

Thread Quantification for Concurrent Shape Analysis

J. Berdine¹, T. Lev-Ami^{2,*}, R. Manevich^{2,**}, G. Ramalingam³, and M. Sagiv²

¹ Microsoft Research Cambridge, jjb@microsoft.com

² Tel Aviv University, {tla,rumster,msagiv}@post.tau.ac.il

³ Microsoft Research India, grama@microsoft.com

Abstract. In this paper we address the problem of shape analysis for concurrent programs. We present new algorithms, based on abstract interpretation, for automatically verifying properties of programs with an unbounded number of threads manipulating an unbounded shared heap.

Our algorithms are based on a new abstract domain whose elements represent *thread-quantified* invariants: i.e., invariants satisfied by all threads. We exploit existing abstractions to represent the invariants. Thus, our technique lifts existing abstractions by wrapping universal quantification around elements of the base abstract domain. Such abstractions are effective because they are thread modular: e.g., they can capture correlations between the local variables of the same thread as well as correlations between the local variables of a thread and global variables, but forget correlations between the states of distinct threads. (The exact nature of the abstraction, of course, depends on the base abstraction lifted in this style.)

We present techniques for computing sound transformers for the new abstraction by using transformers of the base abstract domain. We illustrate our technique in this paper by instantiating it to the Boolean Heap abstraction, producing a Quantified Boolean Heap abstraction. We have implemented an instantiation of our technique with Canonical Abstraction as the base abstraction and used it to successfully verify linearizability of data-structures in the presence of an unbounded number of threads.

1 Introduction

This paper is concerned with verifying (basic safety and other functional correctness) properties of dynamically-allocated data structures in programs with an unbounded number of threads. For example, the techniques in this paper enable, for the first time, automatic verification of linearizability of various implementations of concurrent data structures *even when an unbounded number of client threads manipulate these data structures concurrently*.

Our approach is based on abstract interpretation, which requires us to address the standard two principal challenges:

- to define a finite representation of infinite sets of program states that can concisely and precisely express the properties of interest, and
- to compute sound transformers, which over-approximate a program’s semantics using this representation.

* Supported by an Adams Fellowship through the Israel Academy of Sciences and Humanities.

** This research was partially supported by the Clore Fellowship Programme.

Quantification-Based Abstract Domain. The basis of our approach is the use of an abstract domain whose elements represent quantified invariants of the form $\forall t. \varphi(t)$, where the quantification is over threads. The formulas $\varphi(t)$ correspond to an abstraction of the program state from the perspective of a thread t . A second aspect of our approach is that we exploit existing abstractions to capture the component $\varphi(t)$ inside the quantifier. Informally, assume that we have an underlying abstraction where the abstract domain corresponds to a set of formulas A_{Voc} over a vocabulary Voc . (Usually, the vocabulary captures the dependence of the abstract domain on the program being analyzed.) We refine the abstraction and work with the set of formulas $L_{Voc} = \{\forall t. \varphi(t) \mid \varphi(t) \in A_{Voc \cup \{t\}}\}$. Thus, our technique may be seen as a domain constructor. The thread-quantified domain we construct is bounded to the degree that the underlying domain is: e.g., a finite-height base domain yields a finite-height quantified domain.

Transformers. We show how we can compute sound transformers for our new domain using sound transformers for the base domain. We present a simple technique for computing a *basic* sound transformer. The basic transformer works well when a thread’s action does not (potentially) affect the invariants (or state) observed by other threads. We also present a more sophisticated technique for computing a *refined* transformer, which is useful for thread actions that affect other threads.

The basic ideas underlying the construction of such a quantified abstract domain have appeared in various forms in recent work, see Sec. 5. One of the novel contributions of our work is the use of such quantification for concurrent *shape analysis* by using suitable shape analysis abstractions such as *Canonical Abstraction* [24] and *Boolean Heaps* [23] as the base domain. We have implemented our technique on top of the TVLA [16] system,⁴ which is based on Canonical Abstraction, and used it to verify *linearizability* of fine-grained concurrency algorithms [1]. However, we illustrate our ideas in this paper using Boolean Heaps as the base domain, as its simplicity allows us to focus on the essence of our approach.

The thread-quantified abstract domain is a natural domain to use for reasoning about programs with an unbounded number of threads. It permits expressing properties that correlate a thread’s local variables with each other and with shared global state, but not ad-hoc properties that correlate distinct, threads’ local variables. (By “ad-hoc”, we mean properties that cannot be captured using quantification.) Note that when the underlying base domain is disjunctive, as is the case with Canonical Abstraction and Boolean Heaps, the new domain permits disjunctions inside the quantifier, which is quite useful.

2 Overview

In this section, we present an informal overview of our method.

A Motivating Example. Fig. 1 shows a toy concurrent program used to illustrate the ideas in this paper. Sec. 4 reports on applying these ideas to more realistic programs. The program satisfies a couple of very simple invariants (expressed as assertions) that we would like to automatically infer. The first invariant is that when a thread is at

⁴ We actually implemented our technique in HeDec [18], which generalizes Canonical Abstraction by allowing coarser and more scalable abstractions.

```

Object g = null; // global variable
threadProc() {
    Object x = null, y = null;
    [1] x = new Object();
    [2] y = x;
    [3] assert(x == y);
        g = x;
    [4] assert(g != null);
}

```

Fig. 1. A simple multithreaded program. The program consists of an unbounded number of threads executing `threadProc`.

statement [3], the values of its `x` and `y` variables are equal. This is an example of a *thread-local* invariant (which cannot be affected by the execution of other threads). The second invariant is that when a thread is at statement [4], the global variable `g` is non-null. This is an example of a *non-local thread* invariant, and can be affected by the execution of other threads. In general, a non-local thread invariant could involve global as well as thread-local variables. As an example, consider an assertion that when a thread is at statement [4], the value of `g` and its `x` are equal. This is an assertion that fails to hold for the given program (because of interaction with other threads).

Background: The Boolean Heap Abstraction. As explained in Sec. 1, our approach is to lift an existing abstraction to produce a more precise abstraction that is suitable for programs with an unbounded number of threads. We will illustrate our idea using Boolean Heaps [23] as the underlying base abstraction in this paper. Boolean Heaps are abstractions targeted at shape analysis, and describe sets of states consisting of an unbounded number of heap objects using formulas of the form

$$\bigvee_{i \in I} \{ \forall v : \text{object}. \varphi'_i(v) \}$$

where v ranges over heap objects and $\varphi'(v)$ is a quantifier-free formula, in which v possibly occurs free, over a set of unary predicates, kept in DNF.

The Quantified Boolean Heap Abstraction. As explained earlier, the basis of our approach is to use quantified invariants of the form $\forall t. \varphi(t)$, where the set of formulas $\varphi(t)$ allowed inside the invariant are determined by a base domain. Using Boolean Heaps as our base domain leads to the following definition of *Quantified Boolean Heaps*. Quantified Boolean Heaps approximate sets of states by formulas of the form

$$\forall t : \text{thread}. \bigvee_{i \in I} \{ \forall v : \text{object}. \varphi_i(t, v) \}$$

where t ranges over threads, v ranges over non-thread objects, $\varphi(t, v)$ is a quantifier-free

Table 1. Predicates used for (Quantified) Boolean Heap Abstraction

Predicate	Intended meaning
$x(t, v)$	local variable <code>x</code> of thread t points to object v
$y(t, v)$	local variable <code>y</code> of thread t points to object v
$g(v)$	global variable <code>g</code> points to object v
$null(v)$	v is a special null object
$at[l](t)$	thread t is at program label l

Table 2. Part of the computed quantified invariant for the running example

$$\begin{aligned}
\forall t. \quad & \dots \vee at[4](t) \wedge \\
& \forall v \{ x(t) \wedge y(t) \wedge g \wedge \neg null \}(v) \vee \{ \neg x(t) \wedge \neg y(t) \wedge \neg g \wedge null \}(v) \\
\vee \forall v \{ & x(t) \wedge y(t) \wedge \neg g \wedge \neg null \}(v) \vee \{ \neg x(t) \wedge \neg y(t) \wedge \neg g \wedge null \}(v) \vee \\
& \{ \neg x(t) \wedge \neg y(t) \wedge g \wedge \neg null \}(v) \\
\vee \forall v \{ & x(t) \wedge y(t) \wedge g \wedge \neg null \}(v) \vee \{ \neg x(t) \wedge \neg y(t) \wedge \neg g \wedge null \}(v)
\end{aligned}$$

formula, in which t and v possibly occur free, over a set of unary and binary predicates, kept in DNF.⁵ For the running example, we will use the predicates shown in Tab. 1. We assume that a *null* value is represented by a special heap object.

Notice that a quantified boolean heap is a universally quantified disjunction of standard Boolean heaps, but where some previously unary predicates have been indexed by the universal variable. This increases the expressive power of the abstract elements.

For brevity we use the following, not disjunctive normal, form

$$\forall t:thread. \bigvee_{l \in \text{Labels}} at[l](t) \wedge \left(\bigvee_{i \in I} \{ \forall v:object. \varphi_i(t, v) \} \right)$$

where Labels is the set of program labels and the predicates of the form $at[l](t)$ are implicitly mutually exclusive.⁶ We use the following notational conventions: Logical variables $\{sc, t, t_1, t_2, \dots\}$ range over thread objects and other logical variables range over non-thread objects. We use $\{p \wedge q\}(v)$ as shorthand for $p(v) \wedge q(v)$ and curry binary predicates, i.e., $\{p(t) \wedge q\}(v)$ is a shorthand for $p(t, v) \wedge q(v)$.

Tab. 2 shows the part of the Quantified Boolean Heap describing the invariant of our running example for the threads at program location [4], as computed by our analysis.

Abstract Transformers. Computing abstract transformers is very challenging, especially in the presence of concurrency, as the execution of one thread may affect the state observable by other threads. In Sec. 3 we present effective techniques for computing sound transformers for our lifted abstractions, utilizing transformers of the base abstraction. The main idea is to “instantiate” two symbolic threads, one for the scheduled thread, and one representing another arbitrary thread, and to utilize the transformer of the underlying base domain to compute the change in the abstract state as observed by each of these threads.

Discussion. For comparison, consider the Quantified Boolean Heaps abstraction just described and the abstractions used by the original Boolean Heaps and 3VMC [27], which naturally models unbounded objects and threads in a uniform fashion using Canonical Abstraction. For *the set of predicates described in this section*, our new analysis is capable of inferring the invariants mentioned in the program: namely, that for any thread t at program location 3, we have $x(t) = y(t)$, and that ϱ is not null at the end (for any thread’s execution). On the other hand, without adding different predicates, neither the Boolean Heaps analysis nor 3VMC can infer the above invariants. Indeed, these

⁵ This is similar to Indexed Predicate Abstraction [15], except that the number of index variables is limited to 2, and that we allow a disjunction between the quantifiers.

⁶ The location predicates are written outside the internal universal quantifier because they are independent of v .

abstractions cannot even express these invariants. If a richer set of predicates is used, especially instrumentation predicates, these abstractions can be made more expressive and be used to prove the above invariants. An advantage of the new abstraction is that it can reduce the need for nonstandard or program-specific predicates, or the number of predicates, in a very natural way.

3 The Thread Quantification Domain Constructor

In this section, we describe how thread quantification can be used as a domain construction operator to generate a more precise abstract domain from an existing abstract domain. We illustrate this by applying it to the Boolean Heap domain to obtain the Quantified Boolean Heap domain.

3.1 The Concrete Semantics

We start by defining operational concrete semantics useful for describing concurrent programs without procedures. For simplicity of presentation, we concentrate on reference variables and fields. Let `Threads` and `Locations` (containing a distinguished *null* value) be countable sets representing threads and heap locations, respectively. Let `LVars`, `GVars`, and `Fields` be finite sets of local variables, global variables, and heap fields, respectively. Finally, let `Labels` be a finite set of program labels. Let Σ be the set of possible states. A state $\sigma \in \Sigma$ maps the following: for each global variable g , $\sigma(g) \in \text{Locations}$; for each local variable x , $\sigma(x) : \text{Threads} \rightarrow \text{Locations}$; for each field f , $\sigma(f) : \text{Locations} \rightarrow \text{Locations}$; and for `pc`, $\sigma(\text{pc}) : \text{Threads} \rightarrow \text{Labels}$.

Being interested in invariance properties, we start with a concrete powerset domain $\mathcal{P}(\Sigma)$, for which we assume a concrete semantics of programs $\text{spost}_{(\cdot)} : \text{Threads} \rightarrow \mathcal{P}(\Sigma) \rightarrow \mathcal{P}(\Sigma)$ that maps individual threads to their semantics. This induces the semantics of the overall concurrent program $\text{cpost} : \mathcal{P}(\Sigma) \rightarrow \mathcal{P}(\Sigma)$ by

$$\text{cpost}(S) = \bigcup_{\text{sc} \in \text{Threads}} \text{spost}_{\text{sc}}(S).$$

3.2 The Base Abstraction

We present the lifted abstract interpreter as well as the base abstract interpreter as operating on formulas in a normal form. This is done for simplicity of presentation. For example, Boolean Heaps are already presented using these terms. Details on how Canonical Abstraction can be presented in such terms can be found in [29].

In Sec. 3.6 we will specify the assumptions on the abstract domain input to the thread quantification construction, but it is useful to present in several steps.

Base Domain. Consider a base abstract domain with elements drawn from a set A of formulas, where $(\mathcal{P}(\Sigma), \alpha_A, \gamma_A, A)$ is a Galois Connection, with meet \sqcap_A and join \sqcup_A , and sound sequential transformer $\text{spost}_{(\cdot)}^\# : \text{Threads} \rightarrow A \rightarrow A$. As in the concrete semantics, this induces the abstract concurrent semantics $\text{cpost}^\# : A \rightarrow A$, which overapproximates cpost , by

$$\text{cpost}^\#(a) = \bigsqcup_{\text{sc} \in \text{Threads}} \text{spost}_{\text{sc}}^\#(a).$$

Open Formulas. Abstract elements often correspond to formulas without free variables. E.g., the formula $\forall v.g(v) \Leftrightarrow \text{null}(v)$ represents states where g is *null*. The first step toward thread-quantified formulas is to permit formulas with free variables (e.g., $\forall v.x(t, v) \Leftrightarrow \text{null}(v)$) as abstract domain elements.

For a set V of variables, let $A[V]$ denote the set of formulas in normal form with free variables contained in V . Let $\text{Assign}_V = V \rightarrow \text{Threads}$ be the set of assignments of (thread) variables in V to threads. A state $\sigma \in \Sigma$ and an assignment $\theta \in \text{Assign}_V$ satisfy $\varphi(V) \in A[V]$, denoted $\sigma, \theta \models \varphi(V)$, when assigning the parameters according to θ and interpreting the predicates according to σ yields true. Define Σ_V to be $\Sigma \times \text{Assign}_V$.

Example. The open formula $\forall v.x(t, v) \Leftrightarrow \text{null}(v)$ represents the set of all pairs $\langle \sigma, \theta \rangle$ such that the local variable x of thread $\theta(t)$ is null in σ , i.e., $\sigma(x)(\theta(t)) = \text{null}$.

By defining $\gamma_{A[V]} : A[V] \rightarrow \mathcal{P}(\Sigma_V)$ by $\gamma_{A[V]}(\varphi(V)) = \{\langle \sigma, \theta \rangle \mid \sigma, \theta \models \varphi(V)\}$, the satisfaction relation determines a Galois Connection $(\mathcal{P}(\Sigma_V), \alpha_{A[V]}, \gamma_{A[V]}, A[V])$.

Transformers for Open Formulas. Since the states Σ_V for open formulas are related to program states Σ simply by the first projection, the concrete semantics can be lifted to $\text{spost}_{V,(\cdot)} : \text{Threads} \rightarrow \mathcal{P}(\Sigma_V) \rightarrow \mathcal{P}(\Sigma_V)$ by defining

$$\text{spost}_{V,t}(S) = \bigcup_{\langle \sigma, \theta \rangle \in S} (\text{spost}_t(\{\sigma\}) \times \{\theta\}) .$$

The concurrent semantics $\text{cpost}_V : \mathcal{P}(\Sigma_V) \rightarrow \mathcal{P}(\Sigma_V)$ is induced by $\text{spost}_{V,(\cdot)}$ in the same way as cpost is induced by $\text{spost}_{(\cdot)}$:

$$\text{cpost}_V(S) = \bigcup_{\text{sc} \in \text{Threads}} \text{spost}_{V,\text{sc}}(S) .$$

The thread quantification domain construction requires transformers for open formulas $\text{cpost}_V^\sharp : A[V] \rightarrow A[V]$ that over-approximate cpost_V . While the definition of cpost_V^\sharp from cpost^\sharp varies from one domain to another, note that $\Pi_1(\text{cpost}_V(S)) = \text{cpost}(\Pi_1(S))$ (where Π_1 is the first projection of a pair, lifted pointwise to sets of pairs), and so an abstract transformer is sound with respect to cpost_V if and only if it is sound with respect to cpost . Also note that since cpost_V always leaves the thread assignment unchanged, sound over-approximations must also. Hence the free thread variables can be treated as constant symbols, and binary predicates such as $x(t, v)$ can be carried and then interpreted as unary predicates $(x(t))(v)$, which many base domains A directly support. In particular, assuming the base domain A can handle constant symbols, a domain $A[V]$ can be produced systematically.

We will specifically be interested in the case of formulas with a single free variable t : i.e., the case where $V = \{t\}$. The method can be generalized to multiple free variables and thus multiple universal quantifiers. This is outside the scope of the paper. Note that the union over all threads sc in the concrete transformer cpost_V captures the effect of a *single transition performed by an arbitrary thread sc*. A sound abstract transformer $\text{cpost}_{\{t\}}^\sharp$ must handle two cases: where thread variable t is the same as the scheduled thread sc , and where t is different from sc .

Example. We now illustrate the application of a sound transformer for the transition corresponding to the single statement $y=x$ on the open formula $\varphi(t) = \forall v.x(t, v) \Leftrightarrow \text{null}(v)$. This formula represents states σ and assignments $[t : \tau]$ where the local variable x of thread τ is null in σ . If thread τ executes the statement $y=x$, the resulting state can be described by the formula $\varphi_1(t) = \forall v.y(t, v) \Leftrightarrow x(t, v) \Leftrightarrow \text{null}(v)$. If some thread

other than τ is scheduled, then the local variables of τ are not affected, and the resulting state can be described by $\varphi(t)$ itself. We account for these two cases by taking the disjunction $\varphi_1(t) \vee \varphi(t)$, which simplifies to $\varphi(t)$, yielding the result of the transformer.

3.3 The Lifted Abstraction (with Basic Transformers)

We define the lifted domain $L = \{\forall t. \varphi(t) \mid \varphi(t) \in A[\{t\}]\}$, i.e., with the base domain instantiated with $V = \{t\}$. The lifted domain inherits meet and (an over-approximation of) join operations from $A[\{t\}]$: e.g., $(\forall t. \varphi_1) \sqcup_L (\forall t. \varphi_2) = \forall t. (\varphi_1 \sqcup_{A[\{t\}]} \varphi_2)$. Defining $\gamma_L : L \rightarrow \mathcal{P}(\Sigma)$ by

$$\gamma_L(\forall t. \varphi(t)) = \{\sigma \mid \sigma, \theta \models \varphi(t) \text{ for every } \theta \in \text{Assign}_{\{t\}}\}$$

produces a Galois Connection from $\mathcal{P}(\Sigma)$ to L . We obtain a sound transformer $\text{cpost}_L^\# : L \rightarrow L$ from the sound abstract transformer $\text{cpost}_{\{t\}}^\#$ for formulas with a free variable t discussed earlier as follows:

$$\text{cpost}_L^\#(\forall t. \varphi(t)) = \forall t. \text{cpost}_{\{t\}}^\#(\varphi(t)).$$

Example. Consider the statement $y=x$ from the example program in Fig. 1 and the abstract state $S1$:

$$\begin{aligned} S1 &= \forall t. S1_a(t) \vee S1_b(t) \\ S1_a(t) &= \text{at}[1](t) \wedge \\ &\quad \forall v. \{ x(t) \wedge y(t) \wedge g \wedge \text{null}\}(v) \vee \{\neg x(t) \wedge \neg y(t) \wedge \neg g \wedge \neg \text{null}\}(v) \\ S1_b(t) &= \text{at}[2](t) \wedge \\ &\quad \forall v. \{\neg x(t) \wedge y(t) \wedge g \wedge \text{null}\}(v) \vee \{ x(t) \wedge \neg y(t) \wedge \neg g \wedge \neg \text{null}\}(v) \vee \\ &\quad \{\neg x(t) \wedge \neg y(t) \wedge \neg g \wedge \neg \text{null}\}(v) \end{aligned}$$

Applying the Boolean Heap transformer for $y=x$ to $S1_a(t)$ leaves $S1_a(t)$ unchanged no matter whether t was the scheduled thread or not. Applying the Boolean Heap transformer for $y=x$ to $S1_b(t)$ yields the heaps $S1'_{b_1}(t)$ for the case where t is the scheduled thread, and leaves $S1_b(t)$ unchanged for the complementary case. The final result is obtained by universally quantifying over t , resulting in $S1'$:

$$\begin{aligned} S1' &= \forall t. S1_a(t) \vee S1_b(t) \vee S1'_{b_1}(t) \\ S1'_{b_1}(t) &= \text{at}[3](t) \wedge \\ &\quad \forall v. \{\neg x(t) \wedge \neg y(t) \wedge g \wedge \text{null}\}(v) \vee \{ x(t) \wedge y(t) \wedge \neg g \wedge \neg \text{null}\}(v) \vee \\ &\quad \{\neg x(t) \wedge \neg y(t) \wedge \neg g \wedge \neg \text{null}\}(v) \end{aligned}$$

Let $\phi_{x=y} = \forall t. \text{at}[3](t) \Rightarrow \forall v. x(t, v) \Leftrightarrow y(t, v)$ be the assertion at line [3]. Now, $S1' \models \phi_{x=y}$ (the only disjunct where $\text{at}[3]$ holds is $S1'_{b_1}(t)$, in which x and y point to the same node). The statement $y=x$ changes only information local to one thread and therefore this kind of reasoning is sufficiently precise.

Let $\phi_{g! = \text{null}} = \forall t. \text{at}[4](t) \Rightarrow \forall v. \neg(g(v) \wedge \text{null}(v))$ be the assertion at line [4]. Now, however, consider the statement $g=x$ and the abstract state $S2$:

$$\begin{aligned} S2 &= \forall t. S2_a(t) \\ S2_a(t) &= \text{at}[3](t) \wedge \\ &\quad \forall v. \{\neg x(t) \wedge \neg y(t) \wedge g \wedge \text{null}\}(v) \vee \{ x(t) \wedge y(t) \wedge \neg g \wedge \neg \text{null}\}(v) \vee \\ &\quad \{\neg x(t) \wedge \neg y(t) \wedge \neg g \wedge \neg \text{null}\}(v) \end{aligned}$$

When t is not the scheduled thread, applying the Boolean Heap transformer to the Boolean Heap $S2_a(t)$ yields many Boolean Heaps. This is because of the lack of information about the status of other threads, which we get by dropping the universal quantification over t . The scheduled thread may be different from t . Thus, $S2_a(t)$ has no information about it. In particular, $S2_a(t)$ represents a state where x and y are null for the scheduled thread. As a result, the assignment $g=x$ also creates the Boolean Heap $S_{bad} = at[A](t) \wedge \forall v. \{x(t) \wedge y(t) \wedge g \wedge null\}(v)$. Obviously, $S_{bad} \not\models \phi_{g!=null}$ and thus the transformer is not precise enough for the purpose of our analysis.

The reason is that $g=x$ changes a global variable. This change is visible by other threads and thus the thread-local reasoning used above does not model the effect of the other threads using the information captured by the Quantified Boolean Heap.

3.4 The Semantics of Non-Deterministic Scheduling

In order to obtain a more precise sound transformer for our lifted abstract domain, we exploit the internal structure of the concrete semantics and the base abstract transformer imposed by the semantics of non-deterministic scheduling.

Recall that the concurrent semantics of a program $cpost$ is defined in terms of $spost_t$, which gives the sequential semantics of the individual threads. This function indicates the transitions a given thread t can take. The semantics of non-deterministic scheduling of threads is captured by the union over all threads in the definition of $cpost$.

While the basic transformer $cpost_L^\#$ was defined in terms of $cpost_V^\#$, for the refined transformer we will not use the naive definition of the concurrent semantics in terms of the sequential semantics but will instead define the refined transformer directly in terms of the sequential abstract transformer.

In particular, we assume an abstract transformer $spost_{V,sc}^\# : A[V \cup \{sc\}] \rightarrow A[V \cup \{sc\}]$ for $sc \notin V$ that over-approximates $spost_{V,sc} : \mathcal{P}(\Sigma_{V \cup \{sc\}}) \rightarrow \mathcal{P}(\Sigma_{V \cup \{sc\}})$ given by

$$spost_{V,sc}(S) = \bigcup_{\langle \sigma, \theta \rangle \in S} (spost_{\theta(sc)}(\{\sigma\}) \times \{\theta\}) .$$

The difference between this semantics and $spost_{V,(\cdot)}$ above is that $spost_{V,sc}$ looks up sc in the assignment in the input state to determine which thread to execute. In essence, we are assuming that the scheduled thread is specified as an extra parameter for the transformer of open formulas in the base domain. Lifting the transformers of the base domain to support the scheduled thread as an extra parameter is usually straightforward. Inducing the concrete semantics from $spost_{V,sc}$ by

$$cpost_V(S) = \bigcup_{sc \in \text{Threads}} \{ \langle \sigma', \theta' \rangle_V \mid \langle \sigma', \theta' \rangle \in spost_{V,sc} \{ \langle \sigma, [\theta]_{sc:sc} \rangle \mid \langle \sigma, \theta \rangle \in S \} \}$$

(where $\theta'|_V$ is θ' restricted to domain V) yields the same definition of $cpost_V$ as above.

We also assume that the base abstract domain has an operation $project(sc, (\cdot)) : A[V \cup \{sc\}] \rightarrow A[V]$ for projecting away a thread parameter sc . This is equivalent to over-approximating existential elimination. For example, in Boolean Heaps, we can simply throw away all literals (positive and negative) that contain sc .

Using these operations, the transformer for the overall concurrent program $cpost_V^\# : A[V] \rightarrow A[V]$ is defined, for $sc \notin V$, by

$$cpost_V^\#(\varphi(V)) = project(sc, spost_{V,sc}^\#(\varphi(V))) .$$

Note how this definition allows an arbitrary thread to execute since sc does not occur in $\varphi(V)$, hence $\varphi(V)$ does not constrain the thread assigned to sc , and hence the set of states that satisfy $\varphi(V)$ will include assignments that map sc to any element of Threads.

3.5 A More Precise Transformer for the Lifted Domain

We will now present a more precise sound transformer for the lifted domain. The basic transformer presented in Sec. 3.3 transformed a quantified formula $\forall t. \varphi(t)$ by applying the base domain's (open formula) transformer to $\varphi(t)$. This leads to a loss of precision because the base domain transformer knows only that t satisfies $\varphi(t)$. *It does not know and cannot use the fact that both the scheduled thread sc and another arbitrary thread t satisfy the invariant.* We now show how we can incorporate this extra piece of information, *while still reusing the base domain's transformer*, producing a more precise transformer for the lifted domain.

We define the refined transformer $\text{cpost}'_{L}^{\#} : L \rightarrow L$ by

$$\text{cpost}'_{L}^{\#}(\forall t. \varphi(t)) = \forall t. \text{project}(sc, \text{spost}'_{\{t\}, sc}^{\#}(\varphi(t) \sqcap_{A[\{t, sc\}]} \varphi(sc))) .$$

Specifically, we apply the base domain's transformer to $\varphi(t) \sqcap_{A[\{t, sc\}]} \varphi(sc)$, exploiting the base domain's meet operation to “inform” the base domain's transformer that both $\varphi(t)$ and $\varphi(sc)$ are true in the input state.

Example. We demonstrate the refined transformer by computing $\text{cpost}'_{L}^{\#}(S2_a(t))$. The first step of the transformer is to compute the meet of $S2_a(t)$ and $S2_a(sc)$ (where for brevity, we have not converted the formula to DNF):

$$\begin{aligned} \varphi(sc, t) = S2_a(t) \sqcap S2_a(sc) &= \text{at}[3](t) \wedge \text{at}[3](sc) \wedge \\ &\forall v. \{ \neg x(t) \wedge \neg y(t) \wedge g \wedge \text{null}\}(v) \vee \{x(t) \wedge y(t) \wedge \neg g \wedge \neg \text{null}\}(v) \vee \\ &\quad \{ \neg x(t) \wedge \neg y(t) \wedge \neg g \wedge \neg \text{null}\}(v) \\ \wedge \forall v. \{ \neg x(sc) \wedge \neg y(sc) \wedge g \wedge \text{null}\}(v) &\vee \{x(sc) \wedge y(sc) \wedge \neg g \wedge \neg \text{null}\}(v) \vee \\ &\quad \{ \neg x(sc) \wedge \neg y(sc) \wedge \neg g \wedge \neg \text{null}\}(v) \end{aligned}$$

Next, we apply the Boolean Heaps transformer $\text{spost}'_{\{t\}, sc}^{\#}$ to $\varphi(sc, t)$. As explained earlier, this is a sound transformer of a single transition taken by thread sc . As before, we obtain the result as a disjunction of two heaps, $\varphi'_a(sc, t)$ for the case in which $sc = t$ and $\varphi'_b(sc, t)$ for the case in which $sc \neq t$.

$$\begin{aligned} \varphi'(sc, t) &= \varphi'_a(sc, t) \vee \varphi'_b(sc, t) \\ \varphi'_a(sc, t) &= \text{at}[4](t) \wedge \text{at}[4](sc) \wedge \\ &\quad \forall v. \{ \neg x(t) \wedge \neg y(t) \wedge \neg g \wedge \text{null}\}(v) \vee \{x(t) \wedge y(t) \wedge g \wedge \neg \text{null}\}(v) \vee \\ &\quad \{ \neg x(t) \wedge \neg y(t) \wedge \neg g \wedge \neg \text{null}\}(v) \\ \wedge \forall v. \{ \neg x(sc) \wedge \neg y(sc) \wedge \neg g \wedge \text{null}\}(v) &\vee \{x(sc) \wedge y(sc) \wedge g \wedge \neg \text{null}\}(v) \vee \\ &\quad \{ \neg x(sc) \wedge \neg y(sc) \wedge \neg g \wedge \neg \text{null}\}(v) \\ \varphi'_b(sc, t) &= \text{at}[3](t) \wedge \text{at}[4](sc) \wedge \\ &\quad \forall v. \{ \neg x(t) \wedge \neg y(t) \wedge \neg g \wedge \text{null}\}(v) \vee \{x(t) \wedge y(t) \wedge \neg \text{null}\}(v) \vee \\ &\quad \{ \neg x(t) \wedge \neg y(t) \wedge \neg \text{null}\}(v) \\ \wedge \forall v. \{ \neg x(sc) \wedge \neg y(sc) \wedge \neg g \wedge \text{null}\}(v) &\vee \{x(sc) \wedge y(sc) \wedge g \wedge \neg \text{null}\}(v) \vee \\ &\quad \{ \neg x(sc) \wedge \neg y(sc) \wedge \neg g \wedge \neg \text{null}\}(v) \end{aligned}$$

We project away sc , by removing all literals containing it, which yields:

$$\begin{aligned}
\varphi''(t) &= \varphi_a''(t) \vee \varphi_b''(t) \\
\varphi_a''(t) &= at[4](t) \wedge \\
&\quad \forall v. \{ \neg x(t) \wedge \neg y(t) \wedge \neg g \wedge \text{null} \}(v) \vee \{ x(t) \wedge y(t) \wedge g \wedge \neg \text{null} \}(v) \vee \\
&\quad \{ \neg x(t) \wedge \neg y(t) \wedge \neg g \wedge \neg \text{null} \}(v) \\
\varphi_b''(t) &= at[3](t) \wedge \\
&\quad \forall v. \{ \neg x(t) \wedge \neg y(t) \wedge \neg g \wedge \text{null} \}(v) \vee \{ x(t) \wedge y(t) \wedge \neg \text{null} \}(v) \vee \\
&\quad \{ g \wedge \neg \text{null} \}(v) \vee \{ \neg x(t) \wedge \neg y(t) \wedge \neg \text{null} \}(v)
\end{aligned}$$

The interesting observation here is that g and null are not aliased in both conjuncts of $\varphi_b''(sc, t)$. Thus, after the projection we retain this information.

Finally, the result is universally quantified, i.e., $S_2' = \forall t. \varphi''(t)$. As expected, $S_2' \models \phi_{g \neq \text{null}}$.

3.6 Summary of Construction

In summary, the thread quantification domain construction requires implementations of: an abstract domain $A[V]$ of open formulas that is in a Galois Connection with $\mathcal{P}(\Sigma_V)$ induced by the satisfaction relation; meet and join operations on $A[V]$; sequential transformers $\text{spost}_{V,sc}^\sharp$ of open formulas, parameterized by the scheduled thread, which over-approximate $\text{spost}_{V,sc}$ as in Sec. 3.4; and an over-approximation of existential elimination $\text{project}(sc, (\cdot))$ as in Sec. 3.4. From this the construction produces an implementation of an abstract domain L of quantified formulas, which is in a Galois Connection with $\mathcal{P}(\Sigma)$, with basic transformers cpost_L^\sharp and refined transformers cpost'_L^\sharp for concurrent programs that over-approximate the concrete semantics cpost .

4 Case Study: Proving Linearizability

As a case study for the approach, we have verified linearizability of three well-known concurrent data structure implementations that use fine-grained concurrency.

4.1 Implementation

We have implemented the approach on top of TVLA [16]. (Actually, we implemented our technique in HeDec [18], which generalizes Canonical Abstraction by allowing coarser and more scalable abstractions.) The thread parameters were implemented as unary predicates. Support for treating a binary formula of the form $x(t, v)$ as a unary predicate was done by adding an appropriate instrumentation predicate (i.e., predicate defined using a formula from other predicate and automatically updated by the system).

The meet and join operations required from the base domain are already implemented in TVLA. Thread projection is done by forgetting all information about the unary predicate representing the thread and all instrumentation predicates based on it.

In TVLA, it is easier to implement separate transformers for each statement and let the engine deal with constructing the full post operator. As a result, we are able to use the basic transformer for some statements and the more expensive refined transformer only for statements that require the extra precision. We use the basic transformer for statements that modify only the local state of the scheduled thread and leave the global

state intact. In these cases the abstract state of any thread that is not the scheduled thread is unchanged by the operation, thus the precision of the basic transformer is enough.

4.2 Proving Linearizability

Linearizability [12] is one of the main correctness criteria for implementations of concurrent data structures. Informally, a concurrent data structure is 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.

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.

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. The details are described in [1].

We have taken the instantiation of Canonical Abstraction presented in [1] as the base abstraction for the analysis. That analysis was performed for a bounded number of threads, by using specialized predicates treating each local variable of each thread as a distinct predicate. We removed these extra predicates, instead treating the thread local variables as binary predicates. The analysis has predicates for local and global variables, heap fields and program labels. Finally, we use as is two extra predicates that capture the correlation between the concurrent and sequential executions (see [1]).

4.3 Experimental Results

Tab. 3 summarizes the experimental results of running our linearizability analysis on the algorithms. These benchmarks were run a 2.4GHz E6600 Core 2 Duo processor with 2 GB of memory running Linux. We used two abstractions to analyze these examples. The first is vanilla canonical abstraction as described in Sec. 4.2. The second abstraction is an extension of canonical abstraction with decomposition of the heap as described in [18]. With this abstraction, the state space is significantly reduced, yielding fewer states and better times. The adaptation of the transformer for the decomposing abstraction was no harder than that for vanilla canonical abstraction.

Treiber’s stack algorithm [25] is lock-free, and uses a Compare And Swap (CAS) operation for synchronization. The two-lock queue algorithm [19] has Head and Tail pointers, each protected with its own lock. It allows benign data-races when the queue is empty, i.e., the Head and Tail pointers are aliased. The non-blocking queue algorithm [6] is lock-free and uses CAS for synchronization. It is more complicated than the other two algorithms and has a much larger state space with our abstraction. Canonical Abstraction without decomposition, on this example, resulted in state space explosion.

5 Related Work

The abstract interpretation presented in this paper inherits from, and combines, two lines of prior work: (1) Prior work on abstract domains of quantified formulas, especially in the context of verification of parametrized concurrent systems, and (2) Prior work on shape analysis.

Process-Centric Abstraction. The general approach we use of reasoning about concurrent programs in terms of an *abstraction of the program state relative to a thread* is classic in work on program logic: assertions within the code of a thread refer to the state from that thread’s perspective, and the thread’s concurrent environment is over-approximated by, for instance, invariants [13, 21] or relations [14] on the shared state. This idea has also been used early on for automatic compositional verification [4]. More recently, this approach has led to the notion of thread-modular verification for model checking systems with finitely-many threads [8], and has also been applied more closely to our present domain of heap-manipulating programs with coarse-grained concurrency [9], and less automatically to fine-grained concurrency [2]. This general principle has also previously been used in the context of verification of sequential programs in the form of abstractions of program state relative to one or more non-deterministically chosen objects (e.g., in the heap or an array) [7, 28, 23, 26].

Abstract Interpretation with Open Formulas and Quantified Invariants. In this paper, we realize such a reference-object-centric perspective within the framework of abstract interpretation, using abstract domains consisting of formulas with free variables as a stepping stone toward abstract domains consisting of quantified formulas. This approach has been previously formalized in the work on Indexed Predicate Abstraction [15] and also appears in the work on Environment Abstraction [5, 3]. Indices, or free variables, in the indexed predicate abstraction work can range over anything, depending on the application. Our use of a single variable for a reference process is similar to the approach in Environment Abstraction. A similar quantified invariants approach has also been used in the analysis of heap properties [23] and properties of collections [10] in sequential programs.

Transformers for Quantified Formulas. The chief difficulty, particularly for domain constructions, is defining the transformers: semantics of program statements on elements of the abstract domain. In their work on Indexed Predicate Abstraction, Lahiri et al., outline the idea of using *quantifier instantiation* to compute abstract transformers of quantified formulas. They use a tool to generate candidate instantiations (based on the subexpressions that appear in the predicate and next-state expressions) for this purpose.

Table 3. Experimental results of proving linearizability for an unbounded number of threads

Algorithms	Canonical Abstraction		with decomposition	
	States	secs.	States	secs.
Stack [25]	4,097	53	764	7
Two-lock queue [19]	4,897	47	3,415	17
Non-blocking queue [6]	MemOut	MemOut	10,333	252

We use a very specific and fixed quantifier instantiation strategy: namely, we instantiate it for the reference process and for the executing process.

Concurrent Shape Analysis. One aspect of our work that distinguishes it from the prior work referenced above is that we apply these ideas to the problem of concurrent shape analysis. In particular, to address the heap, we use abstractions that can more readily make distinctions that are not directly expressible in terms of the program (for instance, the distinction between heap cells to which there are and there are not multiple incoming pointers). Also, the abstraction we use expresses correlations between a single thread’s local state and the global shared state, but does not directly express relations between the state of multiple threads. Relations between multiple threads are captured only by the transformers, unlike in Environment Abstraction, which can additionally use predicates that have been chosen to explicitly relate threads. In the way that our abstractions (partially) correlate locals to globals, but not locals to locals, they exhibit a thread-modular character, except that threads need not be entirely uncorrelated.

The most closely related prior work on concurrent shape analysis is that of Yahav [27], which uses Canonical Abstraction for this purpose. The Quantified Canonical Abstraction domain we use is more precise than Canonical Abstraction, and it allows us to automatically verify, for the first time, linearizability of concurrent data structures in the presence of an unbounded number of threads.

Other Related Work. Counter Abstraction [17] (which has been applied to programs in e.g. [11]) provides a reduction from systems with unboundedly-many processes to finite state, though does not offer much help with the abstract transformers for that finite-state system. Invisible Invariants [22] is another technique that employs thread variables, and works by considering systems with a small number of processes and then attempting to generalize the results to unboundedly-many processes. Work on Split Invariants [20] extends Invisible Invariants using a connection with compositional techniques (such as [21]), yielding an analysis with a process-centric abstraction that computes universally quantified invariants using transformers that resemble ours. In particular, if the assertion logic has a small model property with bound k , then an invariant for unboundedly-many threads can be computed using k instantiations of the invariant. In contrast, we define transformers that are sound (but incomplete) for unboundedly-many threads without a small model property, and using many fewer instantiations.

6 Conclusion

In this paper, we have developed a new shape analysis for fine-grained concurrent programs with an unbounded number of threads and demonstrated that it is precise enough to prove linearizability of useful data structure implementations. This is done by a universal lifting domain construction applied to existing shape analysis domains.

References

1. D. Amit, N. Rinetzky, T. Reps, M. Sagiv, and E. Yahav. Comparison under abstraction for verifying linearizability. In *CAV*, 2007.
2. C. Calcagno, M. J. Parkinson, and V. Vafeiadis. Modular safety checking for fine-grained concurrency. In *SAS*, 2007.

3. E. Clarke, M. Talupur, and H. Veith. Proving Ptolemy right: The environment abstraction framework for model checking concurrent systems. In *TACAS*, 2008.
4. E. M. Clarke. Synthesis of resource invariants for concurrent programs. *TOPLAS*, 2(3), 1980.
5. E. M. Clarke, M. Talupur, and H. Veith. Environment abstraction for parameterized verification. In *VMCAI*, 2006.
6. S. Doherty, L. Groves, V. Luchangco, and M. Moir. Formal verification of a practical lock-free queue algorithm. In *FORTE*, 2004.
7. C. Flanagan and S. Qadeer. Predicate abstraction for software verification. In *POPL*, 2002.
8. C. Flanagan and S. Qadeer. Thread-modular model checking. In *SPIN*, 2003.
9. A. Gotsman, J. Berdine, B. Cook, and M. Sagiv. Thread-modular shape analysis. In *PLDI*, 2007.
10. S. Gulwani, B. McCloskey, and A. Tiwari. Lifting abstract interpreters to quantified logical domains. In *POPL*, 2008.
11. T. A. Henzinger, R. Jhala, and R. Majumdar. Race checking by context inference. In *PLDI*, 2004.
12. M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *TOPLAS*, 12(3), 1990.
13. C.A.R. Hoare. Towards a theory of parallel programming. In *Operating System Techniques*, 1972.
14. C. B. Jones. Specification and design of (parallel) programs. In *IFIP Congress*, 1983.
15. S. K. Lahiri and R. E. Bryant. Predicate abstraction with indexed predicates. *TOCL*, 9(1), 2007.
16. T. Lev-Ami and M. Sagiv. TVLA: A framework for implementing static analyses. In *SAS*, 2000. Available from <http://www.cs.tau.ac.il/~tvla/>.
17. B. D. Lubachevsky. An approach to automating the verification of compact parallel coordination programs I. *Acta Inf.*, 21, 1984.
18. R. Manevich, T. Lev-Ami, M. Sagiv, G. Ramalingam, and J. Berdine. Heap decomposition for concurrent shape analysis. In *SAS*, 2008. To appear.
19. M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *PODC*, 1996.
20. K. S. Namjoshi. Symmetry and completeness in the analysis of parameterized systems. In *VMCAI*, 2007.
21. S. S. Owicki and D. Gries. Verifying properties of parallel programs: An axiomatic approach. *CACM*, 19(5), 1976.
22. A. Pnueli, S. Ruah, and L. D. Zuck. Automatic deductive verification with invisible invariants. In *TACAS*, 2001.
23. A. Podelski and T. Wies. Boolean heaps. In *SAS*, 2005.
24. M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *TOPLAS*, 24(3), 2002.
25. R. K. Treiber. Systems programming: Coping with parallelism. Technical Report RJ 5118, IBM Almaden Research Center, 1986.
26. B. Wachter and B. Westphal. The spotlight principle. In *VMCAI*, 2007.
27. E. Yahav. Verifying safety properties of concurrent Java programs using 3-valued logic. *ACM SIGPLAN Notices*, 36(3), 2001.
28. E. Yahav and G. Ramalingam. Verifying safety properties using separation and heterogeneous abstractions. In *PLDI*, 2004.
29. G. Yorsh, T. Reps, and M. Sagiv. Symbolically computing most-precise abstract operations for shape analysis. In *TACAS*, 2004.