

Threads as Resource for Concurrency Verification

Duy-Khanh Le Wei-Ngan Chin Yong Meng Teo

Department of Computer Science, National University of Singapore

{leduykha,chinwn,teoym}@comp.nus.edu.sg
(Technical Report)

Abstract

In mainstream languages, threads are first-class in that they can be dynamically created, stored in data structures, passed as parameters, and returned from procedures. However, existing verification systems support reasoning about threads in a restricted way: threads are often represented by unique tokens that can neither be split nor shared.

In this paper, we propose “*threads as resource*” to enable more expressive treatment of first-class threads. Our approach allows the ownership of a thread (and its resource) to be flexibly split, combined, and (partially) transferred across procedure and thread boundaries. We illustrate the utility of our approach in handling three problems. First, we use “threads as resource” to verify the *multi-join pattern*, i.e. threads can be shared among concurrent threads and joined multiple times in different threads. Second, using inductive predicates, we show how our approach naturally captures the *threadpool idiom* where threads are stored in data structures. Lastly, we present how thread liveness can be precisely tracked. To demonstrate the feasibility of our approach, we implemented it in a tool, called THREADHIP, on top of an existing PARAHIP verifier. Experimental results show that THREADHIP is more expressive than PARAHIP while achieving comparable verification performance.

Categories and Subject Descriptors D.1.3 [Concurrent Programming]: Parallel programming; F.3.1 [Logics and Meanings of Programs]: Specifying and Verifying and Reasoning about Programs

General Terms Languages, Verification

Keywords Threads as Resource; Concurrency Verification; First-class Threads; Separation Logic

1. Introduction

Threads are considered as first-class in mainstream languages such as Java, C#, and C/C++ in that threads can be treated like objects of any other type: they can be dynamically created, stored in data structures, shared among different threads, passed as parameters, and returned from procedures. Hence, it is desirable for verification systems to support reasoning about first-class threads.

One of the most popular techniques for reasoning about concurrent programs is separation logic [21, 23]. Originally, separation logic was used to verify heap-manipulating sequential programs, with the ability to express non-aliasing in the heap [23]. Separation logic was extended to verify shared-memory concurrent programs, e.g. concurrent separation logic [21], where ownerships of heap objects are considered as *resource*, which can be shared and transferred among concurrent threads. Using fractional permissions [2], one can express full ownerships for exclusive write accesses and partial ownerships for concurrent read accesses. Ownerships of stack variables can also be considered as resource and treated in the same way as heap objects [1].

Separation logic was traditionally extended to verify concurrent programs with parallel composition [21]. Recent works also extended separation logic to handle dynamically-created threads [12, 14, 15, 18, 19]. Hobor [14] allows threads to be dynamically created using fork but does not support join. Gotsman et al. [12] use thread handles to represent threads, while CHALICE [19] uses tokens, VERIFAST [15] uses thread permissions, and PARAHIP [18] uses **and**-conjunctions for the same purpose. A fork operation returns a unique handle/token/**and**-conjunction/permission (collectively referred to as thread token) and a join operation on a thread token causes the joining thread (joiner) to wait for the completion of the thread corresponding to the token (jinee). However, existing works [12, 14, 15, 18, 19] support reasoning about threads in a limited way: unique tokens (representing threads) are not allowed to be split and shared among different threads. As such, existing works do not fully consider threads as first-class.

Reasoning about first-class threads is challenging because threads are dynamic and non-lexically-scoped in nature. A thread can be dynamically created in a procedure (or a thread), but shared and joined in other procedures (or threads). In this paper, we propose an expressive treatment of first-class threads, called “*threads as resource*”. Our approach enables threads’ ownerships to be reasoned about in a similar way to other types of resource. A thread’s ownership is created when it is forked, and destroyed when it is joined. In contrast to ownership of a normal heap object which specifies values of its fields, ownership of a thread carries resource that can be obtained by the joiner when the thread is joined. This is to cater for the intuition that when a joiner joins with a jinee, the joiner expects to obtain (in order to later read or write) certain resource transferred from the jinee. As threads in fork/join programs are typically non-lexically-scoped, we allow threads’ ownerships to be soundly split, combined, and (possibly partially) transferred among procedures and threads.

Our approach elegantly solves at least three verification problems that were not properly supported. First, threads can now be passed as arguments, shared, and joined by different threads. This enables verification of intricate fork/join behaviors such as *multi-join pattern* where a thread is shared and joined in multiple threads. Using our approach, the ownership of the jinee (and its resource)

can be split and transferred (or shared) among the multiple joiners, so that they can respectively join with the jonee and get their corresponding portions of the jonee’s resource. Second, by treating threads in a similar way to heap objects, we can apply current advances in separation logic for heap objects to threads. For example, by combining “threads as resource” with inductive predicates, we can naturally capture a programming idiom called threadpool where threads are stored in data structures. Lastly, we can formally reason about the “liveness” of a thread. We achieve this by adding a special predicate that explicitly indicates when a thread is dead (i.e. after it is joined). Our approach has been implemented in a tool, called THREADHIP, on top of the PARAHIP verifier [18]. Experimental results show that our new tool THREADHIP is more expressive than PARAHIP, whilst achieving comparable verification performance.

The rest of this paper is organized as follows. Section 2 motivates our idea of “threads as resource”. Section 3 introduces our core programming language and specification language, with a focus on modeling threads as resource. Section 4 presents our approach in details. Section 5 discusses three main applications of our approach. Section 6 presents our prototype implementation and experimental results. Section 7 summarizes related work. Section 8 concludes our paper.

2. A Motivating Example

This section illustrates our treatment of “threads as resource” for reasoning about programs with first-class threads. Fig. 1 shows a C-

```

1  data cell { int val; }
2
3  void thread1(cell x, cell y)
4    requires  $x \mapsto \text{cell}(vx) * y \mapsto \text{cell}(vy)$ 
5    ensures  $x \mapsto \text{cell}(vy) * y \mapsto \text{cell}(vx)$ ;
6    { int tmp = x.val;   x.val = y.val;   y.val = tmp; }
7
8  void thread2(thrd t1, cell y)
9    requires  $t1 \mapsto \text{thrd}(y \mapsto \text{cell}(vy))$ 
10   ensures  $y \mapsto \text{cell}(vy + 2) \wedge \text{dead}(t1)$ ;
11   { //  $\{t1 \mapsto \text{thrd}(y \mapsto \text{cell}(vy))\}$ 
12     join(t1);
13     //  $\{y \mapsto \text{cell}(vy) \wedge \text{dead}(t1)\}$ 
14     y.val = y.val+2;
15     //  $\{y \mapsto \text{cell}(vy + 2) \wedge \text{dead}(t1)\}$ 
16   }
17
18  void main()
19    requires emp ensures emp;
20   { cell x = new cell(1);   cell y = new cell(2);
21     //  $\{x \mapsto \text{cell}(1) * y \mapsto \text{cell}(2)\}$ 
22     thrd t1 = fork(thread1,x,y);
23     //  $\{t1 \mapsto \text{thrd}(x \mapsto \text{cell}(2) * y \mapsto \text{cell}(1))\}$ 
24     //  $\{t1 \mapsto \text{thrd}(x \mapsto \text{cell}(2)) * t1 \mapsto \text{thrd}(y \mapsto \text{cell}(1))\}$ 
25     thrd t2 = fork(thread2,t1,y);
26     //  $\{t1 \mapsto \text{thrd}(x \mapsto \text{cell}(2)) * t2 \mapsto \text{thrd}(y \mapsto \text{cell}(3) \wedge \text{dead}(t1))\}$ 
27     join(t1);
28     //  $\{x \mapsto \text{cell}(2) * t2 \mapsto \text{thrd}(y \mapsto \text{cell}(3) \wedge \text{dead}(t1)) \wedge \text{dead}(t1)\}$ 
29     x.val = x.val+1;
30     //  $\{x \mapsto \text{cell}(3) * t2 \mapsto \text{thrd}(y \mapsto \text{cell}(3) \wedge \text{dead}(t1)) \wedge \text{dead}(t1)\}$ 
31     join(t2);
32     //  $\{x \mapsto \text{cell}(3) * y \mapsto \text{cell}(3) \wedge \text{dead}(t1) \wedge \text{dead}(t2)\}$ 
33     assert( $x \mapsto \text{cell}(3) * y \mapsto \text{cell}(3)$ ); /*valid*/
34     destroy(x); destroy(y);
35     //  $\{\text{emp} \wedge \text{dead}(t1) \wedge \text{dead}(t2)\}$ 
36   }

```

Figure 1. A Motivating Example

like program posing challenges to existing verification systems. In the program, the main thread executing the procedure `main` (called `main` thread) forks a new thread `t1` executing the procedure `thread1` (line 22). `thread1` will swap the values of the cells `x` and `y`. `main` then forks another thread `t2` executing the procedure `thread2` with `t1` passed as one of its arguments (line 25). Afterward, `t2` will join with `t1` (line 12) and manipulate the cell `y`, while `main` will also join with `t1` (line 27) but manipulate the cell `x`. In separation logic, a heap node $x \mapsto \text{cell}(vx)$ represents the ownership of an object of type `cell` pointed to by `x` and having the field `val` of `vx` (called ownership of `x` for short).

The program is challenging to verify because (1) fork and join operations on `t1` are non-lexically scoped (i.e. `t1` is forked in `main` but joined in thread `t2`), and (2) `t1` is shared and joined in both `t2` and `main` (i.e. a multi-join). In this program, the ownerships of `x` and `y` are flexibly transferred across thread boundaries, between `main`, `t1` and `t2`, via fork/join calls. To the best of our knowledge, we are not aware of any existing approaches capable of verifying this program. We propose “threads as resource” to verify such programs soundly and modularly. The key points to handle this program are (1) considering `t1` as resource, and (2) allowing it to be split and transferred between `main` and `t2` via fork/join calls.

Our approach is based on the following observation: when a thread (joiner) joins with another thread (jonee), the joiner expects to receive (in order to later read or write) certain resource transferred from the jonee. In the example program, `main` joins with `t1` and expects the ownership of `x` transferred from `t1`, while `t2` joins with `t1` and expects the ownership of `y`. Hence, the verification of the program in Fig. 1 is achieved by introducing the thread ownership $v \mapsto \text{thrd}(\Phi)$ indicating that `v` points to a possibly live thread (as resource) carrying certain resource Φ . A thread having the ownership $v \mapsto \text{thrd}(\Phi)$ can perform a `join(v)`, and yield the resource Φ and a pure predicate `dead(v)` after joining. This special predicate `dead(v)` explicitly indicates that thread `v` is no longer alive. In Fig. 1, when `t1` is forked (line 22), its precondition is consumed and exchanged for the thread’s ownership $t1 \mapsto \text{thrd}(x \mapsto \text{cell}(2) * y \mapsto \text{cell}(1))$ carrying the post-state of `thread1` (i.e. `t1`’s state after it has finished its execution). This is sound and modular as other threads can only observe the post-state of `thread1` when they join with `t1`. Our approach enables the thread’s ownership to be split into $t1 \mapsto \text{thrd}(x \mapsto \text{cell}(2))$ and $t1 \mapsto \text{thrd}(y \mapsto \text{cell}(1))$ (from line 23 to line 24). This allows the latter to be transferred to `t2` while the former remains with `main`. Consequently, having the ownerships of `t1`, both `t2` and `main` can perform `join(t1)` and get the corresponding resource: `t2` obtains the ownership of `y` to write to it, while `main` obtains and writes to `x` (i.e. `t2` and `main` write-share the resource transferred from `t1`). Using our “threads as resource” approach, the program can be verified as both data-race-free and functionally correct.

Our treatment of “threads as resource” allows the ownership of a thread to be flexibly split and transferred. For example, in a program similar to Fig. 1, instead of writing to cells `x` and `y`, both `main` and `t2` may want to concurrently read the value of the cells. Using fractional permissions [2], we could now split the ownership of `t1` from $t1 \mapsto \text{thrd}(x \mapsto \text{cell}(2) * y \mapsto \text{cell}(1))$, into $t1 \mapsto \text{thrd}(x \xrightarrow{0.6} \text{cell}(2) * y \xrightarrow{0.6} \text{cell}(1))$ and $t1 \mapsto \text{thrd}(x \xrightarrow{0.4} \text{cell}(2) * y \xrightarrow{0.4} \text{cell}(1))$, and transfer them into the corresponding codes for `main` and `t2`. This allows `main` and `t2` to be able to read concurrently cells `x` and `y` after joining with the `t1` thread.

In summary, we propose to treat threads as resource, thus allowing threads’ ownerships to be soundly split and transferred across procedure and thread boundaries. This supports first-class threads and enables modular reasoning of intricate concurrent programs with non-lexically-scoped fork/join and multi-join.

P	::=	$data_decl^* global_decl^* proc_decl^*$	Program
$data_decl$::=	$\mathbf{data} C \{ field_decl^* \}$	Data declaration
$field_decl$::=	$type f;$	Field declaration
$global_decl$::=	$global\ type\ v$	Global variable declaration
$proc_decl$::=	$type\ pn(param^*)\ spec^* \{ S \}$	Procedure declaration
$spec$::=	$\mathbf{requires} \Phi_{pr} \mathbf{ensures} \Phi_{po};$	Pre/Post-conditions
$param$::=	$type\ v$	Parameter
$type$::=	$\mathbf{void} \mid \mathbf{int} \mid \mathbf{bool} \mid \mathbf{thrd} \mid C$	Type
e	::=	$v \mid v.f \mid k \mid e_1 + e_2 \mid e_1 = e_2 \mid e_1 \neq e_2$ $v = \mathbf{new} C(v^*) \mid \mathbf{destroy}(v) \mid v = e$ $v = \mathbf{fork}(pn, v^*) \mid \mathbf{join}(v) \mid pn(v^*)$	Variable/field/constant/expression
S	::=	$\mathbf{if} e \mathbf{then} S_1 \mathbf{else} S_2 \mid S_1; S_2$ \dots	Statement

Figure 2. Core Programming Language with First-Class Threads

3. Programming and Specification Languages

In this section, we present our core programming language and specification language, with a focus on modeling threads as resource.

3.1 Programming Language

We use the core programming language in Fig. 2 to convey our idea. A program consists of data declarations ($data_decl^*$), global variable declarations ($global_decl^*$), and procedure declarations ($proc_decl^*$). Each procedure declaration is annotated with pairs of pre/post-conditions (Φ_{pr}/Φ_{po}). New objects of type C can be dynamically created and destroyed using **new** and **destroy**. A **fork** receives a procedure name pn and a list of parameters v^* , creates a new thread executing the procedure pn , and returns an object of **thrd** type representing the newly-created thread. **join**(v) waits for the thread that is pointed to by v to finish its execution. Note that a joiner could be joined in multiple joiners. At run-time, the joiners wait for the joiner to complete its execution. If a joiner waits for an already-completed (or dead) thread, it proceeds immediately without waiting (i.e. the join operation becomes no-op). We do not allow canceling a thread. A thread is dead after it is joined or when the entire program has finished its execution. The semantics of other program statements (such as procedure calls $pn(v^*)$, conditionals, loops, assignments) are standard as can be found in the mainstream languages.

3.2 Specification Language

Comp. formula	Δ	::=	$\Phi \mid \Delta_1 \vee \Delta_2 \mid \Delta_1 \wedge \pi \mid \Delta_1 * \Delta_2 \mid \exists v. \Delta$
Disj. formula	Φ	::=	$\bigvee (\exists v^*. \kappa \wedge \pi)$
Heap formula	κ	::=	$\mathbf{emp} \mid \iota \mid \kappa_1 * \kappa_2$
Atomic heap	ι	::=	$v \xrightarrow{\varepsilon} C(v^*) \mid v \mapsto \mathbf{thrd}(\Phi)$
Pure formula	π	::=	$\alpha \mid \pi_1 \wedge \pi_2 \mid \pi_1 \vee \pi_2 \mid \neg \pi$ $\mid \exists v. \pi \mid \forall v. \pi \mid \mathbf{dead}(v)$
Arith. formula	α	::=	$\alpha_1^t = \alpha_2^t \mid \alpha_1^t \neq \alpha_2^t \mid \alpha_1^t < \alpha_2^t \mid \alpha_1^t \leq \alpha_2^t$
Arith. term	α^t	::=	$k \mid v \mid k \times \alpha^t \mid \alpha_1^t + \alpha_2^t \mid -\alpha^t$
Fractional permission var.	$\varepsilon \in (0,1]$		$v \in \mathbf{Variables}$
$k \in \mathbf{Integer}$ or fractional constants			$C \in \mathbf{Data\ names}$

Figure 3. Grammar for Core Specification Language

Fig. 3 shows our specification language for concurrent programs manipulating “threads as resource”. A classical separation logic formula Φ is in disjunctive normal form. Each disjunct in Φ consists of a heap formula κ and a pure formula π . Furthermore, Δ denotes a composite formula which could always be translated into the Φ form. A pure formula π includes standard equality/inequality, Presburger arithmetic, and a pure predicate $\mathbf{dead}(v)$ indicating

that the thread v has completed its execution. π could also be extended to include other constraints such as set constraints. A heap formula κ consists of multiple atomic heap formulas ι connected with each other via the separation connective $*$. An atomic heap formula $v \xrightarrow{\varepsilon} C(v^*)$ (or heap node) represents the fact that the current thread has a certain fractional permission ε to access an object of type C pointed to by v . v^* captures a list of variables representing the fields of the object v .

The atomic heap formula $v \mapsto \mathbf{thrd}(\Phi)$ (or thread node) captures our idea of “threads as resource”: v points to a thread carrying certain resource Φ , which is available after the thread is joined. By representing threads as heap resource, we allow them to be flexibly split and transferred in a similar way to other types of resource such as heap nodes. Note that thread nodes themselves are non-fractional, but their resources can already be flexibly split. Furthermore, no resource leakage from threads is possible since we explicitly track when each thread becomes dead.

Our approach allows for expressive reasoning about threads and their liveness. For example, a formula $t \mapsto \mathbf{thrd}(\Phi) \vee \mathbf{dead}(t)$ specifies the fact that the thread t could be either alive or dead. On the other hand, a formula with $t \mapsto \mathbf{thrd}(\Phi) \wedge \mathbf{dead}(t)$ indicates the fact that t is already dead and hence the resource Φ can be safely released.

4. Threads as Resource

In this section, we first introduce our forward verification rules. We then present a set of sub-structural rules for manipulating resource, especially threads as resource. Finally, we discuss the soundness of our approach.

4.1 Forward Verification Rules

Our verification system is built on top of entailment checking:

$$\Delta_A \vdash \Delta_C \rightsquigarrow \Delta_R$$

This entailment checks if antecedent Δ_A is precise enough to imply consequent Δ_C , and computes the residue Δ_R for the next program state (we write $\Delta_A \vdash \Delta_C$ when ignoring the residue). For example:

$$x \xrightarrow{0.6} cell(1) * y \xrightarrow{0.6} cell(2) \vdash x \xrightarrow{0.6} cell(1) \rightsquigarrow y \xrightarrow{0.6} cell(2)$$

Fig. 4 presents our forward verification rules. Here we only focus on three key constructs affecting threads’ resource: procedure call, fork, and join. Forward verification is formalized using Hoare’s triple for partial correctness: $\{\Phi_{pr}\}P\{\Phi_{po}\}$. Given a program P starting in a state satisfying the pre-condition Φ_{pr} , if the program terminates, it will do so in a state satisfying the post-condition Φ_{po} . For simplicity, in this paper, we describe the verification rules with one pair of pre/post condition. Multiple pre/post specifications can be handled in the same way as [5].

$\frac{\text{spec}(pn) := pn(w^*) \text{ requires } \Phi_{pr} \text{ ensures } \Phi_{po}; \{s\} \quad \Delta \vdash \Phi_{pr} \rightsquigarrow \Delta_1 \quad \Delta_2 \stackrel{\text{def}}{=} \Delta_1 * \Phi_{po}}{\{\Delta\} pn(w^*) \{\Delta_2\}} \quad \text{CALL}$
$\frac{\text{spec}(pn) := pn(w^*) \text{ requires } \Phi_{pr} \text{ ensures } \Phi_{po}; \{s\} \quad \Delta \vdash \Phi_{pr} \rightsquigarrow \Delta_1 \quad \Delta_2 \stackrel{\text{def}}{=} \Delta_1 * v \mapsto \text{thrd}(\Phi_{po})}{\{\Delta\} v := \text{fork}(pn, w^*) \{\Delta_2\}} \quad \text{FORK}$
$\{\Delta * v \mapsto \text{thrd}(\Phi_{po})\} \text{ join}(v) \{\Delta * \Phi_{po} \wedge \text{dead}(v)\} \quad \text{JOIN-1}$
$\{\Delta \wedge \text{dead}(v)\} \text{ join}(v) \{\Delta \wedge \text{dead}(v)\} \quad \text{JOIN-2}$

Figure 4. Selected Verification Rules

In order to perform a procedure call (**CALL**), the caller should be in a state Δ that can entail the pre-condition Φ_{pr} of the callee (i.e. the procedure pn). $\text{spec}(pn)$ denotes the specification of the procedure pn . For conciseness, we omit the substitutions that link actual and formal parameters of the procedure prior to the entailment. After the entailment, the caller subsumes the post-condition Φ_{po} of the callee with the residue Δ_1 to form a new state Δ_2 . Ownerships are transferred across procedure boundaries, from the caller to the callee via the entailment of the pre-condition and from the callee to the caller via the spatial conjunction on the post-condition.

Similarly, when performing a fork (**FORK**), the forker should be in a state Δ that can entail the pre-condition Φ_{pr} of the forkee (i.e. the newly-created thread executing the procedure pn). Afterward, a new thread node $v \mapsto \text{thrd}(\Phi_{po})$ carrying the post-condition Φ_{po} of the forkee is created. The thread node is then combined with the residue Δ_1 to form a new state Δ_2 . The thread node is considered as resource in Δ_2 ; hence, it can be flexibly split and transferred in subsequent parts of the program. The **FORK** rule is sound since other threads can only observe the post-state of the forkee when joining with it. It also ensures modularity as the forker only knows the pre/post-conditions of the forkee.

When joining a thread (**JOIN-1**), the joiner simply exchanges the thread node, which carries a resource Φ_{po} , with the resource itself. Each joiner could be joined by multiple joiners. Our verification rules are based on the observation that when a joiner joins with a joiner, the joiner is expecting to receive certain resource transferred to it from the joiner. Hence, each joiner will receive the current resource carried by the thread node. After a thread has been joined, it becomes dead (indicated by the pure dead predicate). Joining a dead thread is equivalent to a no-op (**JOIN-2**).

Using our verification rules, a **CALL** can be modeled as a **FORK** immediately followed by a **JOIN**. As threads are considered as resource, fork and join operations can be in different lexical scopes and thread nodes can be transferred across procedure and thread boundaries. Furthermore, if there is a recursive fork call in a procedure (also called nested fork) such as the parallel Fibonacci program¹, the verification proceeds normally: a new thread node corresponding to the newly-created thread executing the procedure is created. Therefore, in our system, a nested fork is handled in the same way as a normal fork.

4.2 Manipulating “Threads as Resource”

The notion of “threads as resource” plays a critical role in our approach as it enables threads to be treated in a similar way to other objects: a thread node can be created, stored, split, and transferred

(or shared) among multiple threads, allowing them to join and to receive suitable resource after joining.

Our sub-structural rules for manipulating resource are presented in Fig. 5. The rules rearrange resource in a separation logic formula into equivalent forms. We denote resource equivalence as \iff . By resource equivalence, we mean that the total resource on the left and the right sides of \iff are the same. Our approach allows resource to be split, combined, and transferred across procedures and threads, while it guarantees that the total resource remains unchanged. The rules **R-DISJ**, **R-CONJ**, **R-SCONJ**, **R-COM**, and **R-EMP** are straightforward. With fractional permissions ε , heap nodes can be split and combined in a standard way (**R-FRAC**). The left-to-right direction indicates permission splitting while the right-to-left indicates permission combining. We also allow thread nodes to be split and combined (**R-THRD1**). Splitting a thread node (left-to-right) will split the resource carried by the node while combining thread nodes (right-to-left) will combine the resource of the constituent nodes. Finally, when a thread is dead, its carried resource can be safely released (**R-THRD2**).

4.3 Soundness

This section presents the soundness of our approach. The proof is based on the soundness proof of Gotsman et al. [12], which is tailored towards locks and threads (but not threads as resource). Our proof, on the other hand, is tailored towards threads as resource. We first present the memory model of the specification language presented in Section 3.2. We then introduce the interleaving semantics of the programming language presented in Section 3.1. Finally, we prove the soundness of our approach with respect to the operational semantics.

Memory Model.

Our basic memory model is extended from the model proposed in [12]. As a simplification, we consider objects of type `cell` and `thrd` only. User-defined data types can be supported by extending from `cell` to objects with fields. This, however, unnecessarily complicates the memory model. Instead, we focus on modeling threads as resource. We also impose a restriction whereby thread nodes can only be passed between the caller and callee of the same thread, or between different threads (i.e. from forker to forkee, or from joiner to joiner). Thread nodes, therefore, are not allowed to belong to external environment such as lock invariants. This is to avoid circularity when the two thread nodes of two different threads refer to each other as their carried resource.

In our memory model, formulas are interpreted over program state $\sigma \in \Sigma$, defined as follows:

¹<http://loris-7.ddns.comp.nus.edu.sg/~project/threadhip/>

$\frac{\Phi_1 \iff \Phi'_1 \quad \Phi_2 \iff \Phi'_2}{\Phi_1 \vee \Phi_2 \iff \Phi'_1 \vee \Phi'_2}$	<u>R-DISJ</u>
$\frac{\kappa \iff \kappa'}{\kappa \wedge \pi \iff \kappa' \wedge \pi}$	<u>R-CONJ</u>
$\frac{\kappa_1 \iff \kappa'_1 \quad \kappa_2 \iff \kappa'_2}{\kappa_1 * \kappa_2 \iff \kappa'_1 * \kappa'_2}$	<u>R-SCONJ</u>
$\kappa_1 * \kappa_2 \iff \kappa_2 * \kappa_1$	<u>R-COM</u>
$\kappa * \text{emp} \iff \kappa$	<u>R-EMP</u>
$v \xrightarrow{\varepsilon_1 + \varepsilon_2} C(v^*) \iff v \xrightarrow{\varepsilon_1} C(v^*) * v \xrightarrow{\varepsilon_2} C(v^*)$	<u>R-FRAC</u>
$v \mapsto \text{thrd}\langle \Phi_1 * \Phi_2 \rangle \iff v \mapsto \text{thrd}\langle \Phi_1 \rangle * v \mapsto \text{thrd}\langle \Phi_2 \rangle$	<u>R-THRD1</u>
$v \mapsto \text{thrd}\langle \Phi \rangle \wedge \text{dead}(v) \implies \Phi$	<u>R-THRD2</u>

Figure 5. Sub-structural Rules

$\text{Loc} \stackrel{\text{def}}{=} \{1, 2, \dots\}$	$\text{Perm} \stackrel{\text{def}}{=} (0, 1]$
$\text{Var} \stackrel{\text{def}}{=} \{x, y, \dots\}$	$\text{TIDS} \stackrel{\text{def}}{=} \{1, 2, \dots\}$
$\text{Val} \stackrel{\text{def}}{=} \{\dots, -1, 0, 1, \dots\}$	$\text{TStates} \stackrel{\text{def}}{=} \{F, D\}$
$\Sigma \stackrel{\text{def}}{=} \text{Stack} \times \text{Heap} \times \text{THRDS}$	
$\text{Heap} \stackrel{\text{def}}{=} \text{Loc} \rightarrow \text{cell}(\text{Val} \times \text{Perm})$	
$\text{Stack} \stackrel{\text{def}}{=} \text{Var} \rightarrow \text{Val}$	
$\text{THRDS} \stackrel{\text{def}}{=} \text{TIDS} \rightarrow \text{THR}(\text{TStates} \times \Pi)$	

Program state $\sigma = (s, h, t) \in \Sigma$ consists of a stack $s \in \text{Stack}$, a heap $h \in \text{Heap}$, and a threadpool $t \in \text{THRDS}$. Each heap cell is associated with a fractional permission $\varepsilon \in \text{Perm}$. THRDS maps a thread identifier $k \in \text{TIDS}$ into a thread with a state $\diamond \in \text{TStates}$ ($F = \text{forked}$, and $D = \text{dead}$) and a resource $\Phi \in \Pi$. Our memory model is mostly similar to that described in [12]; it additionally captures a threadpool of threads with their states (i.e. their liveness) and their carried resource.

We denote $f(x) \downarrow$ if the function f is defined on x , $f(x) \uparrow$ if the function f is undefined on x . We write $f = []$ if $\text{dom}(f)$ is empty. We denote $f[(x, v)]$ (defined only if $f(x) \uparrow$) as a function that has the same value as f everywhere, except for x where it has the value v . We now define $*$ on program states, which interprets the $*$ -connective in our logic.

For $s_1, s_2 \in \text{Stack}$,

$$s_1 \# s_2 \iff \forall x. s_1(x) \downarrow \wedge s_2(x) \downarrow \implies (\exists v. s_1(x) = v \wedge s_2(x) = v)$$

If $s_1 \# s_2$, then

$$s_1 * s_2 \stackrel{\text{def}}{=} \{(x, v) \mid (s_1(x) = v \wedge s_2(x) \uparrow) \vee (s_1(x) \uparrow \wedge (s_2(x) = v))\},$$

otherwise $s_1 * s_2$ is undefined. Note that, for simplicity, we do not consider permissions on program variables which can be separately exented as described in [1].

For $h_1, h_2 \in \text{Heap}$,

$$h_1 \# h_2 \iff (\forall l. h_1(l) \downarrow \wedge h_2(l) \downarrow \implies (\exists v, \varepsilon_1, \varepsilon_2. h_1(l) = \text{cell}(v, \varepsilon_1) \wedge h_2(l) = \text{cell}(v, \varepsilon_2) \wedge \varepsilon_1 + \varepsilon_2 \leq 1))$$

If $h_1 \# h_2$, then

$$h_1 * h_2 \stackrel{\text{def}}{=} \{(l, \text{cell}(v, \varepsilon) \mid (h_1(l) = \text{cell}(v, \varepsilon) \wedge h_2(l) \uparrow) \vee (h_1(l) \uparrow \wedge h_2(l) = \text{cell}(v, \varepsilon)) \vee (h_1(l) = \text{cell}(v, \varepsilon_1) \wedge h_2(l) = \text{cell}(v, \varepsilon_2) \wedge \varepsilon_1 + \varepsilon_2 = \varepsilon)\},$$

otherwise $h_1 * h_2$ is undefined.

For $t_1, t_2 \in \text{THRDS}$,

$$t_1 \# t_2 \iff (\forall k. t_1(k) \downarrow \wedge t_2(k) \downarrow \implies (\exists v, h_1, s_1, h_2, s_2, i. t_1(k) = \text{THR}(\diamond, \Phi_1) \wedge t_2(k) = \text{THR}(\diamond, \Phi_2) \wedge s_1 \# s_2 \wedge h_1 \# h_2 \wedge (h_1, s_1, t_1, i) \models_k \Phi_1 \wedge (h_2, s_2, t_2, i) \models_k \Phi_2))$$

For thread states,

$$F * F = F \quad D * D = D \quad F * D = D$$

If $t_1 \# t_2$, then

$$t_1 * t_2 \stackrel{\text{def}}{=} \{(k, \text{THR}(\diamond, \Phi) \mid (t_1(k) = \text{THR}(\diamond, \Phi) \wedge t_2(k) \uparrow) \vee (t_1(k) \uparrow \wedge t_2(k) = \text{THR}(\diamond, \Phi)) \vee (t_1(k) = \text{THR}(\diamond_1, \Phi_1) \wedge t_2(k) = \text{THR}(\diamond_2, \Phi_2) \wedge \diamond = \diamond_1 * \diamond_2 \wedge \Phi = \Phi_1 * \Phi_2)\},$$

otherwise $t_1 * t_2$ is undefined. We lift $*$ to states and set of states pointwise.

The satisfaction relation for our specification language formulas is presented in Fig 6. One can easily show that the sub-structural rules in Fig. 5 are sound with respect to the satisfaction relation. A formula is interpreted with respect to a thread identifier $k \in \{m\} \cup \text{TIDS}$ (m is the identifier of the main thread), a stack s , a heap h , a threadpool t , and an interpretation i mapping logical variables to values. We assume a function $\llbracket e \rrbracket_{(s, i)}$ that evaluates an expression e with respect to the stack s and the interpretation i . We write $\llbracket e \rrbracket_s$ when s is sufficient to evaluate e . Note that, in our definitions, the predicate $t_1 \# t_2$ refers to the satisfaction relation \models , which is defined based on $\#$. However, due to the restriction that two thread nodes of two different threads cannot refer to each other as their carried resource, $\#$, $*$, and \models are simultaneously defined by induction which is guaranteed to terminate. Let $\llbracket \Phi \rrbracket_i^k$ denote the set of states in which the formula Φ is valid with respect to a thread identifier k and an interpretation i . We write $\llbracket \Phi \rrbracket^k$ when the deno-

Satisfaction Relation: $(s, h, t, i) \models_k \Phi$

$(s, h, t, i) \models_k \exists x. \Phi$	$\Leftrightarrow \exists v. (s, h, t, i[(x, v)]) \models_k \Phi$
$(s, h, t, i) \models_k \text{emp}$	$\Leftrightarrow s = [] \wedge h = [] \wedge t = []$
$(s, h, t, i) \models_k \Phi_1 * \Phi_2$	$\Leftrightarrow \exists s_1, s_2, h_1, h_2, t_1, t_2. s = s_1 * s_2 \wedge h = h_1 * h_2 \wedge t = t_1 * t_2 \wedge (s_1, h_1, t_1, i) \models_k \Phi_1 \wedge (s_2, h_2, t_2, i) \models_k \Phi_2$
$(s, h, t, i) \models_k \Phi_1 \vee \Phi_2$	$\Leftrightarrow (s, h, t, i) \models_k \Phi_1 \vee (s, h, t, i) \models_k \Phi_2$
$(s, h, t, i) \models_k \kappa \wedge \pi$	$\Leftrightarrow (s, h, t, i) \models_k \kappa \wedge (s, [], t, i) \models_k \pi$
$(h, s, t, i) \models_k x \xrightarrow{\varepsilon} \text{cell}(v)$	$\Leftrightarrow \exists l. s = [(x, l)] \wedge h = [(l, \text{cell}(v, \varepsilon))]$
$(s, h, t, i) \models_k x \mapsto \text{thrd}(\Phi)$	$\Leftrightarrow \exists id, s_1, t_1. s = s_1 * [(x, id)] \wedge t = t_1 * [(id, \text{THRD}(F, \Phi))] \wedge (s_1, h, t_1, i) \models_k \Phi$
$(s, h, t, i) \models_k \text{dead}(x)$	$\Leftrightarrow \exists id. s = [(x, id)] \wedge t = [(id, \text{THRD}(D, \text{emp}))]$
$(s, h, t, i) \models_k \pi_1 \text{op} \pi_2$	$\Leftrightarrow (s, h, t, i) \models_k \pi_1 \wedge (s, h, t, i) \models_k \pi_2$, where $\text{op} \in \{\wedge, \vee\}$
$(s, h, t, i) \models_k \neg \pi$	$\Leftrightarrow \neg((s, h, t, i) \models_k \pi)$
$(s, h, t, i) \models_k \exists x. \pi$	$\Leftrightarrow \exists v. (s, h, t, i[(x, v)]) \models_k \pi$
$(s, h, t, i) \models_k \forall x. \pi$	$\Leftrightarrow \forall v. (s, h, t, i[(x, v)]) \models_k \pi$
$(s, h, t, i) \models_k x_1 \text{op} x_2$	$\Leftrightarrow [x_1]_{(s,i)} \downarrow \wedge [x_2]_{(s,i)} \downarrow \wedge ([x_1]_{(s,i)} \text{op} [x_2]_{(s,i)})$, where $\text{op} \in \{=, \neq, <, \leq\}$

Figure 6. Satisfaction Relation for Specification Language Formulas

tation of Φ is sufficient with respect to a thread identifier k . When the denotation of a formula Φ does not depend on the thread identifier and the interpretation of logical variables, we simply write $\llbracket \Phi \rrbracket$. Similar to [12], we say that a predicate $p \subseteq \Sigma$ is *precise* if for any state σ , there exists at most one substate σ_0 that satisfies p .

Operational Semantics.

We consider a well-formed program P with the main procedure $\text{main}() \{S_m\}$ and a set of procedures $f^i(v_i^*) \{S_i\}$ ($i = 1..n$). Let Γ denote the procedure context in the program P , i.e. $\Gamma = \{ \Phi_{pr}^i, f^i(v_i^*) \{ \Phi_{po}^i \} \mid i = 1..n \}$. The Hoare's triples $\{ \Phi_{pr}^i \} S \{ \Phi_{po}^i \}$ presented in Section 4.1 are implicitly defined under Γ , that is $\Gamma \vdash \{ \Phi_{pr}^i \} S \{ \Phi_{po}^i \}$.

Definition 1 (Well-formedness). *A program is well-formed if the following conditions hold:*

- In the program text, there exists a procedure called *main*, which indicates the entry point of the program.
- Procedure names are unique within a program. Procedure parameters are unique within a procedure. Free variables in the body of a procedure are the procedure parameters.
- A normal procedure call or a fork statement mentions only procedure names defined in the program text. The number of actual parameters and formal parameters are equal.

The interleaving semantics of the program P is presented in Fig. 7. The semantics is defined by a transition relation \rightsquigarrow_P that transforms pairs of program counters (which map a thread identifier into its corresponding remaining statement, i.e. $pc \in (\text{TIDS} \cup \{m\}) \rightarrow SS$, where SS is the set of statements), and states (s, h, t) . The relation \rightsquigarrow_P is defined as the least one satisfying the rules in Fig. 7. We denote \rightsquigarrow_P^* as the reflexive and transitive closure of \rightsquigarrow_P . The initial program counter pc_0 is $[(m, S_m)]$ which contains only the main thread whose identifier is m and whose remaining statement is S_m .

In Fig. 7, $\text{spec}(pn)$ denotes the specification of the procedure pn in the program, $[v/w]S$ denotes the substitution in S where w is substituted by v . The rules for fork and join are of special interest. In the fork rule, a new thread is spawned and the return value v points to its identifier j . The resource carried by j is captured in t . We explicitly add an end statement to signify the end of each

newly spawned thread.² As a quick observation, a thread identifier corresponds to a thread node in our logic. Any threads (joiners) knowing the identifier can perform a join operation to join with the newly-created thread (jonee). In the join rule, if the jonee has not yet finished its execution (i.e. it is not in a D state), the joiners have to wait for the jonee to finish its execution. Note that when a jonee is joined, it will not be removed from the threadpool. This allows for the multi-join pattern and enables the joiners to immediately proceed without waiting in case the jonee has already finished its execution. There is a direct relation between the D state of a thread during run-time and its dead predicate during verification-time.

Definition 2 (Safety). *A program P is safe when running from an initial state σ_0 if it is not the case that $pc_0, \sigma_0 \rightsquigarrow_P^* pc, \top$ for some pc .*

Proof.

The proof is based on the soundness proof of Gotsman et al. [12]. We first present a thread-local semantics based on a thread-local forward predicate transformer $\text{Post}_k^\gamma(\cdot)$, and define the notion of validity of Hoare's triples for program statements with respect to the thread-local semantics (Definition 3). We then prove the soundness of our approach with respect to the thread-local semantics (Lemma 1). Finally, we prove that the thread-location semantics is adequate with respect to the interleaving semantics, which justifies the soundness of our approach (Lemma 2 and 3). Additionally, in a similar way to [12], using the thread-local semantics, we prove that provable programs are race-free (Lemma 4).

For a state σ and a predicate p , define

$$\text{rest}(\sigma, p) = \begin{cases} \sigma_1, & \text{if } \sigma = \sigma_1 * \sigma_2 \text{ and } \sigma_2 \in p \\ \top, & \text{otherwise} \end{cases}$$

Let the domain \mathcal{D} be topped powerset of states. Then, we define a thread-local forward predicate transformer $\text{Post}_k^\gamma(S) : \mathcal{D} \rightarrow \mathcal{D}$ for every thread k , statement S , and semantical procedure context γ consisting of triples of the form $\{p\}f(v^*)\{q\}$ where $p, q \subseteq \Sigma$ and p is precise. For statement S other than fork, join, and procedure call, $\text{Post}_k^\gamma(S) = \text{Post}_k(S)$ as defined in [12]. Here, we define those for fork, join, and procedure call. Let

$$\text{Post}_k^\gamma(f(v^*), (s, h, t)) = \{(s', h', t')\} * q$$

if $\{p\}f(v^*)\{q\} \in \gamma$, and $\text{rest}((s, h, t), p) = \{(s', h', t')\}$, and $\text{Post}_k^\gamma(f(v^*), (s, h, t)) = \top$ otherwise. Let

²We also add an end statement at the end of the main procedure to signify the end of the main thread.

Transition Relation for Atomic Commands: $S, \sigma \rightsquigarrow_k \sigma'$

$$\begin{array}{ll}
x = \text{new cell}(v), (s[x, l_1], h, t) & \rightsquigarrow_k (s[x, l_2], h[(l_2, \text{cell}(v, 1))], t), \quad \text{if } h(l_2) \uparrow \\
\text{destroy}(x), (s, h[\llbracket x \rrbracket_s, \text{cell}(v, 1)], t) & \rightsquigarrow_k (s, h, t) \\
x = e, (s[x, v], h, t) & \rightsquigarrow_k (s[x, \llbracket e \rrbracket_{s[(x, v)]}], h, t) \\
S, (s, h, t) & \rightsquigarrow_k \top, \quad \text{otherwise}
\end{array}$$

Transition Relation for Programs: $pc, \sigma \rightsquigarrow_P pc', \sigma'$

$$\begin{array}{c}
\frac{k \in \text{TIDS} \quad S_1, (s, h, t) \rightsquigarrow_k \sigma}{pc[(k, S_1; S_2)], (s, h, t) \rightsquigarrow_P pc[(k, S_2)], \sigma} \quad (S_1 \text{ is an atomic command}) \\
\\
\frac{k \in \text{TIDS}}{pc[(k, \text{if true then } S_1 \text{ else } S_2; S)], (s, h, t) \rightsquigarrow_P pc[(k, S_1; S)], (s, h, t)} \\
\\
\frac{k \in \text{TIDS}}{pc[(k, \text{if false then } S_1 \text{ else } S_2; S)], (s, h, t) \rightsquigarrow_P pc[(k, S_2; S)], (s, h, t)} \\
\\
\frac{k \in \text{TIDS}}{pc[(k, \text{if } e \text{ then } S_1 \text{ else } S_2; S)], (s, h, t) \rightsquigarrow_P pc[(k, \text{if } \llbracket e \rrbracket_s \text{ then } S_1 \text{ else } S_2; S)], (s, h, t)} \\
\\
\frac{\begin{array}{l} \text{spec}(pn) := pn(w_1, \dots, w_n) \text{ requires } \Phi_{pr} \text{ ensures } \Phi_{po}; \{ S_1 \} \\ k \in \text{TIDS} \quad \forall i. \llbracket v_i \rrbracket_{s \downarrow} \quad \rho = [v_1/w_1, \dots, v_n/w_n] \quad S'_1 = \rho S_1 \end{array}}{pc[(k, pn(v_1, \dots, v_n); S)], (s, h, t) \rightsquigarrow_P pc[(k, S'_1; S)], (s, h, t)} \\
\\
\frac{\begin{array}{l} \text{spec}(pn) := pn(w_1, \dots, w_n) \text{ requires } \Phi_{pr} \text{ ensures } \Phi_{po}; \{ S_1 \} \\ k, j \in \text{TIDS} \quad t(j) \uparrow \quad \llbracket v \rrbracket_{s \downarrow} \quad \forall i. \llbracket v_i \rrbracket_{s \downarrow} \quad \rho = [v_1/w_1, \dots, v_n/w_n] \quad S'_1 = \rho S_1 \quad \Phi'_{po} = \rho \Phi_{po} \end{array}}{pc[(k, v = \text{fork}(pn, v_1, \dots, v_n); S)], (s, h, t) \rightsquigarrow_P pc[(k, S), (j, S'_1; \text{end})], (s[(v, j)], h, t[(j, \text{THRD}(F, \Phi'_{po}))])} \\
\\
\frac{k \in \text{TIDS} \quad \llbracket v \rrbracket_{s \downarrow}}{pc[(k, \text{join}(v); S), (\llbracket v \rrbracket_s, \text{end})], (s, h, t[(\llbracket v \rrbracket_s, \text{THRD}(F, \Phi))]) \rightsquigarrow_P pc[(k, S)], (s, h, t[(\llbracket v \rrbracket_s, \text{THRD}(D, \text{emp}))])} \\
\\
\frac{k \in \text{TIDS} \quad \llbracket v \rrbracket_{s \downarrow}}{pc[(k, \text{join}(v); S)], (s, h, t[(\llbracket v \rrbracket_s, \text{THRD}(D, \text{emp}))]) \rightsquigarrow_P pc[(k, S)], (s, h, t[(\llbracket v \rrbracket_s, \text{THRD}(D, \text{emp}))])}
\end{array}$$

Figure 7. Operational Semantics of Well-formed Programs. (\top indicates a fault.)

$$\text{Post}_k^\gamma(v = \text{fork}(f, v^*), (s, h, t)) = \{(s'[(v, j)], h', t'[(j, \text{THRD}(F, \Phi_{po}))])\}$$

where $j \in \text{TIDS}$, if $\{p\}f(v^*)\{q\} \in \gamma$, Φ_{po} is the post-condition of $f(v^*)$, and $\text{rest}((s, h, t), p) = \{(s', h', t')\}$, and $\text{Post}_k^\gamma(v = \text{fork}(f, v^*), (s, h, t)) = \top$ otherwise. Let

$$\text{Post}_k^\gamma(\text{join}(v), (s, h, t[(\llbracket v \rrbracket_s, \text{THRD}(F, \Phi))])) = \{(s, h, t[(\llbracket v \rrbracket_s, \text{THRD}(D, \text{emp}))])\} * \llbracket \Phi \rrbracket,$$

$$\text{Post}_k^\gamma(\text{join}(v), (s, h, t[(\llbracket v \rrbracket_s, \text{THRD}(D, \text{emp}))])) = \{(s, h, t[(\llbracket v \rrbracket_s, \text{THRD}(D, \text{emp}))])\},$$

if $\{p\}f(v^*)\{q\} \in \gamma$, $\llbracket v \rrbracket_{s \downarrow}$, and $\text{Post}_k^\gamma(\text{join}(v), (s, h, t)) = \top$ otherwise. The requirement that the preconditions in γ are precise is to ensure the determinism of splitting the state at fork and procedure call. $\text{Post}_k^\gamma()$ for other non-atomic commands are defined as in [12] with the help of a function $F_k(\gamma, S, \sigma)$. We first assume a control-flow relation G where $(v, S, v') \in G$ indicates the program points v before and v' after the statement S in the program's control flow graph ($G_{pr}(S) = v$ and $G_{po}(S) = v'$ for short). In addition, let $\text{proc}(v)$ denote the name of the procedure to which the program point v belongs (*main* for the main procedure in the main thread). We sometimes overload $\text{proc}(S)$ to denote the name of the procedure to which the statement S belongs. We also assume a function g that takes a program point and returns the corresponding program state at that point. In addition, we assume that

a thread executing a procedure f will start and end at the program points start_f and stop_f respectively. Then, $F_k(\gamma, S, \sigma)(g) = g'$ where $g'(\text{start}_f) = \sigma$ and for every program point v_2 such that $v_2 \neq \text{start}_f$, $g'(v_2) = \bigsqcup_{(v_1, S, v_2) \in G} \text{Post}_k(S, g(v_1))$. Hence, $\text{Post}_k(S, \sigma) = (\text{lfp}(F_k(\gamma, S, \sigma)))(\text{stop})$. For a procedure context Γ , we denote $\llbracket \Gamma \rrbracket$ as its corresponding semantical procedure context, i.e. $\{\Phi_{pr}\}f(v^*)\{\Phi_{po}\} \in \Gamma$ iff $\{\llbracket \Phi_{pr} \rrbracket\}f(v^*)\{\llbracket \Phi_{po} \rrbracket\} \in \gamma$. We now extend the validity \models_k of Hoare's triples with respect to the thread-local semantics for thread k .

Definition 3. For a statement S ,

$$\Gamma \models_k \{\Phi_{pr}\}S\{\Phi_{po}\} \Leftrightarrow \text{Post}_k^{\llbracket \Gamma \rrbracket}(S, \llbracket \Phi_{pr} \rrbracket^k) \sqsubseteq \llbracket \Phi_{po} \rrbracket^k.$$

Lemma 1 (Soundness with respect to thread-local semantics). *If $\Gamma \vdash \{\Phi_{pr}\}S\{\Phi_{po}\}$, then for all $k \in \text{TIDS}$, $\Gamma \models_k \{\Phi_{pr}\}S\{\Phi_{po}\}$.*

Proof. Standard verification rules (such as those for creating a new object, deallocating an existing object, etc.) have been proven sound with respect to the thread-local semantics in [12]. Here, it is easy to show that the definition of $\text{Post}_k^\gamma()$ implies the soundness of **CALL**, **FORK**, **JOIN-1**, and **JOIN-2** (presented in Fig. 4) with respect to the thread-local semantics. We can then perform induction on the derivation of $\Gamma \vdash \{\Phi_{pr}\}S\{\Phi_{po}\}$. \square

The following lemma states that the thread-local semantics is an over-approximation of the interleaving operational semantics described in Fig. 7.

Lemma 2 (Over-approximating Lemma). *Consider a program P with the main procedure $\text{main}() \{S_m\}$ equipped with a precondition $p \in \mathcal{D}$, a set of procedures $f^i(v_i^*) \{S_i\}$ ($i = 1..n$), and a semantical procedure context $\gamma = \{\{p^i\} f^i(v_i^*) \{q^i\} \mid i = 1..n\}$ such that $\text{Post}_k^\gamma(S_i, p_i) \sqsubseteq q_i$ for all $i = 1..n$ and $k \in \text{TIDS}$. Let $g_m(\text{main}) = \text{lf}_P(F_m(\gamma, S, p))$ and $g_k(f^i) = \text{lf}_P(F_k(\gamma, S_i, p^i))$ for all $k \in \text{TIDS}$ and $i = 1..n$. Then for any state $\sigma_0 = (h_0, s_0, \square)$ such that*

$$\{\sigma_0\} \sqsubseteq p \quad (1)$$

whenever $pc_0, \sigma_0, \rightsquigarrow_P pc, \sigma_1$, it is the case that

$$\{\sigma_1\} \sqsubseteq \left(\bigotimes_{\{k \mid pc(k) \downarrow\}} g_k(G_{pr}(pc(k))) \right). \quad (2)$$

Proof. The proof is done by induction on the length m of the derivation of σ_1 . For $m = 0$, it trivially holds. Now, suppose that $pc_0, \sigma_0 \rightsquigarrow_P^* pc[(j, S; S'), \sigma_1 \rightsquigarrow_P pc'[(j, S'), \sigma_2, (v, S, v') \in G]$ and

$$\{\sigma_1\} \sqsubseteq \left(\bigotimes_{\{k \mid pc(k) \downarrow\}} g_k(G_{pr}(pc(k))) \right) * g_j(v),$$

we have to prove that

$$\{\sigma_2\} \sqsubseteq \left(\bigotimes_{\{k \mid pc'(k) \downarrow\}} g_k(G_{pr}(pc'(k))) \right) * g_j(v') \quad (3)$$

The proof for statements other than fork, join, and procedure call has been given in [12]. Here, we focus on those for fork, join, and procedure call.

1. S is $v = \text{fork}(f^i, v_1, \dots, v_n)$. Assume that $\sigma_1 = \sigma'_1 * \sigma''_1$, where $\sigma'_1 \in g_j(v)$, $\sigma''_1 = \bigotimes_{\{k \mid pc(k) \downarrow\}} g_k(G_{pr}(pc(k)))$, $\sigma'_1 = (s, h, t)$, $pc' = pc[j' : S_i]$, $\sigma_2 = (s[(v, j'), h, t[(j', \text{THRD}(F, \Phi_{po}^i))]]) * \sigma''_1$, $\text{Post}_k^\gamma(S, \sigma'_1) \sqsubseteq g_j(v')$; otherwise, the right-hand side of Eq. 3 is \top . Now, we have to prove that $\{\sigma_2\} \sqsubseteq g_j(v') * \{\sigma'_1\}$. Let $\sigma_3 = (s[(v, j'), h, t[(j', \text{THRD}(F, \Phi_{po}^i))]])$, then $\text{Post}_k^\gamma(S, \sigma'_1) \sqsubseteq g_j(v')$ implies $\{\sigma_3\} \sqsubseteq g_j(v')$, hence $\{\sigma_2\} = \{\sigma_3\} * \{\sigma''_1\} \sqsubseteq g_j(v') * \{\sigma''_1\}$.

2. S is $\text{join}(v)$. We consider two cases: v is dead and v is alive. When v is dead, assume that $\sigma_1 = \sigma_3 * \sigma_4$, $\sigma_3 = (s, h, t[(\llbracket v \rrbracket_s, \text{THRD}(D, \text{emp}))])$, $\sigma_3 \in g_j(\text{proc}(v))$, $pc' = pc$, and $\sigma_4 = \bigotimes_{\{k \mid pc(k) \downarrow\}} g_k(G_{pr}(pc(k)))$. Since $\text{Post}_k^\gamma(\text{join}(v), \sigma_3) = \{\sigma_3\} \sqsubseteq g_j(v')$, Eq. 3 trivially holds. We now consider the case where v is alive. Assume that $\sigma_1 = \sigma_3 * \sigma_4 * \sigma_5$, $\sigma_3 \in g_j(\text{proc}(v))$, $\sigma_3 = (s, h, t[(\llbracket v \rrbracket_s, \text{THRD}(F, \Phi))])$, $t' = t[(\llbracket v \rrbracket_s, \text{THRD}(D, \text{emp}))]$, $\sigma_4 = \bigotimes_{\{k \mid pc(k) \downarrow\}} g_k(G_{pr}(pc(k)))$, $pc = pc'[(\llbracket v \rrbracket_s, \text{end})]$, $\{\sigma_5\} \sqsubseteq \llbracket \Phi \rrbracket$, and $\sigma_2 = (s, h, t') * \sigma_4 * \sigma_5$; otherwise, either the left-hand side of Eq. 3 is \top or the right-hand side of Eq. 3 is \top . Now, we have to prove that $\sigma_2 \sqsubseteq g_j(v') * \{\sigma_4\}$. By definition of $\text{Post}()$ for $\text{join}(v)$, we get $\{(s, h, t')\} * \{\sigma_5\} \sqsubseteq g_j(v')$. Hence, $\{\sigma_2\} = \{(s, h, t')\} * \{\sigma_5\} * \{\sigma_4\} \sqsubseteq g_j(v') * \sigma_4$.

3. S is $f^i(v_1, \dots, v_n)$. $f^i(v_1, \dots, v_n)$ can be modeled as $v = \text{fork}(f^i, v_1, \dots, v_n)$; $\text{join}(v)$. Since the cases of fork, join, and sequential composition have been proven sound, the case of procedure call follows. \square

The soundness of our approach is now established by the following lemma.

Lemma 3 (Soundness of Threads as Resource). *Consider a program P with the main procedure $\text{main}() \{S_m\}$ and a set of procedures $f^i(v_i^*) \{S_i\}$ ($i = 1..n$) together with their corresponding pre/post-conditions $(\Phi_{pr}^i / \Phi_{po}^i)$ where Φ_{pr}^i is precise. Let $\Gamma = \{\{\Phi_{pr}^i\} f^i(v_i^*) \{\Phi_{po}^i\} \mid i = 1..n\}$. If our verifier derives a proof for P , i.e.*

$$\Gamma \vdash \{\Phi_{pr}^1\} S_1 \{\Phi_{po}^1\}, \dots, \Gamma \vdash \{\Phi_{pr}^n\} S_n \{\Phi_{po}^n\}, \Gamma \vdash \{\Phi_{pr}\} S_m \{\Phi_{po}\},$$

then for any interpretation j and the state $\sigma_0 = (s_0, h_0, \square)$ such that $\sigma_0 \in \llbracket \Phi_{pr} \rrbracket_j^m$ the program P is safe when running from σ_0 and if $pc_0, \sigma_0 \rightsquigarrow_P pc_1, (s, h, t)$, where $\forall k. pc_1(k) \downarrow \implies pc_1(k) = \text{end}$, then $(s, h, t) \in \llbracket \Phi_{po} \rrbracket_j^m * (\bigotimes_{\{k \mid t(k) = \text{THRD}(F, \Phi)\}} \llbracket \Phi \rrbracket)$.

Proof. Consider an interpretation j and the state $\sigma_0 = (s_0, h_0, \square)$ such that $\sigma_0 \in \llbracket \Phi_{pr} \rrbracket_j^m$. By Lemma 1, for all $k \in \text{TIDS}$, $\Gamma \models_k \{\Phi_{pr}^i\} S_i \{\Phi_{po}^i\}$ and $\Gamma \models_m \{\Phi_{pr}\} S_m \{\Phi_{po}\}$. By Definition 3, we have $\text{Post}_k^\gamma(S_i, \llbracket \Phi_{pr}^i \rrbracket) \sqsubseteq \llbracket \Phi_{po}^i \rrbracket$ and $\text{Post}_m^\gamma(S, \llbracket \Phi_{pr} \rrbracket) \sqsubseteq \llbracket \Phi_{po} \rrbracket$. Since Eq. 1 is fulfilled, we are now able to establish the safety of P . Suppose that $pc_0, \sigma_0, \rightsquigarrow_P pc_1, (s, h, t)$ and $\forall k. pc_1(k) \downarrow \implies pc_1(k) = \text{end}$, then from Eq. 2 we have $\{(s, h, t)\} \sqsubseteq \llbracket \Phi_{po} \rrbracket_j^m * (\bigotimes_{\{k \mid t(k) = \text{THRD}(F, \Phi)\}} \llbracket \Phi \rrbracket)$ \square

In addition to partial correctness, in a similar way to [12], we can prove that, when using our approach, provable programs are race-free, as stated in Lemma 4.

Definition 4 (Data Race). *When running from an initial state σ_0 , a program P has a data race if for some pc and a state σ_1 such that $pc_0, \sigma_0 \rightsquigarrow_P pc, \sigma_1$, there exists two atomic statements S_1 in thread k (i.e. $pc(k) = S_1; S'_1$) and S_2 in thread j (i.e. $pc(j) = S_2; S'_2$) ($k \neq j$) such that $(S_1, \sigma_1 \not\rightsquigarrow_k \top)$, $(S_2, \sigma_1 \not\rightsquigarrow_j \top)$, and $S_1 \bowtie_{\sigma_1} S_2$ (i.e., S_1 and S_2 both access the same memory location in state σ_1 and at least one of the accesses is a write).*

Lemma 4 (Data-race Freedom). *Consider a program P with the main procedure $\text{main}() \{S_m\}$ and a set of procedures $f^i(v_i^*) \{S_i\}$ ($i = 1..n$) together with their corresponding pre/post-conditions $(\Phi_{pr}^i / \Phi_{po}^i)$ where Φ_{pr}^i is precise. Let $\Gamma = \{\{\Phi_{pr}^i\} f^i(v_i^*) \{\Phi_{po}^i\} \mid i = 1..n\}$. If our verifier derives a proof for P , i.e.*

$$\Gamma \vdash \{\Phi_{pr}^1\} S_1 \{\Phi_{po}^1\}, \dots, \Gamma \vdash \{\Phi_{pr}^n\} S_n \{\Phi_{po}^n\}, \Gamma \vdash \{\Phi_{pr}\} S_m \{\Phi_{po}\},$$

then the program P has no data race when running from an initial state $\sigma_0 = (s_0, h_0, \square)$ such that $\sigma_0 \in \llbracket \Phi_{pr} \rrbracket_j^m$ for any interpretation j .

Proof. It directly follows from Theorem 15 of [12]. Intuitively, one can prove that, for a provable program, given state σ_1 , two interfering atomic statements S_1 and S_2 , $(S_1, \sigma_1 \not\rightsquigarrow_k \top)$ and $(S_2, \sigma_1 \not\rightsquigarrow_j \top)$, there do not exist σ_2 and σ_3 such that $\sigma_1 = \sigma_2 * \sigma_3 * \sigma_4$, $(S_1, \sigma_2 \not\rightsquigarrow_k \top)$, and $(S_2, \sigma_3 \not\rightsquigarrow_j \top)$. \square

5. Applications

5.1 Verifying the Multi-join Pattern

A program with multi-join pattern allows a thread (jonee) to be shared and joined in multiple threads (joiners). During the execution of the program, the joiners wait for the jonee to finish its execution. If joiners wait for an already-completed jonee, they proceed immediately without waiting. By joining with the jonee, the joiners expect to receive certain resource transferred from the jonee. The motivating program in Fig. 1 is an example of such a multi-join pattern. As we have shown in previous sections, our approach handles the multi-join pattern naturally. Our approach allows the ownership of the jonee to be split, shared, and joined by multiple joiners, where each joiner obtains their corresponding part of the jonee's resource upon join.

```

1 data node { int val; node next; }
2 data list { node head; }
3 data count { int val; }
4 self ↦ ll(n) def self=null ∧ n=0
5 √ ∃q · self ↦ node(.,q) * q ↦ ll(n-1)
6 inv n ≥ 0;
7
8 void countList(list l)
9 requires l ↦ list(h) * h ↦ ll(n) ∧ n ≥ 0
10 ensures l ↦ list(h) * h ↦ ll(n) ∧ res=n;
11 { ... }
12
13 list createList(int n)
14 requires n ≥ 0
15 ensures res ↦ list(h) * h ↦ ll(n);
16 { ... }
17
18 list destroyList(list l)
19 requires l ↦ list(h) * h ↦ ll(n)
20 ensures emp;
21 { ... }
22
23 void mapper(list l, list o, list e)
24 requires l ↦ list(h) * h ↦ ll(n) * o ↦ list(null) * e ↦ list(null)
25 ensures o ↦ list(oh) * oh ↦ ll(n1) * e ↦ list(eh) *
26     eh ↦ ll(n2) ∧ n = n1 + n2;
27 { ... }
28
29 void reducer(thrd m, list l, count c)
30 requires m ↦ thrd(l ↦ list(h) * h ↦ ll(n) ∧ n ≥ 0) * c ↦ count(.)
31 ensures l ↦ list(h) * h ↦ ll(n) * c ↦ count(n) ∧ dead(m);
32 { join(m); /*multi-joined by the two reducers*/
33   c.val = countList(l); }
34
35 void main()
36 requires emp ensures emp;
37 { int n = 10000; list l = createList(n);
38   list ol = new list(null); list el = new list(null);
39   count c1 = new count(0); count c2 = new count(0);
40   /*fork mapper/reducer threads*/
41   thrd m = fork(mapper,l,ol,el);
42   thrd r1 = fork(reducer,m,ol,c1);
43   thrd r2 = fork(reducer,m,el,c2);
44   /*wait for them to finish*/
45   join(r1);
46   join(r2);
47   assert(c1.val + c2.val = n); /*valid*/
48   destroyList(ol); destroyList(el);
49   destroy(c1); destroy(c2);
50 }

```

Figure 8. Map/Reduce Pattern using Multi-join

We now illustrate another example of multi-join concurrency pattern in Fig. 8, based on the map/reduce paradigm. In this program, the main thread concurrently forks three threads: a mapper m to produce two lists, and two reducers $r1$ and $r2$ to process a list each. Both the reducers each take m as a parameter and joins it at an appropriate place to recover their respective lists from m . The main thread subsequently joins up the two reducers before completing its execution. This multi-join program is challenging to verify because (1) fork and join operations on the mapper m are non-lexically scoped (i.e. m is forked in main but joined in threads $r1$ and $r2$), and (2) part of the computed resources from m is made available to $r1$, while another part is made available to $r2$. In this program, the ownerships of two lists produced by the mapper must be flexibly transferred across thread boundaries, via fork/join calls. The key points to handle this program are (1) considering the executing thread of m as resource, and (2) allowing it to be split and transferred

between main, $r1$ and $r2$ via fork/join calls. Using our approach, the program can be verified as both data-race-free and functionally correct.

5.2 Inductive Predicates and Threads as Resource

Modeling threads as resource open opportunities for applying current advances in separation logic, which were originally designed for heap objects, to threads. In this section, we describe how “threads as resource” together with inductive predicates [11, 20] can be used to naturally capture a programming idiom, called threadpool, where threads are stored in data structures.

An example program is presented in Fig. 9. The program receives an input n , and then invokes `forkThreads` to create n concurrent threads executing the procedure `thread`. For simplicity, we assume each thread will have a read permission of the cell x in the pre-condition and will return the read permission in the post-condition. The program will wait for all threads to finish their execution by invoking `joinThreads`. At the end, as threads already

```

1 data cell { int val; }
2 data item { thrd t; item next; }
3
4 int input() requires emp ensures res > 0;
5
6 void thread(cell x, int M)
7   requires x  $\xrightarrow{1/M}$  cell(.) ∧ M > 0 ensures x  $\xrightarrow{1/M}$  cell(.);
8
9 item forkHelper(cell x, int n, int M)
10  case { n = 0 → requires emp ensures emp ∧ res = null;
11        n > 0 → requires x  $\xrightarrow{n/M}$  cell(.) ∧ M ≥ n
12              ensures res ↦ pool(x, n, M); }
13 { if (n==0){ return null; } else {
14   thrd t = fork(thread,x,M);
15   item p = forkHelper(x,n-1,M);
16   item i = new item(t,p);
17   return i; } }
18
19 item forkThreads(cell x, int n)
20  requires x ↦ cell(.) ∧ n > 0
21  ensures res ↦ pool(x, n, n);
22 { return forkHelper(x,n,n); }
23
24 void joinHelper(item tp, cell x, int n, int M)
25  requires tp ↦ pool(x, n, M) ∧ M ≥ n ∧ n >= 0
26  ensures x  $\xrightarrow{n/M}$  cell(.) ∧ n > 0 ∨ emp ∧ n = 0;
27 { if (tp==null){ return; } else {
28   joinHelper(tp.next,x,n-1,M);
29   join(tp.t); destroy(tp); } }
30
31 void joinThreads(item tp, cell x, int n)
32  requires tp ↦ pool(x, n, n) ∧ n > 0;
33  ensures x ↦ cell(.);
34 { return joinHelper(tp,x,n,n); }
35
36 void main() requires emp ensures emp;
37 { cell x = new cell(1); int n = input();
38   item tp = forkThreads(x,n);
39   joinThreads(tp,x,n);
40   destroy(x); }

```

Figure 9. Verification of a Program with Dynamic Threads using Inductive Predicates

finished, it is safe to destroy the cell x . In this program, each item in the threadpool is a data structure of type `item`. Each `item` will store a thread in its field `t` and a pointer `next` to the next item in the pool. The `forkThreads` returns the first item in the pool, while the `joinThreads` receives the item and joins with all threads in the pool. In the program’s specifications, “*res*” is used to denote the returned result of a procedure and “`_`” represents an unknown value.

The key idea to verify this program is to use an inductively defined predicate, called `pool` to abstract the threadpool. As threads are modeled as resource, they can be naturally captured inside the predicate in the same way as other heap resource, as follows:

$$\begin{aligned} self \mapsto pool(x, n, M) &\stackrel{\text{def}}{=} self = null \wedge n = 0 \wedge M > 0 \\ &\vee \exists t, p. self \mapsto item(t, p) * t \mapsto \text{thrd}(x \xrightarrow{1/M} cell(-)) * \\ &\quad p \mapsto pool(x, n-1, M) \\ \text{inv } n &\geq 0 \wedge M > 0; \end{aligned}$$

The above predicate definition asserts that a `pool` can be empty (the base case $self = null$) or consists of a head item (specified by $self \mapsto item(t, p)$), a thread node ($t \mapsto \text{thrd}(x \xrightarrow{1/M} cell(-))$) and a tail data structure which is also a `pool`. The invariant $n \geq 0 \wedge M > 0$ must hold for all instances of the predicate. Using the above definition and case analysis [11], the program can be verified as functionally correct and data-race-free. Although we use linked lists here, our approach easily adapts to other data structures, such as arrays.

5.3 Thread Liveness and Resource Leakage

Using our approach, threads’ liveness can be precisely tracked. For example, we could modify the program in Fig. 9 to additionally keep track of already-completed (or dead) threads. In the procedure `joinHelper`, after a thread is joined, instead of destroying the corresponding item (line 29), we could capture all items and their dead threads in a `deadpool`³, inductively defined as follows:

$$\begin{aligned} self \mapsto deadpool(n) &\stackrel{\text{def}}{=} self = null \wedge n = 0 \\ &\vee \exists t, p. self \mapsto item(t, p) * p \mapsto deadpool(n-1) \wedge \text{dead}(t) \\ \text{inv } n &\geq 0; \end{aligned}$$

Our approach is also able to keep track of threads’ resource in a precise manner. This is important for avoiding leakages of thread resource. As an example, consider the use of a resource split, prior to a join operation.

```
// {t ↦ thrd(Φ1 * Φ2)}
// {t ↦ thrd(Φ1) * t ↦ thrd(Φ2)}
join(t);
// {Φ1 * t ↦ thrd(Φ2) ∧ dead(t)}
// {Φ1 * Φ2 ∧ dead(t)} /*R-THRD2 applied*/
```

This split causes the join operation to release only resource Φ_1 , whilst Φ_2 remains trapped as resource inside a thread node. This results in a resource leakage if the scenario is not properly considered. However, our verification system handles such scenarios by releasing the trapped resource using the R-THRD2 rule in Fig. 5, thus ours avoids the leakages of thread resource.

6. Implementation and Experimental Results

We demonstrate the feasibility of our “threads as resource” approach by implementing it on top of PARAHIP [18], a current state-of-the-art verifier for fork-join concurrency and mutex locks that is able to verify functional correctness and deadlock-freedom. PARAHIP models threads as separate **and**-conjunctions, and its threads are not allowed to be split and shared. In contrast, besides verifying functional correctness, data-race freedom, deadlock

³ We refer interested readers to `deadpool` program in our project webpage for more details.

freedom, and non-lexically-scoped fork/join, our implementation (called THREADHIP) is also capable of verifying first-class threads and multi-join.

The expressiveness of “threads as resource” is beyond that of other verification systems for fork/join programs. However, as there is a lack of commonly accepted benchmarks in the literature, we cannot easily compare THREADHIP with other systems. In order to give readers an idea of the applicability of our approach, we did an experimental comparison between PARAHIP and THREADHIP. Experimental programs consist of 16 deadlock/deadlock-freedom programs from PARAHIP’s benchmark and other intricate programs inspired by the literature.⁴ Besides the theoretical contributions, the empirical questions we investigate are (1) whether the “threads as resource” approach is capable of verifying more challenging programs, and (2) how THREADHIP performs, compared with PARAHIP. All experiments were conducted on a machine with Ubuntu 14.04, 3.20GHz Intel Core i7-960 processor, and 12GB memory.

The experimental results are presented in Table 1. THREADHIP is able to verify all programs of PARAHIP’s benchmark and other programs found in the literature. For these programs, PARAHIP and THREADHIP showed comparable verification times (e.g. for PARAHIP’s benchmark, the difference is +2.0%). Nonetheless, THREADHIP is more expressive than PARAHIP since THREADHIP is also capable of verifying more complex programs that manipulate the multi-join pattern and/or require expressive treatment of threads’ resource such as `mapreduce`, `threadpool`, and `multicast`. We believe that existing verifiers for verifying concurrent programs can easily integrate our “threads as resource” approach into their systems (as we did for PARAHIP) and benefit from its greater expressiveness with negligible performance difference.

7. Related Work

This section discusses related works on reasoning about shared-memory concurrent programs. Our approach currently supports only partial correctness. Proving (non-)termination is an orthogonal issue and could be separately extended.

Traditional works on concurrency verification such as Owicki-Gries [22] and Rely/Guarantee reasoning [16] often focused on simple parallel composition, rather than fork/join. Fork/join concurrency is more general than the parallel composition for two main reasons. First, fork/join supports dynamic thread creation and termination. Second, while threads in a parallel composition are lexically scoped, threads in fork/join programs can be non-lexically scoped. Therefore, fork/join programs are more challenging for verification. Even recent approaches such as CSL [21], RGSep [25], LRG [9], and Views [6], omit fork/join concurrency from their languages. Our “threads as resource” is complementary to the above approaches and could be integrated into them.

There also exist approaches that can handle fork/join operations. Both Hobor [14] and Feng and Shao [10] support fork and omit join with the claim that thread join can be implemented using synchronization. However, without join, the former allows threads to leak resource upon termination while the latter requires global specifications of inter-thread interference. Approaches that can handle both fork and join can be grouped as modeling threads as tokens [12, 15, 19] and modeling threads as **and**-conjunctions [17, 18]. Though syntactically different, they are semantically similar in that the tokens and the **and**-conjunctions are used to represent the post-states of forked threads. However, they offer limited support

⁴ THREADHIP and all experimental programs are available for both online use and download at <http://loris-7.ddns.comp.nus.edu.sg/~project/threadhip/>.

Program	Properties	Verification Time (s)		
		PARAHIP	THREADHIP	Diff
PARAHIP's benchmark	F/L/(N)	*	*	+2.0%
fibonacci [17]	F	0.076	0.077	+1.3%
parallel-mergesort [17]	F	1.326	1.236	-6.8%
oracle [14]	F/L	1.654	1.646	-0.5%
owicki-gries [15]	F/L	1.227	1.241	+1.1%
multi-join1	F/N/M	-	0.075	-
multi-join2	F/N/M	-	0.216	-
mapreduce	F/N/M	-	0.515	-
threadpool	F/N/P	-	0.199	-
deadpool	F/N/P	-	0.261	-
multicast [12]	F/L/N/P	-	1.057	-
no-deadlock-nonlexical2	F/L/N/M	-	0.122	-

Table 1. Experimental Results. The second column indicates properties of a program, i.e. whether it uses fork/join (F), locks (L), non-lexical fork/join (N), multi-join (M), and inductive predicates (P); verification times are average of the 10 runs (in seconds); the final column is computed as $\frac{\text{THREADHIP} - \text{PARAHIP}}{\text{PARAHIP}}$; (*) details of PARAHIP's benchmark are presented in Table 2, here the final column of PARAHIP's benchmark reports the average of its 16 programs.

Program	Properties	Verification Time (s)		
		PARAHIP	THREADHIP	Diff
deadlock1	F/L	0.085	0.085	0.0%
deadlock2	F/L	0.088	0.090	+2.3%
deadlock3	F/L	0.095	0.097	+2.1%
deadlock-nested-forkjoin	F/L	0.150	0.156	+4.0%
disj-deadlock	F/L	0.103	0.102	-1.0%
disj-no-deadlock1	F/L	0.131	0.132	+0.8%
disj-no-deadlock2	F/L	0.136	0.142	+4.4%
double-acquire	F/L	0.760	0.750	-1.3%
fork-join-as-send-recv	F/L	0.176	0.189	+7.4%
no-deadlock1	F/L	0.104	0.099	-4.8%
no-deadlock2	F/L	0.104	0.108	+3.8%
no-deadlock3	F/L	0.137	0.414	+2.9%
ordered-locking	F/L	0.189	0.195	+3.2%
unordered-locking	F/L	0.173	0.180	+4.0%
deadlock-nonlexical	F/L/N	0.098	0.097	+1.0%
no-deadlock-nonlexical	F/L/N	0.116	0.119	+2.6%
Average	-	-	-	+2.0%

Table 2. Experimental Results on PARAHIP's benchmark. 14 out of 16 programs in PARAHIP's benchmark have properties F/L; two programs have properties F/L/N.

for first-class threads: tokens and **and**-conjuncts are not allowed to be split and shared among concurrent threads. As such, they are not expressive enough to verify programs with more intricate fork/join behaviors such as the multi-join pattern where threads are shared and joined in multiple threads. Existing works could encode the multi-join pattern by using synchronization primitives such as channels or locks. However, the encoding requires additional support for the primitives and could complicate reasoning (i.e. we have to reason about channels or locks instead of just focusing on threads). Our approach is more elegant and natural. Inspired by the key notation of resource in separation logic [1, 21], we propose to model threads as resource, thus allow ownerships of threads to be flexibly split and distributed among multiple joiners. This enables verification of the multi-join pattern. In addition, unlike ours, none of related works that we are aware of support explicit reasoning about thread liveness. To the best of our knowledge, only Haack and Hurlin [13] can reason about some multi-join scenarios. In their approach, a thread token can be associated with a fraction and this allows multiple joiners to join with the same

joiner in order to read-share the joiner's resource. However, this simple multiplicative treatment of thread tokens is not expressive enough as it is unable to verify programs that require the joiners to write-share the resource of the joiner (e.g. the program in Fig 1). In order to cater to a more flexible treatment of joiners and their resource, modeling threads as resource is essential.

The concept of higher-orderness, where a heap node carries other heap nodes, has been proposed in the literature. Typical examples are lock nodes that carry the locks' resource invariants [12, 14, 15]. However, thread nodes and lock nodes are fundamentally different. First, locks and threads are used differently. While the resource inside a lock node can be repeatedly acquired/released, the resource inside a thread node can be obtained only once when the thread is joined. Second, unlike the resource in a lock node which is exclusively acquired by a single thread, the resource in a thread node can be split and shared among multiple joiners. Our "threads as resource" also shares some similarities with Concurrent Abstract Predicates (CAP) [7, 8, 24]. The basic idea behind CAP is to provide an abstraction of possible interferences from

concurrently running threads, by partitioning the state into regions with protocols governing how the state in each region is allowed to evolve. Ours is simpler; a thread node is an abstraction of a thread and can be considered as a placeholder for the resource that it carries. The placeholder is created when forking, split and shared across procedure boundaries, and destroyed when joining. Hence, our approach enables reasoning about first-class threads without resorting to any additional protocols.

8. Conclusion

We proposed to model first-class threads as resource to enable expressive treatment of threads' ownerships. Our approach allows resources of threads to be flexibly split, combined, and transferred across procedure boundaries. This enables verification of multi-join pattern, where multiple joiners can share and join the same joiner in order to manipulate (read or write) the resource of the joiner after join. In addition, we demonstrated how threads as resource is combined with inductive predicates to capture the thread-pool idiom. Using a special `dead` predicate, we showed that thread liveness can be precisely tracked. We have implemented our approach in a tool, called `THREADHIP`, to verify partial correctness and data-race freedom of concurrent programs with non-lexically-scoped `fork/join`, first-class threads, and multi-join. Our experiment showed that `THREADHIP` is more expressive than `PARAHIP` while achieving comparable verification time.

As future work, we plan to exploit current advances in resource synthesis (such as [3, 4]) to synthesize "threads as resource" specification automatically.

Acknowledgments

We are grateful to the anonymous reviewers and our colleagues for their insightful comments and feedbacks.

References

- [1] R. Bornat, C. Calcagno, and H. Yang. Variables as Resource in Separation Logic. *ENTCS*, 155:247–276, 2006.
- [2] J. Boyland. Checking Interference with Fractional Permissions. In *SAS*, 2003.
- [3] C. Calcagno, D. Distefano, P. W. O'Hearn, and H. Yang. Compositional Shape Analysis by Means of Bi-Abduction. *JACM*, 58(6):26, 2011.
- [4] C. Calcagno, D. Distefano, and V. Vafeiadis. Bi-abductive Resource Invariant Synthesis. In *APLAS*, pages 259–274, 2009.
- [5] W.N. Chin, C. David, H.H. Nguyen, and S. Qin. Multiple Pre/Post Specifications for Heap-Manipulating Methods. In *HASE*, pages 357–364, 2007.
- [6] T. Dinsdale-Young, L. Birkedal, P. Gardner, M. Parkinson, and H. Yang. Views: Compositional Reasoning for Concurrent programs. In *POPL*, 2013.
- [7] T. Dinsdale-Young, M. Dodds, P. Gardner, M. J. Parkinson, and V. Vafeiadis. Concurrent abstract predicates. In *ECOOP*, pages 504–528, 2010.
- [8] M. Dodds, S. Jagannathan, and M. J. Parkinson. Modular reasoning for deterministic parallelism. In *POPL*, pages 259–270, 2011.
- [9] X. Feng. Local Rely-Guarantee Reasoning. In *POPL*, pages 315–327, 2009.
- [10] X. Feng and Z. Shao. Modular Verification of Concurrent Assembly Code with Dynamic Thread Creation and Termination. In *ICFP*, pages 254–267, 2005.
- [11] C. Gherghina, C. David, S. Qin, and W.N. Chin. Structured Specifications for Better Verification of Heap-Manipulating Programs. In *FM*, pages 386–401, 2011.
- [12] A. Gotsman, J. Berdine, B. Cook, N. Rinetzky, and M. Sagiv. Local Reasoning for Storable Locks and Threads. In *APLAS*, pages 19–37, 2007.
- [13] C. Haack and C. Hurlin. Separation Logic Contracts for a Java-Like Language with Fork/Join. In *AMAST*, pages 199–215, 2008.
- [14] A. Hobor. *Oracle Semantics*. PhD thesis, Princeton University, 2008.
- [15] B. Jacobs and F. Piessens. Expressive Modular Fine-grained Concurrency Specification. In *POPL*, pages 271–282, New York, NY, USA, 2011.
- [16] Cliff B. Jones. Specification and Design of (Parallel) Programs. In *IFIP Congress*, pages 321–332, 1983.
- [17] D.K. Le, W.N. Chin, and Y.M. Teo. Variable Permissions for Concurrency Verification. In *ICFEM*, pages 5–21, 2012.
- [18] D.K. Le, W.N. Chin, and Y.M. Teo. An Expressive Framework for Verifying Deadlock Freedom. In *ATVA*, pages 287–302, 2013.
- [19] K. R. M. Leino, P. Müller, and J. Smans. Deadlock-Free Channels and Locks. In *ESOP*, pages 407–426, 2010.
- [20] H.H. Nguyen, C. David, S. Qin, and W.N. Chin. Automated Verification of Shape and Size Properties via Separation Logic. In *VMCAI*, Nice, France, 2007.
- [21] P. W. O'Hearn. Resources, Concurrency and Local Reasoning. In *CONCUR*, 2004.
- [22] S. Owicki and D. Gries. Verifying Properties of Parallel Programs: an Axiomatic Approach. *CACM*, pages 279–285, 1976.
- [23] J. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *LICS*, Copenhagen, Denmark, July 2002.
- [24] K. Svendsen, L. Birkedal, and M. J. Parkinson. Modular reasoning about separation of concurrent data structures. In *ESOP*, pages 169–188, 2013.
- [25] V. Vafeiadis and M. Parkinson. A Marriage of Rely/Guarantee and Separation Logic. In *CONCUR*, pages 256–271. Springer, 2007.