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 throwed 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

This paper is based on "The Scoped Permission System for Structural Parallel Programs," by Y. Zhao, L. Yu, G. Zhang and J. Bei, which appeared in the Proceedings of the 2nd IEEE International Conference on Computer, Control & Communication (IEEE-IC4), Karachi, Pakistan, Feb. 2009. © 2009 IEEE.
This work was supported in part by National Natural Science Foundation of China under Grant No. 60850002 and the Science and Technology Development Fund of NUST under Grant No. XKF09017.

```
P ::= defn*
defn ::= class  cn {field* meth*}
field ::= [FAn]opt cn f
meth ::= [MAn]* cn  m (arg*) {e}
FAn ::= guarded_by  lock
MAn ::= reads  e | writes  e | requires  lock
        | uses  lock | lock < lock
lock ::= this | g
cn ::= C<g*>
arg ::= cn   x
e ::= new  cn | x | e.f | e.f=e | e;e | let  x=e  in  e | e.m(e*)
    | synch  e  do  e | parbegin  (e||e)  parend

C, f, m ∈ class, field, method names
```

Figure 1. Language syntax.

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 "$lock_1 < lock_2$" 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:

$$(\mu; \langle e \rangle_L) \rightarrow (\mu'; \langle e' \rangle_L)$$

Given a memory μ, an expression e dynamically nested most recently in a synchronized block holding L (\$0 if none) can be one-step evaluated to e' with the memory being changed into μ' accordingly, where memory μ is defined as a mapping from locations (pairs of object address and field name) to other addresses:

$$\mu \in \text{Memory} = (O \times F) \rightharpoonup O$$

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.

[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;

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

  /**@requires this*/
  Int getbalance() {
    return balance
  }
......
```

Figure 2. `Account` class.

```
class Client {
  Account<this> checking, saving;

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

Figure 3. `Client` class.

E-Fork

$$(\mu; \langle \texttt{parbegin}\ (e_1 || e_2)\ \texttt{parend} \rangle_L) \rightarrow (\mu; \langle \langle e_1 \rangle_L || \langle e_2 \rangle_L \rangle_L)$$

E-Parallel

$$\frac{(\mu; \langle e_i \rangle_L) \rightarrow (\mu_i; \langle e_i' \rangle_L) \qquad (e_1'', e_2'') = (i == 1)?(e_1', e_2) : (e_1, e_2')}{(\mu; \langle \langle e_1 \rangle_L || \langle e_2 \rangle_L \rangle_L) \rightarrow (\mu_i; \langle \langle e_1'' \rangle_L || \langle e_2'' \rangle_L \rangle_L)}$$

E-Join

$$(\mu; \langle \langle o_1 \rangle_L || \langle o_2 \rangle_L \rangle_L) \rightarrow (\mu; \langle o_1 \rangle_L)$$

E-Acq

$$\frac{\mu(o_1, \_m) = \$0 \qquad \mu' = \mu[(o_1, \_m) \mapsto o_1]}{(\mu; \langle \texttt{synch}\ o_1\ \texttt{do}\ e_2 \rangle_L) \rightarrow (\mu'; \langle \texttt{hold}\ o_1\ \texttt{do}\ \langle e_2 \rangle_{o_1} \rangle_L)}$$

E-Rel

$$\frac{\mu(o_1, \_m) = o_1 \qquad \mu' = \mu[(o_1, \_m) \mapsto \$0]}{(\mu; \langle \texttt{hold}\ o_1\ \texttt{do}\ \langle o_2 \rangle_{o_1} \rangle_L) \rightarrow (\mu'; \langle o_2 \rangle_L)}$$

Figure 4. Selected evaluation rules

The E-Fork rule indicates that two sub-expressions e_1 and e_2 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 $\_m$ indicating whether or not the object has already been locked. If $\_m$ is null, its object is in an *unlocked* state and is free to be acquired (by setting the $\_m$ field to itself). E-ACQ shows the situation of a lock acquisition [2], while E-REL applies 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 $o.f : \texttt{ref}(o')$.

In order to distinguish reads from writes, we associate fractions ξ as well, such that a write permission is an explicit *whole* permission $\texttt{1}o.f : \texttt{ref}(o')$, while a read permission is just some *partial* permission $\xi o.f : \texttt{ref}(o')$ where ξ is syntactically guaranteed to be positive.

object reference	ρ	$::=$	$o \mid r$
key	k	$::=$	$\rho.f$
fraction	ξ	$::=$	$\texttt{1} \mid \texttt{1/2} \mid z \mid \xi\xi$
dimension	d	$::=$	$0 \mid d \pm 1 \mid p$
type	τ	$::=$	$\texttt{ref}(\rho)$
fact	Γ	$::=$	$\rho \in C \mid \rho{=}\rho \mid \Pi \prec k \mid C(\rho^*)$ $\mid d{<}d \mid \texttt{true} \mid \neg\Gamma \mid \Gamma \wedge \Gamma$
permissions	Π	$::=$	$\emptyset \mid \xi k : \tau \mid \Gamma \mid \Pi \multimap \Pi \mid \Pi \oplus \Pi$ $\mid \Gamma \Rrightarrow \Pi \mid \exists r\,.\,(\Pi) \mid \Omega^d(\rho)$ $\mid \Omega^d(\rho{<}\rho)$

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 $\xi\rho.f : \tau$ indicates some kind of right (depending on ξ) to access field instance $\rho.f$. The fraction ξ could be $\texttt{1}$, $\texttt{1/2}$, a variable z or a product. τ is a permission type which is given a regular pointer $\texttt{ref}(\rho)$.

[2]The special "$\texttt{hold}\ e_1\ \texttt{do}\ e_2$" is an internal expression that is not allowed to be used by program designers.

[3]Permissions are combined using the operator ($\oplus$) which is semantically commutative and associative with $\emptyset$ as the identity.

A *conditional permission* $\Gamma \Rrightarrow \Pi$ presents a possible Π depending on the truth value of Γ.

A *fact* Γ uses a simple logic that can be represented by boolean formulae over type assertions ($\rho \in C$), reference equalities ($\rho{=}\rho$), nesting ($\Pi \prec k$), object type predicates ($C(\rho^*)$), 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 $\Pi \prec k$), which implies that the nested permission is not available unless the nester one is granted.

The *linear implication* $\Pi_1 \multimap \Pi_2$ indicates that one has the rights of the consequent Π_2, except for the ones of the antecedent Π_1. Given a nesting fact $\Pi \prec k$, a nested Π can be carved out from its nester $\texttt{1}k : \tau$. The carving process is then:

$$\Pi \prec k \;\oplus\; \texttt{1}k : \tau \Rrightarrow \Pi \prec k \;\oplus\; \Pi \;\oplus\; \Pi \multimap \texttt{1}k : \tau$$

A *shared permission* $\Omega^d(\ldots)$ has two forms:

- a *wrapped permission* $\Omega^d(\rho)$ that will be transformed into $\texttt{1}\rho.\texttt{Prot} : \tau_{\$0}$ (called *unwrapped permission*) after acquiring lock ρ, but be resumed once the lock is released as described later in rule SYNCH;
- a *shared fact* $\Omega^d(\rho{<}\rho')$ 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 $\Omega^d(\ldots)$ 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 ρ_L (d_L is the dimension of $\Omega^{d_L}(\rho_L)$ when it changes to $\rho_L.\texttt{Prot} : \tau_{\$0}$), 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:

$$E \vdash^{d_L}_{\rho_L} e \Downarrow \tau \dashv E'$$

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 Π. For a well-formed environment $E = (\Delta; \Pi)$, we require that all free variables used in Π are in Δ ($FV(\Pi) \subseteq \Delta$).

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

NEW

$$\frac{r \text{ fresh} \qquad f_i \in \text{Fields}(C) \qquad \Pi' = r \in C \oplus \Omega^1(r) \oplus 1r.f_i : \texttt{ref}(\$0)}{\Delta;\Pi \vdash^{d_L}_{\rho_L} \texttt{new}\ C \Downarrow \texttt{ref}(r) \dashv \{r\} \cup \Delta;\Pi \oplus \Pi'}$$

SYNCH

$$\frac{\begin{array}{c}\Delta;\Pi \vdash^{d_L}_{\rho_L} e_1 \Downarrow \texttt{ref}(\rho_1) \dashv \Delta';\Pi'_1 \oplus \Pi'' \\ \Pi'_1 = \Omega^{d_1}(\rho_1) \oplus (1{<}d_1 \wedge 1{<}d_L) \Rightarrow \Omega^{d_2}(\rho_1{<}\rho_L) \\ \Delta'; 1\rho_1.\texttt{Prot} : \tau_{\$0} \oplus \Pi'' \vdash^{d_1}_{\rho_1} e_2 \Downarrow \texttt{ref}(\rho_2) \dashv \Delta''';\Pi'''_1 \\ \Pi'''_1 = 1\rho_1.\texttt{Prot} : \tau_{\$0} \oplus \Pi'''\end{array}}{\Delta;\Pi \vdash^{d_L}_{\rho_L} \texttt{synch}\ e_1\ \texttt{do}\ e_2 \Downarrow \texttt{ref}(\rho_2) \dashv \Delta''';\Pi'_1 \oplus \Pi'''}$$

FORK

$$\frac{\begin{array}{c}\Pi = \Pi_1 \oplus \Pi_2 \oplus \Pi_\Omega \\ \Pi_i \text{ contain no } \Omega^d(\ldots) \text{ permissions} \\ \Pi'_\Omega = \omega(\Pi_\Omega) \oplus \text{Gen}_<(\Pi_\Omega) \\ \{r_{T_i}\} \cup \Delta; \Pi_i \oplus \Pi'_\Omega \vdash^{d_L}_{\rho_L} e_i \Downarrow \texttt{ref}(\rho_i) \dashv \Delta'_i;\Pi'_i \\ E' = \Delta'_1 \cup \Delta'_2; \omega^{-1}(\Pi'_1 \oplus \Pi'_2)\end{array}}{\Delta;\Pi \vdash^{d_L}_{\rho_L} \texttt{parbegin}\ (e_1 || e_2)\ \texttt{parend} \Downarrow \texttt{ref}(\rho_1) \dashv E'}$$

Figure 6. Selected permission rules

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 $\texttt{ref}(\rho_1)$ for some ρ_1. (2) The wrapped permission $\Omega^{d_1}(\rho_1)$ should be present right before entering the synchronized block. It is also required to compare the latest holding lock ρ_1 with ρ_L 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 $(1{<}d_1 \wedge 1{<}d_L) \Rightarrow \Omega^{d_2}(\rho_1{<}\rho_L)$ shows that only if neither locks is thread-local then the shared fact appears. (3) The wrapped permission $\Omega^{p_1}(\rho_1)$ is transformed into an unwrapped one $1\rho_1.\texttt{Prot} : \tau_{\$0}$ 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 $\Omega^{p_1}(\rho_1)$ guarantees that none of the nested permissions in $\rho_1.\texttt{Prot}$ is available without acquiring lock ρ_1 first.

The FORK shows that a parallel composition can be permission-checked based on an input Π only if the Π can be split into three parts Π_1, Π_2, and Π_Ω, such that Π_i goes into a corresponding child thread while Π_Ω remains shared. Π_Ω will undergo a transformation $\omega(\Pi_\Omega)$ 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 ω^{-1}.

$\to$	ω	ω^{-1}
$\emptyset$	$\emptyset$	$\emptyset$
Γ	Γ	Γ
Ψ	Ψ	Ψ
$\Omega^d(\ldots)$	$\Omega^{d+1}(\ldots)$	$\Omega^{d-1}(\ldots)$ with $1 < d$
$\Pi_1 \mathbin{-\!\!+} \Pi_2$	$\omega(\Pi_1) \mathbin{-\!\!+} \omega(\Pi_2)$	$\omega^{-1}(\Pi_1) \mathbin{-\!\!+} \omega^{-1}(\Pi_2)$
$\Gamma \Rightarrow \Pi$	$\Gamma \Rightarrow \omega(\Pi)$	$\Gamma \Rightarrow \omega^{-1}(\Pi)$
$\Pi_1 \oplus \Pi_2$	$\omega(\Pi_1) \oplus \omega(\Pi_2)$	$\omega^{-1}(\Pi_1) \oplus \omega^{-1}(\Pi_2)$

The ω operation increases the dimension of sharing on each shared permission, but leaves others immutable, while ω^{-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:

$$\omega^{-1}(\Omega^1(\rho)) = \Omega^1(\rho) \qquad \omega^{-1}(\Omega^1(\rho < \rho')) = \emptyset$$

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

The $\text{Gen}_<(\Omega^1(\rho_1), \ldots, \Omega^1(\rho_n), \Pi_s)$ 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 ρ *lower* than all previous ordered locks ($\Omega^{d+1}(\rho')$ in Π_s, or an earlierly considered unshared lock).
- It places ρ *higher* than all previous ordered locks.
- It places ρ *between* an existing order ($\Omega^d(\rho' < \rho'')$ in Π_s).

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 `Prot` field of *lock* which is an implicit location with an uninteresting type $\tau_{\$0}$, such as $1r.f : \tau \prec r_{lock}.\texttt{Prot}$ for some τ. Without this annotation, the whole permission of a field goes into $r_{\texttt{this}}.\texttt{All}$, where the `All` is considered as a location collecting all permissions except for those that are protected by guard objects. Analogously, `All` field is given the type $\tau_{\$0}$ as well.

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

$$\begin{array}{l} \mathit{Account}(r_{\texttt{this}}, r_{\texttt{g}}) = (r_{\texttt{this}} \in \texttt{Account}) \wedge \Gamma_{\texttt{b}} \quad \text{where} \\ \Gamma_{\texttt{b}} = \exists r\,\textbf{.}\,(1r_{\texttt{this}}.\texttt{balance} : \texttt{ref}(r) \oplus \\ \qquad \neg r = \$0 \Rightarrow (\mathit{Int}(r) \oplus 1r.\texttt{All} : \tau_{\$0})) \prec r_{\texttt{g}}.\texttt{Prot} \end{array}$$

The first conjunct shows the static type for $r_{\texttt{this}}$, while the second is a field invariant for the `balance` field. $\Gamma_{\texttt{b}}$ states that the whole permission of the field access is nested in its protector $r_{\texttt{g}}.\texttt{Prot}$. Moreover, if it is not null, then its pointed-to object must be an `Int` object with holding a class invariant and the whole permission of $r.\texttt{All}$ will also be nested in $r_{\texttt{g}}.\texttt{Prot}$. 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 $r_{\texttt{this}}.\texttt{All}$ because no "guarded_by" is attached.

$$\begin{array}{l} \mathit{Client}(r_{\texttt{this}}) = (r_{\texttt{this}} \in \texttt{Client}) \wedge \Gamma_{\texttt{c}} \wedge \Gamma_{\texttt{s}} \quad \text{where} \\ \Gamma_{\texttt{c}} = \exists r\,\textbf{.}\,(1r_{\texttt{this}}.\texttt{checking} : \texttt{ref}(r) \oplus \neg r = \$0 \Rightarrow \\ \qquad (\mathit{Account}(r, r_{\texttt{this}}) \oplus 1r.\texttt{All} : \texttt{ref}(\$0))) \prec r_{\texttt{this}}.\texttt{All} \\ \Gamma_{\texttt{s}} = \exists r\,\textbf{.}\,(1r_{\texttt{this}}.\texttt{saving} : \texttt{ref}(r) \oplus \neg r{=}\$0 \Rightarrow \\ \qquad (\mathit{Account}(r, r_{\texttt{this}}) \oplus 1r.\texttt{All} : \texttt{ref}(\$0))) \prec r_{\texttt{this}}.\texttt{All} \end{array}$$

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` *e.f*" and "`writes` *e.f*": 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 $1r_{lock}.\texttt{Prot} : \tau_{\$0}$ 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 $\Omega^d(r_{lock})$ to allow a lock acquisition as well as a shared $\Omega^{d'}(r_{lock} < r_{\text{holding}})$, 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: $\Omega^d(r_{lock} < r_{lock'})$.

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} (p_{holding} is the dimension of its wrapped permission when it becomes an unwrapped one), then a method type $\Delta;\Pi \xrightarrow[r_{\text{holding}}]{p_{\text{holding}}} \Delta';\Pi'$ is a polymorphic over variables in Δ. It accepts the input Π and returns Π' using perhaps some new variables in Δ' as well as the existing Δ. For instance, the type for `deposit` is

$$\{r_{\texttt{this}}, r_{\texttt{g}}, r_{\texttt{m}}, r_{\text{holding}}, p_{\text{holding}}, z\};$$
$$Account(r_{\texttt{this}}, r_{\texttt{g}}) \oplus 1r_{\texttt{this}}.\texttt{Prot} : \tau_{\$0} \oplus zr_{\texttt{m}}.\texttt{All} : \tau_{\$0}$$
$$\xrightarrow[r_{\text{holding}}]{p_{\text{holding}}} \{r_{\texttt{ret}}\};$$
$$Account(r_{\texttt{this}}, r_{\texttt{g}}) \oplus 1r_{\texttt{this}}.\texttt{Prot} : \tau_{\$0} \oplus zr_{\texttt{m}}.\texttt{All} : \tau_{\$0}$$
$$\oplus\, r_{\texttt{ret}}{=}\$0$$

where $r_{\texttt{this}}$, $r_{\texttt{g}}$ and $r_{\texttt{m}}$ represent *this* object, class parameter, method parameter respectively; r_{holding} and p_{holding} are the holding lock for the latest synchronized block and its dimension (as mentioned before); z is a fraction; $r_{\texttt{ret}}$ is the returned object.

The input permission first assumes that the class invariant for *Account* is held, then gives an unwrapped permission and a read permission coming from two method annotations respectively. The output permission additionally includes a fact that the return value is null.

If a method acquires locks by itself, then the input permissions may include some shared facts to indicate the order. For instance, the permission representations for annotations in `checking2saving` are:

$$\Pi_1 = z_m r_{\texttt{m}}.\texttt{All} : \tau_{\$0}$$
$$\Pi_2 = z_c r_{\texttt{this}}.\texttt{checking} : \texttt{ref}(r_{\texttt{c}})$$
$$\Pi_3 = z_s r_{\texttt{this}}.\texttt{saving} : \texttt{ref}(r_{\texttt{s}})$$
$$\Pi_4 = \Omega^{p_c}(r_{\texttt{c}}) \oplus (1{<}p_c \wedge 1{<}p_{\text{holding}}) \Rightarrow \Omega^{p_1}(r_{\texttt{c}} {<} r_{\text{holding}})$$
$$\Pi_5 = \Omega^{p_s}(r_{\texttt{s}}) \oplus (1{<}p_s \wedge 1{<}p_{\text{holding}}) \Rightarrow \Omega^{p_2}(r_{\texttt{s}} {<} r_{\text{holding}})$$
$$\Pi_6 = (1{<}p_c \wedge 1{<}p_s) \Rightarrow \Omega^{p_3}(r_{\texttt{s}} {<} r_{\texttt{c}})$$
$$\hat{\Pi} = \Pi_1 \oplus \Pi_2 \oplus \Pi_3 \oplus \Pi_4 \oplus \Pi_5 \oplus \Pi_6$$

where Π_1, Π_2, Π_3 originate from three "`reads`" respectively; Π_4, Π_5 are for two "`uses`", while Π_6 interprets the "...<...". Then its method type is:

$$\{r_{\texttt{this}}, r_{\texttt{m}}, r_{\texttt{c}}, r_{\texttt{s}}, r_{\text{holding}}, p_{\text{holding}}, z_m, z_c, z_s, p_c, p_s, p_1, p_2, p_3\};$$
$$Client(r_{\texttt{this}}) \oplus \hat{\Pi}$$
$$\xrightarrow[r_{\text{holding}}]{p_{\text{holding}}} \{r_{\texttt{ret}}\}; \hat{\Pi} \oplus r_{\texttt{ret}}{=}\$0$$

All shared facts in Π_4, Π_5 and Π_6 have been made conditional since a thread-local lock [4] 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.

```
/**@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)
  }
}

/**@reads (m.All,checking,saving),
uses (checking,saving), saving<checking*/
Int deadlock(Int m) {
     (1)
 parbegin (
            (2)                 ||         (3)  //Error!
   checking2saving(m)           ||  saving2checking(m)
 ) parend
}

/**@reads (m.All,checking,saving),
uses (checking,saving), saving<checking*/
Int deadlock-free(Int m) {
  let newa = new Account<this>(this) in {
    (4)
   parbegin (
               (5)              ||        (6)
     synch checking do          ||  synch checking do
       synch newa do            ||    synch newa do
         ...                    ||      ...
   ) parend;
     (7)
   parbegin (
            (8)                 ||        (9)
     synch newa do              ||  synch newa do
       synch saving do          ||    synch saving do
         ...                    ||      ...
   ) parend
   }
 }
```

Figure 7. Other methods in `Client`.

Since both of them use the same annotation as `checking2saving`, we borrow notations Π_1,...,Π_6 above for brevity.

At the beginning of method `deadlock`, all (conditional) shared permissions are Π_4, Π_5, Π_6 which should be applied ω and $\text{Gen}_<$. Then, Π_6 shows up at (1), while $\omega(\Pi_6)$ which is $(1{<}p_c \wedge 1{<}p_s) \Rightarrow \Omega^{p_3+1}(r_{\texttt{s}} {<} r_{\texttt{c}})$ 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

[4]A thread-local object is an object that is created in the current thread and it is recognized as $\Omega^1(\rho)$ in our system.

a different order expressed by a shared fact $(1{<}p'_c \wedge 1{<}p'_s) \Rightarrow \Omega^{p'}(r_c{<}r_s)$ for some variables p'_c, p'_s and p'. The caller can not 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 $\Omega^1(r_a)$ representing a new created local variable `newa`. The ω works as before, but $\text{Gen}_<$ 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 satiable. New shared facts will be eliminated at (7) and $\text{Gen}_<$ may produce different two one-dimension shared facts at (8) and (9): $\Omega^1(r_s{<}r_a)$ 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 μ according to pre-defined operational semantics and permission rules. We call this property "consistency".

A. Prerequisite for Consistency

Any permission Π 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 σ to substitute away all the variables (expressed as $\sigma : \Delta \to \emptyset$) in Π.

Then, there are two assumptions to 'witness" the consistency:

- A_{in} for nesting predicates;
- A_{P} for class invariant predicates.

They are paired as $\mathbb{A} = (A_{\text{in}}, A_{\text{P}})$, with which any fact can be evaluated to get a truth value: $\mathbb{A} \vdash \Gamma \Downarrow \text{bool}$.

Permissions are defined in complicated forms: fractional, conditional, shared ..., but the memory μ 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. \$0 is a particular object reference represented as "null" pointer and uses 0 as its fraction:

$$h \in \text{Fractional Heap} = (O \times F) \to ((\mathbf{Q}^+, O) \cup \{(0, \$0)\})$$

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

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

Definition 1.2 (Combination of Fractional Heaps): Given two fractional heaps h_1 and h_2, then for any $l \in \text{Dom}(h_1) \cup \text{Dom}(h_2)$,

$$(h_1 \hat{+} h_2)(l) = \begin{cases} h_1(l) & \text{if fst}(h_2(l)) = 0 \\ h_2(l) & \text{if fst}(h_1(l)) = 0 \\ (q, \text{snd}(h_1(l))) & \text{if snd}(h_1(l)) = \text{snd}(h_2(l)) \\ & q = \text{fst}(h_1(l)) + \text{fst}(h_2(l)) \\ \text{undefined} & \text{otherwise} \end{cases}$$

h_1 and h_2 are *compatible* if their combination $h_1 \hat{+} h_2$ is defined.

Any fractional heap must be consistent with the actual memory.

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

Besides $\mathbb{A}$, we need another assumption $\mathbb{D}$ to maintain the dynamic orders among locks.

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

Based on the flattening rules in Figure 11 in the Appendix, a read permission $\xi l : \texttt{ref}(o)$ can be flattened as:

$$\dfrac{\vdash \xi \Downarrow q \qquad \dfrac{\{l \mapsto o\}(l) = o}{\{l \mapsto (1, o)\}\,; \emptyset \models_{\{l \mapsto o\}}^{(\emptyset,\emptyset)} l : \texttt{ref}(o)}\text{CP-FIELD}}{\{l \mapsto (q, o)\}\,; \emptyset \models_{\{l \mapsto o\}}^{(\emptyset,\emptyset)} \xi l : \texttt{ref}(o)}\text{CP-FRAC}$$

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-WRAPPEDHOLD since some other thread borrowed it. If not, then it is flattened to the same fractional heap as its unwrapped one by CP-WRAPPEDFREE. A shared fact cannot be flattened if it does not follow the orders given in $\mathbb{D}$ by CP-SHAREDFACT.

B. Consistency Checking

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

$$\dfrac{\begin{array}{c} \nexists(o_1, ..., o_n).(\forall i \in [1..n-1].o_i < o_{i+1} \in \mathbb{D}) \wedge (o_n < o_1 \in \mathbb{D}) \\ (p(o^*) \in \mathbb{A}) \Rightarrow \mathbb{A} \vdash [(r \mapsto o)^*]P(p) \Downarrow \text{true} \\ FV(\Pi) = \emptyset \qquad h; \emptyset \models_\mu^{(\mathbb{A},\mathbb{D})} \Pi \qquad h \leq \mu \end{array}}{\mu; (\mathbb{A}, \mathbb{D}) \models \Pi}$$

Basically, there are three requirements for the consistency between the memory μ with assumptions $(\mathbb{A}, \mathbb{D})$ and the variable-free permissions Π:

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

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 o_L can be checked by a variable-free permission Π such that $\emptyset; \Pi \vdash^{n}_{o_L} e : \texttt{ref}(\rho) \dashv \Delta''; \Pi''$, and a memory μ with assumptions $(\mathbb{A}, \mathbb{D})$ is consistent with Π, then either e is a value of the form o or e can be evaluated one-step further $(\mu; \langle e \rangle_{o_L}) \rightarrow (\mu'; \langle e' \rangle_{o_L})$ and there exists Π', σ and $(\mathbb{A}', \mathbb{D}')$ such that σ will substitute away some of the new type variables ($\sigma : \Delta \rightarrow \emptyset$ with $\Delta \subseteq \Delta''$), $\emptyset; \Pi' \vdash^{o_L}_{n} e' : \texttt{ref}(\sigma\rho) \dashv \sigma\Delta''; \sigma\Pi''$ and μ' with assumptions $(\mathbb{A}', \mathbb{D}')$ is consistent with Π' where $\mathbb{A} \subseteq \mathbb{A}'$.

proof(Sketch): 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(Sketch): We distinguish four operations: read ($\mathcal{R}$), write ($\mathcal{W}$), read with lock ($[\mathcal{R}]_L$), write with lock ($[\mathcal{W}]_L$). In order to prove data-race free, it's sufficient to show that for any location:

- Case 1: if one $\mathcal{W}$ happens, none of the $\mathcal{R}$, $\mathcal{W}$, $[\mathcal{R}]_L$, $[\mathcal{W}]_L$ can happen in parallel threads;
- Case 2: if one $[\mathcal{W}]_L$ happens, neither $\mathcal{R}$ nor $\mathcal{W}$ can happen in parallel threads.

In addition, the maintenance of lock orders according to the acyclic $\mathbb{D}$ 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 semantic 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

E-Write1
$$\frac{(\mu; \langle e_1 \rangle_L) \rightarrow (\mu'; \langle e_1' \rangle_L)}{(\mu; \langle e_1.f{=}e_2 \rangle_L) \rightarrow (\mu'; \langle e_1'.f{=}e_2 \rangle_L)}$$

E-Write2
$$\frac{(\mu; \langle e_2 \rangle_L) \rightarrow (\mu'; \langle e_2' \rangle_L)}{(\mu; \langle o_1.f{=}e_2 \rangle_L) \rightarrow (\mu'; \langle o_1.f{=}e_2' \rangle_L)}$$

E-Write
$$\frac{\mu' = \mu[(o_1, f) \mapsto o_2]}{(\mu; \langle o_1.f{=}o_2 \rangle_L) \rightarrow (\mu'; \langle o_2 \rangle_L)}$$

E-Invoke1
$$\frac{(\mu; \langle e_0 \rangle_L) \rightarrow (\mu'; \langle e_0' \rangle_L)}{(\mu; \langle e_0.m(e^*) \rangle_L) \rightarrow (\mu'; \langle e_0'.m(e^*) \rangle_L)}$$

E-Invoke2
$$\frac{(\mu; \langle e_i \rangle_L) \rightarrow (\mu'; \langle e_i' \rangle_L)}{(\mu; \langle o_0.m(o^*, e_i, e^*) \rangle_L) \rightarrow (\mu'; \langle o_0.m(o^*, e_i', e^*) \rangle_L)}$$

E-Invoke
$$\frac{\text{mbody}(\text{class}(o_0), m) = ((cn\ x)^*, locks, e, \alpha)}{(\mu; \langle o_0.m(o^*) \rangle_L) \rightarrow (\mu; \langle e[this \mapsto o_0, x_i \mapsto o_i] \rangle_L)}$$

E-Seq1
$$\frac{(\mu; \langle e_1 \rangle_L) \rightarrow (\mu'; \langle e_1' \rangle_L)}{(\mu; \langle e_1 ; e_2 \rangle_L) \rightarrow (\mu'; \langle e_1' ; e_2 \rangle_L)}$$

E-Seq
$$(\mu; \langle o_1 ; e_2 \rangle_L) \rightarrow (\mu; \langle e_2 \rangle_L)$$

E-Local
$$(\mu; \langle \texttt{let}\ x{=}o_1\ \texttt{in}\ e_2 \rangle_L) \rightarrow (\mu; \langle [x \mapsto o_1]e_2 \rangle_L)$$

E-Local1
$$\frac{(\mu; \langle e_1 \rangle_L) \rightarrow (\mu'; \langle e_1' \rangle_L)}{(\mu; \langle \texttt{let}\ x{=}e_1 ... \rangle_L) \rightarrow (\mu'; \langle \texttt{let}\ x{=}e_1' ... \rangle_L)}$$

E-New
$$\frac{\forall f \in \text{fields}(C).(o, f) \notin \text{Dom}(\mu)}{(\mu; \langle \texttt{new}\ C \rangle_L) \rightarrow (\mu[(o, f) \mapsto \$0]; \langle o \rangle_L)}$$

E-Sync1
$$\frac{(\mu; \langle e_1 \rangle_L) \rightarrow (\mu'; \langle e_1' \rangle_L)}{(\mu; \langle \texttt{synch}\ e_1 ... \rangle_L) \rightarrow (\mu'; \langle \texttt{synch}\ e_1' ... \rangle_L)}$$

E-Read1
$$\frac{(\mu; \langle e \rangle_L) \rightarrow (\mu'; \langle e' \rangle_L)}{(\mu; \langle e.f \rangle_L) \rightarrow (\mu'; \langle e'.f \rangle_L)}$$

E-Read
$$\frac{o' = \mu(o, f)}{(\mu; \langle o.f \rangle_L) \rightarrow (\mu; \langle o' \rangle_L)}$$

E-Hold
$$\frac{\mu(o_1, \_m) = o_1 \qquad (\mu; \langle e_2 \rangle_{o_1}) \rightarrow (\mu'; \langle e_2' \rangle_{o_1})}{(\mu; \langle \texttt{hold}\ o_1\ \texttt{do}\ \langle e_2 \rangle_{o_1} \rangle_L) \rightarrow (\mu'; \langle \texttt{hold}\ o_1\ \texttt{do}\ \langle e_2' \rangle_{o_1} \rangle_L)}$$

Figure 8. Operational semantics.

References

[1] S. Brookes, "A semantics for concurrent separation logic," in *CONCUR '04*, Aug. 2004.

VARIABLE

$$\frac{r_x \in \Delta}{\Delta;\Pi \vdash^{d_L}_{\rho_L} x : \texttt{ref}(r_x) \dashv \Delta;\Pi}$$

OBJLOC

$$\Delta;\Pi \vdash^{d_L}_{\rho_L} o : \texttt{ref}(o) \dashv \Delta;\Pi$$

SEQ

$$\frac{\Delta;\Pi \vdash^{d_L}_{\rho_L} e_1 : \texttt{ref}(\rho_1) \dashv \Delta';\Pi' \quad \Delta';\Pi' \vdash^{d_L}_{\rho_L} e_2 : \texttt{ref}(\rho_2) \dashv \Delta'';\Pi''}{\Delta;\Pi \vdash^{d_L}_{\rho_L} e_1 ; e_2 : \texttt{ref}(\rho_2) \dashv \Delta'';\Pi''}$$

READ

$$\frac{\Delta;\Pi \vdash^{d_L}_{\rho_L} e : \texttt{ref}(\rho) \dashv \Delta';\Pi' \quad \Pi' = \xi\rho.f : \texttt{ref}(\rho_f), \Pi''}{\Delta;\Pi \vdash^{d_L}_{\rho_L} e.f : \texttt{ref}(\rho_f) \dashv \Delta';\Pi'}$$

WRITE

$$\frac{\Delta;\Pi \vdash^{d_L}_{\rho_L} e_1 : \texttt{ref}(\rho_1) \dashv \Delta';\Pi' \vdash^{d_L}_{\rho_L} e_2 : \texttt{ref}(\rho_2) \dashv \Delta'';\Pi'' \quad \Pi'' = 1\rho_1.f : \texttt{ref}(\rho_f), \Pi'''}{\Delta;\Pi \vdash^{d_L}_{\rho_L} e_1.f{=}e_2 : \texttt{ref}(\rho_2) \dashv \Delta''; 1\rho_1.f : \texttt{ref}(\rho_2), \Pi'''}$$

LOCAL

$$\frac{\Delta;\Pi \vdash^{d_L}_{\rho_L} e_1 : \texttt{ref}(\rho_1) \dashv \Delta';\Pi' \quad r_x \notin \Delta' \quad \{r_x\} \cup \Delta'; \rho_1 = r_x, \Pi' \vdash^{d_L}_{\rho_L} e_2 : \texttt{ref}(\rho_2) \dashv \Delta'';\Pi''}{\Delta;\Pi \vdash^{d_L}_{\rho_L} \texttt{let } x{=}e_1 \texttt{ in } e_2 : \texttt{ref}([r_x \mapsto \rho_1]\rho_2) \dashv \Delta'' \setminus \{r_x\}; [r_x \mapsto \rho_1]\Pi''}$$

PARALLEL

$$\frac{\Pi = \Pi_1, \Pi_2, \Pi_\Omega \quad \Delta_i; \Pi_i, \Pi_\Omega \vdash^{d_L}_{\rho_L} e_i : \texttt{ref}(\rho_i) \dashv \Delta'_i; \Pi'_i}{\Delta_1 \cup \Delta_2; \Pi \vdash^{d_L}_{\rho_L} \langle e_1 \rangle_{\rho_L} || \langle e_2 \rangle_{\rho_L} : \texttt{ref}(\rho_1) \dashv (\Delta'_1 \cup \Delta'_2); \omega^{-1}(\Pi'_1, \Pi'_2)}$$

HOLD

$$\frac{\Delta; 1o_1.\texttt{Prot} : \tau_{\$0}, \Pi' \vdash^{d_1}_{o_1} e_2 : \texttt{ref}(\rho_2) \dashv \Delta''; 1o_1.\texttt{Prot} : \tau_{\$0}, \Pi'' \quad \Pi'_1 = \Omega^{d_1}(o_1), (1{<}d_1 \wedge 1{<}d_L) \multimap \Omega^{d_2}(o_1{<}\rho_L)}{\Delta; \Pi'_1, \Pi' \vdash^{d_L}_{\rho_L} \texttt{hold } o_1 \texttt{ do } e_2 : \texttt{ref}(\rho_2) \dashv \Delta''; \Pi'_1, \Pi''}$$

INVOKE

$$\frac{\begin{array}{c} \Delta;\Pi \vdash^{d_L}_{\rho_L} e_0 : \texttt{ref}(\rho_0) \dashv \Delta_0;\Pi_0 \vdash^{d_L}_{\rho_L} e_1 : \texttt{ref}(\rho_1) \dashv \Delta_1;\Pi_1 \\ \Delta_1;\Pi_1 \vdash^{d_L}_{\rho_L} \ldots e_n : \texttt{ref}(\rho_n) \dashv \Delta_n;\Pi_n \quad \Pi_n = \rho_0 : C_0, \Pi'_n \\ \text{mbody}(C_0, m) = (x^*, e, \Delta'_0; \Pi'_0 \xrightarrow[r_{\text{holding}}]{p_{\text{holding}}} \Delta''_0; \sigma_2\Pi''_0) \\ \sigma_1 : \Delta'_0 \to \Delta_n \quad \Delta' \text{ fresh} \quad \sigma_2 : \Delta' \to \Delta''_0 \\ \Pi_n = \sigma_1\Pi'_0, \Pi' \quad \forall i \in [1..n].(\sigma_1(r_{x_i}) = \rho_i) \\ \sigma_1(r_{\text{this}}) = \rho_0 \quad \sigma_1(r_{\text{holding}}) = \rho_L \\ \sigma_1(p_{\text{holding}}) = d_L \quad r' \in \Delta' \quad \sigma_2(r') = r_{\text{ret}} \end{array}}{\Delta;\Pi \vdash^{d_L}_{\rho_L} e_0.m(e_1, \ldots, e_n) : \texttt{ref}(r') \dashv \Delta_n \cup \Delta'; \sigma_1\Pi''_0, \Pi'}$$

METHOD

$$\frac{\begin{array}{c} \Delta_1; C(r_{\text{this}}, r^*_g), \Pi_1 \vdash^{p_{\text{holding}}}_{r_{\text{holding}}} e : \texttt{ref}(\rho) \dashv \Delta'; \sigma_2(\Pi_2) \\ \{r_{x_1}, \ldots, r_{x_n}, r_{\text{this}}, r_{\text{holding}}, p_{\text{holding}}\} \subseteq \Delta_1 \\ \Delta' \cap \Delta_2 = \emptyset \\ \sigma_2 : \Delta_2 \to \Delta' \quad r_{\text{ret}} \in \Delta_2 \quad \sigma_2(r_{\text{ret}}) = \rho \end{array}}{\vdash \Delta_1; \Pi_1 \xrightarrow[r_{\text{holding}}]{p_{\text{holding}}} \Delta_2; \Pi_2 \text{ is the type for } mn(x^*)\{e\} \text{in class } C}$$

Figure 9. Permission type rules.

[2] A. Greenhouse and W. L. Scherlis, "Assuring and evolving concurrent programs: annotations and policy," in *ICSE '02*, 2002, pp. 453–463.

[3] D. Engler and K. Ashcraft, "Racerx: effective, static detection of race conditions and deadlocks," in *SOSP '03*, 2003, pp. 237–252.

[4] Y. Yu, T. Rodeheffer, and W. Chen, "Racetrack: efficient detection of data race conditions via adaptive tracking," in *SOSP '05*, 2005, pp. 221–234.

[5] C. Flanagan and S. N. Freund, "Types-based race detection for Java," in *PLDI '00*. ACM Press, 2000, pp. 219–232.

[6] C. Boyapati and M. Rinard, "A parameterized type system for race-free Java programs," *SIGPLAN Not.*, vol. 36, no. 11, pp. 56–69, 2001.

[7] C. Boyapati, R. Lee, and M. Rinard, "Ownership types for safe programming: preventing data races and deadlocks," in *OOPSLA '02*. ACM Press, Nov. 2002, pp. 211–230.

[8] N. Kobayashi, "Type systems for concurrent programs," in *Proceedings of UNU/IIST 10th Anniversary Colloquium, LNCS 2757*. Springer, 2002, pp. 439–453.

[9] J. Boyland and W. Retert, "Connecting effects and uniqueness with adoption," in *POPL '05*. New York, NY, USA: ACM Press, 2005, pp. 283–295.

[10] J. Boyland, "Checking interference with fractional permissions," in *SAS '03*, 2003, pp. 55–72.

[11] Y. Zhao and J. Boyland, "A fundamental permission interpretation for ownership types," in *TASE '08*. IEEE Computer Society, June 2008, pp. 65–72.

[12] C. Flanagan and M. Abadi, "Types for safe locking," in *ESOP '99*, Mar. 1999.

[13] A. Greenhouse, "A programmer-oriented approach to safe concurrency," Ph.D. dissertation, CMU, 2003.

TR-SUBST

$$\Pi \oplus r = \rho \equiv [r \mapsto \rho]\Pi \oplus r = \rho$$

TR-DUPLICATE

$$\Gamma \equiv \Gamma \oplus \Gamma$$

TR-SUBBAG

$$\frac{\Pi_1 \Rrightarrow \Pi'_1}{\Pi_1 \oplus \Pi_2 \Rrightarrow \Pi'_1 \oplus \Pi_2}$$

TR-BAGCOMM

$$\Pi_1 \oplus \Pi_2 \equiv \Pi_2 \oplus \Pi_1$$

TR-TRANS

$$\frac{\Pi \Rrightarrow \Pi' \quad \Pi' \Rrightarrow \Pi''}{\Pi \Rrightarrow \Pi''}$$

TR-IDENT

$$\Pi \equiv \Pi$$

TR-DROP

$$\Pi \Rrightarrow \emptyset$$

TR-TRUE

$$\texttt{true} \equiv \emptyset$$

TR-CONDTRUE

$$\Gamma \oplus \Gamma \Rightarrow \Pi \equiv \Gamma \oplus \Pi$$

TR-CONDFALSE

$$\neg\Gamma \oplus \Gamma \Rightarrow \Pi \equiv \neg\Gamma$$

TR-SPLIT

$$\Pi \equiv \texttt{1/2}\Pi \oplus \texttt{1/2}\Pi$$

TR-EMPTY

$$\Pi \equiv \Pi \oplus \emptyset$$

TR-CONDSPLIT

$$\Gamma \Rightarrow (\Pi_1 \oplus \Pi_2) \equiv \Gamma \Rightarrow \Pi_1 \Gamma \Rightarrow \Pi_2$$

TR-CONDNEST

$$\Gamma \Rightarrow (\Gamma' \Rightarrow \Pi) \equiv (\Gamma \wedge \Gamma') \Rightarrow \Pi$$

TR-FRACEMPTY

$$\xi\emptyset \equiv \emptyset$$

TR-FRACFACT

$$\xi\Gamma \equiv \Gamma$$

TR-FRACCOND

$$\xi(\Pi \dashrightarrow \Pi') \equiv (\xi\Pi) \dashrightarrow (\xi\Pi')$$

TR-FRACCOMB

$$\xi(\Pi_1 \oplus \Pi_2) \equiv \xi\Pi_1 \oplus \xi\Pi_2$$

TR-ZERODIM

$$\Omega^0(\ldots) \equiv \emptyset$$

TR-FRACBASE

$$\xi(\xi' k : \tau) \equiv (\xi\xi')k : \tau$$

TR-PACK

$$\xi k : \texttt{ref}(\rho) \oplus [r \mapsto \rho]\Pi \Rrightarrow \xi k : \exists r\,.\,(\Pi)$$

TR-UNPACK

$$\frac{r' \text{ fresh } \Pi' = \xi k : \texttt{ref}(r') \oplus [r \mapsto r']\Pi}{\Delta; \xi k : \exists r\,.\,(\Pi) \Rrightarrow \{r'\} \cup \Delta; \Pi'}$$

TR-ORDERNULL

$$\Omega^d(\rho_1 < \rho_2) \Rrightarrow \Omega^{d+1}(\$0 < \rho_2)$$

Figure 10. Transformation rules.

CP-FRAC

$$\frac{\vdash \xi \Downarrow q \quad h; \Psi \models^{(\mathbb{A},\mathbb{D})}_{\mu} \Pi}{qh; \xi\Psi \models^{(\mathbb{A},\mathbb{D})}_{\mu} \xi\Pi}$$

CP-TRUEIMP

$$\frac{\mathbb{A} \vdash \Gamma \Downarrow \text{true} \quad h; \Psi \models^{(\mathbb{A},\mathbb{D})}_{\mu} \Pi}{h; \Psi \models^{(\mathbb{A},\mathbb{D})}_{\mu} \Gamma \multimap \Pi}$$

CP-FALSEIMP

$$\frac{\mathbb{A} \vdash \Gamma \Downarrow \text{false}}{\hat{\emptyset}; \Psi \models^{(\mathbb{A},\mathbb{D})}_{\mu} \Gamma \multimap \Pi}$$

CP-IMP

$$\frac{h; \Psi', \Psi \models^{(\mathbb{A},\mathbb{D})}_{\mu} \Pi}{h; \Psi' \models^{(\mathbb{A},\mathbb{D})}_{\mu} \Psi \multimap \Pi}$$

CP-WRAPPEDHOLD

$$\frac{\mu(o, \_m) \neq \$0}{\hat{\emptyset}; \emptyset \models^{(\mathbb{A},\mathbb{D})}_{\mu} \Omega^d(o)}$$

CP-SHAREDFACT

$$\frac{\vdash d \Downarrow n \quad n > 0 \quad o < o' \in \mathbb{D}}{\hat{\emptyset}; \emptyset \models^{(\mathbb{A},\mathbb{D})}_{\mu} \Omega^d(o < o')}$$

CP-COMBINE

$$\frac{h_i; \Psi_i \models^{(\mathbb{A},\mathbb{D})}_{\mu} \Pi_i}{h_1 \hat{+} h_2; \Psi_1, \Psi_2 \models^{(\mathbb{A},\mathbb{D})}_{\mu} \Pi_1, \Pi_2}$$

CP-FIELDUNPACK

$$\frac{\mu(l) = o' \;\; h; \Psi \models^{(\mathbb{A},\mathbb{D})}_{\mu} l : \texttt{ref}(o'), [r \mapsto o']\Pi}{h; \Psi \models^{(\mathbb{A},\mathbb{D})}_{\mu} l : \exists r\,.\,(\Pi)}$$

CP-FIELD

$$\frac{\mu(l) = o' \quad h = \{[l \mapsto (1, o')]\}}{h; \emptyset \models^{(\mathbb{A},\mathbb{D})}_{\mu} l : \texttt{ref}(o')}$$

CP-TRUE

$$\frac{\mathbb{A} \vdash \Gamma \Downarrow \text{true}}{\hat{\emptyset}; \emptyset \models^{(\mathbb{A},\mathbb{D})}_{\mu} \Gamma}$$

CP-ADOPTER

$$\frac{\Psi = \sum_{(l_i:\tau_i \prec l) \in \mathbb{A}} \Psi_i \quad h_i; \Psi_i \models^{(\mathbb{A},\mathbb{D})}_{\mu} l_i : \tau_i}{(\hat{\sum}_{(l_i:\tau_i \prec l) \in \mathbb{A}} h_i) \hat{+} \{[l \mapsto (1, \$0)]\}; \Psi \models^{(\mathbb{A},\mathbb{D})}_{\mu} l : \tau_{\$0}}$$

CP-EMPTY

$$\hat{\emptyset}; \emptyset \models^{(\mathbb{A},\mathbb{D})}_{\mu} \emptyset$$

CP-WRAPPEDFREE

$$\frac{\mu(o, \_m) = \$0 \quad \vdash d \Downarrow n \quad n > 0 \quad h; \emptyset \models^{(\mathbb{A},\mathbb{D})}_{\mu} o.\texttt{Prot} : \tau_{\$0}}{h; \emptyset \models^{(\mathbb{A},\mathbb{D})}_{\mu} \Omega^d(o)}$$

Figure 11. Flattening rules: $h; \Psi \models^{(\mathbb{A},\mathbb{D})}_{\mu} \Pi$.

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 is currently 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, China. His current research interests include distributed system, network security and e-commerce.