Formal Verification of Skiplists with Arbitrary Many Levels[⋆]

Alejandro Sánchez[1] and César Sánchez[1,2]

[1] IMDEA Software Institute, Madrid, Spain
[2] Institute for Information Security, CSIC, Spain

Abstract. We present an effective method for the formal verification of skiplists, including skiplists with arbitrary length and unbounded size. The core of the method is a novel theory of skiplists with a decidable satisfiability problem, which up to now has been an open problem.
A skiplist is an imperative software data structure used to implement a set by maintaining several ordered singly-linked lists in memory. Skiplists are widely used in practice because they are simpler to implement than balanced trees and offer a comparable performance. To accomplish this efficiency most implementations dynamically increment the number of levels as more elements are inserted. Skiplists are difficult to reason about automatically because of the sharing between the different layers. Furthermore, dynamic height poses the extra challenge of dealing with arbitrarily many levels. Our theory allows to express the memory layout of a skiplist of arbitrary height, and has an efficient decision procedure. Using an implementation of our decision procedure, we formally verify shape preservation and a functional specification of two source code implementations of the skiplist datatype.
We also illustrate how our decision procedure can also improve the efficiency of the verification of skiplists with bounded levels. We show empirically that a decision procedure for bounded levels does not scale beyond 3 levels, while our decision procedure terminates quickly for any number of levels.

1 Introduction

A skiplist [13] is a data structure that implements a set, maintaining several sorted singly-linked lists in memory. Each node in a skiplist stores a value and at least the pointer corresponding to the list at the lowest level, called the *backbone list*. Some nodes also contain pointers at higher levels, pointing to the next node present at that level. The skiplist property establishes that: (a) the backbone list is ordered; (b) lists at all levels begin and terminate on special *sentinel* nodes called *head* and *tail* respectively; (c) *tail* points to *null* at all levels; (d) the list at level $i+1$ is a sublist of the list at level i. Search in skiplists is probabilistically logarithmic.

[⋆] This work was funded in part by Spanish MINECO Project “TIN2012-39391-C04-01 STRONGSOFT”

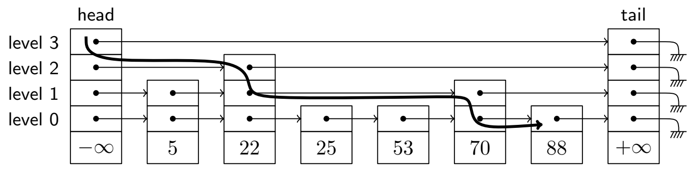

Fig. 1. A skiplist with 4 levels and the traversal searching 88 (heavy arrow).

Consider the skiplist layout in Fig. 1. Higher-level pointers allow to *skip* many elements during the search. A search is performed from left to right in a top down fashion, progressing as much as possible in a level before descending. For instance, in Fig. 1 a lookup for value 88 starts at level 3 of node *head*. The successor of *head* at level 3 is *tail*, which stores value $+\infty$, greater than 88. Consequently, the search continues at *head* by moving down to level 2. The expected logarithmic search follows from the probability of a node being present at a certain level decreasing by $1/2$ as the level increases.

Contributions Most practical implementations of skiplists either can grow dynamically to any height or limit the maximum height of any node to a large value like 32. Both kinds of implementations use a variable to store the current highest level in use. In this paper we introduce TSL, a theory that captures skiplist memory layouts, and we show that the (quantifier-free) satisfiability problem for TSL is decidable, which has been up to now an open problem. This theory builds non-trivially from the family of theories TSL_K (see [17]) that allow to reason about skiplists of a *bounded* number of levels. TSL is a decidable theory which solves the following two *open problems*: (a) verification of skiplist implementations with an unbounded/growing number of levels; and (b) verification of skiplist implementations for any bounded number of levels, even beyond the practical limitation of 3 levels suffered by current verification techniques. With our implementation of the decision procedure for TSL, we verify two implementations of the skiplist datatype: one, part of the industrial open source project KDE [1,2], and a full implementation developed internally. In this paper we also show, empirically, that TSL_K does not scale beyond $K = 3$ levels but TSL allows to verify skiplist implementations of arbitrarily many levels (bounded by some value like 32 or not).

Related Work Reasoning about skiplists requires to deal with unbounded mutable data stored in the heap. One popular approach to the verification of heap programs is Separation Logic [15]. Skiplists, however, are problematic for separation-like approaches due to the aliasing and memory sharing between nodes at different levels. Most of the work in formal verification of pointer programs are based on program logics in the Hoare tradition enriched to deal with the heap and pointer structures [4, 9, 20]. Our approach is complementary, consisting of the design of specialized decision procedures for memory layouts which can be incorporated into a reasoning system for proving temporal properties, in the style of Manna-Pnueli [10]. Proofs (of both safety and liveness properties) are ultimately decomposed into verification conditions (VCs) in the underlying theory used for state assertions. This paper studies the automatic verification of VCs involving the manipulation of skiplist memory layouts. For illustration purposes we restrict the presentation in this paper to safety properties.

Logics like [4,9,20] are very powerful to describe pointer structures, but they require the use of quantifiers to reach their expressive power. Hence, these logics preclude their combination with methods like Nelson-Oppen [12] or BAPA [8] for other aspects of the program state. Other alternatives based on shape analysis [19], like forest automata [3,7] can only handle skiplists only of a bounded height (empirical evaluation also suggest a current limit of 3). Unlike [7] our approach is not fully automatic in the sense that it requires some user provided annotations. On the other hand, our approach can handle skiplists of arbitrary and growing height. The burden of additional annotation can be alleviated with methods like invariant generation, but this is out of the scope of this paper.

Instead, we borrow from [14] a model-theoretic technique to deal with reachability to define the theory TSL and build its decision procedure. Our solution uses specific theories of memory layouts [16,17] that allow to express powerful properties in the quantifier-free fragment through the use of built-in predicates. For example, in [17] we presented a family of theories of skiplists of fixed height, based on a theory of ordered singly-linked lists [16]. However, handling dynamic height was still an open problem that precluded the verification of practical skiplist implementations. We solve this open problem here.

The rest of the paper is structured as follows. Section 2 presents a running example of two implementations of the skiplists datatype. Section 3 introduces TSL. Section 4 contains the decidability proof. Section 5 provides some examples of the use of TSL in the verification of skiplists. Finally, Section 6 concludes the paper. Some proofs are missing due to space limitation and are included in the appendix.

2 An Implementation of Skiplists

Fig. 2 shows the pseudo-code of a sequential implementation of a skiplist, whose basic classes are *Node* and *SkipList*. Each *Node* object contains a *key* field for keeping the list ordered, a field *val* for the actual value stored, and a field *next*: an array of arbitrary length containing the addresses of the following nodes at each level. The program in Fig. 2 implements an unbounded skiplist. The local variables *lvl* in Insert, *i* in Search and *removeFrom* in Remove maintain the maximum level that these algorithms should consider.

An object *sl* of class *SkipList*, maintains fields *sl.head*, *sl.tail* and *sl.maxLevel* to keep the data members storing the addresses of the head and tail sentinel nodes, and the maximum level in use (resp). When the *SkipList* object *sl* is clear from the context, we use *head*, *tail* and *maxLevel* instead of *sl.head*, *sl.tail* and *sl.maxLevel*. The *head* node has $key = -\infty$ and *tail* has $key = +\infty$. These nodes are not removed during the execution and their *key* field is not modified.

Finally, *Node* objects also maintain a "ghost field" *level* for the highest relevant level of *next*. *SkipList* objects maintain two ghost fields: *reg* for the region

```
class  $Node$      {  $Value\ val$;  $Key\ key$;  $Array\langle Node^*\rangle\ next$;  $Int\ @level$;  }
class  $SkipList$  {  $Node^*\ head$;  $Node^*\ tail$;  $Int\ @maxLevel$;  $Set\langle Addr\rangle\ @reg$;  $Set\langle Value\rangle\ @elems$;  }
```

```
procedure MGC($SkipList\ sl$)
    while $true$ do
        $v := NondetPickValue$
        nondet
            call INSERT($sl, v$)
                or
            call SEARCH($sl, v$)
                or
            call REMOVE($sl, v$)
    end while
end procedure

procedure INSERT($SkipList\ sl,\ Value\ v$)
    $Array\langle Node^*\rangle\ [sl.maxLevel + 1] upd$
    $Bool\ valueWasIn := false$
    $Int\ lvl := randomLevel$
    if $lvl > sl.maxLevel$ then
        $i := sl.maxLevel + 1$
        while $i \leq lvl$ do
            $sl.head.next[i] := sl.tail$
            $sl.tail.next[i] := null$
            $sl.maxLevel := i$
            $i := i + 1$
        end while
    end if
    $Node^* pred := sl.head$
    $Node^* curr := pred.next[sl.maxLevel]$
    $Int\ i := sl.maxLevel$
    while $0 \leq i \wedge \neg valueWasIn$ do
        $curr := pred.next[i]$
        while $curr.val < v$ do
            $pred := curr$
            $curr := pred.next[i]$
        end while
        $upd[i] := pred$
        $i := i - 1$
        $valueWasIn := (curr.val = v)$
    end while
    if $\neg valueWasIn$ then
        $x := CreateNode(lvl, v)$
        $i := 0$
        while $i \leq lvl$ do
            $x.next[i] := upd[i].next[i]$
            $upd[i].next[i] := x$
            if $i = 0$ then
                $sl.reg := sl.reg \cup \{x\}$
                $sl.elems := sl.elems \cup \{v\}$
            $i := i + 1$
        end while
    end if
    return $\neg valueWasIn$
end procedure

procedure SEARCH($SkipList\ sl,\ Value\ v$)
    $Node^*\ pred := sl.head$
    $Node^*\ curr := pred.next[maxLevel]$
    $Int\ i := sl.maxLevel$
    while $0 \leq i$ do
        $curr := pred.next[i]$
        while $curr.val < v$ do
            $pred := curr$
            $curr := pred.next[i]$
        end while
        $i := i - 1$
    end while
    return $curr.val = v$
end procedure

procedure REMOVE($SkipList\ sl,\ Value\ v$)
    $Array\langle Node^*\rangle[sl.maxLevel + 1]\ upd$
    $Int\ removeFrom := sl.maxLevel$
    $Node^*\ pred := sl.head$
    $Node^*\ curr := pred.next[sl.maxLevel]$
    $Int\ i := sl.maxLevel$
    while $i \geq 0$ do
        $curr := pred.next[i]$
        while $curr.val < v$ do
            $pred := curr$
            $curr := pred.next[i]$
        end while
        if $curr.val \neq v$ then
            $removeFrom := i - 1$
        end if
        $upd[i] := pred$
        $i := i - 1$
    end while
    $Bool\ valueWasIn := (curr.val = v)$
    if $valueWasIn$ then
        $i := removeFrom$
        while $i \geq 0$ do
            $upd[i].next[i] := curr.next[i]$
            if $i = 0$ then
                $sl.reg := sl.reg \setminus \{curr\}$
                $sl.elems := sl.elems \setminus \{v\}$
            $i := i - 1$
        end while
        free $(curr)$
    end if
    return $valueWasIn$
end procedure
```

Fig. 2. On top, the classes *Node* and *SkipList*. Below, the most general client MGC, and the procedures INSERT, SEARCH and REMOVE.

(set of addresses) managed by the skiplist and *elems* for the set of values stored in the skiplist. We use the @ symbol to denote a ghost field, and boxes (see Fig. 2) to describe "ghost code". These small extra ghost annotations are only added for verification purposes and do not influence the execution of the real program.

Fig. 2 contains the algorithms for insertion (Insert), search (Search) and removal (Remove). Fig. 2 also shows the most general client MGC that non-deterministically performs calls to skiplist operations, and can exercise all possible sequences of calls. We use MGC to verify properties like skiplist preservation. The execution begins with an empty skiplist containing only *head* and *tail* nodes at level 0, that has already been created. New nodes are then added using Insert. To maintain *reg* and *elems*: (a) a new node becomes part of the skiplist when it is connected at level 0 in Insert; and (b) a node stops being part of the skiplist when it is disconnected at level 0 in Remove. For simplicity, we assume in this paper that *val* and *key* contain the same object. We wish to prove that in all reachable states of MGC the memory layout is that of a "skiplist". We will also show that this datatype implements a set.

3 TSL: The Theory of Skiplists of Arbitrary Height

We use many-sorted first order logic to define TSL, as a combination of theories. We begin with a brief overview of notation and concepts. A signature Σ is a triple (S, F, P) where S is a set of sorts, F a set of functions and P a set of predicates. If $\Sigma_1 = (S_1, F_1, P_1)$ and $\Sigma_2 = (S_2, F_2, P_2)$, we define $\Sigma_1 \cup \Sigma_2 = (S_1 \cup S_2, F_1 \cup F_2, P_1 \cup P_2)$. Similarly we say that $\Sigma_1 \subseteq \Sigma_2$ when $S_1 \subseteq S_2$, $F_1 \subseteq F_2$ and $P_1 \subseteq P_2$. Let t be a term and φ a formula. We denote with $V_\sigma(t)$ (resp. $V_\sigma(\varphi)$) the set of variables of sort σ occurring in t (resp. φ). Similarly, we denote with $C_\sigma(t)$ (resp. $C_\sigma(\varphi)$) the set of constants of sort σ occurring in t (resp. φ).

A Σ-interpretation is a map from symbols in Σ to values (see, e.g., [6]). A Σ-structure is a Σ-interpretation over an empty set of variables. A Σ-formula over a set X of variables is satisfiable whenever it is true in some Σ-interpretation over X. A Σ-theory is a pair $(\Sigma, \mathbf{A})$ where Σ is a signature and $\mathbf{A}$ is a class of Σ-structures. Given a theory $T = (\Sigma, \mathbf{A})$, a T-interpretation is a Σ-interpretation $\mathcal{A}$ such that $\mathcal{A}^\Sigma \in \mathbf{A}$, where $\mathcal{A}^\Sigma$ is $\mathcal{A}$ restricted to interpret no variables. Given a Σ-theory T, a Σ-formula φ over a set of variables X is T-satisfiable whenever it is true on a T-interpretation over X.

Formally, the theory of skiplists of arbitrary height is defined as $\mathsf{TSL} = (\Sigma_{\mathsf{TSL}}, \mathbf{TSL})$, where Σ_{TSL} is the union of the following signatures, shown in Fig. 3 $\Sigma_{\mathsf{TSL}} = \Sigma_{\mathsf{level}} \cup \Sigma_{\mathsf{ord}} \cup \Sigma_{\mathsf{array}} \cup \Sigma_{\mathsf{cell}} \cup \Sigma_{\mathsf{mem}} \cup \Sigma_{\mathsf{reach}} \cup \Sigma_{\mathsf{set}} \cup \Sigma_{\mathsf{bridge}}$, and $\mathbf{TSL}$ is the class of Σ_{TSL}-structures satisfying the conditions listed in Fig. 4.

Informally, sort addr represents addresses; elem the universe of elements that can be stored in the skiplist; level the levels of a skiplist; ord the ordered keys used to preserve a strict order in the skiplist; array corresponds to arrays of addresses, indexed by levels; cell models *cells* representing objects of class *Node*; mem models the heap, a map from addresses to cells; path describes finite sequences of non-repeating addresses to model non-cyclic list paths; finally, set models sets of addresses—also known as regions.

Σ_{set} is interpreted as finite sets of addresses. Σ_{level} as natural numbers with order and addition with constants. Σ_{ord} models the order between elements, and contains two special elements $-\infty$ and $+\infty$ for the lowest and highest values in the order $\preceq$. Σ_{array} is the theory of arrays [5, 11] with two operations: $A[i]$ to model that an element of sort addr is stored in array A at position i of sort level, and $A\{i \leftarrow a\}$ for an array update, which returns the array that results from A by replacing the element at position i with a. Σ_{cell} contains the constructors and selectors for building and inspecting cells, including *error* for incorrect dereferences. Σ_{mem} is the signature of heaps, with the usual memory access and single memory mutation functions. The signature Σ_{reach} contains predicates to check reachability of addresses using paths at different levels. Finally, Σ_{bridge} contains auxiliary functions and predicates to manipulate and inspect paths as well as a native predicate for the skiplist memory shape. In the paper, for a variable l of sort level, we generally use $l + 1$ for $s(l)$.

Signt	Sort	Functions	Predicates
Σ_{level}	level	0 : level s : level $\to$ level	$<$: level $\times$ level
Σ_{ord}	ord	$-\infty, +\infty$: ord	$\preceq$: ord $\times$ ord
Σ_{array}	array level addr	$\_[\_]$: array $\times$ level $\to$ addr $\_\{\_ \leftarrow \_\}$: array $\times$ level $\times$ addr $\to$ array	
Σ_{cell}	cell elem ord array addr level	$error$: cell $mkcell$: elem $\times$ ord $\times$ array $\times$ level $\to$ cell $\_.data$: cell $\to$ elem $\_.key$: cell $\to$ ord $\_.arr$: cell $\to$ array $\_.max$: cell $\to$ level	
Σ_{mem}	mem addr cell	$null$: addr rd : mem $\times$ addr $\to$ cell upd : mem $\times$ addr $\times$ cell $\to$ mem	
Σ_{reach}	mem addr path	ϵ : path $[\_]$: addr $\to$ path	$append$: path $\times$ path $\times$ path $reach$: mem $\times$ addr $\times$ addr $\times$ level $\times$ path
Σ_{set}	addr set	$\emptyset$: set $\{\_\}$: addr $\to$ set $\cup, \cap, \setminus$: set $\times$ set $\to$ set	$\in$: addr $\times$ set $\subseteq$: set $\times$ set
Σ_{bridge}	mem addr set path level	$path2set$: path $\to$ set $addr2set$: mem $\times$ addr $\times$ level $\to$ set $getp$: mem $\times$ addr $\times$ addr $\times$ level $\to$ path	$ordList$: mem $\times$ path $skiplist$: mem $\times$ set $\times$ level $\times$ addr $\times$ addr

Fig. 3. The signature of the TSL theory

Each sort σ in Σ_{TSL} is mapped to a non-empty set $\mathcal{A}_\sigma$ such that:

(a) $\mathcal{A}_{\mathsf{addr}}$ and $\mathcal{A}_{\mathsf{elem}}$ are discrete sets	(b) $\mathcal{A}_{\mathsf{level}}$ is the naturals with order
(c) $\mathcal{A}_{\mathsf{ord}}$ is a total ordered set	(d) $\mathcal{A}_{\mathsf{array}} = \mathcal{A}_{\mathsf{addr}}^{\mathcal{A}_{\mathsf{level}}}$
(e) $\mathcal{A}_{\mathsf{cell}} = \mathcal{A}_{\mathsf{elem}} \times \mathcal{A}_{\mathsf{ord}} \times \mathcal{A}_{\mathsf{array}} \times \mathcal{A}_{\mathsf{level}}$	(f) $\mathcal{A}_{\mathsf{path}}$ is the set of all finite sequences of (pairwise) distinct elements of $\mathcal{A}_{\mathsf{addr}}$
(g) $\mathcal{A}_{\mathsf{mem}} = \mathcal{A}_{\mathsf{cell}}^{\mathcal{A}_{\mathsf{addr}}}$	
(h) $\mathcal{A}_{\mathsf{set}}$ is the power-set of $\mathcal{A}_{\mathsf{addr}}$	

Signature	Interpretation
Σ_{level}	• $0^{\mathcal{A}} = 0$ • $s^{\mathcal{A}}(l) = s(l)$, for each $l \in \mathcal{A}_{\mathsf{level}}$
Σ_{ord}	• $x \preceq^{\mathcal{A}} y \wedge y \preceq^{\mathcal{A}} x \rightarrow x = y$ • $x \preceq^{\mathcal{A}} y \vee y \preceq^{\mathcal{A}} x$ • $x \preceq^{\mathcal{A}} y \wedge y \preceq^{\mathcal{A}} z \rightarrow x \preceq^{\mathcal{A}} z$ • $-\infty^{\mathcal{A}} \preceq^{\mathcal{A}} x \wedge x \preceq^{\mathcal{A}} +\infty^{\mathcal{A}}$ for any $x, y, z \in \mathcal{A}_{\mathsf{ord}}$
Σ_{array}	• $A[l]^{\mathcal{A}} = A(l)$ • $A\{l \leftarrow a\}^{\mathcal{A}} = B$, where $B(l) = a$ and $B(i) = A(i)$ for $i \neq l$ for each $A, B \in \mathcal{A}_{\mathsf{array}}$, $l \in \mathcal{A}_{\mathsf{level}}$ and $a \in \mathcal{A}_{\mathsf{addr}}$
Σ_{cell}	• $mkcell^{\mathcal{A}}(e, k, A, l) = \langle e, k, A, l \rangle$ • $error^{\mathcal{A}}.arr^{\mathcal{A}}(l) = null^{\mathcal{A}}$ • $\langle e, k, A, l \rangle.data^{\mathcal{A}} = e$ • $\langle e, k, A, l \rangle.key^{\mathcal{A}} = k$ • $\langle e, k, A, l \rangle.arr^{\mathcal{A}} = A$ • $\langle e, k, A, l \rangle.max^{\mathcal{A}} = l$ for each $e \in \mathcal{A}_{\mathsf{elem}}$, $k \in \mathcal{A}_{\mathsf{ord}}$, $A \in \mathcal{A}_{\mathsf{array}}$, and $l \in \mathcal{A}_{\mathsf{level}}$
Σ_{mem}	• $rd(m, a)^{\mathcal{A}} = m(a)$ • $upd^{\mathcal{A}}(m, a, c) = m_{a \mapsto c}$ • $m^{\mathcal{A}}(null^{\mathcal{A}}) = error^{\mathcal{A}}$ for each $m \in \mathcal{A}_{\mathsf{mem}}$, $a \in \mathcal{A}_{\mathsf{addr}}$ and $c \in \mathcal{A}_{\mathsf{cell}}$
Σ_{reach}	• $\epsilon^{\mathcal{A}}$ is the empty sequence • $[a]^{\mathcal{A}}$ is the sequence containing $a \in \mathcal{A}_{\mathsf{addr}}$ as the only element • $([a_1 \,..\, a_n], [b_1 \,..\, b_m], [a_1 \,..\, a_n, b_1 \,..\, b_m]) \in append^{\mathcal{A}}$ iff $a_k \neq b_l$. • $(m, a_{init}, a_{end}, l, p) \in reach^{\mathcal{A}}$ iff $a_{init} = a_{end}$ and $p = \epsilon$, or there exist addresses $a_1, \ldots, a_n \in \mathcal{A}_{\mathsf{addr}}$ such that: (a) $p = [a_1 \,..\, a_n]$ (c) $m(a_r).arr^{\mathcal{A}}(l) = a_{r+1}$, for $r < n$ (b) $a_1 = a_{init}$ (d) $m(a_n).arr^{\mathcal{A}}(l) = a_{end}$
Σ_{bridge}	for each $m \in \mathcal{A}_{\mathsf{mem}}$, $p \in \mathcal{A}_{\mathsf{path}}$, $l \in \mathcal{A}_{\mathsf{level}}$, $a_i, a_e \in \mathcal{A}_{\mathsf{addr}}$, $r \in \mathcal{A}_{\mathsf{set}}$ • $path2set^{\mathcal{A}}(p) = \{a_1, \ldots, a_n\}$ for $p = [a_1, \ldots, a_n] \in \mathcal{A}_{\mathsf{path}}$ • $addr2set^{\mathcal{A}}(m, a, l) = \{a' \mid \exists p \in \mathcal{A}_{\mathsf{path}} \ . \ (m, a, a', l, p) \in reach^{\mathcal{A}}\}$ • $getp^{\mathcal{A}}(m, a_i, a_e, l) = p$ if $(m, a_i, a_e, l, p) \in reach^{\mathcal{A}}$, and ϵ otherwise • $ordList^{\mathcal{A}}(m, p)$ iff $p = \epsilon$ or $p = [a]$, or $p = [a_1, \ldots, a_n]$ with $n \geq 2$ and $m(a_j).key^{\mathcal{A}} \preceq m(a_{j+1}).key^{\mathcal{A}}$ for all $1 \leq j < n$, for any $m \in \mathcal{A}_{\mathsf{mem}}$ • $skiplist^{\mathcal{A}}(m, r, l, a_i, a_e)$ iff $\left[\begin{array}{c} ordList^{\mathcal{A}}(m, getp^{\mathcal{A}}(m, a_i, a_e, 0)) \quad \wedge \\ r = addr2set^{\mathcal{A}}(m, a_i, 0) \quad \wedge \\ 0 \leq l \wedge \forall a \in r \ . \ m(a).max^{\mathcal{A}} \leq l \quad \wedge \\ m(a_e).arr^{\mathcal{A}}(l) = null^{\mathcal{A}} \quad \wedge \\ (0 = l) \vee \\ (\exists l_p \ . \ s^{\mathcal{A}}(l_p) = l \wedge \forall \, i \in 0, \ldots, l_p \ . \\ m(a_e).arr^{\mathcal{A}}(i) = null^{\mathcal{A}} \wedge \\ path2set^{\mathcal{A}}(getp^{\mathcal{A}}(m, a_i, a_e, s^{\mathcal{A}}(i))) \subseteq \\ path2set^{\mathcal{A}}(getp^{\mathcal{A}}(m, a_i, a_e, i))) \end{array}\right]$

Fig. 4. Characterization of a TSL-interpretation $\mathcal{A}$

4 Decidability of TSL

In this section we prove the decidability of the satisfiability problem of quantifier-free TSL formulas. We first start with some preliminaries.

Preliminaries A flat literal is of the form $x = y$, $x \neq y$, $x = f(y_1, \ldots, y_n)$, $p(y_1, \ldots, y_n)$ or $\neg p(y_1, \ldots, y_n)$, where $x, y, y_1, \ldots, y_n$ are variables, f is a function symbol and p is a predicate symbol defined in the signature of TSL. We first identify a set of normalized literals. All other literals can be converted into normalized literals.

Lemma 1 (Normalized Literals). *Every TSL-formula is equivalent to a disjunction of conjunctions of literals of the following list, called* normalized TSL-literals*:*

$e_1 \neq e_2$	$a_1 \neq a_2$	$l_1 \neq l_2$
$a = \mathit{null}$	$c = \mathit{error}$	$c = \mathit{rd}(m, a)$
$k_1 \neq k_2$	$k_1 \preceq k_2$	$m_2 = \mathit{upd}(m_1, a, c)$
$c = \mathit{mkcell}(e, k, A, l)$	$l_1 < l_2$	$l = q$
$s = \{a\}$	$s_1 = s_2 \cup s_3$	$s_1 = s_2 \setminus s_3$
$a = A[l]$	$B = A\{l \leftarrow a\}$	
$p_1 \neq p_2$	$p = [a]$	$p_1 = \mathit{rev}(p_2)$
$s = \mathit{path2set}(p)$	$\mathit{append}(p_1, p_2, p_3)$	$\neg \mathit{append}(p_1, p_2, p_3)$
$s = \mathit{addr2set}(m, a, l)$	$p = \mathit{getp}(m, a_1, a_2, l)$	
$\mathit{ordList}(m, p)$		$\mathit{skiplist}(m, s, a_1, a_2)$

For instance, $e = c.\mathit{data}$ can be rewritten as $c = \mathit{mkcell}(e, k, A, l)$ for fresh variables k, A and l. The predicate $\mathit{reach}(m, a_1, a_2, l, p)$ can be similarly translated into $a_2 \in \mathit{addr2set}(m, a_1, l) \wedge p = \mathit{getp}(m, a_1, a_2, l)$. Similar translations can be defined for $\neg \mathit{ordList}(m, p)$, $\neg \mathit{skiplist}(m, r, l, a_i, a_e)$, etc.

We will use the following formula ψ as a running example:

$$\psi \quad : \quad i = 0 \wedge A = \mathit{rd}(\mathit{heap}, \mathit{head}).\mathit{arr} \wedge B = A\{i \leftarrow \mathit{tail}\} \wedge \mathit{rd}(\mathit{heap}, \mathit{head}).\mathit{max} = 3.$$

This formula establishes that B is the array obtained from the next pointers of node *head*, by replacing the pointer at the lower level by *tail*. To check the satisfiability of ψ we first normalize it, obtaining ψ_{norm}:

$$\psi_{\text{norm}} \quad : \quad i = 0 \ \wedge \left(\begin{array}{c} c = \mathit{rd}(\mathit{heap}, \mathit{head}) \ \wedge \\ c = \mathit{mkcell}(e, k, A, l) \ \wedge \\ l = 3 \end{array} \right) \wedge \ B = A\{i \leftarrow \mathit{tail}\}.$$

4.1 A Decision Procedure for TSL

Fig. 5 sketches a decision procedure for the satisfiability problem of TSL formulas, by reducing it to the satisfiability of quantifier-free TSL_K formulas and quantifier-free Presburger arithmetic formulas. We start from a TSL formula φ expressed as a normalized conjunction of literals. The main idea is to guess a feasible

```
To decide whether $\varphi_{\text{in}}$ : $\mathsf{TSL}$ is SAT:

STEP 1. Sanitize:
    $\varphi := \varphi_{\text{in}} \wedge \bigwedge_{B=A\{l \leftarrow a\} \in \varphi_{\text{in}}} (l_{new} = l + 1)$
STEP 2. Guess arrangement $\alpha$ of $V_{\mathsf{level}}(\varphi)$.
STEP 3. Split $\varphi$ into $(\varphi^{\text{PA}} \wedge \alpha)$, $(\varphi^{\text{NC}} \wedge \alpha)$.
STEP 4. Check SAT of $(\varphi^{\text{PA}} \wedge \alpha)$.
            If UNSAT $\rightarrow$ return UNSAT
STEP 5. Check SAT of $(\varphi^{\text{NC}} \wedge \alpha)$ as follows:
    5.1 Let $k = |V_{\mathsf{level}}(\varphi^{\text{NC}} \wedge \alpha)|$.
    5.2 Check $\ulcorner \varphi^{\text{NC}} \wedge \alpha \urcorner$ : $\mathsf{TSL_K}(k)$:
            If SAT $\rightarrow$ return SAT
                else return UNSAT.
```

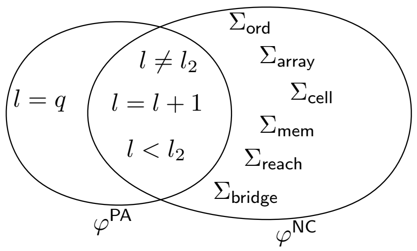

Fig. 5. A decision procedure for the satisfibility of TSL formulas (left). A split of φ obtained after $\mathsf{STEP\ 1}$ into φ^{PA} and φ^{NC} (right).

arrangement between level variables, and then extract from φ the *relevant levels*, generating a $\mathsf{TSL_K}$ formula using only relevant levels. To show correctness, we will see that from the resulting model of the $\mathsf{TSL_K}$ formula we can create a model of the original TSL formula by replicating relevant levels into missing intermediate levels.

STEP 1: Sanitization The decision procedure begins by sanitizing the normalized collection of literals received as input. A formula is sanitized when the level right above array updates is named explicitly by a level variable. Sanitization serves to infer the existence of large models from smaller models where only named levels are populated, in Theorem 1 below.

Definition 1 (Sanitized). *A conjunction of normalized literals is sanitized if for every literal $B = A\{l \leftarrow a\}$ there is a literal of the form $l_2 = l + 1$.*

A formula can be sanitized by adding a fresh variable l_{new} and a literal $l_{new} = l{+}1$ for every literal $B = A\{l \leftarrow a\}$ in case there is no literal $l_2 = l + 1$ already in the formula. Sanitizing a formula does not affect its satisfiability because it only adds an arithmetic constraint $(l_{new} = l + 1)$ for a fresh new variable l_{new}. For example, sanitizing ψ_{norm} we obtain $\psi_{\text{sanit}} : \psi_{\text{norm}} \ \wedge \ l_{new} = i + 1$.

STEP 2: Order arrangements, and STEP 3: Split A model of a formula assigns a natural number to every level variable. Hence, every two variables are either assigned the same value or are ordered by $<$. An *order arrangement* is an arithmetic formula that captures this relation between level variables.

Definition 2 (Order Arrangement). *Given a sanitized formula φ, an order arrangement is a collection of literals containing, for every pair of level variables $l_1, l_2 \in V_{\mathsf{level}}(\varphi)$, exactly one of $(l_1 = l_2), (l_1 < l_2)$, or $(l_2 < l_1)$.*

For instance, an order arrangement of ψ_{sanit} is $\{i < l_{new}, i < l, l_{new} < l\}$. Since there is a finite number of level variables in a formula φ, there is a finite number of order arrangements. Note also that a formula φ is satisfiable if and only if there is an order arrangement α such that $\varphi \wedge \alpha$ is satisfiable. STEP 2 of the decision procedure consists of guessing an order arrangement α.

STEP 3 of the decision procedure first splits the sanitized formula φ into φ^{PA}, which contains precisely all those literals in the theory of arithmetic Σ_{level}, and φ^{NC} containing all literals from φ except those involving constants $(l = q)$. Clearly, φ is equivalent to $\varphi^{\text{NC}} \wedge \varphi^{\text{PA}}$. In our running example, ψ_{sanit} is split into ψ^{PA} and ψ^{NC}:

$$\begin{aligned} \psi^{\text{PA}} &: \quad i = 0 \wedge l = 3 \wedge l_{new} = i + 1 \\ \psi^{\text{NC}} &: \quad \begin{pmatrix} c = rd(heap, head) \wedge \\ c = mkcell(e, k, A, l) \end{pmatrix} \wedge B = A\{i \leftarrow tail\} \wedge l_{new} = i + 1. \end{aligned}$$

STEP 3 uses the order arrangement to reduce the satisfiability of a sanitized formula φ that follows an order arrangement α into the satisfiability of a Presburger Arithmetic formula $(\varphi^{\text{PA}} \wedge \alpha)$, checked in STEP 4, and the satisfiability of a sanitized formula without constants $(\varphi^{\text{NC}} \wedge \alpha)$, checked in STEP 5. An essential notion to show the correctness of this split is that of a *gap*, which is a level in a model that is not named by a level variable.

Definition 3 (Gap). *Let $\mathcal{A}$ be a model of φ. We say that a number n is a* gap *in $\mathcal{A}$ if there are variables l_1, l_2 in $V_{\mathsf{level}}(\varphi)$ such that $l_1^{\mathcal{A}} < n < l_2^{\mathcal{A}}$, but there is no l in $V_{\mathsf{level}}(\varphi)$ with $l^{\mathcal{A}} = n$.*

Consider ψ_{sanit} for which $V_{\mathsf{level}}(\psi_{\text{sanit}}) = \{i, l_{new}, l\}$. A model $\mathcal{A}_\psi$ of ψ that interprets variables i, l_{new} and l as 0, 1 and 3 respectively has a gap at 2.

Definition 4 (Gap-less model). *A model $\mathcal{A}$ of φ is a gap-less model whenever it has no gaps, and for every array C in $\mathsf{array}^{\mathcal{A}}$ and level $n > l^{\mathcal{A}}$ for all $l \in V_{\mathsf{level}}(\varphi)$, $C(n) = null$.*

We will prove the existence of a gap-less model given that there is a model. But, before, we need one last auxiliary notion to ease the construction of similar models, by setting a condition under which reachability at different levels is preserved.

Definition 5. *Two interpretations $\mathcal{A}$ and $\mathcal{B}$ of φ* agree *on sorts σ whenever*

(i) $\mathcal{A}_\sigma = \mathcal{B}_\sigma$,

(ii) for every $v \in V_\sigma(\varphi)$, $v^{\mathcal{A}} = v^{\mathcal{B}}$,

(iii) for every function symbol f with domain and co-domain from sorts in σ, $f^{\mathcal{A}} = f^{\mathcal{B}}$ and for every predicate symbol P with domain in σ, $P^{\mathcal{A}}$ iff $P^{\mathcal{B}}$.

Lemma 2. *Let $\mathcal{A}$ and $\mathcal{B}$ be two interpretations of a sanitized formula φ that agree on sorts $\{\mathsf{addr}, \mathsf{elem}, \mathsf{ord}, \mathsf{path}, \mathsf{set}\}$, and s.t. for every $l \in V_{\mathsf{level}}(\varphi)$, $m \in V_{\mathsf{mem}}(\varphi)$, and $a \in \mathsf{addr}^{\mathcal{A}}$, the following holds: $m^{\mathcal{A}}(a).arr^{\mathcal{A}}(l^{\mathcal{A}}) = m^{\mathcal{B}}(a).arr^{\mathcal{B}}(l^{\mathcal{B}})$ Then, $\left(reach^{\mathcal{A}}(m^{\mathcal{A}}, a^{\mathcal{A}}_{init}, a^{\mathcal{A}}_{end}, l^{\mathcal{A}}, p^{\mathcal{A}})\right)$ iff $\left(reach^{\mathcal{B}}(m^{\mathcal{B}}, a^{\mathcal{B}}_{init}, a^{\mathcal{B}}_{end}, l^{\mathcal{B}}, p^{\mathcal{B}})\right)$.*

Proof (Sketch). The proof follows an inductive argument on the length of the paths returned by reach.

We show now that if a sanitized formula without constants, as the one obtained after the split in STEP 3, has a model then it has a model without gaps.

Lemma 3 (Gap-reduction). *Let $\mathcal{A}$ be a model of a sanitized formula φ without constants, and let $\mathcal{A}$ have a gap at n. Then, there is a model $\mathcal{B}$ of φ such that, for every $l \in V_{\mathsf{level}}(\varphi)$: $l^{\mathcal{B}} = l^{\mathcal{A}} - 1$ if $l^{\mathcal{A}} > n$, and $l^{\mathcal{B}} = l^{\mathcal{A}}$ if $l^{\mathcal{A}} < n$. The number of gaps in $\mathcal{B}$ is one less than in $\mathcal{A}$.*

Proof. (Sketch) We show here the construction of the model $\mathcal{B}$ and leave the exhaustive case analysis of each literal for the appendix. Let $\mathcal{A}$ be a model of φ with a gap at n. We build a model $\mathcal{B}$ with the condition in the lemma. $\mathcal{B}$ agrees with $\mathcal{A}$ on addr, elem, ord, path, set. In particular, $v^{\mathcal{B}} = v^{\mathcal{A}}$ for variables of these sorts. For the other sorts we let $\mathcal{B}_\sigma = \mathcal{A}_\sigma$ for $\sigma =$ level, array, cell, mem. We define transformation maps for elements of the corresponding domains as follows:

$$\beta_{\mathsf{level}}(j) = \begin{cases} j & \text{if } j < n \\ j-1 & \text{otherwise} \end{cases} \qquad \beta_{\mathsf{array}}(A)(i) = \begin{cases} A(i) & \text{if } i < n \\ A(i+1) & \text{if } i \geq n \end{cases}$$

$$\beta_{\mathsf{cell}}((e,k,A,l)) = (e,k,\beta_{\mathsf{array}}(A),\beta_{\mathsf{level}}(l)) \qquad \beta_{\mathsf{mem}}(m)(a) = \beta_{\mathsf{cell}}(m(a))$$

Now, for variables l : level, A : array, c : cell and m : mem, we simply let $l^{\mathcal{B}} = \beta_{\mathsf{level}}(l^{\mathcal{A}})$, $A^{\mathcal{B}} = \beta_{\mathsf{array}}(A^{\mathcal{A}})$, $c^{\mathcal{B}} = \beta_{\mathsf{cell}}(c^{\mathcal{A}})$, and $m^{\mathcal{B}} = \beta_{\mathsf{mem}}(m^{\mathcal{A}})$.

The interpretation of all functions and predicates is preserved from $\mathcal{A}$. An exhaustive case analysis on the normalized literals allows to show that $\mathcal{B}$ is indeed a model of φ. □

For instance, consider formula ψ_{sanit} and model $\mathcal{A}_\psi$ above. We can construct model $\mathcal{B}$ reducing one gap from $\mathcal{A}_\psi$ by stating that $i^{\mathcal{B}} = i^{\mathcal{A}_\psi}$, ${l_{new}}^{\mathcal{B}} = {l_{new}}^{\mathcal{A}_\psi}$ and $l^{\mathcal{B}} = 2$, and completely ignore arrays in model $\mathcal{A}_\psi$ at level 2.

Similarly, by a simple case analysis of the literals of φ and Lemma 2 the following Lemma holds, and the corollary that shows the existence of gapless models.

Lemma 4 (Top-reduction). *Let $\mathcal{A}$ be a model of φ, and n a level such that $n > l^{\mathcal{A}}$ for all $l \in V_{\mathsf{level}}(\varphi)$ and $A \in \mathsf{array}^{\mathcal{A}}$ be such that $A(n) \neq null$. Then the interpretation $\mathcal{B}$ obtained from $\mathcal{A}$ by replacing $A(n) = null$ is also a model of φ.*

Corollary 1. *Let φ be a sanitized formula without constants. Then, φ has a model if and only if φ has a gapless model.*

We show now that STEP 2 and the split in STEP 3 preserve satisfiability.

Theorem 1. *A sanitized TSL formula φ is satisfiable if and only if for some order arrangement α, both $(\varphi^{PA} \wedge \alpha)$ and $(\varphi^{NC} \wedge \alpha)$ are satisfiable.*

Proof. (Sketch) The "$\Rightarrow$" direction follows immediately, since a model of φ contains a model of its subformulas φ^{PA} and φ^{NC}, and a model of φ^{PA} induces a satisfying order arrangement α.

For "$\Leftarrow$", let α be an order arrangement for which both $(\varphi^{\mathrm{PA}} \wedge \alpha)$ and $(\varphi^{\mathrm{NC}} \wedge \alpha)$ are satisfiable, and let $\mathcal{B}$ be a model of $(\varphi^{\mathrm{NC}} \wedge \alpha)$ and $\mathcal{A}$ be a model of $(\varphi^{\mathrm{PA}} \wedge \alpha)$. By Corollary 1, we assume that $\mathcal{B}$ is a gapless model. The obstacle in merging the models is that the values for levels in $\mathcal{A}$ and in $\mathcal{B}$ may differ. We will build a model $\mathcal{C}$ of φ using $\mathcal{B}$ and $\mathcal{A}$. In $\mathcal{C}$, all levels will receive $l^{\mathcal{C}} = l^{\mathcal{A}}$, but all other sorts will be filled according to $\mathcal{B}$, including the contents of cells at level l, which will be the corresponding cells of $\mathcal{B}$ at level $l^{\mathcal{B}}$. The remaining issue is how to fill intermediate levels, not existing in $\mathcal{B}$. Levels can be populated cloning existing levels from $\mathcal{B}$, illustrated in Fig. 6 (a) below. The two reasonable candidates to populate the levels between $l_1^{\mathcal{C}}$ and $l_2^{\mathcal{C}}$, are level $l_1^{\mathcal{B}}$ and level $l_2^{\mathcal{B}}$, but without sanitation both options can lead to a predicate changing its truth value between models $\mathcal{B}$ and $\mathcal{C}$, as illustrated in Fig. 6 (b) and (c). With sanitation, level l_{new} can be used to populate the intermediate levels, preserving the truth values of all predicates between models $\mathcal{B}$ and $\mathcal{C}$. The detailed proof is in the appendix. ☐

STEP 4: Presburger Constraints The formula $(\varphi^{\mathrm{PA}} \wedge \alpha)$ contains only literals of the form $l_1 = q$, $l_1 \neq l_2$, $l_1 = l_2 + 1$, and $l_1 < l_2$ for integer variables l_1 and l_2 and integer constant q, a simple fragment of Presburger Arithmetic.

STEP 5: Deciding Satisfiability of Formulas Without Constants In STEP 5 we reduce a sanitized formula without constants ψ into an equisatisfiable formula $\ulcorner\psi\urcorner$ in the decidable theory $\mathsf{TSL_K}$, for a finite value $\mathsf{K} = |\,V_{\mathsf{level}}(\psi)|$ computed from the formula. This bound provides the number of levels required in necessary to reason about the satisfiability of ψ. We use $[\mathsf{K}]$ as a short for the set $0 \ldots \mathsf{K} - 1$. For ψ_{sanit}, we have $\mathsf{K} = 3$ and thus we construct a formula in $\mathsf{TSL_3}$.

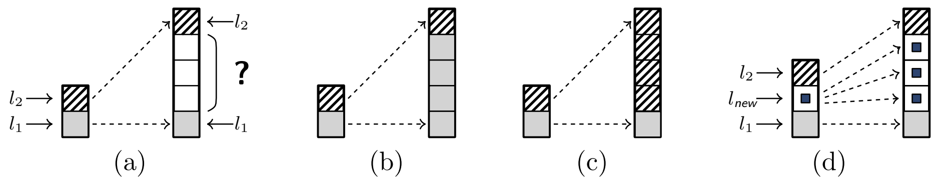

Fig. 6. Pumping a model of φ^{NC} to a model of φ is allowed thanks to the fresh level l_{new}. In (b) the truth value of $A = B\{l_1 \leftarrow e\}$ is not preserved. In (c) $A = B\{l_2 \leftarrow e\}$ is not preserved. In (d) all predicates are preserved.

The translation from ψ into $\ulcorner\psi\urcorner$ works as follows. For every variable A of sort array appearing in some literal in ψ we introduce K fresh new variables $v_{A[0]}, \ldots, v_{A[\mathsf{K}-1]}$ of sort addr. These variables correspond to the addresses from A that the decision procedure for $\mathsf{TSL_K}$ needs to reason about. All literals from ψ are left unchanged in $\ulcorner\psi\urcorner$ except $(c = \mathit{mkcell}(e, k, A, l))$, $(a = A[l])$, $(B = A\{l \leftarrow a\})$, $B = A$ and $\mathit{skiplist}(m, s, a_1, a_2)$ that are changed as follows:

- $c = \mathit{mkcell}(e, k, A, l)$ is transformed into $c = (e, k, v_{A[0]}, \ldots, v_{A[K-1]})$.
- $a = A[l]$ gets translated into: $\bigwedge\limits_{i \in [\mathsf{K}]} l = i \rightarrow a = v_{A[i]}$.
- $B = A\{l \leftarrow a\}$ is translated into:

$$\Big(\bigwedge_{i \in [\mathsf{K}]} l = i \rightarrow a = v_{B[i]} \Big) \wedge \Big(\bigwedge_{j \in [\mathsf{K}]} l \neq j \rightarrow v_{B[j]} = v_{A[j]} \Big)$$

- $\mathit{skiplist}(m, r, a_1, a_2)$ gets translated into:

$$\begin{array}{l} \mathit{ordList}(m, \mathit{getp}(m, a_1, a_2, 0)) \quad \wedge \quad r = \mathit{path2set}(\mathit{getp}(m, a_1, a_2, 0)) \wedge \\ \bigwedge\limits_{i \in [\mathsf{K}]} \mathit{rd}(m, a_2).\mathit{arr}[i] = \mathit{null} \hfill \wedge \\ \bigwedge\limits_{i \in [\mathsf{K}-1]} \mathit{path2set}(\mathit{getp}(m, a_1, a_2, i+1)) \subseteq \mathit{path2set}(\mathit{getp}(m, a_1, a_2, i)) \end{array}$$

Note that the formula $\ulcorner\varphi\urcorner$ obtained using this translation belongs to the theory $\mathsf{TSL_K}$. For instance, in our running example,

$$\ulcorner\psi^{\mathrm{NC}}\urcorner : \begin{bmatrix} i = 0 \rightarrow \mathit{tail} = v_{B[0]} & \wedge\, i = 1 \rightarrow \mathit{tail} = v_{B[1]} & \wedge\, i = 2 \rightarrow \mathit{tail} = v_{B[2]} & \wedge \\ i \neq 0 \rightarrow v_{B[0]} = v_{A[0]} & \wedge\, i \neq 1 \rightarrow v_{B[1]} = v_{A[1]} & \wedge\, i \neq 2 \rightarrow v_{B[2]} = v_{A[2]} & \wedge \\ c = \mathit{rd}(\mathit{heap}, \mathit{head}) & \wedge\, c = \mathit{mkcell}(e, k, v_{A[0]}, v_{A[1]}, v_{A[2]}) & \wedge\, l_{\mathit{new}} = i + 1 & \end{bmatrix}$$

The following lemmas establishes the correctness of the translation.

Lemma 5. *Let ψ be a sanitized TSL formula with no constants. Then, ψ is satisfiable if and only if $\ulcorner\psi\urcorner$ is also satisfiable.*

The main result of this paper is the following decidability theorem, which follows from Lemma 5, Theorem 1 and the fact that every formula can be normalized and sanitized.

Theorem 2. *The satisfiability problem of QF TSL-formulas is decidable.*

5 Shape and Functional Verification

In this section we report an empirical evaluation of the verification of two implementations of a skiplist, including an implementation from the open-source project KDE. The TSL decision procedure has been integrated in LEAP[3], a theorem prover being developed at IMDEA, based on parametrized proof rules [18]. A

[3] LEAP and all examples can be downloaded from *http://software.imdea.org/leap*.

TSL query is ultimately decomposed into simple Presburger arithmetic formulas and $\mathsf{TSL_K}$ formulas. We use the decidability theorem for $\mathsf{TSL_K}$ formulas, which essentially computes a cardinality bound on a small model. Then, the $\mathsf{TSL_K}$ procedure encodes in SMT the existence of one such small model by unrolling predicates (like e.g., *reach*) up to the computed bound.

We verified two kinds of properties: skiplist memory shape preservation and functional correctness. Memory preservation is stated using the predicate *skiplist* from TSL, and verified using the most general client from Fig. 2. For functional verification we use the simple spec in Fig. 7.

The proof of shape preservation requires some auxiliary invariants region, next and order ($\mathsf{skiplist_{KDE}}$, $\mathsf{nodes_{KDE}}$, $\mathsf{pointers_{KDE}}$ and $\mathsf{values_{KDE}}$ in the KDE implementation) that capture the separation between the skiplist region and new cells and the relation between the pointers used to traverse the skiplist. The precise definitions can be found in the web page of Leap. Fig. 8 reports an evaluation of the performance of the decision procedure described in this paper for these two implementations with unbounded number of levels (first 8 rows of the table) compared with the performance of using only $\mathsf{TSL_K}$ on implementations with bounded levels (for bounds 1, 2, 3 and 4). The results for proving skiplist and its auxiliary invariants appear in the first four rows, and for $\mathsf{skiplist_{KDE}}$ in the following four. The corresponding invariants for bounded levels are reported in rows labeled $\mathsf{skiplist}_i$, region_i, next_i and order_i for $i = 1, 2, 3, 4$. Column #VC shows the number of VCs generated. Column $\#\varphi$ shows the number of formulas generated from these VCs (after normalization, etc). Column TSL shows the number of queries to the TSL decision procedure, and column TSL_i the number of queries to $\mathsf{TSL_K}$ for $\mathsf{K} = i$. A query to TSL can result in several queries to TSL_i. In some cases there are fewer queries than formulas, because some formulas are trivially simplified. Finally, the columns labeled “avg” and “slowest” report the average and slowest running time to prove all VCs. The time reported in the column Leap corresponds to the total verification time excluding the invocation to the DPs. The column DP reports the total running time used in invoking all DPs.

```
1: procedure FuncSearch(SkipList sl)
2:     Set⟨Value⟩ elems_before := sl.elems
3:     v := NondetPickValue
4:     result := call Search(sl, v)
5:     assert $\begin{pmatrix} elems = elems\_before \wedge \\ result \leftrightarrow v \in elems \end{pmatrix}$
6: end procedure

1: procedure FuncInsert(SkipList sl)
2:     Set⟨Value⟩ elems_before := sl.elems
3:     v := NondetPickValue
4:     result := call Insert(sl, v)
5:     assert $(elems = elems\_before \cup \{v\})$
6: end procedure

1: procedure FuncRemove(SkipList sl)
2:     Set⟨Value⟩ elems_before := sl.elems
3:     v := NondetPickValue
4:     result := call Remove(sl, v)
5:     assert $(elems = elems\_before \setminus \{v\})$
6: end procedure
```

Fig. 7. Functional specification

This evaluation demonstrates that our decision procedure is practical to verify implementations with a variable number of levels, and allows to scale the verification of implementations with a fixed number of levels where previously known decision procedures time out. In the case of functional verification, Leap using the TSL decision procedure was capable of verifying all three specifications for the skiplist of unbounded height in less than one second.

	Formulas		#Calls to DPs					VC time (s.)		Total time (s.)	
	#VC	$\#\varphi$	TSL	TSL_1	TSL_2	TSL_3	TSL_4	slowest	avg	Leap	DP
skiplist	80	560	28	45	92	38	14	5.40	0.24	0.15	19.64
region	80	1583	56	111	185	76	–	22.66	0.54	1.35	42.93
next	80	1899	30	39	55	22	–	0.32	0.02	1.59	1.60
order	80	2531	57	167	286	116	4	2.35	0.84	4.44	6.75
skiplist$_{\mathsf{KDE}}$	54	214	14	37	61	32	12	5.93	0.24	0.05	13.14
nodes$_{\mathsf{KDE}}$	54	585	32	99	174	76	–	3.10	0.17	0.31	9.36
pointers$_{\mathsf{KDE}}$	54	1115	27	38	42	16	–	0.22	0.01	0.86	0.76
values$_{\mathsf{KDE}}$	54	797	34	120	194	76	–	0.64	0.06	0.69	3.06
skiplist$_1$	77	119	–	32	–	–	–	0.10	0.01	0.20	0.32
region$_1$	77	119	–	27	–	–	–	0.14	0.01	0.37	0.28
next$_1$	77	79	–	19	–	–	–	0.02	0.01	0.15	0.14
order$_1$	77	79	–	25	–	–	–	0.02	0.01	0.58	0.11
skiplist$_2$	79	137	–	–	47	–	–	2.15	0.05	0.35	4.13
region$_2$	79	122	–	–	27	–	–	1.08	0.03	0.46	2.44
next$_2$	79	82	–	–	19	–	–	0.06	0.01	0.18	0.27
order$_2$	79	82	–	–	25	–	–	0.68	0.01	0.95	0.95
skiplist$_3$	80	154	–	–	–	62	–	776.45	15.27	0.45	1221.52
region$_3$	80	124	–	–	–	27	–	17.36	0.34	0.58	26.92
next$_3$	80	84	–	–	–	19	–	0.09	0.01	0.20	0.47
order$_3$	80	84	–	–	–	25	–	7.80	0.10	1.31	8.35
skiplist$_4$	81	171	–	–	–	–	77	**T.O.**	**T.O.**	0.80	**T.O.**
region$_4$	81	126	–	–	–	–	27	226.08	4.30	0.79	348.44
next$_4$	81	86	–	–	–	–	19	0.22	0.01	0.25	0.83
order$_4$	81	86	–	–	–	–	25	43.97	0.56	1.83	45.28

Fig. 8. Number of queries and running times for the verification of skiplist shape preservation. **T.O.** means time out, '–' means no calls to DP were required.

6 Conclusion and Future Work

We have presented TSL, a theory of skiplists of arbitrary many levels, useful to automatically prove the VCs generated during the verification of skiplist implementations. We showed that TSL is decidable by reducing its satisfiability problem to $\mathsf{TSL_K}$, a decidable family of theories restricted to a bounded collection of levels. Our reduction illustrates that the decision procedure only needs to reason those levels explicitly mentioned in the (sanitized) formula. We have implemented our decision procedures on top of off-the-shelf SMT solvers (Yices and Z3), and integrated it into our prototype theorem prover. Our empirical evaluation demonstrates that our decision procedure is practical not only to verify unbounded skiplists but also to scale the verification of bounded implementations to realistic sizes.

Our main line of current and future work is the verification of liveness properties of concurrent skiplist implementations, as well as improving automation by generating and propagating invariants.

References

1. The KDE Platform. `http://kde.org/`.
2. KDE Skiplist implementation. `http://api.kde.org/4.1-api/kdeedu-apidocs/kstars/html/SkipList_8cpp_source.html`.
3. P. A. Abdulla, L. Holík, B. Jonsson, O. Lengál, C. Q. Trinh, and T. Vojnar. Verification of heap manipulating programs with ordered data by extended forest automata. In *Proc. of 11th ATVA*, volume 8172 of *LNCS*, pages 224–239. Springer, 2013.
4. A. Bouajjani, C. Dragoi, C. Enea, and M. Sighireanu. A logic-based framework for reasoning about composite data structures. In *Proc 20th CONCUR*, volume 5710 of *LNCS*, pages 178–195. Springer, 2009.
5. A. R. Bradley, Z. Manna, and H. B. Sipma. What's decidable about arrays? In *Proc. of 7th VMCAI*, volume 3855 of *LNCS*, pages 427–442. Springer, 2006.
6. J. H. Gallier. *Logic for Computer Science: Foundations of Automatic Theorem Proving.* Harper & Row, 1986.
7. L. Holík, O. Lengál, A. Rogalewicz, J. Simácek, and T. Vojnar. Fully automated shape analysis based on forest automata. In *Proc. of 25th CAV*, volume 8044, pages 740–755, 2013.
8. V. Kuncak, H. H. Nguyen, and M. C. Rinard. An algorithm for deciding BAPA: Boolean Algebra with Presburger Arithmetic. In *Proc. of 20th CADE*, volume 3632 of *LNCS*, pages 260–277. Springer, 2005.
9. S. K. Lahiri and S. Qadeer. Back to the future: revisiting precise program verification using SMT solvers. In *Proc. of POPL'08*, pages 171–182. ACM, 2008.
10. Z. Manna and A. Pnueli. *Temporal Verif. of Reactive Systems.* Springer, 1995.
11. J. McCarthy. Towards a mathematical science of computation. In *IFIP Congress*, pages 21–28, 1962.
12. G. Nelson and D. C. Oppen. Simplification by cooperating decision procedures. *ACM Trans. Program. Lang. Syst.*, 1(2):245–257, 1979.
13. W. Pugh. Skip lists: A probabilistic alternative to balanced trees. *Commun. ACM*, 33(6):668–676, 1990.
14. S. Ranise and C. G. Zarba. A theory of singly-linked lists and its extensible decision procedure. In *Proc. of SEFM 2006*. IEEE CS Press, 2006.
15. J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proc. of LICS'02*, pages 55–74. IEEE CS Press, 2002.
16. A. Sánchez and C. Sánchez. Decision procedures for the temporal verification of concurrent lists. In *Proc. of ICFEM'10*, volume 6447 of *LNCS*, pages 74–89. Springer, 2010.
17. A. Sánchez and C. Sánchez. A theory of skiplists with applications to the verification of concurrent datatypes. In *Proc. of NFM 2011*, volume 6617 of *LNCS*. Springer, 2011.
18. A. Sánchez and C. Sánchez. Parametrized invariance for infinite state processes. *CoRR*, abs/1312.4043, 2013.
19. T. Wies, V. Kuncak, K. Zee, A. Podelski, and M. Rinard. Verifying complex properties using symbolic shape analysis. In *Workshop on heap abstraction and verification (collocated with ETAPS)*, 2007.
20. G. Yorsh, A. M. Rabinovich, M. Sagiv, A. Meyer, and A. Bouajjani. A logic of reachable patterns in linked data-structures. In *Proc. of FOSSACS'06*, volume 3921 of *LNCS*, pages 94–110. Springer, 2006.