Compositional Invariant Checking for Overlaid and Nested Linked Lists

Constantin Enea, Vlad Saveluc, Mihaela Sighireanu

Compositional Invariant Checking for Overlaid and Nested Linked Lists

C. Enea, V. Saveluc, and M. Sighireanu

Univ Paris Diderot, Sorbonne Paris Cite, LIAFA CNRS UMR 7089, Paris,
{cenea,sighirea}@liafa.univ-paris-diderot.fr
vlad.saveluc@gmail.com

Abstract. We introduce a fragment of separation logic, called *NOLL*, for automated reasoning about programs manipulating overlaid and nested linked lists, where overlaid means that the lists may share the same set of objects. The distinguishing features of *NOLL* are: (1) it is parametrized by a set of user-defined predicates specifying nested linked list segments, (2) a "per-field" version of the separating conjunction allowing to share locations but not fields, and (3) it can express sharing constraints between list segments. We prove that checking the entailment between two *NOLL* formulas is co-NP complete. For this result, the decision procedure for entailment is based on a small model property. We also provide an effective procedure for checking entailment in *NOLL*, which first constructs a Boolean abstraction of the two formulas, in order to infer all the implicit constraints, and then, it checks the existence of a homomorphism between the two formulas, viewed as graphs. We have implemented this procedure and applied it on verification conditions generated from several interesting case studies that manipulate overlaid and nested data structures.

1 Introduction

Reasoning about behaviors of programs that manipulate dynamic data structures is a challenging problem because of the difficulty of representing (potentially infinite) sets of configurations, and of manipulating these representations for the analysis of the execution of program statements. For instance, pre/post-condition reasoning requires being able, given pre- and post-conditions ϕ resp. ψ, and a straight-line code P, (1) to compute the (strongest) post-condition of executing P starting from ϕ, denoted $\mathtt{post}(P,\phi)$, and (2) to check that it entails ψ. Therefore, an important issue is to investigate logic-based formalisms where pre/post conditions are expressible for the class of programs under interest, and for which it is possible to compute effectively post-conditions, and to efficiently check the entailment. The latter can be done either using theorem provers, where user-provided tactics are needed to guide the proof system, or using decision procedures, when the given annotations are in a decidable fragment. An essential ingredient in order to scale to large programs is being able to perform compositional reasoning and, in this context, Separation Logic [17] (SL) has emerged as a fundamental approach. Its main tool is the frame rule, which states that if the Hoare triple $\{\phi\}P\{\psi\}$ holds then $\{\phi * \sigma\}P\{\psi * \sigma\}$ also holds (under the condition that P does not alter free variables in σ), where $*$ denotes the separating conjunction. Therefore, when reasoning about P we have to manipulate only specifications for the heap region altered by P.

In this paper, we define a fragment of SL, called *NOLL*, suitable for reasoning about programs that manipulate *overlaid and nested* linked lists, built with an arbitrary set of record fields. The logic *NOLL* is parametrized by a fixed, but arbitrary, set of recursive predicates defined in a higher-order extension of *NOLL* and which are expressive enough to specify various types of (nested) linked lists such as singly-linked lists of cyclic singly-linked lists, where all the elements point to some fixed object.

One of the main features of *NOLL* is that it can be used to perform compositional reasoning for programs that manipulate *overlaid* linked structures, where overlaid means that the structures share sets of objects. Such data structures are used in low-level code to link objects with respect to different aspects. For example, the network monitoring software Nagios (`www.nagios.com`) manipulates hash-tables with closed addressing, implemented as arrays of linked lists, where all the elements in the lists are also linked in the order of their insertion time. Here, we have two data structures which are overlaid, an array of linked lists and a singly-linked list. In order to specify such data structures, we consider, besides the classical operator $*$, that we will call object separating conjunction, a field separating conjunction operator $*_w$. Both operators separate the heap into disjoint regions, the only difference being the interpretation of a heap cell. The $*$ version uses the classical interpretation, where a heap cell corresponds to a heap object. In the $*_w$ version, a heap cell corresponds to a record field from a heap object. Thus, the $*_w$ operator allows to share sets of objects between two data structures as long as they are built over disjoint sets of record fields. In the example above, if $\mathsf{ArrOfSl}$ and Sl are formulas describing the array of lists, resp. the list, then $\mathsf{ArrOfSl} *_w \mathsf{Sl}$ expresses the fact that the two structures share some objects.

However, $*w$ alone is not enough to describe precisely overlaid data structures. In the example above, we would also need to express that the objects of the list described by Sl are exactly *all* the list objects in the array of linked lists; let $\mathsf{Sl\_type}$ be their type. To this, we index each atomic formula specifying list segments by a variable, called a *set of locations variable* and interpreted as the set of all heap objects in the list segment. The values of these new variables can be constrained in a logic that uses classical set operators $\subseteq$ and $\cup$. For example, the specification $\mathsf{ArrOfSl}_\alpha *_w \mathsf{Sl}_\beta \wedge \alpha(\mathsf{Sl\_type}) = \beta$ constrains the set of objects in the linked list to be exactly the set of objects of type $\mathsf{Sl\_type}$ in the array of linked lists. (A *NOLL* formula φ can also put constrains over some set of locations variables, which are not associated to some atomic formula in φ.)

The use of the field separating conjunction for the specification of overlaid data structures enables us to establish another frame rule, which is essential for compositional reasoning: if the Hoare triple $\{\phi\}\ P\ \{\psi\}$ holds then $\{\phi *_w \sigma\}\ P\ \{\psi *_w \sigma\}$ also holds, where P is a straight-line code without `free` statements, P does not alter record fields described by σ, and the atomic formulas in σ may be indexed by the set of locations variables which are not bound to atomic formulas in φ or ψ. The consequences of this frame rule are that, to reason about a program fragment P, one has to provide only specifications for the data structures built with record fields altered by P.

We prove that checking satisfiability of *NOLL* formulas is NP-complete and that the problem of checking entailments between *NOLL* formulas is co-NP complete. The upper bound on the complexity of checking satisfiability/entailment is first proved using a small model argument, and subsequently, following the approach in [8]. The second

proof provides also an effective decision procedure for proving the validity of an entailment $\varphi \Rightarrow \psi$ by (1) computing a normal form for the two formulas and (2) checking the existence of a homomorphism from the graph representation of the normal form of ψ to the graph representation of the normal form of φ. The main advantages of this decision procedure are: (i) by defining a Boolean abstraction for *NOLL* formulas, the construction of the normal form is reduced to (un)satisfiability queries to a SAT solver and (ii) checking the existence of a homomorphism between graph representations of formulas can be done in polynomial time.

To summarize, this work makes the following contributions:

- defines a fragment of SL, called *NOLL*, that can be used to perform compositional reasoning about overlaid and nested linked structures,
- proves that checking satisfiability, resp. entailment, of *NOLL* formulas is NP-complete, resp. co-NP complete,
- defines effective procedures for checking satisfiability and entailment of *NOLL* formulas based on SAT solvers, which are implemented in a prototype tool and proven to be quite efficient in practice.

Related Work: SL has been widely used in the literature for the analysis and the verification of programs with dynamic data structures [1–5, 7, 8, 12, 13, 17, 19].

The *NOLL* fragment incorporates several existing features of SL: the separating conjunction $*$ that operates at a per-object granularity as in [12], a separating conjunction $*_w$ that operates at a per-field granularity as in [6], inductive predicates describing nested linked structures used in [1], and set-valued variables for the memory locations contained in lists are similar to the sequences used in [6]. However, [1, 6] use these features in order to define an abstract domain for the analysis of programs manipulating such data structures. The (partial) order relation on elements of such abstract domains can be seen as a sound, but not complete, decision procedure for entailment.

The works in [2, 5, 8] introduce results concerning the decidability/complexity of the satisfiability/entailment problem in fragments of this logic. Berdine et al. [2] defines a fragment that allows to reason about programs with singly-linked lists and proves that the satisfiability of a formula can be decided in NP and that checking an entailment between two formulas belongs to the co-NP complexity class. A decision procedure for entailments in the same fragment is introduced in [16], which combines SL inference rules with a superposition calculus to deal with (in)equalities between variables. These complexity results were improved in [8] where it is proved that the satisfiability/entailment problem for the previous fragment can be solved in polynomial time. In fact, the procedure for checking entailments of *NOLL* formulas based on normal forms and graph homomorphism is inspired by the work in [8]. The differences are that (a) the procedure for computing the normal form of a *NOLL* formula is based on a new approach that uses Boolean abstractions (the procedure in [8] works only for singly-linked lists and can not be extended to *NOLL*) and (b) the notion of graph homomorphism is extended in order to handle the two versions of the separating conjunction, the constraints on sets of locations variables, and more general recursive predicates.

The (sound) decision procedures for satisfiability/entailment introduced in [18, 15] are also based on Boolean abstractions of formulas. As in our case, the Boolean abstractions are used to transform logical validity into simpler decidable problems. However,

they concern different types of logics: algebraic data types specifications for reasoning about functional programs in [18] and a recursive extension of the first-order logic for reasoning about programs manipulating tree data structures in [15].

Semi-automatic frameworks for reasoning about programs within SL, based on theorem provers, have been defined in [7, 4, 13]. In this paper, we target a completely automatic framework based on decision procedures.

2 Overview

In general, *NOLL* formulas have the form $\Pi \wedge \Sigma \wedge \Lambda$, where Π is the pure part, i.e., a conjunction of equalities and inequalities between program variables expressing aliasing constraints, Σ is the spatial part specifying the data structures and the separation properties, and Λ specifies the sharing constraints between the data structures.

$$\varphi := x \neq \texttt{NULL} \wedge Hash_{\alpha}(x, y, \texttt{NULL}) *_w List_{\beta}(z, \texttt{NULL}) \wedge \alpha(\mathsf{Sl\_type}) = \beta \quad (2.1)$$

$$Hash(in, out, dest) \triangleq (in = out) \vee (\exists u, v.\ (in \mapsto \{(g, u); (h, v)\}) * LowList(v, dest) * Hash(u, out, dest)) \quad (2.2)$$

$$LowList(in, out) \triangleq (in = out) \vee (\exists u.\ in \mapsto \{(s, u)\} * LowList(u, out)) \quad (2.3)$$

$$List(in, out) \triangleq (in = out) \vee (\exists u.\ in \mapsto \{(f, u)\} * List(u, out)) \quad (2.4)$$

Fig. 1: *NOLL* specification of a hash table whose elements are shared with a list.

Examples of *NOLL* formulas: Fig. 1 contains a *NOLL* formula describing a list of lists, using the predicate $Hash_{\alpha}(x, y, \texttt{NULL})$, such that the elements of the nested lists are shared with another list, represented by the predicate $List_{\beta}(z, \texttt{NULL})$. This is an abstraction of the hash table sharing all its elements with a singly-linked list, presented in Sec. 1, in the sense that we use a linked list to represent the array structure.

The predicate $Hash_{\alpha}(in, out, dest)$ has a recursive definition, written in a higher-order extension of *NOLL*: either $in = out$, which means that the nested list segment is empty, or in contains a record field h pointing to an inner singly-linked list ($in \mapsto \{\ldots; (h, v)\} * LowList(v, dest)$) and also a record field g pointing to a new location u ($in \mapsto \{(g, u); \ldots\}$), which is the starting point of another nested list segment. Note that the elements of the lists described by $LowList(v, dest)$ are linked by the record field s. In general, we suppose that variables and record fields are typed. Let $\mathsf{Sl\_type}$ be the type of the variables used in the predicate *LowList*; this implies that all the locations in the nested lists are of type $\mathsf{Sl\_type}$. The use of the object separating conjunction $*$ implies that all the inner lists are disjoint.

The overlapping property is expressed using two features of this logic. The first one is the field separating conjunction operator $*_w$ which allows to share objects but not the record fields in these objects. The second feature is the ability to speak about the set of all locations in a list segment. This set of locations is given by the interpretation of the variable that indexes some recursive predicate, e.g., α in $Hash_{\alpha}(\ldots)$. Then, these variables are constrained in the Λ part of a formula. For example, $\alpha(\mathsf{Sl\_type}) = \beta$ says that all the locations of type $\mathsf{Sl\_type}$ in the list of lists are also present in the list starting in z (β stands for the set of locations in $List_{\beta}(z, \texttt{NULL})$).

A similar data structure is considered in [11] where the elements stored in the hash table are shared between two disjoint linked lists. With the predicates defined in Fig. 1, this data structure is described by the following *NOLL* formula:

$$x \neq \texttt{NULL} \wedge \mathit{Hash}_{\alpha}(x, y, \texttt{NULL}) *_{w} (\mathit{List}_{\beta}(z, \texttt{NULL}) * \mathit{List}_{\gamma}(u, \texttt{NULL})) \wedge \alpha(\mathsf{Sl\_type}) = \beta \cup \gamma,$$

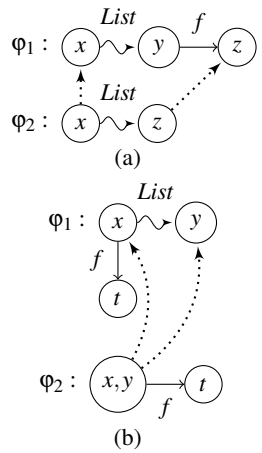


Fig. 2

where $*$ is used to specify the disjointness of the linked lists starting in z and u.

Decision procedure for entailment: We define a procedure for checking entailments of *NOLL* formulas, which is based on the graph homomorphism approach in [8]. The basic idea is to think of formulas as graphs, where nodes represent variables (sets of equal variables) and edges represent spatial constraints, and then, given φ_1 and φ_2 two formulas, if there exists a homomorphism from φ_2 to φ_1 then $\varphi_1 \Rightarrow \varphi_2$ holds. Roughly, the homomorphism is a function mapping each node of φ_2 to a node of φ_1 representing at least the same set of variables. It is required that this function defines a mapping from edges of φ_2 to disjoint paths in φ_1. (Note that the homomorphism is unique.) For example, there exists such a homomorphism from φ_2 to φ_1 in Fig. 2(a), where a snaked edge labeled by *List* from x to y denotes a predicate $\mathit{List}(x,y)$, a straight edge labeled by f from y to z denotes a points-to constraint $y \mapsto \{(f,z)\}$, all spatial constraints are suppose to be separated by $*$, and the dotted edges represent the homomorphism.

In order to be complete, this procedure needs that the formula on the left of an entailment contains the maximum number of equalities and inequalities; in this case, we say that the formula is in *normal form*. Also, if it contains an equality $u = v$ then, it contains no spatial constraint defining a list segment from u to v (as usual in separation logic, $u = v \wedge \mathit{List}(u,v)$ is equivalent to $u = v$). For example, although the entailment $\varphi_1 \Rightarrow \varphi_2$ in Fig. 2(b) holds, there exists no homomorphism from φ_2 to φ_1 (since the record field f is already defined in x, there exists no other non-empty list segment starting in x, and thus, φ_1 implies $x = y$, which shows that $\varphi_1 \Rightarrow \varphi_2$).

Boolean abstractions of *NOLL* formulas: Our first insight in defining such a decision procedure is that the normal form of a *NOLL* formula $\varphi = \Pi \wedge \Sigma \wedge \Lambda$ can be constructed through a boolean abstraction of φ, denoted $F(\varphi)$. For the moment, let us consider the case when $\Lambda = \mathit{true}$. Then, the formula $F(\varphi)$ is defined over a set of boolean variables denoting (in)equalities between variables and atomic formulas from the spatial part Σ.

We illustrate the definition of $F(\varphi)$ on the formula:

$$\varphi := \mathit{List}(x,y) * \mathit{List}(x,z) * y \mapsto \{(f,t)\} * \mathit{List}(y,s). \tag{2.5}$$

The set of boolean variables in $F(\varphi)$ consists of:

- a variable $[u = v]$, for every two variables u and v in φ,
- a variable $[y,t,f]$ to represent the points-to constraint $y \mapsto \{(f,t)\}$, and
- a variable $[\mathit{List}(u,v)]$, for every spatial constraint $\mathit{List}(u,v)$ in φ.

In this case, the formula $F(\varphi) \triangleq F_{eq} \wedge F(\Sigma)$, where:

- F_{eq} expresses the reflexivity and the transitivity of the equality relation, i.e., it is a conjunction between

$$[u = u] \quad \wedge \quad ([u = v] \wedge [v = w]) \Rightarrow [u = w], \qquad \text{for every } u, v, \text{ and } w \text{ in } \varphi.$$

- $F(\Sigma)$ models the spatial part of φ, i.e.,

$$F(\Sigma) \triangleq [y, t, f] \quad \wedge \quad \bigwedge_{u,v} [List(u,v)] \oplus [u = v] \quad \wedge \quad \bigwedge_{A,B \text{ atoms in } \Sigma} F_*(A,B).$$

The first part, $[y,t,f]$, denotes the fact that this points-to constraint must be satisfied by any model of φ and a sub-formula $[List(u,v)] \oplus [u = v]$ denotes the fact that in any model of φ, either $u = v$ or $List(u,v)$ describes a non-empty list segment. The sub-formula $F_*(A,B)$ contains the in(equalities) implied by the use of $*$, i.e,

$$F_*(y \mapsto \{(f,t)\}, List(u,v)) \triangleq \neg[y = u] \vee [u = v], \text{ for any } u, v,$$
$$F_*(List(u_1,v_1), List(u_2,v_2)) \triangleq \neg[u_1 = u_2] \vee [u_1 = v_1] \vee [u_2 = v_2], \text{ for any } u_1, v_1, u_2, v_2$$

In general, the size of $F(\varphi)$ is polynomial in the size of the formula φ. Also, φ is satisfiable iff $F(\varphi)$ is satisfiable.

Computing the normal form: The formula $F(\varphi)$ can be used to compute the normal form of φ since $\varphi \Rightarrow u = v$ iff $F(\varphi) \Rightarrow [u = v]$, for any u and v. Thus, for any valid entailment $F(\varphi) \Rightarrow [u = v]$, the equality $u = v$ is added to φ, and all predicates describing list segments between u and v are removed. For example, the normal form of φ in (2.5) is $y = s \wedge x = z \wedge List(x,y) * y \mapsto \{(f,t)\}$ (the formula $F(\varphi)$ implies $[y = s]$ and $[x = z]$).

Handling sharing constraints: For *NOLL* formulas with sharing constraints, computing the normal form before checking the existence of a graph homomorphism is not enough. Besides (in)equalities, we may have implicit spatial constraints which are not exposed in some formula. Consider the entailment $\varphi_1 \Rightarrow \varphi_2$, where:

$$\varphi_1 := List_\alpha(x,y) *_w LowList_\beta(n,m) \wedge \beta \subseteq \alpha \tag{2.6}$$
$$\varphi_2 := \big(List_\delta(x,n) * List_\gamma(n,y)\big) *_w LowList_\beta(n,m) \wedge \beta \subseteq \delta \cup \gamma \tag{2.7}$$

Note that $\beta \subseteq \alpha$ implies that n is a location on the list segment described by $List_\alpha(x,y)$ and thus $\varphi_1 \Rightarrow \varphi_2$ holds. In this case, $F(\varphi_1)$ includes constraints over a set of boolean variables $[u \in \varepsilon]$ representing the fact that u is a location in the set of locations denoted by ε, for any u and $\varepsilon \in \{\alpha, \beta\}$ (we defer the reader to Sec. 5 for more details).

In general, if the formula $F(\varphi)$ implies $[u \in \varepsilon]$, for some u and ε, then the graph representation of φ includes some additional edges induced by the fact that u is a location on the list segment indexed by ε. In this case, $F(\varphi_1) \Rightarrow [n \in \alpha]$ and the graph representation of φ_1 completed with these additional edges can be found in the middle of Fig. 3. Now, it is easy to see that there exists a homomorphism from G_2 to G_1 (the homomorphism must satisfy additional constraints due to the fact that the newly added edges do not represent list segments separated from all the spatial constraints in the initial formula).

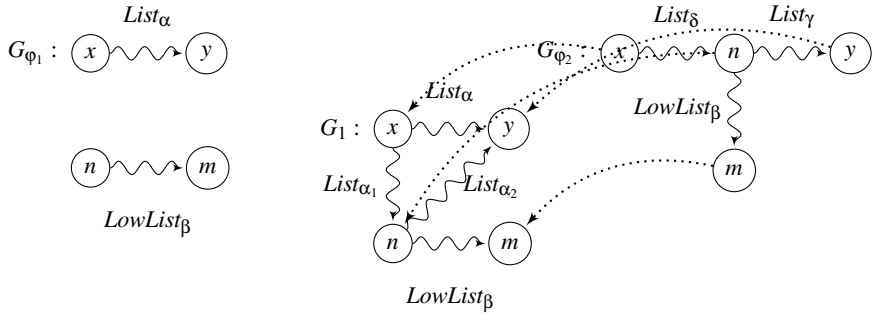


Fig. 3: Homomorphism and graph representations of the *NOLL* formulas in (2.6) and (2.7). G_{φ_1} and G_{φ_2} are the graph representations of the normal forms. G_1 is the completed graph representation of φ_1. Dotted edges represent the homomorphism between the graph representations.

3 Logic *NOLL*

The logic *NOLL* is a multi-sorted fragment of Separation Logic [17]. Let $\mathcal{T}$ be a set of sorts (corresponding to record types defined in the program), *RefFlds* a set of record field names, and τ a typing function mapping each field name into a function type over $\mathcal{T}$. A record field $f \in \mathit{RefFlds}$ is called recursive iff $\tau(f) = R \to R$ with $R \in \mathcal{T}$ and non-recursive, otherwise. The set of recursive record fields is denoted by $\mathit{RefFlds}_{rec}$.
Syntax: Let *LVars* and *SetVars* be two sets of variables, called *location variables* and *set of locations variables*, respectively. We assume that the typing function τ associates a sort, resp. a set of sorts, to every variable in *LVars*, resp. *SetVars*. For simplicity, we assume that *LVars* contains the constant `NULL`. The syntax of *NOLL* is given in Fig. 4.

$x, y, y_i \in \mathit{LVars}$	location variables	$\overrightarrow{z} \in \mathit{LVars}^+$	tuples of location variables
$f, f_i \in \mathit{RefFlds}$	record field names	$\alpha \in \mathit{SetVars}$	set of locations variables
$R \in \mathcal{T}$	sort	$P \in \mathcal{P}$	list segment predicates

$$
\begin{array}{rll}
\varphi & ::= \Pi \wedge \Sigma \wedge \Lambda & \textit{NOLL formula} \\
\Pi & ::= \mathit{true} \mid x \neq y \mid x = y \mid \Pi \wedge \Pi & \text{pure constraints} \\
\Sigma & ::= \mathit{emp} \mid x \mapsto \{(f_1, y_1); \ldots; (f_k, y_k)\} \mid P_\alpha(x, y, \overrightarrow{z}) \mid \Sigma * \Sigma \mid \Sigma *_w \Sigma & \text{spatial constraints} \\
\Lambda & ::= \mathit{true} \mid t \subseteq t' \mid x \in t \mid x \notin t \mid \Lambda \wedge \Lambda & \text{sharing constraints} \\
t & ::= \{x\} \mid \alpha \mid \alpha(R) \mid t \cup t' & \text{set of locations terms}
\end{array}
$$

Fig. 4: Syntax of *NOLL* formulas.

An atomic *points-to constraint* $x \mapsto \{(f_1, y_1); \ldots; (f_k, y_k)\}$ is used to specify the values of record fields $f_1, \ldots, f_k$ in the location denoted by x: the value stored by the field f_i is y_i, for all $1 \leq i \leq k$. The fields shall be pairwise disjoint and the formula shall be well typed, i.e., for any f_i, $\tau(f_i) = \tau(x) \to \tau(y_i)$.

In every *list segment constraint* $P_\alpha(x, y, \overrightarrow{z})$, P is a predicate from a fixed, but arbitrary, set $\mathcal{P}$. The predicates in $\mathcal{P}$ have recursive definitions with the following syntax:

$$P(in, out, \overrightarrow{nhb}) \triangleq (in = out) \vee (\exists u, \overrightarrow{v}. \Sigma_0(in, u \cup \overrightarrow{v} \cup \overrightarrow{nhb}) * \Sigma_1(\overrightarrow{v}, \overrightarrow{nhb}) * P(u, out, \overrightarrow{nhb}))$$

$$\Sigma_0(in, V) ::= in \mapsto \theta, \text{ where } \theta \subseteq \{(f, w) \mid f \in RefFlds, w \in V\}$$

$$\Sigma_1(\overrightarrow{v}, \overrightarrow{nhb}) ::= emp \mid Q(v, b, \overrightarrow{b}) \mid \Sigma_1(\overrightarrow{v}, \overrightarrow{nhb}) * \Sigma_1(\overrightarrow{v}, \overrightarrow{nhb}) \text{ with } b, \overrightarrow{b} \subseteq \overrightarrow{nhb}, \text{ and } Q \in \mathcal{P}$$

where in, out, u and $\overrightarrow{nhb}$, $\overrightarrow{v}$, $\overrightarrow{b}$ are location variables, resp. tuples of location variables.

We add some typing constraints in the definition of every $P \in \mathcal{P}$, i.e., $\tau(in) = \tau(out) = \tau(u)$, and $\tau(in) \neq \tau(v)$, for every $v \in \overrightarrow{v}$, in order to ensure bounded nesting.

A predicate $P(in, out, \overrightarrow{nhb})$ defines possibly empty list segments starting from *in* and ending in *out*. The record fields of each element in this list segment are defined by Σ_0 while the nested lists to which it points to are defined by Σ_1. The parameters $\overrightarrow{nhb}$ 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 $out \cup \overrightarrow{nhb}$ (this path may be defined by more than one record field). Every element of the list segment described by P points to several nested lists, each one of them being described by a predicate Q in $\mathcal{P}$. The use of the object separating conjunction $*$ in the definition of P implies that the inner list segments are disjoint.

For simplicity of the presentation, we have restricted ourselves to such recursive definitions, which are not expressive enough to describe doubly-linked lists or nested lists containing cyclic lists on their inner levels. However, our techniques can be extended to cover such cases. For example, to describe doubly-linked lists, one must allow further points-to constraints and use a special type of existential variables representing the next to last location in a doubly-linked list segment like, e.g., in [1].

We assume that the recursive definitions of the predicates in $\mathcal{P}$ are well typed and also, that they are not cyclic or mutually recursive. For any predicate P, $\Sigma_0(P)$, resp. $\Sigma_1(P)$, denotes the sub-formula Σ_0, resp. Σ_1 of P. Moreover, $RefFlds_0(P)$ denotes the set of record fields of *in* that point to u according to the formula $\Sigma_0(P)$, i.e., $f \in RefFlds_0(P)$ iff $\Sigma_0(P) = in \mapsto \theta$ and $(f, u) \in \theta$.

In every spatial constraint $P_\alpha(x, y, \overrightarrow{z})$, α is a set of locations variable, which is said to *be bounded to* or to *index* the spatial constraint. Note that Λ may contain set of locations variables which are not bounded to some spatial constraint. For simplicity, we assume that a variable in *SetVars* appears in Σ at most once. Also, we consider that all atomic constraints in Λ are well typed, i.e., for any $t \subseteq t'$ in Λ, $\tau(t) \subseteq \tau(t')$ and for any $(x \in t)$ in Λ, $\tau(x) \in \tau(t)$, where τ is extended to set of locations terms as usual.

In the following, we denote by $LVars(\varphi)$ (and $SetVars(\varphi)$) the set of location variables (resp. set of locations variables) used in φ. The set $atoms(\varphi)$ denote the set of atomic formulas in φ. Also, two atoms in Σ are *object separated*, resp. *field separated*, if their least common ancestor in the syntactic tree of φ is $*$, resp. $*_w$.

Semantics: Let *Loc* be a sorted set of *locations* (the typing function τ is extended also to locations in *Loc*). A *program heap* is modeled by a pair $C = (S, H)$, where $S : LVars \to Loc$ maps location variables to locations in *Loc* and $H : Loc \times RefFlds \rightharpoonup Loc$ defines values of record fields for a subset of locations. Intuitively, each allocated object is denoted by a location in *Loc* and then, H defines the record fields for the allocated objects and S gives for each variable, the object it points to.

$(C,J) \models \varphi_1 \wedge \varphi_2$ iff $(C,J) \models \varphi_1$ and $(C,J) \models \varphi_2$

$(C,J) \models x = y$ iff $S(x) = S(y)$

$(C,J) \models x \mapsto \cup_{i \in I}\{(f_i, y_i)\}$ iff $H(S(x), f_i) = S(y_i)$ for all $i \in I$

$(C,J) \models P_\alpha(x, y, \overrightarrow{z})$ iff there exists $k \in \mathbb{N}$ s.t. $(C,J) \models P^k_\alpha(x, y, \overrightarrow{z})$

$(C,J) \models P^0_\alpha(x, y, \overrightarrow{z})$ iff $S(x) = S(y)$ and $J(\alpha) = \emptyset$

$(C,J) \models P^{k+1}_\alpha(x, y, \overrightarrow{z})$ iff $S(x) \neq S(y)$ and there exists $\rho : \{u\} \cup \overrightarrow{v} \to Loc$ and $J' : SetVars \to 2^{Loc}$ s.t.
$(C[S \mapsto S \cup \rho], J') \models \Sigma_0(x, u \cup \overrightarrow{v} \cup \overrightarrow{z}) * \Sigma_1(\overrightarrow{v}, \overrightarrow{z}) * P^k_\alpha(u, y, \overrightarrow{z})$,
$\mathtt{img}(\rho) \cap \mathtt{img}(S) = \emptyset$,
$J'(\alpha) = J(\alpha) \setminus (\{S(x)\} \cup \rho(\overrightarrow{v}))$, and $J'(\beta) = J(\beta)$, for any $\beta \neq \alpha$

$(C,J) \models \Sigma_1 * \Sigma_2$ iff there exist program heaps C_1 and C_2 s.t. $C = C_1 * C_2$,
$(C_1,J) \models \Sigma_1$, and $(C_2,J) \models \Sigma_2$

$(C,J) \models \Sigma_1 *_w \Sigma_2$ iff there exist program heaps C_1 and C_2 s.t. $C = C_1 *_w C_2$,
$(C_1,J) \models \Sigma_1$, and $(C_2,J) \models \Sigma_2$

$(C,J) \models x \in t$ iff $S(x) \in [t]_J$

$(C,J) \models t \subseteq t'$ iff $[t]_J \subseteq [t']_J$

Separation operators over program heaps:

$C = C' * C''$ iff $Loc(C) = Loc(C') \cup Loc(C'')$ and $Loc(C') \cap Loc(C'') = \varnothing$,
$S^{C'} = S^C|_{Loc(C')}$ and $S^{C''} = S^C|_{Loc(C'')}$

$C = C' *_w C''$ iff $\mathtt{dom}(H^C) = \mathtt{dom}(H^{C'}) \cup \mathtt{dom}(H^{C''})$ and $\mathtt{dom}(H^{C'}) \cap \mathtt{dom}(H^{C''}) = \varnothing$,
$S^{C'} = S^C|_{Loc(C')}$ and $S^{C''} = S^C|_{Loc(C'')}$

Fig. 5: Semantics of *NOLL* formulas ($\mathtt{dom}(F)$ denotes the domain of the function F and $S \cup \rho$ denotes a new mapping $K : \mathtt{dom}(S) \cup \mathtt{dom}(\rho) \to Loc$ s.t. $K(x) = \rho(x),\ \forall x \in \mathtt{dom}(\rho)$ and $K(y) = S(y),\ \forall y \in \mathtt{dom}(S)$).

Formulae are interpreted over *NOLL* interpretations, which are pairs (C,J), where $C = (S,H)$ is a program heap and $J : SetVars \to 2^{Loc}$ interprets variables in *SetVars* to finite subsets of *Loc*.

We assume that S, H, and J are well-typed w.r.t. τ. Let Loc_R denote the set of locations in *Loc* of sort R. Given a program heap $C = (S,H)$, the set of locations in C, denoted by $Loc(C)$, is the set of locations $l \in Loc$ for which there exists $f \in RefFlds$ s.t. $H(l,f)$ is defined. The component S, resp. H, of a heap C is denoted by S^C, resp. H^C.

A *NOLL* intepretation (C,J) is a model of a formula φ iff $(C,J) \models \varphi$, where $\models$ is defined in Fig. 5 for its non trivial cases. For simplicity, we consider the intuitionistic semantics of SL [17]: if a formula is true on a model then it remains true for any extension of that model with more locations. Our techniques can be adapted to work also for the non-intuitionistic semantics [10]. The interpretation of a term t in Λ w.r.t. J, denoted by $[t]_J$, is defined as usual: $[\{x\}]_J = \{S(x)\}$, $[\alpha]_J = J(\alpha)$, $[\alpha(R)]_J = J(\alpha) \cap Loc_R$, and $[t \cup t']_J = [t]_J \cup [t']_J$.

Note the difference between the two kinds of separation of heaps: $C = C' * C''$ holds iff the set of locations corresponding to allocated objects in C' and C'' are disjoint while $C = C' *_w C''$ holds iff the domains of the H component in C' and C'' are disjoint.

In the following, w.l.o.g. we suppose that the sharing constraints in Λ are of the form $\alpha \subseteq t$, where t contains at most two set of locations variables. Also, for any atomic formula $\alpha \subseteq t$ in Λ such that α is bound to some spatial constraint $P_\alpha(x, y, \overrightarrow{z})$, we remove from t (1) all the variables α' such that α and α' are bound to object separated

spatial constraints and (2) all the terms of the form $\{x\}$ such that φ contains a points-to constraint $x \mapsto \theta$, which is object separated from the spatial constraint indexed by α. If t becomes empty then, the equality $x = y$ is added to φ.

We denote by $[\varphi]$ the set of pairs (C,J) which are models of φ. The entailment between two *NOLL* formulas is denoted by $\Rightarrow$ and it is defined by $\varphi \Rightarrow \psi$ iff $[\varphi] \subseteq [\psi]$.

Fragment *MOLL*: The fragment of *NOLL* which does not allow the nesting of list segment predicates is denoted by *MOLL*. It allows to specify overlaid multi-linked lists (it is also possible to say that all the elements of a list segment point to some fixed location). We will use this fragment to illustrate some of the constructions in this paper. Formally, the fragment *MOLL* contains all the *NOLL* formulas defined over a set of predicates $\mathcal{P}$ such that, for any $P \in \mathcal{P}$, $\Sigma_1(P) = emp$. (i.e., P is defined by $P(in, out, \overrightarrow{nhb}) \triangleq (in = out) \vee (\exists u. \Sigma_0(in, u \cup \overrightarrow{nhb}) * P(u, out, \overrightarrow{nhb}))$).

4 A model-theoretic procedure for checking entailment

We prove that satisfiability, resp. entailment checking, of *NOLL* formulas is NP-complete, resp. co-NP complete. The upper bound for the complexity of satisfiability is proved using a small model property: if $\varphi \in$ *NOLL* has a model, then it has also a model of size polynomial in the size of φ and $\mathcal{P}$ (the size of $\mathcal{P}$ is defined as the size of all recursive definitions for predicates in $\mathcal{P}$). The co-NP upper bound for entailment checking is obtained by proving a small model property for formulas of the form $\varphi \not\Rightarrow \psi$ (a model for this formula corresponds to a counter-example for $\varphi \Rightarrow \psi$).

4.1 Satisfiability problem

The NP lower bound of the satisfiability problem for *NOLL* formulas is given by the next theorem. The proof is based on a reduction of 3SAT, the satisfiability problem for CNF formulas with 3 literals in each clause, to the satisfiability problem of *MOLL* formulas. The proof of this result is detailed in Appendix A.1.

Theorem 1. *The satisfiability problem for NOLL (MOLL) is NP-hard.*

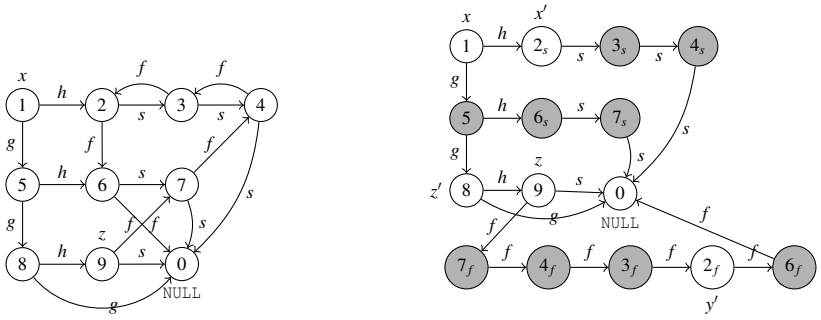
The small model property for the NP upper bound uses an abstraction of the models of *NOLL* formulas by *colored heap graphs*, where a node represents a set of record fields defined at some location. This is useful for collapsing list segments described using spatial constraints connected by $*_w$, which share locations but not record fields.

Intuitively, a model (C,J) of a *NOLL* formula is represented by a colored graph where each location ℓ from C is split into a set of graph nodes V_ℓ. V_ℓ is a singleton (i.e., ℓ is not split) when ℓ is the interpretation of a location variable or it is not shared between list segments described in φ. Otherwise, each node in V_ℓ represents a set of record fields at location ℓ such that two nodes in V_ℓ represent disjoint sets of fields. All nodes in V_ℓ are colored by ℓ and are called *sibling nodes*. The abstraction is built such that the sub-graphs corresponding to list segments defined using different predicates share only nodes which are interpretations of location variables. A node in this graph, which is not colored by the interpretation of a location variable is called *anonymous*.

We show that for any model (C,J), one can identify a set of anonymous nodes, whose size is polynomial in the size of φ and $\mathcal{P}$, called *crucial nodes*, such that by collapsing all the non-crucial anonymous nodes one can still obtain a model of φ. Formally,

Definition 1 (Colored heap graph). *A* colored heap graph *over LVars, RefFlds, and SetVars is a tuple $G = (V,E,\mathcal{P},\mathcal{L},\mathcal{S})$, where (1) V is a finite set of nodes, (2) $E : V \times RefFlds \rightharpoonup V$ is a set of edges, (3) $\mathcal{P} : LVars(\varphi) \to V$ is a labeling of nodes with location variables, (4) $\mathcal{L} : V \to Loc$ is a coloring of nodes with locations, and (5) $\mathcal{S} : SetVars \to 2^V$ is an interpretation of variables in SetVars to sets of nodes.*

Fig. 6 pictures a model of φ in (2.1) and its colored heap graph abstraction. We denote the components of a colored heap graph G using superscripts, e.g., the component V of G is denoted by V^G. The semantics of *NOLL* formulas on colored heap graphs is defined similarly to the one on *NOLL* interpretations, except for the $*$ operator and the constraints in Λ. A colored heap graph G satisfies a formula $\varphi_1 * \varphi_2$ iff G can be split into two disjoint graphs G_1 and G_2 such that $G_1 \models \varphi_1$, $G_2 \models \varphi_2$, and for any two nodes $v_1 \in V^{G_1}$ and $v_2 \in V^{G_2}$, $\mathcal{L}^{G_1}(v_1) \neq \mathcal{L}^{G_2}(v_2)$. Also, for any constraint $P_\alpha(x,y,\overrightarrow{z})$, $\mathcal{S}(\alpha)$ is interpreted as the union of $\mathcal{L}(v)$, for all nodes v in the unique subgraph defined by P_α.



(a) (b)

Fig. 6: A program heap satisfying φ in (2.1) and its colored heap graph. For any $0 \leq n \leq 9$, the nodes n_s and n_f in (b) are colored by the location n from (a). Primed variables x', y', z' label crucial nodes. A small model is obtained by collapsing filled nodes in (b).

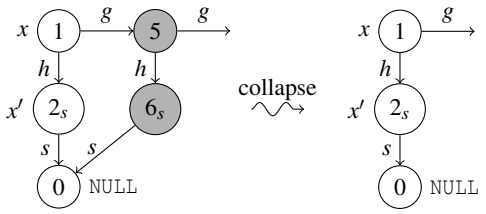
Lemma 1. *If a NOLL formula φ has a model (C,J) then it also has a model (C_s,J_s) of size polynomial in the size of φ and $\mathcal{P}$.*

Proof. (Idea) The proof builds a small model following the steps given in Fig. 7a. Roughly, we show that anonymous locations from (C,J) can be collapsed until the list segments are of bounded length. The bounds are determined by the sharing constraints and the levels of nesting in the definition of the recursive predicates. To collapse anonymous locations on list segments, we use the colored graph abstraction. However, some distinguished set of *crucial* anonymous nodes shall not be collapsed because this will invalidate spatial or sharing constraints in φ (an example is shown below). Also, to preserve the truth value of sharing constraints, if a node is found crucial on some list segment, then all its sibling nodes are also marked as crucial (this corresponds to the fact that the small model contains all the record fields for that location).

The procedure purify removes from (C,J) all the locations not involved in spatial constraints from φ. This is possible because the minimal fragment of C satisfying some spatial constraint is unique. splitLocations builds the colored heap graph abstraction of (C',J') by splitting the nodes not labeled by location variables but shared between several list segments described by predicates in φ. An example is given in Fig. 6.

1: $(C',J') := \mathsf{purify}(\varphi)(C,J)$
2: $G := \mathsf{splitLocations}(C',J')$
3: $V' := \mathsf{crucialNodes}(\varphi,G)$
4: $G' := \mathsf{labelCrucial}(G,V')$
5: $G'' := \mathsf{collapseAnonymous}(G')$
6: $(C_s,J_s) := \mathsf{mergeNodes}(G'')$

(a) Steps for computing a small model.



(b) Example of collapsing.

Fig. 7: Computing a small model for *NOLL* formulas.

The set of crucial nodes is computed by crucialNodes as the closure under the sibling relation of the set of (anonymous) nodes in G which are either (1) the successor of a labeled node by a non recursive record field (e.g., node 2_s in Fig. 6), or (2) the source or the target of a non recursive record field on a witness path between two nodes labeled by location variables (e.g., node 8 in Fig. 6). Because the nesting of recursive predicates is bounded, the size of the set V' is bounded by a polynomial in the size of φ and $\mathcal{P}$ (the number of variables, the nesting depth, and the size of *RefFlds*). The crucial nodes are labeled with a set of additional location variables $LVars'$ in labelCrucial.

Afterwards, the anonymous nodes (not labeled by variables in $LVars(\varphi) \cup LVars'$) are collapsed by collapseAnonymous in a bottom up manner, i.e., starting from the inner list segments to the upper ones. Roughly, the collapsing removes a node (and the sub-graph representing the nested, anonymous structure) if it is between two recursive record fields (see Fig. 7b). Intuitively, this process preserves a model of φ because no edges are added and the nodes marked as important for the satisfaction of the spatial and sharing constraints are kept. Due to the special syntax of predicates in $\mathcal{P}$, we can compute for each list segment the minimal number of anonymous nodes that must be preserved in order to satisfy some given spatial constraint. This number depends only on the size of $\mathcal{P}$ and it is obtained when all the spatial constraints in the predicate definition are interpreted as list segments of length one. Thus, we obtain a colored heap graph G'' where all labeled nodes are preserved and with them some sub-graphs with a bounded number of anonymous nodes. Finally, from G'', a model (C_s,J_s) of φ is built, by applying mergeNodes, which roughly merges sibling nodes in locations. □

Since the complexity of the model-checking problem for *NOLL* formulas is polynomial, the following result holds.

Theorem 2. *The satisfiability problem for NOLL is NP-complete.*

4.2 Entailment problem

The colored heap graph abstraction is also used to prove a small counter-example property for entailments $\varphi \Rightarrow \psi$ when φ and ψ are in *NOLL*. The proof is similar to the proof of Lemma 1, with two main differences. Let (C,J) be a counter-example for $\varphi \Rightarrow \psi$. First, in purify, the locations not used in φ are removed from (C,J) except for locations that are witnesses for some unsatisfied sharing constraint in ψ. It is enough to keep one location per sharing constraint in ψ and thus, their number is bounded by the size of ψ. We label these locations with variables from some set $LVars''$. Second, crucialNodes marks some additional nodes as crucial, in order to keep track if two list segments are sharing at least one location and in order to distinguish between list segments of size 1 and list segments of size at least 2. However, this will only keep at most one more node per constraint, and thus the bound on the number of nodes is increased by a linear term in the size of φ and ψ. This property and the NP-completeness of satisfiability imply:

Theorem 3. *Checking the validity of an entailment between two NOLL formulas is co-NP complete.*

5 Computing the normal form

This section makes a first step towards the effective procedure for checking entailments of *NOLL* formulas by presenting the procedure for computing the normal form of a *NOLL* formula. We say that a *NOLL* formula is in *normal form* if it contains the maximum set of equalities and disequalities between location variables and the minimum set of list segment constraints. Formally,

Definition 2 (Normal form). *A NOLL formula $\varphi = \Pi \wedge \Sigma \wedge \Lambda$ is in normal form iff:*

- *for any $x,y \in LVars(\varphi)$, if $\varphi \Rightarrow x = y$, resp. $\varphi \Rightarrow x \neq y$, then Π contains the atom $x = y$, resp. $x \neq y$, and*
- *for any atomic formula $P_\alpha(x,y,\overrightarrow{z})$ in Σ, there exists a model (C,J) of φ such that $S^C(x) \neq S^C(y)$.*

The normal form *of φ is a formula φ' in normal form and equivalent to φ.*

We now describe the main ideas behind the procedure that computes the normal form and to this, we must define the class of reduced, explicit *NOLL* formulas.

A *NOLL* formula is called *explicit* if it contains $x = y$ or $x \neq y$, for any constraint $P_\alpha(x,y,\overrightarrow{z})$ in φ, and $x \in \alpha$ or $x \notin \alpha$, for any x and α in φ. Then, an explicit formula ψ is called *reduced* if it does not contain both the atoms $x = y$ and $P_\alpha(x,y,\overrightarrow{z})$.

Note that any formula $\varphi \in NOLL$ is equivalent to a disjunction of reduced, explicit formulas $\psi_1 \vee \ldots \vee \psi_n$. The formulas ψ_i are obtained from φ by (1) adding in all possible ways atoms $x = y$, $x \neq y$, $x \in \alpha$, and $x \notin \alpha$ until the obtained formula is explicit and then, (2) if a formula contains $x = y$, by removing atoms $P_\alpha(x,y,\overrightarrow{z})$ together with all occurrences of α in the sharing constraints (e.g., every atom $x \in \alpha$ or $\beta \subseteq \alpha$, where β indexes a constraint $Q_\beta(u,v,\overrightarrow{w})$ and $u \neq v$ belongs to the formula, is replaced by *false*).

The disjunction of reduced, explicit formulas can be used to compute the normal form of φ as follows. An atom $x = y$ or $x \neq y$ is implied by φ iff this atom is included in

all the *satisfiable* formulas in this disjunction. Also, for any $P(x,y,\overrightarrow{z})$ in φ, there exists a model (C,J) of φ s.t. $S^C(x) \neq S^C(y)$ iff this atom is included in some satisfiable ψ_i.

In general, the number of satisfiable formulas in the disjunction $\psi_1 \vee \ldots \vee \psi_n$ above may be exponential w.r.t. the size of φ. However, all these formulas can be represented symbolically as the satisfying assignments of a boolean formula, denoted by $F(\varphi)$.

In order to simplify the presentation, we present the construction of $F(\varphi)$ only for *MOLL* formulas where variables are of the same type. (Appendix A.4 explains the general case.) The formula $F(\varphi)$ is defined over the set of boolean variables $BVars(F(\varphi))$ defined in Table 1. This set consists of a set of variables, which represent spatial atomic formulas in φ, together with a set of variables that represent equalities $x = y$ and membership constraints of the form $x \in \alpha$ (which are not required to be already in φ).

$[x = y]$	for every $x, y \in LVars(\varphi)$
$[x, y, f]$	for every sub-formula $x \mapsto \theta$ of φ with $(f, y) \in \theta$
$[P_\alpha(x, y, \overrightarrow{z})]$	for every sub-formula $P_\alpha(x, y, \overrightarrow{z})$ of φ
$[x \in \alpha]$	for every $x \in LVars$ and $\alpha \in SetVars$ variables in φ

Table 1: Definition of the set $BVars(F(\varphi))$ of boolean variables used in $F(\varphi)$.

Given a satisfying assignment $\sigma : BVars(F(\varphi)) \to \{0,1\}$ for $F(\varphi)$ such that $\sigma([x,y,f]) = 1$, for any $[x,y,f] \in BVars(F(\varphi))$, we define the *NOLL* formula ψ_σ to be φ to which the following transformations are applied:

- if $\sigma([x = y])$ is 0, resp. 1, then ψ_σ includes the pure constraint $x \neq y$, resp. $x = y$,
- if $\sigma([P_\alpha(x, y, \overrightarrow{z})]) = 0$ then $P_\alpha(x, y, \overrightarrow{z})$ and α are removed from φ
- if $\sigma([x \in \alpha])$ is 0, resp. 1, then $x \notin \alpha$, resp. $x \in \alpha$, is added to ψ_σ.

Let $\varphi = \Pi \wedge \Sigma \wedge \Lambda$ be a *NOLL* formula. The formula $F(\varphi)$ is defined by:

$$F(\varphi) = F(\Pi) \wedge F_{eq} \wedge F(\Sigma) \wedge F_{det} \wedge F(\Lambda) \wedge F_{\in}, \tag{5.1}$$

where $F(\Pi)$, $F(\Sigma)$, and $F(\Lambda)$ encode the semantics of the atomic formulas of φ, F_{eq} encodes the reflexivity and the transitivity of the equality relation in Π, F_{det} encodes the semantics of the field separating conjunction, and $F_\in$ encodes the properties of the membership relation $\in$. These sub-formulas are defined inductively on the syntax of *NOLL* formulas. The full definition is given in Appendix A.3. Most of them are not difficult to follow. We provide here some intuition for the most interesting ones.

A list segment constraint $P_\alpha(x, y, \overrightarrow{z})$ in φ is translated into $F(P_\alpha(x, y, \overrightarrow{z})) = [P_\alpha(x, y, \overrightarrow{z})] \oplus [x = y]$, where $\oplus$ is the exclusive or. This expresses the fact that the atom is kept in a reduced, explicit *NOLL* formula only if its endpoints are not equal.

The separation of record fields (defined for locations which are interpretations of location variables) induced by the use of the field separating conjunction is expressed in the formula F_{det} in Table 2. Thus, F_{det} states that for any location variable x and any record field $f \in RefFlds$, at most one of the following conditions is true:

1. the reduced, explicit formula contains the equality $x = x'$ and a points-to constraint $x' \mapsto \theta$ such that $(f, y) \in \theta$, for some y,

$$
\begin{aligned}
F_{det} = & \bigwedge \text{ for any } [x_1,y_1,f],[x_2,y_2,f] \in \mathit{BVars}(F(\varphi)) \text{ different variables} \\
& \quad [x_1 = x_2] \Rightarrow [x_1,y_1,f] \oplus [x_2,y_2,f] && (5.2) \\
& \bigwedge \text{ for any } [x_1,y_1,f],[P_\alpha(x_2,y_2,\overrightarrow{z_2})] \in \mathit{BVars}(F(\varphi)) \text{ s.t. } f \in \mathit{RefFlds}_0(P) \text{ and } x \in \mathit{LVars}(\varphi) \\
& \quad [x_1 = x] \wedge [x \in \alpha] \Rightarrow [x_1,y_1,f] \oplus [P_\alpha(x_2,y_2,\overrightarrow{z_2})] && (5.3) \\
& \bigwedge \text{ for any } [P_\alpha(x_1,y_1,\overrightarrow{z_1})],[Q_\beta(x_2,y_2,\overrightarrow{z_2})] \in \mathit{BVars}(F(\varphi)) \text{ different variables} \\
& \quad \text{s.t. } \mathit{RefFlds}_0(P) \cap \mathit{RefFlds}_0(Q) \neq \emptyset \text{ and } x,x' \in \mathit{LVars}(\varphi) \\
& \quad [x \in \alpha] \wedge [x' \in \beta] \wedge [x = x'] \Rightarrow [P_\alpha(x_1,y_1,\overrightarrow{z_1})] \oplus [Q_\beta(x_2,y_2,\overrightarrow{z_2})] && (5.4)
\end{aligned}
$$

Table 2: Definition of F_{det} for an *MOLL* formula $\varphi = \Pi \wedge \Sigma \wedge \Lambda$.

2. the reduced, explicit formula contains the atoms $x \in \alpha$ and $P_\alpha(x', y, \overrightarrow{z})$ (therefore it also includes $x' \neq y$), for some y and $\overrightarrow{z}$, such that $f \in \mathit{RefFlds}_0(P_\alpha)$.

The main definitions of $F(\Lambda)$ are given in Table 3. For instance, $F(x \in \alpha_1)$ in eq. (5.6) states that the boolean variable $[x \in \alpha_1]$ is true and that the list segment bound to α_1 in φ, if any, is non empty. In eq. (5.7), $F(\alpha_1 \subseteq \alpha_2)$ expresses the fact that (1) if there exists some variable x such that $x \in \alpha_1$ is true then $x \in \alpha_2$ also holds and (2) if α_1 is the index of a list segment constraint $P_{\alpha_1}(x_1, y_1, \overrightarrow{z})$ in φ, which is interpreted to a non-empty list segment, then the left end of this list segment, i.e., x_1, belongs to α_2.

$$
\begin{aligned}
F(x \in \bigcup_{1 \leq i \leq n} \{u_i\}) &= \bigvee_{1 \leq i \leq n} [x = u_i] && (5.5) \\
F(x \in \alpha_1) &= \begin{cases} [x \in \alpha_1] \wedge [P_{\alpha_1}(x_1,y_1,\overrightarrow{z})] & \text{, if } P_{\alpha_1}(x_1,y_1,\overrightarrow{z}) \text{ in } \varphi \\ [x \in \alpha_1] & \text{, otherwise} \end{cases} && (5.6) \\
F(\alpha_1 \subseteq \alpha_2) &= \bigwedge_{x \in \mathit{LVars}(\varphi)} [x \in \alpha_1] \Rightarrow [x \in \alpha_2] && (5.7) \\
& \quad \wedge \begin{cases} [P_{\alpha_1}(x_1,y_1,\overrightarrow{z})] \Rightarrow F(x_1 \in \alpha_2) & \text{, if } P_{\alpha_1}(x_1,y_1,\overrightarrow{z}) \text{ in } \varphi \\ 1 & \text{, otherwise} \end{cases}
\end{aligned}
$$

Table 3: Main definitions of $F(\Lambda)$ for an *MOLL* formula $\varphi = \Pi \wedge \Sigma \wedge \Lambda$.

Proposition 1. *Let φ be a formula. For any satisfying assignment σ of $F(\varphi)$, ψ_σ is an explicit, reduced, and satisfiable formula. Also, φ is equivalent to the disjunction of ψ_σ, for all satisfying assignments σ of $F(\varphi)$. $F(\varphi)$ is of size polynomial in the size of φ.*

Theorem 4. *The problem of computing the normal form of a formula φ is in co-NP.*

Proof. To compute the maximum set of (in)equalities that should be included in the normal form of φ, we iterate over every pair of location variables x, y in φ and check if $F(\varphi) \Rightarrow [x = y]$ or $F(\varphi) \Rightarrow \neg[x = y]$ is valid. In the first (resp., second) case, $x = y$ (resp., $x \neq y$) is included in the normal form. When some equality $x = y$ is added to the normal form, the atoms $P_\alpha(x, y, \overrightarrow{z})$ in φ are removed, and all occurrences of α are interpreted as the empty set. Since we need to perform a polynomial number of Boolean formula validity tests, the overall complexity of this procedure is co-*NP* time. □

6 An effective procedure for checking entailment

We present a procedure for checking entailments $\varphi \Rightarrow \psi$ between *NOLL* formulas, that (1) computes the normal form of φ and ψ, denoted by φ' and ψ', respectively, (2) computes additional spatial constraints, which are implied by φ, and (3) checks if the graph representation of ψ' is *homomorphic* to the graph representation of both φ' and the additional constraints computed in the previous step.

In the following, we first described the second step above, then we define graph representations for *NOLL* formulas, called (complete) *NOLL* graphs, and finally, we define the notion of homomorphism between *NOLL* graphs. *Moreover, we assume that* φ *and* ψ *are satisfiable.* Otherwise, Proposition 1 implies that a *NOLL* formula φ is satisfiable iff $F(\varphi)$ is satisfiable, which allows to decide in co-NP time entailments of the form $\varphi \Rightarrow \psi$ when φ or ψ is unsatisfiable.

6.1 Inferring additional spatial constraints

In order to give an intuition about the additional spatial constraints deduced from φ, recall the entailment $\varphi_1 \Rightarrow \varphi_2$, where φ_1 and φ_2 are defined in eq. (2.6) resp. (2.7) at page 6. The entailment holds because the list segments linking x to n and n to y, and described by $\mathit{List}_\delta(x,n) * \mathit{List}_\gamma(n,y)$, exist in every model of φ_1. To obtain a complete decision procedure for entailment, such constraints must be made explicit before checking the existence of a homomorphism between the two formulas viewed as graphs.

Remark 1. Note that φ_1 does not imply $\varphi_1 *_w \big(\mathit{List}_\delta(x,n) * \mathit{List}_\gamma(n,y)\big)$ but, $\varphi_1 \wedge \big(\mathit{List}_\delta(x,n) * \mathit{List}_\gamma(n,y)\big)$, where the semantics of $\wedge$ between *NOLL* formulas is defined as usual, i.e., $(C,J) \models \varphi_1 \wedge \varphi_2$ iff $(C,J) \models \varphi_1$ and $(C,J) \models \varphi_2$, for any (C,J), φ_1, and φ_2. Thus, these implicit constraints will be added only to the graph representation of *NOLL* formulas and not to the formula itself, as explained in the next section.

For simplicity, we consider only formulas φ in *MOLL*. Let ξ be a set of atoms in φ of the form $Q_\beta(u,v,\overrightarrow{w})$. For any such ξ, $\mathcal{P}(\xi)$ denotes the set of recursive predicates in ξ, $\mathit{SetVars}(\xi)$ denotes the set of variables $\beta \in \mathit{SetVars}$ bounded to atoms in ξ, and t_ξ is the term defined as the union of all variables in $\mathit{SetVars}(\xi)$.

An atom $P_\alpha(x,y,\overrightarrow{z})$ is called *implicit in* ξ iff one of the following holds:

- ξ consists of one atom $P_\beta(u,v,\overrightarrow{w})$, $\varphi \Rightarrow y \in \beta$, and $\varphi \Rightarrow x = u$;
- (1) $\varphi \Rightarrow x \in t_\xi$, (2) t_ξ is a minimal term t such that $\varphi \Rightarrow x \in t$, i.e., for every other term t', which is the union of the variables from a strict subset of $\mathit{SetVars}(\xi)$, $\varphi \not\Rightarrow x \in t'$, (3) $\mathit{RefFlds}_0(P) = \bigcap_{Q \in \mathcal{P}(\xi)} \mathit{RefFlds}_0(Q)$, and (4) $\varphi \Rightarrow \bigwedge_{Q_\beta(u,v,\vec{w}) \in \xi} y = v$.

Similarly, an atom $x \mapsto \{(f,y)\}$ is called implicit in ξ iff the conditions (1) and (2) above hold, (3′) an atom $u \mapsto \theta_i$ with $(f,d_i) \in \theta_i$ is included in the definition of Q, for all $Q \in \mathcal{P}(\xi)$, and (4′) $\varphi \Rightarrow \bigwedge_{1 \leq i \leq n} y = d_i$.

For example, if $\xi = \{\mathit{List}_\alpha(x,y)\}$ is a set of atoms in φ_1 in eq. (2.6), the atom $\mathit{List}_\delta(x,n)$ is implicit in ξ because $\beta \subseteq \alpha$ in φ_1 implies that $n \in \alpha$ and the equality $x = x$ is trivially implied by φ_1. Similarly for the atom $\mathit{List}_\gamma(n,y)$.

By definition, the formula $F(\varphi)$ defined in the previous section can also be used to infer all the atoms which are implicit in some set of spatial constraints in φ. For example, conditions (1) and (2) above are equivalent to:

$$F(\varphi) \Rightarrow \bigvee_{1\leq i\leq n} [x \in \alpha_i] \text{ and, for any } j,\ F(\varphi) \not\Rightarrow \bigvee_{1\leq i\leq n, i\neq j} [x \in \alpha_i].$$

The conditions (3) and (3′) can be checked syntactically on the definition of the recursive predicates. Thus, the computation of the implicit spatial constraints for a formula is co-NP complete.

6.2 *NOLL* graphs

We define *NOLL graphs*, a graph representation for *NOLL* formulas. Roughly, the nodes of these graphs represent location variables and the edges represent spatial or difference constraints. The $*$ separation is represented in the *NOLL* graph by a binary relation Ω_* over edges while the sharing constraints are kept unchanged. A representation for a formula together with the implicit spatial constraints is called a *complete NOLL graph*.

Definition 3 (*NOLL* graph). *Given a NOLL formula $\varphi = \Pi \wedge \Sigma \wedge \Lambda$ over a set of predicates $\mathcal{P}$, we define the NOLL* graph *of φ, denoted $G(\varphi)$, as a tuple $(V, E_P, E_R, E_D, \ell, \Omega_*, \Lambda)$ or the error graph $\perp$, where:*

- *each node in V denotes an equivalence class over elements of LVars w.r.t. the equality relation defined in Π; the equivalence class of x, is denoted by $[x]$. If Π contains both $x \neq y$ and $x = y$ then G is the error graph $\perp$;*
- *$E_P \subseteq V \times RefFlds \times V$ represents the points-to constraints: $([x], f, [y]) \in E_P$ iff $x \mapsto \theta$ with $(f, y) \in \theta$ is an atomic formula in Σ;*
- *$E_R \subseteq V \times \mathcal{P} \times V^+ \times V$ represents list segment constraints: $([x], P_\alpha, [\overrightarrow{z}], [y]) \in E_R$ iff $P_\alpha(x, y, \overrightarrow{z})$ is an atomic formula in Σ;*
- *$E_D \subseteq V \times V$ represents inequalities: $([x], [y]) \in E_D$ iff $x \neq y$ is an atom in Π;*
- *$\ell : LVars \to V$ is called the* variable labeling *and it is defined by $\ell(x) = [x]$, for any $x \in LVars$;*
- *Ω_* contains all pairs of edges in $E_P \cup E_R$ denoting $*$ separated atoms in Σ.*

In the following, $V(G)$, denotes the set of nodes in the *NOLL* graph G. We use a similar notation for all the other components of G. Also, for any $n \in V(G)$, $vars_G(n)$ denotes the set of all the variables labeling the node n in G. The graph on the right of Fig. 3 represents $G(\varphi_2)$, where $V = \{x, y, n, m\}$, $E_P = E_D = \emptyset$, E_R contains the three edges corresponding to the three list segments, Ω_* contains only one pair $\langle([x], List_\alpha, [n]), ([n], List_\beta, [y])\rangle$, and Λ is $\beta \subseteq \delta \cup \gamma$.

A graph representation for φ which contains the edges corresponding to all implicit spatial constraints of φ is called a complete *NOLL* graph. In addition to the *NOLL* graph components, a complete graph has an attribute Δ, which identifies the set of atoms where a spatial constraint is implicit in.

Definition 4 (complete *NOLL* graph). *Given a NOLL formula $\varphi = \Pi \wedge \Sigma \wedge \Lambda$, the* complete *NOLL* graph *of φ, denoted by $\overline{G}(\varphi)$ is a tuple (G, Δ) where:*

- *G is an NOLL graph where all components except E_R, E_P, Ω_*, and Λ are equal to the components of $G(\varphi)$,*
- *$E_R(G)$ and $E_P(G)$ include $E_R(G(\varphi))$ resp. $E_P(G(\varphi))$ and, for any atom $P_\alpha(x,y,\overrightarrow{z})$, resp. $x \mapsto \{(f,y)\}$, which is implicit in some set of atoms ξ, $e = ([x], P_\alpha, [\overrightarrow{z}], [y]) \in E_R(G)$, resp. $e = ([x], f, [y]) \in E_P(G)$.*
- *$\Omega_*(G)$ includes $\Omega_*(G(\varphi))$ and, for any edge e representing an implicit constraint in ξ, and for any other edge $e' \in E_P \cup E_R$, if there exists an edge e'' representing an atom in ξ such that $(e', e'') \in \Omega_*(G(\varphi))$ then $(e, e') \in \Omega_*(G(\varphi))$.*
- *$\Delta \subseteq (E_P \cup E_R) \times 2^{E_R}$ represents the relation between edges and the sets of list segments where they are implicit in, i.e., for every $P_\alpha(x,y,\overrightarrow{z})$, resp., $x \mapsto \{(f,y)\}$, implicit in ξ, $(([x], P_\alpha, [\overrightarrow{z}], [y]), E_\xi) \in \Delta$, resp. $(([x], f, [y]), E_\xi) \in \Delta$, where E_ξ is the set of edges representing the atoms in ξ,*
- *if $P_{\alpha_1}(x,y,\overrightarrow{z})$ and $P_{\alpha_2}(y,t,\overrightarrow{z})$ are implicit in $\xi = \{P_\alpha(x,t,\overrightarrow{z})\}$ then, $\alpha = \alpha_1 \cup \alpha_2$ is added to Λ.*

The graph on the left of Fig. 3 represents $\overline{G}(\varphi_1)$, where $V = \{x,y,n,m\}$, $E_P = E_D = \Omega_* = \emptyset$, and E_P contains the four edges: two edges represent the spatial constraints in φ_1, and the edges $([x], List_{\alpha_1}, [n])$ and $([n], List_{\alpha_2}, [m])$ represent implicit constraints in $\xi = \{List_\alpha(x,y)\}$. Λ is $\beta \subseteq \alpha \wedge \alpha = \alpha_1 \cup \alpha_2$ and Δ is the relation $\{(([x], List_{\alpha_1}, [n]), \xi), (([n], List_{\alpha_2}, [m]), \xi)\}$.

6.3 *NOLL* graph homomorphism

Given a *NOLL* graph G_1 and a complete *NOLL* graph G_2, a *homomorphism from G_1 to G_2* is a mapping $h : V(G_1) \mapsto V(G_2)$, which:

1. preserves the labeling with location variables, i.e., $vars_{G_1}(n) \subseteq vars_{G_2}(h(n))$, for any $n \in V(G_1)$,
2. maps each difference, resp., points-to, edge of G_1 to a difference, resp., points-to, edge of G_2, (e.g., for any $(n,f,n') \in E_P(G_1)$, $(h(n),f,h(n')) \in E_P(G_2)$), and
3. maps each edge representing a list segment in G_1 to a path in G_2 formed of edges in $E_P(G_2) \cup E_R(G_2)$.

To explain the mapping of edges in $E_P(G_1)$ to paths of G_2, let us consider the case of an edge $(n, P_\alpha, m, n') \in E_P(G_1)$, where $n, m, n' \in V(G_1)$ and

$$P(\alpha)(in, out, b) \triangleq (in = out) \vee (\exists u. \Sigma_0(in, u, b) * P(u, out, b)).$$

The definition of h requires that there exists a sequence of nodes $\pi = \pi_1 \ldots \pi_k$, $k \geq 1$, in G_2 s.t. $\pi_1 = h(n)$, $\pi_k = h(n')$, and for every two consecutive nodes π_i and π_{i+1}, either

- $E_P(G_2)$ contains some set of edges between π_i, π_{i+1}, and $h(m)$, which prove that $\Sigma_0(x_i, x_{i+1}, x_{h(m)})$ holds, where x_i, x_{i+1}, and $x_{h(m)}$ are some variables labeling π_i, π_{i+1}, and $h(m)$, respectively, or
- there exists an edge $(\pi_i, P'_\beta, \overrightarrow{m'}, \pi_{i+1})$ in $E_R(G_2)$, representing a stronger predicate than P_α, i.e., $h(m) \in \overrightarrow{m'}$ and $P'_\beta(x_i, x_{i+1}, \overrightarrow{z}) \Rightarrow P_\alpha(x_i, x_{i+1}, x_{h(m)})$, where x_i, x_{i+1}, and $x_{h(m)}$ are as above, and $\overrightarrow{z}$ is a set of variables labeling $\overrightarrow{m'}$ such that $x_{h(m)} \in \overrightarrow{z}$ (this

is possible because $h(m) \in \overrightarrow{m'}$). The entailment between recursive predicates can be checked syntactically in polynomial time. In the *MOLL* fragment, this entailment is reduced an entailment between points-to constraints (i.e., the constraints in Σ_0).

In addition to the requirements (1–3) above, the function h, must also satisfy constraints which are related to the semantics of the separating conjunctions, the special status of the implicit spatial constraints, and the sharing constraints.

To define these additional constraints, for every edge e in $E_P(G_1) \cup E_R(G_1)$, we define a set $used(e) \subseteq E_P(G_2) \cup 2^{(E_R(G_2) \times RefFlds)}$, which intuitively represents all the edges/record fields used in the path from G_2 to which e is mapped by h. Thus, $used(e)$ consists of (1) the set of points-to edges in the path associated to e and (2) the set of pairs of the form (e', f), where e' represents a list segment from the same path, if such an edge exists, and $f \in RefFlds_0(P_\alpha)$ is a record field from the definition of the predicate P_α used in the spatial constraint denoted by e (the path associated to e contains an edge in $E_R(G_2)$ only if $e \in E_R(G_1)$). When the path associated to e contains an edge e', which denotes a spatial constraint implicit in some set ξ, i.e., $(e', E_\xi) \in \Delta(G_2)$, then $used(e)$ includes all pairs (e'', f) with $e'' \in E_\xi$ and $f \in RefFlds_0(P_\alpha)$. This is because the atom represented by e' is not $*_w$ separated from the spatial constraints in ξ.

Then, to express the semantics of $*_w$, we require that $used(e_1) \cap used(e_2) = \emptyset$, for any two edges e_1 and e_2 in $E_P(G_1) \cup E_R(G_1)$. Concerning $*$, it is required that for any two edges e_1 and e_2 in $E_P(G_1) \cup E_R(G_1)$ s.t. $(e_1, e_2) \in \Omega_*(G_1)$, we have that $(e'_1, e'_2) \in \Omega_*(G_2)$, for any e'_1 an edge appearing in $used(e_1)$ and e'_2 an edge appearing in $used(e_2)$.

Finally, for the sharing constraints, the mapping by h of edges in $E_R(G_1)$ to paths in G_2 defines a substitution Γ for set of locations variables in $\Lambda(G_1)$ to terms over set of locations variables in $\Lambda(G_2)$. For example, the homomorphism in Fig. 3 defines the substitution $\Gamma(\delta) = \alpha_1$, $\Gamma(\gamma) = \alpha_2$, and $\Gamma(\beta) = \beta$. The implicit constraints in G_1 gives that $\Lambda(G_1) := \beta \subseteq \alpha \wedge \alpha = \alpha_1 \cup \alpha_2$. Given a formula Λ over variables in *SetVars*, $\Lambda[\Gamma]$ denotes the formula obtained from Λ by applying the substitution Γ. Then, it is required that $\Lambda(G_2) \Rightarrow \Lambda(G_1)[\Gamma]$. Such a formula belongs for instance, to the fragment of BAPA [14], and thus its validity can be decided in NP-time. In our example, we obtain the trivial entailment $\beta \subseteq \alpha \wedge \alpha = \delta \cup \gamma \Rightarrow \beta \subseteq \delta \cup \gamma$.

6.4 Checking entailments of *NOLL* formulas

The procedure `CheckEntl` for entailment-checking in *NOLL* is given in Fig. 8.

`procedure CheckEntl(`$\varphi \Rightarrow \psi$`)`
- $\varphi' :=$ `the normal form of` φ
- $\psi' :=$ `the normal form of` ψ
- $G_1 :=$ `the complete` *NOLL* `graph of` φ'
- $G_2 :=$ `the` *NOLL* `graph of` ψ'
- $h :=$ `the function` $h : V(G_2) \rightharpoonup V(G_1)$ `s.t.` $vars_{G_2}(n) \subseteq vars_{G_1}(h(n)),\ \forall n \in V(G_2)$
- `return (`h `is total) and (`h `is a homomorphism from` G_2 `to` G_1`)`

Fig. 8: The procedure `CheckEntl(`$\varphi \Rightarrow \psi$`)`.

For our running example at page 6, the graphs and the homomorphism computed by the procedure `CheckEntl(`$\varphi_1 \Rightarrow \varphi_2$`)` are illustrated on Fig. 3. Note that formulae φ_1 and φ_2 are already in the normal form.

The following theorem states the correctness and the complexity of `CheckEntl`.

Theorem 5. *Given two NOLL formulas* φ *and* ψ, $\varphi \Rightarrow \psi$ *holds iff* `CheckEntl`$(\varphi \Rightarrow \psi)$ *returns* `true`*. Moreover, the complexity of* `CheckEntl` *is co-NP time.*

Proof. (Sketch) The main steps in proving the direction ($\Leftarrow$) are: (1) prove that a program configuration (C,J) is a model of φ iff there exists a homomorphism from the *NOLL* graph of φ to the *NOLL graph of* (C,J) (a model (C,J) can be seen as a *NOLL* formula that uses only points-to spatial constraints), and (2) prove that the composition of two homomorphisms is again a homomorphism. For the direction ($\Rightarrow$), it is enough to prove that if h is not total or it is not a homomorphism from G_2 to G_1 then one can build a counter-example for $\varphi \Rightarrow \psi$. The co-NP complexity follows from Th. 4, the fact that implicit constraints are discovered in co-NP time, and the fact that checking if h is a homomorphism is done in polynomial time. □

7 Experimental results

We have implemented the procedure for entailment checking in a solver which takes as input the specification of predicates in $\mathcal{P}$ and two formulas $\varphi, \psi \in$ *NOLL* defined over $\mathcal{P}$ and returns as result either the homomorphism found when $\varphi \Rightarrow \psi$ or a diagnosis explaining why the entailment is not valid. The diagnosis is given as a list of variables or atomic spatial constraints in φ and ψ for which the conditions for the homomorphism are not satisfied, i.e., there are no paths in $\overline{G}(\varphi)$ corresponding to edges in $\overline{G}(\psi)$, or the set location variables bound to the atomic spatial constraints do not satisfy the sharing constraints. The solver is implemented in C. It uses MiniSat [9] to compute normal forms and an ad-hoc solver for the sharing constraints.

We have used this solver to check verification conditions generated for procedures working on singly linked lists, doubly linked lists, and overlaid hash tables and lists in the Nagios network monitoring example. We consider mainly the procedures for inserting or moving elements in these data structures. The post-condition computation follows the standard approach: introducing primed variables to denote old values and unfolding recursive predicates for statements that involve record fields. (When unfolding predicates, new location and set-of-locations variables are introduced.) To generate simpler verification conditions, we use the frame rules for the separating conjunction operators. In this way, the graph representations for the *NOLL* formulas have less than ten vertices and twenty edges (including the inferred edges), and less than five set of locations variables. Each verification condition is decided in less than 0.1 seconds. The diagnosis feature of the solver has been very useful to obtain valid Hoare triples for the proof. Examples of verification conditions dealt by our solver are given in the Appendix A.8.

References

1. 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, 2007.

2. J. Berdine, C. Calcagno, and P.W. O'Hearn. A decidable fragment of separation logic. In *FSTTCS*, volume 3328 of *LNCS*, pages 97–109. Springer, 2004.
3. J. Berdine, C. Calcagno, and P.W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *FMCO*, volume 4111 of *LNCS*, pages 115–137. Springer, 2005.
4. F. Bobot and J.C. Filliatre. Separation predicates: a taste of separation logic in first-order logic. In *ICFEM*, 2012.
5. C. Calcagno, H. Yang, and P.W. O'Hearn. Computability and complexity results for a spatial assertion language for data structures. In *FSTTCS*, volume 2245 of *LNCS*, pages 108–119, 2001.
6. B.-Y.E. Chang and X. Rival. Relational inductive shape analysis. In *POPL*, pages 247–260. ACM, 2008.
7. A. Chlipala. Mostly-automated verification of low-level programs in computational separation logic. In *PLDI*, pages 234–245. ACM, 2011.
8. 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.
9. N. Eén and N. Sörensson. An extensible sat-solver. In *SAT*, volume 2919 of *LNCS*, pages 502–518. Springer, 2003.
10. C. Enea, V. Saveluc, and M. Sighireanu. Composite invariant checking for nested, overlaid linked lists, 2012.
11. P. Hawkins, A. Aiken, K. Fisher, M.C. Rinard, and M. Sagiv. Data representation synthesis. In *PLDI*, pages 38–49. ACM, 2011.
12. S. Ishtiaq and P.W. O'Hearn. Bi as an assertion language for mutable data structures. In *POPL*, pages 14–26. ACM, 2001.
13. B. Jacobs, J. Smans, and F. Piessens. A quick tour of the VeriFast program verifier. In *APLAS*, pages 304–311, 2010.
14. V. Kuncak, H.H. Nguyen, and M.C. Rinard. An algorithm for deciding bapa: Boolean algebra with presburger arithmetic. In *CADE*, volume 3632 of *LNCS*, pages 260–277. Springer, 2005.
15. P. Madhusudan, X. Qiu, and A. Stefanescu. Recursive proofs for inductive tree data-structures. In *POPL*. ACM, 2012.
16. J.A. Navarro Pérez and A. Rybalchenko. Separation logic + superposition calculus = heap theorem prover. In *PLDI*, pages 556–566, 2011.
17. J.C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, pages 55–74. IEEE Computer Society, 2002.
18. P. Suter, M. Dotta, and V. Kuncak. Decision procedures for algebraic data types with abstractions. In *POPL*, pages 199–210, 2010.
19. 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, 2008.

A Appendices

A.1 Proof of Theorem 1

We reduce 3SAT, the satisfiability problem of CNF formulas with 3 literals in each clause, to the satisfiability problem of some *MOLL* formula. Let Γ be a CNF formula with 3 literals in each clause.

We build a *MOLL* formula φ_Γ which is satisfiable iff Γ is satisfiable as follows. We introduce one location variable $root \in LVars$ and two record fields $g_\gamma, h_\gamma \in RefFlds_{rec}$ for each clause γ in Γ. For each boolean variable a, we introduce one record field $f_a \in RefFlds_{rec}$, two location variables $x_a, x_{\neg a}$, and another six location variables $y_{\gamma,a}, y_{\gamma,\neg a}, z_{\gamma,a}, z_{\gamma,\neg a}, t_{\gamma,a}, t_{\gamma,\neg a}$, for each clause γ in Γ. Each recursive record field s introduced above is used to define a list segment predicate denoted $s^*(in, out)$ and defined by $(in = out) \vee (\exists u.\, in \mapsto \{(s,u)\} * s^*(u, out))$.

The basic building blocks of φ_Γ are the following formulas:

$$\psi_a \triangleq x_a \neq x_{\neg a} \wedge f_a^*(root, x_a) *_w f_a^*(root, x_{\neg a}), \text{ for each boolean variable } a, \quad \text{(A.1)}$$

$$\psi_{\gamma,\ell} \triangleq z_{\gamma,\ell} \neq t_{\gamma,\ell} \wedge g_\gamma^*(x_\ell, y_{\gamma,\ell}) *_w h_\gamma^*(y_{\gamma,\ell}, z_{\gamma,\ell}) *_w h_\gamma^*(y_{\gamma,\ell}, t_{\gamma,\ell})), \quad \text{(A.2)}$$

$$\text{for each literal } \ell \text{ in some clause } \gamma \text{ in } \Gamma.$$

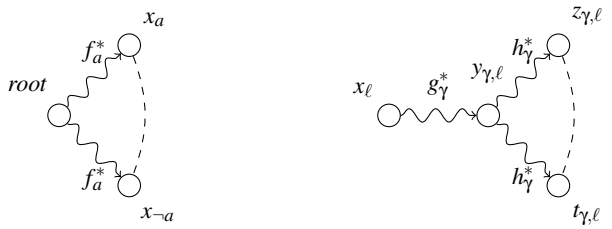
To help the understanding, we picture the building blocks above as graphs in Fig. 9: location variables are presented by vertices, list segment constraints are represented by snacked lines labeled by the predicate name, and difference constraints are represented by dashed lines.

The formula ψ_a is used to assign a truth value to the variable a. A model of this formula must keep only one of the list segment constraints using the fields f_a not empty, i.e., either $root = x_a \neq x_{\neg a}$ or $root = x_{\neg a} \neq x_a$. The first (resp. second) case is interpreted as an assignment of a to *true* (resp. *false*).

The formulas $\psi_{\gamma,a}$ and $\psi_{\gamma,\neg a}$, for some boolean variable a, are used to encode the clause γ such that the assignments that falsify all the literals in γ (a literal being either a or $\neg a$) do not correspond to models of φ_Γ. For example, given Γ_1 a CNF formula with one clause $\gamma = a_1 \vee \neg a_2 \vee \neg a_3$, φ_{Γ_1} is defined as follows:

$$\varphi_\gamma \triangleq \psi_{a_1} *_w \psi_{a_2} *_w \psi_{a_3} *_w \psi_{\gamma,a_1} *_w \psi_{\gamma,\neg a_2} *_w \psi_{\gamma,\neg a_3}$$

(Here we use the distributivity of $*_w$ operator over the conjunction, i.e., $(\Pi_1 \wedge \Sigma_1) *_w (\Pi_2 \wedge \Sigma_2) \equiv (\Pi_1 \wedge \Pi_2) \wedge (\Sigma_1 *_w \Sigma_2)$.) The graph representation of φ_{Γ_1} is given in Fig. 10. This formula is satisfiable iff at most two of the location variables $x_{a_1}, x_{\neg a_2}, x_{\neg a_3}$ are interpreted to the same location as *root*, i.e., at least one of the literals $a_1, \neg a_2, \neg a_3$ is true. Indeed, if all these literals are set to false, three list segment constraints g_γ^* shall start from *root*; this is possible only if two of the three list segment constraints are empty. But this situation can not lead to a model because it requires that four list segment constraints h_γ^* start from *root* and at least two of them shall be not empty (because of the difference constraints $z_{\gamma,a_i} \neq t_{\gamma,a_i}$ and $z_{\gamma,\neg a_i} \neq t_{\gamma,\neg a_i}$). Consider now that only a_1 and $\neg a_2$ are interpreted to false, then two list segment constraints g_γ^* shall start from *root*, which is possible only one one of them being empty. Thus, φ_γ specifies a model with five non empty list segments starting from *root*: one for each $f_{a_i}^*$, one for g_γ^*, and one for h_γ^*.



(a) ψ_a (b) $\psi_{\gamma,\ell}$

Fig. 9: Subformulas for the reduction from 3*SAT* to the satisfiability of *MOLL*.

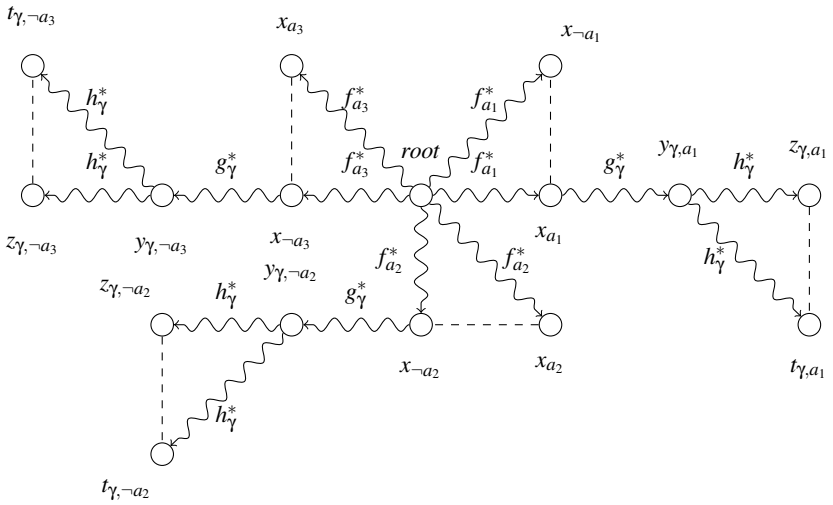


Fig. 10: The *MOLL* formula equi-satisfiable to the formula Γ_1 that contains only one clause $\gamma = (a_1 \vee \neg a_2 \vee \neg a_3)$.

A.2 Small model property for *NOLL*

Proof of the Lemma 1 We prove that each step of the procedure in Fig. 7a generates a model or a colored heap graph for φ, where $\varphi = \Pi \wedge \Sigma \wedge \Lambda$.

The procedure $\mathsf{purify}(C,J)$ does a traversal of C and mark each location (1) labeled by a location variable free in φ or (2) used by some spatial constraint in Σ. The locations not marked in (C,J) are removed, i.e., $H^{C'}$ is undefined for the unmarked locations and the image of J' does not use unmarked locations.

Lemma 2. *If (C,J) is a model of φ then $(C',J') = \mathsf{purify}(C,J)$ is also a model of φ.*

Proof. The pure constraints in Π are still satisfied since C' contains all the locations labeled by the location variables used in φ. The spatial constraints in Σ are also preserved because, since C is deterministic (S^C and H^C are functions), the part of C satisfying some spatial constraint is unique. Or C' contains all the locations involved in the spatial constraints in Σ, thus these constraints are still satisfied by (C',J'). For the sharing constraints, notice that $J(\alpha) = J'(\alpha)$ for any $\alpha \in \mathit{SetVars}$ bound to a predicate constraint. For variables $\beta \in \mathit{SetVars}$ not bound in σ, we obtain that $J'(\beta) = J(\beta) \cap S^C(\mathit{LVars}(\varphi))$. Thus, the constraints of the form $x \in t$ and $x \notin t$ are still satisfied by (C',J'). For the constraints $t \subseteq t'$, the removing operation deletes the same locations in $[t]_J$ and $[t]_{J'}$, thus it preserves these constraints.

The procedure $\mathsf{splitLocations}(C',J')$ builds the colored heap graph abstraction of (C',J') by splitting the nodes not labeled by location variables but shared between several list segments described by predicates in φ.

Lemma 3. *If (C,J) is a model of φ then $G = \mathsf{splitLocations}(C,J)$ satisfies φ.*

Proof. The pure constraints in Π are still satisfied since locations labeled with *LVars* are not split. The atomic spatial constraints points-to and list segment are also satisfied in G because all the edges are preserved by the splitting. Moreover, the sub-graphs of G satisfying different list segment constraints do not share any edge from the fact that C is a model of φ and it contains only edges involved by spatial constraints in φ. Thus, the conditions for the $*_w$ operators used in φ are preserved in G. The conditions required for the satisfaction of the spatial constraints linked by $*$ on colored heap graphs are consequences of the definition of $*$ the model (C,J). Since the splitting preserves all the locations and does not introduce nodes not labelled by locations in C, the sharing constraints are also preserved.

The procedure $\mathsf{computeCrucial}(G)$ uses a set of auxiliary location variables LVars' disjoint from *LVars*. If a node is labeled by some variable in LVars', its sibling node is also labeled by another variable in LVars'.

computeCrucial starts by labeling the nodes successor by a non recursive record field of a node labeled with variables in *LVars* (e.g., node 2_s in Fig. 6). Intuitively, we label these node in order to keep them together with the nested list segments starting from this node. Let denote the set of nodes labeled in this way (including their sibling) by V'_{nr}.

Secondly, we choose for each pair on nodes (n,m) where n is labeled in $LVars \cup LVars'$ and m is labeled in $LVars$ one (directed) path starting in n and ending in m. Notice that several paths may exists from n to m. For example, in Fig. 6, four paths link the node 1 to the node 0. However, no path exists linking 0 to 1, neither linking 9 to 1. On each designated path, computeCrucial labels with variables in $LVars'$ the source or the target of a non recursive record field/ on the path. For example, in Fig. 6, for the path from 1 to 9, the node 8 is labeled by z'. Intuitively, we label these nodes in order to witness about the paths between location variables referenced in φ. Let denote the set of nodes labeled in this way (including their sibling) by V'_p.

Lemma 4. *The size of* $V' = \mathsf{computeCrucial}(G)$ *is polynomial in the size of* φ *and* $\mathcal{P}$.

Proof. The size of V'_{nr} is bounded by $|RefFlds| \times |LVars(\varphi)|$. (A tighter bound is obtained by looking at the type of each variable in $LVars(\varphi)$.) Because the nesting of the recursive predicates is bounded, the paths between nodes labeled by variables in $LVars$ do not contain cycles and the number of non recursive record fields is bounded by the nesting depth N. Notice that the nesting depth is also bounded by $|RefFlds \setminus RefFlds_{rec}|$. Thus, the size of V'_p is bounded by $|LVars(\varphi)|^2 \times |RefFlds| \times 2N$.

Lemma 5. *If* G *satisfies* φ *and* V' *is the set of crucial nodes computed by* $\mathsf{computeCrucial}(G)$ *then* $G' = \mathsf{labelCrucial}(G,V')$ *also satisfies* φ.

Proof. Notice that only nodes of G which are not already labeled with variables in $LVars$ are in V'. Moreover, the labels in $LVars'$ do not appear in φ. Thus, the constraints satisfied by G are also satisfied in G'.

The procedure collapseAnonymous applies the following rule to collapse the "anonymous" nodes (i.e., not labeled by variables in $LVars(\varphi) \cup LVars'$) in G':

$$G_1(n_1) \xrightarrow{s} G_2(n_2) \xrightarrow{s} n'', \quad \text{and } G_2 \text{ contains only anonymous nodes} \qquad \rightsquigarrow G_1(n_1) \xrightarrow{s} n''$$

where $G_i(n_i)$ means that n_i is the root (the upper level node in the list segment definition) of the sub-graph G_i, and s is a recursive field in $RefFlds$. Roughly, the collapsing removes a node (and the sub-graph representing the nested, anonymous structure) if it is between two recursive record fields (see Fig. 7b). collapseAnonymous starts in a bottom up manner, i.e., starting from the inner list segments to the upper ones. Due to the special syntax of predicates in $\mathcal{P}$, we can compute for each list segment the minimal number of anonymous nodes that must be preserved in order to satisfy some given spatial constraint. Thus, we can prove the following property:

Lemma 6. *Given a recursive predicate* $P \in \mathcal{P}$*, there is a minimal colored heap graph satisfying* $\Sigma_1(P)(\overrightarrow{v}, \overrightarrow{nhb})$ *and having only anonymous nodes except nodes in* $\overrightarrow{nhb}$.

Proof. The property is a consequence of the fact that predicates in $\mathcal{P}$ are not mutually recursive, the spatial constraints in $\Sigma_1(P)$ are $*$ separated, and that an empty list segment or a list segment with only one element is a model of some list segment constraint. For predicates with no nesting, i.e., $\Sigma_1(P) \equiv emp$, the minimal size, denoted

by $min(P)$ is 0. For predicates with nesting, the minimal size of this model is given by $|\overrightarrow{v}| + \Sigma_{v \in \overrightarrow{v}} min(Q_v)$. Intuitively, the minimal model is obtained when all the spatial constraints in the predicate definition are interpreted as list segments of length one.

Thus, the collapsing of anonymous nodes terminates for each sub-graph built only from anonymous nodes. After the collapsing process, the nodes labeled in $LVars \cup LVars'$ may be the root of a graph but, due to the way the labeling has beet done, these graphs does not contain anonymous nodes. Thus, if the collapsing process keeps only nodes labeled in $LVars \cup LVars'$.

It remains to show that the graph obtained from collapsing is still a model of φ.

Lemma 7. *If G satisfies φ and all its crucial nodes are labeled then $G' = \mathsf{collapseAnonymous}(G')$ also satisfies φ.*

Proof. The collapsing process preserves a model of φ because no edges are added and the nodes marked as important for the satisfaction of the spatial and sharing constraints are kept.

The $\mathsf{mergeNodes}(G)$ procedure builds a model (C,J) from G by putting together node colored by the same location.

Lemma 8. *If G is obtained by collapsing of anonymous nodes in φ then $(C,J) = \mathsf{mergeNodes}(G)$ is a model of φ.*

Proof. Recall that the crucial nodes have been computed by closing under the sibling relation. Thus, all the nodes obtained by splitting a locations has been kept during the collapsing process if they has been computed as crucial. The merging is then simple: all the sibling nodes are put together with their edges in a location in C and J is built from $\mathcal{S}^G$ in the same way. From the fact that G satisfies all the atomic constraints and the separation constraints in φ, we obtain that (C,J) also satisfies them.

A.3 Full definition of $F(\varphi)$ for *MOLL* formulas

Let $\varphi = \Pi \wedge \Sigma \wedge \Lambda$ be a *MOLL* formula where all variables are of the same type. The formula $F(\varphi)$ is defined by:

$$F(\varphi) = F(\Pi) \wedge F_{eq} \wedge F(\Sigma) \wedge F_{det} \wedge F(\Lambda) \wedge F_{\in}, \tag{A.3}$$

where the sub-formulas of $F(\varphi)$ are defined inductively on the syntax of *NOLL* formulas in Tables 4, 5, 2, and 6.

The separation of locations (which are interpretations of location variables) induced by the use of the object separating conjunction is encoded in the formula $F(\Sigma_1 * \Sigma_2)$ (in Table 5, equation (A.12)). For any two atoms $A \in \Sigma_1$ and $B \in \Sigma_2$, $F_*(A,B)$ encodes the fact that if A and B represent non-empty list segments then the left ends of these segments are disjoint.

The formula $F_{\in}$ expresses the fact that the formulas of the form $x \in \alpha$ are closed under the equality between location variables and the relation between these formulas and list segment constraints in φ.

$$F(\mathit{true}) = 1 \qquad F_{eq} = \wedge_{x \in LVars(\varphi)} [x = x] \tag{A.7}$$

$$F(x = y) = [x = y] \tag{A.4}$$

$$\wedge \bigwedge_{x,y,z \in LVars(\varphi)} \big([x = y] \wedge [y = z]\big) \Rightarrow [x = z]$$

$$F(x \neq y) = \neg[x = y] \tag{A.5}$$

$$F(\Pi_1 \wedge \Pi_2) = F(\Pi_1) \wedge F(\Pi_2) \tag{A.6}$$

Table 4: Definition of $F(\Pi)$ and F_{eq} for an *MOLL* formula $\varphi = \Pi \wedge \Sigma \wedge \Lambda$.

$$F(\mathit{emp}) = 1 \tag{A.8}$$

$$F(x \mapsto \theta) = \bigwedge_{(f,y) \in \theta} [x, y, f] \tag{A.9}$$

$$F(P_\alpha(x, y, \overrightarrow{z})) = [P_\alpha(x, y, \overrightarrow{z})] \oplus [x = y] \tag{A.10}$$

$$F(\Sigma_1 *_w \Sigma_2) = F(\Sigma_1) \wedge F(\Sigma_2) \tag{A.11}$$

$$F(\Sigma_1 * \Sigma_2) = F(\Sigma_1) \wedge F(\Sigma_2) \wedge \bigwedge_{A \in atom(\Sigma_1), B \in atom(\Sigma_2)} F_*(\{A, B\}) \tag{A.12}$$

$$F_*(\{x_1 \mapsto \theta_1, x_2 \mapsto \theta_2\}) = \neg[x_1 = x_2] \tag{A.13}$$

$$F_*(\{x_1 \mapsto \theta_1, P_\alpha(x_2, y_2, p_2)\}) = \neg[x_1 = x_2] \vee [x_2 = y_2] \tag{A.14}$$

$$F_*(\{P_\alpha(x_1, y_1, p_1), Q_\beta(x_2, y_2, p_2)\}) = \neg[x_1 = x_2] \vee [x_2 = y_2] \vee [x_1 = y_1] \tag{A.15}$$

Table 5: Definition of $F(\Sigma)$ for an *MOLL* formula $\varphi = \Pi \wedge \Sigma \wedge \Lambda$.

$$F(true) = true \tag{A.16}$$

$$F(x \in \bigcup_{1\le i\le n} \{u_i\}) = \bigvee_{1\le i\le n} [x = u_i] \tag{A.17}$$

$$F(x \in \alpha_1) = \begin{cases} [x \in \alpha_1] \wedge [P_{\alpha_1}(x_1, y_1, \overrightarrow{z})] & \text{, if } P_{\alpha_1}(x_1, y_1, \overrightarrow{z}) \text{ in } \varphi \\ [x \in \alpha_1] & \text{, otherwise} \end{cases} \tag{A.18}$$

$$F(x \notin \alpha_1) = \neg F(x \in \alpha_1) \tag{A.19}$$

$$F(x \in \alpha_1 \cup \bigcup_{1\le i\le n} \{u_i\}) = F(x \in \alpha_1) \vee \bigvee_{1\le i\le n} [x = u_i] \tag{A.20}$$

$$F(x \in \alpha_1 \cup \alpha_2) = F(x \in \alpha_1) \vee F(x \in \alpha_2) \tag{A.21}$$

$$\begin{aligned} F(\alpha_1 \subseteq \alpha_2) = & \bigwedge_{x \in LVars(\varphi)} [x \in \alpha_1] \Rightarrow [x \in \alpha_2] \\ & \wedge \begin{cases} [P_{\alpha_1}(x_1, y_1, \overrightarrow{z})] \Rightarrow F(x_1 \in \alpha_2) & \text{, if } P_{\alpha_1}(x_1, y_1, \overrightarrow{z}) \text{ in } \varphi \\ 1 & \text{, otherwise} \end{cases} \end{aligned} \tag{A.22}$$

$$\begin{aligned} F(\alpha_1 \subseteq \alpha_2 \cup \bigcup_{1\le i\le n} \{u_i\}) = & \bigwedge_{x \in LVars(\varphi)} [x \in \alpha_1] \Rightarrow \left([x \in \alpha_2] \vee \bigvee_{1\le i\le n} [x = u_i]\right) \\ & \wedge \left(F(\alpha_1 \subseteq \alpha_2) \vee \bigvee_{1\le i\le n} F(u_i \in \alpha_1)\right) \\ & \wedge \begin{cases} [P_{\alpha_1}(x_1, y_1, \overrightarrow{z})] \Rightarrow F(x_1 \in \alpha_2 \cup \bigcup_{1\le i\le n} \{u_i\}) & \text{, if } P_{\alpha_1}(x_1, y_1, \overrightarrow{z}) \text{ in } \varphi \\ 1 & \text{, otherwise} \end{cases} \end{aligned} \tag{A.23}$$

$$\begin{aligned} F(\alpha_1 \subseteq \alpha_2 \cup \alpha_3) = & \bigwedge_{x \in LVars(\varphi)} [x \in \alpha_1] \Rightarrow \left([x \in \alpha_2] \vee [x \in \alpha_3]\right) \\ & \wedge \begin{cases} [P_{\alpha_1}(x_1, y_1, \overrightarrow{z})] \Rightarrow \bigvee_{i \in \{2,3\}} F(x_1 \in \alpha_i) & \text{, if } P_{\alpha_1}(x_1, y_1, \overrightarrow{z}) \text{ in } \varphi \\ 1 & \text{, otherwise} \end{cases} \end{aligned} \tag{A.24}$$

$$F(\Lambda_1 \wedge \Lambda_2) = F(\Lambda_1) \wedge F(\Lambda_2) \tag{A.25}$$

$$\begin{aligned} F_{\in} = & \bigwedge_{x, x', \alpha \text{ in } \varphi} ([x = x'] \wedge [x' \in \alpha]) \Rightarrow [x \in \alpha] \\ & \wedge \bigwedge_{P_\alpha(x, y, \overrightarrow{z}) \text{ in } \varphi} [P_\alpha(x, y, \overrightarrow{z})] \Rightarrow [x \in \alpha] \end{aligned} \tag{A.26}$$

Table 6: Definition of $F(\Lambda)$ and $F_{\in}$ for an *MOLL* formula $\varphi = \Pi \wedge \Sigma \wedge \Lambda$.

A.4 Boolean abstractions of *NOLL* formulas

Let $\varphi = \Pi \wedge \Sigma \wedge \Lambda$ be a *NOLL* formula. We first introduce some notations for the predicates in $\mathcal{P}$ parametrizing the logic *NOLL*.

We say that a sort R is *of level i in* $P(in, out, \overrightarrow{nhb})$ iff:

- $R = \tau(in) = \tau(out)$ and $i = 0$, or
- R is of level $i-1$ in $Q(v, b, \overrightarrow{b})$, for some $Q(v, b, \overrightarrow{b})$ in $\Sigma_1(P)$.

A record field $f \in RefFlds$ is called *of level i in P* iff:

- f is used in $\Sigma_0(P)$ and $i = 0$, or
- f is of level $i-1$ in $Q(v, b, \overrightarrow{b})$, for some $Q(v, b, \overrightarrow{b})$ in $\Sigma_1(P)$.

Note that, for any record field f of level i in P, $\tau(f) = R \to R'$, where R is a sort of level i in P.

$[x = y]$	for every $x, y \in LVars(\varphi)$ s.t. $\tau(x) = \tau(y)$
$[x, y, f]$	for every sub-formula $x \mapsto \theta$ of φ with $(f, y) \in \theta$
$[P_\alpha(x, y, \overrightarrow{z})]$	for every sub-formula $P_\alpha(x, y, \overrightarrow{z})$ of φ
$[x', \_, f]$	for every field f of level $i \geq 1$ in some predicate P from φ and $x' \in LVars(\varphi)$, such that $\tau(f) = R \to R'$ and $\tau(x') = R$, for some sorts R and R'
$[x \in \alpha]$	for every $x \in LVars$ and $\alpha \in SetVars$ variables in φ s.t. $\tau(x) \in \tau(\alpha)$

Table 7: Definition of the set $BVars(F(\varphi))$ of boolean variables used in $F(\varphi)$.

The formula $F(\varphi)$ is defined over the set of boolean variables $BVars(F(\varphi))$ introduced in Table 7. Thus,

$$F(\varphi) = F(\Pi) \wedge F_{eq} \wedge F(\Sigma) \wedge F_{det} \wedge F(\Lambda) \wedge F_{\in}, \tag{A.27}$$

where the sub-formulas $F(\Pi)$, F_{eq}, and $F(\Sigma)$ are defined as in the case of *MOLL* formulas (see Appendix A.3). The sub-formulas $F(\Lambda)$, $F_{\in}$, and F_{det} are defined in Table 8 (for $F(\Lambda)$ we give only the cases which are different w.r.t *MOLL* formulas – Table 6).

Concerning $F(\Lambda)$, it is now necessary to take into consideration the typing constraints on the variables. Thus, the only modifications concern the definition of $F(x \in \alpha_1)$ and $F(x \in \alpha_1 \cup \bigcup_{1 \leq i \leq n} \{u_i\})$.

Concerning $F_{\in}$ and F_{det}, it is necessary to add more constraints because atoms of the form $x \in \alpha$ may impose that x belongs to some inner part of a list segment described by a recursive predicate $P_\alpha(x, y, \overrightarrow{z})$.

$$F(x \in \alpha_1) = \begin{cases} [x \in \alpha_1] \wedge [P_{\alpha_1}(x_1, y_1, \overrightarrow{z})], & \text{if } P_{\alpha_1}(x_1, y_1, \overrightarrow{z}) \text{ in } \varphi \text{ and } \tau(x) \in \tau(\alpha_1) \\ [x \in \alpha_1], & \text{if } \alpha_1 \text{ is not bounded to a spatial constraint in } \varphi \text{ and } \tau(x) \in \tau(\alpha_1) \\ \mathit{false}, & \text{otherwise} \end{cases}$$

$$F(x \in \alpha_1 \cup \bigcup_{1 \le i \le n} \{u_i\}) = F(x \in \alpha_1) \vee \bigvee_{\substack{1 \le i \le n \\ \tau(x) = \tau(u_i)}} [x = u_i]$$

$$\begin{aligned} F_{\in} = & \bigwedge_{x, x', \alpha \text{ in } \varphi} ([x = x'] \wedge [x' \in \alpha]) \Rightarrow [x \in \alpha] \\ & \wedge \bigwedge_{P_\alpha(x, y, \overrightarrow{z}) \text{ in } \varphi} [P_\alpha(x, y, \overrightarrow{z})] \Rightarrow [x \in \alpha] \\ & \wedge \bigwedge_{\substack{x', P_\alpha(x, y, \overrightarrow{z}) \text{ in } \varphi \\ \tau(x') \text{ of level } i \ge 1 \text{ in } P}} [x' \in \alpha] \Leftrightarrow \Big(\bigvee_{f \text{ of level } i \text{ in } P} [x', \_, f] \Big) \end{aligned}$$

$$\begin{aligned} F_{det} = & \bigwedge \text{ for any } [x_1, y_1, f], [x_2, y_2, f] \in \mathit{BVars}(F(\varphi)) \text{ different variables} \\ & [x_1 = x_2] \Rightarrow [x_1, y_1, f] \oplus [x_2, y_2, f] \\ & \bigwedge \text{ for any } [x_1, \_, f], [x_2, \_, f] \in \mathit{BVars}(F(\varphi)) \text{ different variables} \\ & [x_1 = x_2] \Rightarrow [x_1, \_, f] \oplus [x_2, \_, f] \\ & \bigwedge \text{ for any } [x_1, y_1, f], [x_2, \_, f] \in \mathit{BVars}(F(\varphi)) \\ & [x_1 = x_2] \Rightarrow [x_1, y_1, f] \oplus [x_2, \_, f] \\ & \bigwedge \text{ for any } [x_1, y_1, f], [P_\alpha(x_2, y_2, \overrightarrow{z_2})] \in \mathit{BVars}(F(\varphi)) \text{ s.t. } f \in \mathit{RefFlds}_0(P) \text{ and } x \in \mathit{LVars}(\varphi) \\ & [x_1 = x] \wedge [x \in \alpha] \Rightarrow [x_1, y_1, f] \oplus [P_\alpha(x_2, y_2, \overrightarrow{z_2})] \\ & \bigwedge \text{ for any } [x_1, \_, f], [P_\alpha(x_2, y_2, \overrightarrow{z_2})] \in \mathit{BVars}(F(\varphi)) \text{ s.t. } f \in \mathit{RefFlds}_0(P) \text{ and } x \in \mathit{LVars}(\varphi) \\ & [x_1 = x] \wedge [x \in \alpha] \Rightarrow [x_1, \_, f] \oplus [P_\alpha(x_2, y_2, \overrightarrow{z_2})] \\ & \bigwedge \text{ for any } [P_\alpha(x_1, y_1, \overrightarrow{z_1})], [Q_\beta(x_2, y_2, \overrightarrow{z_2})] \in \mathit{BVars}(F(\varphi)) \text{ different variables} \\ & \text{s.t. } \mathit{RefFlds}_0(P) \cap \mathit{RefFlds}_0(Q) \neq \emptyset \\ & \text{and } x, x' \in \mathit{LVars}(\varphi) \text{ s.t. } \tau(x) = \tau(x') \text{ is of level 0 in } P \text{ and } Q \\ & [x \in \alpha] \wedge [x' \in \beta] \wedge [x = x'] \Rightarrow [P_\alpha(x_1, y_1, \overrightarrow{z_1})] \oplus [Q_\beta(x_2, y_2, \overrightarrow{z_2})] \end{aligned}$$

Table 8: Definition of $F(\Lambda)$, $F_{\in}$, and F_{det} for an *NOLL* formula $\varphi = \Pi \wedge \Sigma \wedge \Lambda$.

A.5 *NOLL* graph homomorphism

Let G_1 be a *NOLL* graph and G_2 a complete *NOLL* graph. We first explain how edges denoting recursive predicates in G_1 shall be mapped to paths of G_2 by a homomorphism $h : V(G_1) \mapsto V(G_2)$ (in Section 6.3 we have explained only the case of recursive predicates in *MOLL*). Along with the characterization of the path corresponding to some edge $e \in E_R(G_1)$ we also present the definition of $used(e)$ needed to put separation constraints as explained in Section 6.3.

Let $e = (n, P_\alpha, \overrightarrow{m}, n')$ be an edge in $E_R(G_1)$, where P_α is a predicate not in *MOLL* and defined by:

$$\begin{array}{rcl} P(in, out, \overrightarrow{nhb}) & \triangleq & (in = out) \vee \\ & & (\exists u, \overrightarrow{v}. \Sigma_0(in, u \cup \overrightarrow{v} \cup \overrightarrow{nhb}) * \Sigma_1(\overrightarrow{v}, \overrightarrow{nhb}) \ * \ P(u, out, \overrightarrow{nhb})) \end{array}$$

$$\Sigma_0(in, u \cup \overrightarrow{v} \cup \overrightarrow{nhb}) ::= in \mapsto \theta, \text{ where } \{(h_i, v_i) \mid 1 \leq i \leq k\} \subseteq \theta \subseteq \{(f, w) \mid f \in RefFlds, w \in V\}$$

$$\Sigma_1(\overrightarrow{v}, \overrightarrow{nhb}) ::= Q_1(v_1, b_1, \overrightarrow{b_1}) * \ldots * Q_k(v_k, b_k, \overrightarrow{b_k}) \text{ with } b_i, \overrightarrow{b_i} \subseteq \overrightarrow{nhb}, \text{ and } Q_i \in \mathcal{P}, \text{ for any } i.$$

Intuitively, $P(in, out, \overrightarrow{nhb})$ describes a list segment from *in* to *out* where every element points to k nested lists described by Q_i, $1 \leq i \leq k$. Note that the list described by Q_i is referenced by the field h_i, it starts in v_i and ends in b_i.

The definition of h requires that there exists a sequence of nodes $\pi = \pi_1, \pi_2, \ldots, \pi_t$ in G_2 s.t. $\pi_1 = h(n)$, $\pi_t = h(n')$, and for every $1 \leq i \leq t-1$, one of the following holds:

- let V_i be the set of nodes, which is the union of π_i, π_{i+1}, and all the other nodes connected by points-to edges to π_i or π_{i+1}. Then, $E_P(G_2)$ contains some set of edges between the nodes in V_i, which prove that $\Sigma_0(in, u \cup \overrightarrow{v} \cup \overrightarrow{nhb})[in \mapsto x_i, u \mapsto x_{i+1}]$ holds, where x_i is a variable labeling π_i and x_{i+1} is a variable labeling π_{i+1}, that is, for any $(f, w) \in \theta$, either
  - $w = u$ and then $E_P(G_2)$ contains an edge (π_i, f, π_{i+1}), or
  - $w \in \overrightarrow{nhb}$ and then $E_P(G_2)$ contains an edge (π_i, f, v), where $v \in V(G_2)$ is labeled by w, or
  - $w \in \overrightarrow{v}$ and then, there exists a node $n_w \in V(G_2)$ such that $(\pi_i, f, n_w) \in E_P(G_2)$ (note that this node is unique).

  Moreover, for any $1 \leq j \leq k$, there exists a homomorphism h_j from G'_j, the *NOLL* graph of $Q_j(y_j, b_j, \overrightarrow{b_j})$, where y_j is a variable labeling n_{v_j}, to G_2 (by definition, h_j will map the node labeled by y_j to n_{v_j}). For any $1 \leq j \leq k$, let $edge_j$ be the only edge of G'_j. The fact that the predicates $Q_j(y_j, b_j, \overrightarrow{b_j})$ are $*$ separated in the definition of P is enforced by: for all $j \neq j'$, for all edges a in $used(edge_j)$ and b in $used(edge_{j'})$, $(a, b) \in \Omega_*(G_2)$.
  We define $used(e)_i$ as the union of (1) the set of points-to edges in $E_P(G_2)$ used to prove that $\Sigma_0(in, u \cup \overrightarrow{v} \cup \overrightarrow{nhb})[in \mapsto x_i, u \mapsto x_{i+1}]$ holds and (2) $used(edge_j)$, for all $1 \leq j \leq k$.
- there exists an edge $e' = (\pi_i, P'_\beta, \overrightarrow{m'}, \pi_{i+1})$ in $E_R(G_2)$ and $P'(x, \overrightarrow{z'}, y) \Rightarrow P(x, \overrightarrow{z}, y)$, where x and y are some variables labeling π_i and π_{i+1}, resp., and z and z' are some

vectors of variables labeling $\overrightarrow{m}$ and $\overrightarrow{m'}$ such that $h(m[i]) = m'[j]$ implies $z[i] = z'[j]$. We define

$$used(e)_i = \{(e', f) \mid f \text{ is a record field in } \Sigma_0\}.$$

We define $used(e) = \bigcup_{1 \leq i \leq t-1} used(e)_i$.

Above we use the fact that the entailment between recursive predicates can be checked syntactically in polynomial time.

A (complete) *NOLL* graph is called *of level n* if it represents a *NOLL* formula over a set of predicates $\mathcal{P}$ with at most n levels of nesting. Note that *NOLL* graphs denoting *MOLL* formulas are of level 0. The definition of an homomorphism is recursive in the sense that the definition of an homomorphism between two *NOLL* graphs of level n uses the definition of an homomorphism between two *NOLL* graphs of level $n-1$.

Next, we give the formal definition of the substitution Γ for set of locations variables in $\Lambda(G_1)$ to terms over set of locations variables in $\Lambda(G_2)$, defined by the mapping of edges in $E_R(G_1)$ to paths in G_2. Thus, let α be a variable in $\Lambda(G_1)$. If α is not bounded to a spatial atom then $\Gamma(\alpha) = \alpha$. Otherwise, suppose that there exists $e = (n, P_\alpha, \overrightarrow{m}, n')$ in $E_R(G_1)$. Then, $\Gamma(\alpha)$ is the union of all set of locations variables bounded to spatial constraints denoted by $E_R(G_2)$-edges in $used(e)$ and all singletons $\{x\}$, where x is variable labeling the left-end of a points-to edge in $used(e)$. Formally,

$$\Gamma(\alpha) = \bigcup_{e'=(\pi_i, P'_\beta, \overrightarrow{m'}, \pi_{i+1}) \in E_R(G_2) \cap used(e)} \beta \cup \bigcup_{\substack{e' = (n, f, n') \in E_P(G_2) \cap used(e) \\ x = \ell(G_2)(n)}} \{x\}.$$

A.6 Properties of *NOLL* graph homomorphisms

Note that any *NOLL* interpretation $(C = (S, H), J)$ can be represented by a *NOLL* graph $(V, E_P, E_R, E_D, \ell, \Omega_*, \Lambda)$ where

- V is the set of locations used in the definition of H and the image of J,
- E_P is defined according to H, i.e., $(l, f, l') \in E_P$ iff $H(l, f) = l'$,
- $E_R = \emptyset$,
- E_D contains any pair of distinct locations,
- $\ell(x) = l$ iff $S(x) = l$ and $\ell(x_l) = l$, with $x_l \notin \mathtt{dom}(S)$, for every location l which is not in the image of S,
- Ω_* contains any pair of distinct edges, and
- Λ is the conjunction:

$$\bigwedge_{\alpha \in \mathtt{dom}(J)} \alpha = \bigcup_{l \in J(\alpha)} \{l\}.$$

In Section 6.3 we have defined homomorphisms from *NOLL* graphs to complete *NOLL* graphs. In the following, we also consider homomorphisms between *NOLL* graphs which are defined in a similar way (in the definition of $used(e)$, for some edge e, we consider that $\Delta(G_2)$ is empty).

Lemma 9. *Let $(C = (S, H), J)$ be a NOLL interpretation and φ a NOLL formula. If there exists an homomorphism from the NOLL graph of φ to the NOLL graph of (C, J) then, (C, J) is a model of φ.*

Proof. First, suppose that there exists an homomorphism h from the *NOLL* graph of φ to the *NOLL* graph of (C,J). Note that this homomorphism is unique and it is given by the labeling with location variables. We prove that (C,J) is a model of φ by induction on the length of φ:

- when φ is an atomic formula the claim is straightforward.
- let $\varphi = \Pi \wedge (\Sigma * x \mapsto \{(f,y)\}) \wedge \Lambda$. We have to show that there exists C_1 and C_2 such that $(C_1,J) \models \Pi \wedge \Sigma \wedge \Lambda$, $(C_2,J) \models x \mapsto \{(f,y)\}$, and $C = C_1 * C_2$. The homomorphism h requires that there exists an edge $(h([x]), f, h([y]))$ in the *NOLL* graph of (C,J) which corresponds to the fact that there exist two locations $l, l' \in Loc$ such that $S(x) = l$, $S(y) = l'$, and $H(l,f) = l'$. We define $C_1 = (S, H_1)$, where H_1 is defined exactly as H except for (l,f) where it is undefined, and $C_2 = (S_2, H_2)$, where H_2 is defined only for (l,f) by $H_2(l,f) = l'$, $S_2(x) = l$, and $S_2(y) = l'$. By definition, we have that $C = C_1 * C_2$ and $(C_2,J) \models x : f \mapsto y$. Furthermore, the $*$ separation between Σ and $x \mapsto \{(f,y)\}$ implies that the edge $(h([x]), f, h([y]))$ is not involved in other paths from the *NOLL* graph of (C,J) corresponding to edges in φ. Thus, h remains an homomorphism from $\Pi \wedge \Sigma \wedge \Lambda$ to (C_1,J) and consequently, by the induction hypothesis, $(C_1,J) \models \Pi \wedge \Sigma \wedge \Lambda$.
- let $\varphi = \Pi \wedge (\Sigma * P_\alpha(x,y,\overrightarrow{z})) \wedge \Lambda$. We have to show that there exists C_1 and C_2 such that $(C_1,J) \models \Pi \wedge \Sigma \wedge \Lambda$, $(C_2,J) \models P_\alpha(x,y,\overrightarrow{z})$, and $C = C_1 * C_2$. The homomorphism h requires that there exists a sequence of nodes $\pi = (\pi_1, \pi_2, \ldots, \pi_k)$ in the *NOLL* graph of (C,J) satisfying all the properties in Section A.5. Since (C,J) is a concrete model, there are only points-to edges between every two successive nodes in π. The proof proceeds as in the previous case, where the domain of S^{C_2} contains all the pairs (l,f) corresponding to points-to edges in $used(P_\alpha(x,y,\overrightarrow{z}))$ and the domain of $S^{C_1} = \mathtt{dom}(S^C) \setminus \mathtt{dom}(S^{C_2})$. As in the previous case, h remains an homomorphism from $\Pi \wedge \Sigma \wedge \Lambda$ to the *NOLL* graph of (C_1,J).
- the cases $\varphi = \Pi \wedge (\Sigma *_w x \mapsto \{(f,y)\}) \wedge \Lambda$ and $\varphi = \Pi \wedge (\Sigma *_w P_\alpha(x,y,\overrightarrow{z})) \wedge \Lambda$ can be handled in a similar manner.
- let $\varphi = \Pi \wedge \Sigma \wedge (\Lambda \wedge t \subseteq t')$. We have to show that (C,J) is a model of φ. First, note that the normal form of $\Pi \wedge \Sigma \wedge \Lambda$ may contain fewer edges that represent recursive predicates. This is because we have possibly removed a constraint that shows that the data structure described by some recursive predicate is included in the data structure defined by some other recursive predicate. Thus, h remains again an homomorphism from the normal form of $\Pi \wedge \Sigma \wedge \Lambda$ to (C,J) and by the induction hypothesis, $(C,J) \models \Pi \wedge \Sigma \wedge \Lambda$. Then, the fact that h is an homomorphism from φ to (C,J) implies directly that $t \subseteq t'$ is satisfied in (C,J), which concludes the proof of this case.
- the cases $\varphi = (\Pi \wedge x = y) \wedge \Sigma \wedge \Lambda$ and $\varphi = (\Pi \wedge x \neq y) \wedge \Sigma \wedge \Lambda$ are easy to prove. □

Lemma 10. *Let $(C = (S,H),J)$ be a NOLL interpretation and φ a NOLL formula. If (C,J) is a model of φ then, there exists an homomorphism from the NOLL graph of φ to the NOLL graph of (C,J).*

Proof. Let G_1 be the *NOLL* graph of (C,J) and G_2 the *NOLL* graph of φ.

We first prove that for any node n_2 in G_2, there exists a unique node n_1 in G_1 such that $vars_{G_2}(n_2) \subseteq vars_{G_1}(n_1)$. Suppose that this is not true for some node n_2 in G_2.

Then, there exist two location variables x and y which label different nodes in G_1 and the same node in G_2 (the fact that they label the same node in G_2 means that the equality $x = y$ is an atom of φ). But, this is not possible because (C,J) is a model of φ. The fact that the node n_1 is unique follows from the definition of *NOLL* interpretations where the interpretation of a location variable is a unique location.

We define a mapping $h : V(G_2) \to V(G_1)$ such that, for any $n_2 \in V(G_2)$, $h(n_2)$ is the unique node in G_1 such that $vars_{G_2}(n_2) \subseteq vars_{G_1}(n_1)$.

It can be proved by induction on the length of φ that h is an homomorphism from G_2 to G_1. We describe only one of the cases, the others being similar:

- let $\varphi = \Pi \wedge (\Sigma * x \mapsto \{(f,y)\}) \wedge \Lambda$. The semantics of $*$ implies that there exists C_1 and C_2 such that $(C_1,J) \models \Pi \wedge \Sigma \wedge \Lambda$, $(C_2,J) \models x \mapsto \{(f,y)\}$, and $C = C_1 * C_2$. By the induction hypothesis, we obtain that the projection of h on the variables in $\Pi \wedge \Sigma \wedge \Lambda$ is an homomorphism from the *NOLL* graph of $\Pi \wedge \Sigma \wedge \Lambda$ to the *NOLL* graph of (C_1,J) (note that if φ is in normal form then so is $\Pi \wedge \Sigma \wedge \Lambda$ – by removing one points-to constraint, we can not deduce new (in)equalities). We also obtain that there exists an homomorphism h' from the *NOLL* graph of $x \mapsto \{(f,y)\}$ to the *NOLL* graph of (C_2,J). This clearly implies that the mapping h defined above is an homomorphism from the *NOLL* graph of φ to the *NOLL* graph of (C,J). □

The following lemma is also an easy consequence of the homomorphism definition.

Lemma 11. *Let G_1, G_2, and G_3 be three NOLL graphs. If h is a homomorphism from G_1 to G_2 and h' is a homomorphism from G_2 to G_3 then $h' \circ h$ is a homomorphism from G_1 to G_3.*

A.7 Correctness of `CheckEntl`

Proposition 2. *Let φ and φ' be two NOLL formulas. If there exists a homomorphism h from the NOLL graph of φ' to the NOLL graph of φ then $\varphi \Rightarrow \varphi'$.*

Proof. Let (C,J) be a model of φ. By Lemma 9, there exists a homomorphism h' from the *NOLL* graph of φ to the *NOLL* graph of (C,J). Then, by Lemma 11, $h' \circ h$ is a homomorphism from the *NOLL* graph of φ' to the *NOLL* graph of (C,J), which, by Lemma 10, implies that (C,J) is a model of φ'. □

Theorem 6. *Let φ and φ' be two NOLL formulas in normal form, G_1 the complete NOLL graph of φ, G_2 the NOLL graph of φ', and $h : V(G_2) \rightharpoonup V(G_1)$ s.t. $vars_{G_2}(n) \subseteq vars_{G_1}(h(n))$, for any $n \in V(G_2)$. Then, $\varphi \Rightarrow \varphi'$ iff h is total and a homomorphism from G_2 to G_1.*

Proof. First, suppose that h is total and a homomorphism from G_2 to G_1. The proof of the fact that $\varphi \Rightarrow \varphi'$ is very similar to the one of Proposition 2. The only difference comes from the fact that h is a homomorphism from the *NOLL* graph of φ' to the *complete NOLL* graph of φ. Thus, let (C,J) be a model of φ. Recall that h maps edges of G_2 to paths of G_1 that under certain restrictions. Let E_1 be the set of edges in $used(e)$, for some edge $e \in E_P(G_2) \cup E_R(G_2)$. Note that (C,J) is also a model of

the constraints denoted by the subgraph G_1' of G_1 where $E_P(G_1') = E_P(G_1) \cap E_1$ and $E_R(G_1') = E_R(G_1) \cap E_1$ (the other components of G_1' are defined exactly as in G_1). Moreover, h is a homomorphism from G_2 to G_1'. By Lemma 9, there exists a homomorphism h' from G_1' to the *NOLL* graph of (C,J). Then, by Lemma 11, $h' \circ h$ is a homomorphism from the *NOLL* graph of φ' to the *NOLL* graph of (C,J), which, by Lemma 10, implies that (C,J) is a model of φ'.

Now, suppose by contradiction that $\varphi \Rightarrow \varphi'$ and h is not total, that is, there exists $n \in V(G_2)$ such that for all $n' \in V(G_1)$, we have that $vars_{G_2}(n) \not\subseteq vars_{G_1}(n')$. Then, there exist two location variables x and y which label different nodes in G_1 and the same node in G_2. Since φ is in normal form (it contains all the implicit equalities) there exists some model $(C = (S,H),J)$ of φ s.t. $S(x) \neq S(y)$. It can be easily seen that (C,J) is not a model of φ', which contradicts the hypothesis.

Now, suppose that h is total but not an homomorphism from G_2 to G_1. It can be proved that if one of the conditions from Section 6.3 is not satisfied by h then $\varphi \Rightarrow \varphi'$ doesn't hold. We give only some representative cases:

- suppose that there exist two nodes $[x]$ and $[y]$ in G_2 such that $([x],[y]) \in E_D(G_2)$ and $(h([x]),h([y])) \notin E_D(G_1)$. The latter together with the fact that φ is in normal form (that is, φ contains all the implied inequalities) implies that there exists a model $(C = (S,H),J)$ of φ where $S(y) = S(x)$. Clearly, (C,J) is not a model of φ' which contradicts the hypothesis.
- suppose that there exist two nodes $[x]$ and $[y]$ in G_2 such that $([x],f,[y]) \in E_P(G_2)$ and $(h([x]),f,h([y])) \notin E_P(G_1)$. First, suppose that x and y have the same type. There are several cases to consider:
  1. suppose that there exists no edge $([x],P_\alpha,[\overrightarrow{z}],[y])$ in G_1 such that $f \in RefFlds_0(P)$. Let $(C = (S,H),J)$ be a model of φ. If $H(S(x),f) \neq S(y)$ then (C,J) is clearly not a model of φ'. Otherwise, we have two cases:
     (a) the link $H(S(x),f) = S(y)$ is not included in a list segment specified by a spatial constraint in φ. Then, let (C',J) be a model of φ where C' is obtained from C by removing $(S(x),f)$ from the domain of H. Clearly, (C',J) is a model of φ but it is not a model of φ'.
     (b) otherwise, if the link $H(S(x),f) = S(y)$ is included in a list segment specified by a spatial constraint in φ then, the fact that G_1 is a complete *NOLL* graph of φ implies that there exists a model (C',J) of φ where x does not belong anymore to this list segment (we use the fact that G_1 contains all the implicit spatial constraints). This model is obtained by introducing a fresh location which replaces the interpretation of x in this list segment. Clearly, (C',J) is not a model of φ'.
  2. otherwise, the fact that G_1 contains only recursive predicate edges $([x],P_\alpha,[\overrightarrow{z}],[y])$ with $f \in RefFlds_0(P)$ (and no points-to edges) implies that there exists a model (C,J) of φ, that contains a path of length two between x and y with edges labeled by f (we use the fact that the recursive predicates are kept in the complete *NOLL* graph only if there exist models where they describe non-empty list segments of length greater than two). Clearly, (C,J) is not a model of φ'.

Now, suppose that x and y have different types and let $(C = (S,H),J)$ be a model of φ. If $H(S(x),f) \neq S(y)$ then (C,J) is clearly not a model of φ'. Otherwise, there are two cases similar to 1(a) and 1(b) above. □

A.8 Program verification using *NOLL*

This section gives an example of verification conditions checked by our decision procedure. The example is taken from the Nagios network monitoring software and concerns the data structures described in Sec. 1, i.e., the hash table sharing all its elements with a singly-linked list. To perform the verification, we work with the abstraction used in Sec. 2, i.e., we use a linked list to represent the array structure in the hash table. Therefore, the set of predicates $\mathcal{P}$ is the one defined by eq. (2.2)–(2.4).

The data structures used are declared in Fig. 11a: `sll_t` for the overlaid linked list and `htb_t` for the list of lists representing the hash table. The procedure of adding a cell in the overlaid data structures is given in Fig. 11b.

```
1:  typedef struct _sll_t  sll_t;
2:  typedef struct _htb_t  htb_t;
3:  struct _htb_t {
4:    htb_t* g;
5:    sll_t* h;
6:  };
7:  struct _sll_t {
9:    sll_t* f;
10:   sll_t* s;
11: };

/* global variables */
12: htb_t* x = ...; /* global htable */
13: sll_t* z = ...; /* global list */
```

(a)

```
/* add a value to the list */
14: sll_t* v = alloc_sll();

/* add v to hash table x */
15: htb_t* xi = ...; /* compute the entry */
16: sll_t* vi = xi->h; /* get its list */
17: v->s = vi; /* insert v */
18: xi->h = v;

/* add v to global list z */
19: sll_t* zi = ...; /* compute the entry */
20: v->f = zi->f; /* insert after */
21: zi->f = v;
22:
```

(b)

Fig. 11: Adding a cell in a hash table x overlaid with a list z.

We denote by ϕ_ℓ the *NOLL* formula used to annotate the line ℓ of the program in Fig. 11b, and by $\texttt{post}_\ell$ the post-condition of executing the statement at line ℓ starting from φ_ℓ. Some of the pre- and post-condition used for the verification are given in Fig. 12.

The specification at line 14 is the invariant of the overlaid data structure: from x starts a hash table and from z a linked lists such that the set β of the list objects is exactly the set of objects of type `sll_t` used in the hash table. The allocation statement at line 14 introduces a location labeled by v which is separated from the data structures in the heap. (The constraint $x \notin \beta$ is implied by the semantics of the $*$ operator.)

From line 15 to line 18, the program works only on the hash table data structure. We can then use specifications that concerns only this part of the heap. At the line 19, the frame rule for the field separating conjunction allows us to obtain a proof for the whole overlaid data structure. The same manipulation is done for the lines 19–21, where the programs works only on the list data structure. The same frame rule is used to restore the data structure invariant at line 22.

The first verification condition to be checked is $\mathtt{post}_{14} \Rightarrow \phi_{15}$. The computation of the normal form of $\mathtt{post}_{14}$ introduces the implicit constraint $\mathit{LowList}_\gamma(z, \mathtt{NULL})$ with $\gamma \in \alpha(\mathtt{sll\_t})$, but this edge is not used to map edges from the *NOLL* graph of ψ_{15} to the complete *NOLL* graph of $\mathtt{post}_{14}$.

The computation of the post-conditions introduces additional set of location variables when the statement unfolds a list segment, e.g., α_1 and α_2 in $\mathtt{post}_{18}$. The entailment $\mathtt{post}_{18} \Rightarrow \mathtt{post}'_{18}$ follows due to the mapping of the edge corresponding to $\mathit{Hash}_{\alpha'}(z, \mathtt{NULL})$ on the nested path from x to xi, from xi to $\mathtt{NULL}$ by the *LowList*, from xi to xi' (the successor of xi by field g), and from xi' to $\mathtt{NULL}$. This mapping allows to prove the sharing constraint. A similar computation is done to prove the entailment $\mathtt{post}_{21} \Rightarrow \mathtt{post}'_{21}$ but on simpler paths for the *List* list segments.

$$
\begin{aligned}
\phi_{14} &:= x \neq \mathtt{NULL} \wedge z \neq \mathtt{NULL} \wedge \mathit{Hash}_\alpha(x, \mathtt{NULL}) *_w \mathit{List}_\beta(z, \mathtt{NULL}) \wedge \beta = \alpha(\mathtt{sll\_t}) \\
\mathtt{post}_{14} &:= x \neq \mathtt{NULL} \wedge z \neq \mathtt{NULL} \wedge v \neq \mathtt{NULL} \wedge \\
&\quad (\mathit{Hash}_\alpha(x, \mathtt{NULL}) *_w \mathit{List}_\beta(z, \mathtt{NULL})) * v \mapsto \{(s, \mathtt{NULL}); (f, \mathtt{NULL})\} \wedge \\
&\quad \beta = \alpha(\mathtt{sll\_t}) \wedge v \notin \beta \\
\phi_{15} &:= x \neq \mathtt{NULL} \wedge v \neq \mathtt{NULL} \wedge \big(\mathit{Hash}_\alpha(x, \mathtt{NULL}) * v \mapsto \{(s, \mathtt{NULL})\}\big) \wedge v \notin \alpha \\
\mathtt{post}_{15} &:= x \neq \mathtt{NULL} \wedge xi \neq \mathtt{NULL} \wedge v \neq \mathtt{NULL} \wedge \\
&\quad \big(\mathit{Hash}_{\alpha_1}(x, xi) * \mathit{Hash}_{\alpha_2}(xi, \mathtt{NULL}) * v \mapsto \{(s, \mathtt{NULL})\}\big) \wedge \\
&\quad \alpha = \alpha_1 \cup \alpha_2 \wedge v \notin \alpha \\
\mathtt{post}_{18} &:= x \neq \mathtt{NULL} \wedge xi \neq \mathtt{NULL} \wedge v \neq \mathtt{NULL} \wedge \\
&\quad \big(\mathit{Hash}_{\alpha_1}(x, xi) * \mathit{Hash}_{\alpha_3}(xi', \mathtt{NULL}) * xi \mapsto \{(g, xi'), (h, v)\} \\
&\quad\ * v \mapsto \{(s, vi)\} * \mathit{LowList}_{\beta'}(vi, \mathtt{NULL})\big) \wedge \\
&\quad \alpha = \alpha_1 \cup \alpha_3 \cup \{xi\} \cup \beta' \wedge v \notin \alpha \\
\mathtt{post}'_{18} &:= x \neq \mathtt{NULL} \wedge v \neq \mathtt{NULL} \wedge \mathit{Hash}_{\alpha'}(x, \mathtt{NULL}) \wedge \alpha \cup \{v\} = \alpha' \wedge v \notin \alpha \\
\phi_{19} &:= z \neq \mathtt{NULL} \wedge v \neq \mathtt{NULL} \wedge \big(\mathit{List}_\beta(z, \mathtt{NULL}) * v \mapsto \{(f, \mathtt{NULL})\}\big) \wedge v \notin \beta \\
\mathtt{post}_{19} &:= z \neq \mathtt{NULL} \wedge zi \neq \mathtt{NULL} \wedge v \neq \mathtt{NULL} \wedge \\
&\quad \big(\mathit{List}_{\beta_1}(z, zi) * \mathit{List}_{\beta_2}(zi, \mathtt{NULL}) * v \mapsto \{(f, \mathtt{NULL})\}\big) \wedge \beta = \beta_1 \cup \beta_2 \wedge v \notin \beta \\
\mathtt{post}_{21} &:= z \neq \mathtt{NULL} \wedge zi \neq \mathtt{NULL} \wedge v \neq \mathtt{NULL} \wedge \\
&\quad \big(\mathit{List}_{\beta_1}(z, zi) * \mathit{List}_{\beta'_2}(zi', \mathtt{NULL}) * zi \mapsto \{(f, v)\} * v \mapsto \{(f, zi')\}\big) \wedge \\
&\quad \beta = \beta_1 \cup \beta'_2 \cup \{zi\} \wedge v \notin \beta \\
\mathtt{post}'_{21} &:= z \neq \mathtt{NULL} \wedge v \neq \mathtt{NULL} \wedge \mathit{List}_{\beta'}(z, \mathtt{NULL}) \wedge \beta \cup \{v\} = \beta'' \wedge v \notin \beta \\
\phi_{22} &:= x \neq \mathtt{NULL} \wedge z \neq \mathtt{NULL} \wedge v \neq \mathtt{NULL} \wedge \mathit{Hash}_{\alpha'}(x, \mathtt{NULL}) *_w \mathit{List}_{\beta''}(z, \mathtt{NULL}) \wedge \\
&\quad \beta'' = \alpha'(\mathtt{sll\_t}) \wedge v \in \beta''
\end{aligned}
$$

Fig. 12: Samples of pre and post-conditions for the example from Fig. 11.