Local Shape Analysis for Overlaid Data Structures[⋆]

Cezara Drăgoi[1], Constantin Enea[2], and Mihaela Sighireanu[2]

[1] IST Austria, cezarad@ist.ac.at
[2] Univ Paris Diderot, Sorbonne Paris Cite, LIAFA CNRS UMR 7089, Paris,
{cenea,sighirea}@liafa.univ-paris-diderot.fr

Abstract. We present a shape analysis for programs that manipulate overlaid data structures which share sets of objects. The abstract domain contains Separation Logic formulas that (1) combine a per-object separating conjunction with a per-field separating conjunction and (2) constrain a set of variables interpreted as sets of objects. The definition of the abstract domain operators is based on a notion of homomorphism between formulas, viewed as graphs, used recently to define optimal decision procedures for fragments of the Separation Logic. Based on a Frame Rule that supports the two versions of the separating conjunction, the analysis is able to reason in a modular manner about non-overlaid data structures and then, compose information only at a few program points, e.g., procedure returns. We have implemented this analysis in a prototype tool and applied it on several interesting case studies that manipulate overlaid and nested linked lists.

1 Introduction

Automatic synthesis of valid assertions about heap-manipulating programs, such as loop invariants or procedure summaries, is an important and highly challenging problem. In this paper, we address this problem for sequential programs manipulating overlaid and nested linked lists. The term overlaid refers to the fact that the lists share some set of objets. Such data structures are often used in low-level code in order to organize a set of objects with respect to different criteria. For example, the network monitoring software Nagios (www.nagios.com) groups sets of tasks in nested lists, according to the user that spawned them, but also in two lists of pending and, respectively, executed tasks. These structures are overlaid because they share the objects that represent tasks.

We propose an analysis based on abstract interpretation [10], where the elements of the abstract domain are formulas in *NOLL* [13], a fragment of Separation Logic (SL) [17]. The main features of *NOLL* are (1) two separating conjunction operators, the *per-object* separation $*$ and the *per-field* separation $*_w$, (2) recursive predicates indexed by *set variables*, which are interpreted as the set of all heap objects in the data structure described by the predicate, and (3) constraints over set variables, which relate sets of objects that form different data structures. The analysis has as parameter the set of recursive predicates used in the *NOLL* formulas. Although per-object separation can be expressed using per-field separation and constraints on set variables, we prefer to keep both versions for two reasons: (i) the formulas are more concise and (ii) as a design principle, the analysis should introduce per-field separation only when it is necessary, i.e., when it detects overlaid data structures.

[⋆] This work was supported in part by the Austrian Science Fund NFN RiSE, by the ERC Advanced Grant QUAREM, and by the French ANR Project Veridyc.

The main characteristics of the analysis are (1) *compositionality*: we define a frame rule for *NOLL*, which allows to reason locally, on a subset of allocated objects and considering only a subset of their fields and (2) abstract domain operators based on *graph homomorphism*, used recently in optimal decision procedures for SL fragments [9, 13].

The frame rule for SL with only per-object separation [17] states that, in order to compute the effect of a program P on the input specified by a formula ϕ, one has to split ϕ into $\phi_i * \sigma$, where ϕ_i describes all the heap objects reachable from program variables read in P without being written before, compute the post-condition ϕ_o of P on ϕ_i and then, infer that the effect of P on ϕ is $\phi_o * \sigma$. Programs with overlaid data structures can be usually partitioned in blocks, e.g. procedures, that manipulate just one non-overlaid data structure at a time. Thus, for the sake of compositionality, only the description of this data structure should be considered when computing the effect of some block. Having both the per-field and the per-object separation, the frame rule we define refines the decomposition of ϕ into $(\phi_i *_w \sigma_1) * \sigma_2$, where ϕ_i describes *the list segments in the heap built with fields accessed in P*, which start in variables read in P. As before, if ϕ_o represents the effect of P on ϕ_i then the post-condition of P on ϕ is $(\phi_o *_w \sigma_1) * \sigma_2$.

The constraints on set variables are important to define a precise local analysis. They are used to relate heap regions accessed in different blocks of the program. For example, consider a program that traverses a list segment L_1 and then, another list segment L_2, these list segments being overlaid. In a local analysis that considers only one list segment at a time, the constraints on set variables are used to preserve the fact that some heap objects, materialized on the list segment L_1, belong also to L_2. This information may be used when fields of these heap objects are accessed.

The elements of the abstract domain are existentially-quantified disjunctions of *NOLL* formulas that use only (separating) conjunctions. To obtain efficient abstract domain operators, we use a graph representation for *NOLL* formulas. Each disjunction-free formula φ is represented by a graph where nodes correspond to variables of φ and edges correspond to atoms of φ that describe (nested) list segments.

The definition of the order relation $\preceq$ between the abstract domain values uses the entailment relation $\models$ between *NOLL* formulas. One can prove [9, 13] that $\varphi_1 \models \varphi_2$ whenever there exists an *homomorphism* from the graph representations of φ_2 to the one of φ_1. Assuming that atoms describe only singly-linked lists, an homomorphism from a graph G_1 to a graph G_2 maps edges of G_1 to (possibly empty) paths of G_2 such that the paths of G_2 associated to two distinct edges of G_1 do not share edges. If atoms include recursive predicates that describe nested list segments, the homomorphism maps edges of G_1 to more general sub-graphs of G_2 that represent unfoldings of these predicates. Comparing to the previous approaches for proving entailments of SL formulas, which are based on inference rules, the homomorphism approach has the same precision but it is more efficient because, intuitively, it defines also a strategy for applying the inference rules. We introduce an effective procedure for checking graph homomorphism, which is based on testing membership in languages described by tree automata.

The widening operator ∇ is based on two operations: (1) a fold operator that "recognizes" data structures used in the program, if they are describable by one of the predicates parametrizing the analysis and (2) a procedure that uses graph homomorphism in order to identify the constraints which are true in both of its arguments (implicitly, it tries to preserve the predicates discovered by fold). More precisely, given two graphs G_1 and G_2, the widening operator searches for a maximal sub-graph of G_1 which is homomorphic to a sub-graph of G_2 (and thus weaker) and a maximal sub-graph of G_2 which is homomorphic to a sub-graph of G_1. All these graphs should be disjoint. If all edges of G_1 and G_2 are included in these sub-graphs then the widening returns the union of the two weaker sub-graphs. Otherwise, it returns a disjunction of two graphs or, if the number of nodes which correspond to existential variables is greater than some fixed bound, it applies the operator fold which replaces unfoldings of recursive predicates by instantiations of these predicates. Folding the same set of nodes in two different list segments introduces the per-field separation although the initial formula may use only the per-object separation.

The analysis is implemented in a prototype tool that has been successfully applied on some interesting set of benchmarks that includes fragments from Nagios.

Related work: There are many works that develop static analyses based on SL, e.g., [2, 7, 6, 8, 11, 12, 14, 16, 19, 20]. Most of them, except the work in [16], are not precise enough in order to deal with overlaid data structures. The abstract domain operators defined in [7, 20] can be seen as instantiations of the operators based on graph homomorphism introduced in this paper (provided that the definition of the graph homomorphism is adapted to the respective logics). In [16], overlaid data structures are described using the classical conjunction, instead of the per-field separation as in our work. The analysis is defined as a reduced product of several sub-analyses, where each sub-analysis "observes" a different set of fields. The reduction operator, used to exchange information between the sub-analyses, is called at control points, which are determined using a preliminary data-flow analysis. The same data-flow analysis is used to anticipate the updates on the set variables. In our work, compositionality is achieved using the frame rule and thus, it avoids the overhead of duplicate domain operations and calls to the reduction operator. Moreover, the updates on the set variables are determined during the analysis and thus, they can produce more precise results. Static analyses for reasoning about the size of memory regions are introduced in [15]. They are based on combining a *set domain*, that partitions the memory into (not necessarily) disjoint regions, and a numerical domain, that is used to relate the cardinalities of the memory regions. The abstract domain defined in this paper can be seen as an instance of a set domain.

2 Overview

Our running example is extracted from the network monitoring software Nagios which uses a task manager to store pending and executed tasks, grouped or linked according to different criteria. The implementation given in Fig. 1 wraps tasks in objects of type `Task` composed of a field `op`, which stores a description of the task, a field `succ`, which links the tasks spawned by the same user, and fields `next` and `prev`, which link pending or executed tasks. A task manager is implemented by an object of type `Manager` containing a field `tab`, which stores the tasks of each user using a `NestedList` object, a field `todo` which is used to access the pending tasks, and a field `log`, which points to the list of executed tasks. Each element of type `NestedList` has an integer field encoding the user id and a field `tasks` pointing to the list of tasks spawned by the user. To specify the lists

```
typedef struct Task {
  char* op;
  struct Task* succ;
  struct Task* prev, *next;
} Task;

typedef struct NestedList {
  int user;
  struct NestedList* nextu;
  Task* tasks;
} NestedList;

typedef struct Manager {
  NestedList* tab;
  Task* todo;
  Task* log;
} Manager;

Task* lookup(int user,char* str,
             NestedList* tab)
{ ... return ret; }
```

```
Task* add(Task* x,Task* log)
{ x->prev = NULL;
  x->next = log;
  return x;
}
Task* cut(Task* x,Task* todo)
{
  Task* tmp = x->next;
  if (tmp != NULL)      tmp->prev = x->prev;
  if (x->prev == NULL)  return tmp;
  else
  { x->prev->next = tmp;
    return todo; }
}
void execute(int user,char* str,Manager* man)
{
  Task* x = lookup(user,str,man->tab);
  if ( x != NULL && (x->prev != NULL || x==man->todo) )
  { man->todo = cut(x,man->todo);
    man->log = add(x,man->log); }
}
```

Fig. 1: Task manager

pointed to by the fields of an object of type Manager, we use the following SL formula:

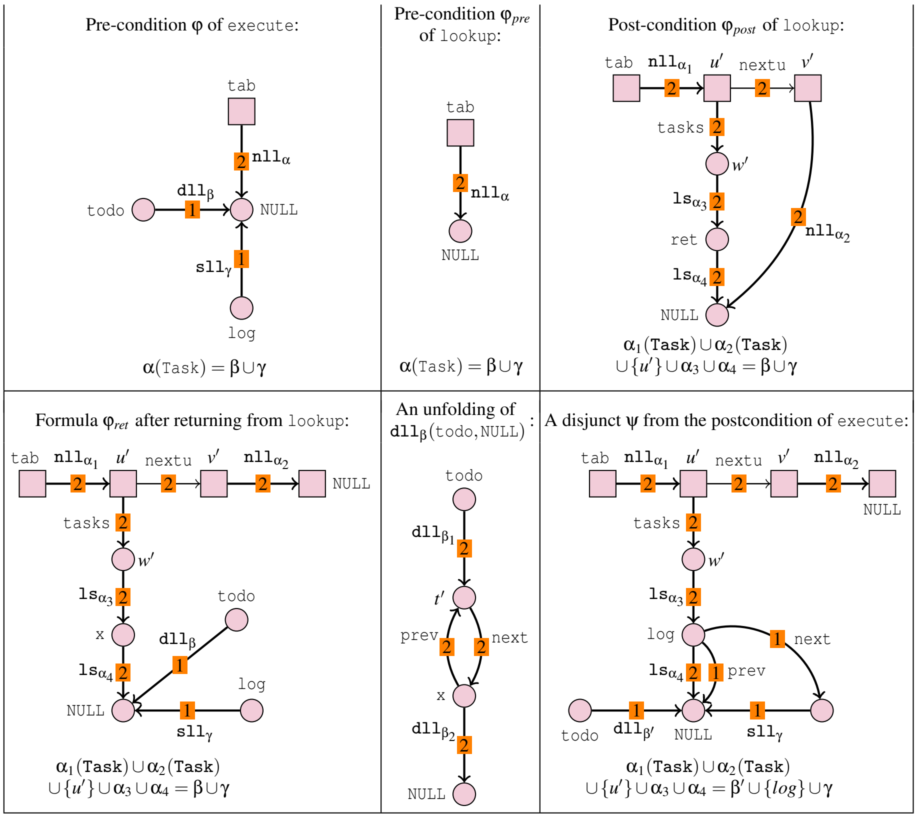
$$\varphi \triangleq \mathtt{nll}_\alpha(\mathtt{tab},\mathtt{NULL},\mathtt{NULL}) *_w \big(\mathtt{dll}_\beta(\mathtt{todo},\mathtt{NULL}) * \mathtt{sll}_\gamma(\mathtt{log},\mathtt{NULL},\mathtt{NULL})\big) \\ \wedge\ \alpha(\mathtt{Task}) = \beta \cup \gamma, \tag{1}$$

where $\mathtt{nll}_\alpha(\mathtt{tab},\mathtt{NULL},\mathtt{NULL})$ describes a list of lists pointed to by tab and ending in NULL, where all the inner lists end also in NULL, $\mathtt{dll}_\beta(\mathtt{todo},\mathtt{NULL})$ describes[3] a doubly-linked list from todo to NULL, and $\mathtt{sll}_\gamma(\mathtt{log},\mathtt{NULL},\mathtt{NULL})$ describes a list of objects with two fields next and prev such that prev points always to NULL (this is a common way to factorize one type declaration in order to represent both doubly-linked and singly-linked lists). In general, the list segments described by these predicates can be empty. (For a formal definition of these predicates, we defer the reader to Sec. 4.) The set variables α, β, and γ are interpreted as the set of heap objects in the list segments described by the corresponding predicates. Also, $\alpha(\mathtt{Task})$ is interpreted as the set of heap objects of type Task in the interpretation of α. To simplify the notation, we represent an object of type Manager by three pointer variables tab, todo, and log. The per-field separation $*_w$ allows the list of lists to share objects with the other two lists.

Such formulas have an equivalent graph representation, which is more intuitive and easier to work with. For example, Fig. 2 shows the graph representation of φ. In $\mathtt{nll}_\alpha(\mathtt{tab},\mathtt{NULL},\mathtt{NULL})$, the first two arguments represent the start and, resp., the end of the list of NestedList objects. Therefore, this atom is represented by an edge from tab to NULL labeled by $\mathtt{nll}_\alpha$ (actually, the edge label contains also the third argument NULL but we have omit it for simplicity). Any two edges labeled by the same integer (resp., different integers) represent per-object (resp., per-field) separated atoms.

We define an abstract domain, denoted $\mathcal{ASL}$, which is parametrized by a set of (recursive) predicates as above and contains disjunctions of formulas as in (1).

[3] This predicate is actually a shorthand for the formula $\mathtt{todo} \mapsto \{(\mathtt{prev},\mathtt{NULL})\} *_w \mathtt{dll}_\beta(\mathtt{todo},u') *_w u' \mapsto \{(\mathtt{next},\mathtt{NULL})\}$, where $\mathtt{dll}_\beta$ is the recursive predicate defined in Ex. 1, page 9, and u' an existential variable.



Pre-condition φ of `execute`:

Pre-condition φ_{pre} of `lookup`:

Post-condition φ_{post} of `lookup`:

Formula φ_{ret} after returning from `lookup`:

An unfolding of $\mathtt{dll}_\beta(\mathtt{todo},\mathtt{NULL})$:

A disjunct ψ from the postcondition of `execute`:

Fig. 2: Graph representations of formulas in the analysis of `execute` (the square nodes have type `NestedList` and the circle nodes have type `Task`)

Next, we focus on the analysis of the procedure `execute`, which moves a task from the list of pending tasks, pointed to by `todo`, to the list of executed tasks, pointed to by `log`. To check that the task pointed to by x belongs to the `todo` list, it tests if `x` equals the head of this list or if the `prev` field is not `NULL`. Given the precondition φ in (1), the analysis proves that, at the end of the procedure, the property φ remains true, i.e., all the data structures are preserved.

The procedure starts by calling `lookup` in order to search for an object of type `Task`. The analysis we define is compositional in two ways. First, each procedure is analyzed on its "local heap", i.e., the heap region reachable from the actual parameters of the call. Second, we restrict the local heap to paths that use only fields accessed by the procedure. For example, the procedure `lookup` accesses only the fields `nextu`, `tasks`, and `succ` and consequently, it is analyzed on a sub-formula φ_{pre} of φ that contains only $\mathtt{nll}_\alpha(\mathtt{tab},\mathtt{NULL},\mathtt{NULL}) \wedge \alpha(\mathtt{Task}) = \beta \cup \gamma$. The constraint on set variables is included because it constrains a set of objects in the local heap of `lookup`. The graph representation of φ_{pre} is given in Fig. 2.

The post-condition of `lookup` computed by the analysis contains several disjuncts; one of them, denoted φ_{post}, is given in Fig. 2. This graph represents an unfolding of the list segment described by $\texttt{nll}_\alpha(\texttt{tab},\texttt{NULL},\texttt{NULL})$, where u', v', and w' are existentially quantified variables (by convention, all the existential variables are primed). Edges labeled by fields, e.g., the edge from u' to v' labeled by `nextu`, represent values of fields. The term $\{u'\}$ in the constraint on set variables is interpreted as the singleton containing the object u'. The output parameter `ret` points to an object in some inner list of the nested list segment pointed to by `tab`. The inner lists are described by the predicate `ls`.

The abstract value reached when returning from the call to `lookup`, φ_{ret}, is given in Fig. 2. It is obtained from φ by replacing the sub-formula φ_{pre} with φ_{post} (we consider two copies of the node labeled by `NULL` for readability).

We now consider the `if` statement in `execute`. The abstract element where we assume that `x!=NULL` is true is obtained from φ_{post} by adding the constraint $\texttt{x} \neq \texttt{NULL} \wedge \texttt{x} \in \alpha_4$ (the object pointed to by x belongs to α_4 only if $x \neq \texttt{NULL}$; otherwise, the interpretation of α_4 is $\emptyset$). To compute the abstract element where we assume that `x->prev != NULL`, we need to materialize the `prev` field of `x`. For this, we use the fact that the set constraints imply that $x \in \beta \cup \gamma$ and we compute a disjunction of three graphs where we unfold either the predicate $\texttt{dll}_\beta(\texttt{todo},\texttt{NULL})$ (with `x` as the first element or as some element different from the first one) or $\texttt{sll}_\gamma(\texttt{log},\texttt{NULL},\texttt{NULL})$ (with `x` as some arbitrary element). Only one graph satisfies `x->prev != NULL` and this graph contains the unfolding of $\texttt{dll}_\beta(\texttt{todo},\texttt{NULL})$ given in the second row of Fig. 2. With this unfolding, the set variable β is replaced by $\beta_1 \cup \{t'\} \cup \beta_2$. Note that the node labeled by `x` in this unfolding is the same as the node labeled by `x` in the unfolding of $\texttt{nll}_\alpha(\texttt{tab},\texttt{NULL},\texttt{NULL})$.

If we continue on the branch of the `if` statement where `x!=NULL` and `x->prev!=NULL` are true then, we analyze the call to `cut` starting from a precondition that contains only the sub-formula describing the unfolding of $\texttt{dll}_\beta(\texttt{todo},\texttt{NULL})$ and the constraint on the set variables. This is because `cut` accesses only the fields `prev` and `next`, and the list $\texttt{sll}_\gamma(\texttt{log},\texttt{NULL},\texttt{NULL})$ is per-object separated from the doubly-linked list. A similar analysis is done for the call to `add`. One of the disjuncts from the post-condition of `execute`, denoted by ψ, is given in Fig. 2 (local variables have been projected out and, for simplicity, we abstract the unfolding of $\texttt{dll}_{\beta'}$).

The analysis proves that the data structures are preserved by a call to `execute`, i.e., its postcondition implies φ in (1). This is because there exists an homomorphism from φ to every disjunct in the post-condition. Intuitively, the homomorphism maps nodes of φ to nodes of the disjunct, labeled by at least the same set of program variables, and edges e of φ to sub-graphs of the disjunct, that represent unfoldings of the predicate labeling e. For example, this is the case for the disjunct ψ in Fig. 2. Concerning the constraints on set variables, the edge mapping defines a substitution Γ for set variables of φ to terms over variables of ψ, e.g., α is substituted by the union of all set variables in the unfolding of $\texttt{nll}_\alpha(\texttt{tab},\texttt{NULL},\texttt{NULL})$, i.e., $\alpha_1 \cup \alpha_2 \cup \{u'\} \cup \alpha_3 \cup \alpha_4$ ($\{u'\}$ is also considered because some field of u' is explicit in this unfolding). If Λ_1 and Λ_2 are the constraints over set variables in ψ, resp., φ, then Λ_1 implies $\Lambda_2[\Gamma]$.

3 Programs

We consider strongly typed imperative programs. The types used in the program are references to record types belonging to some set $\mathcal{T}$. A record type contains a set of fields, each field being a reference to a record type. We suppose that each field has a unique name and we denote by *Flds* the set of field names. Let τ be a typing function, that maps each variable into a type in $\mathcal{T}$ and each field into a function type over $\mathcal{T}$.

Program configurations: We use a classical storage model, where a *program configuration* is a pair $C = (S,H)$, where S represents the stack of program variables and H represents the heap of dynamically allocated objects. To give a formal definition, we consider three additional countable sets which are pairwise disjoint and disjoint from *Flds*: a set *Loc* of addresses (called also *locations*), a set *Vars* of program variables x,y,z, and a set *Vars*$'$ of "primed" variables x',y',z' that do not appear in the program but only in assertions where they are implicitly existentially quantified. We assume that all these elements are typed by τ to records in $\mathcal{T}$. For simplicity, we also assume that `NULL` is an element of *Vars* mapped always to a distinguished location $\sharp \in Loc$. Then,

$$S \in \texttt{Stacks} = [(Vars \cup Vars') \to Loc] \qquad\qquad H \in \texttt{Heaps} = [Loc \times Flds \rightharpoonup Loc]$$
$$C \in \texttt{Configs} = \texttt{Stacks} \times \texttt{Heaps}$$

We consider that S and H are *well typed*, e.g., if $S(x) = \ell$ then $\tau(x) = \tau(\ell)$. For simplicity, the constant `NULL` and the location $\sharp$ are typed by τ in any record in $\mathcal{T}$.

The set of locations l for which $H(l, f)$ is defined, for some f, is called the set of locations in C, and it is denoted by $Loc(C)$. The component S (resp. H) of a heap C is denoted by S^C (resp. H^C).

Programs: Aside the definition of record types, programs are collections of procedures. The procedures are written in a classical imperative programming language that contains memory allocation/deallocation statements (`new`/`free`), field updates (x->$f :=$ $\ldots$), variable assignments ($x := y$/$x := y$->f), call statements (`call Proc`$(\vec{x})$), and composed statements like sequential composition ;, `if-then-else`, and `while` loops. The formal meanings of the *basic statements* (not containing ;, conditionals, loops, and procedure calls) are given in terms of functions from $2^{\texttt{Configs}}$ to $2^{\texttt{Configs}}$, where `Configs` contains a special value $C_\perp$ that corresponds to a memory fault.

4 Assertion Language

The language we consider for writing program assertions, that describe sets of program configurations, is the logic *NOLL* [13] enriched with existential quantifiers and disjunction (to simplify the notation, we use primed variables instead of existential quantifiers).

Syntax: The logic *NOLL* is a multi-sorted fragment of Separation Logic [17]. It is defined over two sets of variables $LVars = Vars \cup Vars'$ and *SetVars*, called *location variables* and *set variables*, respectively. We assume that the typing function τ associates a sort, resp., a set of sorts, to every variable in *LVars*, resp., *SetVars*. A variable in *LVars* is interpreted as a location in *Loc* while a variable in *SetVars* is interpreted as a set of locations in *Loc*. The syntax of *NOLL* is given in Fig. 3.

The atoms of *NOLL* are either (1) *pure*, i.e., (dis)equalities between location variables, (2) *spatial*, i.e., the predicate *emp* denoting the empty heap, points-to constraints $E \mapsto \{(f_1,E_1);\ldots;(f_k,E_k)\}$, saying that the value stored by the field f_i of E equals E_i, for any $1 \le i \le k$, or predicate applications $P_\alpha(\vec{E})$, or (3) *sharing*, i.e., membership and

$E, F, E_i \in \mathit{LVars}$ location variables $\qquad \vec{E} \in \mathit{LVars}^+$ tuple of location variables
$f, f_i \in \mathit{Flds}$ field names $\qquad \alpha \in \mathit{SetVars}$ set variable
$R \in \mathcal{T}$ sort $\qquad P \in \mathcal{P}$ list segment predicate

$\Phi ::= \Pi \wedge \Sigma \wedge \Lambda \mid \Phi \vee \Phi$ *NOLL* formulas
$\Pi ::= E = F \mid E \neq F \mid \Pi \wedge \Pi$ pure formula
$\Sigma ::= \mathit{true} \mid \mathit{emp} \mid E \mapsto \{(f_1, E_1), \ldots, (f_k, E_k)\} \mid P_\alpha(\vec{E}) \mid \Sigma * \Sigma \mid \Sigma *_w \Sigma$ spatial formula
$\Lambda ::= E \in t \mid E \notin t \mid t = t' \mid t \cap t' = \emptyset \mid \Lambda \wedge \Lambda$ sharing formula
$t ::= \{E\} \mid \alpha \mid \alpha(R) \mid t \cup t'$ set terms

Fig. 3: Syntax of *NOLL* formulas

inclusion constraints over set terms. The predicates P in $P_\alpha(\vec{E})$ are used to describe recursive data structures starting or ending in locations denoted by variables in $\vec{E}$. The set variable α is interpreted as the set of all locations in the data structure defined by P.

NOLL includes two versions of the separating conjunction: the *per-object separating conjunction* $*$ expresses the disjointness between two sets of heap objects (of record type) while the *per-field separating conjunction* $*_w$ expresses the disjointness between two sets of heap cells, that correspond to fields of heap objects.

The values of the set variables can be constrained in a logic that uses the classical set operators $\in$, $\subseteq$, and $\cup$.

Semantics: The formal semantics of *NOLL* formulas is given by a satisfaction relation $\models$ between pairs (C, J), where $C = (S, H)$ is a program configuration and $J : \mathit{SetVars} \to 2^{\mathit{Loc}}$ interprets variables in *SetVars* to finite subsets of *Loc*, and *NOLL* formulas. Sample clauses of the definition of $\models$ appear in Fig. 4. Given Φ_1 and Φ_2, $\Phi_1 \models \Phi_2$ iff for any (C, J), if $(C, J) \models \Phi_1$ then $(C, J) \models \Phi_2$.

Two spatial atoms are *object separated*, resp. *field separated*, if their least common ancestor in the syntactic tree of the formula is $*$, resp. $*_w$.

Recursive predicates for describing (nested) list segments: In the following, we consider a set of predicates $\mathcal{P}$ that describe *nested list segments* and have recursive definitions of the following form:

$$\begin{aligned} P_\alpha(in, out, \vec{nhb}) \triangleq\ & (in = out) \vee \\ & (\exists u', \vec{v'}, \alpha', \vec{\beta}.\ \Sigma(in, u', \vec{v'}, \vec{nhb}, \vec{\beta}) *_w P_{\alpha'}(u', out, \vec{nhb}) \wedge T_\Sigma \cap \alpha' = \emptyset) \end{aligned} \tag{2}$$

where $in, out, u' \in \mathit{LVars}$, $\vec{nhb}, \vec{v'} \in \mathit{LVars}^*$, $\alpha' \in \mathit{SetVars}$, $\vec{\beta} \in \mathit{SetVars}^*$, Σ is a spatial formula, and T_Σ is a set term, defined as the union of (1) the location variables appearing in the left of a points-to constraint, *except* u', and (2) the set variables in $\vec{\beta}$.

A predicate $P_\alpha(in, out, \vec{nhb})$ defines possibly empty list segments starting from *in* and ending in *out*. The fields of each element in this list segment and the nested lists to which it points to are defined by Σ. The parameters $\vec{nhb}$ are used to define the "boundaries" of the nested list segment described by P, in the sense that every location described by P belongs to a path between *in* and some location in $out \cup \vec{nhb}$ (this path may be defined by more than one field). The constraint $T_\Sigma \cap \alpha = \emptyset$ expresses the fact that the inner list segments are disjoint. We assume several restrictions on the definition of P_α: (1) $\tau(in) = \tau(out) = \tau(u')$, and $\tau(in) \neq \tau(v')$, for every $v' \in \vec{v'}$; this is to ensure that the nesting of different predicates is bounded, and (2) the predicate P does not occur in Σ.

$$\begin{array}{ll}
(C,J) \models emp & \text{iff } Loc(C) = \emptyset \\
(C,J) \models E = F & \text{iff } S^C(E) = S^C(F) \\
(C,J) \models E \mapsto \cup_{i\in I}\{(f_i,E_i)\} & \text{iff } \mathtt{dom}(H^C) = \{(S^C(E),f_i) \mid i \in I\}, \forall i \in I. H^C(S^C(E),f_i) = S^C(E_i) \\
(C,J) \models P_\alpha(\vec{E}) & \text{iff } (C,J) \in [\![P_\alpha(\vec{E})]\!] \text{ and } J(\alpha) = Loc(C). \\
(C,J) \models E \in t & \text{iff } S^C(E) \in [t]_{S^C,J} \\
(C,J) \models t \subseteq t' & \text{iff } [t]_{S^C,J} \subseteq [t']_{S^C,J} \\
(C,J) \models \varphi_1 * \varphi_2 & \text{iff there exist program heaps } C_1 \text{ and } C_2 \text{ s.t. } C = C_1 * C_2, \\
 & \quad (C_1,J) \models \varphi_1, \text{ and } (C_2,J) \models \varphi_2 \\
(C,J) \models \varphi_1 *_w \varphi_2 & \text{iff there exist program heaps } C_1 \text{ and } C_2 \text{ s.t. } C = C_1 *_w C_2, \\
 & \quad (C_1,J) \models \varphi_1, \text{ and } (C_2,J) \models \varphi_2
\end{array}$$

Separation operators over program configurations:

$$\begin{array}{ll}
C = C_1 * C_2 & \text{iff } Loc(C) = Loc(C_1) \cup Loc(C_2) \text{ and } Loc(C_1) \cap Loc(C_2) = \varnothing, \\
 & \quad H^{C_1} = H^C \mid_{Loc(C_1)}, H^{C_2} = H^C \mid_{Loc(C_2)}, \text{ and } S^C = S^{C_1} = S^{C_2} \\
C = C_1 *_w C_2 & \text{iff } \mathtt{dom}(H^C) = \mathtt{dom}(H^{C_1}) \cup \mathtt{dom}(H^{C_2}) \text{ and } \mathtt{dom}(H^{C_1}) \cap \mathtt{dom}(H^{C_2}) = \varnothing, \\
 & \quad H^{C_1} = H^C \mid_{\mathtt{dom}(H^{C_1})}, H^{C_2} = H^C \mid_{\mathtt{dom}(H^{C_2})}, \text{ and } S^C = S^{C_1} = S^{C_2}
\end{array}$$

Interpretation of a set term t, $[t]_{S,J}$:

$$[\{E\}]_{S,J} = \{S(E)\}, \quad [\alpha]_{S,J} = J(\alpha), \quad [\alpha(R)]_{S,J} = J(\alpha) \cap Loc_R, \quad [t \cup t']_{S,J} = [t]_{S,J} \cup [t']_{S,J}.$$

Fig. 4: Semantics of *NOLL* formulas (the set of program configurations satisfying $P(\vec{E})$ is denoted by $[\![P(\vec{E})]\!]$, $\mathtt{dom}(F)$ denotes the domain of the function F, and Loc_R denotes the set of elements in *Loc* of type R)

$$\begin{array}{ll}
(C,J) \in [\![P_\alpha(in,out,\vec{nhb})]\!] & \text{iff there exists } k \in \mathbb{N} \text{ s.t. } (C,J) \in [\![P^k_\alpha(in,out,\vec{nhb})]\!] \\
(C,J) \in [\![P^0_\alpha(in,out,\vec{nhb})]\!] & \text{iff } S(in) = S(out) \text{ and } J(\alpha) = \emptyset \\
(C,J) \in [\![P^{k+1}_\alpha(in,out,\vec{nhb})]\!] & \text{iff } S(in) \neq S(out) \text{ and} \\
 & \quad \text{there exists } \rho : \{u'\} \cup \vec{v'} \to Loc \text{ and } \nu : \{\alpha'\} \cup \vec{\beta} \to 2^{Loc} \text{ s.t.} \\
 & \quad (C[S \mapsto S \cup \rho], J \cup \nu) \models \Sigma(in,u',\vec{v'},\vec{nhb},\vec{\beta}) *_w P^k_{\alpha'}(u',out,\vec{nhb}) \wedge T_\Sigma \cap \alpha' = \emptyset \\
 & \quad \text{and } J(\alpha) = \nu(\alpha') \cup [T_\Sigma]_{\rho,\nu}.
\end{array}$$

Fig. 5: Semantics of list segments predicates ($S \cup \rho$ denotes a new mapping $K : \mathtt{dom}(S) \cup \mathtt{dom}(\rho) \to Loc$ s.t. $K(in) = \rho(x)$, $\forall x \in \mathtt{dom}(\rho)$ and $K(y) = S(y)$, $\forall y \in \mathtt{dom}(S)$)

For any predicate P_α, $Flds_0(P_\alpha)$ is the set of all fields used in points-to constraints of Σ. Also, $Flds(P_\alpha) = Flds_0(P_\alpha) \cup \bigcup_{Q_\beta \text{ in } \Sigma} Flds(Q_\beta)$. If Σ has only points-to constraints then P_α is a 1*-level predicate*. For any $n \geq 2$, if Σ contains only m-level predicates with $m \leq n-1$ and at least one $(n-1)$-level predicate then P_α is a *n-level predicate*.

To simplify the presentation of some constructions, we may use less expressive predicates of the form (Σ contains no points-to constraints having u' on the left side):

$$P_\alpha(in,out,\vec{nhb}) \triangleq (in = out) \vee (\exists u', \vec{v'}, \alpha', \vec{\beta}. \Sigma(in,u',\vec{v'},\vec{nhb},\vec{\beta}) * P_{\alpha'}(u',out,\vec{nhb})) \quad (3)$$

Example 1. The predicates used in the analysis from Sec. 2 are defined as follows:

$$\begin{array}{l}
\mathtt{nll}_\alpha(x,y,z) \triangleq (x = y) \vee \big(\exists u', v', \alpha', \beta. x \mapsto \{(\mathtt{nexth}, u'), (\mathtt{tasks}, v')\} * \mathtt{ls}_\beta(v',z) * \mathtt{nll}_{\alpha'}(u',y,z)\big), \\
\qquad\qquad \text{where } \mathtt{ls}_\alpha(x,y) \triangleq (x = y) \vee \big(\exists u', \alpha'. x \mapsto \{(\mathtt{succ}, u')\} * \mathtt{ls}_{\alpha'}(u',y)\big) \\
\mathtt{dll}_\alpha(x,y) \quad \triangleq (x = y) \vee \big(\exists u', \alpha'. (x \mapsto \{(\mathtt{next}, u')\} * u' \mapsto \{(\mathtt{prev}, x)\}) *_w \mathtt{dll}_{\alpha'}(u',y) \wedge x \notin \alpha'\big) \\
\mathtt{sll}_\alpha(x,y,z) \triangleq (x = y) \vee \big(\exists u', \alpha'. x \mapsto \{(\mathtt{next}, u'), (\mathtt{prev}, z)\} * \mathtt{sll}_{\alpha'}(u',y,z)\big)
\end{array}$$

5 Abstract domain

We define an abstract domain parametrized by a set of predicates $\mathcal{P}$, denoted by $\mathcal{A}SL(\mathcal{P})$, whose elements are *NOLL* formulas over $\mathcal{P}$ represented as sets of graphs. We define the order relation $\preceq$ between two sets of graphs and a widening operator $\triangledown$. We assume that the definitions in $\mathcal{P}$ are not mutually recursive.

5.1 Abstract domain elements

Each disjunct φ of an $\mathcal{A}SL(\mathcal{P})$ element is represented by a labeled directed multi-graph $G[\varphi]$, called *heap graph* [13].

Given $\varphi = \Pi \wedge \Sigma \wedge \Lambda$, every node of $G[\varphi]$ represents a maximal set of equal location variables (according to Π) and it is labeled by the location variables in this set. If Π contains both $E \neq F$ and $E = F$ then $G[\varphi]$ is the bottom element $\perp$. We also assume that nodes are typed according to the variables they represent.

The set of edges in $G[\varphi]$ represent spatial or disequality atoms different from *true* and *emp*. An atom $E \neq F$ is represented by an unlabeled edge from the node labeled by E to the node labeled by F, called a *disequality edge*. An atom $E \mapsto \{(f,F)\}$ is represented by an edge labeled by f from the node of E to the node of F, called a *points-to edge*. An atom $P_\alpha(E,F,\vec{B})$, where $E,F \in LVars$ and $\vec{B} \in LVars^*$, is represented by an edge from the node of E to the node of F labeled by $(P_\alpha,\vec{N_B})$, where $\vec{N_B}$ is the sequence of nodes labeled by $\vec{B}$; such an edge is called a *predicate edge*. A *spatial edge* is a points-to or a predicate edge. The spatial formula *true* is represented by a special node labeled by *true*. A heap graph that does not contain this node is called *precise*. The object separated spatial constraints are represented in $G[\varphi]$ by a binary relation Ω_* over edges. The sharing constraints of φ are kept unchanged in $G[\varphi]$.

Formally, $G[\varphi] = (V,E,\pi,\ell,\Omega_*,\Lambda)$, where V is the set of nodes, E is the set of edges, π is the node typing function, ℓ is the node labeling function, and Ω_* is a symmetric relation over edges in E. The set of all heap graphs is denoted by $\mathcal{HG}$.

In the following, $V(G)$, denotes the set of nodes in the heap graph G; we use a similar notation for all the other components of G. For any node $n \in V(G)$, $PVars_G(n)$ denotes the set of all *program variables* labeling the node n in G, i.e., $PVars_G(n) = \ell(n) \cap Vars$. A node n is called *anonymous* iff $PVars_G(n) = \emptyset$.

The concretization of an $\mathcal{A}SL(\mathcal{P})$ element Φ is defined as the set of models of Φ.

Remark 1. In the elements of $\mathcal{A}SL(\mathcal{P})$, the disjunction is used only at the top most level. In practice, this may be a source of redundancy and inefficiency and some specialized techniques have been proposed in order to deal with disjunctive predicates, e.g., [8] and inner-level disjunction, e.g., [1, 16]. For example, [16] allows disjunctions under the level of the field separated formulas instead of the top-most level. These techniques can be embedded in our framework, by adapting the graph homomorphism approach for defining the order relation between $\mathcal{A}SL(\mathcal{P})$ elements.

5.2 Order relation

The order relation between abstract elements, denoted by $\preceq$, over-approximates the entailment (i.e., if $\Phi_1 \preceq \Phi_2$ then $\Phi_1 \models \Phi_2$) and it is defined using the graph homomorphism approach [9, 13], extended to disjunctions of existentially quantified formulas.

Given two elements $\varphi_1 = \Pi_1 \wedge \Sigma_1 \wedge \Lambda_1$ and $\varphi_2 = \Pi_2 \wedge \Sigma_2 \wedge \Lambda_2$ of $\mathcal{ASL}$, $\varphi_1 \preceq \varphi_2$ iff $G[\varphi_1] = \bot$ or there exists an homomorphism from $G[\varphi_2]$ to $G[\varphi_1]$ defined as follows.

Let G_1 and G_2 be two heap graphs such that G_1 is not precise. An *homomorphism* from G_1 to G_2 is a mapping $h : V(G_1) \to V(G_2)$ such that the following five conditions hold. The constraints imposed by $*$ and $*_w$ are expressed using the function *used* : $E(G_1) \to 2^{E(G_2) \times 2^{Flds}}$, defined meanwhile edges of G_1 are mapped to sub-graphs of G_2.

node labeling preservation: For any $n \in V(G_1)$, $PVars_{G_1}(n) \subseteq PVars_{G_2}(h(n))$.

disequality edge mapping: For any disequality edge $(n,n') \in E(G_1)$, there exists a disequality edge $(h(n),h(n'))$ in $E(G_2)$.

points-to edge mapping: For any points-to edge $e = (n,n') \in E(G_1)$ labeled by f, there exists a points-to edge $e' = (h(n),h(n'))$ labeled by f in $E(G_2)$. We define $used(e) = (e', f)$.

predicate edge mapping: Let $e = (n,n')$ be an edge in G_1 representing a predicate $P_\alpha(in, out, \vec{nhb})$ as in (3) (the extension to predicate definitions as in (2) is straightforward). We assume that $\preceq$ is a partial order on the predicates in $\mathcal{P}$ which is an (over-approximation of) the semantic entailment $\models$ [4].

It is required that either $h(n) = h(n')$ or there exists an homomorphism h_e from the graph representation of a formula that describes an unfolding of P_α to a sub-graph $G_2(e)$ of G_2. Formulas that describe unfoldings of P are of the form:

$$\psi := \phi[E_0, E_1] * \phi[E_1, E_2] * \ldots * \phi[E_{n-1}, E_n], \tag{4}$$

where $E_0 = E$, $E_n = F$, $n \geq 1$, and for any $0 \leq i < n$, $\phi[E_i, E_{i+1}]$ is the formula

$$\Sigma(E_i, E_{i+1}, u', \vec{v'}, \vec{nhb}, \vec{\beta}) \quad \text{or} \quad P'_{\alpha_i}(E_i, E_{i+1}, \vec{nhb}) \text{ with } P'_\alpha \preceq P_\alpha.$$

If $\phi[E_i, E_{i+1}]$ is of the form $P'_{\alpha_i}(E_i, E_{i+1}, \vec{B'})$, then h_e must match the edge corresponding to $\phi[E_i, E_{i+1}]$ with exactly one edge of $G_2(e)$. That is, if m and m' are the nodes labeled by E_i, resp., E_{i+1}, then $G_2(e)$ contains an edge $(h_e(m), h_e(m'))$ labeled by $(P'_{\alpha_i}, \vec{N_{B'}})$, where $\vec{N_{B'}}$ are the nodes labeled by the variables in $\vec{B'}$.

Above, we have reduced the definition of the homomorphism for n-level predicate edges to the definition of the homomorphism for $(n-1)$-level predicate edges. For 1-level predicates, the sub-formula Σ contains only points-to constraints and the definition above reduces to matching points-to edges and checking the order relation between predicates in $\mathcal{P}$.

We define $used(e)$ as the union of $used(e')$ for any edge e' in the graph representation of ψ. If e' is a points-to edge then $used(e')$ is defined as above. If e' is the edge representing some formula $\phi[E_i, E_{i+1}]$ of the form $P'_{\alpha_i}(E_i, E_{i+1}, \vec{nhb})$ and e'' the edge associated to e' by h_e then $used(e') = \{(e'', f) \mid f \in Flds_0(P_\alpha)\}$, where $Flds_0(P_\alpha)$ denotes the set of fields used in points-to constraints of Σ.

separating conjunctions semantics: The semantics of $*_w$ requires that, for any two spatial edges e_1 and e_2 in G_1, $used(e_1) \cap used(e_2) = \emptyset$. The semantics of $*$ requires that for any two edges e_1 and e_2 s.t. $(e_1, e_2) \in \Omega_*(G_1)$, we have that $(e'_1, e'_2) \in \Omega_*(G_2)$, for any edge e'_1 in $used(e_1)$ and any edge e'_2 in $used(e_2)$.

[4] For the set of predicates we have used in our experiments, $\models$ can be checked syntactically.

entailment of sharing constraints: Based on the mapping of edges in $E(G_1)$ to sub-graphs of G_2, we define a substitution Γ for set variables in $\Lambda(G_1)$ to terms over variables in $\Lambda(G_2)$. Let α be a variable in $\Lambda(G_1)$. If α is not bound to a spatial atom then $\Gamma(\alpha) = \alpha$. Otherwise, let α be bound to a spatial atom represented by some edge $e \in E(G_1)$. Then, $\Gamma(\alpha)$ is the union of (1) the set variables bound to spatial atoms denoted by predicate edges in $used(e)$ and (2) the terms $\{x\}$, where x labels the left-end of a points-to edge in $used(e)$. It is required that $\Lambda(G_1)[\Gamma] \Rightarrow \Lambda(G_2)$.

If both G_1 and G_2 are precise then we add the following constraints: (1) predicate ordering $\preceq$ is the equality relation, (2) every edge of G_2 belongs to the image of *used*, and (3) the paths of G_2 associated by h to predicate edges of G_1 can not be interpreted into lasso-shaped or cyclic paths in some model of G_2. Also, by convention, there exists no homomorphism from a precise heap graph to one which is not precise.

The order relation is extended to general $\mathcal{A}SL$ elements as usual: $\Phi_1 \preceq \Phi_2$ iff for each disjunct φ_1 of Φ_1 there exists a disjunct φ_2 in Φ_2 such that $\varphi_1 \preceq \varphi_2$.

The complexity of our procedure for finding a homomorphism is NP time. It extends the unique mapping induced by program variables in a non-deterministic way to the anonymous nodes.

5.3 An effective homomorphism check for predicate edges

In order to check that some predicate edge of G_1 is homomorphic to a sub-graph of G_2, we define an effective procedure based on tree automata. For the simplicity of the presentation, we consider that predicates in $\mathcal{P}$ have the following form:

$$\begin{aligned} P(in, out, \vec{nhb}) \triangleq\ & (in = out) \vee (\exists u', v'. \\ & in \mapsto \{(f, u'), (g, b_1), (h, v')\} * R(v', b_2, \vec{b}) \ * \ P(u', out, \vec{nhb})), \\ & \text{where } b_1, b_2, \vec{b} \subseteq \vec{nhb}, f, h, g \in Flds, \text{ and } R \in \mathcal{P}. \end{aligned} \tag{5}$$

Such predicates describe nested list segments where every two consecutive elements are linked by f, the g field of every element points to some fixed location b, and the h field of every element points to a list segment described by R. Note that the results below can be extended to predicates describing doubly-linked list segments like in (2) or cyclic nested list segments.

Essentially, we model heap graphs by (unranked) labeled trees and then, for each recursive predicate P, we define a (non-deterministic) top-down tree automaton that recognizes exactly all the heap graphs that describe unfoldings of P. The fact that some predicate edge e of G_1 is homomorphic to a sub-graph of G_2 reduces to the fact that the tree-modeling of the sub-graph of G_2 is accepted by the tree automaton corresponding to the predicate labeling e.

Tree automata: A *tree* over an alphabet Σ is a partial mapping $t : \mathbb{N}^* \to \Sigma$ such that $\mathtt{dom}(t)$ is a finite, prefix-closed subset of $\mathbb{N}^*$. Let ε denote the empty sequence. Each sequence $p \in \mathtt{dom}(t)$ is called a *vertex* and a vertex p with no children (i.e., for all $i \in \mathbb{N}$, $p\,i \notin \mathtt{dom}(t)$) is called a *leaf.*

A *(top-down) tree automaton* is a tuple $\mathcal{A} = (Q, \Sigma, I, \delta)$, where Q is a finite set of states, $I \subseteq Q$ is a set of initial states, Σ is a finite alphabet, and δ is a set of transition rules of the form $q \xrightarrow{f} (q_1, \ldots, q_n)$, where $n \geq 0$, $q, q_1, \ldots, q_n \in Q$, and $f \in \Sigma$.

A *run* of $\mathcal{A}$ on a tree t over Σ is a mapping $\pi : \mathtt{dom}(t) \to Q$ such that $\pi(\varepsilon) \in I$ and for each non-leaf vertex $p \in \mathtt{dom}(t)$, δ contains a rule of the form $q \xrightarrow{t(p)} (q_1, \ldots, q_n)$, where $q = \pi(p)$ and $q_i = \pi(p\,i)$, for all i such that $p\,i \in \mathtt{dom}(t)$. The *language* of $\mathcal{A}$ is the set of all trees t for which there exists a run of $\mathcal{A}$ on t.

Tree-modeling of heap graphs: Note that the minimal heap graphs that are homomorphic to a predicate edge have a special form: nodes with more than one incoming edge have no successors. Such heap graphs are transformed into tree-shaped graphs with labeled nodes by (1) moving edge labels to the source node and (2) introducing copies of nodes with more than one incoming edge (i.e., for every set of edges E' having the same destination n, introduce $|E'|$ copies of n and replace every edge (m,n) by (m,n_m), where n_m is a copy of n). We also replace labels of predicate edges of the form $(P_\alpha, \vec{N_B})$ with $(P, \vec{B})$, where $\vec{B}$ is a tuple of variables labeling the nodes in N_B.

Given a finite set of variables $\mathcal{V}$, let $\Sigma_{\mathcal{V}} = 2^{Flds \cup \mathcal{V}} \cup \{(P, \vec{B}) \mid P \in \mathcal{P}, \vec{B} \in \mathcal{V}^+\}$. The *tree-modeling* of a heap graph G is the tree $t[G]$ over $\Sigma_{\mathcal{V}}$ isomorphic to the tree-shaped graph described above; $\mathcal{V}$ is the set of variables labeling nodes of the graph.

Tree automata recognizing unfoldings of recursive predicates: The definition of the tree-automata associated to a predicate P as in (5), denoted $\mathcal{A}_P$, follows its recursive definition. $I(\mathcal{A}_P) = \{q_0^P\}$ and the transition rules in $\delta(\mathcal{A}_P)$ are defined as follows:

$$\begin{array}{lll}
q_0^P \xrightarrow{\{in,f,g,h\}} (q_{rec}^P, q_g^P, q_h^P) & q_{rec}^P \xrightarrow{\{f,g,h\}} (q_{rec}^P, q_g^P, q_h^P) & q_h^P \xrightarrow{h} q_0^R \\
q_0^P \xrightarrow{\{in,out\}} \varepsilon & q_{rec}^P \xrightarrow{(P',\vec{B})} q_{rec}^P, \text{ with } P'(E,F,\vec{B}) \preceq P(E,F,\vec{B}) & q_g^P \xrightarrow{b} \varepsilon \\
 & q_{rec}^P \xrightarrow{out} \varepsilon & Rules(R)[\Upsilon]
\end{array}$$

where q_0^R is the initial state of $\mathcal{A}_R$, the tree automaton for R, and $Rules(R)[\Upsilon]$ denotes the set $\delta(\mathcal{A}_R)$ where variables are substituted by the actual parameters v', b_2, and $\vec{b}$.

6 Widening

We describe the widening operator $\triangledown$, which satisfies the following properties: (1) it defines an upper bound for any two elements of $\mathcal{ASL}$, i.e., given $\Phi_1, \Phi_2 \in \mathcal{ASL}$, $\Phi_1 \preceq \Phi_1 \triangledown \Phi_2$ and $\Phi_2 \preceq \Phi_1 \triangledown \Phi_2$, and (2) for any infinite ascending chain $\Phi_1 \preceq \Phi_2 \preceq \ldots \preceq \Phi_n \preceq \ldots$ in $\mathcal{ASL}$, there exists a finite chain $\Phi_1^\triangledown \preceq \ldots \preceq \Phi_k^\triangledown$ in $\mathcal{ASL}$ such that $\Phi_1^\triangledown = \Phi_1$, $\Phi_i^\triangledown = \Phi_{i-1}^\triangledown \triangledown \Phi_{i+1}$, for every $2 \le i \le k$, and $\Phi_k^\triangledown = \Phi_k^\triangledown \triangledown \Phi_{k+1}$. The widening operator is used to ensure the termination of fixed point computations over elements of $\mathcal{ASL}$.

We define $\varphi_1 \triangledown \varphi_2$, for any φ_1 and φ_2 two disjunction-free formulas in $\mathcal{ASL}$. The extension to disjunctions is straightforward. The widening operator is parametrized by a natural number K such that $\varphi_1 \triangledown \varphi_2$ returns a set of heap graphs with at most $max(annon(\varphi_1), K)$ anonymous nodes, where $annon(\varphi_1)$ is the number of anonymous nodes in $G[\varphi_1]$. Therefore, an infinite ascending chain is over-approximated by a finite one, whose elements have at most as many anonymous nodes as either the first element of the infinite ascending chain or the parameter K, depending on which one is bigger.

Widening operator description: Intuitively, the result of $\varphi_1 \triangledown \varphi_2$ should preserve the properties which are present in both φ_1 and φ_2. To this, each graph $G_i = G[\varphi_i]$, $i \in \{1,2\}$, is split into three sub-graphs G_i^+, G_i^-, and $G_i^\sim$ such that:

- the sets of spatial edges in these sub-graphs form a partition of the set of spatial edges in G_i;
- there exists an homomorphism $h_\rightarrow$ from the sub-graph G_1^+ to G_2^- and an homomorphism $h_\leftarrow$ from G_2^+ to G_1^-;
- G_1^+ and G_2^+ are maximal.

Here, a heap graph G' is called a *sub-graph* of a heap graph G if it contains only a subset of the nodes and edges in G, for each node n, the labeling of n in G' is a subset of the labeling of n in G such that $PVars_G(n) \neq \emptyset$ implies $PVars_{G'}(n) \neq \emptyset$, and the Ω_* and Λ components of G' are also subsets of the corresponding components of G.

The two tuples of sub-graphs are called an *homomorphism induced partition*.

Example 2. Let G_1 and G_2 be the graphs pictured in the first row of Fig. 7. The tuples of sub-graphs $(G_1^+, G_1^-, G_1^\approx)$ and $(G_2^+, G_2^-, G_2^\approx)$ given in the second row of Fig. 7 define homomorphism induced partitions for G_1 and, resp., G_2. These partitions correspond to the homomorphisms $h_{ij} : G_i^+ \rightarrow G_j^-$, which map the node labeled by y (resp., z) in G_i^+ to the node labeled by y (resp., z) in G_j^-, for all $1 \leq i \neq j \leq 2$.

If $G_1^\approx = \emptyset$ and $G_2^\approx = \emptyset$ then each spatial edge of G_1 is homomorphic to a sub-graph of G_2 or vice-versa. In this case, the result of the widening is the graph defined as the union of G_1^+ and G_2^+, i.e., the weakest among the comparable sub-graphs. The union operator $\uplus$ is parametrized by the two homomorphisms $h_\rightarrow$ and $h_\leftarrow$ in order to (1) merge nodes which are matched according to these homomorphisms, (2) identify the object separated spatial constraints in the union of the two sub-graphs, and (3) compute the union of the sharing constraints. In this case, the number of anonymous nodes in $\varphi_1 \triangledown \varphi_2$ is bounded by the number of anonymous nodes in φ_1.

Otherwise, the widening operator tries to return two graphs G_1' and G_2', each of them being obtained from G_1 resp. G_2, by replacing the sub-graph G_i^- with its weaker version G_j^+, where $i, j \in \{1,2\}$ and $i \neq j$. However, to preserve the bound on the number of anonymous nodes, this operation is possible only if the number of anonymous nodes in G_1' and G_2' is smaller than $max(annon(\varphi_1), K)$.

Example 3. In Fig. 7, if $K = 3$ then $G_1 \triangledown G_2$ has two disjuncts. The first one, G_1', is obtained from G_1 using the homomorphism h_{21}: (1) the graph G_1^- is replaced by G_2^+, and (2) the sharing constraints $\beta = \alpha^1 \cup \alpha^2$ become $\alpha = \beta$, where β is the set of locations in the doubly-linked list defined by $\mathtt{dll}_\beta$. The second disjunct, G_2', is obtained similarly from G_2 using h_{12}.

Operator fold**:** If the above condition on the number of anonymous nodes is not satisfied then, we apply an operator called fold on each of the two graphs. Given a heap graph G and some $b \in \mathbb{N}$, fold builds a graph F with at most b anonymous nodes, obtained from G by replacing edges used by non-empty unfoldings of recursive predicates with predicate edges (labeled with fresh set variables). Moreover, nodes in F without incident edges are removed. The graph F is homomorphic to G and fold also returns the homomorphism h from F to G. The operator fold may fail to obtain a graph with at most b anonymous nodes in which case it returns $\top$.

The object/field separation between predicate edges copied from G to F is preserved. Two predicate edges in F are object separated if (1) they replace two unfoldings in G that contain disjoint sets of nodes and any two predicate edges from the two unfoldings are object separated in G or (2) one of them, denoted by e_1, replaces an unfolding in

G, the other one, denoted by e_2, is copied from G, and all the edges from the unfolding are object separated from e_2 in G. Otherwise, the predicate edges are field separated.

```
$G_1 \triangledown G_2$ ::=
  let $(G_1^+, G_1^-, G_1^\sim)$ and $(G_2^+, G_2^-, G_2^\sim)$
    be an homomorphism induced partition
    of $G_1$ and $G_2$
  let $\pi = (h_\rightarrow, h_\leftarrow)$
  if $(G_1^\sim = \emptyset \wedge G_2^\sim = \emptyset)$ then
    return $G_1^+ \uplus^\pi G_2^+$;
  else
    $G_1' = (G_1^+ \cup G_1^\sim) \uplus^{h_\leftarrow} G_2^+$;
    $G_2' = (G_2^+ \cup G_2^\sim) \uplus^{h_\rightarrow} G_1^+$;
    $\mathsf{bound} = max(annon(\varphi_1), K)$;
    if $(annon(G_1') \leq \mathsf{bound}$
      $\wedge annon(G_2') \leq \mathsf{bound})$ then
        return $G_1' \vee G_2'$;
    else
        for each $1 \leq i \leq 2$ do
          $b_i = \mathsf{bound} - annon(G_i^+ \cup G_i^-)$;
          $(F_i, h_i) = \mathsf{fold}(G_i^\sim, b_i)$;
          $G_i' = (G_i^+ \cup G_i^-) \uplus^{h_i} F_i$;
        return $G_1' \triangledown G_2'$;
```

Fig. 6: The definition of $\triangledown$

Another important property of fold is to maintain relations between sets of locations that correspond to unfoldings of recursive predicates, replaced by predicate edges. For any sub-graph G' of G representing the unfolding of a recursive predicate, let $T_{G'}$ be the set term defined as the union of all set variables labeling edges in G' and all location variables, which are the source of at least one points-to edge in G'. Based on the equalities between location variables in G, the sharing constraints $\Lambda(G)$, and the inference rule "if $t_1 = t_1'$ and $t_2 = t_2'$ then $t_1 \cup t_2 = t_1' \cup t_2'$", for any set terms t_1, t_1', t_2, and t_2', fold generates new equalities between (unions of) set terms $T_{G'}$ or between (unions of) set terms $T_{G'}$ and set variables labeling edges copied from G to F. Once a predicate edge e in F replaces the unfolding of a recursive predicate G', the set term $T_{G'}$ is substituted by the set variable labeling e. Similarly, fold generates constraints of the form $t \cap t' = \emptyset$ with t and t' set terms, and constraints of the form $x \in t$ or $x \notin t$.

In the definition of $\triangledown$, the argument of fold is the graph $G_i^\sim$ where anonymous nodes that are incident to edges in G_i^+ or G_i^- are labeled by ghost program variables so they are preserved in the output of fold. Then, the graph G_i' is defined by replacing the sub-graph $G_i^\sim$ of G_i with the one returned by fold, i.e., F_i. This replacement is written as the union of the sub-graph $G_i^+ \bigcup G_i^-$ with F_i, where the homomorphism h_i is used to merge nodes which are associated by h_i and to identify the object separated constraints. Finally, the widening operator is called recursively on the new graphs. Notice that, the widening operator contains at most one recursive call. The first execution of $\triangledown$ recognizes unfoldings of recursive predicates and, if needed, eliminates enough anonymous nodes in order to make the recursive call succeed.

Example 4. In Fig. 7, if $K = 0$ then `bound` $= 1$ because G_1 contains one anonymous node. The computation of $\triangledown$ based only on $\uplus$ doesn't satisfy the bound on the number of anonymous nodes because $G_2^\sim$ contains two anonymous nodes labeled by v' and u'. Therefore, we apply $\mathsf{fold}(G_2^\sim, 1)$ and the result is the graph F_2 given on the third row of Fig. 7. The graph F_2 is obtained from $G_2^\sim$ by replacing the sub-graph of $G_2^\sim$ that consists of all edges labeled by `succ` with a predicate edge labeled by $\mathtt{ls}_\delta$ and the sub-graph of $G_2^\sim$ that consists of all edges labeled by `next` and `prev` with a predicate edge labeled by $\mathtt{dll}_\gamma$. The set terms which correspond to these two unfoldings are $\{\mathtt{x}\} \cup \{u'\} \cup \{v'\}$ and respectively, $\{\mathtt{x}\} \cup \{u'\} \cup \{v'\}$. Since they contain exactly the same location variables the equality $\gamma = \delta$ is added to the sharing constraints of the output graph. Also, because

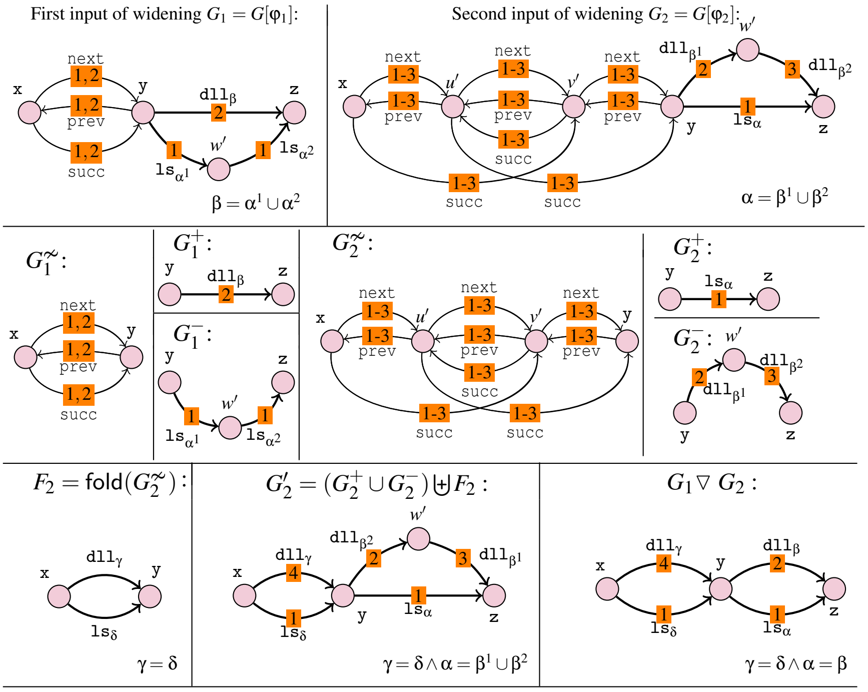

First input of widening $G_1 = G[\varphi_1]$:
x
y
z
w'
next
prev
succ
1,2
$\mathtt{dll}_\beta$
2
1
$\mathtt{ls}_{\alpha^1}$
$\mathtt{ls}_{\alpha^2}$
$\beta = \alpha^1 \cup \alpha^2$
Second input of widening $G_2 = G[\varphi_2]$:
u'
v'
w'
1-3
$\mathtt{dll}_{\beta^1}$
$\mathtt{dll}_{\beta^2}$
3
$\mathtt{ls}_\alpha$
$\alpha = \beta^1 \cup \beta^2$
$G_1^{\sim}$:
G_1^+:
G_1^-:
$G_2^{\sim}$:
G_2^+:
G_2^-:
$F_2 = \mathsf{fold}(G_2^{\sim})$:
$\mathtt{dll}_\gamma$
$\mathtt{ls}_\delta$
$\gamma = \delta$
$G_2' = (G_2^+ \cup G_2^-) \uplus F_2$:
4
$\gamma = \delta \wedge \alpha = \beta^1 \cup \beta^2$
$G_1 \triangledown G_2$:
$\gamma = \delta \wedge \alpha = \beta$

Fig. 7: Steps in the computation of $\triangledown$ (the object separated edges are defined by $\Omega_* = \{(e,e') \mid \text{the integers labeling } e \text{ are included in the integers labeling } e' \text{ or vice-versa}\}$).

all the edges incident to u' and v' are included in the two unfoldings, these two nodes are removed. The two edges of F_2 are field separated because the corresponding unfoldings share some node. Since $G_1^{\sim}$ has no anonymous nodes, $\mathsf{fold}(G_1^{\sim}, 1) = G_1^{\sim}$ and $G_1' = \left(G_1^+ \bigcup G_1^-\right) \uplus^{h_i} F_1 = G_1$.

The second column in the last row of Fig. 7 shows the graph G_2' obtained by replacing in G_2, $G_2^{\sim}$ with F_2. Finally, $G_1 \triangledown G_2$ equals $G_1 \triangledown G_2'$, which is given in the bottom right corner of Fig 7. Notice that the computation of $G_1 \triangledown G_2'$ requires only the union.

Operator $\uplus^h$: Formally, $\uplus^h$ replaces in a given graph G, a sub-graph G' by another graph G'' s.t. h is an homomorphism from G'' to G'. For example, $\left(G_1^+ \bigcup G_1^{\sim}\right) \uplus^{h_\leftarrow} G_2^+$ replaces the sub-graph G_1^- of G_1 with the graph G_2^+ ($h_\leftarrow$ is an homomorphism from G_2^+ to G_1^-). The result of $\uplus^h$ on G, G', G'', and h is the heap graph $(V, E, \pi, \ell, \Omega_*, \Lambda)$, where:

- V is obtained from $V(G \setminus G') \cup V(G'')$, where $V(G \setminus G')$ is the set of nodes in G, which have at least an incident edge not included in G', by merging every $m \in V(G)$ with one of the nodes in $h^{-1}(m)$, provided that $h^{-1}(m) \neq \emptyset$;
- for any $n \in V$, the set of variables $\ell(n)$ is the union of the label of n in G, $\ell(G)(n)$, and the label of n in G'', $\ell(G'')(n)$;
- $E = (E(G) \setminus E(G')) \cup E(G'')$;
- Ω_* is defined from $\Omega_*(G)$ and $\Omega_*(G'')$ as follows: $(e,e') \in \Omega_*$ iff either (i) $(e,e') \in \Omega_*(G)$, (ii) $(e,e') \in \Omega_*(G'')$, or (iii) $e \in E(G)$, $e'' \in E(G'')$, and for any edge e' in the sub-graph of G to which e'' is mapped by the homomorphism h, $(e,e') \in \Omega_*(G)$;

- Λ is the union of (1) the constraints $\Lambda(G'')$ in G'' and (2) the constraints $\Lambda(G)$ in G where every set term t, denoting all the locations in some heap region described by a sub-graph of G' associated by the homomorphism h to an edge e'' of G'', is replaced by the set variable α in the label of e''. Note that Λ contains only set variables which appear in labels of E.

Operator $\uplus^{\pi}$: Given $(G_1^+, G_1^-, G_1^{\sim})$ and $(G_2^+, G_2^-, G_2^{\sim})$ two homomorphism induced partitions of G_1 and resp., G_2, and $\pi = (h_{\rightarrow}, h_{\leftarrow})$, $G_1^+ \uplus^{\pi} G_2^+$ is defined similarly to $G_1^+ \uplus^{h_{\leftarrow}} G_2^+$ [5] except for the fact that the constraint on set variables is defined as the conjunction of all the constraints which appear in both $G_1^+ \uplus^{h_{\leftarrow}} G_2^+$ and $G_2^+ \uplus^{h_{\rightarrow}} G_1^+$.
Complexity: The search for homomorphism induced partitions is done in linear time in the size of the input graphs. The correspondence between anonymous nodes in $h_{\leftarrow}$ and $h_{\rightarrow}$ is chosen arbitrarily (in practice, they are chosen according to some heuristics). In order to obtain a more precise result, one should enumerate an exponential number of such mappings. Given an arbitrary but fixed order relation on the recursive predicates, the complexity of the operator fold which replaces predicate unfoldings with predicate edges according to this order is PTIME. In our implementation, the operator fold is parametrized by a set of order relations among recursive predicates (in practice, we have used a small number of such order relations) and it enumerates all of them until it succeeds to eliminate the required number of anonymous nodes.

7 Abstract Transformers

In this section, we describe the abstract transformers associated with intra-procedural statements (in Sec. 7.1) and procedure calls and returns (in Sec. 7.2).

7.1 Intra-procedural Analysis

For any basic statement C, which does not contain a procedure call, the analysis uses an abstract transformer $[\![C]\!]^{\sharp} : \mathcal{HG} \rightarrow 2^{\mathcal{HG}}$ that, given a heap graph $G \in \mathcal{HG}$, it returns either $\top$ meaning that a possible memory error has been encountered, or a set of heap graphs representing the effect of the statement C on G. The transformers $[\![C]\!]^{\sharp}$ are defined as the composition of three (families of) functions:

materialization $\rightarrow_{x,f}$: transforms a heap graph, via case analysis, into a set of heap graphs where the value of the field f stored at the address pointed to by the program variable x is concretized. It returns $\top$ if it can not prove that x is allocated in the input.
symbolic execution $\rightsquigarrow$: expresses the concrete semantics of the statements in terms of heap graphs, e.g., for an assignment $x := y$, it merges the nodes labeled by x and y.
consistency check $\rightarrow^{\#}$: takes the graphs from the output of the symbolic execution and checks if their concretization is empty or if they contain garbage. All graphs with empty concretization are removed and if garbage is detected then the result is $\top$.

[5] In $G_1^+ \uplus^{h_{\leftarrow}} G_2^+$, G_1^- is replaced by G_2^+.

Materialization $\rightarrow_{x,f}$: The relation $\rightarrow_{x,f}$ for dereferencing the field f of the object pointed to by x is defined by a set of rewriting rules over $\mathcal{ASL}$ elements. When the value of f is characterized by a predicate edge that starts or ends in a node labeled by x then the rules are straightforward. If the sharing constraints imply that x belongs to the list segment described by some predicate edge (which is not incident to a node labeled by x) and this list segment uses the field f then, we use the following rules:

$G \quad \rightarrow_{x,f} \quad \texttt{unfoldMiddle}(G,e,x)$
if e is a predicate edge in G labeled by P_α such that $\Lambda(G) \Rightarrow x \in \alpha$
and $f \in Flds_0(P_\alpha)$

$G \quad \rightarrow_{x,f} \quad \texttt{unfoldMiddle}(G,e,\_)$
if e is a predicate edge in G labeled by P_α such that $\Lambda(G) \Rightarrow x \in \alpha$
and $f \in Flds(P_\alpha) \setminus Flds_0(P_\alpha)$

The function $\texttt{unfoldMiddle}(G,e,\xi)$ concretizes the fields of an *arbitrary* object in the list segment described by e, that has the same type as the nodes incident to e. Suppose that e starts in the node n, labeled by y_1, and ends in the node n', labeled by y_2. For simplicity, suppose that e is labeled by a predicate P_α defined as in (3). Then, `unfoldMiddle` replaces the edge e with the graph representation of $P_{\alpha'}(y_1,u'_1,\vec{nhb}) *_w \Sigma(u'_1,u'_2,\vec{v}',\vec{nhb},\vec{\beta}) *_w P_{\alpha''}(u'_2,y_2,\vec{nhb})$. If ξ is a program variable and the formula Σ in the definition of P_α contains a points-to constraint of the form $in \mapsto \{(f,w)\}$, for some $w \in LVars$, i.e., the direction of f is from x to y then, the node labeled by ξ is merged with the node labeled by u'_1. Otherwise, the node labeled by ξ is merged with the node labeled by u'_2. Finally, the constraints $T_\Sigma \cap \alpha' = \emptyset$, $T_\Sigma \cap \alpha'' = \emptyset$, $\alpha' \cap \alpha'' = \emptyset$ are added to $\Lambda(G)$ and all occurrences of α in $\Lambda(G)$ are substituted with $\alpha' \cup T_\Sigma \cup \alpha''$.

In general, the output of $\rightarrow_{x,f}$ depends on the (dis)equalities which are explicit in the input heap graph (e.g., an equality $x_1 = x_2$ is explicit if there exists a node labeled by both x_1 and x_2). In order to make explicit all the (dis)equalities which are implied by the input graph, (e.g., $ls(x,y) * ls(x,z) * y \mapsto \{(f,t)\}$ implies $x = z$), one can use the normalization procedure introduced in [13], based on SAT solvers.

7.2 Inter-procedural analysis

We consider an inter-procedural analysis based on the *local heap* semantics [18]. The analysis explores the call graph of the program starting from the `main` procedure and proceeds downward, computing for each procedure `Proc`, a set of summaries of the form (φ_i, Φ_o), where φ_i and Φ_o are $\mathcal{ASL}$ elements, φ_i disjunction-free. Essentially, φ_i is an abstract element built at the entry point of the procedure `Proc` by the transformer associated to $\texttt{call Proc}(\vec{x})$, which describes the part of the heap reachable from the actual parameters $\vec{x}$. Then, Φ_o is the abstract element obtained by analyzing the effect of `Proc` on the input described by φ_i. The transformer associated to the `return` statement takes a summary (φ_i, Φ_o) of `Proc` and returns a set of heap graphs obtained from the ones associated to the control point of the caller that precedes the procedure call by replacing the graph φ_i with each of the graphs in Φ_o. The important operations for the inter-procedural analysis are the computation of the local heap at the procedure call and the substitution of the local heap with the output heap at the procedure return.

These operations are more difficult in the presence of *cutpoints*, i.e., locations in the local heap of the callee which are reachable from local variables of other procedures in the call stack without passing through the actual parameters. For simplicity, we define the `call` transformer such that it returns $\top$ whenever it detects cutpoints. This is still relevant, because in practice most of the procedure calls are cutpoint-free.

Frame rules: We follow the approach in Gotsman et al. [14] and define these transformers based on a frame rule for procedure calls. The fragment of *NOLL* without $*_w$ and sharing constraints satisfies the following frame rule [14]:

$$\frac{\phi \models \phi_{call}\sigma * \phi_1 \quad \phi_{ret}\sigma * \phi_1 \models \phi' \quad \{\phi_{call}\}\mathtt{Proc}(\vec{x})\{\phi_{ret}\}}{\{\phi\}\mathtt{Proc}(\vec{x}\sigma)\{\phi'\}}$$

where σ is the substitution from formal to actual parameters. Intuitively, this means that it is possible to analyze the effect of $\mathtt{Proc}(\vec{x})$ on a part of the heap, ϕ_{call}, while holding the rest of the heap ϕ_1 aside, to be added to the heap ϕ_{ret} that results from executing `Proc`. In general, $\phi_{call}\sigma$ and ϕ_1 can be chosen arbitrarily but, in order to be precise, $\phi_{call}\sigma$ should contain all the heap locations reachable from the actual parameters.

In the following, we extend this frame rule to work for both per-object and per-field separating conjunction. Given a disjunction-free *NOLL* formula φ, $T[\varphi]$ denotes the set term which is the union of all set variables in φ and all location variables which are on the left side of a points-to constraint. Then, the following holds:

$$\frac{\phi \models (\phi_{call}\sigma *_w \phi_1) * \phi_2 \quad (\phi_{ret}\sigma *_w \phi_1) * \phi_2 \wedge \Lambda \models \phi' \quad \{\phi_{call}\}\mathtt{Proc}(\vec{x})\{\phi_{ret}\}}{\{\phi\}\mathtt{Proc}(\vec{x}\sigma)\{\phi'\}}$$

where σ is the substitution from formal to actual parameters and

$$\Lambda \triangleq T[\phi_{ret}] \cap T_1 = \emptyset \text{ with } T_1 \text{ a set term over variables in } \phi_1 \text{ s.t. } \phi \models T[\phi_{call}\sigma] \cap T_1 = \emptyset.$$

The formula Λ expresses the fact that if all heap locations in the interpretation of T_1 are disjoint from the ones included in the local heap of `Proc` then, they will remain disjoint from all the locations in the output of `Proc`.

Computing the local heap: To compute ϕ_{call}, we proceed as follows (as before, this is important only for precision). For any procedure `Proc`, *Flds*(`Proc`) denotes the set of fields accessed by `Proc`, which consists of (1) the fields f such that x->f appears in some expression and (2) all the fields of the type *RT* such that `Proc` contains a `free` statement over a variable of type *RT*. This set of fields can be computed easily from the syntactic tree of `Proc`. Given an $\mathcal{A}$*SL* element ϕ, a spatial edge in ϕ is called a `Proc`-edge iff it is a points-to edge labeled by some $f \in Flds(\mathtt{Proc})$ or a predicate edge labeled by P_α with $Flds(P_\alpha) \cap Flds(\mathtt{Proc}) \neq \emptyset$.

The set of spatial edges in ϕ_{call} is the union of (1) the subgraph G_r of ϕ that contains all the `Proc`-edges which are reachable from the actual parameters using only `Proc`-edges and (2) all the `Proc`-edges e such that the set of locations characterized by e is not disjoint from the set of locations characterized by G_r, according to ϕ (e.g., if e is a predicate edge labeled by P_α then $\phi \not\Rightarrow \alpha \cap T[G_r] = \emptyset$). The pure and sharing constraints in ϕ_{call} are all the pure and the sharing constraints in ϕ that contain variables from the spatial constraints in ϕ_{call}.

Table 1: Experimental results on an Intel Core i3 2.4 GHz with 2GB of memory ($3 \times$ `dll` means 3 instances of the predicate `dll` over 3 disjoint sets of pointer fields)

program	*size*		*spec NOLL*	*analysis – max*			time (sec)
	#fun	#lines	$\mathcal{P}$	#iter	#graphs	#anon (K)	
`list-dio`	5	134	$2 \times$ `dll`	5	16	3	<3
`many-keys`	4	87	$3 \times$ `dll`	3	8	2	<1
`cache`	4	88	`dll`	3	5	2	<1
`nagios-event`	4	90	`nll`, $2 \times$ `ls`	3	5	2	<2
`nagios-task`	4	112	`nll`,`dll`,`sll`,`ls`	5	8	3	<2
`nagios-queue`	4	101	`nll`,`dll`,`ls`	3	5	2	<2

8 Experiments

We have implemented our inter-procedural analysis in the plugin CELIA [3, 5] of the FRAMA-C platform [4] for C program analysis. The domain $\mathcal{ASL}$ has been implemented as a C library which takes as input a set of predicates defined in the logic ACSL of FRAMA-C. To reduce the number of disjuncts in a formula, we define an heuristic for choosing when to replace pairs of disjuncts G_1, G_2 by their widening $G_1 \triangledown G_2$.

We have considered two classes of programs. The first class contains the examples from [16] which manipulate (i.e., create, find, add, delete) doubly linked lists (DLL): (`list-dio`) manipulates two overlaid, circular DLL per-object separated from a third circular DLL; (`many-keys`) manipulates three overlaid and circular DLL; (`cache`) manipulates one circular DLL with a pointer to the last added cell. The second class of examples is extracted from the Nagios data structures and work on nested linked lists (NLL) combined with singly (SLL) or doubly linked lists: (`nagios-event`) manipulates an NLL where all the nested cells are shared with a SLL, (`nagios-task`) is the example considered in the overview, and (`nagios-queue`) manipulates an NLL where some of the nested list cells are shared with a DLL.

Table 1 presents the results of our analysis on the above benchmark. Column $\mathcal{P}$ indicates the set of predicates used by the analysis for each example. The last five columns gives collected informations about the analysis, e.g., the number of widening points, the maximal number of graphs in the abstract values and the maximal size of these graphs, and the maximal time for the analysis of included functions. For the first three examples, the comparison with the execution times reported in [16] raises the following comments. The experiments have been done on different hardware configurations and the set of recursive predicates supported in [16] does not include predicates for describing nested lists, but predicates for describing tree data structures.

The precision and the efficiency of our analysis depends on several factors. One factor is the choice of an adequate set $\mathcal{P}$ of recursive predicates. The set $\mathcal{P}$ should be expressive enough to describe all the data structures manipulated by the program and, for efficiency, it should be minimal and not contain unused predicates (such predicates may slow down the fold procedure used in the widening operator). The scalability of our analysis also depends on the modularity of the input program: if the code is structured in functions that deal with one non-overlapped list at a time then the analysis is more efficient. However, this does not have an influence on the precision of the analysis.

References

1. T. Ball, A. Podelski, and S. K. Rajamani. Boolean and cartesian abstraction for model checking c programs. In *TACAS*, volume 2031 of *LNCS*, pages 268–283. Springer, 2001.
2. J. Berdine, C. Calcagno, B. Cook, D. Distefano, P.W. O'Hearn, T. Wies, and H. Yang. Shape analysis for composite data structures. In *CAV*, volume 4590 of *LNCS*, pages 178–192. Springer, 2007.
3. A. Bouajjani, C. Dragoi, C. Enea, and M. Sighireanu. On inter-procedural analysis of programs with lists and data. In *PLDI*, pages 578–589. ACM, 2011.
4. CEA. *Frama-C Platform*. `htp://frama-c.com`.
5. Celia plugin. `http://www.liafa.univ-paris-diderot.fr/celia`.
6. B.-Y.E. Chang and X. Rival. Relational inductive shape analysis. In *POPL*, pages 247–260. ACM, 2008.
7. B.-Y.E. Chang, X. Rival, and G. C. Necula. Shape analysis with structural invariant checkers. In *SAS*, volume 4634 of *LNCS*, pages 384–401. Springer, 2007.
8. W.N. Chin, C. Gherghina, R. Voicu, Q. Loc Le, F. Craciun, and S. Qin. A specialization calculus for pruning disjunctive predicates to support verification. In *CAV*, volume 6806 of *LNCS*, pages 293–309. Springer, 2011.
9. B. Cook, C. Haase, J. Ouaknine, M. J. Parkinson, and J. Worrell. Tractable reasoning in a fragment of separation logic. In *CONCUR*, volume 6901 of *LNCS*, pages 235–249, 2011.
10. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*. ACM, 1977.
11. D. Distefano, P.W. O'Hearn, and H. Yang. A local shape analysis based on separation logic. In *TACAS*, volume 3920 of *LNCS*, pages 287–302. Springer, 2006.
12. K. Dudka, P. Müller, P. Peringer, and T. Vojnar. Predator: A verification tool for programs with dynamic linked data structures - (competition contribution). In *TACAS*, volume 7214 of *LNCS*, pages 545–548. Springer, 2012.
13. C. Enea, V. Saveluc, and M. Sighireanu. Compositional invariant checking for overlaid and nested linked lists. In *ESOP*, volume 7792 of *LNCS*, pages 178–195. Springer, 2013.
14. A. Gotsman, J. Berdine, and B. Cook. Interprocedural shape analysis with separated heap abstractions. In *SAS*, volume 4134 of *LNCS*, pages 240–260. Springer, 2006.
15. S. Gulwani, T. Lev-Ami, and M. Sagiv. A combination framework for tracking partition sizes. In *POPL*, pages 239–251. ACM, 2009.
16. O. Lee, H. Yang, and R. Petersen. Program analysis for overlaid data structures. In *CAV*, volume 6806 of *LNCS*, pages 592–608. Springer, 2011.
17. J.C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, pages 55–74. IEEE Computer Society, 2002.
18. N. Rinetzky, J. Bauer, T.W. Reps, S. Sagiv, and R. Wilhelm. A semantics for procedure local heaps and its abstractions. In *POPL*, pages 296–309. ACM, 2005.
19. A. Toubhans, B.-Y.E. Chang, and X. Rival. Reduced product combination of abstract domains for shapes. In *VMCAI*, volume 7737 of *LNCS*, pages 375–395. Springer, 2013.
20. H. Yang, O. Lee, J. Berdine, C. Calcagno, B. Cook, D. Distefano, and P.W. O'Hearn. Scalable shape analysis for systems code. In *CAV*, volume 6901 of *LNCS*, pages 385–398. Springer, 2008.