Abstraction-guided Runtime Checking of Assertions on Lists

Alex Gyori, Pranav Garg, Edgar Pek, P. Madhusudan

University of Illinois at Urbana-Champaign, Urbana, IL, USA
{gyori, garg11, pek1, madhu}@illinois.edu

Abstract. We investigate ways to specify and check, at runtime, assertions that express properties of dynamically manipulated linked-list data structures. Checking an assertion involving whether pointers point to a valid linked list and separation properties of these lists typically requires linear or even quadratic time on the size of the heap. Our main contribution is a way to scale this checking by orders of magnitude, using a novel idea called abstraction-guided runtime checking, whereby we maintain an accurate abstraction of the dynamic heap by utilizing the evolving runtime state, and where the abstraction helps in checking the runtime assertions much faster. We develop this synergistic combination of abstractions and runtime checking for lists, list-segments, and their separation, implement it, and show the tremendous performance gains it yields. In particular, when lists are manipulated using library functions, maintenance of the abstraction is within the libraries and yields constant runtime checking of assertions in the client code. We show that, as the number of assertions get frequent and the data structures get large, abstraction-guided runtime checking, which includes maintenance of the abstraction and the runtime checks, gives close to constant-time per assertion overhead in practice.

1 Introduction

Assertions are one of the most useful techniques for detecting errors and providing information about fault locations [8,12], and are used widely in testing production code. The Eiffel system pioneered the systematic usage of assertions in terms of contracts and invariants, and led to wider adoption in mainstream programming languages. The JML notation [15] and the Spec# language [4] and Code Contracts [3] systems at Microsoft are examples of specification languages that were influenced by Eiffel [19]; the latter were used for writing specifications mainly for testing code within Microsoft. Assertions are widely used; for instance, a study showed that there are more than a quarter million assertions in the Microsoft Office suite [12]. Since they are simple to write and popular because programmers use them for testing, they are the most available form of specifications for more sophisticated automated analyses, such as for unit-testing [28], test-input generation [10,7], regression testing [20], etc..

In this paper, we study the problem of expressing and efficiently checking assertions over list-segments, lists, and the way they merge or remain separate, when dynamically allocated and manipulated by a program using pointers. Mainstream programming languages lack any standard assertion logic for the heap, and programmers often write

assertions by writing *procedures* that check properties. For instance, in Java, programmers write so-called REPOK methods for checking representation class invariants during testing [16]. We develop efficient runtime checkinf for a *declarative and logical* specification language for expressing properties of list data-structures.

In this paper, we focus on a *separation logic* [22,25] over lists and list-segments. Separation logic is a succinct logic that can express complex properties of the heap, including structure (e.g., "x points to a list") and separation ("the lists pointed to by x and y are disjoint"). For instance, the succinct assertion:

```
assert lseg(x,y) * list(z) * true
```

expresses that there is a list segment from x to y (defined by a *next* pointer), and z points to a list, and these two are disjoint sets of locations of the heap.

The simplest runtime check on the heap for the above property would take quadratic time on the length of the lists involved; in general, checking `list` or `lseg` requires linear time, while checking separation properties that involve checking whether heaplets (subparts of the heap) are disjoint take quadratic time (though this can be made linear with techniques such as hashing or using additional linear space).

Note that when the sizes of the data structures get large (several thousand nodes), linear-time algorithms cause nontrivial overhead (and quadratic-time algorithms are just not acceptable), and hence the runtime checking of such data structures, *whether written by the programmer* or generated automatically from the specification, do not scale.

The key contribution of this paper is a new idea, which we call *abstraction-guided runtime assertion checking*. The idea is to maintain an abstraction of the concrete heap dynamically, as the program executes, which will maintain certain structural properties of the heap symbolically. (Note that the abstraction is *not* part of a static analysis—the abstraction is maintained as the program executes.) This abstraction obviates the need to dynamically check crucial and expensive properties, like separation, on the concrete heap. However, note that in classical abstractions of heaps, such as static shape analysis [18,26], the abstraction will always lose accuracy in reflecting the program's runtime state, and hence will quickly evolve into a "blob" of uncertainty that isn't very useful.

We propose an abstraction of lists and list segments that involves both a structural abstraction and a concrete mapping that relates the abstract vertex to concrete memory addresses on the runtime heap. Using this map, we show that we can keep the abstraction in check by using the runtime state as the program executes. Consequently, the structural abstraction always accurately reflects the runtime heap. Furthermore, when we reach a point in the program with an assertion, we show that we can, instead of evaluating the formula on the concrete heap, check it much faster by evaluating it on its abstraction instead.

We emphasize that the curious synergy between the abstraction and runtime checking is also inexpensive. Maintaining the abstract state relies on the concrete state and imposes a low overhead on the runtime execution, but brings huge performance gains when assertions are checked using the abstraction. As far as we know, this is the first time abstractions have been shown to be useful in speeding up runtime checking.

Let us consider software that consists of *library code* implementing low-level manipulations of lists, etc., with abstract data-type interfaces for collections or sequences,

and *client code* calling such libraries to implement a higher-level functionality. Both the library code and the client code could contain assertions about the lists in the heap, written by the programmer. In such a setting, our scheme would work as follows:

- We maintain an abstraction A of the dynamic heap pertaining to all list nodes globally.
- Assertions in the client code are transformed to check assertions on the abstraction A instead. The client code (which does not manipulate lists directly) is left untouched otherwise.
- The assertions in the library code are also transformed to check assertions on the abstraction A. In addition, all manipulations of the lists within the library methods are *instrumented* to update and maintain the abstract heap A. Furthermore, updating the abstraction correctly and accurately itself *requires the concrete heap* of the program.

The abstractions are typically of *constant* length (i.e., they are as long as the number of program pointer variables, but typically do not grow unboundedly with the length of the lists they represent) and hence checking of assertions is achieved usually in *constant time*, as opposed to linear/quadratic in the size of the heap. However, note that maintaining the abstraction does incur an overhead— typically, every update to the heap requires an instrumentation to maintain the abstract heap accurately, which generally incurs constant time execution overhead. Consequently, when the number of assertions is large and the lists are long (which is precisely when overheads matter), our technique gives close to constant time, amortized, for checking an assertion.

Evaluation: We implement both assertion checking techniques for specifications that express properties of lists, list-segments, and separation— the one that evaluates assertions on the concrete heap and the technique based on abstraction-based runtime checking. We evaluate both techniques on a suite of 25 programs, including both library code manipulating lists and client code calling these libraries. Our evaluation shows that cost of evaluating an assertion on the concrete heap grows with the size of the list, as expected, typically growing quadratically with the sizes of the lists. However, remarkably, checking an assertion on the abstraction performs orders of magnitude faster, and seems to essentially take only constant time in checking an assertion, when there are many assertions and when the sizes of the data structures are large.

Our evaluation shows that our new idea of using the synergy of abstractions and runtime state, where the runtime state helps keep the abstraction in check and where the abstraction helps in performing runtime assertion checking fast, is a powerful idea, and holds promise for building truly scalable assertion checking for dynamically manipulated data structures.

2 Assertion Logic

In this section we present the separation logic on lists that we consider and the class of assertions that we allow programmers to write.

We will first define program configurations, consisting of a store and a heap. Let *Loc* be a countably infinite set of heap locations and let `nil` be the special location term

$j \in$ *Scalar Int Variables* $\quad q \in$ *Scalar Bool Variables*
$x, y \in$ *Loc* Variables $\quad c \in$ *Int* Constant

$$
\begin{aligned}
\textit{Loc Terms: } lt &::= x \mid \texttt{nil} \\
\textit{Scalar Int Terms: } st &::= c \mid j \mid st + st \mid st - st \\
\textit{Scalar Formulas: } sf &::= q \mid st = st \mid st \neq st \mid st \leq st \mid st < st \mid lt = lt \mid lt \neq lt \\
\\
\text{Formulas: } \varphi &::= \texttt{true} \mid \texttt{false} \mid sf \\
&\quad\ \texttt{emp} \mid x \longmapsto lt \mid x \longmapsto ? \mid \texttt{list}(x) \mid \texttt{lseg}(x, y) \mid \varphi \wedge \varphi \mid \varphi * \varphi \\
\text{Assertions: } \alpha &::= \varphi * \texttt{true} \mid \alpha \vee \alpha
\end{aligned}
$$

Fig. 1. Syntax of a quantifier-free separation logic on lists and list segments

for the null pointer. Let us assume that each heap location has a single pointer field *next* and let $next_c : Loc \backslash \{\texttt{nil}\} \rightarrow Loc$ be that function which maps non-nil heap locations to the adjacent location in the concrete heap. Let *PV* be the set of program variables pointing to locations, *IV* be the set of variables of the type *Int* and *BV* be the set of variables of type *Boolean*. Let S_c be the concrete store that maps program variables to constants of the appropriate type– that is, S_c maps *PV* to *Loc*, *BV* to *true* or *false* and *IV* to integer constants. Let *PC* be the set of concrete program configurations where a configuration is a tuple of the heap $next_c$ and the store S_c.

The syntax of our assertion separation logic is shown in Figure 1. The semantics is fairly standard separation logic [25], and we skip giving a formal semantics. The formulas are evaluated over program configurations, where the store gives the valuation of scalar and pointer variables and a heaplet containing a subdomain of the dynamic heap with the pointer field *next*. Scalar formulas depend only on the store and not on the heaplet. The formula `emp` evaluates to true iff the heaplet is empty. $x \longmapsto lt$ evaluates to true iff the next-pointer from x points to lt and the heaplet is a singleton containing the location x points to. The formula $x \longmapsto ?$ evaluates to true iff x is a non-nil location and the heaplet is again a singleton set containing the location that variable x points to. `list(x)` evaluates to true iff x points to a list (wrt `next` pointer) ending with the `nil` location, and the heaplet is the set of all locations on this list. Similarly, `lseg(x,y)` evaluates to true iff the next-pointer from x reaches y eventually, and the heaplet is the locations on this segment that includes x and excludes y, unless $x = y$ when the heaplet is empty. Conjunction of two formulas hold on a heaplet iff both sub-formulas hold on that *same* heaplet. Finally, $\alpha * \beta$ holds iff the heaplet can be partitioned into two parts such that α holds in one and β holds in the other.

Assertions: While writing specifications for a program in separation logic, using assertions and pre/post-conditions, the pre-conditions implicitly delineate the part of the heap that the method should access. Conveying the footprint on which the program works implicitly through separation logic specifications has several advantages, the most important one being the *frame rule* [5] that one gets for free for static checking.

However, we do not want to insist on programmers to write pre-conditions to all methods in separation logic to delineate the fragment of the heap the method will change. It turns out that checking whether a method stays within the heaplet defined by its pre-condition is inherently expensive anyway to check at runtime; consequently, in the literature, there have been suggestions of delineating the heaplet using compiler and

Formula	Truthhood	Domain-exact	Scope
$true$	$true$	$false$	$\emptyset$
$false$	$false$	$false$	$\emptyset$
sf	$eval(sf)$	$false$	$\emptyset$
emp	$true$	$true$	$\emptyset$
$x \longmapsto lt$	$S_c(x) \neq nil \wedge next_c(S_c(x)) = S_c(lt)$	$true$	$\{S_c(x)\}$
$x \longmapsto ?$	$S_c(x) \neq nil$	$true$	$\{S_c(x)\}$
$list(x)$	$IsList(S_c(x))$	$true$	$between(S_c(x), nil)$
$lseg(x, y)$	$IsLseg(S_c(x), S_c(y))$	$true$	$between(S_c(x), S_c(y))$
$\varphi \wedge \varphi'$	$th(\varphi) \wedge th(\varphi') \wedge comp(\varphi, \varphi')$	dom-ext(φ) $\vee$ dom-ext(φ')	scope(φ) $\cup$ scope(φ')
$\varphi * \varphi'$	$th(\varphi) \wedge th(\varphi') \wedge$ scope(φ) $\cap$ scope(φ') $= \emptyset$	dom-ext(φ) $\wedge$ dom-ext(φ')	scope(φ) $\cup$ scope(φ')

where $comp(\varphi, \varphi')$:= (scope(φ) = scope(φ'), if dom-ext(φ) and dom-ext(φ')
:= ($scope(\varphi) \subseteq$ scope(φ')), if $\neg$dom-ext(φ) and dom-ext(φ')
:= ($scope(\varphi') \subseteq$ scope(φ)), if dom-ext(φ) and $\neg$dom-ext(φ')
:= $true$, otherwise.

Fig. 2. Truthhood, Domain-exactness and scope functions.

hardware-based isolation techniques [2]. In this paper, we do *not* consider this problem of a method staying within the confines of a heaplet. Rather, we assume that all assertions are disjunctions of formulas of the form $\varphi * true$ (see Figure 1), and evaluated on the store for the variables currently in scope and the global heap of the program. When preconditions to methods are lacking, this is the only reasonable way to evaluate assertions.

2.1 Evaluating assertions on the concrete heap

In general, when checking a formula of the kind $\alpha * \beta$ on a concrete heap, we need to find a way to divide the heaplet into two parts to evaluate α and β on. However, the separation logic we have has the property that heaplets for the atomic formulas (including `list` and `lseg`) are precisely determined. This allows us to evaluate any separation formula *bottom-up*, computing the relevant heaplets the formulas hold on, and hence is algorithmically efficient.

Following the work on Dryad [24,23], which is based on similar ideas, we evaluate a separation logic formula on a concrete heap bottom-up by computing a triple $\langle th(\beta), dom\text{-}exact(\beta), scope(\beta)\rangle$, for every subformula β of α. The boolean $th(\beta)$ captures the truth or falsehood of β. If $th(\beta)$ is true, then the boolean $dom\text{-}exact(\beta)$ captures whether the formula is true only on a particular heaplet. The set $scope(\beta)$ is a set of locations— and when $dom\text{-}exact(\beta)$ is true, it demands that the heaplet be precisely this scope, and when $dom\text{-}exact(\beta)$ is false, it demands that the heaplet is some *superset* of the scope. The functions $th(\beta)$, $dom\text{-}exact(\beta)$ and $scope(\beta)$ are defined in Figure 2. In the figure, the method *IsList(x)* checks whether location x points to a list along the *next* pointer field; similarly, *IsLseg(x,y)* checks if the traversal of the heap along *next* from x eventually reaches y. The method *between(x, y)* returns a set of locations depending on whether traversing *next* from x reaches y or not. If there exists a heap traversal from x to y, then *between(x, y)* returns the set of locations on the path including x and excluding y. When $x = y$ or when there is no traversal from x to y along the *next* field, *between(x, y)* is the empty set.

3 Runtime Guided Abstractions and Abstraction Guided Runtime Checking

In this section, we present our main contribution— an abstraction for lists that helps scale runtime checking of separation logic assertions. The base abstraction for lists that we define is itself straightforward, and is a mild adaptation of common abstraction of lists used in shape analysis [18]— the abstraction is a graph that tracks only the concrete nodes pointed to by program variables and the points where lists "merge" into one another, and also keeps track of precise memory addresses corresponding to these nodes. However, the salient aspect of our abstraction is that we keep the abstract graph *accurate* using knowledge acquired at runtime. This essentially means that for any assertion α in our logic, the evaluation of α on the abstract heap is the same as its evaluation on the concrete heap. The resulting scheme is an interesting *synergy* between the runtime state and the abstract state— the abstract state helps in runtime checking by providing it details that can be easily inferred using the abstraction (such as whether x points to a list, whether x to y forms a list segment, etc.), and the runtime state helps keep the abstract state in check, ensuring that it most accurately describes the current state of the system, which in turn is crucial in using it for runtime checks.

3.1 Abstracting Lists

The abstraction we use for lists is similar to those used in classical shape analysis literature [18,26], except that it stores some amount of information in terms of concrete pointers. In particular, we track only the concrete nodes that are currently pointed to by some pointer variables in the program and the concrete nodes that are the first nodes of the merging lists (i.e., the first nodes where the lists start to overlap). However, instead of using summary nodes that stand for unbounded sections of the lists, we have two kinds of edges in the abstraction. The first kind is a concrete edge— a *concrete edge* denotes that the source node of the edge points to the destination node through the $next_c$ pointer on the concrete heap. The second kind of edge is a summary edge— a *summary edge* from node u to v represents a list segment with length larger than one, stretching from u to v in the concrete heap.

We must emphasize that though the abstraction itself is fairly standard (and is described next), the abstract transition relation is not standard— in our setting, we can use the runtime state to build the next abstract state, and we exploit this to keep the abstraction accurate. Note that the program can do a lot of things that the abstraction is not intended to track (see below for an example). We need to keep the abstraction accurate no matter what the program does. We also emphasize that we are *not* verifying that the program satisfies an assertion (and hence getting rid of the assertion). Our abstraction is woefully inadequate in proving any such property of programs; the abstraction does, however, help in runtime checking.

Let us fix a finite set of program pointer variables PV, and the set of all memory locations (concrete heap locations) Loc.

Definition 1. *An* abstract heap of lists *over the set of pointer variables PV is a graph $G_a = (V, next_a, S_a, addr)$, where*

- *V is a finite set of nodes that has two special nodes v_{nil} and v_{undef},*
- *$next_a : V \setminus \{v_{nil}, v_{undef}\} \longrightarrow V \times \{c, s\}$ is a function that encodes the next-pointer, where a node maps to another node either through a concrete edge (c), or through a summary edge (s),*
- *$S_a : PV \rightarrow V$ associates each program pointer variable to the node it points to, and*
- *$addr : V \setminus \{v_{undef}\} \longrightarrow Loc$ is a map that associates each node with a location in the concrete heap.*

We also require the graph to satisfy the following:

- *for every program pointer variable $p \in PV$, there is precisely one node $v \in V$ such that $S_a(p) = v$,*
- *the address labels of the nodes are all different,*
- *for every $v \in V \setminus \{v_{nil}, v_{undef}\}$, either there exists $p \in PV$ such that $S_a(p) = v$ or there are at least two nodes u, u' such that $u \neq u'$ and $next_a(u) = next_a(u') = (v, e_t)$, where e_t can be either c or s.* □

In the above definition, v_{undef} is a special node in the abstract graph that represents all unallocated or uninitialized heap locations. We do *not* consider programs that result in undefined behavior on dereferencing an uninitialized heap location or manipulating ill-formed linked-list structures that can reach an uninitialized location.

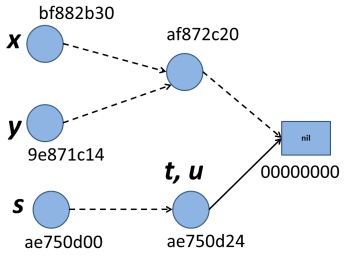

Fig. 3. Abstraction representing heap where x and y point to two lists that merge, but they are different from the list pointed to by s, and where s to t forms a list segment, and where $t = u$. Dashed lines represent the summary edges, while solid lines represent the concrete edges.

Figure 3 shows an abstraction of a heap that contains lists (we omit the node v_{undef} in the pictures and any pointer variable not depicted is assumed to point to v_{undef}). In the figure, the S_a and *addr* are depicted using labels on vertices. This abstract heap represents all heaps where (a) x and y both point to lists that merge, and merge first at address `af872c20`, (b) s to t forms a list segment and `t` → `next` points to the `nil` location, and the list from s is disjoint from the list from x and the list from y, (c) $u = t$, and (d) x, y, s and t are the pointers `bf882b30`, `9e871c14`, `ae750d00` and `ae750d24`, respectively.

In general, any abstract graph $G_a = (V, next_a, S_a, addr)$ depicts the set of concrete program configurations $(next_c, S_c) \in PC$ where (a) the program pointer variables map to the addresses stored at the abstract nodes representing them, i.e., $addr(S_a(pv)) = S_c(pv)$ for every pointer variable $pv \in PV$, (b) for every concrete edge in the abstract graph such that $next_a(u) = (v, c)$, the next pointer from the address at u points to the address at v, (c) for every summary edge (u, v) in the abstract graph such that $next_a(u) = (v, s)$, the next pointer from the address at u eventually leads to the address of v through a non-empty set of intermediate addresses, and (d) the concrete list segments represented by every edge are disjoint in the concrete heap. Let $\gamma : G_a \rightarrow 2^{PC}$ be the concretization

function that maps an abstract graph to the set of concrete heaplets corresponding to it, as described above.

It is easy to see that every concrete heap corresponds to precisely one abstract heap. Note that our abstraction does not allow expressing arbitrary sets of concrete heaps, and hence is not a standard abstraction used for static analysis. The set of concrete heaps that an abstract heap represents have the property that they all satisfy the *same* formulas in our assertion logic.

Note that though an assertion is checked only when the program reaches the assertion, keeping track of the abstract graph accurately demands instrumenting *every* heap operation the program does. However, the amount of help required to keep the abstraction in check is very small (close to constant time per instrumented operation of the program) while the reductions given by the abstraction to runtime checking is significant (often reducing from linear or quadratic checks on lists to close to constant time when the assertion is checked frequently).

3.2 Evaluating assertions on the abstract heap

We can evaluate our assertion logic on abstract heaps. Notice that our abstraction includes all nodes pointed to by program variables in scope, includes the reachability information between these nodes, and includes nodes where these list segments merge as well, which helps deciding separation properties. Consequently, we can evaluate an assertion bottom-up on an abstract heap (and the concrete store) similar to the evaluation of assertions on the concrete heap and store.

More formally, we define the scope of a formula in the abstract heap as a set of nodes and summary edges of the abstract graph. Intuitively, a summary edge (u, v) in the abstract graph represents all locations along the list segment from u to v. More specifically, the scope for *true*, *false* and the scalar formulas *sf* is the empty set; the scope for the formula $x \mapsto lt$ and $x \mapsto ?$ is the singleton set containing the node pointed to by variable x; the scope for $list(x)$ is the set of nodes and summary edges that occur along the path (if one exists) from the node pointed to by x to the node v_{nil}, otherwise the scope is an empty set if x does not point to a list; and similarly, the scope for $lseg(x, y)$ is the set of nodes and summary edges that occur along the path between the nodes pointed to by x and y. The scope for conjunction and separating conjunction of formulas is defined in the same way as in Figure 2. Provided the new definition of scope, an assertion formula α can be evaluated on the abstract heap and abstract store in the same way as its evaluation on the concrete heap (see Figure 2). Note that domain-exactness is a property of the formula and hence is independent of whether a formula is being evaluated in the abstract heap or in the concrete heap.

The salient property of our abstraction is that it is very precise for the assertion logic under consideration, and *yet* it can be maintained at this level of precision using the runtime concrete heap. More precisely, we can show the following:

Theorem 1. *Let A be an abstract heap of lists and let C be a concrete heaplet such that $C \in \gamma(A)$. Then, for any assertion φ, φ evaluates to true on the abstract heap iff φ evaluates to true on the concrete heap C.*

A proof gist is presented in the Appendix A. The above theorem gives us the ability to evaluate the assertion on the abstract heap instead of the concrete heap. When the runtime reaches a configuration with a concrete heap c, we can evaluate φ on it by evaluating φ on its abstraction a. For instance, if we come across an assertion of the form $list(x) * list(s) * true$ with the abstract graph of Figure 3, we can quickly answer that the assertion holds by examining the abstraction, while we can quickly also declare that the assertion $list(x) * list(y) * true$ is false.

Note that the above theorem does not hold for certain abstractions of the heap in the literature— for example, a naive shape analysis would maintain an abstraction that over-approximates any set of concrete configurations, and hence it may be the case that φ evaluates to false on an abstract state but true on a particular concrete configuration in the set of concrete configurations corresponding to the abstract state.

3.3 The synergy between abstraction and the runtime state

The goal of our runtime procedure is to maintain the abstract graph above. As we will show below, we can always maintain the abstract graph accurately using runtime checks, aided by the concrete addresses in the abstract graph. The naive way would be to compute the abstract graph from the concrete heap each time; this is, of course, too expensive— we will show how to maintain the graph with much less cost.

Let us now illustrate the way the runtime state helps keeping a check on the abstraction. Consider the scenario depicted in Figure 3, where s points to a list and t points somewhere on this list. Now, assume that the program apriori knows that there is a key k stored somewhere in the list segment from s to t (a recursive search for a key in a sorted list, like in quicksort, may result in such a situation). Consequently, assume that the program executed the following code:

```
while (s.key != k) { s:= s->next };
assert (lseg(s,t) * list(t) * true);
```

Note that the assertion after the while-loop is true, as s would not have passed the pointer t, since the key would have been found before that.

Now, if we just kept track of the abstraction (as in static shape analysis), we would have no idea when or whether the pointer s would go past t (since the abstraction cannot track every detail of the program, and in particular would not know whether the key k would occur before t). However, in our runtime guided abstraction, we *will* precisely know when s passes t. Every time we execute the statement $\texttt{s} := \texttt{s} \rightarrow \texttt{next}$, we will check whether the successive concrete address, namely `ae750d24`, has been reached by s; if not, the abstract graph would stay the same (except the address associated with s would be updated). If s does reach `ae750d24`, then the entire abstraction graph would change (with s, t, and u, all pointing to the same location).

Consequently, in the above setting where s stays in the list before t, the assertion will hold and will be validated to be true by just using the abstraction graph. In pure static verification, this shape graph would get divided into *two* graphs— one assuming s reached t and one that doesn't, and usually soon leads to very coarse abstractions of the heap. The runtime check using concrete addresses prevents this effectively and at low cost.

3.4 Maintaining the abstract state

In this section, we describe the instrumentation for capturing the abstraction transformation for various basic heap-manipulating statements in an imperative sequential programming language. Our programming language includes standard control-flow statements like `if` and `while`, dynamic memory manipulation, and function calls.

Figure 4 describes the operations we perform for maintaining the list abstraction. We only need to perform transformations on our abstract state whenever list manipulating statements occur. We rely on the type system of our programming language to detect which statements are manipulating lists. More formally, we instrument each list-manipulating statement and perform our abstract-state update after the statement execution and the concrete-state update. The abstract transformer, $[\![\_]\!]_a$, takes as input a statement, a concrete state produced by the statement execution, and an abstract state and it produces the updated abstract state. Note that $[\![\_]\!]_a$ inspects the updated concrete state only in the immediate neighborhood of the manipulated pointer (the `next` address of the manipulated pointer). The concrete transformer, $[\![\_]\!]_c$, just updates the concrete state based on the statement.

$$\begin{array}{rlll}
\texttt{x},\texttt{y} : \mathit{List}^* & x^{ctx}, y^{ctx} : \mathit{PV} & \mathit{loc} : \texttt{Loc} & \mathit{var} : \texttt{PV} \\
S_c : \texttt{PV} \to \texttt{Loc} & \mathit{next}_c : \texttt{Loc} \setminus \{\texttt{loc}_{\texttt{NULL}}\} \to \texttt{Loc} & & \\
S_a : \texttt{PV} \to \texttt{V} & \mathit{next}_a : \texttt{V} \setminus \{\texttt{v}_{\texttt{undef}}, \texttt{v}_{\texttt{nil}}\} \to \texttt{V} \times \{\texttt{c}, \texttt{s}\} & & \\
[\![_]\!]_{a\,_\,_} : \texttt{Stmt} \times \mathit{PC} \times G_a \to G_a & & [\![_]\!]_{c\,_} : \texttt{Stmt} \times \mathit{PC} \to \mathit{PC} &
\end{array}$$

$$\tau = ((\mathit{next}'_c, S'_c), (V, \mathit{next}_a, S_a, \mathit{addr})) \;\; \mathit{where} \;\; (\mathit{next}'_c, S'_c) = [\![\texttt{stmt}]\!]_c(\mathit{next}_c, S_c)$$

$$\begin{array}{rcl}
[\![\texttt{List}^*\ \texttt{x}]\!]_a(\tau) & = & (V, \mathit{next}_a, S'_a, \mathit{addr}) \; \mathit{where} \;\; S'_a = S_a[x^{ctx} \mapsto v_{\mathit{undef}}] \\
[\![\texttt{x} := \texttt{y}]\!]_a(\tau) & = & (V, \mathit{next}_a, S'_a, \mathit{addr}) \; \mathit{where} \;\; S'_a = S_a[x^{ctx} \mapsto S_a(y^{ctx})] \\
[\![\texttt{x} := \texttt{y} \to \texttt{next}]\!]_a(\tau) & = & (V', \mathit{next}'_a, S'_a, \mathit{addr}') \\
\multicolumn{3}{r}{\mathit{where} \quad \texttt{if}\ \mathit{next}_a(S_a(y^{ctx})) \;=\; (v_{y \to \mathit{next}}, c) :} \\
\multicolumn{3}{r}{S'_a = S_a[x^{ctx} \mapsto v_{y \to \mathit{next}}]} \\
\multicolumn{3}{r}{\mathit{next}'_a = \mathit{next}_a,\; V' = V,\; \mathit{addr}' = \mathit{addr}} \\
\multicolumn{3}{r}{\texttt{else}\ \; \mathit{let}\ v = \mathit{fresh_vertex}()} \\
\multicolumn{3}{r}{V' = V \cup \{v\},\; \mathit{addr}' = \mathit{addr}[v \mapsto S'_c(x^{ctx})]} \\
\multicolumn{3}{r}{\mathit{next}'_a = \mathit{next}_a[S_a(y^{ctx}) \mapsto (v, c)][v \mapsto (\mathit{next}_a(S_a(y^{ctx})), s)]} \\
\multicolumn{3}{r}{S'_a = S_a[x^{ctx} \mapsto v]} \\
[\![\texttt{x} \to \texttt{next} := \texttt{y}]\!]_a(\tau) & = & (V, \mathit{next}'_a, S_a, \mathit{addr}) \; \mathit{where} \;\; \mathit{next}'_a = \mathit{next}_a[S_a(x^{ctx}) \mapsto (S_a(y^{ctx}), c)] \\
[\![\texttt{x} := \texttt{malloc()}]\!]_a(\tau) & = & (V', \mathit{next}'_a, S'_a, \mathit{addr}') \; \mathit{where} \;\; \mathit{let}\ v = \mathit{fresh_vertex}() \\
\multicolumn{3}{r}{V' = V \cup \{v\},\; \mathit{addr}' = \mathit{addr}[v \mapsto S'_c(x^{ctx})]} \\
\multicolumn{3}{r}{\mathit{next}'_a = \mathit{next}_a[v \mapsto (v_{\mathit{undef}}, c)],\; S'_a = S_a[x^{ctx} \mapsto v]} \\
[\![\texttt{free(x)}]\!]_a(\tau) & = & (V', \mathit{next}'_a, S'_a, \mathit{addr}') \\
\multicolumn{3}{r}{\mathit{where} \;\; \forall\ pv.\, S_a(pv) = S_a(x^{ctx}) \Rightarrow S'_a = S_a[pv \mapsto v_{\mathit{undef}}]} \\
\multicolumn{3}{r}{\forall\ v \in \mathit{pred}(v, V, \mathit{next}_a, S_a, \mathit{addr}).\; \mathit{next}'_a = \mathit{next}_a[v \mapsto (v_{\mathit{undef}}, s)]} \\
\multicolumn{3}{r}{V' = V \setminus \{S_a(x)\},\; \mathit{addr}' = \mathit{addr}[S_a(x) \mapsto .]}
\end{array}$$

$$\begin{array}{l}
\mathit{clean}(V, \mathit{next}_a, S_a, \mathit{addr}) = \forall v. |\{\mathit{var} \mid S_a(\mathit{var}) \;=\; v\}| = 0 \wedge |\mathit{pred}(v, V, \mathit{next}_a, S_a, \mathit{addr})| \leq 1 \\
\qquad\qquad \Rightarrow \mathit{remove}(v, \mathit{next}_a)
\end{array}$$

Fig. 4. Abstract State Updates

Next we describe our notation in Figure 4. We use $f' = f[x \mapsto y]$ to denote that f' returns the same value as f for all arguments except x; for x it returns y. Further we use $f' = f[x \mapsto .]$ to denote the fact that f' is not defined on x. We use *pred* to get a set of vertices representing the predecessors of a node in the abstract graph. The *remove* function removes a node from the abstract graph, linking its successor and predecessor through a summary edge. Note that *clean* is always called after each abstract update, on the updated abstract graph; we omit explicitly calling *clean* in each update for brevity. The `refine_edges` function ensures that edges collapsed to summary edges after an abstract graph update are refined to a concrete edge when the source node address maps to the destination node address through the `next` pointer. The `refine_edges` function is called also at every point after the abstract update.

When a pointer variable $x \in PV$ is declared in the program, we update the abstract store S_a to map x to v_{undef}. Similarly, on a pointer assignment `x := y`, the abstract store is updated to map variable x to the node pointed to by variable y. The instrumentation for statement `x := y→next` is more involved. We distinguish two cases. If the abstract node corresponding to the new value of x is already in the abstract graph we just update the S_a such that x now points to the corresponding abstract node. If not, a new node is created, the *addr* map is updated to map this node to the concrete address of x. The newly created node is introduced after the abstract node corresponding to y by linking it through a concrete incoming edge and an outgoing summary edge. Note that *refine_edges* will check whether the summary edge needs to be changed to a concrete edge. The statement `x→next := y` does not change the store nor the address map in the abstraction, but it only modifies the shape of the abstract graph. In particular, the instrumentation code updates the edge function $next_a$ to now have a concrete edge from x to y. Abstraction for the *malloc* statement involves adding a fresh node v to the abstract graph, updating the abstract store to point x to v, $addr(v)$ is the location pointed to by x after the *malloc* and adding a concrete edge in the abstraction from v to v_{undef}. Abstraction for the $free(x)$ statement involves removing the node v in the abstract graph corresponding to variable x, updating the $next_a$ edges to join all predecessors of v to v_{undef}, and updating the store to map all variables pointing to v to point to v_{undef}.

The transformations in Fig. 4 extend to function calls. Each variable is scoped using ctx, which represents the dynamic context based on the call-stack. The scoping enables us to uniquely identify variables in our abstraction, across (potentially recursive) function calls. First, at each call-site, we introduce a mapping in our S_a from the formal function parameters to the actual arguments at the call-site. Next, at returns, we remove from S_a the variables in the current scope.

Theorem 2. *For any statement s, an abstract state A and concrete program configuration C such that $C \in \gamma(A)$, if C' is the concrete program configuration obtained after executing s in configuration C, i.e., $C' = [\![\mathtt{s}]\!]_c(C)$, and A' is the abstract state obtained after executing the instrumented code in configuration C' and abstract state A, i.e., $A' = [\![\mathtt{s}]\!]_a(C', A)$, then $C' \in \gamma(A')$.*

Using Theorem 1 and Theorem 2 we can show that for every assertion φ in our logic, φ evaluates to true on the updated concrete state c' if and only if φ evaluates to true on the updated abstract state a'.

4 Evaluation

In this section we validate the claim that using abstractions for runtime checking results in faster assertion checking than just standard checking on the concrete heap.

Our benchmarks consists of (a) programs manipulating singly-linked lists obtained from the C GLib library [11], such as concatenation of two lists, insertion into and deletion from a list, reversal of a list, etc., and (b) programs that use linked lists as a library, such as stack and queue implementations, LRU-cache implementations, etc. Our LRU implementation is based on the the Least Recently Used page replacement algorithm, used for memory page management. We based our implementation on the algorithm description in Bovet et al. [6]. The first column of Table 1 lists the names of the benchmark programs we use in our evaluation; the names are self-descriptive. The list programs are fairly concise, ranging in size from 15 lines to 50 lines. Our programs that use lists as library are moderately concise, with up to 200 lines of code. The programs vary in complexity, some executing in constant time, e.g., prepend, and others being linear time.

Node* *insert_before*(**Node*** `slist`, **Node*** `sibling`, **int** `data`)
`requires` *lseg*(`slist`, `sibling`) * *list*(`sibling`)
`invariant` *list*(`slist`) ∧ (*lseg*(`slist`, `last`) *
(`last` ⟼ `node`) * *lseg*(`node`, `sibling`) * *true*)
`ensures` *list*(`return`)

Fig. 5. The contract and assertions for insert_before

We annotated GLib subjects with assertions for preconditions, postconditions, and loop invariants; recall that we do not check that the function stays within the heaplet. Figure 5 shows an example of an annotated program in our framework. It shows the precondition, loop invariant, and postcondition, for a program that inserts a new node with key **data** before the node **sibling**. `last` is the pointer used to iterate over the list and it is the last node before sibling when the loop stops. Our assertions check structural correctness using the *list* and *lseg* (for list segments) recursive predicates.

Note that the assertions in the GLib programs occur very often, within the loops/recursive calls. Consequently, the assertions are checked a large number of times, typically linear in the length of the lists manipulated. Some of the clients, such as the queue implementations do not have loop invariants, having only preconditions or postconditions, and hence the assertions occur less frequently in those subjects.

We performed all our experiments on an Intel Core i7 with 32 GB of RAM. We vary the workload in our programs by increasing the input list sizes (lists' sizes range from 10 to 16384 nodes). We report the average running time for each assertion check, obtained over 100 repeating runs, to account for noise in measurement, for the increasing list sizes.

We present the results of our evaluation in Table 1. The various columns denote the length of the list input to the program. The first column lists the names of the programs we used, and the second column indicates the number of assertion checks that the program performs for a list of length n. Each 2-column group shows, for lists of varying sizes ranging from 10 to 16384, the average time for checking a *single* assertion in

⇒ List Length		10		2048		4096		16384	
⇓ Program	#Asrt	Conc	Abs	Conc	Abs	Conc	Abs	Conc	Abs
Library									
append	$O(n)$	6	6	10368	8.00	t.o.	8.78	t.o.	9.45
concat	$O(n)$	5	5	10528	7.79	t.o.	8.51	t.o.	9.97
copy	$O(n)$	5	4	11840	4.98	t.o.	5.55	t.o.	6.70
find	$O(n)$	4	3	10344	2.67	t.o.	2.19	t.o.	2.68
free	$O(n)$	0	1	3.10	1.12	6.33	1.06	27	1.07
insert	$O(n)$	3	5	1562	7.57	t.o.	8.48	t.o.	9.79
last	$O(n)$	5	6	10343	7.08	t.o.	7.64	t.o.	9.14
reverse	$O(n)$	3	3	t.o.	3.42	t.o.	3.52	t.o.	4.01
remove-link	$O(n)$	4	5	t.o.	7.28	t.o.	8.25	t.o.	9.61
remove-all	$O(n)$	4	4	t.o.	7.63	t.o.	8.01	t.o.	9.59
position	$O(n)$	6	4	t.o.	4.17	t.o.	4.07	t.o.	4.14
nth-data	$O(n)$	3	3	t.o.	2.81	t.o.	2.73	t.o.	2.73
nth	$O(n)$	3	3	t.o.	2.75	t.o.	2.69	t.o.	2.74
length	$O(n)$	4	4	t.o.	3.24	t.o.	2.75	t.o.	3.16
insert-at-pos	$O(n)$	3	3	t.o.	5.00	t.o.	5.47	t.o.	6.36
index	$O(n)$	2	2	18	0.01	35	0.00	141	0.00
prepend	$O(1)$	0	1	0.01	0.00	0.01	0.00	0.01	0.00
create	$O(n)$	0	1	3.09	1.33	6.33	1.34	27.22	1.34
swap	$O(1)$	0	0	0.01	0.00	0.01	0.00	0.01	0.00
Client									
split	$O(n)$	2	2	t.o.	1.35	t.o.	1.33	t.o.	1.45
merge	$O(n)$	7	6	t.o.	5.17	t.o.	5.10	t.o.	5.00
reverse-sublist	$O(n)$	1	1	t.o.	1.13	t.o.	1.13	t.o.	1.12
insert-sorted	$O(n)$	1	1	4.52	0.00	8.85	0.00	35	0.00
queue	$O(1)$	0	0	87.95	3.39	171	3.48	695	5.08
stack-queue	$O(1)$	0	0	48.82	4.63	82.27	5.37	286	11.37
LRU	$O(n)$	9	2	94.97	2.11	162	2.29	574	2.24

Table 1. Comparison of average running time per assertion check using concrete checks versus leveraging abstraction. Times are shown in μs. Timeout (t.o.): 40min for checking all assertions

the program with *runtime checking on the concrete state* (**Conc**) and *runtime checking using the abstract heap* (**Abs**).

The average time taken to check a single assertion is calculated, both for the concrete checking and the abstraction-guided checking, by calculating the time taken by the program with assertions, subtracting the time taken by the program without assertions, and then dividing by the total number of assertions checked. When the program with assertions takes too long, exceeding 40 minutes, we denote a timeout (t.o.).

The runtime checking of the assertions on the concrete heaplet (Conc) grows with the size of the list, and takes typically linear to quadratic time to check the property. This is reasonable and acceptable for smaller lists (say a few hundred), but gets prohibitive for larger lists, timing out on larger lists. We emphasize that even with a *manual encoding by the programmer*, the check will typically take this time— checking properties of lists should, after all, take longer when the lists are longer.

However, the time taken per runtime assertion aided by the abstract heap is almost *constant*, varying very little with the length of the input! In lists ranging to 16K, this shows 20x to 3000x speedup in checking the assertions, in comparison with the check-

ing on the concrete heap. Consequently, runtime checking using the abstract heap scales with acceptable overheads even for our largest input sizes.

In the `stack-queue` client the number of checks is smaller because we do not require the loop invariant for the properties we check. The smaller number of checks is reflected as an increase in the assertion checking time.

Intuitively, the size of the abstract heaplet stays constant, and hence checking an assertion on the abstract heaplet can be done in constant time. However, the abstract heaplet requires maintenance, but this is typically a constant amount of work for each heap manipulation. When the number of assertions are high and the lengths of the lists are large, the maintenance cost of the heaplet is not significant, and we obtain essentially a constant amount of cost for checking an assertion, independent of the length of the list.

Our experiments clearly show that the use of runtime-guided abstractions can make runtime checking of data-structure properties much faster, when the assertions are checked often and the sizes of the data structure are large.

We believe that using these in settings where assertions abound, such as in *class invariants*, where invariants are checked every time the data structure is accessed, can benefit greatly by our approach. However, when assertions are sparse, the cost of maintaining the abstract state may get expensive, and it may be more prudent to check the assertion on the concrete heap. An automatic hybrid approach that exercises these choices to instrument large programs is an interesting future direction.

5 Related Work

Runtime assertion checking has been successfully used in software engineering and programming language design (e.g., [8]). Debugging using assertions expressed as Boolean formulas is a routine software development practice [12]. However, there are relatively few approaches that can be used for checking properties of programs manipulating structurally complex data. A common technique of specification for that class of program are *representation invariants* (i.e., RepOk [16]). Implementing representation invariants can be hard to get right, also it imposes a significant burden on the developer [17,7]. Jump et al. [13] introduce dynamic shape analysis and check structural properties of the heap. Crane and Dingel [9] present a declarative language for specifying object models using the *Alloy* language, and perform runtime checks to ensure that certain user specified locations conform to an object model. However, runtime checking of these properties incurs large overheads.

Some recent approaches (e.g., [21,2]) propose using runtime checking assertions written in separation logic. Separation logic has been successfully used as a specification language in deductive verification of programs manipulating complex data structures, making it an attractive choice for runtime assertion checking. However, as noted by Nguyen et al. [21], runtime checks of separation logic assertions can be challenging due to implicit footprint and existential quantification. A main focus of the work described by Nguyen et al. [21] is alleviating potentially exponential blow-up of sets of locations that needs to be considered when splitting the heap in two parts when evaluating separation connective. They use a marking technique to limit the set of footprints

that needs to be explored when evaluating the formula. Their approach works well when checking only preconditions and postconditions of data structures at the boundary between statically verified and unverified code, however performing multiple checks often incurs prohibitively large overheads. In contrast, we focus on choosing a separation logic for lists, suitable for runtime assertion checking, and developing a technique that will allow checking properties in near constant time using abstractions. Our technique works well both in cases where assertions are at the boundary and also when they are intensively checked throughout the program.

Agten et al. [2] devised another technique for run-time checks of separation logic annotations. This approach combines deductive verification with run-time checks of the unverified parts of the code to provide stronger run-time guarantees for the verified parts. Run-time checks are introduced at the boundary between verified and un-verified parts. A key difference comparing to our approach is that Agten et al. [2] the assertions are meant to be checked sparsely (only when crossing the verified-unverified boundary), while our approach excels even when assertions are checked frequently.

A technique proposed by Shankar and Bodik [27] reduces the run-time overhead through incremental assertion checking. The technique devises automatic memoization for a class of side-effect-free representation invariants. However, it is developer's responsibility to provide correct representation invariants. Koukoutos and Kuncak [14] also focus on reducing run-time overhead induced by dynamic checks of complex program properties. They tackle the run-rime overhead using memoization techniques, restricting the domain to purely functional Scala programs.

Our abstraction is inspired by a common abstraction used in shape analysis [18]. However, comparing to static shape analysis approaches, we exploit runtime information to maintain accurate abstract graph, which we use to reduce overhead of runtime checks.

Vechev et al. [29] present PHALANX, a tool that uses parallelism to speed up assertion checking. PHALANX evaluates the assertions in a different thread, on a snapshot of the entire state at the assertion. Similarly, Atandilian et al. [1] introduce asynchronous assertions, which can be used during debugging. Our work shares the goal of speeding up assertion checking, but we use abstractions to perform the assertions in constant time; these techniques are complementary to our work.

6 Conclusions

Our main contribution is an abstraction-guided runtime checking for linked lists, where we have shown how an abstraction maintained using the runtime state helps in efficient runtime assertion checking, often close to constant time per assertion.

We would like to, in the future, explore abstraction-guided runtime checking for more rich data structures (such as doubly-linked lists, trees, etc.) and for recursively defined functions (not just predicates) on these structures (such as the length of lists, heights of trees, set of keys stored in the structure, etc.). It is presently unclear how to build such abstraction guidance effectively. Furthermore, we believe that in larger program contexts, abstraction-guidance should be used for often checked assertions (like class invariants) while the concrete structure is checked for sparse assertions; building an effective hybrid scheme would be interesting.

References

1. E. E. Aftandilian, S. Z. Guyer, M. Vechev, and E. Yahav. Asynchronous assertions. In *OOPSLA '11*, pages 275–288, 2011.
2. P. Agten, B. Jacobs, and F. Piessens. Sound modular verification of C code executing in an unverified context. In *POPL'15*, 2015.
3. M. Barnett, M. Fahndrich, and F. Logozzo. Embedded contract languages. In *ACM SAC - OOPS*. Association for Computing Machinery, Inc., March 2010.
4. M. Barnett, K. R. M. Leino, and W. Schulte. The spec# programming system: An overview. In *CASSIS'04*, pages 49–69, 2005.
5. A. Borgida, J. Mylopoulos, and R. Reiter. On the frame problem in procedure specifications. *Software Engineering, IEEE Transactions on*, 21(10):785–798, Oct 1995.
6. D. Bovet and M. Cesati. *Understanding The Linux Kernel*. Oreilly & Associates Inc, 2005.
7. C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated testing based on java predicates. In *ISSTA'02*, pages 123–133, 2002.
8. L. A. Clarke and D. S. Rosenblum. A historical perspective on runtime assertion checking in software development. *SIGSOFT Softw. Eng. Notes*, pages 25–37, 2006.
9. M. L. Crane and J. Dingel. Runtime conformance checking of objects using Alloy. In *Electronic Notes in Theoretical Computer Science*, volume 89, pages 2–21. Elsevier, 2003.
10. B. Elkarablieh, Y. Zayour, and S. Khurshid. Efficiently generating structurally complex inputs with thousands of objects. In *ECOOP'07*, pages 248–272, 2007.
11. `https://developer.gnome.org/glib/`, 2015.
12. C. A. R. Hoare. Assertions: A personal perspective. *IEEE Ann. Hist. Comput.*, pages 14–25, 2003.
13. M. Jump and K. S. McKinley. Dynamic shape analysis via degree metrics. In *ISMM '09*, pages 119–128, 2009.
14. E. Koukoutos and V. Kuncak. Checking data structure properties orders of magnitude faster. In *RV'14*, pages 263–268, 2014.
15. G. Leavens, A. Baker, and C. Ruby. JML: A notation for detailed design. In *Behavioral Specifications of Businesses and Systems*, volume 523, pages 175–188. Springer US, 1999.
16. B. Liskov and J. Guttag. *Program Development in Java: Abstraction, Specification, and Object-Oriented Design*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 2000.
17. M. Z. Malik, A. Pervaiz, E. Uzuncaova, and S. Khurshid. Deryaft: A tool for generating representation invariants of structurally complex data. In *ICSE '08*, pages 859–862, 2008.
18. R. Manevich, E. Yahav, G. Ramalingam, and S. Sagiv. Predicate abstraction and canonical abstraction for singly-linked lists. In *VMCAI'05*, pages 181–198, 2005.
19. B. Meyer. *Eiffel: The Language*. Prentice-Hall, Inc., 1992.
20. G. J. Myers and C. Sandler. *The Art of Software Testing*. John Wiley & Sons, 2004.
21. H. H. Nguyen, V. Kuncak, and W.-N. Chin. Runtime checking for separation logic. In *VMCAI'08*, pages 203–217, 2008.
22. P. W. O'Hearn, J. C. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *CSL '01*, pages 1–19, 2001.
23. E. Pek, X. Qiu, and P. Madhusudan. Natural proofs for data structure manipulation in C using separation logic. In *PLDI '14*, pages 440–451, 2014.
24. X. Qiu, P. Garg, A. Ştefănescu, and P. Madhusudan. Natural proofs for structure, data, and separation. In *PLDI '13*, pages 231–242, 2013.
25. J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, LICS '02, 2002.

26. M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Trans. Program. Lang. Syst.*, 24(3):217–298, 2002.
27. A. Shankar and R. Bodík. Ditto: Automatic incrementalization of data structure invariant checks (in Java). In *PLDI '07*, pages 310–319, 2007.
28. N. Tillmann and W. Schulte. Parameterized unit tests. In *ESEC/FSE-13*, pages 253–262, 2005.
29. M. Vechev, E. Yahav, and G. Yorsh. Phalanx: Parallel checking of expressive heap assertions. In *ISMM '10*, pages 41–50, 2010.

A Theorem 1 (Proof Sketch)

Lemma 1. *For a formula α*
if α is domain exact then
$\forall\, H \subseteq G.\ H \models \alpha \quad \Rightarrow \quad H = scope(G, \alpha)$
else if α is not domain exact then
$\forall\, H \subseteq G.\ H \models \alpha \quad \Rightarrow \quad H \supseteq scope(G, \alpha) \wedge \forall H' \supseteq scope(G, \alpha).H' \models \alpha$

Proof. The proof is straightforward and it follows by induction on the structure of α. □

Definition 1 **($rep_{A,C}$).** *Let $rep_{A,C} : V \cup next_a \rightarrow 2^{LOC} \setminus \{\emptyset\}$ be a function denoting a mapping from nodes and edges in the abstract graph to the corresponding concete heap locations.*

Let the map $rep_{A,C}$ be extended to sets of nodes and edges in the abstract graph in the natual way. For a set S of nodes and edges, $rep_{A,C}(S)$ is the union of $rep_{A,C}(s)$ for every $s \in S$.

Lemma 2. *Let G_a, G_c be an abstract and concrete heap (resp.), α a formula, and $scope(G_a, \alpha)$ the scope of the formula α in the abstract heap G_a, $scope(G_c, \alpha)$ the scope of the formula α in the concrete heap G_c. The following holds:*

$$rep_{A,C}(scope(G_a, \alpha)) = scope(G_c, \alpha)$$

Proof. The proof is a straightforward induction on the structure of α and follows from Definition 1. □

Lemma 3. *Let x and y be abstract heaplets, then the following holds:*

$$rep_{A,C}(x) \cap rep_{A,C}(y) \neq \emptyset \quad \textit{iff} \quad x \cap y \neq \emptyset$$

Proof. The proof follows from the definition of $rep_{A,C}$. □

Corollary 1. *Given abstract heap G_a and concrete heap G_c s.t. $G_c \in \gamma(G_a)$, and formulas α, β:*

$$scope(G_a, \alpha) \subseteq scope(G_a, \beta) \quad \textit{iff} \quad scope(G_c, \alpha) \subseteq scope(G_c, \beta)$$

.

Proof. The proof simply follows from Defintion 1 and Lemma 2. □

Corollary 2. *Given abstract heap G_a and concrete heap G_c s.t. $G_c \in \gamma(G_a)$, and formulas α, β:*

$$scope(G_a, \alpha) \cap scope(G_a, \beta) = \emptyset \quad \textit{iff} \quad scope(G_c, \alpha) \cap scope(G_c, \beta) = \emptyset$$

Proof. The proof follows simply from Lemma 2 and Lemma 3. □

Lemma 4. *For an abstract store S_a, abstract heap G_a, concrete store S_c, concrete heap G_c, and any formula α:*

$$S_c, scope(G_c, \alpha) \models \alpha \quad \textit{iff} \quad S_a, scope(G_a, \alpha) \models \alpha$$

Proof. The proof is by induction on the structure of α using Lemmas 1, 2, 3 and Corollaries 1, 2. □

Next we prove Theorem 1. Note that the following is a slight reformulation of the theorem presented in Section 3.2.

Theorem 1. *For a concrete heap G_c, abstract heap G_a, s.t. $G_c \in \gamma(G_a)$ and a formula α the following holds:*

$$G_c \models \alpha * true \quad \textit{iff} \quad G_a \models \alpha * true$$

Proof. Assume $G_c \models \alpha * true$, then there exists precisely one concrete heaplet (with the corresponding concrete store) on which it holds, namely its scope: $S_c, scope(G_c, \alpha) \models \alpha$. By Lemma 4 it follows that $S_a, scope(G_a, \alpha) \models \alpha$. Assume now that $G_a \not\models \alpha * true$, implying that there does not exist an abstract heaplet on which $\alpha * true$ holds, namely it does not hold on its scope with store S_a – a contradiction.

Assume $G_c \not\models \alpha * true$, then there does not exist a concrete heaplet (with the corresponding store) on which the formula holds: $S_c, scope(G_c, \alpha) \not\models \alpha$. Hence, by Lemma 4 it follows that $S_a, scope(G_a, \alpha) \not\models \alpha$, that is, there does not exist an abstract heaplet on which α holds. Assume $G_a \models \alpha * true$, which means there is precisely one abstract heaplet (and the corresponding abstract store) on which the formula holds: $S_a, scope(G_a, \alpha) \models \alpha$ – a contradiction. □