) detect which part of $\mathrm{t}^\sharp_{\mathrm{pre}}$ may be modified by `f`, (3) check whether the context summary covers this context, and (4) apply the summary, if (3) succeeds (the case where it fails is studied in Section 7.2).

Parameter passing. Parameter passing boils down to creating the variables $\mathtt{p}_0, \ldots, \mathtt{p}_k$ using transfer function $\mathbf{newV}_{\mathbb{T}}{}^\sharp$ and then to analyzing assignment statements $\mathtt{p}_0 = \mathtt{x}_0, \ldots, \mathtt{p}_k = \mathtt{x}_k$. These operations can all be done using the transfer functions defined in Section 5:

$$\mathrm{t}^\sharp_{\mathrm{pars}} = ([\![\mathtt{p}_k = \mathtt{x}_k]\!]^\sharp_{\mathbb{T}} \circ \ldots \circ [\![\mathtt{p}_0 = \mathtt{x}_0]\!]^\sharp_{\mathbb{T}} \circ \mathbf{newV}_{\mathbb{T}}{}^\sharp[\mathtt{p}_0, \ldots, \mathtt{p}_k])(\mathrm{t}^\sharp_{\mathrm{pre}})$$

Procedure footprint. To identify the fragment of the abstract heap that `f` can view and may modify, the analysis should first extract from $\mathrm{t}^\sharp_{\mathrm{pars}}$ an abstraction of the set of states that enter the body of `f`. This is the goal of function $\mathcal{O} : \mathbb{T} \to \mathbb{H}$:

$$\mathcal{O}(\mathtt{Id}(\mathrm{h}^\sharp)) = \mathrm{h}^\sharp \qquad \mathcal{O}(\mathrm{t}^\sharp_0 *_{\mathtt{T}} \mathrm{t}^\sharp_1) = \mathcal{O}(\mathrm{t}^\sharp_0) *_{\mathtt{S}} \mathcal{O}(\mathrm{t}^\sharp_1)$$
$$\mathcal{O}([\mathrm{h}^\sharp_0 \dashrightarrow \mathrm{h}^\sharp_1]) = \mathrm{h}^\sharp_1 \qquad \mathcal{O}(\mathrm{t}^\sharp \wedge \mathrm{c}^\sharp) = \mathcal{O}(\mathrm{t}^\sharp) \wedge \mathrm{c}^\sharp$$

Intuitively, $\mathcal{O}$ projects the "output" part of a transformation. Thus $\mathcal{O}(\mathrm{t}^\sharp_{\mathrm{pars}})$ over-approximates the set of states that enter `C`. However, only the fragment of $\mathcal{O}(\mathrm{t}^\sharp_{\mathrm{pars}})$ that is reachable from the parameters of `f` is relevant. Given an abstract heap $\mathrm{h}^\sharp$, we can compute the set of symbolic names $R[\mathrm{h}^\sharp]$ that are relevant based on the following rules:

$$\frac{}{(\&\mathtt{p}_i) \in R[\mathrm{h}^\sharp]} \qquad \frac{\mathrm{n} \in R[\mathrm{h}^\sharp] \qquad \mathrm{h}^\sharp \text{ contains a term } \mathrm{n} \cdot \underline{\mathtt{f}} \mapsto \mathrm{n}' \text{ or } \mathbf{lseg}(\mathrm{n}, \mathrm{n}')}{\mathrm{n}' \in R[\mathrm{h}^\sharp]}$$

The slice $\mathcal{R}[\mathtt{p}_0, \ldots, \mathtt{p}_k](\mathrm{h}^\sharp)$ of $\mathrm{h}^\sharp$ with respect to $\mathtt{p}_0, \ldots, \mathtt{p}_k$ retains only the terms of $\mathrm{h}^\sharp$ that contain only names in $R[\mathrm{h}^\sharp]$ defined as the least solution of the above rules. Similarly, we let $\mathcal{I}[\mathtt{p}_0, \ldots, \mathtt{p}_k](\mathrm{h}^\sharp)$ be the abstract heap made of the remaining terms. Therefore, we have the equality $\mathrm{h}^\sharp = \mathcal{R}[\mathtt{p}_0, \ldots, \mathtt{p}_k](\mathrm{h}^\sharp) *_{\mathtt{S}} \mathcal{I}[\mathtt{p}_0, \ldots, \mathtt{p}_k](\mathrm{h}^\sharp)$.

Context summary coverage test. Based on the above, the set of states that reach the body of `f` under the calling context defined by $\mathrm{t}^\sharp_{\mathrm{pre}}$ can be over-approximated by $\mathrm{h}^\sharp_{\mathrm{in},\mathtt{f}} = \mathcal{R}[\mathtt{p}_0,\ldots,\mathtt{p}_k](\mathcal{O}(\mathrm{t}^\sharp_{\mathrm{pars}}))$. We let $\mathrm{h}^\sharp_{\mathrm{rem}} = \mathcal{I}[\mathtt{p}_0,\ldots,\mathtt{p}_k](\mathcal{O}(\mathrm{t}^\sharp_{\mathrm{pars}}))$ be the remainder part. The context summary $(\mathrm{h}^\sharp_{\mathtt{f}}, \mathrm{t}^\sharp_{\mathtt{f}})$ covers $\mathrm{h}^\sharp_{\mathrm{in},\mathtt{f}}$ if and only if $\mathrm{h}^\sharp_{\mathrm{in},\mathtt{f}} \sqsubseteq_{\mathbb{H}} \mathrm{h}^\sharp_{\mathtt{f}}$ holds where $\sqsubseteq_{\mathbb{H}}$ is a sound abstract state inclusion test as defined in, e.g., [13,6]. When this condition does not hold, the context summary should be re-computed with a more general pre-condition (this point is discussed in Section 7.2).

Summary application. Given the above notation, the effect of the procedure (described by $\mathrm{t}^\sharp_{\mathtt{f}}$) should be applied to $\mathrm{h}^\sharp_{\mathrm{in},\mathtt{f}}$ whereas $\mathrm{h}^\sharp_{\mathrm{rem}}$ should be preserved. To do this, the following transformation should be composed with the abstract transformation $\mathrm{t}^\sharp_{\mathtt{f}} *_{\mathtt{T}} \mathtt{Id}(\mathrm{h}^\sharp_{\mathrm{rem}})$ (note that the identity part is applied to the part of the pre-transformation that is not relevant to the execution of the body of `f`). Thus, the transformation that accounts for the computation from the program entry point till the return point of `f` is:

$$\mathbf{delV}_{\mathbb{T}^\sharp}[\mathtt{p}_0,\ldots\mathtt{p}_k](\mathbf{comp}^\sharp(\mathrm{t}^\sharp_{\mathrm{pars}}, \mathrm{t}^\sharp_{\mathtt{f}} *_{\mathtt{T}} \mathtt{Id}(\mathrm{h}^\sharp_{\mathrm{rem}})))$$

Example 9 (Context summary coverage and application). In this example, we assume the context summary defined in Example 3 for procedure `append`:

- $\mathrm{h}^\sharp = \&\mathtt{l0} \mapsto \alpha_0 *_{\mathtt{S}} \mathbf{lseg}(\alpha_0,\alpha_1) *_{\mathtt{S}} \alpha_1 \cdot \underline{\mathtt{n}} \mapsto \mathbf{0x0} *_{\mathtt{S}} \&\mathtt{l1} \mapsto \alpha_2 *_{\mathtt{S}} \mathbf{list}(\alpha_2)$;
- $\mathrm{t}^\sharp = \mathtt{Id}(\&\mathtt{l0} \mapsto \alpha_0 *_{\mathtt{S}} \mathbf{lseg}(\alpha_0,\alpha_1)) *_{\mathtt{T}} [(\alpha_1 \cdot \underline{\mathtt{n}} \mapsto \mathbf{0x0}) \dashrightarrow (\alpha_1 \cdot \underline{\mathtt{n}} \mapsto \alpha_2)] *_{\mathtt{T}} \mathtt{Id}(\&\mathtt{l1} \mapsto \alpha_2 *_{\mathtt{S}} \mathbf{list}(\alpha_2))$

Moreover, we consider the call `append(a,b)` with the abstract transformation below as a pre-condition (note that variable `c` is not accessed by `append`):

$$\mathtt{Id}(\&\mathtt{a} \mapsto \alpha_0 *_{\mathtt{S}} \alpha_0 \cdot \underline{\mathtt{n}} \mapsto \mathbf{0x0} *_{\mathtt{S}} \&\mathtt{b} \mapsto \alpha_1 *_{\mathtt{S}} \alpha_1 \cdot \underline{\mathtt{n}} \mapsto \mathbf{0x0} *_{\mathtt{S}} \&\mathtt{c} \mapsto \alpha_3)$$

After parameter passing, computation of the heap fragment `f` may view, and projection of the output, we obtain the abstract state $\&\mathtt{l0} \mapsto \alpha_0 *_{\mathtt{S}} \alpha_0 \cdot \underline{\mathtt{n}} \mapsto \mathbf{0x0} *_{\mathtt{S}} \&\mathtt{l1} \mapsto \alpha_1 *_{\mathtt{S}} \alpha_1 \cdot \underline{\mathtt{n}} \mapsto \mathbf{0x0}$, which is obviously included in $\mathrm{h}^\sharp$. The composition with the summary of the procedure produces the abstract transformation below:

$$\begin{aligned}\mathtt{Id}(\&\mathtt{a} \mapsto \alpha_0) *_{\mathtt{T}} [(\alpha_0 \cdot \underline{\mathtt{n}} \mapsto \mathbf{0x0}) \dashrightarrow (\alpha_0 \cdot \underline{\mathtt{n}} \mapsto \alpha_1)] \\ *_{\mathtt{T}} \mathtt{Id}(\&\mathtt{b} \mapsto \alpha_1 *_{\mathtt{S}} \alpha_1 \cdot \underline{\mathtt{n}} \mapsto \mathbf{0x0} *_{\mathtt{S}} \&\mathtt{c} \mapsto \alpha_3)\end{aligned}$$

The whole algorithm is shown in Figure 7. It implements the steps described above and in Section 7.2. The case considered in this subsection (when $\mathrm{h}^\sharp_{\mathrm{in},\mathtt{f}} \sqsubseteq_{\mathbb{H}} \mathrm{h}^\sharp_{\mathtt{f}}$ holds) corresponds to the case where the **if** branch at lines 5–6 is not taken.

7.2 Inference of a new context summary

We now discuss the case where the previously existing context summary of `f` does not cover the executions corresponding to $\mathrm{t}^\sharp$. As mentioned above, this corresponds to the case where the abstract inclusion $\mathrm{h}^\sharp_{\mathrm{in},\mathtt{f}} \sqsubseteq_{\mathbb{H}} \mathrm{h}^\sharp_{\mathtt{f}}$ does not hold.

Data: Existing context summary $(\mathrm{h}^\sharp_{\mathtt{f}}, \mathrm{t}^\sharp_{\mathtt{f}})$ for **proc** $f(\mathtt{p}_0, \ldots, \mathtt{p}_k)\{\mathtt{C}\}$
Data: Input transformation $\mathrm{t}^\sharp_{\mathrm{pre}}$ (computation before the call)
Result: Output transformation $\mathrm{t}^\sharp_{\mathrm{post}}$ (computation before the call + body of $\mathtt{f}$)
Result: Update of the context summary if it does not cover the context of $\mathrm{t}^\sharp_{\mathrm{pre}}$

1 $\mathrm{t}^\sharp_{\mathrm{pars}} \leftarrow [\![\mathtt{p}_k = \mathtt{x}_k]\!]^\sharp_{\mathbb{T}} \circ \ldots \circ [\![\mathtt{p}_0 = \mathtt{x}_0]\!]^\sharp_{\mathbb{T}} \circ \mathbf{newV}_{\mathbb{T}^\sharp}[\mathtt{p}_0, \ldots, \mathtt{p}_k](\mathrm{t}^\sharp_{\mathrm{pre}})$;
2 $\mathrm{h}^\sharp_{\mathrm{in},\mathtt{f}} = \mathcal{R}[\mathtt{p}_0, \ldots, \mathtt{p}_k](\mathcal{O}(\mathrm{t}^\sharp_{\mathrm{pars}}))$;
3 $\mathrm{h}^\sharp_{\mathrm{rem}} = \mathcal{I}[\mathtt{p}_0, \ldots, \mathtt{p}_k](\mathcal{O}(\mathrm{t}^\sharp_{\mathrm{pars}}))$;
4 **if** $\neg(\mathrm{h}^\sharp_{\mathrm{in},f} \sqsubseteq_{\mathbb{H}} \mathrm{h}^\sharp_f)$ **then**
5 | $\mathrm{h}^\sharp_{\mathtt{f}} \leftarrow \mathrm{h}^\sharp_{\mathtt{f}} \triangledown_{\mathbb{H}} \mathrm{h}^\sharp_{\mathrm{in},\mathtt{f}}$;
6 | $\mathrm{t}^\sharp_{\mathtt{f}} \leftarrow [\![\mathtt{C}]\!]^\sharp_{\mathbb{T}}(\mathtt{Id}(\mathrm{h}^\sharp_{\mathtt{f}}))$
7 **end**
8 $\mathrm{t}^\sharp_{\mathrm{post}} \leftarrow \mathbf{delV}_{\mathbb{T}^\sharp}[\mathtt{p}_0, \ldots \mathtt{p}_k](\mathbf{comp}^\sharp(\mathrm{t}^\sharp_{\mathrm{pars}}, \mathrm{t}^\sharp_{\mathtt{f}} *_{\mathbb{T}} \mathtt{Id}(\mathrm{h}^\sharp_{\mathrm{rem}})))$

Fig. 7. Interprocedural analysis: algorithm for the analysis of a procedure call.

Summary computation. The computation of a new context summary should take into account a context that is general enough to encompass both $\mathrm{h}^\sharp_{\mathtt{f}}$ and $\mathrm{h}^\sharp_{\mathrm{in},\mathtt{f}}$:

- the new abstract context is $\mathrm{h}^\sharp_{\mathtt{f}} \triangledown_{\mathbb{H}} \mathrm{h}^\sharp_{\mathrm{in},\mathtt{f}}$ using abstract state widening $\triangledown_{\mathbb{H}}$ [6];
- the new summary related to this abstract context is derived by analysis of the body of $\mathtt{f}$, thus by updating $\mathrm{t}^\sharp_{\mathtt{f}}$ with $[\![\mathtt{C}]\!]^\sharp_{\mathbb{T}}(\mathtt{Id}(\mathrm{h}^\sharp_{\mathtt{f}} \triangledown_{\mathbb{T}} \mathrm{h}^\sharp_{\mathrm{in},\mathtt{f}}))$.

Then, the context summary for $\mathtt{f}$ is updated with this new context summary.

Application. Once a new summary has been computed, by definition, it satisfies the inclusion $\mathrm{h}^\sharp_{\mathrm{in},\mathtt{f}} \sqsubseteq_{\mathbb{H}} \mathrm{h}^\sharp_{\mathtt{f}}$, thus it can be applied so as to compute an abstract transformation after the call to $\mathtt{f}$ as shown in Section 7.1.

The overall procedure call analysis algorithm is displayed in Figure 7. The case examined in this subsection corresponds to the case where the **if** branch at lines 5–6 is taken. We observe that it necessarily occurs whenever a procedure is analyzed for the first time, as context summaries are initially set to $(\bot, \bot)$. Moreover, we note that the application of the summary after its update is done as in Section 7.1.

The following result formalizes the soundness of Figure 7, under the assumption that there is no recursive call.

Theorem 3 (Soundness of the analysis of a procedure call using a context summary). *We consider the call to f with the abstract transformation $\mathrm{t}^\sharp_{\mathrm{pre}}$ as input, and the post-condition $\mathrm{t}^\sharp_{\mathrm{post}}$ returned by the algorithm of Figure 7. We denote by $(\mathrm{h}^\sharp_f, \mathrm{t}^\sharp_f)$ the context summary for f after the analysis of the call. We let $(\sigma_0, \sigma_1, \nu) \in \gamma_{\mathbb{T}}(\mathrm{t}^\sharp_{\mathrm{pre}})$, $\sigma'_1 \in [\![p_0 = x_0]\!]_{\mathcal{T}} \circ \ldots \circ [\![p_k = x_k]\!]_{\mathcal{T}} \circ \mathbf{newV}[p_0, \ldots p_k](\{\sigma_1\})$, $\sigma'_2 \in [\![\mathtt{C}]\!]_{\mathcal{T}}(\{\sigma'_1\})$, and $\sigma_2 \in \mathbf{delV}[p_0, \ldots, p_k](\{\sigma'_2\})$ (i.e., $\sigma_2 \in [\![f(x_0, \ldots, x_k)]\!]_{\mathcal{T}}(\{\sigma_1\})$). Then, the following property holds:*

$$(\sigma_0, \sigma_2, \nu) \in \gamma_{\mathbb{T}}(\mathrm{t}^\sharp_{\mathrm{post}}) \wedge (\sigma'_1, \nu) \in \gamma_{\mathbb{H}}(\mathrm{h}^\sharp_f) \wedge (\sigma'_1, \sigma'_2, \nu) \in \gamma_{\mathbb{T}}(\mathrm{t}^\sharp_f)$$

This result means that not only the transformation $\mathrm{t}^\sharp_{\mathrm{post}}$ over-approximates the state after the call, but also the new context summary accounts for this call. This

entails that context summaries account for all the calls that are encountered in the analysis of a complete interprocedural program. Moreover, Theorem 3 entails Equation 1 for procedure calls.

Example 10 (Context summary computation). In this example, we consider the function `append` again, but assume that the analysis starts with the $(\bot, \bot)$ context summary for it. We study the code `append(a, b); append(a, c);` where `a`, `b`, and `c` are initially lists of length 1. Then:

- the first call results in the update of the summary with a context summary $(\mathrm{h}^\sharp, \mathrm{t}^\sharp)$ where $\mathrm{h}^\sharp$ assumes that the first argument is a single list element, i.e., is of the form $\&\mathtt{l0} \mapsto \alpha_0 *_\mathtt{s} \alpha_0 \cdot \underline{\mathtt{n}} \mapsto \mathbf{0x0} *_\mathtt{s} \&\mathtt{l1} \mapsto \alpha_1 *_\mathtt{s} \ldots$;
- when the second call is encountered this first summary does not cover the second context (at this point, the argument `a` has length 2), thus a new context summary needs to be calculated; this new summary $(\mathrm{h}^\sharp, \mathrm{t}^\sharp)$ is such that $\mathrm{h}^\sharp$ only assumes that the first argument may be a list of any length (as derived by widening), i.e., it is of the form $\&\mathtt{l0} \mapsto \alpha_0 *_\mathtt{s} \mathbf{lseg}(\alpha_0, \alpha_1) *_\mathtt{s} \alpha_1 \cdot \underline{\mathtt{n}} \mapsto \mathbf{0x0} *_\mathtt{s} \&\mathtt{l1} \mapsto \alpha_1 *_\mathtt{s} \ldots$.

This last summary may still not be as general as the summary shown in Example 9, and may thus be generalized even more at subsequent call points.

Analysis of recursive calls. So far, we have focused on the case where there are no recursive calls. Actually, the presence of recursive calls changes relatively little to the principle of our analysis (although the formalization would be significantly heavier and is left out). Indeed, it only requires to extend the algorithm of Figure 7 with a fixpoint computation over context summaries, so as to determine an over-approximation of both components of the procedure context summary.

- when a recursive call is encountered and when the calling context is not accounted for in the current context summary (Figure 7, condition at line 4 evaluated to false), the $\mathrm{h}^\sharp_\mathtt{f}$ component should be widened and the current $\mathrm{t}^\sharp_\mathtt{f}$ should be used directly;
- at the end of the analysis of the procedure body, the $\mathrm{t}^\sharp_\mathtt{f}$ component should be widened with the previously known transformation, and the analysis of the procedure body iterated until this transformation stabilizes.

Convergence is ensured by widening both on abstract states and transformations.

8 Experimental evaluation

In this section, we report on the evaluation of the interprocedural analysis based on function summaries, with respect to the following questions:

1. Is it able to infer precise invariants?
2. Does it scale better than a classical call-string-based analysis?
3. How effective are context summaries, i.e., do they often have to be recomputed?

We have implemented the interprocedural analysis based on context summaries for a large fragment of the C language. Our tool is a plugin for Frama-C [20]. It supports conventional control flow structures of C and can be parameterized by the inductive definition of the structure to consider, as in [5]. However, it leaves

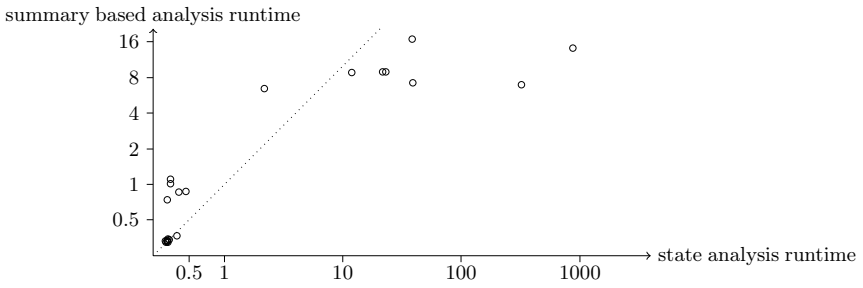

Fig. 8. Per-function comparison of analysis runtime (times in seconds)

out non-recursive structures and data-types that are not immediately related to the analysis principle (strings, arrays, numeric types). Furthermore, we have also implemented as another extension of Frama-C an analysis relying on the call-string approach, i.e., that inlines procedures at call sites.

Experiments and raw data. We did two experiments. First, we ran the two analyses on a fragment of the GNU Emacs 25.3 editor. This fragment comprises 22 functions of the `Fx_show_tip` feature and is implemented in C. It manipulates descriptions of Lisp objects including lists built as Cons pairs. The analyzed code corresponds to about 3 kLOC. Analyses were ran on an Intel Core i7 laptop at 2.3 GHz with 16Gb of RAM. The raw data are provided in Appendix A and the following paragraphs discuss the main points related to the above questions. Second, we analyzed a set of basic recursive functions on trees and lists to validate the recursive call analysis.

Precision. We compared the result of the analysis of the body of each procedure. More precisely, we checked whether the transformation computed for the whole procedure body for its entry abstract state (the first component of context summaries) is at least as precise as the post-condition produced by the state analysis. For 15 functions, the body contains nested calls and the result is as precise. The remaining 7 functions do not contain any call, thus are not relevant to validate the absence of precision loss at call sites.

Scalability. The total measured analysis time was 14.20s for the transformation-based analysis against 877.12s for the state analysis, which shows a high overall speedup. Secondly, we show the average analysis time of the body of each function in Figure 8 (these values are average analysis times, and not total time spent analyzing each function, as the effectiveness of summary reuse is the topic of the next paragraph). We observe that for some functions the speedup is low or even negative. These functions mostly correspond to low analysis times. Upon inspection, they all occupy a low position in the call tree: they call few functions, and the transformation analysis overhead is not compensated by a high gain from many summary applications to avoid the analysis of down calls. Conversely functions at the top of the call graph (such as the entry point) show a very high gain.

Effectiveness. Finally, we assessed the effectiveness of the summary reuse, which depends not only on the call graph shape (functions that are called at a single site offer no gain opportunity) but also on the function behavior and the analysis (depending on the contexts that are encountered, some procedure may need to be reanalyzed multiple times or not). We observed that only one procedure needed to be reanalyzed (`Fcons` was reanalyzed 3 times). All other summaries were computed only once (i.e., the branch at lines 4–6 in Figure 7 is taken only once). For functions called at a single point (11 of the total) summaries could not be reused, but for 8 functions, summaries were reused multiple times (3 to 44 times). By contrast, the state analysis had to reanalyze most functions several times: in particular 11 functions were reanalyzed 15 times or more (up to 296 times). Therefore, the summary-based analysis provides significant gain even for small functions.

Recursive calls. We ran the analysis on a series of recursive implementations of classical functions on lists and binary trees, including size, allocation, deallocation, insertion, search and deep copy, and also list concatenation and filter. For all these functions, the expected invariants could be inferred in under 5ms (Appendix A).

9 Related works and conclusion

Since Sharir and Pnueli's seminal paper [35], many works have studied approaches to interprocedural analyses. The relational approach strives for modularity, which has been a concern in static analysis even beyond interprocedural code [10,8]. A major advantage of modular approaches is to avoid the reanalysis of program fragments that are called repeatedly. However, it is generally difficult to apply as it requires an accurate and efficient representation for summaries. While relational numerical abstract domains [11] naturally provide a way to capture numerical input/output relations [26,27], this is harder to achieve when considering memory abstractions. The TVLA framework [33] supports such relation using the classical "primed variables" technique [19,18] where input and output variables are distinguished using prime symbols. Some pointer analyses such as [12] rely on specific classes of procedure summaries to enable modular analysis. However, separation logic [28] does not naturally support this since formulas describe regions of a given heap. The classical solution involves the tabulation of pairs of separation logic formulas [2,15,22], but this approach does not allow to relate the description of heap regions in a fine manner. To circumvent this, we use transformations introduced in [17], which are built on connectors inspired by separation logic. The advantage is twofold: it enables not only a more concise representation of transformations (since tables of pairs may need to grow large to precisely characterize the effect of procedures) but also a more local description of the relation between procedure input and output states. Our graph representation of abstract states and transformations opens the way to a resolution of the frame problem [29,30] using intersection operation. The results of our top-down, context summary-based analysis confirm that this approach brings a gain in analysis scalability once the upfront cost of summaries is amortized.

Many pointer analyses and weak forms of shape analyses have introduced specific techniques in order to construct and compute procedure summaries [21,25,23]. These works typically rely on some notion of graph that describes both knowledge about memory entities and procedure calls, therefore the interprocedural analysis reduces to graph algorithms. Moreover, context sensitivity information may be embedded into these graphs. Our approach differs in that it relies on a specific algebra of summaries, although we may also augment our summaries with context information. Another difference is that the summary predicates our abstract domain is based on allow a very high level of precision and that the abstract procedure call analysis algorithm (with intersection and composition) aims at preserving this precision. We believe that two interesting avenues for future works would consider the combination of various levels of context sensitivity and of less less expressive summaries with our analysis framework.

A very wide range of techniques have been developed to better cope with interprocedural analysis, many of which could be combined with context summaries. First, we do not consider tabulation of procedure summaries [10], however, we could introduce this technique together with some amount of context sensitivity [1]. Indeed, while relations reduce the need for tables of abstract pre- and post-conditions, combining context summaries and finer context abstraction may result in increased precision [7]. Bi-abduction [3] has been proposed as a technique to infer relevant abstract pre-conditions of procedures. In [3], this process was implemented on a state abstract domain, but the core principle of the technique is orthogonal to that choice, thus bi-abduction could also be applied to abstract transformations. Moreover, while our analysis proceeds top-down, it would be interesting to consider the combination with a bottom-up inference of summaries for some procedures [4]. Last, many works have considered the abstraction of the stack-frame and of the relations between the stack frame structure and the heap structures manipulated by procedures [31,32]. The notion of cut-points has been proposed in [29,30] to describe structures tightly intertwined with the stack. An advantage of our technique is that the use of an abstraction based on transformations which can express that a region of the heap is preserved reduces the need to reason over cutpoints.

Acknowledgments. We would like to thank the anonymous reviewers for their suggestions that helped greatly improve the quality of this paper. This work has received support from the French ANR as part of the VeriAMOS grant.

References

1. François Bourdoncle. Abstract interpretation by dynamic partitioning. *Journal of Functional Programming*, 2(4):407–423, 1992.
2. Cristiano Calcagno, Dino Distefano, Peter O'Hearn, and Hongseok Yang. Footprint analysis : A shape analysis that discovers preconditions. In *Static Analysis Symposium (SAS)*, pages 402–418. Springer, 2007.
3. Cristiano Calcagno, Dino Distefano, Peter O'Hearn, and Hongseok Yang. Compositional shape analysis by means of bi-abduction. In *Symposium on Principles of Programming Languages (POPL)*, pages 289–300. ACM, 2009.

4. Ghila Castelnuovo, Mayur Naik, Noam Rinetzky, Mooly Sagiv, and Hongseok Yang. Modularity in lattices: A case study on the correspondence between top-down and bottom-up analysis. In *Static Analysis Symposium (SAS)*, pages 252–274. Springer, 2015.
5. Bor-Yuh Evan Chang and Xavier Rival. Relational inductive shape analysis. In *Symposium on Principles of Programming Languages (POPL)*, pages 247–260. ACM, 2008.
6. Bor-Yuh Evan Chang, Xavier Rival, and George Necula. Shape analysis with structural invariant checkers. In *Static Analysis Symposium (SAS)*, pages 384–401. Springer, 2007.
7. Ramkrishna Chatterjee, Barbara G Ryder, and William A Landi. Relevant context inference. In *Symposium on Principles of Programming Languages (POPL)*, pages 133–146. ACM, 1999.
8. Michael Codish, Saumya K. Debray, and Roberto Giacobazzi. Compositional analysis of modular logic programs. In *Symposium on Principles of Programming Languages (POPL)*, pages 451–464. ACM, 1993.
9. Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Symposium on Principles of Programming Languages (POPL)*. ACM, 1977.
10. Patrick Cousot and Radhia Cousot. Modular static program analysis. In *Conference on Compiler Construction (CC)*, pages 159–179. Springer, 2002.
11. Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Symposium on Principles of Programming Languages (POPL)*, pages 84–97. ACM, 1978.
12. Isil Dillig, Thomas Dillig, Alex Aiken, and Mooly Sagiv. Precise and compact modular procedure summaries for heap manipulating programs. In Mary W. Hall and David A. Padua, editors, *Conference on Programming Languages Design and Implementation (PLDI)*, pages 567–577. ACM, 2011.
13. Dino Distefano, Peter O'Hearn, and Hongseok Yang. A local shape analysis based on separation logic. In *Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 287–302. Springer, 2006.
14. Kamil Dudka, Petr Peringer, and Tomáš Vojnar. Predator: A practical tool for checking manipulation of dynamic data structures using separation logic. In *Conference on Computer Aided Verification (CAV)*, pages 372–378. Springer, 2011.
15. Bhargav S Gulavani, Supratik Chakraborty, Ganesan Ramalingam, and Aditya V Nori. Bottom-up shape analysis. In *Static Analysis Symposium (SAS)*, pages 188–204. Springer, 2009.
16. Hugo Illous. *Abstract Heap Relations for a Compositional Shape Analysis*. PhD thesis, École Normale Supérieure, 2018.
17. Hugo Illous, Matthieu Lemerre, and Xavier Rival. A relational shape abstract domain. In Clark W. Barrett, Misty Davies, and Temesghen Kahsai, editors, *NASA Formal Methods International Symposium*, volume 10227 of *Lecture Notes in Computer Science*, pages 212–229. Springer, 2017.
18. Bertrand Jeannet, Alexey Loginov, Thomas Reps, and Mooly Sagiv. A relational approach to interprocedural shape analysis. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 32(2):5, 2010.
19. Bertrand Jeannet, Alexey Loginov, Thomas W. Reps, and Shmuel Sagiv. A relational approach to interprocedural shape analysis. In *Static Analysis Symposium (SAS)*, pages 246–264, 2004.

20. Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-c: A software analysis perspective. *Formal Aspects of Computing*, 27(3):573–609, 2015.
21. Chris Lattner, Andrew Lenharth, and Vikram S. Adve. Making context-sensitive points-to analysis with heap cloning practical for the real world. In Jeanne Ferrante and Kathryn S. McKinley, editors, *Conference on Programming Languages Design and Implementation (PLDI)*, pages 278–289. ACM, 2007.
22. Quang Loc Le, Cristian Gherghina, Shengchao Qin, and Wei-Ngan Chin. Shape analysis via second-order bi-abduction. In *Conference on Computer Aided Verification (CAV)*, pages 52–68. Springer, 2014.
23. Yuxiang Lei and Yulei Sui. Fast and precise handling of positive weight cycles for field-sensitive pointer analysis. In Bor-Yuh Evan Chang, editor, *Static Analysis Symposium (SAS)*, volume 11822 of *Lecture Notes in Computer Science*, pages 27–47. Springer, 2019.
24. Zohar Manna and Amir Pnueli. Axiomatic approach to total correctness of programs. *Acta Informatica*, 3:243–263, 1974.
25. Mark Marron, Manuel V. Hermenegildo, Deepak Kapur, and Darko Stefanovic. Efficient context-sensitive shape analysis with graph based heap models. In Laurie J. Hendren, editor, *Conference on Compiler Construction (CC)*, volume 4959 of *Lecture Notes in Computer Science*, pages 245–259. Springer, 2008.
26. Markus Müller-Olm and Helmut Seidl. Precise interprocedural analysis through linear algebra. In *Symposium on Principles of Programming Languages (POPL)*, pages 330–341. ACM, 2004.
27. Corneliu Popeea and Wei-Ngan Chin. Inferring disjunctive postconditions. In *ASIAN Computing Sciences Conference*, pages 331–345. Springer, 2006.
28. John Reynolds. Separation logic: A logic for shared mutable data structures. In *Symposium on Logics In Computer Science (LICS)*, pages 55–74. IEEE, 2002.
29. Noam Rinetzky, Jörg Bauer, Thomas W. Reps, Shmuel Sagiv, and Reinhard Wilhelm. A semantics for procedure local heaps and its abstractions. In *Symposium on Principles of Programming Languages (POPL)*, pages 296–309, 2005.
30. Noam Rinetzky, Mooly Sagiv, and Eran Yahav. Interprocedural shape analysis for cutpoint-free programs. In *Static Analysis Symposium (SAS)*, pages 284–302, 2005.
31. Noam Rinetzky and Shmuel Sagiv. Interprocedural shape analysis for recursive programs. In *Conference on Compiler Construction (CC)*, pages 133–149. Springer, 2001.
32. Xavier Rival and Bor-Yuh Evan Chang. Calling context abstraction with shapes. In *Symposium on Principles of Programming Languages (POPL)*, pages 173–186. ACM, 2011.
33. Mooly Sagiv, Thomas Reps, and Reinhard Whilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 20(1):50, 1998.
34. Ina Schaefer and Andreas Podelski. Local reasoning for termination. In *COSMICAH 2005: workshop on verification of COncurrent Systems with dynaMIC Allocated Heaps*, pages 16–30, 2005.
35. Micha Sharir and Amir Pnueli. Two approaches to interprocedural data flow analysis. In *Program Flow Analysis: Theory and Applications*, chapter 7. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1981.

A Raw experimental data

Analysis of `Fx_show_tip`. The table below lists the per call average analysis times of each procedure, whether the summary analysis is as precise as the state analysis ("as precise" means the results are at least as precise; "irrel." means the measure is irrelevant as the function body does not contain any call) and the depth in the call graph.

Function name	State time (s)	Summary time (s)	Precision	Depth
`Fcons`	0.33	0.34	irrel.	8
`list2`	0.32	0.32	as precise	7
`list4`	0.33	0.32	as precise	7
`Fassq`	2.16	6.46	irrel.	6
`Fcar`	0.32	0.33	irrel.	3
`Fcdr`	0.33	0.33	irrel.	2
`Fnthcdr`	0.33	0.34	irrel.	3
`Fnth`	0.34	0.34	as precise	2
`make_monitor_attribute_list`	0.33	0.74	as precise	6
`check_x_display_info`	0.33	0.33	irrel.	3
`Fx_display_monitor_attributes_list`	0.41	0.86	as precise	2
`x_get_monitor_for_frame`	0.40	0.37	irrel.	6
`x_make_monitor_attribute_list`	0.47	0.87	as precise	5
`x_get_monitor_attributes_fallback`	0.35	1.01	as precise	4
`x_get_monitor_attributes`	0.35	1.11	as precise	3
`x_get_arg`	11.82	8.76	as precise	5
`x_frame_get_arg`	21.75	8.87	as precise	4
`x_default_parameter`	23.00	8.91	as precise	3
`compute_tip_xy`	38.24	16.80	as precise	1
`x_default_font_parameter`	39.06	7.17	as precise	2
`x_create_tip_frame`	321.77	6.96	as precise	1
`Fx_show_tip`	877.12	14.20	as precise	0

The table below lists the number of times the body of a procedure is reanalyzed:

- column #state counts reanalyses by the state analysis;
- column #total counts the number of times a call site to this procedure is encountered by the summary analysis;
- column #recomp counts the number of times the summary based analysis needs to reanalyze the body of this procedure to come up with more general summary;
- column #reuse counts the number of times a summary is reused without recomputation.

Function name	#state	#total	#recomp	#reuse
`Fcons`	296	47	3	44
`list2`	12	2	1	1
`list4`	24	4	1	3
`Fassq`	64	16	1	15
`Fcar`	24	1	1	0
`Fcdr`	12	4	1	3
`Fnthcdr`	24	1	1	0
`Fnth`	24	8	1	7
`make_monitor_attribute_list`	6	2	1	1
`check_x_display_info`	3	1	1	0
`Fx_display_monitor_attributes_list`	3	1	1	0
`x_get_monitor_for_frame`	6	2	1	1
`x_make_monitor_attribute_list`	3	1	1	0
`x_get_monitor_attributes_fallback`	3	1	1	0
`x_get_monitor_attributes`	3	1	1	0
`x_get_arg`	19	5	1	4
`x_frame_get_arg`	15	1	1	0
`x_default_parameter`	15	15	1	14
`compute_tip_xy`	3	3	1	2
`x_default_font_parameter`	1	1	1	0
`x_create_tip_frame`	1	1	1	0
`Fx_show_tip`	1	1	1	0

Analysis of recursive list and tree classical algorithms. The table below lists the analysis times of a series of classical functions over lists and trees.

Structure	function	time (ms)
List	`length`	1.256
List	`get_n`	2.179
List	`alloc`	1.139
List	`dealloc`	0.842
List	`concat`	1.833
List	`map`	0.904
List	`deep_copy`	1.540
List	`filter`	3.357
Tree	`visit`	1.078
Tree	`size`	1.951
Tree	`search`	3.818
Tree	`dealloc`	1.391
Tree	`insert`	5.083
Tree	`deep_copy`	2.603