Assuring Structural Parallel Programs based on Scoped Permissions

Yang Zhao, Ligong Yu, Gongxuan Zhang
School of Computer Science, Nanjing University of Science & Technology, Nanjing, China
Email: \{yangzhao, yuligong, gongxuan\}@mail.njust.edu.cn

Jia Bei
Software Institute, Nanjing University, Nanjing, China
Email: beijia@software.nju.edu.cn

Abstract—This paper proposes a “scoped permission” system for a simple object-oriented language with shared-memory and structural parallelism. The permission is abstracted as a linear value associated with some piece of state in a program and it is normally adopted in program analysis and verification. In this paper, the permission nesting is utilized to model the protection mechanism associated with field instances, while the partial order among different locks is specified when parallel executions start. By generating and eliminating shared facts, the order in our system is designed to be scoped and mutable. We show the operational semantics as well as some permission rules, and demonstrate how to interpret program annotations into permission representations.

Index Terms—scoped permission, structural parallelism, nesting, program annotations

I. INTRODUCTION

It is well known that multithreaded programming greatly increases the performance of programs, but it also provides some potential intermittent concurrency errors that are hard to be detected using traditional program analysis, because the effect of interactions among parallel threads is usually undeterministic.

There are two common errors that often happen in parallel programs. Data race happens when two or more parallel threads sharing state try to access the same location simultaneously and at least one of them is a write operation. As a result, unexpected behaviors or serious runtime exceptions may be exhibited. Assuming two threads are trying to access an object through the same pointer, one goes to dereference some field of the object, while the other attempts to deallocate that object at the same time, if the deallocation happens to win the contention and be executed first, then the dangling pointer error appears and a null pointer exception may be threwed out when the dereference start to take action.

Deadlock, however, occurs when two or more threads are waiting on a condition that cannot be satisfied. Deadlock most often occurs when two (or more) threads are each waiting for the other(s) to do something and the whole program gets stuck. If multiple concurrent threads are holding some resources but waiting for more that are currently held by some others, then it is possible that none of the parallel threads can make progress and the whole program gets stuck.

Numerous static analysis techniques are designed to ensure parallel programs are free of data races or deadlocks [1]–[4], and some of them are based upon the language’s type system with annotations [5]–[8].

Proposed by Boyland et al., the permission system originates from comprehending program annotations into a semantic foundation [9]. Within the permission system, every expression will be checked to determine whether it is permitted to be executed under certain permissions that are granted. Different operations in a program require different permissions, it has some advantages over previous type systems. For instance, it is capable of distinguishing reads from writes. In consequence, some safe interference patterns, like parallel reads without lock, could be assured. Furthermore, various nesting facts among permissions could naturally be used to model lock protection relations in parallel programs.

In the previous work, a fractional permission system is designed for a non-synchronizing language [10] and adoptions (nestings) are used to connect effects and uniqueness [9]. In this paper, we further extend the previous work by proposing a scoped and shared permission, such that the lock protection is modeled by permission nesting as usual, while the partial order among locks is built by shared facts which could be mutable. To demonstrate this idea, we define a toy language with structural parallelism as well as synchronization.

In the following text, Section II defines a simple object-oriented language as well as its operational semantics. In Section III, we introduce the scoped permission system
with some selected permission rules, based upon which we show how to interpret program annotations into permission representations and give several examples. Section IV shows the consistency between permissions and dynamic runtime states, based on which the soundness property is established. Section V gives some related work and we conclude in Section VI. The Appendix contains all the rules mentioned in this paper.

II. A SIMPLE LANGUAGE

This section describes a simple object-oriented language with parameterized classes and field/method annotations. We adopt structural parallelism, but omit object inheritance for simplicity.

A. Syntax

The language syntax is given in Figure 1. A shared-memory program consists of several class definitions and each class may be parameterized by zero or more formal guard objects that could occur in the annotation clauses. A “guarded_by lock” clause for field f indicates that the lock is designed to protect any access operation for f. “reads” and “writes” are self-explanatory method effects. A “requires lock” states the lock has already been held at the method entry, while “uses lock” says that the method body may acquire lock by itself. A partial order between two locks is enforced inside of a method body once a “lock1 < lock2” is specified.

Expressions include pure allocation 1, variable read, field read, field write, sequential composition, local declaration, method invocation, synchronization and structural parallel composition. Figure 2 and 3 provide some code examples using this syntax.

B. Operational Semantics

The operational semantics is defined in terms of a small-step evaluation:

\[
\mu; (e) \rightarrow (\mu'; (e')_L)
\]

Given a memory \(\mu\), an expression \(e\) dynamically nested most recently in a synchronized block holding \(L\) ($\emptyset$ if

1

A regular object allocation is normally implemented by a pure allocation followed by a constructor call.

class Account<g> {
  /**@guarded_by g*/
  Int balance;

  /**@read m.All, requires this*/
  void deposit(Int m) {
    balance += m
  }
}

......

Figure 2. Account class.

class Client {
  Account<this> checking, saving;

  /**@read m.All, checking, saving; uses (checking, saving), saving<checking*/
  void checking2saving(Int m) {
    synch saving.deposit(m);
    synch checking.withdraw(m);
  }
}

......

Figure 3. Client class.

none) can be one-step evaluated to \(e'\) with the memory being changed into \(\mu'\) accordingly, where memory \(\mu\) is defined as a mapping from locations (pairs of object address and field name) to other addresses:

\[
\mu \in \text{Memory} = (O \times F) \rightarrow O
\]

We choose several evaluation rules in Figure 8 and give explanation in the following text, while other self-explanatory rules are listed in the Appendix.

E-FORK

\[
(\mu; (\text{parbegin} (e1||e2) \text{parend})_L) \rightarrow (\mu; ((e1)_L||((e2)_L)_L))
\]

E-PARALLEL

\[
(\mu; (e1)_L) \rightarrow (\mu'; (e1')_L)
\]

\[
e1' \rightarrow e2' = (i == 1)?(e1', e2') : (e1, e2)
\]

\[
(\mu; ((e1)_L)||((e2)_L)_L) \rightarrow (\mu; ((e1')_L)||((e2')_L)_L)
\]

E-JOIN

\[
(\mu; ((o1)_L)||((o2)_L)_L) \rightarrow (\mu; (o1)_L)
\]

E-ACQ

\[
\mu(o1, m) = \emptyset \quad \mu' = \mu|(o1, \_m) \rightarrow o1]
\]

\[
(\mu; (\text{synch} o1 \text{ do e2})_L) \rightarrow (\mu'; (\text{hold} o1 \text{ do } (e2)_{o1})_L)
\]

E-REL

\[
\mu(o1, m) = o1 \quad \mu' = \mu|(o1, \_m) \rightarrow \emptyset]
\]

\[
(\mu; (\text{hold} o1 \text{ do } (e2)_{o1})_L) \rightarrow (\mu'; (o2)_L)
\]

Figure 4. Selected evaluation rules

The E-FORK rule indicates that two sub-expressions \(e1\) and \(e2\) are going to be executed in parallel, both of which have the same surrounding lock as the whole parallel composition. Evaluating two expressions in parallel is nondeterministic: either side may be evaluated one step
further, based on which we use E-PARALLEL, such that i could be 1 or 2. According to E-JOIN, once both sides are done, two parallel expressions are then eliminated with retaining the result from one side (we pick the left).

Every lock object is designed to include an implicit monitor zn indicating whether or not the object has already been locked. If zn is null, its object is in an unlocked state and is free to be acquired (by setting the zn field to itself). E-ACQ shows the situation of a lock acquisition 2, while E-REL applies the situation when the lock is released and the context thread exits from the synchronized block.

III. SCOPED PERMISSIONS

Permission is used as a semantic foundation for different annotations and hence is capable of verifying many program properties [9]–[11].

A. Definition

A permission is an abstract token associated with some piece of state in a program and it is designed to permit certain operations. Every piece of state is associated with exactly one permission which is used as the right to access the associated state. Assuming the execution needs to access a field f of object o which is currently pointing to another o′, then it is required to be granted a permission written as of: reff(o′).

In order to distinguish reads from writes, we associate fractions ξ as well, such that a write permission is an explicit whole permission 1of: reff(o′), while a read permission is just some partial permission ξof: reff(o′) where ξ is syntactically guaranteed to be positive.

<table>
<thead>
<tr>
<th>object reference</th>
<th>ρ := o</th>
<th>r</th>
</tr>
</thead>
<tbody>
<tr>
<td>key</td>
<td>k := ρf</td>
<td></td>
</tr>
<tr>
<td>fraction</td>
<td>ξ := 1</td>
<td>1/2</td>
</tr>
<tr>
<td>dimension</td>
<td>d := 0</td>
<td>d ± 1</td>
</tr>
<tr>
<td>type</td>
<td>τ := reff(ρ)</td>
<td></td>
</tr>
<tr>
<td>fact</td>
<td>Γ := ρ ∈ C</td>
<td>π-ρ</td>
</tr>
<tr>
<td></td>
<td>d &lt; d</td>
<td>∃e</td>
</tr>
<tr>
<td>permissions</td>
<td>Π := ∅</td>
<td>ξk: τ</td>
</tr>
<tr>
<td></td>
<td>Γ ⊢ Π</td>
<td>∃r.(Π)</td>
</tr>
<tr>
<td></td>
<td>Ω(ρ&lt;ρ)</td>
<td></td>
</tr>
</tbody>
</table>

Figure 5. Permission syntax.

The formal permission syntax is given in Figure 5. Here, o and r are used to range literal addresses and variables respectively. A permission could be empty, fractional, a fact, a linear implication, a combination 3, conditional, existential or shared.

A ξρf: τ indicates some kind of right (depending on ξ) to access field instance ρf. The fraction ξ could be 1, 1/2, a variable z or a product. τ is a permission type which is given a regular pointer reff(ρ).

A conditional permission Γ ⇒ Π presents a possible II depending on the truth value of Γ.

A fact Γ uses a simple logic that can be represented by boolean formulae over type assertions (ρ ∈ C), reference equalities (ρ=ρ), nesting (Π ⊆ k), object type predicates (C(ρ′)), dimension comparing (d<d) as well as some standard boolean logic. In particular, the nesting expresses a relation that some permission Π is nested into location k (written Π ⊆ k), which implies that the nested permission is not available unless the nester one is granted.

The linear implication Π1 ⇒ Π2 indicates that one has the rights of the consequent Π2, except for the ones of the antecedent Π1. Given a nesting fact Π ⊆ k, a nested II can be carved out of its nester 1k : τ. The carving process is then:

Π ⊆ k + Π1 ⊆ Π2 ⇒ Π ⊆ k + Π ⊆ Π2 → 1k : τ

A shared permission Ω(ρ′) has two forms:

• a wrapped permission Ω(ρ) that will be transformed into 1ρ. Prot : τ ⊆ 0 (called unwrapped permission) after acquiring lock ρ, but be resumed once the lock is released as described later in rule SYNCH;

• a shared fact Ω(ρ′) that enforces an order between two locks, such that it is not allowed to acquire the ρ′ when ρ is currently held by the context thread.

The dimension d is used to remember the duplication: a shared permission Ω(ρ′) increases its dimension by one when it is duplicated at the beginning of two parallel executions, but decreases by one when two parallel executions join afterwards. Any shared permission disappears once its dimension becomes zero. The difference between a normal fact and a shared fact is that the former is able to be duplicated and eliminated arbitrarily, while the latter should maintain its dimension explicitly, with which we are capable of tracking the thread information and implementing the thread-scope order among locks.

B. Permission Checking

Given an environment E and an expression e nested most recently in a synchronized block holding pL (dL is the dimension of Ω(ρL) when it changes to ρL. Prot : τ|0), if e can be permission checked using E, then it has a permission type τ with the environment being changed to E′, written as a judgment:

E ⊢ dL ρL e ⊥ τ | E′

An environment is composed by two parts: a type context Δ which is a set of object variables r, fraction variables z and dimension variables p; and a granted permissions II. For a well-formed environment E = (Δ; II), we require that all free variables used in II are in Δ (FV(II) ⊆ Δ).

1) Permission Checking Rules: Regular permission rules, such as READ, WRITE, SEQ, CALL and so on are given in the Appendix. We only explain the rules which are related to the parallel execution and synchronization in Figure 6.

For rule NEW, we pick a fresh variable r to represent the new created object and initialize all its fields to be
null. Moreover, we combine a type fact \( r \in C \) as well as the wrapped permission \( \Omega^1(r) \) indicating it is thread-local and free to be locked afterwards.

In SYNCH, there are several issues: (1) The lock expression is permission-checked to make sure it has a type \( \text{ref}(\rho_2) \) for some \( \rho_1 \). (2) The wrapped permission \( \Omega^{d2}(\rho_1) \) should be present right before entering the synchronized block. It is also required to compare the latest holding lock \( \rho_1 \) with \( \rho_2 \) to determine whether they follow a descending order using a shared fact, but wait, what if one of the two locks is thread-local? The conditional permission \((1 < d_2 \land 1 < d_1) \Rightarrow \Omega^{d2}(\rho_1 < \rho_2)\) shows that only if neither locks is thread-local then the shared fact appears. (3) The wrapped permission \( \Omega^{d2}(\rho_1) \) is transformed into an unwrapped one \( \rho_1.\text{Prot} : T_0 \) inside of the synchronized block. (4) Finally, the unwrapped permission goes back to the wrapped one when exiting the synchronized block. Re-entering a synchronization with the same lock (acquiring the same lock multiple times) is not possible since we won’t get the wrapped permission any more when holding the lock. The wrapped permission \( \Omega^{d2}(\rho_1) \) guarantees that none of the nested permissions in \( \rho_1.\text{Prot} \) is available without acquiring lock \( \rho_1 \) first.

The FORK shows that a parallel composition can be permission-checked based on an input \( \Pi \) only if the \( \Pi \) can be split into three parts \( \Pi_1, \Pi_2, \) and \( \Pi_3 \), such that \( \Pi_1 \) goes into a corresponding child thread while \( \Pi_2 \) remains shared. \( \Pi_3 \) will undergo a transformation \( \omega(\Pi_3) \) that is explained below, and new partial order may be generated for unshared objects. Results are then re-combined after the shared parts are transformed back by \( \omega^{-1} \).

<table>
<thead>
<tr>
<th>( \rightarrow )</th>
<th>( \omega )</th>
<th>( \omega^{-1} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \emptyset )</td>
<td>( \emptyset )</td>
<td>( \emptyset )</td>
</tr>
<tr>
<td>( r )</td>
<td>( r )</td>
<td>( r )</td>
</tr>
<tr>
<td>( \psi )</td>
<td>( \psi )</td>
<td>( \psi )</td>
</tr>
</tbody>
</table>
| \( \Omega^1(...) \) | \( \Omega^+(...) \) | \( \Omega^-(...) \) with \( 1 < d \)
| \( \Pi_1 \parallel \Pi_2 \) | \( \omega(\Pi_1) \parallel \omega(\Pi_2) \) | \( \omega^{-1}(\Pi_1) \parallel \omega^{-1}(\Pi_2) \)
| \( 1 \Rightarrow \Pi \) | \( 1 \Rightarrow \omega(\Pi) \) | \( 1 \Rightarrow \omega^{-1}(\Pi) \)
| \( \Pi_1 \parallel \Pi_2 \) | \( \omega(\Pi_1) \parallel \omega(\Pi_2) \) | \( \omega^{-1}(\Pi_1) \parallel \omega^{-1}(\Pi_2) \)

The \( \omega \) operation increases the dimension of sharing on each shared permission, but leaves others immutable, while \( \omega^{-1} \) is an opposite relation except that an unshared (\( d = 1 \)) wrapped permission represents a locally created object and is preserved, while a shared fact with one dimension is discarded:

\[
\omega^{-1}(\Omega^1(\rho)) = \Omega^1(\rho) \quad \omega^{-1}(\Omega^1(\rho < \rho')) = \emptyset
\]

Essentially, the type for \( e_1 \) is picked and we simply merge the output permissions from two sides.

The \( \text{Gen}_c(\Omega^1(\rho_1), \ldots, \Omega^1(\rho_k), \Pi_c) \) operation considers each unshared wrapped permission \( \Omega^1(\rho) \) in an arbitrary order (not a lock order!) and for each one generates lock orders of one of the following cases (non-deterministically):

- It places \( \rho \) lower than all previous ordered locks \( (\Omega^{d+1}(\rho) \text{ in } \Pi_c, \text{ or an earlierarily considered unshared lock}) \).
- It places \( \rho \) higher than all previous ordered locks.
- It places \( \rho \) between an existing order \( (\Omega^d(\rho' < \rho') \text{ in } \Pi_c) \).

Furthermore, we need to make sure no cycle in the (scoped) lock order is generated.

2) Field and Class Invariant: Permissions are granted to permit certain operations and they are provided in two cases: class invariants and method types.

A “guarded by lock” clause is modeled by nesting the whole permission of its field into the \( \text{Prot} \) field of lock which is an implicit location with an uninteresting type \( T_0 \), such as \( \text{ref} : \tau \prec \text{Prot} \) for some \( \tau \). Without this annotation, the whole permission of a field goes into \( r_{\text{this}, \text{All}} \), where the \( \text{All} \) is considered as a location collecting all permissions except for those that are protected by guard objects. Analogously, \( \text{All} \) field is given the type \( T_0 \) as well.

A class invariant is a conjunction of all included field invariants. It is given as \( C(r_{\text{this}}, r_{s}^g) \) where \( r_{\text{this}} \) represents the this object and \( r_{s}^g \) is a sequence of variables for the guards \( g^* \) occurring as class parameters. For instance, the Account in Figure 2 has a class invariant:

\[
\text{Account}(r_{\text{this}}, r_s) = (r_{\text{this}} \in \text{Account}) \land \Gamma_{k} \quad \text{where} \quad \Gamma_{k} = \exists \ r. (\text{ref}(\tau) \prec r_{\text{Prot}}) \Rightarrow (\text{Int}(r) \lor \text{ref}(\emptyset)) \prec r_{\text{Prot}}
\]

The first conjunct shows the static type for \( r_{\text{this}} \), while the second is a field invariant for the balance field. \( \Gamma_{k} \) states that the whole permission of the field access is nested in its protector \( r_{\text{Prot}} \). Moreover, if it is not null, then its pointed-to object must be an \( \text{Int} \) object with holding a class invariant and the whole permission of \( r_{\text{All}} \) will also be nested in \( r_{\text{Prot}} \). In other words, the guard object protects not only the field access but also the field object.

Here, the class invariant for Client is in below, such that both field permissions are nested into \( r_{\text{this}, \text{All}} \) because no “guarded by” is attached.

\[
\text{Client}(r_{\text{this}}) = (r_{\text{this}} \in \text{Client}) \land \Gamma_{s} \land \Gamma_{c} \quad \text{where} \quad \Gamma_{c} = \exists \ r. (\text{ref}(r) \prec r_{\text{Prot}}) \Rightarrow (\text{ref}(\emptyset)) \prec r_{\text{Prot}}
\]
3) Method Type in Permission: Method annotations usually indicate the requirements and effects of method calls. Besides traditional “reads” and “writes,” our system further includes “requires,” “uses” as well as “<.” Different annotations are interpreted in different ways.

- “reads ef” and “writes ef”: The caller needs to provide a read (fractional) and a write (whole) permission respectively.

- “requires lock”: The lock is required to be held at the method entry, so the caller is responsible to provide an unwrapped permission $\Pi \rhd \Pi_{holding}$ and the callee returns it back equally.

- “uses lock”: The lock will be acquired inside of the method body. It needs the caller to provide a wrapped $\Omega_{\text{locking}}(r_{lock})$ to allow a lock acquisition as well as a shared $\Omega_{\text{locking}}(r_{lock}<r_{holding})$, where $r_{holding}$ is the surrounding lock when this call happens.

- lock < lock′: A partial order among two locks is interpreted as a shared fact: $\Omega_{\text{locking}}(r_{lock}<r_{lock′})$.

Each method type is a mapping from an input to an output environment. Assuming the call happens in a latest synchronized block with holding $r_{holding}$ ($h_{holding}$ is the dimension of its wrapped permission when it becomes an unwrapped one), then a method type $\Delta; \Pi \rhd \Pi_{\text{locking}} \rhd \Pi′; \Pi′$ is a polymorphic over variables in $\Delta$. It accepts the input $\Pi$ and returns $\Pi′$ using perhaps some new variables in $\Delta′$ as well as the existing $\Delta$. For instance, the type for deposit is

$$\text{Account}(r_{\text{this}},r_{\text{holding}},z)$$

and

$$\text{saving2checking}(\text{Int} m)$$

void saving2checking(m) {
    synch saving do {
        saving.withdraw(m);
        synch checking do {
            checking.deposit(m);
        }
    }
}

All shared facts in $\Pi_{\Delta}$, $\Pi_{\Delta′}$ and $\Pi_{\Delta''}$ have been made conditional since a thread-local lock does not need the ordering requirement.

4) Examples: We briefly show how the FORK rule works for deadlock and deadlock-free methods in Figure 7.

```java
/**@reads (m.All,checking,saving)
  @uses (checking,saving), checking < saving*/
void saving2checking(Int m) {
    synch saving do {
        saving.withdraw(m);
        synch checking do {
            checking.deposit(m);
        }
    }
}
```

Since both of them use the same annotation as checking2saving, we borrow notations $\Pi_1,...,\Pi_6$ above for brevity.

At the beginning of method deadlock, all (conditional) shared permissions are $\Pi_1$, $\Pi_5$, $\Pi_6$ which should be applied $\omega$ and Gen<. Then, $\Pi_6$ shows up at (1), while $\omega(\Pi_6)$ which is $(1<_p c \land 1<_p p) \Rightarrow \Pi_{\Pi'}(r_{<\text{locking}})$ will be at (2) and (3). The call of checking2saving at (2) is perfectly fine with this order, but the saving2checking call at (3) does need

4A thread-local object is an object that is created in the current thread and it is recognized as $\Omega^0(\rho)$ in our system.
a different order expressed by a shared fact \((1 < p'_c \land 1 < p'_s) \Rightarrow \Omega^p(r_c < r_s)\) for some variables \(p'_c, p'_s\) and \(p\). The caller cannot provide the permission that the callee requires, thus the permission checking fails.

At (4), the shared permissions are similar to the ones at (1) except an additional \(\Omega^\mu(r_a < r_c)\) representing a new created local variable \(new\). The \(\omega\) works as before, but \(Gen_\omega\) may produce two one-dimension shared facts \(\Omega^1(r_a < r_c)\) and \(\Omega^1(r_a < r_s)\) at (5) and (6), with which the ordering for the following two lock acquisitions is satisifiable. New shared facts will be eliminated at (7) and \(Gen_\omega\) may produce different two one-dimension shared facts at (8) and (9): \(\Omega^1(r_a < r_s)\) and \(\Omega^1(r_a < r_c)\), which also fit for the code. Since the lock acquisition always follows the order expressed by the shared facts, the deadlock condition is excluded.

IV. CONSISTENCY AND SOUNDNESS

In order to make sure that a permission checked program can never have data races and deadlock at runtime, we need to match the static environment \(E\) and dynamic runtime state \(\mu\) according to pre-defined operational semantics and permission rules. We call this property “consistency”.

A. Prerequisite for Consistency

Any permission \(\Pi\) may use three kinds of variables: object variables, fraction variables or dimension variables which should be substituted by absolute addresses, numbers in \((0..1]\) and nature numbers respectively. We define a \(\sigma\) to substitute away all the variables (expressed as \(\sigma : \Delta \rightarrow \emptyset\)) in \(\Pi\).

Then, there are two assumptions to ‘witness’ the consistency:

- \(A_\alpha\) for nesting predicates;
- \(A_p\) for class invariant predicates.

They are paired as \(h = (A_\alpha, A_p)\), with which any fact can be evaluated to get a truth value: \(h \vdash \Gamma \Downarrow \text{bool}\).

Permissions are defined in complicated forms: fractional, conditional, shared ..., but the memory \(\mu\) is very simple. How to match all kinds of permissions to the memory? We use a fractional heap to bridge them which maps every location to a pair of a positive fraction and an object value. \(\emptyset\) is a particular object reference represented as “null” pointer and uses 0 as its fraction:

\[
th \in \text{Fractional Heap} = (O \times F) \rightarrow ((\mathbb{Q}^+, O) \cup \{(0, \emptyset)\})
\]

We use \(l\) to range over addresses \((o,f)\).

Definition 1.1 (Empty Fractional Heap): The empty fractional heap (written \(\emptyset\)) maps every address to \((0,\emptyset)\): \(\emptyset(l) = (0,\emptyset)\).

Definition 1.2 (Combination of Fractional Heaps): Given two fractional heaps \(h_1\) and \(h_2\), then for any \(l \in \text{Dom}(h_1) \cup \text{Dom}(h_2)\), \n
\[
(h_1 \oplus h_2)(l) = \begin{cases} h_1(l) & \text{if } \text{fst}(h_2(l)) = 0 \\ h_2(l) & \text{if } \text{fst}(h_1(l)) = 0 \\ (q, \text{snd}(h_1(l))) & \text{if } \text{snd}(h_1(l)) = \text{snd}(h_2(l)) \\ q = \text{fst}(h_1(l)) + \text{fst}(h_2(l)) & \text{undefined} \\ \text{otherwise} \end{cases}
\]

\(h_1\) and \(h_2\) are compatible if their combination \(h_1 \oplus h_2\) is defined.

Any fractional heap must be consistent with the actual memory.

Definition 1.3: A fractional heap \(h\) is consistent with memory \(\mu\) (written \(h \leq \mu\)) iff \(\forall l \in \text{Dom}(h), (\text{fst}(h(l)) \in [0..1]) \land ((\text{snd}(h(l)) > 0) \Rightarrow ((l \in \text{Dom}(\mu)) \land (\text{snd}(h(l)) = \mu(l))))\).

Besides \(h\), we need another assumption \(\Pi\) to maintain the dynamic orders among locks.

Definition 1.4 (Flattening): Given a memory \(\mu\) with assumptions \(A\) and \(\Pi\), if a permission \(\Pi\) with an obligation permission \(\Psi\) can be modeled by a fractional heap \(h\) such that \(h \leq \mu\), then we say the permission \(\Pi\) can be flattened (written \(h; \Psi \models_{(A,\Pi)}\)), where the obligation \(\Psi\) is treated as a restricted permission that can be discharged symbolically from the \(h\).

Based on the flattening rules in Figure 11 in the Appendix, a read permission \(\xi l : r \in f(o)\) can be flattened as:

\[
\begin{align*}
\vdash \xi l \Downarrow q & \quad \{l \rightarrow o\} (l) = o \\
\{l \rightarrow (1,o)\} ; \emptyset & \Downarrow o(\xi l) \quad \text{CP-FIELD} \\
\{l \rightarrow (q,o)\} ; \emptyset & \Downarrow o(\xi l) \quad \text{CP-FRACT}
\end{align*}
\]

Flattening wrapped permissions may depend on whether the lock has already been held by some thread. If yes, then it is flattened to an empty fractional heap by CP-WRAPPEHOLD since some other thread borrowed it. If not, then it is flattened to the same fractional heap as its unwrapped one by CP-WRAPPEFREE. A shared fact cannot be flattened if it does not follow the orders given in \(\Pi\) by CP-SHAREDFACT.

B. Consistency Checking

We use \(\mu; (A,\Pi) \models \Pi\) to indicate that a variable-free permission \(\Pi\) is consistent with the memory \(\mu\) assuming \(A\) and \(\Pi\). This is given as a judgement:

\[
\vdash(o_1, ..., o_n) (\forall i \in [1..n-1] . o_i < o_{i+1} \in \mathbb{D} \land (o_n < o_1 \in \mathbb{D}) \land (p(o') \in A) \Rightarrow \mu \Downarrow ((r \rightarrow o^* ) | P(p) \Downarrow \text{true}) \land FV(\Pi) = \emptyset \\
\mu; (A,\Pi) \models \Pi \\
\mu; (A,\Pi) \models I\Pi \\
\]

Basically, there are three requirements for the consistency between the memory \(\mu\) with assumptions \((A,\Pi)\) and the variable-free permissions \(\Pi\):

- The assumption \(\mathbb{D}\) is used to give orders among locks and it must be acyclic;
- For any named predicate in \(A\), its entire definition (substitute object variables for object values) must be true;
- Given the \(\mu\) and \((A,\Pi)\), the permission \(\Pi\) can be flattened to a fractional heap \(h\) such that \(h\) is consistent with \(\mu\).

C. Soundness

The fundamental soundness of this permission type system depends on a theorem:
**Theorem 3.1 (Progress and Preservation):** If a well-typed expression e dynamically nested most recently in a synchronized block holding oL can be checked by a variable-free permission Π such that \( \emptyset; \Pi \vdash o_L : e \in \tau \in \rho \vdash \Delta''; \Pi'' \), and a memory μ with assumptions (\( A, \emptyset \)) is consistent with Π, then either e is a value of the form o or e can be evaluated one-step further (\( (μ; (e)_{oL}) \vdash (μ'; (e')_{oL}) \) and there exists \( Π', σ \) and (\( A', \emptyset' \)) such that σ will substitute away some of the new type variables (σ : \( Δ \to \emptyset \) with \( Δ \subseteq Δ'' \), \( Π' \vdash o_{μ'} : e' : r(e, σ) \vdash Δ''; σΠ'' \)) and \( μ' \) with assumptions (\( A', \emptyset' \)) is consistent with Π' where \( A \subseteq A' \).

**proof:** We combine the permission type rules with the operational semantics defined in section II and prove by induction on permission checking rules case by case. This is similar to our previous work [9], [10].

Soundness indicates that a well-typed program can never go wrong. Here, we say a program is well-typed if all the method’s bodies are well-typed and can be checked by their method types in permission.

**Theorem 3.2:** A well-typed program is guaranteed to be free of data races and deadlocks.

**proof:** We distinguish four operations: read (R), write (W), read with lock ([R]L), write with lock ([W]L). In order to prove data-race free, it’s sufficient to show that for any location:

- Case 1: if one \( W \) happens, none of the \( R, W, [R]L, [W]L \) can happen in parallel threads;
- Case 2: if one \( [W]L \) happens, neither \( R \) nor \( W \) can happen in parallel threads.

In addition, the maintenance of lock orders according to the acyclic \( D \) in permission checking makes sure that any lock with a higher level cannot be acquired when the current thread is holding some lower orders.

**V. RELATED WORK**

Flanagan et al. [5], [12] introduce a static race detection analysis for multithreaded Java programs using similar annotations as ours. However, every field in their system must have a guard, thus no state is delegated to the holder of the reference. Boyapati et al. [6], [7] use a variant of ownership types to prevent data races. They do not protect individual fields directly. Instead, the object’s state is protected by its owner. They also prevent deadlock albeit with statically fixed lock levels. However, neither the above related work includes a formally defined dynamic semantics of synchronization, to our knowledge.

Kobayashi [8] introduces an advanced type systems for linearity, deadlock-free using π-calculus. In order to prevent deadlocks, he associated each input and output action an *obligation level* and a *capability level* in order to prevent cyclic dependencies between communications.

Brookes [1] defines a semantics of concurrency in separation logic. He converts every *command* into action traces and uses separation logic to prove all the possible interleavings between parallel traces are race free. The soundness property has been established, but there is no method call and pointer alias in his language, since it seems they are very hard to be handled and may cause infinite recursion. Moreover, since he uses action traces to simulate all the possibilities of interleavings, the number may be exponential.

Greenhouse et al. [2], [13] uses annotations and policy to express the concurrency-related design intents for a Java-style shared-memory programs. They use annotations to express some properties such as lock-state associations, uniqueness of some references and the aggregations of some states. They use the concurrency policy of a class implementation to specify which methods have potential executions that can be interleaved with others safely. A potential race condition exists if a conservative analysis cannot assure consistent regard to the defined policy, while we use class invariants to indicate protections associated to each field definition.

**VI. CONCLUSION**

This paper shows an ongoing work with focusing on formalization of the permission system. We extend the current permission system with shared permissions based on a simple OO language with structural parallelism and synchronization. Our permission type system requires adding some additional annotations to express field protection mechanism as well as lock order. Furthermore, some lock orders are allowed to be thread scoped, which means they can be altered according to the creation and elimination of parallel threads. We establish the consistency between the runtime memory and the static permission types, based on which we show the soundness.

A well-typed program is guaranteed to be free of data races and deadlocks.

**APPENDIX**

Figure 8. Operational semantics.

**REFERENCES**

Local $\Delta; \Pi; \xi$; $\Delta; \Pi; \xi$; $\Delta; \Pi; \xi$; $\Delta; \Pi; \xi$; $\Delta; \Pi; \xi$

Write

$\Delta; \Pi; \xi; \Delta; \Pi; \xi; \Delta; \Pi; \xi; \Delta; \Pi; \xi; \Delta; \Pi; \xi$

Informal: $\Delta; \Pi; \xi; \Delta; \Pi; \xi; \Delta; \Pi; \xi; \Delta; \Pi; \xi; \Delta; \Pi; \xi$

Informal: $\Delta; \Pi; \xi; \Delta; \Pi; \xi; \Delta; \Pi; \xi; \Delta; \Pi; \xi; \Delta; \Pi; \xi$

Method

$\Delta; \Pi; \xi; \Delta; \Pi; \xi; \Delta; \Pi; \xi; \Delta; \Pi; \xi; \Delta; \Pi; \xi$

$\Delta; \Pi; \xi; \Delta; \Pi; \xi; \Delta; \Pi; \xi; \Delta; \Pi; \xi; \Delta; \Pi; \xi$

Figure 9. Permission type rules.


Yang Zhao was born in 1978. He received his Ph.D. degree in computer science from University of Wisconsin, Milwaukee in 2007, his M.S. degree in computer science from Nanjing University in 2003. He currently is an Assistant Professor at Nanjing University of Sci.&Tech., China. His current research interests include program analysis and software engineering.

Ligong Yu was born in 1980. He received his M.S. degree in computer science from Zhejiang University in 2005. He is currently a Ph.D. candidate at Nanjing University of Sci.& Tech., China. His current research interests include software engineering and computer music.

Gongxuan Zhang was born in 1961. He received his M.S. and Ph.D. degrees in computer science from Nanjing University of Sci.& Tech. in 1991 and 2005, respectively. He is currently a Professor at Nanjing University of Sci.& Tech., China. His current research interests include distributed system, dependable computing and software engineering.

Jia Bei was born in 1979. He received his M.S. and Ph.D. degrees in computer science from Nanjing University in 2003 and 2006, respectively. He is currently an Assistant Professor at Nanjing University of China. His current research interests include distributed system, network security and e-commerce.