Byte-Precise Verification of Low-Level List Manipulation

FIT BUT Technical Report Series

Kamil Dudka, Petr Peringer, and Tomáš Vojnar

Technical Report No. FIT-TR-2012-04
Faculty of Information Technology, Brno University of Technology

Last modified: April 13, 2013

Byte-Precise Verification of Low-Level List Manipulation*

Kamil Dudka, Petr Peringer, and Tomáš Vojnar

FIT, Brno University of Technology, IT4Innovations Centre of Excellence, Czech Republic

Abstract. We propose a new approach to shape analysis of programs with linked lists that use low-level memory operations. Such operations include pointer arithmetic, safe usage of invalid pointers, block operations with memory, reinterpretation of the memory contents, address alignment, etc. Our approach is based on a new representation of sets of heaps, which is to some degree inspired by works on separation logic with higher-order list predicates, but it is graph-based and uses a more fine-grained (byte-precise) memory model in order to support the various low-level memory operations. The approach was implemented in the Predator tool and successfully validated on multiple non-trivial case studies that are beyond the capabilities of other current fully automated shape analysis tools.

1 Introduction

Dealing with programs with pointers and dynamic linked data structures belongs among the most challenging tasks of formal analysis and verification due to a need to cope with infinite sets of reachable program configurations having the form of complex graphs. This task becomes even more complicated when considering low-level memory operations such as pointer arithmetic, safe usage of pointers with invalid targets, block operations with memory, reinterpretation of the memory contents, or address alignment. Despite the rapid progress in the area of formal program analysis and verification, fully automated approaches capable of efficiently handling sufficiently general classes of dynamic linked data structures in the form used in low-level code are still missing.

In this paper, we propose a new fully automated approach to formal verification of list manipulating programs designed to cope with all of the above mentioned low-level memory operations. Our approach is based on a new representation of sets of heaps, which is to some degree inspired by works on separation logic with higher-order list predicates [1], but it is graph-based and uses a much more fine-grained memory model. In particular, our memory model allows one to deal with *byte-precise* offsets of fields of objects, offsets of pointer targets, as well as object sizes. Together with the new heap representation, we propose original algorithms for all the operations needed for a use of the new representation in a fully automated shape analysis. As our experiments show, these algorithms allow our analysis to successfully handle many programs on

* This work was supported by the Czech Science Foundation (project P103/10/0306), the Czech Ministry of Education (project MSM 0021630528), the EU/Czech IT4Innovations Centre of Excellence project CZ.1.05/1.1.00/02.0070, and the BUT FIT project FIT-S-12-1.

which other state-of-the-art fully automated approaches fail (by not terminating or by producing false positives or even false negatives).

In particular, we represent sets of heap graphs using the so-called *symbolic memory graphs* (SMGs) with two kinds of nodes: *objects* and *values*. Objects represent allocated memory and are further divided into *regions* representing individual memory areas and *list segments* encoding linked sequences of n or more regions uninterrupted by external pointers (for some $n \geq 0$). Values represent addresses and other data stored inside objects. Objects and values are linked by two kinds of edges: *has-value* edges from objects to values and *points-to* edges from value nodes representing addresses to objects. For efficiency reasons, we represent equal values by a single value node. We explicitly track sizes of objects, byte-precise offsets at which values are stored in them, and we allow pointers to point to objects with an arbitrary offset, i.e., a pointer can point *inside* as well as *outside* an object, not just at its beginning as in many current analyses.

We are capable of handling possibly cyclic, nested (with an arbitrary depth), and/or shared singly- as well as doubly-linked lists (for brevity, below, we concentrate on doubly-linked lists only). Our analysis can fully automatically recognise linking fields of the lists as well as the way they are possibly hierarchically nested. Moreover, the analysis can easily handle lists in the form common in system software (in particular, the Linux kernel), where list nodes are linked through the middle of them, pointer arithmetic is used to get to the beginning of the nodes, pointers iterating through such lists can sometimes safely point to unallocated memory, the forward links are pointers to structures while the backward ones are pointers to pointers to structures, etc.

To reduce the number of SMGs generated for each basic block of the analysed program, we propose a join operator working over SMGs. Our join operator is based on simultaneously traversing two SMGs while trying to merge the encountered pairs of objects and values according to a set of rules carefully tuned through many experiments to balance precision and efficiency (see Section 3.2 for details). Moreover, we use the join operator as the core of our abstraction, which is based on merging neighbouring objects (together with their sub-heaps) into list segments. This approach leads to a rather easy to understand and—according to our experiments—quite efficient abstraction algorithm. In the abstraction algorithm, the join is not applied to two distinct SMGs, but a single one, starting not from pairs of program variables, but the nodes to be merged. Further, we use our join operator as a basis for checking entailment on SMGs too (by observing which kind of pairs of objects and values are merged when joining two SMGs). In order to handle lists whose nodes *optionally* refer to some regions or sub-lists (which can make some program analyses diverge and/or produce false alarms [15]), our join and abstraction support the so-called 0/1 abstract objects.

Since on the low level, the same memory contents can be interpreted in different ways (e.g., via unions or type-casting), we incorporate into our analysis the so-called read, write, and join *reinterpretation*. In particular, we formulate general conditions on the reinterpretation operators that are needed for soundness of our analysis, and then instantiate these operators for the quite frequent case of dealing with blocks of nullified memory. Due to this, we can, e.g., efficiently handle initialization of structures with tens or hundreds of fields commonly allocated and nullified in practice through a single call of `calloc`, at the same time avoiding false alarms stemming from that some field

was not explicitly nullified. Moreover, we provide a support for block operations like `memmove` or `memcpy`. Further, we extend the basic notion of SMGs to support pointers having the form of not just a single address, but an interval of addresses. This is needed, e.g., to cope with address alignment or with list nodes that are equal up to their incoming pointers arrive with different offsets (as common, e.g., in memory allocators).

We have implemented the proposed approach in a new version of our tool Predator [7]. Predator automatically proves absence of various memory safety errors, such as invalid dereferences, invalid free operations, or memory leaks. Moreover, Predator can also provide the user with the derived shape invariants. Due to SMGs provide a rather detailed memory model, Predator produces fewer false alarms compared with other tools, and on the other hand, it can discover bugs that may be undetected by other state-of-the-art tools (as illustrated by our experimental results). In particular, Predator can discover out-of-bound dereferences (including stack smashing or buffer overflows) as well as nasty bugs resulting from dealing with overlapping blocks of memory in operations like `memcpy`. We have successfully validated the new version of Predator on a number of case studies, including various operations on lists commonly used in the Linux kernel as well as code taken directly from selected low-level critical applications (without any changes up to adding a test environment). In particular, we considered the memory allocator from the Netscape portable runtime (NSPR), used, e.g., in Firefox, and the `lvm2` logical volume manager. All of the case studies are available within the distribution of Predator.[1] To the best of our knowledge, many of our case studies are out of what other currently existing fully automated shape analysis tools can handle.

Related Work. Many approaches to formal analysis and verification of programs with dynamic linked data structures have been proposed. They differ in their generality, level of automation, as well as the formalism on which they are based. As said already above, our approach is inspired by the fully automated approaches [1, 16] based on separation logic with higher-order list predicates implemented in two well-known tools, namely, Space Invader and SLAyer [2]. Compared with them, however, we use a purely graph-based memory representation. In fact, a graph-based representation was used already in the older version of our tool Predator [7]. However, that representation was a rather straightforward graph-based encoding of separation logic formulae, which is no more the case for the representation proposed in this paper. Our new heap representation is much finer, which on one hand complicates its formalization, but on the other hand, it allows us to treat the different peculiarities of low-level memory manipulation. Moreover, somewhat surprisingly, despite our new heap representation is rather detailed, it still allowed us to propose algorithms for all the needed operations such that they are quite efficient. Indeed, the new version of Predator is much faster than the old one while at the same time producing fewer false positives. Compared with Space Invader and SLAyer, Predator based on the new memory representation and new algorithms is not only faster, but also terminates more often, avoids false positives and, in particular, is able to detect additional classes of program errors that the other tools silently ignore (as illustrated in the section on experiments).

[1] `https://github.com/kdudka/predator/tree/master/tests`

Both Space Invader and SLAyer provide some support for pointer arithmetic, but its systematic description is (to the best of our knowledge) not available, and moreover, the support seems to be rather basic as illustrated by our experimental results. The same is the case with some other fully automated tools for verification of programs with dynamic linked data structures based on other formalisms, such as Forester [8] based on automata. A support for pointer arithmetic in combination with separation logic appears in [5], which is, however, highly specialised for a particular kind of linked lists with variable length entries used in some memory allocators.

As for the memory model, probably the closest to our work is [10], which uses the so-called separating shape graphs. They support tracking of the size of allocated memory areas, pointers with byte-precise offsets wrt. addresses of memory regions, dealing with offset ranges, as well as multiple views on the same memory contents. A major difference is that [10] and the older work [6], on which [10] is based, use the so-called summary edges annotated by *user-supplied* data structure invariants to summarize parts of heaps of an unbounded size. This approach is more general in terms of the supported shapes of data structures but less automatic because the burden of describing the shape lies on the user. We use abstract objects (list segments) instead, which are capable of encoding various forms of hierarchically nested lists (very often used in practice) and are carefully designed to allow for *fully automatic* and *efficient* learning of the concrete forms of such lists (the concrete fields used, the way the lists are hierarchically nested, their possible cyclicity, possibly shared nodes, optional nodes, etc.). Also, the level of nesting is not fixed in advance—our list segments are labelled by an integral nesting level, which allows us to represent hierarchically nested data structures as flattened graphs. Finally, although [10] points out a need to reinterpret the memory contents upon reading/writing, the corresponding operations are not formalized there. One of our contributions is thus also a definition of read/write reinterpretation operators in a way that can be used by a fully automatic shape analysis algorithm.

A graph-based abstraction of sets of heap configurations is used in [11] too. On one hand, the representation allows one to deal even with tree-like data structures, but on the other hand, the case of doubly-linked lists is not considered. Further, the representation does not consider the low-level memory features covered by our symbolic memory graphs. Finally, the abstraction and join operations used in [11] are more aggressive and hence less precise than in our case.

The work [9], which is based on an instantiation of the TVLA framework [13], focuses on analysis of Linux-style lists, but their approach relies on an implementation-dependent way of accessing list nodes, instead of supporting pointer arithmetics, unions, and type-casts in a generic way. Finally, the work [14] provides a detailed treatment of low-level C features such as alignment, byte-order, padding, type-unsafe casts, etc. in the context of theorem proving based on separation logic. Our reinterpretation operators provide a lightweight treatment of these features designed to be used in the context of a fully automated analysis based on abstraction.

2 Symbolic Memory Graphs

We encode sets of program configurations using the so-called *symbolic memory graphs* (SMGs) together with a mapping from global (static) and local (stack) variables to nodes of the SMGs. In particular, SMGs have a form of node- and edge-labelled directed graphs. Below, we start by an informal description of SMGs, followed by their formalisation. For an illustration of the notions discussed below, we refer the reader to Fig. 1, which shows how SMGs represent cyclic Linux-style DLLs (with a head node without any data part, other nodes including the head structure as well as custom data, and with the next/prev pointers pointing *inside* list nodes, not at their beginning). Some more examples illustrating the notion of SMGs, including its use for encoding various low-level Linux-style lists, can be found in Appendix A.

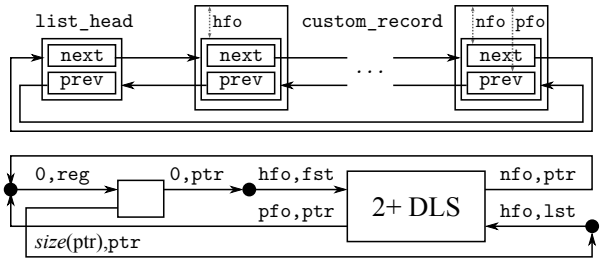

Fig. 1. A cyclic Linux-style DLL (top) and its SMG (bottom), with some SMG attributes left out for readability.

2.1 The Intuition behind SMGs

An SMG consists of two kinds of nodes: *objects* and *values* (in Fig. 1, they are represented by boxes and circles, respectively). Objects are further divided to *regions* and (doubly-linked) *list segments* (DLSs)[2]. A region represents a contiguous area of memory allocated either statically, on the stack, or on the heap. Each consistent SMG contains a special region called the *null object*, denoted #, which represents the target of `NULL`. DLSs arise from abstracting sequences of doubly-linked regions that are not interrupted by any external pointer. For example, in the lower part of Fig. 1, the left box is a region corresponding to the list head from the upper part of the figure whereas the right box is a DLS summarizing the sequence of `custom_record` objects from the upper part. Values are then used to represent *addresses* and other *data* stored in objects. All values are abstract in that we only distinguish whether they represent equal or possibly different concrete values. The only exception is the value 0 that is used to represent sequences of zero bytes of any length, which includes the zeros of all numerical types, the address of the null object, as well as nullified blocks of any size. Zero values are supported since they play a rather crucial role in C programs. In the future, a better distinction of values can be easily added.

SMGs have two kinds of *edges*: namely, *has-value edges* leading from objects to values and *points-to edges* leading from addresses to objects (cf. Fig. 1). Intuitively, the edges express that objects have values and addresses point to objects. Has-value edges are labelled by the *offset* and *type* of the *field* in which a particular value is stored within an object. Note that we allow the fields to overlap. This is used to represent different

[2] Our tool Predator supports *singly-linked list segments* too. Such segments can be viewed as a restriction of DLSs, and we omit them from the description in order to simplify it.

interpretations that a program can assign to a given memory area in order not to have to recompute them again and again. Points-to edges are labelled by an *offset* and a *target specifier*. The offset is used to express that the address from which the edge leads may, in fact, point *before, inside, or behind* an object. The target specifier is only meaningful for list segments to distinguish whether a given edge represents the address (or addresses) of the first, last, or each concrete region abstracted by the segment. The last option is used to encode links going to list nodes from the structures nested below them (e.g., in a DLL of DLLs, each node of the top-level list may be pointed from its nested list).

A key advantage of representing values (including addresses) as a separate kind of nodes is that a single value node is then used to represent values which are guaranteed to be equal in all concrete memory configurations encoded by a given SMG. Hence, distinguishing between *equal* values and *possibly different* values reduces to a simple identity check, not requiring a use of any prover. Thanks to identifying fields of objects by offsets (instead of using names of struct/union members), comparing their addresses for equality simplifies to checking identity of the address nodes. For example, `(x == &x->next)` holds iff `next` is the first member of the structure pointed by `x`, in which case both `x` and `&x->next` are guaranteed to be represented by a single address node in SMGs. Finally, the distinction of has-value and points-to edges saves some space since the information present on points-to edges would otherwise have to be copied multiple times for a single target.

Objects and values in SMGs are labelled by several *attributes*. First, each object is labelled by its *kind*, allowing one to distinguish regions and DLSs. Next, each object is labelled by its *size*, i.e., the amount of memory allocated for storing it. For DLSs, the size gives the size of their nodes. All objects and values have the so-called *nesting level* which is an integer specifying at which level of hierarchically nested structures the object or value appears (level 0 being the top level). All objects are further labelled by their *validity* in order to allow for safe pointer arithmetic over freed regions (which are marked invalid, but kept as long as there is some pointer to them).

Next, each DLS is labelled by the *minimum length* of the sequence of regions represented by it.[3] Further, each DLS is associated with the offsets of the *"next"* and *"prev" fields* through which the concrete regions represented by the segment are linked forward and backward[4]. Each DLS is also associated with the so-called *head offset* at which a sub-structure called a *list head* is stored in each list node (cf. Fig. 1). The usage of list heads is common in system software. They are predefined structures, typically containing the next/prev fields used to link list nodes. When a new list is defined, its node structure contains the list head as a nested structure, its nodes are linked by pointers pointing not at their beginning but inside of them (in particular, to the list head), and pointer arithmetic is used to get to the beginning of the actual list nodes.

Global and stack *program variables* are represented by regions like heap objects and can thus be manipulated in a similar way (including their manipulation via pointers, checking for out-of-bounds accesses leading to stack smashing, etc.). Regions corresponding to program variables are tagged by their names and hence distinguishable whenever needed (e.g., when checking for invalid frees of stack/global memory, etc.).

[3] Later, in Section 4, special list segments of length 0 or 1 are mentioned too.

[4] The names "next" and "prev" (i.e., previous) are used within our definition of list segments only. The concrete names of these fields in the programs being analysed are irrelevant.

2.2 Symbolic Memory Graphs

Let $\mathbb{B}$ be the set of Booleans, $\mathbb{T}$ a set of types, $size(t)$ the size of instances of a type $t \in \mathbb{T}$, $\texttt{ptr} \in \mathbb{T}$ a unique pointer type[5], $\mathbb{K} = \{\texttt{reg}, \texttt{dls}\}$ the set of kinds of objects (distinguishing regions and DLSs), and $\mathbb{S} = \{\texttt{fst}, \texttt{lst}, \texttt{all}, \texttt{reg}\}$ the set of points-to target specifiers. A *symbolic memory graph* is a tuple $G = (O, V, \Lambda, H, P)$ where:

- O is a finite set of objects including the special null object $\#$.
- V is a finite set of *values* such that $O \cap V = \emptyset$ and $0 \in V$.
- Λ is a tuple of the following labelling functions:
  - The kind of objects $kind : O \to \mathbb{K}$ where $kind(\#) = \texttt{reg}$, i.e., $\#$ is formally considered a region. We let $R = \{r \in O \mid kind(r) = \texttt{reg}\}$ be the set of regions and $D = \{d \in O \mid kind(d) = \texttt{dls}\}$ be the set of DLSs of G.
  - The nesting level of objects and values $level : O \cup V \to \mathbb{N}$.
  - The size of objects $size : O \to \mathbb{N}$.
  - The minimum length of DLSs $len : D \to \mathbb{N}$.
  - The validity of objects $valid : O \to \mathbb{B}$.
  - The head, next, and prev field offsets of DLSs $hfo, nfo, pfo : D \to \mathbb{N}$.
- H is a partial edge function $O \times \mathbb{N} \times \mathbb{T} \rightharpoonup V$ which defines *has-value edges* $o \xrightarrow{of,t} v$ where $o \in O$, $v \in V$, $of \in \mathbb{N}$, and $t \in \mathbb{T}$. We call (of, t) a *field* of the object o that stores the value v of the type t at the offset of.
- P is a partial injective edge function $V \rightharpoonup \mathbb{Z} \times \mathbb{S} \times O$ which defines *points-to edges* $v \xrightarrow{of,tg} o$ where $v \in V$, $o \in O$, $of \in \mathbb{Z}$, and $tg \in \mathbb{S}$ such that $tg = \texttt{reg}$ iff $o \in R$. Here, of is an offset wrt. the base address of o.[6] If o is a DLS, tg says whether the edge encodes pointers to the *first*, *last*, or *all* concrete regions represented by o.

We define the first node of a list segment such that the next field of the node points inside the list segment (and the last node such that the prev field of the node points inside the list segment). As already mentioned, the *all* target specifier is used in hierarchically nested list structures where each nested data structure points back to the node of the parent list below which it is nested. Fig. 2 illustrates how the target specifier affects the semantics of points-to edges (and the corresponding addresses): The DLS d is concretized to the two regions r_1 and r_2, and the nested abstract region r' to the two concrete regions r'_1 and r'_2. Note that if r' was not nested, i.e., if it had $level(r') = 0$, it would concretise into a single region pointed by both r_1 and r_2.

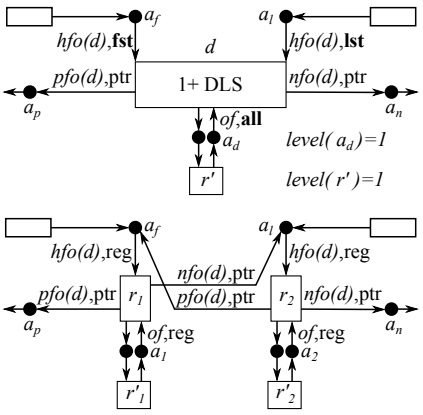

Fig. 2. An SMG and its possible concretisation for the case when the DLS d represents exactly two regions (only important attributes are shown).

[5] We assume $size(\texttt{ptr})$ to be a constant, which implies that separate verification runs are needed for verifying a program for target architectures using different address sizes.

[6] Note that the offset can even be negative, which happens, e.g., when traversing a Linux list.

Let $G = (O, V, \Lambda, H, P)$ be an SMG with a set of regions R and a set of DLSs D. We denote a DLS $d \in D$ of minimum length n, for which $\mathit{len}(d) = n$, as an $n+$ DLS. We use $\bot$ to denote cases where H or P is not defined. For any $v \in V$ for which $P(v) \neq \bot$, we denote by $\mathit{of}(P(v))$, $\mathit{tg}(P(v))$, and $o(P(v))$ the particular items of the triple $P(v)$. Further, for $o \in O$, we let $H(o) = \{H(o, \mathit{of}, t) \mid \mathit{of} \in \mathbb{N},\ t \in \mathbb{T},\ H(o, \mathit{of}, t) \neq \bot\}$. We let $A = \{v \in V \mid P(v) \neq \bot\}$ be the set of all *addresses* used in G. Next, a *path* in G is a sequence (of length one or more) of values and objects such that there is an edge between every two neighbouring nodes of the path. An object or value $x_2 \in O \cup V$ is *reachable* from an object or value $x_1 \in O \cup V$ iff there is a path from x_1 to x_2.

We call G *consistent* iff the following holds:

- *Basic consistency of objects.* The null object is invalid, has size and level 0, and its address is 0, i.e., $\mathit{valid}(\#) = \mathit{false}$, $\mathit{size}(\#) = \mathit{level}(\#) = 0$, and $0 \xrightarrow{0,\mathtt{reg}} \#$. All DLSs are valid, i.e., $\forall d \in D : \mathit{valid}(d)$. Invalid regions have no outgoing edges.
- *Field consistency.* Fields do not exceed boundaries of objects, i.e., $\forall o \in O\ \forall \mathit{of} \in \mathbb{N}\ \forall t \in \mathbb{T} :\ H(o, \mathit{of}, t) \neq \bot \Rightarrow \mathit{of} + \mathit{size}(t) \leq \mathit{size}(o)$.
- *DLS consistency.* Each DLS $d \in D$ has a next pointer and a prev pointer, i.e., there are addresses $a_n, a_p \in A$ s.t. $H(d, \mathit{nfo}(d), \mathtt{ptr}) = a_n$ and $H(d, \mathit{pfo}(d), \mathtt{ptr}) = a_p$ (cf. Fig. 2). The next pointer is always stored in memory before the prev pointer, i.e., the next and prev offsets are s.t. $\forall d \in D : \mathit{nfo}(d) < \mathit{pfo}(d)$. Points-to edges encoding links to the first and last node of a DLS d are always pointing to these nodes with the appropriate head offset, i.e., $\forall a \in A : \mathit{tg}(P(a)) \in \{\mathtt{fst}, \mathtt{lst}\} \Rightarrow \mathit{of}(P(a)) = \mathit{hfo}(d)$ where $d = o(P(a))$.[7] Finally, there is no cyclic path containing $0+$ DLSs (and their addresses) only in a consistent SMG since its semantics would include an address not referring to any object.
- *Nesting consistency.* Each nested object $o \in O$ of level $l = \mathit{level}(o) > 0$ has precisely one *parent DLS*, denoted $\mathit{parent}(o)$, that is of level $l - 1$ and there is a path from $\mathit{parent}(o)$ to o whose inner nodes are of level l and higher only (e.g., in Fig. 2, d is the parent of r'). Addresses with `fst`, `lst`, and `reg` targets are always of the same level as the object they refer to (as is the case for a_f, a_l, a_1, a_2 in Fig. 2), i.e., $\forall a \in A : \mathit{tg}(P(a)) \in \{\mathtt{fst}, \mathtt{lst}, \mathtt{reg}\} \Rightarrow \mathit{level}(a) = \mathit{level}(o(P(a)))$. On the other hand, addresses with the `all` target go up one level in the nesting hierarchy, i.e., $\forall a \in A : \mathit{tg}(P(a)) = \mathtt{all} \Rightarrow \mathit{level}(a) = \mathit{level}(o(P(a))) + 1$ (cf. a_d in Fig. 2). Finally, edges representing back-pointers to all nodes of a list segment can only lead from objects (transitively) nested below that segment (e.g., in Fig. 2, such an edge leads from region r' back to the DLS d, but it cannot lead from any other regions). Formally, for any $o, o' \in O$, $a \in H(o)$, $o(P(a)) = o'$, and $\mathit{level}(o) > \mathit{level}(o')$, $\mathit{tg}(P(a)) = \mathtt{all}$ iff $o' = \mathit{parent}^k(o)$ for some $k \geq 1$.

From now on, we assume working with consistent SMGs only. Let *GVar* be a finite set of global variables, *SVar* a countable set of stack variables such that $\mathit{GVar} \cap \mathit{SVar} = \emptyset$, and let $\mathit{Var} = \mathit{GVar} \cup \mathit{SVar}$. A symbolic program configuration (SPC) is a pair $C = (G, \nu)$ where G is an SMG with a set of regions R, and $\nu : \mathit{Var} \to R$ is a finite injective

[7] The last two requirements are not necessary, but they significantly simplify the below presented algorithms (e.g., the DLS materialisation given in Section 2.3).

map such that $\forall x \in \mathit{Var} : \ \mathit{level}(\nu(x)) = 0 \ \wedge \ \mathit{valid}(\nu(x))$. Note that ν gives the regions in which values of variables are stored, not directly the values themselves. We call each object o such that $\nu(x) = o$ for some $x \in \mathit{GVar}$ a *static object*, and each object o such that $\nu(x) = o$ for some $x \in \mathit{SVar}$ a *stack object*. All other objects are called *heap objects*. An SPC is called *garbage-free* iff all its heap objects are reachable from static or stack objects.

We define the *empty SMG* to consist solely of the null object, its address 0, and the points-to edge between them. The *empty SPC* then consists of the empty SMG and the empty variable mapping. An SMG $G' = (O', V', \Lambda', H', P')$ is a *sub-SMG* of an SMG $G = (O, V, \Lambda, H, P)$ iff (1) $O' \subseteq O$, (2) $V' \subseteq V$, and (3) H', P', and Λ' are restrictions of H, P, and Λ to O' and V', respectively. The sub-SMG of G *rooted at* an object or value $x \in O \cup V$, denoted G_x, is the smallest sub-SMG of G that includes x and all objects and values reachable from x. Given $F \subseteq \mathbb{N}$, the *F-restricted* sub-SMG of G rooted at an object $o \in O$ is the smallest sub-SMG of G that includes o and all objects and values reachable from o apart from the addresses $A_F = \{H(o, \mathit{of}, \mathtt{ptr}) \mid \mathit{of} \in F\}$ and nodes that are reachable from o through A_F only. Finally, the sub-SMG of G *nested below* $d \in D$, denoted $\widehat{G}_d$, is the smallest sub-SMG of G including d and all objects and values of level higher than $\mathit{level}(d)$ that are reachable from d via paths that, apart from d, consist exclusively of objects and values of a level higher than $\mathit{level}(d)$.

2.3 The Semantics of SMGs

We define the semantics of SMGs in two steps, namely, by first defining it in terms of the so-called memory graphs whose semantics is subsequently defined in terms of concrete memory images. In particular, a *memory graph* (MG) is defined exactly as an SMG up to it is not allowed to contain any list segments. An SMG then represents the class of MGs that can be obtained (up to isomorphism) by applying any number of times the following two transformations: (1) *materialisation* of fresh regions from DLSs (i.e., intuitively, "pulling out" concrete regions from the beginning or end of segments) and (2) *removal* of $0+$ DLSs (which may have become $0+$ due to the preceding materialisation).

Materialisation and Removal of DLSs. Let $G = (O, V, \Lambda, H, P)$ be an SMG with the sets of regions R, DLSs D, and addresses A. Let $d \in D$ be a DLS of level 0. Further, let $a_n, a_p \in A$ be the next and prev addresses of d, i.e., $H(d, \mathit{pfo}(d), \mathtt{ptr}) = a_p$ and $H(d, \mathit{nfo}(d), \mathtt{ptr}) = a_n$. The DLS d can be *materialised* as follows—for an illustration of the operation, see the upper part of Fig. 3:

1. G is extended by a fresh copy G'_r of the sub-SMG $\widehat{G}_d$ nested below d. In G'_r, d is replaced by a fresh region r such that $\mathit{size}(r) = \mathit{size}(d)$, $\mathit{level}(r) = 0$, and $\mathit{valid}(r) = \mathit{true}$. The nesting level of each object and value in G'_r (apart from r) is decreased by one.
2. Let $a_f \in A$ be the address pointing to the beginning of d, i.e., such that $P(a_f) = (\mathit{hfo}(d), \mathtt{fst}, d)$. If a_f does not exist in G, it is added. Next, A is extended by a fresh address a_d that will point to the beginning of the remaining part of d after the concretisation (while a_f will be the address of r). Finally, H and P are changed s.t. $P(a_f) = (\mathit{hfo}(d), \mathtt{reg}, r)$, $H(r, \mathit{pfo}(d), \mathtt{ptr}) = a_p$, $H(r, \mathit{nfo}(d), \mathtt{ptr}) = a_d$, $P(a_d) = (\mathit{hfo}(d), \mathtt{fst}, d)$, and $H(d, \mathit{pfo}(d), \mathtt{ptr}) = a_f$.

3. For any object o of $\widehat{G}_d$, let o' be the corresponding copy of o in G'_r (for $o = d$, let $o' = r$). For each field $(of, t) \in (\mathbb{N} \times \mathbb{T})$ of each object o in $\widehat{G}_d$ whose value is of level 0, i.e., $\mathit{level}(H(o, of, t)) = 0$, the corresponding field of o' in G'_r is set to the same value, i.e., the set of edges is extended such that $H(o', of, t) = H(o, of, t)$.
4. If $\mathit{len}(d) > 0$, $\mathit{len}(d)$ is decreased by one.

Next, let $d \in D$ be a DLS as above with the additional requirement of $\mathit{len}(d) = 0$ with the addresses a_n, a_p, a_f, and a_l defined as in the case of materialisation. The DLS d can be *removed* as follows—for an illustration, see the lower part of Fig. 3: (1) Each has-value edge $o \xrightarrow{of,t} a_f$ is replaced by the edge $o \xrightarrow{of,t} a_n$. (2) Each has-value edge $o \xrightarrow{of,t} a_l$ is replaced by the edge $o \xrightarrow{of,t} a_p$. (3) The subgraph $\widehat{G}_d$ is removed together with the addresses a_f, a_l, and the edges adjacent with the removed objects and values.

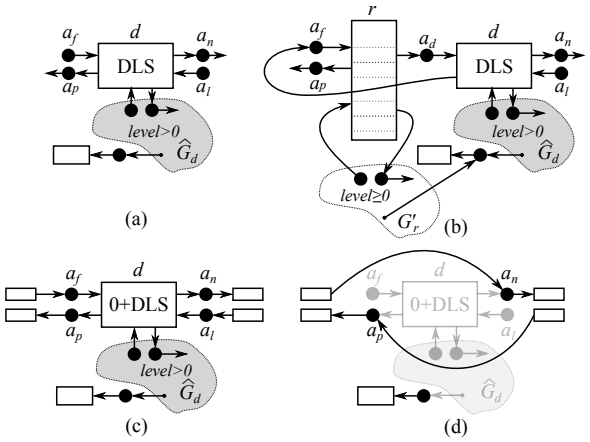

Fig. 3. Materialisation of a DLS: (a) input, (b) output (region r got materialised from DLS d). Removal of a DLS: (c) input, (d) output. Sub-SMGs $\widehat{G}_d$ and G'_r are highlighted without their roots.

Given an SMG $G = (O, V, \Lambda, H, P)$ with a set of DLSs D, we denote by $\mathit{MG}(G)$ the class of all MGs that can be obtained (up to isomorphism) by materializing each DLS $d \in D$ at least $\mathit{len}(d)$ times and by subsequently removing all DLSs.

Concrete Memory Images. The semantics of an MG $G = (R, V, \Lambda, H, P)$ is the set $\mathit{MI}(G)$ of *memory images* $\mu : \mathbb{N} \to \{0, \ldots, 255\}$ mapping *concrete addresses* to *bytes* such that there exists a function $\pi : R \to \mathbb{N}$, called a *region placement*, for which the following holds:

1. Only the null object is placed at address zero, i.e., $\forall r \in R : \ \pi(r) = 0 \Leftrightarrow r = \#$.
2. No two valid regions overlap, i.e., $\forall r_1, r_2 \in R : \ \mathit{valid}(r_1) \wedge \mathit{valid}(r_2) \Rightarrow \langle \pi(r_1), \pi(r_1) + \mathit{size}(r_1)) \cap \langle \pi(r_2), \pi(r_2) + \mathit{size}(r_2)) = \emptyset$.
3. Pointer fields are filled with the concrete addresses of the regions they refer to. Formally, for each pair of has-value and points-to edges $r_1 \xrightarrow{of_1, \mathtt{ptr}} a \xrightarrow{of_2, \mathtt{reg}} r_2$ in H and P, resp., $\mathit{addr}(\mathit{bseq}(\mu, \pi(r_1) + of_1, \mathit{size}(\mathtt{ptr}))) = \pi(r_2) + of_2$ where $\mathit{bseq}(\mu, p, \mathit{size})$ is the sequence of bytes $\mu(p)\mu(p+1)...\mu(p+\mathit{size}-1)$ for any $p, \mathit{size} > 0$, and $\mathit{addr}(\sigma)$ is the concrete address encoded by the byte sequence σ.
4. Fields having the same values are filled with the same concrete values (up to nullified blocks that can differ in their length), i.e., for every two has-value edges $r_1 \xrightarrow{of_1, t_1} v$ and $r_2 \xrightarrow{of_2, t_2} v$ in H, where $v \neq 0$, $\mathit{bseq}(\mu, \pi(r_1) + of_1, \mathit{size}(t_1)) = \mathit{bseq}(\mu, \pi(r_2) + of_2, \mathit{size}(t_2))$.
5. Finally, nullified fields are filled with zeros, i.e., for each has-value edge $r \xrightarrow{of,t} 0$ in H, $\mu(\pi(r) + of + i) = 0$ for all $0 \leq i < \mathit{size}(t)$.

For an SMG G, we let $MI(G) = \bigcup_{G' \in MG(G)} MI(G')$. Note that it may happen that it is not possible to find concrete values satisfying the needed constraints. In such a case, the semantics of an (S)MG is empty. Note also that we restrict ourselves to a flat address space, which is, however, sufficient for most practical cases. Finally, note that, for simplicity, we assume that each sequence of bytes of length $size(t)$ corresponds to some instance of the type t, which can be an indeterminate value in the worst case.

3 Operations on SMGs

In this section, we propose algorithms for all the operations on SMGs that are needed for their application in program verification. In particular, we discuss data reinterpretation, join of SMGs (which we use for entailment checking, too), abstraction, inequality checking, and symbolic execution of C programs. Due to limited space, the description is mostly informal. More details can be found in Appendices B–E.

Below, we denote by $I(of, t)$ the right-open integer interval $\langle of, of + size(t))$, and for a has-value edge $e : o \xrightarrow{of,t} v$, we write $I(e)$ as the abbreviation of $I(of, t)$.

3.1 Data Reinterpretation

SMGs allow fields of a single object to overlap and even to have the same offset and size, being distinguishable by their types only. In line with this feature of SMGs, we introduce the so-called *read reinterpretation* that can create multiple views (*interpretations*) of a single memory area without actually changing the semantics. On the other hand, if we write to a field that overlaps with other fields, we need to reflect the change of the memory image in the overlapping fields, for which the so-called *write reinterpretation* is used. These two operations form the basis of all operations reading and writing memory represented by SMGs. Apart from them, we also use *join reinterpretation* which is applied when joining two SMGs to preserve as much information shared by the SMGs as possible even when this information is not explicitly represented in the same way in both the input SMGs.

Defining reinterpretation for all possible data types (and all of their possible values) is hard (cf. [14]) and beyond the scope of this paper. Instead of that, we define minimal requirements that must be met by the reinterpretation operators so that our verification approach is sound. This allows different concrete instantiations of these operators to be used in the future. Currently, we instantiate the operators for the particular case of dealing with nullified blocks of memory, which is essential for handling low-level pointer manipulating programs that commonly use functions like `calloc()` or `memset()` to obtain large blocks of nullified memory.[8]

Read Reinterpretation. A read reinterpretation operator takes as input an SMG G with a set of objects O, an object $o \in O$, and a field (of, t) to be read from o such that $of + size(t) \leq size(o)$. The result is a couple (G', v) where G' is an SMG with a set

[8] Apart from the nullified blocks, our implementation also supports tracking of uninitialized blocks of memory and certain manipulations of null-terminated strings.

of has-value edges H' such that (1) $H'(o, of, t) = v \neq \bot$ and (2) $MI(G) = MI(G')$. The operator thus preserves the semantics of the SMG but ensures that it contains a has-value edge for the field being read. This edge can lead to a value already present in the SMG but also to a new value derived by the operator from the edges and values existing in the SMG. In the extreme case, a fresh, completely unconstrained value node can be added, representing an unknown value, which can, however, become constrained by the further program execution. In other words, read reinterpretation installs a new view on some part of the object o, but it cannot modify the semantics of the SMG in any way.

For the particular case of dealing with nullified memory, we use the following concrete read reinterpretation (cf. Appendix B.1). If G contains an edge $o \xrightarrow{of,t} v$, (G, v) is returned. Otherwise, if each byte of the field (of, t) is nullified by some edge $o \xrightarrow{of',t'} 0$ present in G, $(G', 0)$ is returned where G' is obtained from G by adding the edge $o \xrightarrow{of,t} 0$. Otherwise, (G', v) is returned with G' obtained from G by adding an edge $o \xrightarrow{of,t} v$ leading to a fresh value v (representing an unknown value). It is easy to see that with the current support of types and values in SMGs, this is the most precise read reinterpretation that is possible from the point of view of reading nullified memory.

Write Reinterpretation. The write reinterpretation operator takes as input an SMG G with a set of objects O, an object $o \in O$, a field (of, t) within o, i.e., such that $of + size(t) \leq size(o)$, and a value v that is to be written into the field (of, t) of the object o. The result is an SMG G' with a set of has-value edges H' such that (1) $H'(o, of, t) = v$ and (2) $MI(G) \subseteq MI(G'')$ where G'' is the SMG G' without the edge $e : o \xrightarrow{of,t} v$. In other words, the operator makes sure that the resulting SMG contains the edge e that was to be written while the semantics of G' without e over-approximates the semantics of G. Indeed, one cannot require equality here since the new edge may collide with some other edges, which may have to be dropped in the worst case.

For the case of dealing with nullified memory, we propose the following write reinterpretation (cf. Appendix B.2, which include an illustration too). If G contains the edge $e : o \xrightarrow{of,t} v$, G is returned. Otherwise, all has-value edges leading from o to a non-zero value whose fields overlap with (of, t) are removed. Subsequently, if $v = 0$, the edge e is added, and the obtained SMG is returned. Otherwise, all remaining has-value edges leading from o to 0 that define fields overlapping with (of, t) are split and/or shortened such that they do not overlap with (of, t), the edge e is added, and the resulting SMG is returned. Again, it is easy to see that this operator is the most precise write reinterpretation from the point of view of preserving information about nullified memory that is possible with the current support of types and values in SMGs.

3.2 Join of SMGs

Join of SMGs is a binary operation that takes two SMGs G_1, G_2 and returns an SMG G that is their common generalisation, i.e., $MI(G_1) \subseteq MI(G) \supseteq MI(G_2)$, and that satisfies the following further requirements intended to minimize the involved information loss: If both input SMGs are semantically equal, i.e., $MI(G_1) = MI(G_2)$, denoted $G_1 \simeq G_2$, we require the resulting SMG to be semantically equal to both the input ones, i.e., $MI(G_1) = MI(G) = MI(G_2)$. If $MI(G_1) \supset MI(G_2)$, denoted $G_1 \sqsupset G_2$, we require that

$MI(G) = MI(G_1)$. Symmetrically, if $MI(G_1) \subset MI(G_2)$, denoted $G_1 \sqsubset G_2$, we require that $MI(G) = MI(G_2)$. Finally, if the input SMGs are semantically incomparable, i.e., $MI(G_1) \not\supseteq MI(G_2) \ \wedge \ MI(G_1) \not\subseteq MI(G_2)$, denoted $G_1 \bowtie G_2$, no further requirements are put on the result of the join (besides the inclusion stated above, which is required for the soundness of our analysis). In order to distinguish which of these cases happens when joining two SMGs, we tag the result of our join operator by the so-called *join status* with the domain $\mathbb{J} = \{\simeq, \ \sqsupset, \ \sqsubset, \ \bowtie\}$ referring to the corresponding relations above. Moreover, we allow the join operation to fail if the incurred information loss becomes too big. Below, we give an informal description of our join operator, for a full description see Appendix C.

The basic idea of our join algorithm is the following. The algorithm simultaneously traverses a given pair of source SMGs and tries to join each pair of nodes (i.e., objects or values) encountered at the same time into a single node in the destination SMG. A single node of one SMG is not allowed to be joined with multiple nodes of the other SMG. This preserves the distinction between different objects as well as between at least possibly different values.[9]

The rules according to which it is decided whether a pair of objects simultaneously encountered in the input SMGs can be joined are the following. First, they must have the same size, validity, and in case of DLSs, the same head, prev, and next offsets. It is possible to join DLSs of different lengths as well as DLSs with regions (approximated as 1+ DLSs). The result is a DLS whose length is the minimum of the lengths of the joined DLSs (hence, e.g., joining a region with a 2+ DLS gives a 1+ DLS). The levels of the joined objects must also be the same up to the following case. When joining a sub-SMG nested below a DLS with a corresponding sub-SMG rooted at a region (restricted by ignoring the next and prev links), objects corresponding to each other appear on different levels: E.g., objects nested right below a DLS of level 0 are on level 1, whereas the corresponding objects directly referenced by a region of level 0 are on level 0 (since for regions, nested and shared sub-SMGs are not distinguished). This difference can, of course, increase when descending deeper in a hierarchically nested data structure as it is essentially given by the different numbers of DLSs passed on the different sides of the join. This difference is tracked by the join algorithm, and only the objects whose levels differ in the appropriate way are allowed to be joined.

When two objects are being joined, a *join reinterpretation* operator is used to ensure that they share the same set of fields and hence have the same number and labels of outgoing edges (which is always possible albeit sometimes for the price of introducing has-value edges leading to unknown values). A formalization of join reinterpretation is available in Appendix C.1, including a concrete join reinterpretation operator designed to preserve maximum information on nullified blocks in both of the objects being joined. The join reinterpretation allows the fields of the joined objects to be processed in pairs of the same size and type. As for joining values, we do not allow join-

[9] Two separately allocated objects are always different, values are only possibly different. Not to restrict the semantics, different objects or (possibly) different values cannot be changed into equal objects or values. Equal values could be changed into possibly different ones, but we currently do not allow this either since this would complicate the algorithm and we did not see any need for that in our case studies.

ing addresses with unknown values.[10] Moreover, the zero value cannot be joined with a non-zero value. Further, addresses can be joined only if the points-to edges leading from them are labelled by the same offset, and when they lead to DLSs, they must have the same target specifier. On the other hand, apart from the already above expressed requirement of not joining a single value in one SMG with several values in the other SMG, no further requirements are put on joining non-address values, which is possible since we currently track their equalities only.

To increase chances for successfully joining two SMGs, the basic algorithm from above is extended as follows. When a pair of objects cannot be joined and at least one of them is a DLS (call it d and the other object o), the algorithm proceeds as though o was preceded by a 0+ DLS d' that is up to its length isomorphic with d (including the not yet visited part of the appropriate sub-SMG nested below d). Said differently, the algorithm virtually inserts d' before o, joins d and d' into a single 0+ DLS, and then continues by trying to join o and the successor of d. This extension is possible since the semantics of a 0+ DLS includes the empty list, which can be safely assumed to appear anywhere, compensating a missing object in one of the SMGs.

Note, however, that the virtual insertion of a 0+ DLS implies a need to relax some of the requirements from above. For instance, one needs to allow a join of two different addresses from one SMG with one address in the other (the prev and next addresses of d get both joined with the address preceding o). Moreover, the possibility to insert 0+ DLSs introduces some non-determinism into the algorithm since when attempting to join a pair of incompatible DLSs, a 0+ DLS can be inserted into either of the two input DLSs, and we choose one of them. The choice may be wrong, but for performance reasons, we never backtrack. Moreover, we use the 0+ DLS insertion only when a join of two objects fails locally (i.e., without looking at their successors). When a pair of objects can be locally joined, but then the join fails on their successors, one could consider backtracking and trying to insert a 0+ DLS, which we again do not do for performance reasons (and we did not see a need for that in our cases studies so far).

The described join algorithm is used in two scenarios: (1) When joining garbage-free SPCs to reduce the number of SPCs obtained from different paths through the program, in which case the traversal starts from pairs of identical program variables. (2) As a part of the abstraction algorithm for merging a pair of neighbouring objects (together with the non-shared parts of the sub-SMGs rooted at them) of a doubly-linked list into a single DLS, in which case the algorithm is started from the neighbouring objects to be merged. In the join algorithm, the join status is computed on-the-fly. Initially, the status is set to $\simeq$. Next, whenever performing a step that implies a particular relation between G_1 and G_2 (e.g., joining a $0+$ DLS from G_1 with a $1+$ DLS from G_2 implies that $G_1 \sqsupset G_2$, assuming that the remaining parts of G_1 and G_2 are semantically equal), we appropriately update the join status.

[10] Allowing a join of an address and an unknown value could lead to a need to drop a part of the allocated heap in one of the SMGs (in case it was not accessible through some other address too), which we consider to be a too big loss of information.

3.3 Abstraction

Our abstraction is based on *merging uninterrupted sequences* of neighbouring objects, together with the $\{nfo, pfo\}$-restricted sub-SMGs rooted at them, into a single DLS. This is done by repeatedly applying a slight extension of the join algorithm on the $\{nfo, pfo\}$-restricted sub-SMGs rooted at the neighbouring objects. The sequences to be merged are identified by the so-called *candidate DLS entries* that consist of an object o_c and next, prev, and head offsets such that o_c has a neighbouring object with which it can be merged into a DLS linked through the given offsets. The abstraction is driven by the *cost* to be paid in terms of the loss of precision caused by merging certain objects and the sub-SMGs rooted at them (in particular, we distinguish joining of equal, entailed, or incomparable sub-SMGs). The higher the loss of precision is, the longer sequence of mergeable objects is required to enable a merge of the sequence.

In the extended join algorithm used in the abstraction (cf. Appendix C.8), the two simultaneous searches are started from two neighbouring objects o_1 and o_2 of the same SMG G that are the roots of the $\{nfo_c, pfo_c\}$-restricted sub-SMGs G_1, G_2 to be merged. The extended join algorithm constructs the sub-SMG $G_{1,2}$ that is to be nested below the DLS resulting from the join of o_1 and o_2. The extended join algorithm also returns the sets O_1, V_1 and O_2, V_2 of the objects and values of G_1 and G_2, respectively, whose join gives rise to $G_{1,2}$. Unlike when joining two distinct SMGs, the two simultaneous searches can get to a single node at the same time. Clearly, such a node is shared by G_1 and G_2, and it is therefore *not* included into the sub-SMG $G_{1,2}$ to be nested below the join of o_1 and o_2.

Below, we explain in more detail the particular steps of the abstraction. For the explanation, we fix an SPC $C = (G, \nu)$ where $G = (O, V, \Lambda, H, P)$ is an SMG with the sets of regions R, DLSs D, and addresses A.

Candidate DLS Entries. A quadruple $(o_c, hfo_c, nfo_c, pfo_c)$ where $o_c \in O$ and hfo_c, $nfo_c, pfo_c \in \mathbb{N}$ such that $nfo_c < pfo_c$ is considered a *candidate DLS entry* iff the following holds: (1) o_c is a valid heap object. (2) o_c has a neighbouring object $o \in O$ with which it is doubly-linked through the chosen offsets, i.e., there are $a_1, a_2 \in A$ such that $H(o_c, nfo_c, \texttt{ptr}) = a_1$, $P(a_1) = (hfo_c, tg_1, o)$ for $tg_1 \in \{\texttt{fst}, \texttt{reg}\}$, $H(o, pfo_c, \texttt{ptr}) = a_2$, and $P(a_2) = (hfo_c, tg_2, o_c)$ for $tg_2 \in \{\texttt{lst}, \texttt{reg}\}$.

Longest Mergeable Sequences. The *longest mergeable sequence* of objects given by a candidate DLS entry $(o_c, hfo_c, nfo_c, pfo_c)$ is the longest sequence of distinct valid heap objects whose first object is o_c, all objects in the sequence are of level 0, all DLSs that appear in the sequence have hfo_c, nfo_c, pfo_c as their head, next, prev offsets, and the following holds for any two neighbouring objects o_1 and o_2 in the sequence (for a formal description, cf. Appendix D): (1) The objects o_1 and o_2 are doubly linked through their nfo_c and pfo_c fields. (2) The objects o_1 and o_2 are a part of a sequence of objects that is not pointed from outside of the detected list structure. (3) The $\{nfo_c, pfo_c\}$-restricted sub-SMGs G_1 and G_2 of G rooted at o_1 and o_2 can be joined using the extended join algorithm into the sub-SMG $G_{1,2}$ to be nested below the join of o_1 and o_2. Let O_1, V_1 and O_2, V_2 be the sets of non-shared objects and values of G_1 and G_2, respectively, whose join gives rise to $G_{1,2}$. (4) The non-shared objects and values of G_1 and G_2 (other than o_1 and o_2 themselves) are reachable via o_1 or o_2, respectively, only. Moreover, the sets O_1 and O_2 contain heap objects only.

Merging Sequences of Objects into DLSs. Sequences of objects are merged into a single DLS *incrementally*, i.e., starting with the first two objects of the sequence, then merging the resulting new DLS with the third object in the sequence, and so on. Each of the *elementary merge operations* is performed as follows (see Fig. 4 for an illustration).

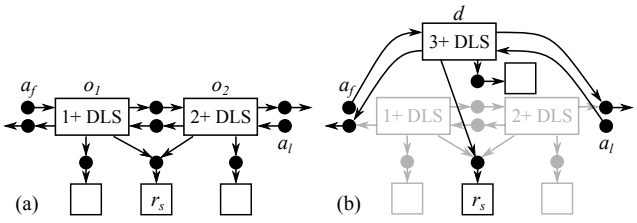

Fig. 4. The elementary merge operation: (a) input (b) output

Assume that G is the SMG of the current SPC (i.e., the initial SPC or the SPC obtained from the last merge) with the set of points-to edges P and the set of addresses A, o_1 is either the first object in the sequence or the DLS obtained from the previous elementary merge, o_2 is the next object of the sequence to be processed, and *hfo*$_c$, *nfo*$_c$, *pfo*$_c$ are the offsets from the candidate DLS entry defining the sequence to be merged. First, we merge o_1 and o_2 into a DLS d using *hfo*$_c$, *nfo*$_c$, and *pfo*$_c$ as its defining offsets (cf. Appendix C.8). The sub-SMG nested below d is created using the above mentioned extended join algorithm. Next, the DLS-linking pointers arriving to o_1 and o_2 are redirected to d. In particular, if there is $a_f \in A$ such that $P(a_f) = (o_1, \mathit{hfo}_c, \mathit{tg})$ for some $\mathit{tg} \in \{\texttt{fst}, \texttt{reg}\}$, then P is changed such that $P(a_f) = (d, \mathit{hfo}_c, \texttt{fst})$. Similarly, if there is $a_l \in A$ such that $P(a_l) = (o_2, \mathit{hfo}_c, \mathit{tg})$ for some $\mathit{tg} \in \{\texttt{lst}, \texttt{reg}\}$, then P is changed such that $P(a_l) = (d, \mathit{hfo}_c, \texttt{lst})$. Finally, each heap object and each value (apart from the null address and null object) that are not reachable from any static or stack object of the obtained SPC are removed from its SMG together with all the edges adjacent to them.

The Top-level Abstraction Algorithm. Assume we are given an SMG G, and a candidate DLS entry $(o_c, \mathit{hfo}_c, \mathit{nfo}_c, \mathit{pfo}_c)$ defining the longest mergeable sequence of objects $\sigma = o_1 o_2 \dots o_n$ in G of length $|\sigma| = n \geq 2$. We define the *cost* of merging a pair of objects o_1, o_2, denoted *cost*(o_1, o_2), as follows. First, *cost*$(o_1, o_2) = 0$ iff the $\{\mathit{nfo}_c, \mathit{pfo}_c\}$-restricted sub-SMGs G_1 and G_2 rooted at o_1, o_2 are equal (when ignoring the kinds of o_1 and o_2). This is indicated by the $\simeq$ status returned by the modified join algorithm applied on G_1, G_2. Further, *cost*$(o_1, o_2) = 1$ iff G_1 entails G_2, or vice versa, which is indicated by the status $\sqsupset$ or $\sqsubset$. Finally, *cost*$(o_1, o_2) = 2$ iff G_1 and G_2 are incomparable, which is indicated by status $\bowtie$. The cost of merging a sequence of objects $\sigma = o_1 o_2 \dots o_n$, denoted *cost*$(\sigma)$, is defined as the maximum of *cost*(o_1, o_2), *cost*(o_2, o_3), ..., *cost*(o_{n-1}, o_n).

Our abstraction is parameterized by associating each cost $c \in \{0, 1, 2\}$ with the *length threshold*, denoted *lenThr*(c), defining the minimum length of a sequence of mergeable objects allowed to be merged for the given cost. Intuitively, the higher is the cost, the bigger loss of precision is incurred by the merge, and hence a bigger number of objects to be merged is required to compensate the cost. In our experiments discussed in Section 5, we, in particular, found as optimal the setting *lenThr*$(0) =$ *lenThr*$(1) = 2$ and *lenThr*$(2) = 3$. Our tool, however, allows the user to tweak these values.

Based on the above introduced notions, the process of *abstracting an SPC* can now be described as follows. First, all candidate DLS entries are identified, and for each of them, the corresponding longest mergeable sequence is computed. Then each longest mergeable sequence σ for which $|\sigma| < \mathit{lenThr}(\mathit{cost}(\sigma))$ is discarded. Out of the remaining ones, we select those that have the lowest cost. From them, we then select those that have the longest length. Finally, out of them, one is selected arbitrarily. The selected sequence is merged, and then the entire abstraction process is repeated till there is a sequence that can be merged taking its length and cost into account.

3.4 Checking Equality and Inequality of Values

Checking equality of values in SMGs amounts to simply checking their identity. For checking inequality, we use an algorithm which is sound and efficient but incomplete. It is designed to succeed in most common cases, but in order not to harm its efficiency, we allow it to fail in some exceptional cases (e.g., when comparing addresses out of bounds of two distinct objects). The basic idea of the algorithm is as follows (cf. Appendix E): Let v_1 and v_2 be two distinct values of level 0 to be checked for inequality (other levels cannot be directly accessed by program statements). First, if the same value or object can be reached from v_1 and v_2 through 0+ DLSs only (using the next/prev fields when coming through the `fst`/`lst` target specifiers, respectively), then the inequality between v_1 and v_2 is not established. This is due to v_1 and v_2 may become the same value when the possibly empty 0+ DLSs are removed (or they may become addresses of the first and last node of the same 0+ DLS, and hence be equal in case the list contains a single node). Otherwise, v_1 and v_2 are claimed different if the final pair of values reached from them through 0+ DLSs represents different addresses due to pointing (1) to different valid objects (each with its own unique address) with offsets inside their bounds, (2) to the null object and a non-null object (with an in-bound offset), (3) to the same object with different offsets, or (4) to the same DLS with length at least 2 using different target specifiers. Otherwise, the inequality is not established.

3.5 A Brief Note on Symbolic Execution

The symbolic execution algorithm based on SMGs is similar to [1]. It uses the read reinterpretation operator for memory lookup (as well as type-casting) and the write reinterpretation operator for memory mutation. Whenever a DLS is about to be accessed (or its address with a non-head offset is about to be taken), a materialisation (as described in Section 2.3) is performed so that the actual program statements are always executed over concrete objects. If the minimum length of the DLS being materialised is zero, the computation is split into two branches—one for the empty segment and one for the non-empty segment. In the former case, the DLS is removed (as described in Section 2.3) while in the latter case, the minimum length of the DLS is incremented. When executing a conditional statement, the algorithm for checking (in)equality of values from Section 3.4 is used. If neither equality nor inequality are established, the execution is split into two branches, one of them assuming the compared values to be equal, the other assuming them not to be equal. This may again involve removing 0+ DLSs in one of the branches and incrementing their length in the other (cf. Appendix E).

A Note on Soundness of the Analysis. In the described analysis, program statements are always executed on concrete objects only, closely following the C semantics. The read reinterpretation is defined such that it cannot change the semantics of the input SMG, and the write reinterpretation can only over-approximate the semantics in the worst case. Likewise, our abstraction and join algorithms are allowed to only over-approximate the semantics—indeed, when joining a pair of nodes, the semantics of the resulting node is always generic enough to cover the semantics of both of the joined nodes (e.g., the join of a $2+$ DLS with a compatible region results in a $1+$ DLS, etc.). Moreover, the entailment check used to terminate the analysis is based on the join operator and consequently conservative. Hence, it is not difficult to see that the proposed analysis is sound (although a full proof of this fact would be rather technical).

4 Extensions of SMGs

In this section, we point out that the above introduced notion of SMGs can be easily extended in various directions, and we briefly discuss several such extensions (including further kinds of abstract objects), most of which are already implemented in our tool Predator.

Explicit Non-equivalence Relations. When several objects have the same concrete value stored in their fields, this is expressed by that the appropriate has-value edges lead from these objects to the same value node in the SMG. On the other hand, two different value nodes in an SMG do not necessarily represent different concrete values. To express that two abstract values represent distinct concrete values, SMGs can be extended with a symmetric, irreflexive relation over values, which we call an *explicit non-equivalence relation.*[11] Clearly, SMGs can be quite naturally extended by allowing more predicates on data, which is, however, beyond the scope of this paper (up to a small extension by tracking more concrete values than 0 that is mentioned below).

Singly-linked List Segments (SLSs). Above, we have presented all algorithms on SMGs describing doubly-linked lists only. Nevertheless, the algorithms work equally well with singly-linked lists represented by an additional kind of abstract objects, SLSs, that have no *pfo* offset, and their addresses are allowed to use the `fst` and `all` target specifiers only. The algorithm looking for DLS entry candidates then simply starts looking for SLS entry candidates whenever it does not discover the back-link.

0/1 Abstract Objects. In order to enable summarization of lists whose nodes can *optionally* point to some region or that point to nested lists whose length never reaches 2 or more, we introduce the so-called *0/1 abstract objects*. We distinguish three kinds of them with different numbers of neighbour pointers. The first of them represents 0/1 SLSs with one neighbour pointer, another represents 0/1 DLSs with two neighbour pointers[12]. These objects can be later joined with compatible SLSs or DLSs. The third kind has no neighbour pointer, and its address is assumed to be NULL when the region

[11] This is similar to the equality and non-equality constraints in separation logic, but only non-equality constraints are kept explicit.

[12] If a DLL consists of exactly one node, the value of its next pointer is equal to the value of its prev pointer. There is no point in distinguishing them, so we call them both neighbour pointers.

is not allocated. This kind is needed for optionally allocated regions referred from list nodes but never handled as lists themselves. The 0/1 abstract objects are created by the join algorithm when a region in one SMG cannot be matched with an object from the other SMG and none of the above described join mechanisms applies.

Offset Intervals and Address Alignment. The basic SMG notion labels points-to edges with scalar offsets within the target object. This labelling can be generalized to *intervals of offsets*. The intervals can be allowed to arise by joining objects with incoming pointers compatible up to their offset. This feature is useful, e.g., to handle lists arising in higher-level memory allocators discussed in the next section where each node points to itself with an offset depending on how much of the node has been used by sub-allocation. Offset intervals also naturally arise when the analysis is allowed to support *address alignment*, which is typically implemented by masking several lowest bits of pointers to zero, resulting in a pointer whose offset is in a certain interval wrt. the base address. Similarly, one can allow the *object size* to be given by an interval, which in turn allows one to abstract lists whose nodes are of a variable size.

Integral Constants and Intervals. The basic SMG notion allows one to express that two fields have the same value (by the corresponding has-value edges leading to the same value node) or that their values differ (using the above mentioned explicit non-equivalence relation). In order to improve the support of dealing with integers, SMGs can be extended by associating value nodes with concrete integral numbers. These can be respected by the join algorithm (at least up to some bound as done in the Predator tool discussed below), or they can be abstracted to intervals (also supported by Predator) or some other abstract numerical domains (which we plan for the future).

5 Implementation

We have implemented the above described algorithms (including the extensions) in a new version of our tool called Predator.[13] Predator is a GCC plug-in, which allows one to experiment with industrial source code without manually preprocessing it first. The verified program must, however, be closed in that it must allocate and initialize all the data structures used. Modular verification of code fragments is planned for the future. By default, Predator disallows calls to external functions in order to exclude any side effect that could potentially break memory safety. The only allowed external functions are those that Predator recognizes as built-in functions and properly models them wrt. proving memory safety. Besides `malloc` and `free`, the set of supported built-in functions includes certain memory manipulating functions defined in the C standard, such as `memset`, `memcpy`, or `memmove`. It also provides a few built-in functions that are specific to our verification approach, e.g., functions to dump SMGs or program traces to files. Predator uses the same style of error and warning messages as GCC itself, and hence it can be used with any IDE that can use GCC. It also supports error recovery to report multiple program errors during one run. For example, if a memory leak is detected, Predator only reports a warning, the unreachable part of SMG is removed, and the symbolic execution then continues.

[13] `http://www.fit.vutbr.cz/research/groups/verifit/tools/predator/`

Predator implements an inter-procedural analysis based on [12]. It does not support recursive programs yet, but it supports indirect calls, which is necessary for verification of programs with callbacks (e.g., Linux drivers). Regions for stack variables are created automatically as needed and destroyed as soon as they become dead according to a static live variables analysis[14], performed before running the symbolic execution. When working with initialized variables[15], we take advantage of our efficient representation of nullified blocks—we first create a has-value edge $o \xrightarrow{0,\texttt{char}[\mathit{size}(o)]} 0$ for each initialized variable represented by a region o, then we execute all explicit initializers, which themselves automatically trigger the write reinterpretation. The same approach is used for `calloc`-based heap allocation. Thanks to this, we do not need to initialize each structure member explicitly, which would not scale for complex structures.

The algorithms for abstraction and join implemented in Predator use some further optimizations of the basic algorithms described in Section 3. While objects in SMGs are type-free, Predator tracks their *estimated type* given by the type of the pointers through which objects are manipulated. The estimated type is used during abstraction to postpone merging a pair of objects with incompatible types. Note, however, that this is really a heuristic only—we have a case study that constructs list nodes using solely `void` pointers, and it can still be successfully verified by Predator. Another heuristic is that certain features of the join algorithm (e.g., insertion of a non-empty DLS or introduction of an 0/1 abstract object) are disabled when joining SMGs while enabled when merging nodes during abstraction. Predator tracks integral values precisely up to a certain bound (± 10 by default) and once the bound is reached, the values are abstracted out. Predator also supports intervals aligned to a power of two as well as tracking of simple dependences between intervals, such as a shift by a constant and a multiplication by -1. All these features are optional and can be easily disabled.

Predator iteratively computes sets of SMGs for each basic block entry of the control-flow graph of the given program, covering all program configurations reachable at these program locations. Termination of the analysis is aided by the abstraction and join algorithms described above. Since the join algorithm is expensive, it is used at loop boundaries only. When updating states of other basic block entries, we compare the SMGs for equality[16] only, which makes the comparison way faster, especially in case a pair of SMGs cannot be joined. Similarly, the abstraction is by default used only at loop boundaries in order not to introduce abstract objects where not necessary (reducing the space for false positives that can arise due to breaking assumptions sometimes used by programmers for code inside loops as witnessed by some of our case studies).

Predator is able to discover or prove absence of various kinds of *memory safety errors*, including various forms of illegal dereferences (null dereferences, dereferences of freed or unallocated memory, out-of-bound dereferences), illegal free operations (double free operations, freeing non-heap objects), as well as memory leakage. Memory leakage checks are optimized by collecting sets of lost addresses for each operation that can introduce a memory leak (write reinterpretation, `free`, etc.), followed by check-

[14] If a program variable is referenced by a pointer, its destruction needs to be postponed.

[15] According to the C99 standard, all static variables are initialized, either explicitly or implicitly.

[16] The join algorithm can be easily restricted to check for equality only—if any action that would imply inequality of the input SMGs is about to be taken, the join operation fails immediately.

ing whether reachability of allocated objects from program variables depend on the collected addresses. Moreover, Predator also uses the fact that SMGs allow for easy checking whether a given pair of memory areas overlap. Indeed, if both of them are inside of two distinct valid regions, they have no overlaps, and if both of them are inside the same region, one can simply check their offset ranges for intersection. Such checks are used for reporting invalid uses of `memcpy` or the C-language assignment, which expose undefined behavior if the destination and source memory areas (partially) overlap with each other.

6 Experiments

The new version of Predator based on the above proposed method was successfully tested on a number of case studies. Among them there are more than 256 case studies (freely available with Predator[17]) illustrating various programming constructs typically used when dealing with linked lists. These case studies include various advanced kinds of lists used in the Linux kernel and their typical manipulation, typical error patterns that appear in code operating with Linux lists, various sorting algorithms (insert sort, bubble sort, merge sort), etc. These case studies have up to 300 lines of code, but they consist almost entirely of complex memory manipulation (unlike larger programs whose big portions are often ignored by tools verifying memory safety). Next, we successfully tested Predator on the driver code snippets distributed with SLAyer [2] as well as on the `cdrom` driver originally checked by Space Invader [16]. As discussed below, in some of these examples, we identified errors not found by the other tools due to their more abstract (not byte-precise) treatment of memory.

Further, we also considered two real-life low-level programs (which, to the best of our knowledge, have not yet been targeted by fully automated formal verification tools): a memory allocator from the Netscape portable runtime (NSPR) and a module taken from the `lvm2` logical volume manager. The NSPR allocator allocates memory from the operating system in blocks called *arenas*, grouped into singly-linked lists called *arena pools*, which can in turn be grouped into lists of arena pools (giving lists of lists of arenas). User requests are then satisfied by sub-allocation within a suitable arena of a given arena pool. We have considered a fixed size of the arenas and checked safety of repeated allocation and deallocation of blocks of aligned size randomly chosen up to the arena size from arena pools as well as lists of arena pools. For this purpose, a support for offset intervals as described above was needed. The intervals arise from abstracting lists whose nodes (arenas) point with different offsets to themselves (one byte behind the last sub-allocated block within the arena) and from address alignment, which the NSPR-based allocator is also responsible for. Our approach allowed us to verify that pointers leading from each arena to its so-far free part never point beyond the arena and that arena headers never overlap with their data areas, which are the original assertions checked by NSPR arena pools at run-time (if compiled with the debug support). Our `lvm2`-based case studies then exercise various functions of the module implementing the volume metadata cache. As in the case of NSPR arenas, we

[17] `https://github.com/kdudka/predator/tree/master/tests`

Table 1. Selected experimental results showing either the verification time or one of the following outcomes: FP = false positive, FN = false negative, T = time out (900 s), x = parsing problems

Test Origin	Test	Invader	SLAyer	Predator 2011-10	Predator 2013-02
SLAyer	`append.c`	<0.01 s	10.47 s	<0.01 s	<0.01 s
	`cromdata_add_remove_fs.c`	<0.01 s	FN	<0.01 s	<0.01 s
	`create_kernel.c`	T	FN	<0.01 s	<0.01 s
	`cromdata_add_remove.c`	T	FN	<0.01 s	<0.01 s
	`reverse_seg_cyclic.c`	FP	0.68 s	<0.01 s	<0.01 s
	`is_on_list_via_devext.c`	T	34.43 s	0.20 s	0.02 s
	`callback_remove_entry_list.c`	T	71.46 s	0.14 s	0.10 s
Invader	`cdrom.c`	FN	x	2.44 s	0.66 s
Predator	`five-level-sll-destroyed-top-down.c`	FP	x	FP	0.05 s
	`linux-dll-of-linux-dll.c`	T	x	0.41 s	0.05 s
	`merge-sort.c`	FP	x	1.08 s	0.21 s
	`list-of-arena-pools-with-alignment.c`	FP	x	FP	0.50 s
	`lvmcache_add_orphan_vginfo.c`	x	x	FP	1.07 s
	`five-level-sll-destroyed-bottom-up.c`	FP	x	FP	1.14 s

use the original (unsimplified) code of the module, but (for now) we use a simplified test harness where the `lvm2` implementation of hash tables is replaced by the `lvm2` implementation of doubly-linked lists.

We have compared the capabilities and performance of Invader, SLAyer, and Predator on the above case studies on an Intel® Core™ i7-3770K machine. The memory consumption was below 128 MB in all cases. As we can see in Table 1, Predator successfully verified even the test-cases that were causing problems to Invader or SLAyer. We have also revealed issues of memory safety violation in the examples distributed with Invader and SLAyer because Invader did not check memory manipulation via array subscripts and SLAyer did not check size of the blocks allocated on the heap.[18] All the tools were run in their default configurations. Better results can sometimes be obtained for particular case studies by tweaking certain configuration options (abstraction threshold, call cache size, etc.). However, while such changes may improve the performance in some case studies, they may harm it in others, trigger false positives, or even prevent the analysis from termination.

We have also compared the new version of Predator with its older version that participated in the 1st International Competition on Software Verification (SV-COMP'12). The old Predator produced false positives on many of the more advanced case studies, including NSPR arenas and lvm2, and it was also slower. For example, the merge-sort case study, presented as the most expensive in [7] (Predator 2011-02), now runs approximately $25\times$ faster on the same machine ($5\times$ due to the algorithms presented above and $5\times$ due to an improved live variable analysis). The new Predator participated in the 2nd International Competition on Software Verification (SV-COMP'13) [4], where it won three categories. Moreover, the fact that Predator did not have any false negative over the whole SV-COMP'13 benchmark confirms the soundness of our analysis algorithm.

[18] We used the latest publicly available version of SLAyer from [2]. The version from [3] was not available, but [3] targets mainly checking of spuriousness of counterexamples. SLAyer is, however, still in an active development, thus it is possible that its newer versions may provide better results.

7 Conclusion and Future Work

We have presented a new approach to fully automated formal verification of list manipulating programs capable of handling various features of low-level memory manipulation. We have experimentally validated the approach on a number of case studies showing its efficiency and capability of handling program behaviour that is beyond what current fully automated shape analysis tools can handle. In the future, a number of extensions of our approach are possible. We are planning a support of (low-level) tree structures, a better support of integer data, a support of arrays and hash tables, as well as a support for modular verification in order to remove the burden of having to write environments for the code to be verified.

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 *Proc. CAV'07*, *LNCS* 4590, Springer, 2007.
2. J. Berdine, B. Cook, and S. Ishtiaq. Memory Safety for Systems-level Code. In *Proc. of CAV'11*, *LNCS* 6806, Springer, 2011.
3. J. Berdine, A. Cox, S. Ishtiaq, and C.M. Wintersteiger. Diagnosing Abstraction Failure for Separation Logic-Based Analyses. In *Proc. of CAV'12*, *LNCS* 7358, Springer, 2012.
4. D. Beyer. Second Competition on Software Verification (Summary of SV-COMP 2013). To appear in *Proc. of TACAS'13*, *LNCS 7795*, Springer, 2013.
5. C. Calcagno, D. Distefano, P.W. O'Hearn, H. Yang. Beyond Reachability: Shape Abstraction in the Presence of Pointer Arithmetic. In *Proc. of SAS'06*, *LNCS* 4134, Springer, 2006.
6. Bor-Yuh Evan Chang, Xavier Rival, and George C. Necula. Shape analysis with structural invariant checkers. In *International Static Analysis Symposium (SAS)*, 2007.
7. K. Dudka, P. Peringer, and T. Vojnar. Predator: A Practical Tool for Checking Manipulation of Dynamic Data Structures Using Separation Logic. In *Proc. of CAV'11*, *LNCS* 6806, 2011.
8. P. Habermehl, L. Holík, A. Rogalewicz, J. Šimáček, and T. Vojnar. Forest Automata for Verification of Heap Manipulation. In *Proc. of CAV'11*, LNCS 6806, Springer, 2011
9. J. Kreiker, H. Seidl, and V. Vojdani: Shape Analysis of Low-Level C with Overlapping Structures. In *Proc. of VMCAI'10*, *LNCS 5944*, Springer, 2010.
10. V. Laviron, B.-Y.E. Chang, and X. Rival. Separating Shape Graphs. In *Proc. of ESOP'10*, *LNCS* 6012, Springer, 2010.
11. M. Marron, M. Hermenegildo, D. Kapur, D. Stefanovic. Efficient Context-Sensitive Shape Analysis with Graph Based Heap Models. In *Proc. of CC'08*, *LNCS* 4959, Springer, 2008.
12. T. Reps, S. Horwitz, and M. Sagiv. Precise Interprocedural Dataflow Analysis via Graph Reachability. In *Proc. of POPL'95*, ACM Press, 1995.
13. M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. In *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 24(3), ACM, 2002.
14. H. Tuch. Formal Verification of C Systems Code. In *Journal of Automated Reasoning*, 42(2–4), Springer, 2009.
15. H. Yang, O. Lee, C. Calcagno, D. Distefano, and P.W. O'Hearn. On Scalable Shape Analysis. Technical report RR-07-10, Queen Mary, University of London, 2007.
16. H. Yang, O. Lee, J. Berdine, C. Calcagno, B. Cook, D. Distefano, and P.W. O'Hearn. Scalable Shape Analysis for Systems Code. In *Proc. of CAV'08*, *LNCS* 5123, Springer, 2008.

A The Notion of SMGs

In this appendix, we illustrate how SMGs represent various data structures common in practice. The upper part of Fig. 5 shows a Linux-style cyclic DLL of cyclic DLLs. All nodes of all the nested DLLs point to a shared memory region. The lower part of the figure shows an SMG representing this structure. Note that the top-level DLS as well as the shared region are on level 0 whereas the nested DLSs are on level 1.

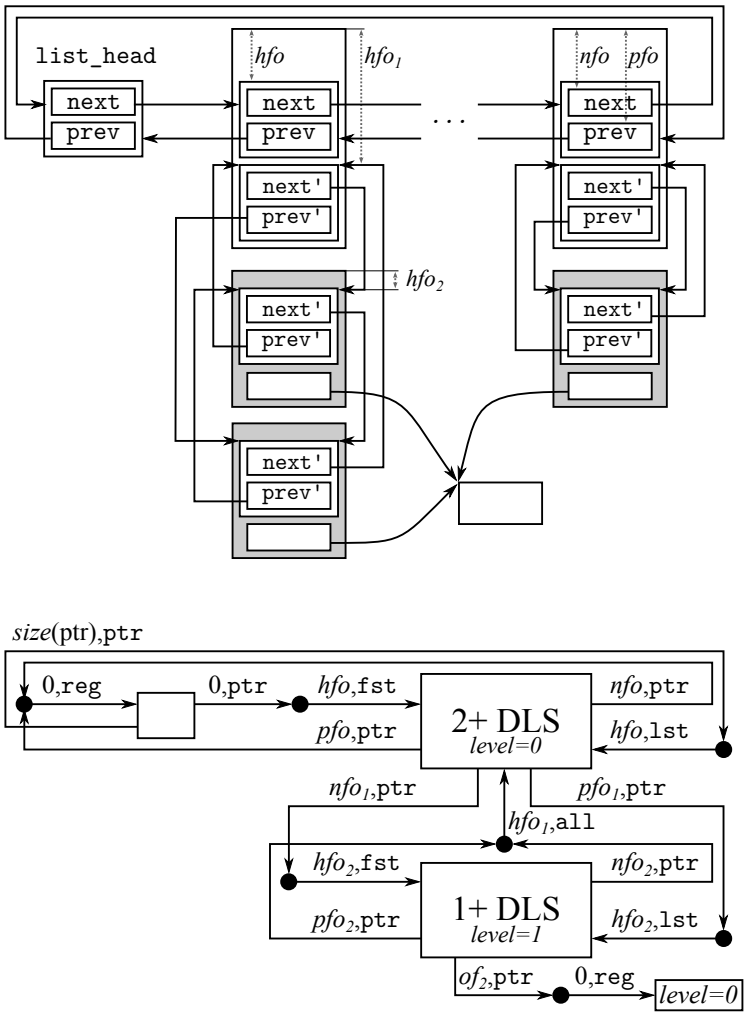

Fig. 5. A cyclic Linux-style DLL of DLLs with a shared data element (top) and its SMG (bottom)

The upper part of Fig. 6 shows another variant of Linux-style DLLs which is optimised for use in hash tables. The lower part of the figure shows an SMG representing this kind of lists. For lists used in hash tables, the size of list headers determines the amount of memory allocated by an empty hash table. That is why the lists presented in Fig. 6 have reduced headers for the price of having forward and backward links of different types. In particular, forward links are pointers to structures whereas backward links are pointers to pointers to structures. This asymmetry may cause problems to analyzers that use a selector-based description of list segments, but it is not a problem for us due to our representation is purely offset based.[19]

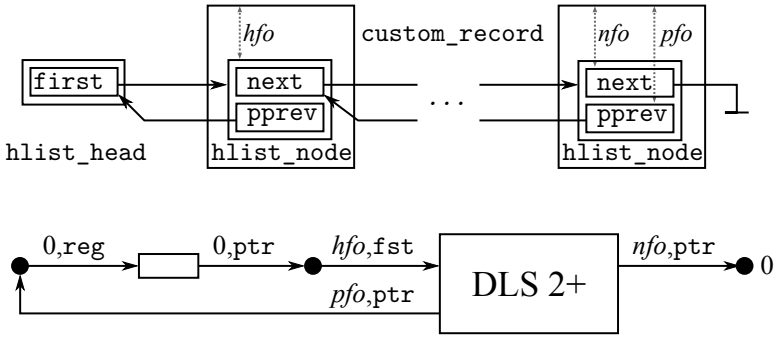

Fig. 6. A Linux-style list used in hash tables (top) and its SMG (bottom)

[19] A need to use a special kind of list segments would arise in SMGs if the head and next offsets were different, but that is unlikely to happen in this special case since it would prevent the list head from having the size of a single pointer only.

B Data Reinterpretation of Nullified Blocks

In this appendix, we present a detailed description of the algorithms for read and write reinterpretation of nullified blocks, which were briefly introduced in Section 3.1. Given an SMG $G = (O, V, \Lambda, H, P)$, we define $H_{ov}(o, of, t)$ as the set of all has-value edges leading from o whose fields overlap with the field (of, t), i.e.:

$$H_{ov}(o, of, t) = \{(o \xrightarrow{of';t'} v) \in H \mid I(of, t) \cap I(of', t') \neq \emptyset\}.$$

Further, we define $H_{zr}(o, of, t)$ as the subset of $H_{ov}(o, of, t)$ containing all its edges leading to 0, i.e.:

$$H_{zr}(o, of, t) = \{(o \xrightarrow{of';t'} 0) \in H_{ov}(o, of, t)\}.$$

B.1 Read Reinterpretation of Nullified Blocks

Algorithm 1 gives the algorithm of read reinterpretation instantiated for dealing with nullified blocks of memory as precisely as possible.

B.2 Write Reinterpretation of Nullified Blocks

Algorithm 2 gives the algorithm of write reinterpretation instantiated for dealing with nullified blocks of memory as precisely as possible. An illustration of how the algorithm works can be found in Fig. 7.

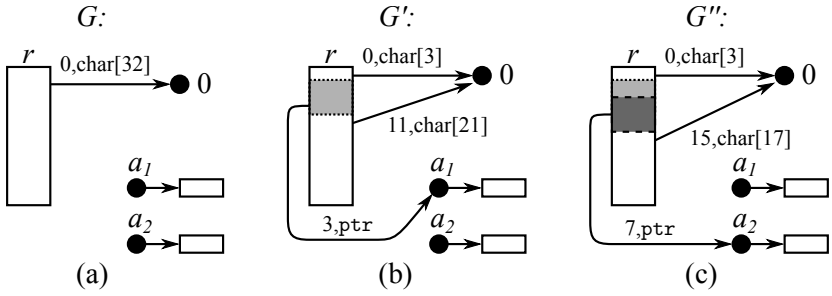

Fig. 7. An illustration of write reinterpretation: (a) an initial SMG G, (b) SMG G' obtained by $writeValue(G, r, 3, ptr, a_1)$, (c) SMG G'' obtained by $writeValue(G', r, 7, ptr, a_2)$. Note that G'' contains an undefined value in a field of size of 4 bytes at offset 3.

Algorithm 1 $readValue(G, o, of, t)$

Input:
- An SMG $G = (O, V, \Lambda, H, P)$.
- An object $o \in O$.
- A field (of, t) within o, i.e., $of + size(t) \leq size(o)$.

Output:
- A tuple (G', v) that is the result of read reinterpretation of G wrt. the object o and the field (of, t) such that fields representing nullified memory are read as precisely as the notion of SMGs allows (i.e., the operator recognises that the field to be read is nullified iff the input SMG guarantees that each byte of the field is indeed zero).

Method:
1. Let $v := H(o, of, t)$.
2. If $v \neq \bot$, return (G, v).
3. If the field to be read is covered by nullified blocks, i.e., if $\forall of \leq i < of + size(t)\ \exists e \in H_{zr}(o, of, t) :\ i \in I(e)$, let $v := 0$. Otherwise extend V by a fresh value node v.
4. Extend H by the has-value edge $o \xrightarrow{of,t} v$ and return (G, v) based on the obtained SMG G.

Algorithm 2 $writeValue(G, o, of, t, v)$

Input:
- An SMG $G = (O, V, \Lambda, H, P)$.
- An object $o \in O$.
- A field (of, t) within o, i.e., $of + size(t) \leq size(o)$.
- A value v such that $v \notin O$ (needed so that v can be safely added into V).

Output:
- An SMG G' that is the result of write reinterpretation of G wrt. the object o, the field (of, t), and the value v such that as much information on nullified memory as is allowed by the notion of SMGs is preserved (i.e., each byte that is nullified in the input SMG will stay nullified in the output SMG unless it is overwritten by a possibly non-zero value).

Method:
1. If $H(o, of, t) = v$, return G.
2. Let $V := V \cup \{v\}$.
3. Remove from H all edges leading from o to non-zero values whose fields overlap with the given field, i.e., the edges in $H_{ov}(o, of, t) \setminus H_{zr}(o, of, t)$.
4. If $v \neq 0$, then for each edge $(e_z : o \xrightarrow{of_z, t_z} 0) \in H_{zr}(o, of, t)$ do:
   (a) Remove the edge e_z from H.
   (b) Let $of' := of + size(t)$ and $of'_z := of_z + size(t_z)$.
   (c) If $of_z < of$, then extend H by the edge $o \xrightarrow{of_z, \texttt{char}[of - of_z]} 0$.
   (d) If $of' < of'_z$, then extend H by the edge $o \xrightarrow{of', \texttt{char}[of'_z - of']} 0$.
5. Extend H by the has-value edge $o \xrightarrow{of,t} v$ and return the obtained SMG.

C The Join Algorithms

This appendix provides a detailed description of the join algorithms introduced in Section 3.2. We first describe the $joinSubSMGs$ function, which implements the core functionality on top of which both joining garbage-free SPCs (to reduce the number of SPCs obtained from different paths through the program) as well as merging a pair of neighbouring objects of a doubly-linked list into a single DLS within abstraction are built. Subsequently, we describe the functions $joinValues$, $joinTargetObjects$, and $insertLeftDlsAndJoin$ / $insertRightDlsAndJoin$ on which $joinSubSMGs$ is based. In fact, $joinSubSMGs$ calls $joinValues$ on pairs of corresponding values that appear below the roots of the sub-SMGs to be joined, $joinValues$ then calls $joinTargetObjects$ on pairs of objects that are the target of value nodes representing addresses, and the $joinTargetObjects$ algorithm recursively calls $joinSubSMGs$ to join the sub-SMGs of the objects to be joined. The $insertLeft(Right)DlsAndJoin$ functions are called from $joinValues$ when the given pair of addresses cannot be joined since their target objects are incompatible, and an attempt to save the join from failing is done by trying to compensate a DLS missing in one of the SMGs by inserting it with the minimum length being 0 (which is possible since a 0+ DLS is a possibly empty list segment). Finally, we describe the $joinSPCs$ and $mergeSubSMGs$ functions implemented on top of the generic $joinSubSMGs$ function. The $joinSPCs$ function joins garbage-free SPCs into a single SPC that semantically covers both. The $mergeSubSMGs$ function merges a pair of objects during abstraction into a single DLS while the non-shared part of the sub-SMGs rooted at them is joined as the nested data structures of the resulting DLS.

Fig. 8 illustrates how pairs of objects are joined and how a DLS from one SMG can be paired with an empty list from the other SMG. Both the mechanisms are described in detail further in this appendix.

As mentioned in Section 3.2, the join algorithm computes on the fly the so-called *join status* which compares the semantics of the SMGs being joined (with the semantics being either equal, in an entailment relation, or incomparable). For the purpose of maintaining the join status, the table shown on the right defines the function $updateJoinStatus : \mathbb{J} \times \mathbb{J} \longrightarrow \mathbb{J}$ that combines the current join status s_1 obtained from joining the so-far explored parts of the SMGs being joined with a status $s_2 \in \mathbb{J}$ comparing the semantics of the objects/values being currently joined. Note that the function is monotone in that once the status, which is initially $\simeq$, becomes $\sqsupset$ or $\sqsubset$, it can never get back to $\simeq$, and once the status becomes $\bowtie$, it cannot change any more.

s_1 \ s_2	$\simeq$	$\sqsupset$	$\sqsubset$	$\bowtie$
$\simeq$	$\simeq$	$\sqsupset$	$\sqsubset$	$\bowtie$
$\sqsupset$	$\sqsupset$	$\sqsupset$	$\bowtie$	$\bowtie$
$\sqsubset$	$\sqsubset$	$\bowtie$	$\sqsubset$	$\bowtie$
$\bowtie$	$\bowtie$	$\bowtie$	$\bowtie$	$\bowtie$

In case of the $joinSPCs$ function, $joinSubSMGs$ needs to be called multiple times for a single pair of SPCs (starting from different program variables), and it is necessary to keep certain state information between the calls. Besides the join status mentioned above, the algorithm maintains a mapping of values and objects between the source SMGs and the destination SMG. This is needed in order to identify potentially conflicting mappings arising when starting the join from different program variables as well as to identify parts of SMGs that have already been processed. The mapping is encoded as a pair of partial injective functions $m_1 : (O_1 \rightharpoonup O) \cup (V_1 \rightharpoonup V)$ and $m_2 : (O_2 \rightharpoonup O) \cup (V_2 \rightharpoonup V)$. Additionally, we assume that the $\#$ object and the

0 address, which have their pre-defined unique roles in all SMGs, never appear in the ranges of m_1, m_2. In case of the $mergeSubSMGs$ function, the mapping of objects and values is used to obtain the sets of nodes recognized as nested data structures.

In the following, we write $kind_1, size_1, level_1, len_1, valid_1, nfo_1, pfo_1$, and hfo_1 to denote *kind*, *size*, *level*, *len*, *valid*, *nfo*, *pfo*, and *hfo* from Λ_1. Likewise for Λ_2. We further define *len*′ as a wrapper function of *len* such that $len'(o) = len(o)$ if $kind(o) = \texttt{dls}$, and $len'(o) = 1$ if $kind(o) = \texttt{reg}$.

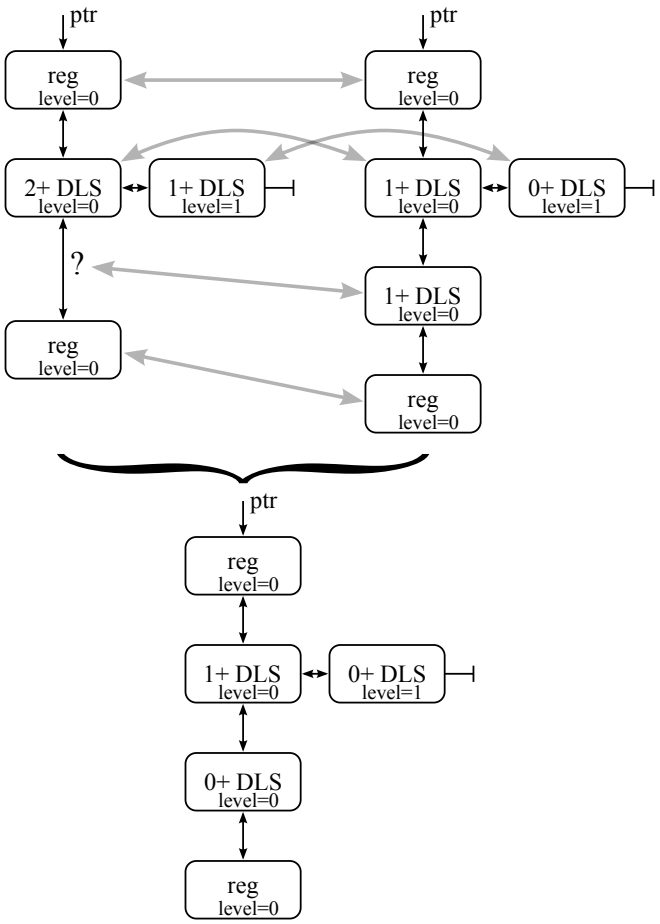

Fig. 8. An illustration of the basic principle of the join algorithm. In the figure, a simplified notation for describing SMGs is used from which value nodes have been left out. The pair of input SMGs is at the top. The gray arrows show the pairs of objects joined by the algorithm during the simultaneous traversal of the input SMGs. The resulting SMG is at the bottom.

C.1 Join Reinterpretation

The read and write reinterpretations described in Section 3.1 operate on a single object of a single SMG. However, when joining a pair of SMGs, we need to compare pairs of their objects, figure out what they have semantically in common, and modify their sets of fields such that they become the same (even if for the price of loosing some information), allowing one to subsequently attempt to join their corresponding sub-SMGs. For that purpose, we introduce *join reinterpretation*.

A join reinterpretation operator inputs a pair of SMGs G_1 and G_2, whose sets of objects are O_1 and O_2, respectively, and a pair of objects $o_1 \in O_1$ and $o_2 \in O_2$ such that $\mathit{size}_1(o_1) = \mathit{size}_2(o_2)$. It returns a triple (s, G'_1, G'_2) where $s \in \mathbb{J}$ is a join status, and G'_1, G'_2 are two SMGs with sets of has-value edges H'_1, H'_2, respectively, such that: (1) The sets of fields of o_1 and o_2 are the same, i.e., $\forall \mathit{of} \in \mathbb{N}\, \forall t \in \mathbb{T} :\ H'_1(o_1, \mathit{of}, t) \neq \bot \Leftrightarrow H'_2(o_2, \mathit{of}, t) \neq \bot$. (2) The status s and the semantics of G'_1 and G'_2 are defined according to the table shown on the right. Intuitively, the status $\sqsupset$ means that some aspects of G_1 are less restrictive than those of G_2, implying a need to lift these restrictions in G_2 to keep chance for its successful join with G_1, which increases the semantics of G_2 while that of G_1 stays the same. Likewise for the other symbols of $\mathbb{J}$.

s	semantics of G'_1	semantics of G'_2
$\simeq$	$\mathit{MI}(G_1) = \mathit{MI}(G'_1)$	$\mathit{MI}(G_2) = \mathit{MI}(G'_2)$
$\sqsupset$	$\mathit{MI}(G_1) = \mathit{MI}(G'_1)$	$\mathit{MI}(G_2) \subset \mathit{MI}(G'_2)$
$\sqsubset$	$\mathit{MI}(G_1) \subset \mathit{MI}(G'_1)$	$\mathit{MI}(G_2) = \mathit{MI}(G'_2)$
$\bowtie$	$\mathit{MI}(G_1) \subset \mathit{MI}(G'_1)$	$\mathit{MI}(G_2) \subset \mathit{MI}(G'_2)$

For the particular case of dealing with nullified memory, we implement join reinterpretation as follows (cf. Algorithm 3). First, nullified fields are shortened, split, and/or composed in each of the objects with the aim of obtaining the smallest possible number of nullified fields such that either (a) each byte of such fields is nullified in both original SMGs, or (b) the field is nullified in one SMG, and in the other, it contains a non-null address. The former is motivated by preserving as much information about nullified memory as possible when joining two objects. The latter is motivated by the fact that a null pointer may be interpreted as a special case of a null-terminated 0+ DLS and hence possibly joined with an address in the other SMG if its target is a DLS. Finally, whenever a field (of, t) remains defined in o_1 but not in o_2 after the described transformations, i.e., if $H_1(o_1, \mathit{of}, t) \neq \bot$ and $H_2(o_2, \mathit{of}, t) = \bot$, H_2 is extended such that $H_2(o_2, \mathit{of}, t) = v'$ for some fresh v' (and likewise the other way around).

Algorithm 3 $joinFields(G_1, G_2, o_1, o_2)$

Input:

- SMGs $G_1 = (O_1, V_1, \Lambda_1, H_1, P_1)$ and $G_2 = (O_2, V_2, \Lambda_2, H_2, P_2)$ with sets of addresses A_1 and A_2, respectively.
- Objects $o_1 \in O_1$ and $o_2 \in O_2$ such that $\mathit{size}_1(o_1) = \mathit{size}_2(o_2)$.

Output:

- A tuple (s', G_1', G_2') consisting of a join status and two SMGs that is the result of join reinterpretation of G_1 and G_2 wrt. o_1 and o_2 which reorganizes the nullified fields of o_1 and o_2 with the aim of obtaining the smallest possible number of nullified fields such that either (a) each byte of such fields is nullified in both G_1 and G_2, or (b) the field is nullified in one of them, and in the other, it contains a non-null address.

Method:

1. Let $H_1' := H_1,\ H_2' := H_2$.
2. Process the set $H_{1,0} = \{o_1 \xrightarrow{of,t} 0 \in H_1'\}$ of edges leading from o_1 to 0 in G_1 as follows:
   (a) Remove the edges that are in $H_{1,0}$ from H_1'.
   (b) Extend H_1' by the smallest set of edges $H_{1,0}'$ in which for each $0 \leq i < \mathit{size}_1(o_1)$ there is an edge $o_1 \xrightarrow{of',t'} 0 \in H_{1,0}'$ such that $i \in I(of', t')$ where $t' = \texttt{char}[n]$ for some $n > 0$ iff $\exists (o_1 \xrightarrow{of_1,t_1} 0) \in H_{1,0}\ \exists (o_2 \xrightarrow{of_2,t_2} 0) \in H_2' :\ i \in I(of_1, t_1) \cap I(of_2, t_2)$.
   (c) For each adress $a_2 \in A_2 \setminus \{0\}$ and each edge $(o_2 \xrightarrow{of,\texttt{ptr}} a_2) \in H_2'$ for which there is no $a_1 \in A_1 \setminus \{0\}$ such that $(o_1 \xrightarrow{of,\texttt{ptr}} a_1) \in H_1'$, but $I(of, \texttt{ptr}) \subseteq \bigcup_{e \in H_{1,0}} I(e)$, extend H_1' by the edge $o_1 \xrightarrow{of,\texttt{ptr}} 0$.

   Then do the same for o_2 with swapped sets of edges H_1' and H_2', using A_1 instead of A_2, and $H_{2,0}$ and $H_{2,0}'$ instead of $H_{1,0}$ and $H_{1,0}'$, respectively.
3. Let $s := \simeq$.
4. For each $0 \leq i < \mathit{size}_1(o_1)$:
   - If $\exists (e :\ o_1 \xrightarrow{of,t} 0) \in H_1$ such that $i \in I(e)$ and $\forall (e' : o_1 \xrightarrow{of',t'} 0) \in H_1' : i \notin I(e')$, let $s := \mathit{updateJoinStatus}(s, \sqsubset)$.
   - If $\exists (e :\ o_2 \xrightarrow{of,t} 0) \in H_2$ such that $i \in I(e)$ and $\forall (e' : o_2 \xrightarrow{of',t'} 0) \in H_2' : i \notin I(e')$, let $s := \mathit{updateJoinStatus}(s, \sqsupset)$.
5. For all fields (of, t) such that $H_1'(o_1, of, t) \neq \bot\ \wedge\ H_2'(o_2, of, t) = \bot$, extend H_2' such that $H_2'(o_2, of, t) = v$ for some fresh v added into V_2. Proceed likewise for non-nullified fields of o_2 not defined in o_1.
6. Return (s, G_1, G_2) where $G_1 = (O_1, V_1, \Lambda_1, H_1', P_1)$ and $G_2 = (O_2, V_2, \Lambda_2, H_2', P_2)$.

C.2 Join of Sub-SMGs

The $joinSubSMGs$ function (cf. Alg. 4) is responsible for joining a pair of sub-SMGs rooted at a given pair of objects and for constructing the resulting sub-SMG within the given destination SMG. The function inputs a triple of SMGs G_1, G_2, G (two source SMGs and one destination SMG) and a triple of equally sized objects o_1, o_2, o from the SMGs G_1, G_2, G, respectively. If the $joinSubSMGs$ function fails in joining the given sub-SMGs, it returns $\bot$. Otherwise, it returns a triple of SMGs G'_1, G'_2, G' such that:

- $MI(G_1) \subseteq MI(G'_1)$ and $MI(G_2) \subseteq MI(G'_2)$ where G_1 and G_2 can differ from G'_1 and G'_2, respectively, due to an application of join reinterpretation on some of the pairs of objects being joined only.
- The sub-SMGs G''_1 and G''_2 of G'_1 and G'_2 rooted at o_1 and o_2, respectively, are joined into the sub-SMG G'' of G' rooted at o, i.e., it is required that $MI(G''_1) \subseteq MI(G'') \supseteq MI(G''_2)$.
- The sub-SMG $G' \setminus G''$ is exactly the sub-SMG of G that consists of objects and values that are not removed in Step 11 of the $joinTargetObjects$ function due to using the principle of delayed join of sub-SMGs described in Appendix C.6.

The $joinSubSMGs$ function first applies the join reinterpretation operator (denoted $joinFields$) on G_1, G_2 and o_1, o_2, which ensures that the sets of fields of o_1 and o_2 are identical. The function then iterates over the set of fields of these objects, and for each field (of, t) does the following:

- finds the pair of values v_1 and v_2 which the has-value edges of o_1 and o_2 labelled by (of, t) lead to,
- calls the $joinValues$ function for v_1 and v_2, which is responsible for recursively joining the remaining parts of the sub-SMGs rooted at them, and
- extends the set of edges of G by $o \xrightarrow{of,t} v$ where v is the value returned by the $joinValues$ function.

Adjusting the Nesting Level Difference. In Section 3.2, it is said that the levels of the objects being joined can differ since the objects may sometimes appear below a DLS and sometimes below a region (and while an object that appears below a DLS may be considered nested—provided that each node of the segment has a separate copy of such an object—there is no notion of nesting below regions—since for regions which represent concrete objects there is no need to distinguish private and shared sub-SMGs). The functionality of $joinSubSMGs$ therefore includes tracking of the difference in levels (denoted l_{diff}) at which objects and values to be joined within some sub-SMG can appear. When objects o_1 and o_2 are being joined, the difference is computed as follows: If o_1 is a DLS and o_2 is a region, the current value of l_{diff} is increased by one. Symmetrically, if o_1 is a region and o_2 is a DLS, the value of l_{diff} is decreased by one. The new difference is then used when joining the values of the fields of o_1 and o_2 (apart from the next and prev fields of course).

Algorithm 4 $joinSubSMGs(s, G_1, G_2, G, m_1, m_2, o_1, o_2, o, l_{diff})$

Input:
- Initial join status $s \in \mathbb{J}$.
- SMGs $G_1 = (O_1, V_1, \Lambda_1, H_1, P_1)$, $G_2 = (O_2, V_2, \Lambda_2, H_2, P_2)$, and $G = (O, V, \Lambda, H, P)$.
- Injective partial mappings of nodes m_1, m_2 as defined in Appendix C.
- Objects $o_1 \in O_1$, $o_2 \in O_2$, $o \in O$.
- Nesting level difference $l_{diff} \in \mathbb{Z}$.

Output:
- $\bot$ in case the sub-SMGs of G_1 and G_2 rooted at o_1 and o_2 cannot be joined.
- Otherwise, a tuple $(s', G_1', G_2', G', m_1', m_2')$ where:
  - $s' \in \mathbb{J}$ is the resulting join status.
  - G_1', G_2', G' are SMGs as defined in Appendix C.2.
  - m_1', m_2' are the resulting injective partial mappings of nodes.

Method:
1. Let $res := joinFields(G_1, G_2, o_1, o_2)$. If $res = \bot$, return $\bot$. Otherwise let $(s', G_1, G_2) := res$ and $s := updateJoinStatus(s, s')$.
2. Collect the set F of all pairs (of, t) occurring in has-value edges leading from o_1 or o_2.
3. For each field $(of, t) \in F$ do:
   - Let $v_1 = H_1(o_1, of, t)$, $v_2 = H_2(o_2, of, t)$, and $l'_{diff} := l_{diff}$.
   - If $kind_1(o_1) =$ `dls` and (of, t) is not next/prev field of o_1, let $l'_{diff} := l'_{diff} + 1$.
   - If $kind_2(o_2) =$ `dls` and (of, t) is not next/prev field of o_2, let $l'_{diff} := l'_{diff} - 1$.
   - Let $res := joinValues(s, G_1, G_2, G, m_1, m_2, v_1, v_2, l'_{diff})$. If $res = \bot$, return $\bot$. Otherwise let $(s, G_1, G_2, G, m_1, m_2, v) := res$.
   - Introduce a new has-value edge $o \xrightarrow{of,t} v$ in H.
4. Return $(s, G_1, G_2, G, m_1, m_2)$.

C.3 Join of Values

The $joinValues$ function (cf. Alg. 5) joins a pair of sub-SMGs rooted at a given pair of values and returns a single value node that represents both the input values in the destination SMG. The function inputs a triple of SMGs G_1, G_2, G (two source SMGs and one destination SMG) and a pair of values v_1 and v_2 from G_1 and G_2, respectively. If the function fails in joining the given values, it returns $\bot$. Otherwise, it returns a triple of SMGs G_1', G_2', G', and a value v from G' such that:

- $MI(G_1) \subseteq MI(G_1')$ and $MI(G_2) \subseteq MI(G_2')$ where G_1 and G_2 can differ from G_1' and G_2', respectively, due to an application of join reinterpretation on some of the pairs of objects being joined only.
- The sub-SMGs G_1'' and G_2'' of G_1' and G_2' rooted at v_1 and v_2, respectively, are joined as the sub-SMG G'' of G' rooted at v, i.e., it is required that $MI(G_1'') \subseteq MI(G'') \supseteq MI(G_2'')$.
- The sub-SMG $G' \setminus G''$ is exactly the sub-SMG of G that consists of objects and values that are not removed in Step 11 of the $joinTargetObjects$ function due to using the principle of delayed join of sub-SMGs described in Appendix C.6.

Algorithm 5 $joinValues(s, G_1, G_2, G, m_1, m_2, v_1, v_2, l_{diff})$

Input:

- Initial join status $s \in \mathbb{J}$.
- SMGs $G_1 = (O_1, V_1, \Lambda_1, H_1, P_1)$, $G_2 = (O_2, V_2, \Lambda_2, H_2, P_2)$, and $G = (O, V, \Lambda, H, P)$.
- Injective partial mappings of nodes m_1, m_2 as defined in Appendix C.
- Values $v_1 \in V_1$ and $v_2 \in V_2$.
- Nesting level difference $l_{diff} \in \mathbb{Z}$.

Output:

- $\bot$ in case the sub-SMGs of G_1 and G_2 rooted at v_1 and v_2 cannot be joined.
- Otherwise, a tuple $(s', G_1', G_2', G', m_1', m_2', v')$ where:
  - $s' \in \mathbb{J}$ is the resulting join status.
  - G_1', G_2', G' are SMGs as defined in Appendix C.3.
  - m_1', m_2' are the resulting injective partial mappings of nodes.
  - v' is a value in G' satisfying the conditions stated in Appendix C.3.

Method:

1. If $v_1 = v_2$, return $(s, G_1, G_2, G, m_1, m_2, v_1)$. In this case, the given pair of values matches trivially, which happens whenever a shared value is reached during abstraction.
2. If $m_1(v_1) = m_2(v_2) = v \neq \bot$, return $(s, G_1, G_2, G, m_1, m_2, v)$. In this case, the pair of values is already joined.
3. If both v_1 and v_2 are non-address values, i.e., $P_1(v_1) = \bot$ and $P_2(v_2) = \bot$, then:
   - If $m_1(v_1) \neq \bot$ or $m_2(v_2) \neq \bot$, return $\bot$.
   - Create a new value node $v \in V$ such that $\mathit{level}(v) = max(\mathit{level}_1(v_1), \mathit{level}_2(v_2))$.
   - Extend the mapping of nodes such that $m_1(v_1) = m_2(v_2) = v$.
   - If $\mathit{level}_1(v_1) - \mathit{level}_2(v_2) < l_{diff}$, let $s := \mathit{updateJoinStatus}(s, \sqsubset)$.
   - If $\mathit{level}_1(v_1) - \mathit{level}_2(v_2) > l_{diff}$, let $s := \mathit{updateJoinStatus}(s, \sqsupset)$.
   - Return $(s, G_1, G_2, G, m_1, m_2, v_1)$.
4. If $P_1(v_1) = \bot$ or $P_2(v_2) = \bot$, return $\bot$.
5. Let $res := \mathit{joinTargetObjects}(s, G_1, G_2, G, m_1, m_2, v_1, v_2, l_{diff})$.
   If $res = \bot$, return $\bot$. If $res \neq \hookleftarrow$, then return res.
6. Let $o_1 := o(P_1(v_1))$ and $o_2 := o(P_2(v_2))$.
7. If $\mathit{kind}_1(o_1) = \texttt{dls}$, let $res = \mathit{insertLeftDlsAndJoin}(s, G_1, G_2, G, m_1, m_2, v_1, v_2, l_{diff})$.
   If $res = \bot$, return $\bot$. If $res \neq \hookleftarrow$, then return res.
8. If $\mathit{kind}_2(o_2) = \texttt{dls}$, let $res = \mathit{insertRightDlsAndJoin}(s, G_1, G_2, G, m_1, m_2, v_1, v_2, l_{diff})$.
   If $res \in \{\bot, \hookleftarrow\}$, return $\bot$. Otherwise, return res.

If the input values are identical ($v_1 = v_2$), the resulting value v is the same identical value, i.e., v_1 or v_2, which prevents shared nodes from being processed as nested data structures during abstraction. If both values are mapped to the same value node in the destination SMG, i.e., $m_1(v_1) = m_2(v_2)$, the node $m_1(v_1)$ (or, equivalently, $m_2(v_2)$) to which they are mapped is returned since such a pair of values has already been successfully joined before. A pair of non-address values visited for the first time, i.e., a pair of values for which $m_1(v_1) = \bot = m_2(v_2)$, is joined by creating a fresh value node v in G' with the appropriate nesting level, and the mapping of nodes is extended such that $m_1(v_1) = m_2(v_2) = v$. Non-address values are never joined with addresses—if such a situation occurs, the whole join operation fails.[20] For addresses seen for the first time, the algorithm tries to join their targets with each other with three possible outcomes:

1. The join succeeds, and $joinValues$ then succeeds too.
2. The join fails in a recoverable way (denoted by the result being $\hookleftarrow$). Intuitively, this happens when trying to join addresses that are found to be obviously different, i.e., addresses that are found to be different without going deeper into the sub-SMGs rooted at them—e.g., due to they point to some object with different offsets, different target specifiers, incompatible levels, size, validity, linking fields (for DLSs), or when the objects they point to are already mapped to some other objects. In case of the recoverable failure, if at least one of the target objects is a DLS, the algorithm tries to virtually *insert a DLS* in one of the SMGs, which allows it to create a $0+$ DLS in the destination SMG and continue by joining the appropriate successor values. If this fails too, the whole join operation fails.[21]
3. The join fails in an irrecoverable way (denoted by the result being $\bot$) in which case $joinValues$ fails too.

[20] Note that the handling of non-address values is kept simple since the basic notion of SMGs does not distinguish any special kinds of non-address values (numbers, intervals, etc.), but there is still room for improvement, especially in conjunction with the extensions described in Section 4.

[21] As mentioned already in Section 3.2, the recovery could be tried even when the impossibility of joining two addresses is discovered much later during joining the sub-SMGs rooted at the given addresses. Then, however, back-tracking would be necessary, which we try to avoid for efficiency reasons.

C.4 Join of Target Objects

The $joinTargetObjects$ function (cf. Alg. 6) joins a pair of sub-SMGs rooted at the given pair of addresses and returns a single address node that represents both the input addresses in the destination SMG. The function inputs a triple of SMGs G_1, G_2, G (two source SMGs and one destination SMG) and a pair of addresses a_1 and a_2 from G_1 and G_2, respectively. If the function fails in joining the given addresses, it returns $\bot$ in case of an irrecoverable failure and $\hookleftarrow$ in case of a recoverable failure (intuitively, for efficiency reasons, this happens only when the addresses or their targets are found different without going deeper in the sub-SMGs rooted at them). If the function succeeds, it returns a triple of SMGs G'_1, G'_2, G' and an address a from G' such that:

- $MI(G_1) \subseteq MI(G'_1)$ and $MI(G_2) \subseteq MI(G'_2)$ where G_1 and G_2 can differ from G'_1 and G'_2, respectively, due to an application of join reinterpretation on some of the pairs of objects being joined only.
- The sub-SMGs G''_1 and G''_2 of G'_1 and G'_2 rooted at a_1 and a_2, respectively, are joined as the sub-SMG G'' of G' rooted at a, i.e., it is required that $MI(G''_1) \subseteq MI(G'') \supseteq MI(G''_2)$.
- The sub-SMG $G' \setminus G''$ is exactly the sub-SMG of G that consists of objects and values that are not removed in Step 11 of the function due to using the principle of delayed join of sub-SMGs described in Appendix C.6.

The algorithm first compares the offsets and target specifiers that the points-to edges leading from a_1 and a_2 are labelled with. Then it checks whether the corresponding target objects $o_1 = o(P_1(a_1))$ and $o_2 = o(P_2(a_2))$ can be safely joined, in which case it creates a fresh object o in G that semantically covers both o_1 and o_2, a fresh address $a \in A$, and a points-to edge going from a to o. The mapping of nodes is then extended such that $m_1(o_1) = m_2(o_2) = o$ and $m_1(a_1) = m_2(a_2) = a$. Finally, the $joinSubSMGs$ function is called recursively for the triple o_1, o_2, o.

We allow DLSs to be joined with regions as well as with DLSs of a different minimal length, which requires the minimal length of the object o to be adjusted so that it covers both cases, i.e., $len(o) := min(len'_1(o_1), len'_2(o_2))$. It may also happen that the target objects were joined already but they are now reached through a different pair of addresses—in that case, we only create the appropriate address $a \in A$, the points-to leading from a to the already existing object $m_1(o_1) = m_2(o_2)$, and extend the mapping of addresses such that $m_1(a_1) = m_2(a_2) = a$. However, if o_1 and o_2 are mapped to different objects in G, i.e., $m_1(o_1) \neq m_2(o_2)$ and $m_1(o_1) \neq \bot \neq m_2(o_2)$, the join of target objects has to fail.[22]

The $joinTargetObjects$ function uses two auxiliary functions: in particular, the $mapTargetAddress$ function (cf. Alg. 7) and the $matchObjects$ function (cf. Alg. 8). The former is responsible for joining a pair of addresses pointing to a pair of objects that have already been joined and the latter compares a pair of objects and checks whether they can be joined safely.

[22] Due to the effect of the DLS insertion algorithm described in Appendix C.5, it may also happen that exactly one of the mappings $m_1(o_1)$, $m_2(o_2)$ is defined, which does not necessarily imply a join failure. This case is further discussed in Appendix C.6.

Algorithm 6 $joinTargetObjects(s, G_1, G_2, G, m_1, m_2, a_1, a_2, l_{diff})$

Input:
- Initial join status $s \in \mathbb{J}$.
- SMGs $G_1 = (O_1, V_1, \Lambda_1, H_1, P_1)$, $G_2 = (O_2, V_2, \Lambda_2, H_2, P_2)$, and $G = (O, V, \Lambda, H, P)$.
- Injective partial mappings of nodes m_1, m_2 as defined in Appendix C.
- Addresses $a_1 \in V_1$ and $a_2 \in V_2$.
- Nesting level difference $l_{diff} \in \mathbb{Z}$.

Output:
- $\perp$ in case of an irrecoverable failure.
- $\hookleftarrow$ in case of a recoverable failure.
- Otherwise, a tuple $(s', G_1', G_2', G', m_1', m_2', a')$ where:
  - $s' \in \mathbb{J}$ is the resulting join status.
  - G_1', G_2', G' are SMGs as defined in Appendix C.4.
  - m_1', m_2' are the resulting injective partial mappings of nodes.
  - a' is an address in G' satisfying the conditions stated in Appendix C.4.

Method:

1. If $of(P_1(a_1)) \neq of(P_2(a_2))$ or $level_1(a_1) - level_2(a_2) \neq l_{diff}$, return $\hookleftarrow$.
2. Let $o_1 := o(P_1(a_1))$ and $o_2 := o(P_2(a_2))$.
3. If $o_1 = \# = o_2$ or $m_1(o_1) = m_2(o_2) \neq \perp$, let $(G, m_1, m_2, a) := mapTargetAddress(G_1, G_2, G, m_1, m_2, a_1, a_2)$ and return $(s, G_1, G_2, G, m_1, m_2, a)$. In this case, the targets are already joined and we need to create a new address for the corresponding object $o \in O$.
4. If $kind_1(o_1) = kind_2(o_2)$ and $tg(P_1(a_1)) \neq tg(P_2(a_2))$, return $\hookleftarrow$.
5. If $kind_1(o_1) \neq kind_2(o_2)$ and $m_1(o_1) \neq m_2(o_2)$, return $\hookleftarrow$.
6. Let $s := matchObjects(s, G_1, G_2, m_1, m_2, o_1, o_2)$. If $s = \perp$, return $\hookleftarrow$.
7. Create a new object $o \in O$.
8. Initialize the labelling of o to match the labelling of o_1 if $kind_1(o_1) = \texttt{dls}$, or to match the labelling of o_2 if $kind_2(o_2) = \texttt{dls}$, otherwise take the labelling from any of them since both o_1 and o_2 are equally labelled regions.
9. If $kind_1(o_1) = \texttt{dls}$ or $kind_2(o_2) = \texttt{dls}$, let $len(o) := min(len_1'(o_1), len_2'(o_2))$.
10. Let $level(o) := max(level_1(o_1), level_2(o_2))$.
11. If $m_1(o_1) \neq \perp$, replace each edge leading to $m_1(o_1)$ by an equally labelled edge leading to o, remove $m_1(o_1)$ together with all nodes and edges of G that are reachable via $m_1(o_1)$ only, and remove the items of m_1 whose target nodes were removed. Likewise for m_2 and o_2. Note that this mechanism is called a *delayed join of sub-SMGs* and explained in Appendix C.6.
12. Extend the mapping of nodes such that $m_1(o_1) = m_2(o_2) = o$.
13. Let $(G, m_1, m_2, a) := mapTargetAddress(G_1, G_2, G, m_1, m_2, a_1, a_2)$.
14. Let $res := joinSubSMGs(s, G_1, G_2, G, m_1, m_2, o_1, o_2, o, l_{diff})$. If $res = \perp$, return $\perp$. Otherwise return $(s, G_1, G_2, G, m_1, m_2, a)$.

Algorithm 7 $mapTargetAddress(G_1, G_2, G, m_1, m_2, a_1, a_2)$

Input:

- SMGs $G_1 = (O_1, V_1, \Lambda_1, H_1, P_1)$, $G_2 = (O_2, V_2, \Lambda_2, H_2, P_2)$, and $G = (O, V, \Lambda, H, P)$ with the corresponding sets of addresses A_1, A_2, A.
- Injective partial mappings of nodes m_1, m_2 as defined in Appendix C.
- Addresses $a_1 \in A_1$ and $a_2 \in A_2$ referring with the same offset $of = of(P_1(a_1)) = of(P_2(a_2))$ to objects $o_1 = o(P_1(a_1))$ and $o_2 = o(P_2(a_2))$, respectively, which are already joined into an object $o = m_1(o_1) = m_2(o_2)$.

Output:

- A tuple (G', m_1', m_2', a) where:
  - G' is an SMG obtained from G by extending its set of addresses by an address a representing the join of a_1 and a_2 (unless G already contains this address) together with a points-to edge from a to o representing the join of the points-to edges between a_1, a_2 and o_1, o_2, respectively.
  - m_1' and m_2' are the resulting injective partial mappings of nodes that are either identical to m_1 and m_2 (if a already exists in G) or obtained from m_1 and m_2 by extending them such that $m_1'(a_1) = m_2'(a_2) = a$.

Method:

1. Let $o_1 := o(P_1(a_1))$, $of := of(P_1(a_1))$.
2. If $o_1 = \#$, let $o := \#$. Otherwise, let $o := m_1(o_1)$.
3. If $kind_1(o_1) = \texttt{dls}$, let $tg := tg(P_1(a_1))$. Otherwise, let $tg := tg(P_2(a_2))$.
4. If there is an address $a \in A$ such that $P(a) = (of, tg, o)$, return (G, m_1, m_2, a).
5. Extend A by a fresh address a, then extend P by a new points-to edge $a \xrightarrow{of,tg} o$.
6. Extend the mapping of nodes such that $m_1(a_1) = m_2(a_2) = a$.
7. Return (G, m_1, m_2, a).

Algorithm 8 $matchObjects(s, G_1, G_2, m_1, m_2, o_1, o_2)$

Input:
- Initial join status $s \in \mathbb{J}$.
- SMGs $G_1 = (O_1, V_1, \Lambda_1, H_1, P_1)$ and $G_2 = (O_2, V_2, \Lambda_2, H_2, P_2)$.
- Injective partial mappings of nodes m_1, m_2 as defined in Appendix C.
- Objects $o_1 \in O_1$ and $o_2 \in O_2$.

Output:
- $\bot$ in case o_1 and o_2 cannot be joined.
- Otherwise $s \in \mathbb{J}$ reflecting the impact of the labels of o_1 and o_2 on the relation of the semantics of G_1 and G_2.

Method:
1. If $o_1 = \#$ or $o_2 = \#$, return $\bot$.
2. If $m_1(o_1) \neq \bot \neq m_2(o_2)$ and $m_1(o_1) \neq m_2(o_2)$, return $\bot$.
3. If $m_1(o_1) \neq \bot$ and $\exists o_2' \in O_2 : \ m_1(o_1) = m_2(o_2')$, return $\bot$.
4. If $m_2(o_2) \neq \bot$ and $\exists o_1' \in O_1 : \ m_1(o_1') = m_2(o_2)$, return $\bot$.
5. If $\mathit{size}_1(o_1) \neq \mathit{size}_2(o_2)$ or $\mathit{valid}_1(o_1) \neq \mathit{valid}_2(o_2)$, return $\bot$.
6. If $\mathit{kind}_1(o_1) = \mathit{kind}_2(o_2) = \texttt{dls}$, then:
   - If $\mathit{nfo}_1(o_1) \neq \mathit{nfo}_2(o_2)$, $\mathit{pfo}_1(o_1) \neq \mathit{pfo}_2(o_2)$ or $\mathit{hfo}_1(o_1) \neq \mathit{hfo}_2(o_2)$, return $\bot$.
7. Collect the set F of all pairs (of, t) occurring in has-value edges leading from o_1 or o_2.
8. For each field $(\mathit{of}, t) \in F$ do:
   - Let $v_1 = H_1(o_1, \mathit{of}, t)$ and $v_2 = H_2(o_2, \mathit{of}, t)$.
   - If $v_1 \neq \bot \neq v_2$ and $m_1(v_1) \neq \bot \neq m_2(v_2)$ and $m_1(v_1) \neq m_2(v_2)$, return $\bot$.
9. If $\mathit{len}_1'(o_1) < \mathit{len}_2'(o_2)$ or $\mathit{kind}_1(o_1) = \texttt{dls} \ \wedge \ \mathit{kind}_2(o_2) = \texttt{reg}$, let $s := \mathit{updateJoinStatus}(s, \sqsupset)$.
10. If $\mathit{len}_1'(o_1) > \mathit{len}_2'(o_2)$ or $\mathit{kind}_1(o_1) = \texttt{reg} \ \wedge \ \mathit{kind}_2(o_2) = \texttt{dls}$, let $s := \mathit{updateJoinStatus}(s, \sqsubset)$.
11. Return s.

C.5 DLS Insertion

Assume that a pair of addresses a_1 and a_2 from SMGs G_1 and G_2 is to be joined in order to allow G_1 and G_2 to be joined into an SMG G. Further, assume that the objects that the addresses refer to cannot be joined, but at least one of them is a DLS—in particular, assume it is the object with address a_1, denote it d_1, and denote the other object o_2 (the other possibility being symmetric). As mentioned already at the beginning of Appendix C, in such a case, we proceed as though a_2 pointed to a 0+ DLS d_2 preceding o_2 and labelled equally as d_1 up to its length. We then join d_1 and d_2 into a single 0+ DLS d in G and continue by joining the addresses a_{next} and a_2 where a_{next} is the value stored in the next/prev pointer of d_1 (depending on whether we came to d_1 via the `fst` or `lst` target specifier, respectively). We call this mechanism a *DLS insertion* because it can be seen as if the join of objects was preceded by a virtual insertion of a DLS from one of the SMGs into the other SMG. This extension is possible since the semantics of a 0+ DLS includes the empty list, which can be safely assumed to appear anywhere, compensating a missing object in one of the SMGs.

Algorithm 9 implements the DLS insertion. The algorithm first checks whether the DLS d_1 from G_1 that we would like to virtually insert into G_2 has not been processed by the join algorithm already in the past. If this is the case and there is some object o in G_2 that has been joined with d_1 into d, the join fails since the d_2 segment (possibly represented by a region as its concrete instance) is not missing, but it is not connected to the rest of the SMG in a way compatible with d_1 (at least not for the current way G_1 and G_2 are being joined). If no such object o exists, the DLS d to which d_1 is mapped in G is used as the result of joining d_1 with the virtually added segment d_2, and the join continues by the addresses a_{next} and a_2. Intuitively, this latter case arises when inserting a single missing segment that should be reachable through several paths in the SMG. Note that such a situation is, in fact, quite usual since a DLS can be reached both forward and backward. The algorithm, however, has to insert a single virtual segment d_2 for all such paths.

If d_1 has not yet been processed, the algorithm checks whether there is some hope that the virtual insertion of d_2 could help (or whether it is better to try to proceed with the join in some other way: e.g., try to insert a DLS from G_2 into G_1 in case both of the addresses a_1 and a_2 point to DLSs or fall-back to introducing a 0/1 abstract object as mentioned in Section 4). However, unlike in the function $joinTargetObjects$, if we do not want to go deeper in the SMGs (which we do not want for efficiency reasons), there is not so many properties to check since we do not have two objects whose labelling we could compare, but the single DLS d_1 whose counterpart we want to virtually insert into G_2 only. So, we at least check that there is no conflict of the successor addresses a_{next} and a_2 according to the current mapping of nodes, i.e., we require $m_1(a_{next}) = \bot \ \vee \ m_2(a_2) = \bot \ \vee \ m_1(a_{next}) = m_2(a_2)$.

If the above checks pass, the DLS insertion proceeds as follows: Let F be the set of the linking fields of d_1 that are oriented forward wrt. the direction of the traversal. In particular, let $F = \{nfo_1(d_1)\}$ if $tg(P_1(a_1)) = \texttt{fst}$, or $F = \{pfo_1(d_1)\}$ if $tg(P_1(a_1)) = \texttt{lst}$. First, the DLS d representing the join of d_1 and the virtually inserted d_2 is created in the destination SMG with the same labelling as that of d_1 up to $len(d) = 0$. Together with basically copying the DLS d_1 from G_1 to G, we also copy the F-restricted sub-SMG rooted at it from G_1 into G, excluding the nodes for which m_1 is already defined (these were already reached through some other paths in the past, and the newly copied part of the F-restricted sub-SMG rooted at d_1 is just linked to them). Subsequently, the algorithm extends the mapping m_1 for the nodes newly inserted to G, creates the appropriate address node $a \in A$ as well as the points-to edge leading from a to d, and extends the mapping of addresses such that $m_1(a_1) = a$. The algorithm then continues by joining the pair of successor values a_{next} and a_2.

Algorithm 9 *insertLeftDlsAndJoin*$(s, G_1, G_2, G, m_1, m_2, a_1, a_2, l_{diff})$

Input:
- Initial join status $s \in \mathbb{J}$.
- SMGs $G_1 = (O_1, V_1, \Lambda_1, H_1, P_1)$, $G_2 = (O_2, V_2, \Lambda_2, H_2, P_2)$, and $G = (O, V, \Lambda, H, P)$.
- Injective partial mappings of nodes m_1, m_2 as defined in Appendix C.
- Values $a_1 \in V_1$ such that $\mathit{kind}_1(o(P_1(a_1))) = \texttt{dls}$ and $a_2 \in V_2$.
- Nesting level difference $l_{diff} \in \mathbb{Z}$.

Output:
- $\bot$ in case of unrecoverable failure.
- $\hookleftarrow$ in case of recoverable failure.
- Otherwise, a tuple $(s', G_1', G_2', G', m_1', m_2', a')$ where:
  - $s' \in \mathbb{J}$ is the resulting join status.
  - G_1', G_2', G' are SMGs as defined in Appendix C.4.
  - m_1', m_2' are the resulting injective partial mappings of nodes.
  - a' is an address in G' satisfying the conditions stated in Appendix C.4.

Method:

1. Let $(d_1, \mathit{of}, \mathit{tg}) := P_1(a_1)$.
2. If $\mathit{tg} = \texttt{fst}$, let $nf := \mathit{nfo}_1(d_1)$; if $\mathit{tg} = \texttt{lst}$, let $nf := \mathit{pfo}_1(d_1)$; otherwise return $\hookleftarrow$.
3. Let $a_{next} := H_1(d_1, nf, \texttt{ptr})$.
4. If $m_1(d_1) \neq \bot$, then:
   - Let $d := m_1(d_1)$.
   - If $\exists o \in O : \ m_2(o) = d$, return $\hookleftarrow$.
   - If $m_1(a_1) = \bot$, create a new value node $a \in V$ and a new edge $a \xrightarrow{\mathit{of},\mathit{tg}} d$ in P, and extend the mapping of nodes such that $m_1(a_1) = a$. Otherwise let $a := m_1(a_1)$ and return $(s, G_1, G_2, G, m_1, m_2, a)$.
   - Let $res := \mathit{joinValues}(s, G_1, G_2, G, m_1, m_2, a_{next}, a_2, l_{diff})$. If $res = \bot$, return $\bot$. Otherwise let $(s, G_1, G_2, G, m_1, m_2, a) := res$.
5. If $m_1(a_{next}) \neq \bot$ and $m_2(a_2) \neq \bot$ and $m_1(a_{next}) \neq m_2(a_2)$, return $\hookleftarrow$.
6. Let $s' := (\mathit{len}_1(d_1) = 0) \ ? \ \sqsupset \ : \ \bowtie$. Let $s := \mathit{updateJoinStatus}(s, s')$.
7. Extend G by a fresh copy of the $\{nf\}$-restricted sub-SMG of G_1 rooted at d_1, but excluding the nodes that are already mapped in m_1 such that the copy of d_1 is a DLS d. Then extend the mapping m_1 such that the newly created nodes in $O \cup V$ are mapped from the corresponding nodes of $O_1 \cup V_1$.
8. Initialize the labelling of d to match the labelling of d_1 up to minimum length, which is fixed to zero, i.e., $\mathit{len}(d) = 0$.
9. Let $a \in V$ be the address such that $P(a) = (\mathit{of}, \mathit{tg}, d)$ if such an address exists in G. Otherwise, create a new value node $a \in V$ and a new edge $a \xrightarrow{\mathit{of},\mathit{tg}} d$ in P, and extend the mapping of nodes such that $m_1(a_1) = a$.
10. Let $res := \mathit{joinValues}(s, G_1, G_2, G, m_1, m_2, a_{next}, a_2, l_{diff})$. If $res = \bot$, return $\bot$. Otherwise let $(s, G_1, G_2, G, m_1, m_2, a') := res$.
11. Introduce a new has-value edge $d \xrightarrow{nf,\texttt{ptr}} a'$ in H.
12. Return $(s, G_1, G_2, G, m_1, m_2, a)$.

C.6 Delayed Join of Sub-SMGs

The mechanism of DLS insertion increases the chances for two SMGs to be successfully joined, but it brings one complication to be taken care of. Assume that SMGs G_1 and G_2 are being joined into an SMG G. When applying the DLS insertion mechanism on a DLS d_1 from G_1, the not yet traversed sub-SMG G'_1 of G_1 reachable from d_1 is copied into G too (likewise for the symmetric case). However, some nodes of G'_1 that are inserted into G_2 may in fact exist in G_2 and be reachable through some other address than the address at which the DLS insertion is started. Note that this *may* but *needs not* happen, and at the time when the DLS insertion is run, it is unknown which of the two cases applies. In theory, a backward traversal through the SMGs could be used here, but we chose not to use it since we were afraid of its potential bad impact on the performance. That is why we always insert a DLS together with the sub-SMG rooted at it, and as soon as we realize that some DLS d_1 from G_1 whose counterpart was inserted into G_2 does have a real counterpart in G_2, the result of the join of d_1 with the inserted DLS is deleted (together with all the values and objects reachable from that DLS only) and a proper join—which we denote as the so-called *delayed join*—is used instead (cf. Point 11 of the function $joinTargetObjects$). Note that running the delayed join is indeed necessary since the insertion of a DLS is optimistic in that its sub-SMG is either missing too, or if it is not missing, it is the same as the sub-SMG of the inserted DLS. This needs, however, not to be the case.

C.7 Join of SPCs

The $joinSPCs$ function (cf. Alg. 10) is the top-level of the join algorithm used when reducing the number of SPCs generated for particular basic blocks of the program being analyzed. It inputs a pair of garbage-free SPCs $C_1 = (G_1, \nu_1)$, $C_2 = (G_2, \nu_2)$ where $range(\nu_1) = range(\nu_2) =$ *Var* is the common set of program variables, and for each $x \in$ *Var*, the objects $\nu_1(x)$ and $\nu_2(x)$ are labelled equally (this condition necessarily holds for SPCs generated for the same program). The algorithm either fails and returns $\perp$, or it returns the resulting join status and an SPC $C = (G, \nu)$ where $range(\nu) =$ *Var*, and the triple G_1, G_2, G satisfies the assertions about joined SMGs stated in Section 3.2.

The function starts by initializing the mappings of nodes m_1 and m_2 to the empty set and the join status s to $\simeq$. Then, for each program variable $x \in$ *Var*, a fresh region r is created in G, labelled equally as r_1 (or r_2), and the mappings are extended such that $\nu(x) = m_1(r_1) = m_2(r_2) = r$ where $r_1 = \nu_1(x)$ and $r_2 = \nu_2(x)$. Next, for each program variable $x \in$ *Var*, the $joinSubSMGs$ function is called with the corresponding triple of objects $\nu_1(x), \nu_2(x), \nu(x)$. The value of m_1, m_2, and s gets propagated between each pair of subsequent calls. Subsequently, the $joinSPCs$ function checks whether there was not created any cycle consisting of 0+ DLSs only in G, and if so, the algorithm fails.

Algorithm 10 $joinSPCs(C_1, C_2)$

Input:
- Garbage-free SPCs $C_1 = (G_1, \nu_1)$, $C_2 = (G_2, \nu_2)$ with SMGs $G_1 = (O_1, V_1, \Lambda_1, H_1, P_1)$, $G_2 = (O_2, V_2, \Lambda_2, H_2, P_2)$ where $range(\nu_1) = range(\nu_2) =$ *Var*, and for each $v \in$ *Var*, the labelling of $\nu_1(v)$ is equal to the labelling of $\nu_2(v)$.

Output:
- $\bot$ in case C_1 and C_2 cannot be joined.
- Otherwise, a tuple (s, C) where:
  - $s \in \mathbb{J}$ is the resulting join status.
  - $C = (G, \nu)$ where $range(\nu) =$ *Var* and the SMG G satisfies the condition *MI*$(G_1) \subseteq$ *MI*$(G) \supseteq$ *MI*(G_2).

Method:
1. Let G be an empty SMG G, $\nu = m_1 = m_2 = \emptyset$, $s = \simeq$.
2. For each program variable $v \in$ *Var*:
   - Let $r_1 := \nu_1(v)$ and $r_2 := \nu_2(v)$.
   - Create a fresh region $r \in O$, initialize its labelling to match the labelling of r_1.
   - Extend the mappings such that $m_1(r_1) = m_2(r_2) = \nu(v) = r$.
3. For each program variable $v \in$ *Var*:
   - Let $r_1 := \nu_1(v)$, $r_2 := \nu_2(v)$, and $r := \nu(v)$.
   - Let $res := joinSubSMGs(s, G_1, G_2, G, m_1, m_2, r_1, r_2, r, 0)$.
   - If $res = \bot$, return $\bot$. Otherwise let $(s, G_1, G_2, G, m_1, m_2) := res$.
4. If there is any cycle consisting of 0+ DLSs only in G, return $\bot$.
5. Return (s, C) where $C = (G, \nu)$.

C.8 Join of Sub-SMGs within Abstraction

The $joinSubSMGsForAbstraction$ function (cf. Alg. 11) implements the core functionality of the elementary merge operation used as a part of our abstraction mechanism (Section 3.3). It inputs an SMG $G = (O, V, \Lambda, H, P)$, a pair of objects $o_1, o_2 \in O$, and a triple of binding offsets *hfo*, *nfo*, *pfo* $\in \mathbb{N}$. If it succeeds, it returns an SMG $G' = (O', V', \Lambda', H', P')$ and a fresh DLS $d \in O'$ which represents the merge of o_1 and o_2 in G' and which is the entry point of the sub-SMG representing a join of the sub-SMGs rooted at o_1 and o_2. What remains to be done in the elementary merge operation is the reconnection of the pointers surrounding o_1 and o_2 to d (apart from those related to their nested sub-SMGs), cf. Section 3.3. As an auxiliary result (used in the algorithm of searching for longest mergeable sequences), the algorithm returns the join status $s \in \mathbb{J}$ comparing the semantics of the sub-SMGs rooted at o_1 and o_2 as well as the sets $O_1, O_2 \subseteq O'$ and $V_1, V_2 \subseteq V'$ that contain those objects and values whose join produced the sub-SMG nested below d. Note that the join status returned by $mergeSubSMGs$ is not affected by the kinds of o_1, o_2 and the values in their next/prev fields since the loss of information due to merging o_1 and o_2 into a single list segment is deliberate.

The function proceeds as follows. Using the offsets *nfo*, *pfo*, the values of the next/prev fields of o_1 and o_2 are remembered and temporarily replaced by 0 (in order for the subsequently started join of sub-SMGs not to go through these fields). Then a fresh DLS d is created in O and labelled by the given offsets *hfo*, *nfo*, *pfo* and the minimum length equal to $len'_1(o_1) + len'_2(o_2)$, other labels are taken from o_1 (or o_2 since the other labels are equal). The mapping of objects is initialized such that $m_1(o_1) = m_2(o_2) = d$ and the nesting level difference is initialized based on $kind(o_1)$ and $kind(o_2)$ using the rules stated in Appendix C.2. The generic algorithm $joinSubSMGs$ is then called on the triple o_1, o_2, d. If it fails or the resulting SMG contains any cycle consisting of 0+ DLSs only, the algorithm exits unsuccessfully. If it succeeds, the values of the next/prev fields in o_1, o_2, which were temporarily replaced by 0, are restored to their original values. If $kind(o_1) = kind(o_2) =$ `reg`, the level of each node that appears in the image of m_1 (or m_2) is increased by one (since these nodes are now recognized as nested), and all points-to edges leading to d are relabelled by the `all` target specifier.

The resulting sets of nodes are computed as follows: $O_1 := range(m_1) \cap O$, $O_2 := range(m_2) \cap O$, $V_1 := range(m_1) \cap V$, and $V_2 = range(m_2) \cap V$. The resulting join status is the status returned by $joinSubSMGs$.

Algorithm 11 $joinSubSMGsForAbstraction(G, o_1, o_2, \mathit{hfo}, \mathit{nfo}, \mathit{pfo})$

Input:
- SMGs $G = (O, V, \Lambda, H, P)$.
- Objects $o_1, o_2 \in O$ of the same level that are the roots of the $\{\mathit{nfo}, \mathit{pfo}\}$-restricted sub-SMGs G_1 and G_2 of G that are to be joined.
- Candidate DLS offsets $\mathit{hfo}, \mathit{nfo}, \mathit{pfo} \in \mathbb{N}$.

Output:
- $\bot$ in case G_1 and G_2 cannot be joined.
- Otherwise, a tuple $(s, G', d, O_1, V_1, O_2, V_2)$ where:
  - $s \in \mathbb{J}$ is the resulting join status (determines the cost of joining G_1 and G_2).
  - G' is an SMG obtained from the input SMG G by extending it with a new DLS d below which the join of G_1 and G_2 is nested.
  - $O_i \subseteq O$ and $V_i \subseteq V$ for $i = 1, 2$ are sets of non-shared objects and values of G_1 and G_2, respectively.

Method:

1. Let $a_p := H(o_1, \mathit{pfo}, \texttt{ptr})$, $a_n := H(o_2, \mathit{nfo}, \texttt{ptr})$, $a_1 := H(o_1, \mathit{nfo}, \texttt{ptr})$, and $a_2 := H(o_2, \mathit{pfo}, \texttt{ptr})$.
2. Replace each has-value edge of H leading from o_1 or o_2 and labelled by $(\mathit{nfo}, \texttt{ptr})$ or $(\mathit{pfo}, \texttt{ptr})$ by a has-value edge leading to 0 and having the same label.
3. Extend O with a fresh valid DLS d and label it with the head, next, and prev offsets *hfo*, *nfo*, and *pfo*, the minimum length $\mathit{len}'(o_1) + \mathit{len}'(o_2)$, level $\mathit{level}(o_1)$, and the size $\mathit{size}(o_1)$, which are assumed to be the same as $\mathit{level}(o_2)$ and $\mathit{size}(o_2)$, respectively.
4. If $\mathit{kind}(o_1) = \mathit{kind}(o_2)$, let $l_{diff} := 0$. Otherwise, let $l_{diff} := (\mathit{kind}(o_1) = \texttt{dls})?1 : -1$.
5. Let $res := joinSubSMGs(\simeq, G_1, G_2, G, \{(o_1, d)\}, \{(o_2, d)\}, o_1, o_2, d, l_{diff})$. If $res = \bot$, return $\bot$. Otherwise let $(s, \_, \_, G, m_1, m_2) := res$.
6. If G contains any cycle consisting of 0+ DLSs only, return $\bot$.
7. If $\mathit{kind}(o_1) = \mathit{kind}(o_2) = \texttt{reg}$, increase by one the level of each object and value that appears in the image of m_1 or m_2, and relabel all points-to edges leading to d by the `all` target specifier.
8. Drop the temporarily created has-value edges of H leading from o_1 and o_2 to 0 and labelled by $(\mathit{nfo}, \texttt{ptr})$ or $(\mathit{pfo}, \texttt{ptr})$ and restore the original has-value edges $o_1 \xrightarrow{\mathit{pfo}, \texttt{ptr}} a_p$, $o_1 \xrightarrow{\mathit{nfo}, \texttt{ptr}} a_1$, $o_2 \xrightarrow{\mathit{pfo}, \texttt{ptr}} a_2$, and $o_2 \xrightarrow{\mathit{nfo}, \texttt{ptr}} a_n$.
9. Return $(s, G, d, O \cap range(m_1), V \cap range(m_1), O \cap range(m_2), V \cap range(m_2))$.

D Longest Mergeable Sequences

In this appendix, we formalize the notion of longest mergeable sequences informally introduced in Section 3.3. Assume an SPC $C = (G, \nu)$ where $G = (O, V, \Lambda, H, P)$ is an SMG with the sets of regions R, DLSs D, and addresses A. The *longest mergeable sequence* of objects given by a candidate DLS entry $(o_c, \mathit{hfo}_c, \mathit{nfo}_c, \mathit{pfo}_c)$ where $o_c \in O$ is the longest sequence of distinct valid heap objects of G whose first object is o_c, all objects in the sequence are of level 0, all DLSs that appear in the sequence have hfo_c, nfo_c, pfo_c as their head, next, prev offsets, and the following holds for any two neighbouring objects o_1 and o_2 in the sequence:

1. The objects o_1 and o_2 are doubly linked, i.e., there are $a_1, a_2 \in A$ such that $o_1 \xrightarrow{\mathit{nfo}_c,\mathtt{ptr}} a_1 \xrightarrow{\mathit{hfo}_c,\mathit{tg}_2} o_2$ for $\mathit{tg}_2 \in \{\mathtt{fst}, \mathtt{reg}\}$ and $o_2 \xrightarrow{\mathit{pfo}_c,\mathtt{ptr}} a_2 \xrightarrow{\mathit{hfo}_c,\mathit{tg}_1} o_1$ for $\mathit{tg}_1 \in \{\mathtt{lst}, \mathtt{reg}\}$.
2. The $\{\mathit{nfo}_c, \mathit{pfo}_c\}$-restricted sub-SMGs G_1, G_2 of G rooted at o_1 and o_2 can be joined using the extended join algorithm that yields the sub-SMG $G_{1,2}$ to be nested below the join of o_1 and o_2 as well as the sets O_1, V_1 and O_2, V_2 of non-shared objects and values of G_1 and G_2, respectively, whose join gives rise to $G_{1,2}$.
3. The non-shared objects and values of G_1 and G_2 (other than o_1 and o_2 themselves) are reachable via o_1 or o_2, respectively, only. This is, $\forall a \in A \setminus V_1\ \forall o' \in O_1 \setminus \{o_1\} : o(P(a)) \neq o',\ \forall o' \in O \setminus O_1\ \forall v \in V_1 :\ v \notin H(o')$, and likewise for o_2, V_2, and O_2. Moreover, the sets O_1 and O_2 contain heap objects only.
4. The objects o_1 and o_2 are a part of an uninterrupted sequence. Therefore:
   (a) Regions that are not the first nor last in the sequence can be pointed to their head offset from their predecessor, successor, or from their non-shared restricted sub-SMG only. Formally, if $o_1 \in R \setminus \{o_c\}$, then $\neg\, \exists o \in O \setminus (O_1 \cup \{o_2, o'\})\ \exists a \in A\ \exists \mathit{of} \in \mathbb{N} :\ o \xrightarrow{\mathit{of},\mathit{ptr}} a \xrightarrow{\mathit{hfo}_c,\mathtt{reg}} o_1$ where o' is the predecessor of o_1, i.e., $o' = o(P(H(o_1, \mathit{pfo}_c, \mathtt{ptr})))$.[23]
   (b) If o_1 (o_2) is a DLS, the only object that can point to its end (beginning) is o_2 (o_1), respectively. Formally, if $o_1 \in D$, then $\neg\, \exists o \in O \setminus \{o_2\}\ \exists a \in A\ \exists \mathit{of} \in \mathbb{N} : o \xrightarrow{\mathit{of},\mathtt{ptr}} a \xrightarrow{\mathit{hfo}_c,\mathtt{lst}} o_1$. If $o_2 \in D$, then $\neg\, \exists o \in O \setminus \{o_1\}\ \exists a \in A\ \exists \mathit{of} \in \mathbb{N} : o \xrightarrow{\mathit{of},\mathtt{ptr}} a \xrightarrow{\mathit{hfo}_c,\mathtt{fst}} o_2$.
   (c) Finally, only non-shared objects of G_1 and G_2 can point to non-head offsets of o_1 and o_2, respectively. Formally, $\neg\, \exists o \in O \setminus O_1\ \exists a \in A\ \exists \mathit{of}, \mathit{of}' \in \mathbb{N}\ \exists \mathit{tg} \in \mathbb{S} :\ o \xrightarrow{\mathit{of},\mathtt{ptr}} a \xrightarrow{\mathit{of}',\mathit{tg}} o_1\ \wedge\ \mathit{of}' \neq \mathit{hfo}_c$, and likewise for o_2 and O_2.

[23] Note that no special formal treatment is needed for o_2 since it will take the role of o_1 in the next step. The above also implicitly ensures that pointers to the head offset of the last object are not restricted.

E Symbolic Execution of Conditional Statements

Checking equality of values is trivial in SMGs since it reduces to identity checking. To check non-equality, we propose a sound, efficient, but incomplete approach as mentioned already in Section 3.4. This approach is formalized in the function $proveNeq$ shown as Algorithm 13 that contains a number of comments to make it self-explaining. The algorithm uses the $lookThrough$ function (Algorithm 12) for traversing chains of $0+$ DLSs while looking for objects whose existence is guaranteed and whose unique addresses can serve as a basis for a non-equality proof. Note that the algorithms do not allow for comparing values of fields with incompatible types. Hence, we require all type-casts to be explicitly represented as separate instructions of the intermediate code so that the fields being compared are always of compatible types.

If neither equality nor inequality of a pair of values v_1 and v_2, which are compared in a conditional statement, can be established, the symbolic execution must follow both branches of the conditional statement. For each of the branches, we attempt to reflect the condition allowing the execution to enter that branch in the SMG G to be processed in the branch, effectively reducing the semantics of G. However, for efficiency reasons, we do again not reflect all consequences of the branch conditions that could in theory be reflected, but only the easy to handle ones, which is sound, and it suffices in all the case studies that we have considered. In particular, we restrict SMGs according to the branch conditions as follows (if none of the below described cases applies, the SMGs are not modified):

- If v_1 and/or v_2 are non-address values, one of them is replaced by the other in the $v_1 = v_2$ branch (a non-address value can be replaced by an address but not vice versa).
- If there is a chain of $0+$ DLSs connected into a doubly-linked list in the given SMG such that the `fst` address of the first DLS is v_1 and the last DLS contains v_2 in its next field (or vice versa), the chain is removed by calling the DLS removal algorithm repeatedly in the $v_1 = v_2$ branch. In the $v_1 \neq v_2$ branch, the computation is split to as many cases as the number of $0+$ DLSs in the chain is, and in each of the cases, the minimum length of one of the DLSs is incremented (reflecting that at least one of them must be non-empty).
- If v_1 points to a DLS d with the `fst` target specifier and v_2 points to d with the `lst` target specifier (or vice versa), it is clear that $\mathit{len}(d) < 2$ since otherwise $proveNeq$ would succeed in proving the inequality between v_1 and v_2. In this case, the following two modifications of the encountered SMGs can be applied in the different branches of the encountered conditional statement:
  - In the $v_1 = v_2$ branch, if $\mathit{len}(d) = 1$, the DLS d is replaced by an equally labelled region (excluding the DLS-specific labels) since d must consist of exactly one concrete node in this case.
  - In the $v_1 \neq v_2$ branch, if $\mathit{len}(d) = 1$ or the value of the next address is equal to the value of the prev address, i.e., $H(d, \mathit{pfo}(d), \texttt{ptr}) = H(d, \mathit{nfo}(d), \texttt{ptr})$, then the minimum length of d is increased to 2 since d must consist of at least two concrete nodes in this case.

Besides equality checking, we also allow for comparisons of addresses using the *less than* or *greater than* operators in case both of the addresses point to the same (concrete) object—we simply compare the offsets. This functionality is needed for successful verification of the NSPR-based case studies mentioned in Section 6.

Algorithm 12 $lookThrough(G, v)$

Input:
- An SMG $G = (O, V, \Lambda, H, P)$.
- A value $v \in V$ such that *level*$(v) = 0$.

Output:
- A pair $(v', visited)$ where:
  - $v' \in V$ is the value reached after all 0+ DLSs are traversed.
  - $visited$ is the set of all 0+ DLSs reachable from v in the forward direction without traversing any other object.

Method:
1. Let $visited := \emptyset$.
2. Let $o := o(P(v))$.
3. If $o \notin \{\bot, \#\}$ and *len*$'(o) = 0$, then:
   - Let $visited := visited \cup \{o\}$.
   - If *tg*$(P(v)) =$ `fst`, let $v := H(o,$ *nfo*$(o),$ `ptr`$)$ and continue with step 2.
   - If *tg*$(P(v)) =$ `lst`, let $v := H(o,$ *pfo*$(o),$ `ptr`$)$ and continue with step 2.
4. Return $(v, visited)$.

Algorithm 13 $proveNeq(G, v_1, v_2)$

Input:
- An SMG $G = (O, V, \Lambda, H, P)$.
- A pair of values $v_1, v_2 \in V$ such that *level*$(v_1) =$ *level*$(v_2) = 0$.

Output:
- *true* if the inequality between the given pair of values was proven, *false* otherwise.

Method:
1. Let $(v_1, O_1) := lookThrough(G, v_1)$.
2. Let $(v_2, O_2) := lookThrough(G, v_2)$.
3. If $v_1 = v_2$ or $O_1 \cap O_2 \neq \emptyset$, return *false*. // possible sharing of values
4. If $v_1 \notin A$ or $v_2 \notin A$, return *false*. // simplified handling of data values
5. Let $o_1 := o(P(v_1))$ and $o_2 := o(P(v_2))$.
6. If $o_1 = o_2$:
   - If *tg*$(P(v_1)) =$ *tg*$(P(v_2))$, return *true*. // same object, different offsets
   - If *tg*$(P(v_1)) =$ `fst` and *tg*$(P(v_2)) =$ `lst`, return *len*$'(o_1) \geq 2$.
   - If *tg*$(P(v_1)) =$ `lst` and *tg*$(P(v_2)) =$ `fst`, return *len*$'(o_1) \geq 2$.
   - Otherwise return *false*.
7. If *of*$(P(v_1)) < 0$ or *of*$(P(v_2)) < 0$, return *false*. // out of bounds
8. If $v_1 \neq 0$ and *size*$(o_1) \leq$ *of*$(P(v_1))$, return *false*. // out of bounds
9. If $v_2 \neq 0$ and *size*$(o_2) \leq$ *of*$(P(v_2))$, return *false*. // out of bounds
10. If $v_1 = 0$ or $v_2 = 0$, return *true*. // 0 and a valid address of an object
11. Return *valid*$(o_1) \wedge$ *valid*(o_2). // addresses of allocated objects