Heap Decomposition for Concurrent Shape Analysis

Tel Aviv University, School of Computer Science, Technical Report TR-2007-11-85453

R. Manevich[1,⋆], T. Lev-Ami[1,⋆⋆], M. Sagiv[1], G. Ramalingam[2], and J. Berdine[3]

[1] Tel Aviv University, {rumster,msagiv,tla}@post.tau.ac.il
[2] Microsoft Research India, grama@microsoft.com
[3] Microsoft Research Cambridge, jjb@microsoft.com

Abstract. We demonstrate shape analyses that can achieve a state space reduction exponential in the number of threads compared to the state-of-the-art analyses, while retaining sufficient precision to verify sophisticated properties such as linearizability. The key idea is to abstract the global heap by decomposing it into (not necessarily disjoint) subheaps, abstracting away some correlations between them. These new shape analyses are instances of an analysis framework based on heap decomposition. This framework allows rapid prototyping of complex static analyses by providing efficient abstract transformers given user-specified decomposition schemes. Initial experiments confirm the value of heap decomposition in scaling concurrent shape analyses.

1 Introduction

The problem of verifying concurrent programs that manipulate heap-allocated data structures is challenging: it requires considering arbitrarily interleaved threads manipulating unbounded data structures. Both heap-allocated data structures and concurrency can introduce state explosion. Their combination only makes matters worse. This paper develops new static analysis algorithms that address the state space explosion problem in a systematic and generic way. The result of these analyses can be used to automatically establish interesting properties of concurrent heap-manipulating programs such as the absence of null dereferences, the absence of memory leaks, the preservation of data structure invariants, and *linearizability* [10].

The Intuition. Typical programs manipulate a large number of (instances of) data structures (possibly nested within other data structures). Each individual data structure can usually be in one of several different states (even in an abstract representation). This can lead to a combinatorial explosion in the number of distinct abstract states that can arise during abstract interpretation.

The essential idea we pursue is that of *decomposing* the heap into multiple subheaps and abstracting away some correlations between the subheaps. Decomposition allows reusing subheaps that were decomposed from different heaps, thus representing a set of

⋆ This research was partially supported by the Clore Fellowship Programme.
⋆⋆ Supported by an Adams Fellowship through the Israel Academy of Sciences and Humanities.

heaps more compactly (and more abstractly). For example, consider a program maintaining k disjoint lists. A powerset-based shape analysis such as the one in [20] uses a lattice whose height is exponential in k. An abstraction that ignores the correlations between the k lists reduces the lattice height to be linear in k, leading to exponentially faster analysis. (The savings come from not maintaining the correlations between different states of the different lists, which we observe are often irrelevant for a specific property of interest.) Similar situations arise in the kind of multithreaded programs discussed earlier, where the size of the state space is a function of the number of threads rather than the number of data structures. In this paper, we allow decomposing the heap into non-disjoint (i.e., overlapping) subheaps, which is important for handling programs with fine-grained concurrency (where different threads can simultaneously access the same objects) in a thread-modular way.

Fine-Grained Concurrency. Fine-grained concurrent heap-manipulating programs allow multiple threads to use the same data structure *simultaneously*. They trade the simplicity of the single-thread-owning-a-data-structure model, which is at the heart of the coarse-grained concurrency approach, to achieve a higher degree of concurrency. However, the additional performance comes with a price: these programs are notoriously hard to develop and prove correct, even when the manipulated data structures are singly-linked lists (see, e.g., [5]).

It is hard to employ thread-modular approaches that exploit locking [8] to analyze fine-grained concurrent programs because they have *intentional* (benign) data-races. Thus, state-of-the-art shape analyses capable of verifying intricate properties of fine-grained concurrent heap-manipulating programs, e.g., linearizability (explained in Sec. 3), track all correlations between the states of all the threads [1]. This makes these analyses hard to scale. For example, the shape analysis in [1] handles at most 3 threads.

It is interesting to observe, however, that it is often the case that although proving properties of these programs requires tracking sophisticated correlations between every thread and the part of the heap that it manipulates, the correlations between the states of different threads is often irrelevant. Intuitively, this is because fine-grained concurrent programs are often written in a way which *attempts* to ensure the correct operation of every thread *regardless* of the actions taken by other threads. This programming paradigm makes these programs an ideal match with our approach explained below.

The Conceptual Framework. To permit the use of heap decomposition in several settings, we first present it as a parametric abstraction that can be tuned by the analysis designer in three ways:

Decomposition: Specify along what lines a concrete heap should be decomposed into (possibly overlapping) subheaps. One of the strengths of the specification mechanism is that the decomposition of a heap depends on its properties. This allows us, for example, to decompose the state of a concurrent program based on the association between threads and data-structures in that state, which is usually not known a priori.

Subheap abstraction: Create a bounded abstract heap representation from concrete subheaps (which are unbounded). Subheap abstractions can be obtained from existing whole-heap abstractions that satisfy certain properties.

Combiner Sets: The framework is parametric with respect to transformers. Computing sound and precise transformers for statements is quite challenging with a heap decomposition. Transforming each subheap independently can end up being very imprecise (or potentially incorrect, if not done carefully), especially when subheaps overlap. At the other extreme, combining subheaps together into a full heap prior to transforming it can be very inefficient and defeats the purpose of using heap decomposition. Achieving the desired precision and efficiency, without compromising soundness, can be tricky. Our framework allows the analysis designer to specify only which subheaps should be combined together for a given transformer, called combiner sets. The framework automatically generates a corresponding sound transformer, letting the analysis designer easily explore alternatives without worrying about soundness.

HeDec. We implemented our conceptual framework for the family of canonical abstractions [20] in a system called HeDec (for **He**ap **Dec**omposition), which is publicly available. This implementation retains the parametricity of the conceptual framework, which allows analysis designers to rapidly prototype different shape analysis algorithms by defining heap decomposition schemes.

Instances of the Framework. We have used our framework to develop several shape analyses, including the following, and have implemented these analyses in HeDec.

(a) A shape analysis for sequential programs manipulating singly-linked lists that abstracts away the correlations between disjoint lists . The resultant shape analysis algorithm emulates the algorithm of [13], with some interpretative overhead. Unlike the tedious proof of soundness of [13], the soundness of this instance immediately follows from the soundness of the underlying subheap abstraction.

(b) A new shape analysis for sequential programs manipulating singly-linked lists and trees by abstracting away the correlations between segments which do not contain an element pointed-to by a variable. We confirmed that it is precise enough to prove memory safety and preservation of data-structure invariants. This is encouraging for scaling shape analysis for programs with densely connected heaps.

(c) A shape analysis for fine-grained concurrent programs with a bounded number of threads which is precise enough to prove memory safety and preservation of data-structure invariants. Here, we obtain exponential speed-up in terms of time and space, in comparison to similar whole-heap analysis without decomposition. Our algorithm goes beyond [8] by supporting fine-grained concurrency and handling programs with intentional data races.

(d) A shape analysis algorithm for concurrent programs with a bounded number of threads that manipulate singly-linked lists, which proves linearizability. The resultant algorithm is exponentially faster than the one in [1], being polynomial in the number of threads. Our initial empirical results confirm that our algorithm is able to prove linearizability with 20 threads, ten times more than in [1].

Main Results. The contributions of this paper can be summarized as follows:

1. We present a generic analysis framework (in an abstract interpretation setting) for exploiting state decomposition effectively. The main technical contributions are in

introducing a family of sound abstract transformers that admit flexibly exploring the efficiency/precision spectrum.
2. We propose scalable analyses for several interesting problems involving coarse-grained as well as fine-grained concurrency, including proving linearizability. These algorithms scale much better (e.g., polynomially) over the number of threads than the previous algorithms for these problems.
3. The implementation of the framework for canonical abstraction is publicly available, together with the above mentioned analyses, as well as other benchmarks, which show the benefit of the approach.

Outline of the Paper. In Sec. 2, we demonstrate heap decomposition for fine-grained concurrent programs. In Sec. 3, we describe an analysis based on heap decomposition for proving linearizability of non-blocking data structures. In Sec. 4 we present the technical details of our abstract domain and its transformers. In Sec. 5 we report on our experiments with HeDec. In Sec. 6, we discuss related work, and in Sec. 7, we conclude the paper. In App. A, we formally describe decomposition applied to concrete states. The reader is referred to [14] for technical details on the instances of the framework and proofs of soundness for arbitrary decompositions. In App. C, we describe optimizations implemented in HeDec. In App. **??**, we demonstrate heap decomposition on an example program with coarse-grained concurrency.

2 Heap Decomposition for Fine-Grained Concurrency

In this section, we develop decomposition schemes for performing shape analysis of fine-grained concurrent programs and show that HeDec can be used to automatically obtain shape analysis implementations that are precise enough to prove the desired properties of programs (the absence of null pointer dereferences, absence of memory leaks, and data structure invariants) while scaling up to a large number of threads. The material in this section is presented informally, deferring formal definitions and technical details to Sec. 4.

2.1 Decomposing Non-blocking Implementations

A Running Example. Fig. 1 shows a simple running example of a non-blocking stack implementation from [21]. Producers push elements onto the stack by allocating an element, copying the current global pointer to the top of the stack, connecting the new element to that copied top, and then using CAS (**C**ompare **A**nd **S**wap) to atomically check that the top of the stack has not changed and replace it with the new element. Consumers pop elements from the stack by copying the current global pointer to top and recording its next element and then using CAS to atomically check that the top of the stack has not changed and replace it with the new top, i.e., the recorded next element. In both cases, a failed CAS results in a restart.

The goal here is to prove the absence of null pointer dereferences, absence of memory leaks, and the preservation of data structure invariants, i.e., that `stack` points to an acyclic list.

```
#define EMPTY -1
typedef int data_type;
typedef struct node t {
      data_type d;
      struct node t *n
} Node;
typedef struct stack t {
      struct node t *Top;
} Stack;

[1]  void push(Stack *S, data_type v){
[2]    Node *x = alloc(sizeof(Node));
[3]    x->d = v;
[4]    do {
[5]      Node *t = S->Top;
[6]      x->n = t;
[7]    } while (!CAS(&S->Top,t,x));
[8]  }
```

```
[9]  data_type pop(Stack *S){
[10]   do {
[11]     Node *t = S->Top;
[12]     if (t == NULL)
[13]       return EMPTY;
[14]     Node *s = t->n;
[15]     data_type r = t->d;
[16]   } while (!CAS(&S->Top,t,s));
[17]   return r;
[18] }
```

Fig. 1. A non-blocking stack implementation

Concrete Execution. Fig. 2(a) shows an example of two states occurring in the non-blocking implementation shown in Fig. 1; for now ignore the $corr$ annotations (which is used by the linearizability analysis in the next section). The figure shows two consumer threads and two producer threads. Both **cons1** and **prod1** can succeed with the CAS if they are the next threads to be scheduled. Concrete states are depicted by graphs. To avoid clutter the `data` field is not shown. Hexagonal nodes denote thread objects and square nodes denote list elements. The program label of every thread is written inside the hexagon. Edges from text labels to nodes correspond to global pointers (`Top`). Labeled edges from thread nodes to list nodes denote thread-local pointer variables (`t` and `x`). Edges between list nodes, labeled by `n` correspond to the `next` field of the list.

Exponential State Space. There are several sources of exponential explosion in the state space exploration of the stack algorithm. The first one is the correlation between the program locations of the different threads. The second source is the next pointers of the just allocated elements. The stack can grow after the next pointer has already been set, but before the CAS, thus the next pointers of the different producers can point to all possible stack elements and have all possible aliasing between each other. The third source of state-space explosion is the recorded next pointer of the consumer threads. Note that the state space explosion occurs even if the list has a bounded number of elements. This is a general problem when maintaining correlations between the properties of different threads. Exponential blow-ups also occur in sequential programs because of aliasing. However, for the purpose of our analysis, these correlations are unimportant and tracking them is pointless and only reduces the efficiency of the analysis.

Heap Decomposition Abstraction. We reduce the size of the state space by decomposing the heap into a set (or tuple) of subheaps and abstractly interpreting the program over the subheaps.

For each subheap to be used in the decomposition, a user of HeDec specifies the part of the heap it should include. This is done by defining a *location selection predicate*, which specifies the subset of the nodes in the state for which abstract properties (such as aliasing, heap-reachability, etc.) are maintained. For each location selection predicate, the program state is projected onto the nodes satisfying that predicate, thus obtaining a *substate* of the original state. We refer to the domain of substates pertaining to a location selection predicate pt as the *subdomain* of pt.

The Decomposition Scheme. For the purpose of our analysis, we define for each thread t the location selection predicate *pt*$[t]$ that holds for: (a) the thread object of t, (b) the objects pointed-to by its local variables (`t` and `x`), and (c) the objects pointed-to by the global variables (`Top`). In addition, we define the location selection predicate *Globals*, which holds for the objects reachable from global variables.

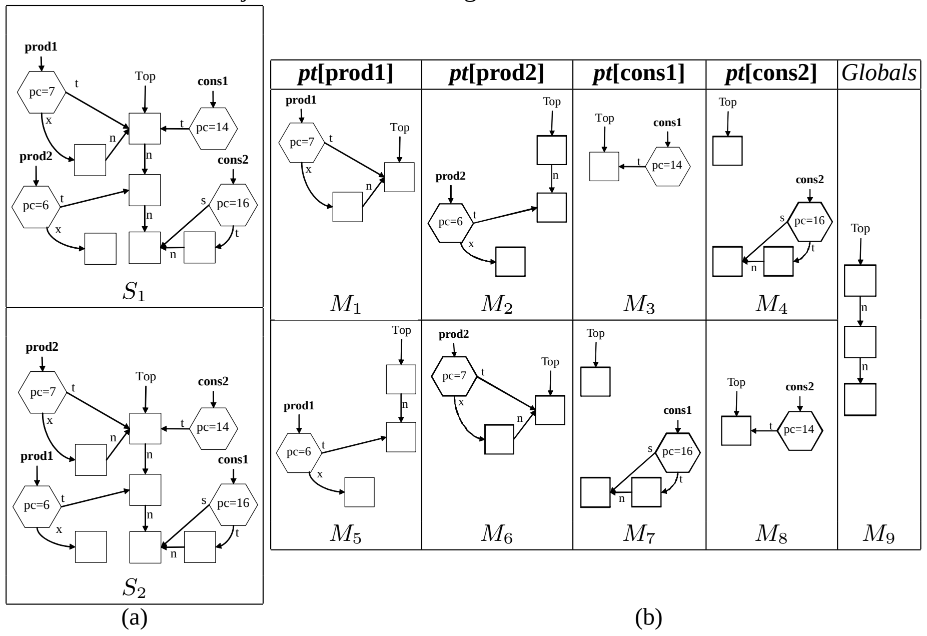


Fig. 2. (a) Two concrete states in the non-blocking stack implementation shown in Fig. 1; and (b) The decomposed states abstracting the full states in (a). The names of the sub-domains appear above the substates

Fig. 2(b) shows the result of applying the decomposition scheme explained above to the states in Fig. 2(a). Notice that different location selection predicates may occasionally overlap. For example, in the decomposition explained above, the objects reachable from the global variables appear in each subheap.

Intuitively, the meaning of a substate M, decomposed by a location selection predicate $p(v)$, is the set of all full states that contain M and any disjoint substate M', such that the objects in M satisfy $p(v)$ and the objects in M' do not satisfy $p(v)$. A sequence of sets of substates $\{M_1, M_5\} \times \{M_2, M_6\} \times \{M_3, M_7\} \times \{M_4, M_8\} \times \{M_9\}$ represents the set of full states obtained by choosing one structure from each subdomain and intersecting their meanings. For example, composing the substates $\{M_1, M_2, M_3, M_4, M_9\}$ together yields S_1 and composing the substates $\{M_5, M_6, M_7, M_8, M_9\}$ together yields S_2. The loss of precision by the abstraction can be observed by the fact that other compositions, such as $\{M_1$, M_6, M_7, M_8, $M_9\}$ yield full states other than S_1 and S_2.

State Space Savings. In general, for n threads, if the set of objects reachable from a thread is bounded, then the number of substates resulting from the reachability-based decomposition is linear in n (even though the number of full states generated by the program is exponential in n). Although we do not show the state space reduction in the figures, one can imagine how running the program with n threads generates states similar to the ones in Fig. 2(a). By permuting the thread ids between producers threads and between consumer threads, we obtain an exponential number of full states that are all reachable by the program execution. Decomposing these states results in a number of substates that is linear in n.

Transformers. HeDec is guaranteed to be sound, in the sense that when the analysis terminates all reachable concrete states are represented by some abstract state.

While the abstraction ignores correlations between substates, transforming substates in isolation using an "independent-attribute" style of analysis [17] leads to debilitating loss of precision. For example, the analysis executes the statement `6: x->n=t` where thread **prod1** is scheduled. Substate M_3 does not contain information about the local variables of thread **prod1**. Therefore, M_3 also represents a state S_{bad} in which the local variables `t` and `x` of thread **prod1** point to the first cell and to the last cell of the list, respectively. Thus, a conservative transformer of `6: x->n=t` must emit a warning about a possible creation of a cyclic list.

To avoid this kind of loss of precision, a user of HeDec can specify which substates, obtained from different location selection predicates, should be (temporarily) composed by the transformer. This is done in terms of *combiner sets*, which are subsets of node selection predicates. In this example, for the transformer of `6: x->n=t`, we can specify the combiner sets $\{pt[\textbf{prod1}], pt[\textbf{prod2}]\}$, $\{pt[\textbf{prod1}], pt[\textbf{cons1}]\}$, $\{pt[\textbf{prod1}], pt[\textbf{cons2}]\}$, and $\{pt[\textbf{prod1}], pt[Globals]\}$. Then, the generated transformer composes, separately, the substates $\{M_1, M_5\}$ with each of the sets of substates $\{M_2, M_6\}$, $\{M_3, M_7\}$, $\{M_4, M_8\}$, and $\{M_9\}$. For the substates composed with M_5 (which is the only substate in the **prod1**-subdomain that can execute `6: x->n=t`) the transformer updates the `n` field appropriately, avoiding the false alarm. Finally, the transformer decomposes the substates again into each one of the subdomains. The resulting abstract substates are the same as in Fig. 2, except that M_5 has an `n`-link between the object pointed-to by `t` and the object pointed-to by `x` and its program counter is 7.

This example shows how, by combining a small number (linear in the number of location selection predicates, in this case) of substates decomposed by different predicates, the transformer is able to increase precision without incurring an unreasonable time/space blow-up.

A Methodology for Combiner Sets. We now briefly discuss the issue of choosing combiner sets for a transformer (which is done by the analysis designer in our framework). Every transformer can be thought of as having a *frame* as well as a *footprint*. The frame identifies the part of a program state that is completely irrelevant to the transformer. Thus, it contains no information that is either used or modified by the transformer. The footprint is the complement and contains adequate information to perform the transformer as precisely as possible.

A straightforward approach for computing the footprint of an operation affecting several subdomains is combining all the affected subdomains. Unfortunately, this approach might be too expensive. We apply a more efficient approach, which according to our experience is precise enough. Specifically, for each operation we choose a set of *core subdomains* which contain the heap objects and variables that participate in the operation. We compute the *core footprint* by combining the core subdomains (in practice, there are usually no more than two). We then independently combine the core footprint with the other affected subdomains. For example, the core subdomains for a statement of the form "`x->f = g`", where `x` of thread t is a local variable and `g` is a global variable, are the subdomains containing thread t and the subdomain of the global variable `g`. The affected subdomains are any subdomains which may alias these variables.

Conditional branches pose an interesting puzzle. Note that because the condition essentially filters states it can affect *all* subdomains. Thus, for a conditional "`if (x == g)`", we identify the core subdomains to be the ones containing (the nodes pointed-to by) `x` and `g`. However, we will independently combine them with all other subdomains.

3 Using Decomposition to Prove Linearizability

Linearizability [10] is one of the main correctness criteria for implementations of concurrent data structures. Informally, a concurrent data structure is said to be linearizable if the concurrent execution of a set of operations on it is equivalent to some sequential execution of the same operations, in which the global order between non-overlapping operations is preserved. The equivalence is based on comparing the arguments and results of operations (responses). The permitted behavior of the concurrent object is defined in terms of a specification of the desired behavior of the object in a sequential setting. Linearizability is a widely-used concept, and there are numerous non-automatic proofs of linearizability for concurrent objects.

Verifying linearizability is challenging because it requires correlating any concurrent execution with a corresponding permitted sequential execution. Verifying linearizability for concurrent dynamically allocated linked data structures is particularly challenging, because it requires correlating executions that may manipulate memory states of unbounded size. Interestingly, proving linearizability does not require directly proving safety properties such as preservation of data structure invariants. Instead, one can first prove that the sequential implementation satisfies the required safety properties and then prove that the concurrent implementation is linearizable, thereby, satisfies the safety property. Finally, linearizability of complex systems can be shown by separately proving the linearizability of each of the individual data structure implementations.

Intuitively, we verify linearizability by representing, in the concrete state, both the state of the concurrent program and the state of the reference sequential program. Each element entered into the data structure is correlated at linearization points with the matching object from the sequential execution. This works well under abstraction when the differences between the heaps of the sequential and concurrent implementations are bounded. The details are described in [1].

In order to guarantee that the shape analysis scales-up in the number of threads, in HeDec we have defined a decomposition scheme that abstracts away the correlations between the threads (as in Sec. 2). Also, there is no need to track reachability from program variables. Instead, the subheap abstraction tracks elements whose values in the sequential and the concurrent implementations are correlated.

3.1 A Decomposition Scheme for Linearizability Analysis

In HeDec, we have defined such a decomposition scheme by decomposing the heap into $n + 1$ components where n is the number of threads: (i) For each thread the objects pointed-to by local variables of the thread and objects pointed-to by global variables. This captures the relationships between local pointer variables and global pointer variables. Each subheap abstracts away the values of the local variables of the other threads. (ii) A separate subheap with the objects pointed-to by global variables and the part of the heap already correlated with the sequential execution. Here, the values of the local variables of all the threads are abstracted away. We call this the *corr* subdomain as it represents the correlated elements. Fig. 3 shows the effect of applying this decomposition to the full state S_1 in Fig. 2(a).

Intuitively, this decomposition is appropriate for verifying linearizability for the program in Fig. 1 because of the following. The list consisting of correlated objects changes locally when a thread executes a successful `CAS` operation. In fact, successful `CAS` operations are the linearization points for this program. Precisely interpreting these operations (`CAS(&S->Top,t,x)` and `CAS(&S->Top,t,s)`) in the analysis requires tracking correlations between local and global variables, which we do in the subheap we decompose for each thread.

The subheap captured by the *corr* subdomain is important only during successful `CAS` operations, which is when a (non-correlated) node allocated by a thread is passed into the list. Maintaining the subheap of the *corr* subdomain for each thread is wasteful, and thus we separate these correlations into different subdomains.

The important thing to notice is that all the exponential explosion in the state space that is due to the number of threads in the full heap is eliminated by this decomposition. The number of possible subheaps of each thread becomes independent of the number of threads in the system (for more than two threads).

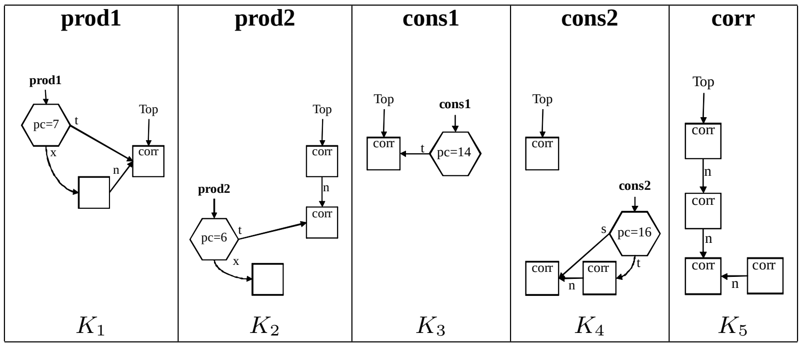


Fig. 3. The decomposed states abstracting the full state S_1 in Fig. 2(a). The names of the subdomains appear above each substate

Transformers. The combiner sets used in the transformers of the analysis are the application of the methodology described in Sec. 2.1 to this decomposition scheme. For example, copying a global variable into a local variable does not require decomposition as the executing thread has all the needed information. Copying a local variable into a global variable combines the subdomain of the executing thread with each of the other subdomains. Other operations that change the global state such as changes to pointer fields and performing CAS operations behave the same. Dereferencing a pointer requires composing the subdomain for the current thread and the *corr* subdomain as the information on the next element of the stack is not available in the thread's subdomain.

4 The Heap Decomposition Abstraction

In this section, we formally define our new parametric heap abstraction and a family of sound abstract transformers.

4.1 Heap Decomposition as a Cartesian Product of Subheaps

We first define the (parameterized) abstract domain of decomposed heaps. (See Appendix A for an illustration of the concepts defined below.)

Let $(\Sigma, \preceq, \otimes)$ be a semilattice, where elements of Σ represent (total and partial) states, $\preceq$ is a partial ordering on Σ capturing the "is a substate of" relation, and $\otimes$ is the join operation with respect to $\preceq$ (which composes substates together). We extend $\otimes$ to sets of states as follows. Let $X_1 \subseteq \Sigma$ and $X_2 \subseteq \Sigma$. We define $X_1 \otimes X_2 = \{\sigma_1 \otimes \sigma_2 \mid \sigma_1 \in X_1, \sigma_2 \in X_2\}$. For purposes of abstraction, we shall also make use of the information ordering defined by $\sigma \sqsubseteq \sigma'$ iff $\sigma' \preceq \sigma$.

Let $(\mathcal{P}(\Sigma), \sqsubseteq)$ denote the powerset domain of Σ with the Hoare ordering: i.e., for every $X, Y \subseteq \Sigma$, we write $X \sqsubseteq Y$ iff $\forall x \in X : \exists y \in Y : x \sqsubseteq y$.

A *substate extraction* function is a function $\eta : \Sigma \to \Sigma$ that satisfies $\eta(\sigma) \preceq \sigma$. Assume we have a sequence of k substate extraction functions η_1 to η_k. We use the k-fold product $\mathcal{P}(\Sigma)^k = \mathcal{P}(\Sigma) \times \cdots \times \mathcal{P}(\Sigma)$ as our domain of abstract states. The abstraction function $\alpha : \mathcal{P}(\Sigma) \to \mathcal{P}(\Sigma)^k$ is defined by:

$$\alpha(S) = (\hat{\eta}_1(S), \ldots, \hat{\eta}_k(S)) \tag{1}$$

where $\hat{\eta}_i$ is the pointwise extension of η_i defined by:

$$\hat{\eta}_i(S) = \{\eta_i(\sigma) \mid \sigma \in S\} \tag{2}$$

We define the meaning, or *concretization*, of a tuple $I_1, \ldots, I_k \in \mathcal{P}(\Sigma)^k$ by

$$\gamma(I_1, \ldots, I_k) = I_1 \otimes \cdots \otimes I_k. \tag{3}$$

Example 1. Let S denote the set of states $\{S_1, S_2\}$ shown in Fig. 2(a). For any thread t, we define the predicate *pt*$[t]$ to be true for: (a) the thread object of t, (b) the objects pointed-to by its local variables (`t` and `x`), and (c) the objects pointed-to by the global variables (`Top`). In addition, we define the location selection predicate *Globals*, which holds for the objects reachable from global variables. Given any predicate p, the substate extraction function δ_p maps a state σ to the substate consisting only of the locations satisfying p. We define η_1 to be $\delta_{pt[\textbf{prod1}]}$, η_2 to be $\delta_{pt[\textbf{prod2}]}$, η_3 to be $\delta_{pt[\textbf{cons1}]}$, η_4 to be $\delta_{pt[\textbf{cons2}]}$, and η_5 to be $\delta_{Globals}$. Now, $\eta_1(S_1) = M_1$, $\eta_2(S_1) = M_2$, $\eta_3(S_1) = M_3$, $\eta_4(S_1) = M_4$, and $\eta_5(S_1) = M_9$.

4.2 Abstract Transformers

We now turn our attention to the more challenging aspect of decomposition: computing sound abstract transformers.

The semantics of a program statement is given by a function $\tau : \Sigma \to \mathcal{P}(\Sigma)$. We make the standard assumption that the transformer is monotonic in the information order, i.e., if $\sigma_1 \sqsubseteq \sigma_2$ then $\tau(\sigma_1) \sqsubseteq \tau(\sigma_2)$. We extend this function pointwise to $\tau : \mathcal{P}(\Sigma) \to \mathcal{P}(\Sigma)$, by defining $\tau(S) = \bigcup\{\tau(\sigma) \mid \sigma \in S\}$. (Note that the extended transformer is monotone in the information order as well.) For purposes of abstract interpretation, we need to define a corresponding sound abstract transformer on $\mathcal{P}(\Sigma)^k$. Given an input value $I = (I_1, \dots, I_k)$, the abstract transformer needs to compute the output value $O = (O_1, \dots, O_k)$.

A straightforward sound transformer is the pointwise transformer τ^{pw} defined as follows:

$$\tau^{pw}(I_1, \dots, I_k) = (\hat{\eta}_1(\tau(I_1)), \dots, \hat{\eta}_k(\tau(I_k))). \tag{4}$$

Proposition 41 *The pointwise transformer τ^{pw} is sound. That is, for every input value $I = (I_1, \dots, I_k)$ where $I \in \mathcal{P}(\Sigma)^k$, the following holds:*

$$\tau(\gamma(I)) \sqsubseteq \gamma(\tau^{pw}(I)) \ . \tag{5}$$

Proof: Note that since $\preceq$ and $\sqsubseteq$ are reversed, $\otimes$ is a meet (i.e., greatest lower bound) operator on $\mathcal{P}(\Sigma)$. Let j be any index in $\{1, \dots, k\}$.

$$I_1 \otimes \dots \otimes I_k \sqsubseteq I_j \tag{6}$$

$\triangleright$ $\otimes$ is a meet operator

$$\gamma(I) \sqsubseteq I_j \tag{7}$$

$\triangleright$ by (6) and (3)

$$\tau(\gamma(I)) \sqsubseteq \tau(I_j) \tag{8}$$

$\triangleright$ τ is monotone

$$\tau(\gamma(I)) \sqsubseteq \hat{\eta}_j(\tau(I_j)) \tag{9}$$

$\triangleright$ by (8) and since $\hat{\eta}_j$ is extensive

$$\tau(\gamma(I)) \sqsubseteq \hat{\eta}_1(\tau(I_1)) \otimes \dots \otimes \hat{\eta}_k(\tau(I_k)) \tag{10}$$

$\triangleright$ $\otimes$ is a meet operator

$$\tau(\gamma(I)) \sqsubseteq \gamma(\hat{\eta}_1(\tau(I_1)), \dots, \hat{\eta}_1(\tau(I_k)) \tag{11}$$

$\triangleright$ by (3) and (10)

$$\tau(\gamma(I)) \sqsubseteq \gamma(\tau^{pw}(I)) \tag{12}$$

$\triangleright$ by (4) and (11)

□

Example 2. While the pointwise transformer is simple and efficient, it can lead to imprecise results when the transformer has to update a substate that does not have all the relevant information. Recall the example from Sec. 2, and consider the substate M_3. Substate M_3 does not contain information about the local variables of other threads. Therefore, M_3 also represents a state S_{bad} in which the local variables `t` and `x` of thread **prod1** point to the first cell and to the last cell of the list, respectively. Thus, a conservative transformer of `6: x->n=t`, when **prod1** serves as the scheduled thread, must emit a warning about a possible creation of a cyclic list. As explained in Sec. 2, we can avoid this imprecision by composing substate M_3 with other substates (M_1) to produce a more precise substate that can be transformed without making such worst-case assumptions. This motivates the following definitions.

A *combiner set* is a set $R \subseteq \{1, \ldots, k\}$ identifying a set of subheap domains. We define the *partial concretization function* γ_R, which combines the information from the specified set of subdomains $R = \{j_1, \ldots, j_m\}$, as follows:

$$\gamma_R(I_1, \ldots, I_k) = \bigotimes_{r \in R} I_r = I_{j_1} \otimes I_{j_2} \cdots \otimes I_{j_m} \quad . \tag{13}$$

One-Level Composition. We define the *partial transformer* $\tau_1[R, i]$, which computes the substate corresponding to the i-th subdomain using the subdomains identified by R, by

$$\tau_1[R, i](I) = \hat{\eta}_i(\tau(\gamma_R(I))). \tag{14}$$

We use the term *one-level* transformer to indicate that combining (or composing) information from a set of subdomains (identified by R above) occurs in one step.

We define a *one-level transformer specification* $\textsc{ts}$ to be a tuple $(\textsc{ts}_1, \ldots, \textsc{ts}_k)$ where each $\textsc{ts}_i \subseteq \{1, \ldots, k\}$. We define the transformer $\tau_1[\textsc{ts}]$ by

$$\tau_1[\textsc{ts}](I) = (\tau_1[\textsc{ts}_1, 1](I), \ldots, \tau_1[\textsc{ts}_k, k](I)). \tag{15}$$

Theorem 1. *For any one-level transformer specification $\textsc{ts}$, the transformer $\tau_1[\textsc{ts}]$ is sound. That is, for every input value $I \in \mathcal{P}(\Sigma)^k$: $\tau(\gamma(I)) \sqsubseteq \gamma(\tau_1[\textsc{ts}](I))$.*

Theorem 2. *Let $\textsc{ts} = (\textsc{ts}_1, \ldots, \textsc{ts}_k)$ where each $\textsc{ts}_i \subseteq \{1, \ldots, k\}$ be a one-level transformer specification. Then, the one-level transformer $\tau_1[\textsc{ts}]$ is sound. That is, for every input value $I \in \mathcal{P}(\Sigma)^k$, the following holds:*

$$\tau(\gamma(I)) \sqsubseteq \gamma(\tau_1[\textsc{ts}](I)) \quad . \tag{16}$$

Two-Level Composition. We now present a generalization of the above definition. As motivation for this generalization, consider a situation where we want to compute an output value O_j by combining the input values from a set of subdomains R_1 or by combining the input values from a set of subdomains R_2 (but we are unable to say which of these combinations to use statically). We could, of course, combine the input values from the set of subdomains $R_1 \cup R_2$, but this could be expensive. Instead, we can utilize the two combinations *independently* of each other by using

$$(\hat{\eta}_j(\tau(\gamma_{R_1}(I)))) \sqcap (\hat{\eta}_j(\tau(\gamma_{R_2}(I))))$$

as the desired output value. We call transformers derived in this fashion two-level transformers, as the use of the meet operation $\sqcap$ constitutes a second stage of combining (composing) information.

Let Y be a set of combiner sets. We define the *partial transformer* $\tau_2[Y, i]$, which computes the substate corresponding to the i-th subdomain using the combiner sets in Y independently, as follows:

$$\tau_2[Y, i](I) = \prod_{R \in Y} \tau_1[R, i](I) \tag{17}$$

We define a *two-level transformer specification* $\textsc{ts}$ to be a tuple $(\textsc{ts}_1, \ldots, \textsc{ts}_k)$ where each $\textsc{ts}_i \subseteq \mathcal{P}(\{1, \ldots, k\})$. We define the transformer $\tau_2[\textsc{ts}]$ by

$$\tau_2[\textsc{ts}](I) = (\tau_2[\textsc{ts}_1, 1](I), \ldots, \tau_2[\textsc{ts}_k, k](I)). \tag{18}$$

(Note that the computation of the above transformer involves a partial concretization for every R in every $\textsc{ts}_i$. In practice, different $\textsc{ts}_i$ and $\textsc{ts}_j$ may have common elements, and it is sufficient for the transformer implementation to do the corresponding partial concretization just once.)

Theorem 3. *For any two-level transformer specification* $\textsc{ts}$*, the transformer* $\tau_2[\textsc{ts}]$ *is sound. That is, for every input value* $I \in \mathcal{P}(\Sigma)^k$*:* $\tau(\gamma(I)) \sqsubseteq \gamma(\tau_2[\textsc{ts}](I))$*.*

Theorem 4. *Let* $\textsc{ts} = (\textsc{ts}_1, \ldots, \textsc{ts}_k)$ *where each* $\textsc{ts}_i \subseteq 2^{\{1, \ldots, k\}}$ *be a two-level transformer specification. Then, the two-level transformer* $\tau_2[\textsc{ts}]$ *is sound. That is, for every input value* $I \in \mathcal{P}(\Sigma)^k$*, the following holds:*

$$\tau(\gamma(I)) \sqsubseteq \gamma(\tau_2[\textsc{ts}](I)) \ . \tag{19}$$

5 Empirical Results

We implemented the HeDec system in Java on top of the TVLA system [12]. HeDec allows analysis designers to rapidly prototype different shape analysis algorithms by defining heap decomposition schemes. HeDec, however, is not a panacea — the designer needs to carefully select suitable heap decompositions. Nevertheless, HeDec relieves the designer from the task of developing and implementing the static analysis algorithms, including the transformers.

Fig. 4 compares the results of our decomposition-based analysis with a full heap analysis.[4]

Concurrent Benchmarks. We use the analysis of [1] as the underlying shape analysis.

Both analyses successfully prove linearizability and absence of null dereferences for the three concurrent programs. For a given number of threads, t, the table shows the time and the number of states resulting in the analysis of t threads invoking an arbitrary sequence of operations on a single instance of the analyzed concurrent data

[4] All benchmarks except NBQ were run on a 2.4 GHz E6600 Core 2 Duo processor with 2 GB of memory running Linux.

structure. Stack is the non-blocking stack example of Sec. 2.1. TLQ is the two-lock queue implementation described in [16]. NBQ is a non-blocking queue implementation from [6]. [5]

Note that while [1] can analyze at most 3 threads, our approach, on the other hand,runs for 15 threads or more. Furthermore, [1] runs out of memory when analyzing 3 threads manipulating a non-blocking-queue.

Sequential Benchmarks. Both analyses successfully prove absence of null dereferences, absence of memory leaks, and data structure invariants for the following sequential benchmarks: 6`-list-prepend` adds elements, non-deterministically, into one of 6 lists; 6`-list-join` joins 6 lists into one list; and 4`-tree-insert` inserts nodes, non-deterministically, into one of 4 binary search trees.

		Full Heap		Decomposition	
Example	# of threads	# of states	secs.	# of substates	secs.
Stack	2	3,424	3	1,608	7
	3	10,6296	71	4,103	13
	4	MemOut	-	7,728	22
	20	-	-	212,048	3,421
TLQ	3	8,783	12	8,911	30
	5	44,285	35	23,585	90
	8	MemOut	-	58,796	307
	15	-	-	202,555	2,122
NBQ	2	39,583	69	20,646	263
	3	MemOut	-	57,065	694
	15	-	-	2,017,280	1 day

(a)

	Full Heap		Decomposition	
Example	# of states	secs.	# of substates	secs.
6-list-prepend	17,496	16	557	5
6-list-join	37,689	40	1,282	6
4-tree-insert	43,031	44	5,316	29

(b)

Fig. 4. Empirical results for: (a) concurrent benchmarks, and (b) sequential benchmarks

6 Related Work

The framework of Cartesian abstraction via state decomposition we have presented is relevant to a number of previous lines of work.

Heterogeneous Abstractions. Yahav and Ramalingam [25] defined a notion of heterogeneous abstractions. There, Cartesian abstractions are used as a way to achieve decomposition (or separation, in the terminology of that paper). One contribution of this paper is to show that that previous analysis is based on a (simple form of) Cartesian abstraction. On the other hand, in that work, heterogeneity was used only within a single structure (to abstract the substructure of interest differently from its context), where our framework supports different abstractions for different factors of the product, yielding heterogeneity across different structures. Furthermore, while [25] relies on the point-wise transformer, we introduce a generalized family of transformers that allow (de)composition when transformers are applied. This generalization allows specifying more precise transformers, and gives us dynamic separation/decomposition.

[5] This benchmark was run on a 2.66 GHz Quad Xeon with 16 GB of memory running Windows XP 64 bit.

Region-based Heap Analyses. Like [25], [9] also decomposes heap abstractions to independently analyze different parts of the heap. There the analysis/verification problem is itself decomposed into a set of problem instances, and the heap abstraction is specialized for each instance and consists of one subheap for the part of the heap relevant to the instance, and a coarser abstraction of the remaining part of the heap, e.g. a points-to graph. In contrast, we simultaneously maintain abstractions of different parts of the heap and also consider the interaction between these parts. (E.g., our decomposition dynamically changes as components get connected and disconnected.)

Local Transformers. The importance of modularity for the ability to compute transformers is well known. For example, the first proof rule for procedure calls, the *rule of adaptation*, was given in [11]. It allows reusing a proof of a procedure body in different invocations of the procedure.

Local reasoning [18, 19] enables reasoning about programs that alter heap-allocated data by combining claims about disjoints parts of the heap. The use of decomposition here is intuitively similar to that of separation in [18]. The chief difference is that here a decomposition may be used that is finer than the transformers in the underlying domain are precise for, which we react to by performing composition in the transformers. The transformers used in analyses based on separation logic [3], on the other hand, when applied to substates either produce exactly as precise information as on full states, or produce top. Our treatment of decomposition as an abstraction allows more flexibility in this regard. This flexibility is central to the concurrency analysis we presented: By not basing decomposition on disjointness, the analysis does not necessarily need to be thread-modular. In particular, we have the option of introducing predicates which track important correlations between different threads' local states. Approaches based on disjointness such as [8] have trouble with such situations unless auxiliary state is added to the invariants, which is beyond the ability of the existing automatic analyses.

Partially Disjunctive Heap Abstraction. Manevich et al. [15] describe a heap abstraction based on merging sets of graphs with the same set of nodes into one (approximate) graph. The abstraction in this paper is based on decomposing a graph into a set of subgraphs. The abstraction in [15] is orthogonal to the one in this paper.

Handling Concurrency for an Unbounded Number of Threads. In [2], we use thread quantification to analyze programs with an unbounded number of threads. Thread quantification can be thought of as an unbounded variant of a particular decomposition strategy, which we use to abstract away correlations between local variables of different threads. In the thread quantification analysis, we report that using an additional heap decomposition abstraction in order to abstract away correlations between values of some local variables and global variables effects drastic state-space savings. This made the analysis feasible in the example of proving linearizability of a non-blocking queue implementation.

Proving Linearizability of Data Structures. Shape analysis of concurrent programs with unbounded dynamic allocation have been investigated. The analysis in [24] addresses an unbounded number of threads by losing distinctions that cannot be made based on thread-independent information. This analysis has been extended to verify linearization

[1] of programs with a bounded number of threads. Here we use the decomposition abstraction to define an analysis that can be exponentially faster than that in [1].

Manual linearizability proofs using rely-guarantee have been given in [23], and using a manual translation to automata followed by an interactive proof in PVS in [4]. Recently, [22] automatically verifies linearizability from manual specifications in a combination of rely-guarantee and separation logic, using the proof technique of [1].

7 Conclusions

We present systematic and generic techniques for scaling up shape analyses using heap decomposition, implemented in the HeDec system. A user of HeDec can quickly prototype a shape analysis by: (a) defining any heap decomposition she believes is appropriate for the class of programs and properties of interest, and (b) supplying for every type of program statement any (possibly empty) combiner set she believes supplies the right balance between efficiency and precision. HeDec then automatically generates a sound analysis.

Acknowledgements. We thank Noam Rinetzky, Greta Yorsh, Byron Cook, and Thomas Ball for supplying us with helpful comments on early drafts of the paper. We thank Daphna Amit for explaining and helping us use her linearizability analysis and commenting on earlier drafts of this paper.

References

1. D. Amit, N. Rinetzky, T. Reps, M. Sagiv, and E. Yahav. Comparison under abstraction for verifying linearizability. In *CAV*, pages 477–490, 2007.
2. G. Arnold, R. Manevich, M. Sagiv, and R. Shaham. Combining shape analyses by intersecting abstractions. Lecture Notes in Computer Science, pages 33–48. Springer, January 2006.
3. J. Berdine, C. Calcagno, and P. W. O'Hearn. Symbolic execution with separation logic. In Kwangkeun Yi, editor, *APLAS 2005*, volume 3780 of *Lecture Notes in Computer Science*, pages 52–68. Springer-Verlag, 2005.
4. R. Colvin, S. Doherty, and L. Groves. Verifying concurrent data structures by simulation. *Electr. Notes Theor. Comput. Sci.*, 137(2):93–110, 2005.
5. S. Doherty, D. L. Detlefs, L. Groves, C. H. Flood, V. Luchangco, P. A. Martin, M. Moir, N. Shavit, and Jr. G. L. Steele. DCAS is not a silver bullet for nonblocking algorithm design. In *SPAA*, pages 216–224, 2004.
6. S. Doherty, L. Groves, V. Luchangco, and M. Moir. Formal verification of a practical lock-free queue algorithm. In *FORTE*, pages 97–114, 2004.
7. H. Eo and K. Yi. A differential fixoint iteration method for static analysis specifications. Technical Memorandum ROPAS-2004-21, Programming Research Laboratory, School of Computer Science & Engineering, Seoul National University, February 2004.
8. A. Gotsman, J. Berdine, B. Cook, and M. Sagiv. Thread-modular shape analysis. In *PLDI*, pages 266–277, 2007.
9. B. Hackett and R. Rugina. Region-based shape analysis with tracked locations. In *POPL*, pages 310–323, 2005.
10. M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *TOPLAS*, 12(3):463–492, 1990.

11. C. A. R. Hoare. Procedures and parameters: An axiomatic approach. *Lecture Notes in Mathematics*, 188:102–116, 1971.
12. T. Lev-Ami and M. Sagiv. TVLA: A framework for implementing static analyses. In *SAS*, pages 280–301, 2000.
13. R. Manevich, J. Berdine, B. Cook, G. Ramalingam, and M. Sagiv. Shape analysis by graph decomposition. In *TACAS*, pages 3–18, 2007.
14. R. Manevich, T. Lev-Ami, M. Sagiv, G. Ramalingam, and J. Berdine. Heap decomposition for concurrent shape analysis. Technical Report TR-2008-01-85453, Tel Aviv University, January 2008. Available at http://www.cs.tau.ac.il/∼rumster/TR-2007-11-85453.pdf.
15. R. Manevich, M. Sagiv, G. Ramalingam, and J. Field. Partially disjunctive heap abstraction. In *SAS*, pages 265–279, 2004.
16. M.M. Michael and M.L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *PODC*, pages 267–275, 1996.
17. F. Nielson, H. R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer, 1999.
18. P. O'Hearn, J. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. *Lecture Notes in Computer Science*, 2142, 2001.
19. J. Reynolds. Separation logic: a logic for shared mutable data structures, 2002.
20. M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems*, 24(3):217–298, 2002.
21. R. K. Treiber. Systems programming: Coping with parallelism. Technical Report RJ 5118, IBM Almaden Research Center, April 1986.
22. V. Vafeiadis. Shape-value abstraction for verifying linearizability. draft, 2008.
23. V. Vafeiadis, M. Herlihy, T. Hoare, and M. Shapiro. Proving correctness of highly-concurrent linearisable objects. In *PPOPP*, pages 129–136, 2006.
24. E. Yahav. Verifying safety properties of concurrent Java programs using 3-valued logic. *ACM SIGPLAN Notices*, 36(3):27–40, March 2001.
25. E. Yahav and G. Ramalingam. Verifying safety properties using separation and heterogeneous abstractions. In *PLDI*, pages 25–34, 2004.

A Heap Decomposition for Concrete Heaps

In this section, we illustrate decomposition, using the domain of concrete states (heaps). (The analyses discussed in the paper exploit decomposition by applying them to abstract heaps produced by Canonical Abstraction [20].)

A.1 Concrete Domain

We first define the set of concrete states, $\mathcal{H}$, using the notion of a two-level store. For simplicity, we focus on sequential program states, however, the generalization to multi-threaded program states is straightforward. We also restrict primitive values to be pointers. It is straightforward to extend the definitions to allow primitive values of other types (such as integers and booleans). As a result, we will use the terms state and heap interchangeably and do the same with the terms substate and subheap.

Let *Var* be a finite set of pointer variables and let *Field* be a finite set of object pointer fields. Let *locs* be a (potentially infinite) set of heap locations. Let *Loc*$^+$ denote the set *locs* $\cup \{null\}$, where *null* is a special value not in *locs*.

Definition A1 (Concrete States) *A concrete state σ is a pair $(env^\sigma, field^\sigma)$ where $env^\sigma : Var \rightharpoonup Loc^+$ is a partial mapping from variables to locations (or* null*) and $field^\sigma : Field \times locs \rightharpoonup Loc^+$ is a partial mapping that maps a pair consisting of a field name and a location to a location (or the special value* null*).*

Note that the set $\mathcal{H}$ includes *total states* (a state σ for which both env^σ and fld^σ are total functions) as well as *partial states* (states consisting of partial functions). Partial states are used to enable heap decomposition as illustrated now. In the sequel, we use $fld^\sigma(u)$ to stand for $field^\sigma(fld, u)$.

We define a partial ordering on concrete states as follows.

Definition A2 (Substate Ordering) *Let σ and σ' be two concrete states. We say that σ' is a substate of σ, written $\sigma' \preceq \sigma$, if (i) for every program variable $x \in Var$ if $env^{\sigma'}$ is defined for x then so is env^σ and $env^{\sigma'}(x) = env^\sigma(x)$; and (ii) for every field $fld \in Field$ and heap location $u \in locs$, if $field^{\sigma'}$ is defined for u then so is $field^\sigma$ and $field^{\sigma'}(u) = field^\sigma(u)$.*

Intuitively, $\sigma' \preceq \sigma$ means that σ' contains less information than σ about the values of variables and fields but agrees with σ on any information it does contain.

Let $\mathcal{H}_\top$ denote the set $\mathcal{H}$ augmented with a special element $\top$. We extend the relation $\preceq$ to $\mathcal{H}_\top$ by defining $\sigma \preceq \top$ for every $\sigma \in \mathcal{H}_\top$. Note that any two elements σ_1 and σ_2 of $\mathcal{H}_\top$ have a least upper bound with respect to $\preceq$, which we denote by $\sigma_1 \otimes \sigma_2$. Note that $\sigma_1 \otimes \sigma_2$ denotes the result of *composing* substates σ_1 and σ_2. We say that two susbtates σ_1 and σ_2 substates are *inconsistent* if there is no substate $\sigma' \in \mathcal{H}$ such that $\sigma_1 \preceq \sigma'$ and $\sigma_2 \preceq \sigma'$. In such a case, $\sigma_1 \otimes \sigma_2$ will be $\top$.

A.2 Subheap Extraction

One of the cornerstones of our heap decomposition abstraction is the notion of a *subheap extraction* function which, given a heap, extracts the part of the heap that is of interest. We illustrate this concept below.

We first present a subheap extraction function that works by restricting attention to a set of locations of interest. Assume that we are given a *location selection predicate* $\phi : \mathcal{H} \to (locs \to \{0, 1\})$. Note that ϕ is a *state-sensitive* predicate: the set of locations selected depends on the state. This gives us the flexibility to extract subheaps from different states differently.

For a state σ, let $locs^\sigma(\phi)$, denote the set of locations for which the predicate holds, i.e., $locs^\sigma(\phi) \overset{\text{def}}{=} \{v \in Loc^+ \mid \phi(\sigma)(v) = 1\}$. The subheap extraction operation $\delta_\phi : \mathcal{H} \to \mathcal{H}$ is defined as: $\delta_\phi(\sigma) = (env', field')$, where $env' \overset{\text{def}}{=} env^\sigma \cap (Var \times locs^\sigma(\phi))$, and $field' \overset{\text{def}}{=} field^\sigma \cap (Field \times locs^\sigma(\phi) \times locs^\sigma(\phi))$. A property of subheap extraction is that it returns smaller substates, i.e., $\delta_\phi(\sigma) \preceq \sigma$ for every $\sigma \in \mathcal{H}$.

B Proofs for Sec. 4

Proof (of Th. 2). Let j be any index in $\{1, \ldots, k\}$, and let TS_j be the corresponding combiner set.

$$I_1 \otimes \ldots \otimes I_k \sqsubseteq \bigotimes_{r \in TS_j} I_r \tag{20}$$

$\triangleright$ since $\otimes$ is monotone, i.e., $X \subseteq Y \implies \bigotimes_{r \in X} X \sqsubseteq \bigotimes_{r \in Y} Y$

$$\gamma(I) \sqsubseteq \gamma_{TS_j}(I) \tag{21}$$

$\triangleright$ by (20), (13), and (3)

$$\tau(\gamma(I)) \sqsubseteq \tau(\gamma_{TS_j}(I)) \tag{22}$$

$\triangleright$ since τ is monotone

$$\tau(\gamma(I)) \sqsubseteq \hat{\eta}_j(\tau(\gamma_{TS_j}(I))) \tag{23}$$

$\triangleright$ by (22) and since $\hat{\eta}_j$ is extensive

$$\tau(\gamma(I)) \sqsubseteq \tau_1[\mathit{TS}_j, j](I) \tag{24}$$

$\triangleright$ by (23) and (14)

$$\tau(\gamma(I)) \sqsubseteq \tau_1[\mathit{TS}_1, 1](I) \otimes \ldots \otimes \tau_1[\mathit{TS}_k, k](I) \tag{25}$$

$\triangleright$ since $\otimes$ is a meet operator

$$\tau(\gamma(I)) \sqsubseteq \gamma(\tau_1[\mathit{TS}_1, 1](I), \ldots, \tau_1[\mathit{TS}_k, k](I)) \tag{26}$$

$\triangleright$ by (3) and (25)

$$\tau(\gamma(I)) \sqsubseteq \gamma(\tau_1[\mathit{TS}](I))$$

$\triangleright$ by (15) and (26)

□

Proof (of Th. 4). Let j be any index in $\{1, \ldots, k\}$, let $\mathit{TS}_j \subseteq \mathcal{P}(\{1, \cdots, k\})$ be the corresponding set of combiner sets, and let $Y \subseteq \{1, \cdots, k\}$ be a combiner set in TS_j.

$$\tau(\gamma(I)) \sqsubseteq \tau_1[R, j](I) \tag{27}$$

by (24) in $Th.$ 2

$$\tau(\gamma(I)) \sqsubseteq \bigsqcap_{R \in TS_j} \tau_1[R, j](I) \tag{28}$$

by (27) and the properties of $\sqcap$

$$\tau(\gamma(I)) \sqsubseteq \tau_2[\mathit{TS}_j, j](I) \tag{29}$$

by (28) and (17)

$$\tau(\gamma(I)) \sqsubseteq \tau_2[\mathit{TS}_k, k](I) \otimes \ldots \otimes \tau_2[\mathit{TS}_k, k](I) \tag{30}$$

by (29) and since $\otimes$ is a meet operator

$$\tau(\gamma(I)) \sqsubseteq \gamma(\tau_2[\mathit{TS}_1, 1](I), \ldots, \tau_2[\mathit{TS}_k, k](I)) \tag{31}$$

by (31) and (3)

$$\tau(\gamma(I)) \sqsubseteq \gamma(\tau_2[\mathit{TS}](I))$$

by (18) and (31)

□

C HeDec System Optimizations

In this section we explain some of the important implementation details of the HeDec system.

HeDec implements standard fixed point iteration techniques where the abstract elements are tuples of sets of substates, one set per location selection predicate.

C.1 Incremental Transformers

We optimize the fixed point iteration by reusing the results from previous iterations. Without composition, the transformers are distributive and thus they are trivially incremental. The challenge is handling changes to sets from different tuples when they are combined. Combining sets is defined as $X_1 \otimes X_2 = \{\sigma_1 \otimes \sigma_2 \mid \sigma_1 \in X_1, \sigma_2 \in X_2\}$ where $\sigma_1 \otimes \sigma_2$ is an operation that combines individual substates.

For two sets of substates X and Y, let $\mathit{\Delta} X$ and $\mathit{\Delta} Y$ be new substates for each set, respectively. Now, we would like to compute $\tau((X \sqcup \mathit{\Delta} X) \otimes (Y \sqcup \mathit{\Delta} Y))$ by reusing $\tau(X \otimes Y)$. We use a known technique in computing differential fixpoint iterations (see, e.g., [7]), and use the transformer

$$\begin{aligned}\tau((X \sqcup \mathit{\Delta} X) \otimes (Y \sqcup \mathit{\Delta} Y)) = {} & \tau(X \otimes Y) \sqcup \\ & \tau(X \otimes \mathit{\Delta} Y) \sqcup \\ & \tau(Y \otimes \mathit{\Delta} X) \sqcup \\ & \tau(\mathit{\Delta} X \otimes \mathit{\Delta} Y)\end{aligned}$$

where the first joined element is taken from the previous iteration.

The use of incremental transformer is very important for efficiency. For example, on the non-blocking stack of Sec. 2.1, the incremental transformers improve the running times of 5 threads from 206 seconds to 36 seconds and of 10 threads from 2612 seconds to 211 seconds. More than 10-fold improvement that increases as the complexity of the problem and the number of threads increase.

C.2 Optimized Composition for Sets of Substates

One of the costly operations in our framework is the combination operator on sets $X \otimes Y$ (which is implemented using the algorithm from [2]). The number of substates that need to be combined grows exponentially with the number of sets. In our benchmarks, we usually compose at most 3 sets but this is still very costly, in practice.

However, many of the pairs of substates that are combined are inconsistent, and thus do not contribute substates in the output. We therefore use pruning techniques to avoid combining many inconsistent substates unnecessarily.

For a state $\sigma \in X$, we say that $\mathit{signature}_X(\sigma)$ is a signature of σ in X, if for every $\sigma' \in X$, we have the property that if $\mathit{signature}_X(\sigma) \neq \mathit{signature}_X(\sigma')$ then σ and σ' are inconsistent. We use signatures based on unary predicates to combine sets of substates by:

$$X \otimes Y = \{\sigma_1 \otimes \sigma_2 \mid \mathit{signature}_{X \cup Y}(\sigma_1) = \mathit{signature}_{X \cup Y}(\sigma_2)\} \ .$$

We have observed, in our experiments, that using the optimized combination for sets reduces the amount of useless combinations operations by up to a factor of 100.

C.3 Case Study: Proving Linearizability for a Two-Lock Queue

```
#define EMPTY -1
typedef struct queue_t {
  struct element_t *Head;
  struct element_t *Tail;
  lock_type HLock;
  lock_type TLock;
} Queue;

//  @pre Queue->Head!=NULL &&
//      Queue->Tail!=NULL
void enqueue(Queue *Q, data_type v){
  Node *x = alloc(sizeof(Node));
  x->d = v;
  lock(&Q->TLock);
    Node *t = Q->Tail;
    t->n = x;
    Q->Tail = x;
  unlock(&Q->TLock);
}
```

```
// @pre Queue->Head!=NULL &&
//    Queue->Tail!=NULL
data_type dequeue(Queue *Q){
  lock(&Q->HLock);
    Node *h = Q->Head;
    Node *s = h->n;
    if (s == NULL)
      unlock(&Q->HLock);
      return EMPTY;
    data_type r = s->d;
    Q->Head = s;
  unlock(&Q->HLock);
  return r;
}
```

Fig. 5. Two-lock queue implementation

A running example Fig. 5 shows the two-lock queue implementation described in [16]. The queue has Head and Tail pointers, each protected with its own lock. Note that although the implementation uses locks, the algorithm allows benign data-races in case the queue is empty, i.e., the Head and Tail pointers are aliased.

Concrete Execution Fig. 6 shows one example of a store occurring in the two-lock queue implementation shown in Fig. 5. The figure shows two consumer threads and two producer threads. The elements of the heap already correlated with the sequential execution are marked with *corr*. Locks are depicted by arrows to the locking thread.

prod1 and **cons2** are waiting in the corresponding lock acquire point, waiting for the lock. **cons1** finished dequeuing an element from the queue and is about to release the lock. Finally, **prod2** has already added an element to the tail of the queue, but has not yet updated the Tail pointer. The source of exponential explosion in the state space exploration of the two-lock queue algorithm is the correlation between the program locations of the different threads as in the coarse-grained concurrency.

The Decomposition Scheme We refine the decomposition scheme of Sec. 2.1 by adding a subdomain to represent the locks. The subheap contains the objects pointed-to by global variables and for each lock, the thread object acquiring it. Fig. 7 shows the the effect of applying this decomposition to the full state in Fig. 6.

The important thing to notice is that all the exponential explosion in the state space that existed in the full heap is eliminated by this decomposition. The possible subheaps of each thread become independent of the number of threads in the system (for more than 2 threads). The subheaps of the locks subdomain ($\{T_3\}$) only contain the thread information of 2 threads at most at a time.

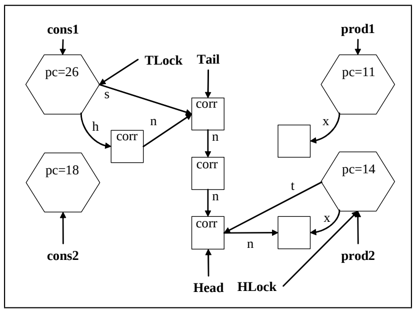
R60mm 

Fig. 6. A concrete memory in the two-lock queue implementation shown in Fig. 5

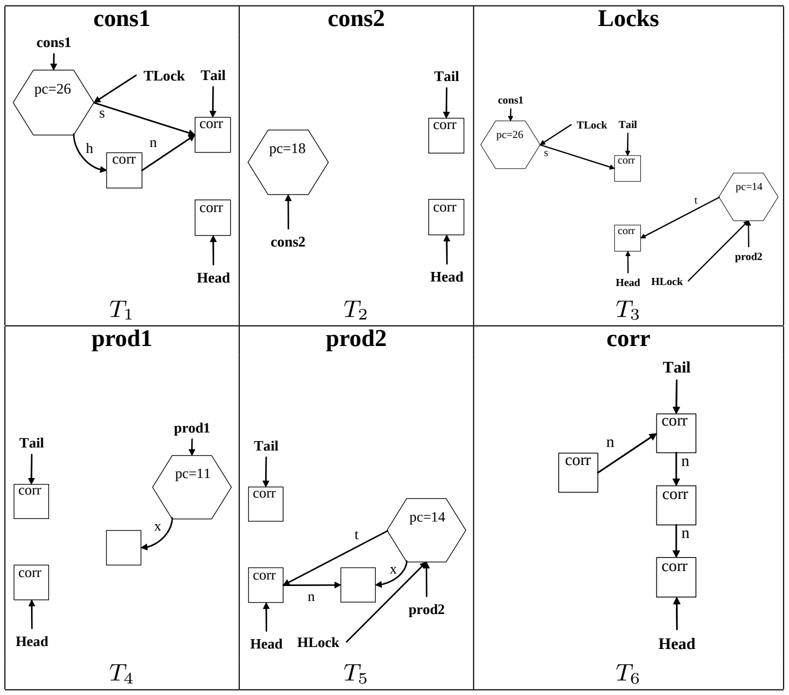


Fig. 7. The decomposed states abstracting the full state in Fig. 6. The names of the sub-domains appear above each substate

Transformers The compositions described in Sec. 2.1 work here as well. In the added operations of acquiring and releasing a lock, the subdomain of the currently executing thread is combined with the locks subdomain and each of the other components.