Interference Verification for Structural Parallel Programs

Yang Zhao
School of Computer Science, Nanjing University of Science & Technology, Nanjing, China
Email: yangzhao@mail.njust.edu.cn

Abstract— **This paper presents a simplified "fractional permission" type system for a toy imperative language with structured parallelism and synchronization. In this system, we make the permission as a kind of linear values associated with some piece of state in a program. Different permissions permit different operations and "fractions" are attached to permissions to distinguish reads from writes, with which we are able to check interference among parallel threads. The main purpose is to detect race conditions and deadlocks in a multithreaded program. Program expressions and statements are transformed into action traces, then we permission-check all possible interleavings among traces in parallel. Operational semantics as well as permission type rules are provided and a type soundness result is then established.**

Index Terms— **fractional permission, structural parallelism, action trace**

I. INTRODUCTION

Data race and deadlock are two common programming errors in parallel programs. A data race happens when two or more concurrent threads sharing state try to access the same data simultaneously and at least one of them is a write operation. This may cause unexpected results or runtime errors. Suppose two threads are trying to access an object through the same pointer, one dereferences some field of the object, while the other attempts to deallocate that object concurrently. Assuming that the deallocation operation happens to win the competition and be executed first, then the dangling pointer error appears and a null pointer exception will be throwed out when the dereference action happens.

Deadlock occurs when multiple concurrent threads are holding some resources but waiting for more which are held by some others. It is possible that none of them can make progress. For example, thread t_1 holds resource r_1 waiting for r_2, thread t_2 holds resource r_2 waiting for r_3, ..., and thread t_n holds resoource r_n waiting for r_1. Clearly, a cyclic waiting chain is formed. All threads are blocked and the whole program then gets stuck. Because of many unexpected or nondeterministic interactions among different threads in parallel, analysis or verification for concurrent programs is usually very hard. Here, we give two code examples to show these problems.

Example 1: Assuming there exists an account object pointed to by a variable `v` and its balance is $100 currently, then two threads are trying to deposit some money at the same time (we use $e_1||e_2$ to show that two expressions are executed concurrently), written as:

```
*v=*v+10 || *v=*v+20
```

This code segment may exhibit unexpected behaviors. In particular, the left side thread may load the account value from memory (`*v`) and add a constant `10`, then just before stores the sum value to memory (`*v`) it yields to the right side thread to finish the whole deposit process. Thereafter, the left side continues to finish the store operation. Based on this execution order, the final balance of the account is then $110, which only reflects one deposit operation. Actually, the date race between these two store operations on different branches causes some unexpected result.

This race problem could be fixed easily by using synchronization:

```
synch v do {*v=*v+10}
||synch v do {*v=*v+20}
```

Either thread attempting to execute the deposit process has to obtain the lock of account in the first place. Since this lock is mutually exclusive, only one thread is able to hold the lock at any time. In fact, the synchronization ensures that two deposit operations are not interrupted. In other words, they are atomic.

Example 2: Two fund transfers between two accounts happen to be executed at the same time, but in a different locking order for a checking account (`c`) and a saving account (`s`), such as

```
synch c do {synch s do ...}
|| synch s do {synch c do ...}
```

This code segment shows race free, but may cause a deadlock: after the left side thread locks `c`, it yields to the right side thread to lock `s`. Then when the right side thread continues to lock `c`, it must fail, since `c` is being held by the left side thread who is currently expecting `s`.

This is a revised and extended version of the conference paper [1] that appeared in the Proceedings of ICACTE 2008.

This work was supported in part by NSFC grants 61103002, NJUST Research Funding 2010GJPY016 and SRF for ROCS of SEM.

doi:10.4304/jsw.7.8.1889-1896

Neither of them gives up, and thus could execute further more, therefore the whole program gets stuck.

There are quite a lot of previous work addressed these problems by devising type systems [2]–[6] and other static [7]–[9] or dynamic checking tools [10]. Brookes introduced a novel method to detect data races in pointer-free parallel programs [7] by converting every *command* into action traces and using the separation logic to prove all the possible interleavings between parallel traces are race free. In order to analyze multithreaded programs with pointer aliases, we need to check interferences among parallel executions. Boyland introduced a fractional permission type system to check interference [11]. In this paper, we attempt to extend this permission system to detect the race conditions and deadlocks based on action traces.

The contributions of this paper beyond previously published work include:

- Transforming expressions and statements into action traces and modelling the parallel statement as traces in parallel;
- Combining the pointer alias and synchronization together to model flexible parallel programs;
- Permission type checking to detect race conditions and deadlocks in parallel programs.

In the following sections, we first describe a simplified permission system briefly. Section III gives the formal system for a toy language including operational semantics and some permission rules for sequential executions. In section IV, we describe the ideas of *actions* and *traces* and show how to transform expressions and statements into traces, then we address the race and deadlock detection among interleavings of parallel traces. Section V provides the consistency of this permission system. Section VI describes some further work and we conclude in section VII.

II. FRACTIONAL PERMISSIONS

Permissions are used as the semantic foundation for multiple program annotations described in previous work [11], [12]. We additionally attach "fractions" to indicate partial permission values that allow to distinguish reads from writes.

A *permission* is a linear (non-duplicable) value associated with some piece of state in a program [12]. Based on different permission values, different program operations are allowed to be executed. If some operation is not granted certain permission, it is recognized to be "invalid" operation which will cause program state to go wrong.

Every piece of state is associated with exactly one permission and then a permission seems like some kind of right to access the associated state. The syntax of permissions is given in below:

$$\begin{array}{llll}
\text{base permission} & \beta & ::= & v : \texttt{ptr}(\rho) \mid \rho \\
\text{fraction} & \xi & ::= & 1 \mid q \mid \xi\xi \mid z \\
\text{fact} & \Gamma & ::= & \rho\text{=}\rho' \mid \texttt{not}\ \Gamma \\
\text{single permission} & \pi & ::= & \xi\beta \mid \Gamma \\
\text{permission bag} & \Pi & ::= & \emptyset \mid \pi \mid \Pi, \Pi
\end{array}$$

We have two kinds of base permissions: for the pointer variable v and for the location variable ρ. Fraction constants represent the known fraction values between 0 and 1, and fraction variable z represents an unknown fraction value. A whole permission ($1\ \beta$ or β) allows a write access to certain variable in β, while the fractional permission ($\xi\beta$ with ξ known not to be 1) only allows read access. A single permission could be either a base permission or a predicate (Γ) which indicates a fact. In this paper, the fact we care about is whether two location variables are actually pointing to the same location in the memory. A permission bag contains nothing ($\emptyset$) or a set of base permissions combined by comma notations. In next sections, we usually call the permission bag as permissions instead.

III. FORMAL SYSTEM

This section describes a toy multithreaded language which is derived from previous work [11] with some additional concurrent features. We give the formal syntax, operational semantics and sequential permission checking rules.

A. Syntax

This toy language combines the shared-memory parallelism with pointer operations. The syntax is given in below:

$$\begin{array}{l}
P ::= D^*\ \ s \\
D ::= p(\overline{a})\ \texttt{requires}\ a^*\{s\} \\
s ::= \texttt{skip} \mid v\texttt{=new} \mid *v\texttt{=}e \mid \texttt{let}\ x\texttt{=}e\ \texttt{in}\ s \mid s\,;\,s \\
\qquad \mid \texttt{call}\ p(\overline{v}) \mid \texttt{if}\ b\ \texttt{then}\ s\ \texttt{else}\ s \\
\qquad \mid s \Rsh s \mid s || s \\
\qquad \mid \texttt{synch}\ v\ \texttt{do}\ s \mid \texttt{hold}\ v\ \texttt{in}\ s \\
e ::= n \mid x \mid e \pm e \mid *v \\
b ::= e\,!\!=\!n \mid v\texttt{==}v
\end{array}$$

A program consists several procedure definitions followed by a statement acting as the main body of this program. The definition of a procedure needs to provide a formal argument list $\overline{a}$ as well as the required lock sets a^*. Every required lock a is required to be included in $\overline{a}$. For simplicity, parallel statements are allowed to be used inside not the procedure body, but the program body.

Most of statements appear in the syntax are straightforward. One may allocate a cell using v`=new` and initialize the datum stored in this cell as 0 which could be updated by $*v = e$ later on. This language also contains *conditional* (`if` b `then` s `else` s'), *sequential composition* ($s\,;\,s'$) as well as a *procedure call* (`call` $p(\overline{v})$).

In order to model multithreaded features, we use the parallel statement ($s \Rsh s'$) to indicate that a new spawn thread to execute the s' in parallel with the execution of s. Moreover, the new thread cannot hold any locks originally. We use the $s \mid\mid s'$ as an internal representation for two statements in parallel. A similar case happens for `hold` v `in` s which is also internal form when evaluating a body of synchronization statement.

There are two primitive types used in our system: Integer and Boolean. Integer expressions include literals, local variables, arithmetic expressions and dereferences of pointer variables. Boolean expressions permit comparison with zero and pointer comparison as well.

B. Operational Semantics

There are a finite set of global variables V and an infinite set of memory locations L. The memory M maps variables to locations or locations to integers (**Z**):

$$(\mu_V, \mu_L) = \mu \in M = (V \rightarrow L) \times (L \rightarrow \mathbf{Z})$$

For simplicity purpose, we do not distinguish μ_V and μ_L and use μ instead. A thread may hold some locks for corresponding cells, so we list the cell locations for the held locks of each thread. Operational semantics is given in Figure 1 in terms of a small step evaluation. Evaluating an expression will update neither the memory nor the held locks, but evaluating a statement may do, such as creating a new cell, updating a cell content, acquiring a lock, and so on.

Figure 1 gives all evaluation rules, most of which are straightforward and have been mentioned in previous work [11]. Here, we only address several rules that are related to synchronization and parallel executions. Here, all the evaluation rules are based on single thread or two parallel threads which are easy to be extended to more parallel threads.

In fact, every object in Java is associated with a monitor field, that the context thread can *lock* or *unlock*. At most one thread at a time may hold a lock on the monitor of that object. Any other threads attempting to lock that monitor will be blocked until they can obtain a lock on that monitor sucessfully. In order to model the lock status of a cell, we require every cell implicitly associated with an additional address to simulate the "monitor" field used in Java-like language. We simply choose the next address of the cell to represent whether the location is locked or not: a zero to represent the unlocked state, but a positive number to represent how many times the cell has been locked by one thread. Therefore, every cell will actually contain two continuous addresses in memory: l (for the cell content) and $l+1$ (for the lock value).

Rule E-SYNC provides the synchronization capability: it is required to acquire the lock for the cell v before evaluating the body. There are three possibilities for v:

- in the *unlocked* state;
- in the *locked* state, but being held by the current thread;
- in the *locked* state, but being held by the some other thread.

We increase the lock value by 1 for the first and second cases, but insert $\mu(v)$ to the held locks of the current thread only in the first case. Thereafter, we evaluate the body using rule E-HOLD which indicates v has already been held before entering the body s. When finishing evaluating the body, the lock value must be decreased by 1 and if it becomes 0, we remove $\mu(v)$ from the current held locks using rule E-RELEASE.

Rule E-PARALLEL0 is used when starting to evaluate a parallel statement $s_1 \Rsh s_2$ by creating a parallel locks $\overline{l_1}||\cdot$. Since the new spawn thread does not inherit any locks originally, the rule then indicates this scenario using an empty lock sequence $\cdot$. Thereafter we use the parallel locks to evaluate two statements in parallel $s_1 \;||\; s_2$. Rules E-PARALLEL1 and E-PARALLEL2 will be applied nondeterministically: either branch could make progress. Thus, the parallel composition can be eliminated once both branches are done.

C. Permission Type Rules

Permissions are basically some kind of special type in a program, thus we borrow typical type checking methods to handle permission checking. Sequential permission rules are given in Figure 2, where the basic judgments are:

$$E \vdash s \dashv \left\{\overline{E'}\right\} \text{ or } E \vdash e : \tau \text{ where } E = \Delta; \Pi; \Phi; \hat{\Phi}$$

A permission environment E has four parts: a type context Δ which is a set of variables drawn from pointer variables, local integer variables and fraction variables; some permissions Π owned by the current thread; a lock sequence Φ and a lock set $\hat{\Phi}$. The Φ is used to represent the locks being held by the current thread, while $\hat{\Phi}$ is a set which indicates the locks being held in the global system. Thus, the former is just a local property for current thread, while the latter keeps the global lock information. Moreover, the Φ allows duplicated locks, but the $\hat{\Phi}$ does not.

The execution any expression certainly does not change the permission environment, but the treatment for statements is opposite. A statement may be executed nondeterministically, so we have to keep all possible output environments $\left\{\overline{E'}\right\}$. The nondeterminism obviously comes from the conditional and parallel statements inside. Based upon the *one-in-multiple-out* permission checking idea, we also borrow a syntactic sugar for *multiple-in-multiple-out* judgment:

$$\frac{\forall i \in [1..m] : E_i \vdash s \dashv \left\{\Delta_{i,j}; \Pi_{i,j}; \Phi_{i,j}; \hat{\Phi}_{i,j} \mid_{j=1}^{n_i}\right\}}{\{E_i \mid_{i=1}^{m}\} \vdash s \dashv \left\{\Delta_{i,j}; \Pi_{i,j}; \Phi_{i,j}; \hat{\Phi}_{i,j} \mid_{i=1,j=1}^{m,n_i}\right\}}$$

Therefore, we can chain two sequence executions s_1 and s_2 together:

$$E \vdash s_1 \dashv \left\{\overline{E'}\right\} \vdash s_2 \dashv \left\{\overline{E''}\right\}$$

Most rules have been explained in previous work [11], while several rules bear some explanation. Within rule READ, we must have at least some fractional permission for the pointer variable to be dereferenced as well as some other fractional permission for the cell content to be read. A similar consideration occurs for rule WRITE except that we require the whole permission for the cell content to be updated which is represented as 1ρ assuming ρ is the location where the pointer v is pointing to.

E-NEW
$$\frac{l', l'+1 \notin \text{Dom}(\mu_L) \qquad \mu' = \mu[v \mapsto l',\ l' \mapsto 0,\ (l'+1) \mapsto 0]}{((\mu;\bar{l}); v\texttt{=new}) \rightarrow ((\mu';\bar{l}); \texttt{skip})}$$

E-READ
$$((\mu;\bar{l}); *v) \rightarrow ((\mu;\bar{l}); \mu(\mu(v)))$$

E-LET1
$$\frac{((\mu;\bar{l}); e) \rightarrow ((\mu;\bar{l}); e')}{((\mu;\bar{l}); \texttt{let}\ x\texttt{=}e\ \texttt{in}\ s) \rightarrow ((\mu;\bar{l}); \texttt{let}\ x\texttt{=}e'\ \texttt{in}\ s)}$$

E-LET
$$\frac{x \notin \mu_V}{((\mu;\bar{l}); \texttt{let}\ x\texttt{=}i\ \texttt{in}\ s) \rightarrow ((\mu;\bar{l}); [x \mapsto i]s)}$$

E-WRITE1
$$\frac{((\mu;\bar{l}); e) \rightarrow ((\mu;\bar{l}); e')}{((\mu;\bar{l}); *v\texttt{=}e) \rightarrow ((\mu;\bar{l}); *v\texttt{=}e')}$$

E-WRITE
$$\frac{\mu' = \mu[(\mu v) \mapsto i]}{((\mu;\bar{l}); *v\texttt{=}i) \rightarrow ((\mu';\bar{l}); \texttt{skip})}$$

E-SEQ1
$$\frac{((\mu;\bar{l}); s_1) \rightarrow ((\mu';\overline{l'})); s'_1)}{((\mu;\bar{l}); s_1\texttt{;}\, s_2) \rightarrow ((\mu';\overline{l'}); s'_1\texttt{;}\, s_2)}$$

E-SEQ
$$((\mu;\bar{l}); \texttt{skip;}\, s) \rightarrow ((\mu;\bar{l}); s)$$

E-IF
$$\frac{((\mu;\bar{l}); b) \rightarrow ((\mu;\bar{l}); b')}{((\mu;\bar{l}); \texttt{if}\ b\ \texttt{then}\ s_1\ \texttt{else}\ s_2) \rightarrow ((\mu;\bar{l}); \texttt{if}\ b'\ \texttt{then}\ s_1\ \texttt{else}\ s_2)}$$

E-CALL
$$\frac{\text{Pbody}(p) = s \qquad \text{Formals}(p) = \overline{a}}{((\mu;\bar{l}); \texttt{call}\ p(\overline{v})) \rightarrow ((\mu[a_i \mapsto v_i];\bar{l}); s)}$$

E-PARALLEL0
$$((\mu;\bar{l}); s_1 \Rsh s_2) \rightarrow ((\mu;\bar{l}||\cdot); s_1 \mid\mid s'_2)$$

E-PARALLEL1
$$\frac{((\mu;\overline{l_1}); s_1) \rightarrow ((\mu';\overline{l'_1}); s'_1)}{((\mu;(\overline{l_1}||\overline{l_2})); s_1 \mid\mid s_2) \rightarrow ((\mu';\overline{l'_1}||\overline{l_2}); s'_1 \mid\mid s_2)}$$

E-PARALLEL2
$$\frac{((\mu;\overline{l_2}); s_2) \rightarrow ((\mu';\overline{l'_2}); s'_2)}{((\mu;(\overline{l_1}||\overline{l_2})); s_1 \mid\mid s_2) \rightarrow ((\mu';(\overline{l_1}||\overline{l'_2})); s_1 \mid\mid s'_2)}$$

E-PARALLEL
$$((\mu;(\overline{l_1}||\cdot)); \texttt{skip} \mid\mid \texttt{skip}) \rightarrow ((\mu;\overline{l_1}); \texttt{skip})$$

E-SYNC
$$\frac{(\mu(\mu(v)+1) = 0 \vee \mu(v) \in \bar{l}) \qquad \mu' = \mu[(\mu(v)+1) \mapsto (\mu(\mu(v)+1)) + 1]}{((\mu;\bar{l}); \texttt{synch}\ v\ \texttt{do}\ s) \rightarrow ((\mu';\bar{l},\mu(v)); \texttt{hold}\ v\ \texttt{in}\ s)}$$

E-HOLD
$$\frac{((\mu;\bar{l}); s) \rightarrow ((\mu';\overline{l'}); s')}{((\mu;\bar{l}); \texttt{hold}\ v\ \texttt{in}\ s) \rightarrow ((\mu';\overline{l'}); \texttt{hold}\ v\ \texttt{in}\ s')}$$

E-RELEASE
$$\frac{\mu(v) = l_1 \qquad \mu' = \mu[(l_1+1) \mapsto (\mu(l_1+1) - 1)]}{((\mu;\bar{l},l_1); \texttt{hold}\ v\ \texttt{in}\ \texttt{skip}) \rightarrow ((\mu';\bar{l}); \texttt{skip})}$$

Figure 1. Operational semantics.

INTLITERAL
$$E \vdash n : \texttt{int}$$

VARIABLE
$$\frac{E = \Delta;\Pi;\Phi;\hat{\Phi} \qquad x \in \Delta}{E \vdash x : \texttt{int}}$$

ARITHMATIC
$$\frac{E \vdash e_1 : \texttt{int} \qquad E \vdash e_2 : \texttt{int}}{E \vdash e_1 \pm e_2 : \texttt{int}}$$

NONZERO
$$\frac{E \vdash e : \texttt{int}}{E \vdash e\texttt{!=}0 : \texttt{bool}}$$

SKIP
$$E \vdash \texttt{skip} \dashv \{E\}$$

NEW
$$\frac{E = \Delta; 1v : \texttt{ptr}(\rho'), \Pi; \Phi; \hat{\Phi} \qquad \rho \text{ fresh}}{E \vdash v\texttt{=new} \dashv \left\{\{\rho\} \cup \Delta; 1v : \texttt{ptr}(\rho), 1\rho, \Pi; \Phi; \hat{\Phi}\right\}}$$

ALIAS
$$\frac{E = \Delta;\Pi;\Phi;\hat{\Phi} \qquad \Pi = 1v : \texttt{ptr}(\rho), \xi v' : \texttt{ptr}(\rho'), \Pi'}{E \vdash v\texttt{=}v' \dashv \left\{\Delta; 1v : \texttt{ptr}(\rho'), \xi v' : \texttt{ptr}(\rho'), \Pi'; \Phi; \hat{\Phi}\right\}}$$

READ
$$\frac{E = \Delta;\Pi;\Phi;\hat{\Phi} \qquad \Pi = \xi v : \texttt{ptr}(\rho), \xi'\rho, \Pi'}{E \vdash *v : \texttt{int}}$$

WRITE
$$\frac{E = \Delta;\Pi;\Phi;\hat{\Phi} \qquad E \vdash e : \texttt{int} \qquad \Pi = \xi v : \texttt{ptr}(\rho), 1\rho, \Pi'}{E \vdash *v\texttt{=}e \dashv \{E\}}$$

PARALLEL
$$\frac{\Delta;\Pi;(\Phi \mid\mid \cdot);\hat{\Phi} \vdash s_1 \mid\mid s_2 \dashv \left\{\overline{\Delta';\Pi';\Phi';\hat{\Phi}'}\right\}}{\Delta;\Pi;\Phi;\hat{\Phi} \vdash s_1 \Rsh s_2 \dashv \left\{\overline{\Delta';\Pi';\Phi';\hat{\Phi}'}\right\}}$$

SEQ
$$\frac{E \vdash s_1 \dashv \{\overline{E'}\} \vdash s_2 \dashv \{\overline{E''}\}}{E \vdash s_1\texttt{;}\, s_2 \dashv \{\overline{E''}\}}$$

LOCAL
$$\frac{E = \Delta;\Pi;\Phi;\hat{\Phi} \qquad E \vdash e : \texttt{int} \qquad x \notin \Delta \qquad \{x\} \cup \Delta;\Pi;\Phi;\hat{\Phi} \vdash s \dashv \left\{\overline{\Delta';\Pi';\Phi';\hat{\Phi}'}\right\}}{E \vdash \texttt{let}\ x\texttt{=}e\ \texttt{in}\ s \dashv \left\{\overline{\Delta \setminus \{x\};\Pi';\Phi';\hat{\Phi}'}\right\}}$$

IF1
$$\frac{E \vdash e : \texttt{int} \qquad E \vdash s_1 \dashv \{\overline{E'}\} \qquad E \vdash s_2 \dashv \{\overline{E''}\}}{E \vdash \texttt{if}\ e\texttt{!=}0\ \texttt{then}\ s_1\ \texttt{else}\ s_2 \dashv \{\overline{E'}\} \cup \{\overline{E''}\}}$$

IF2
$$\frac{\Pi = \xi v : \texttt{ptr}(\rho), \xi' v' : \texttt{ptr}(\rho'), \Pi_1 \qquad \Delta; \rho = \rho', \Pi; \Phi; \hat{\Phi} \vdash s_1 \dashv \{\overline{E'}\} \qquad \Delta; \rho \neq \rho', \Pi; \Phi; \hat{\Phi} \vdash s_2 \dashv \{\overline{E''}\}}{\Delta;\Pi;\Phi;\hat{\Phi} \vdash \texttt{if}\ v\texttt{==}v'\ \texttt{then}\ s_1\ \texttt{else}\ s_2 \dashv \{\overline{E'}\} \cup \{\overline{E''}\}}$$

CALL
$$\frac{\begin{array}{c}\text{Formals}(p) = \overline{a} \\ |\overline{a}| = |\overline{v}| = m \qquad \forall i \in [1..m], \exists \Pi'_1 : \sigma_1\Pi_1 = \xi_i v_i : \texttt{ptr}(\rho_i), \Pi'_1 \qquad \forall a_i \in \text{Requires}(p) : \rho_i \in \Phi \\ p : \Delta_1;\Pi_1 \rightarrow \Delta_2;\sigma_2\Pi_3 \qquad \Delta_1 \supseteq \{\overline{a}\} \qquad \vdash \sigma_1 : \Delta_1 \rightarrow \Delta \vdash \sigma_2 : \Delta_3 \rightarrow \Delta_2 \qquad \Delta_3 \text{ fresh}\end{array}}{\Delta;\sigma_1\Pi_1,\Pi;\Phi;\hat{\Phi} \vdash \texttt{call}\ p(\overline{v}) \dashv \left\{\Delta \cup \Delta_3; \sigma_1\Pi_3, \Pi; \Phi; \hat{\Phi}\right\}}$$

PROCEDURE
$$\frac{\begin{array}{c}\overline{a} = \text{Formals}(p) \qquad |\overline{a}| = m \qquad \forall i \in [1..m] : \Pi_1 = \xi_i a_i : \texttt{ptr}(\rho_i), \Pi_i \qquad \Phi = \overline{\rho_i} | i \in [1..m] \\ \hat{\Phi} = \{\rho_i | i \in [1..m]\} \qquad \Delta_1;\Pi_1;\Phi;\hat{\Phi} \vdash \text{Body}(p) \dashv \left\{\overline{\Delta;\sigma\Pi_2;\Phi;\hat{\Phi}}\right\} \qquad \overline{\Delta \cap \Delta_2 = \emptyset} \qquad \overline{\sigma : \Delta_2 \rightarrow \Delta}\end{array}}{\vdash p : \Delta_1;\Pi_1 \rightarrow \Delta_2;\Pi_2}$$

Figure 2. Permission type rules.

In rule CALL which applies to a statement of procedure call, we adopt a procedure type $\Delta_1; \Pi_1$ `->` $\Delta_2; \Pi_2$. The output permissions Π_2 will use the (existentially quantified) variables in Δ_2 as well as the (universally quantified) variables in Δ_1. The first context Δ_1 will include all formal parameters $\overline{a}$. For each procedure call, we borrow σ_1 to map from Δ_1 to actual parameters $\overline{v}$. The new created variables set Δ_3 is used to substitute for the existentially quantified variables (in Δ_2). Therefore, permission checking a procedure call requires to:

- make sure the required locks have been held;
- use σ_1 to map formals to actuals;
- split incoming permission into two parts: $\sigma_1 \Pi_1$ and Π, such that the latter will not be affected by this call;
- fetch the procedure type and find a σ_2 which will map from new created fresh variable set into the result variable set Δ_2.

The other important rule is PARALLEL. Here we assume that the new spawn child thread does not inherit any lock originally, thus the initial lock sequence for the child thread is empty. Then we use parallel locks Φ and $\cdot$ to permission check the parallel statements s_1 and s_2. Because of the possible aliasing problem, these two parallel statements may interleave with each other. The general idea presented in this paper is to convert the complicated statements and expressions into action traces, then permission check all possible interleavings of parallel traces.

IV. PARALLEL CHECKING

Most rules in Figure 2 work for sequential executions. For parallel programs, we need to consider whether or not the current thread may be affected by some others. Since the runtime execution is unpredictable for multithreaded programs, we have to check all the possibilities within the permission system. In order to model actual program behaviors, we convert statements and expressions into a sequence of atomic actions which is supposed to be executed without interruption. Furthermore, we transform the parallel statements into the parallel traces and then permission check their possible interleavings to make sure that data races and deadlocks can never happen.

A. Action and Trace

The behaviors of a program can be described in terms of atomic actions [7], for instance, allocating a new cell, dereferencing a pointer, updating a cell's content and so on. However, executions in the current thread may be interrupted by other parallel threads. Let's take an assignment statement $*v = *v + 10$ for example, we do not know whether or how often this statement will be interrupted, but we do know all the positions where interruptions may happen. Assuming the read, write and plus are atomic operations, the possible interruption positioins could be:

- before the execution of this statement;
- after loading the content $*v$;
- after reading the constant 10;
- after the plus operation;
- after the update of the content $*v$.

To analyze the possible interleavings in permission checking, we define two concepts: *actions* and *traces*.

Definition 1: An action is an atomic execution that cannot be interrupted in any actual parallel execution together with other threads. In this paper, they include

- an *idle* action: `skip`;
- *integer* actions: n, x, $\pm$;
- *boolean* actions: $v == v'$, $v! = v'$
- *heap* actions: v=`new`, $*v$, $*v =$;
- a *store* action: $v = v'$;
- *lock* actions: `try`(v), `rel`(v);
- a *fork* action: `fork`.

Every expression or statement can be converted into one or more sequences of actions.

We borrow λ to range over the set of actions, and then a trace is defined in below.

Definition 2: A trace is a group of actions defined as:

$$\iota ::= \lambda \ \mid \ \iota \sharp \iota \ \mid \ \iota || \iota$$

The concatenation $\sharp$ indicates a deterministic trace in which the latter trace must follow the former. The parallel $||$ indicates a trace is constructed by two traces in parallel.

We use $\langle e \rangle$ and $\langle s \rangle$ to represent the set of traces for the expression e and statement s respectively. Here, we build an approximation: a procedure call is considered as an action with two requirements:

- no parallel statements are in the procedure body
- all the parameters are unchangeable.

It is provable that a call to a well-formed procedure satisfying these two requirements can be considered as a action if its required lock set is a subset of the held locks for the current thread.

All transformation rules are given in Figure 3. The difference between TRACE-IFNONZERO and TRACE-IFPOINTEREQ is that the trace for non-zero comparation needs to be added to both branches equally, while the different results of the pointer comparation will be added into different branches correspondingly.

B. Permission Checking for Parallel Traces

In order to check parallel statements, we need to have permission checking rules for actions. Type rules in Figure 2 do not work for actions (although some of them are very similar). Some rules for actions are then given in Figure 4.

There are two rules (TRYSUCCESSFUL and TRYFAILED) for the `try`(v) action, corresponding to whether or not the lock is acquired successfully. In RELEASE1 and RELEASE2, we distinguish the cases that the lock ρ_1 has been held only once and more than one times respectively. In the latter case, the `rel`(v) action removes the lock from not $\hat{\Phi}$, but Φ (since the current thread is still holding this lock somewhere).

Trace-Const

$$\langle n\rangle = \{n\}$$

Trace-Variable

$$\langle x\rangle = \{x\}$$

Trace-Arithmetic

$$\frac{\iota \in \langle e\rangle \qquad \iota' \in \langle e'\rangle}{\langle e \pm e'\rangle = \{\iota\sharp\iota'\sharp\pm\}}$$

Trace-Read

$$\langle *v\rangle = \{*v\}$$

Trace-Skip

$$\langle \texttt{skip}\rangle = \{\texttt{skip}\}$$

Trace-New

$$\langle v\texttt{=new}\rangle = \{v\texttt{=new}\}$$

Trace-Seq

$$\frac{\iota_1 \in \langle e_1\rangle \qquad \iota_2 \in \langle e_2\rangle}{\langle e_1 ; e_2\rangle = \{\iota_1\sharp\iota_2\}}$$

Trace-IfNonZero

$$\frac{\iota \in \langle e\rangle \qquad \iota_1 \in \langle s_1\rangle \qquad \iota_2 \in \langle s_2\rangle}{\langle \texttt{if } e\texttt{!=}0 \texttt{ then } s_1 \texttt{ else } s_2\rangle = \{\iota\sharp\iota_1\} \cup \{\iota\sharp\iota_2\}}$$

Trace-Local

$$\frac{\iota \in \langle e\rangle \qquad \iota' \in \langle s\rangle}{\langle \texttt{let } x\texttt{=}e \texttt{ in } s\rangle = \{\iota\sharp\iota'\}}$$

Trace-Write

$$\frac{\iota \in \langle e\rangle}{\langle *v\texttt{=}e\rangle = \{\iota\sharp(*v\texttt{=})\}}$$

Trace-IfPointerEq

$$\frac{\iota_1 \in \langle s_1\rangle \qquad \iota_2 \in \langle s_2\rangle}{\langle \texttt{if } v\texttt{==}v' \texttt{ then } s_1 \texttt{ else } s_2\rangle = \{(v\texttt{==}v')\sharp\iota_1\} \cup \{(v\texttt{!=}v')\sharp\iota_2\}}$$

Trace-Sync

$$\frac{\iota \in \langle s\rangle}{\langle \texttt{synch } v \texttt{ do } s\rangle = \{\texttt{try}(v)\sharp\iota\sharp\texttt{rel}(v)\}}$$

Trace-Parallel

$$\frac{\iota_1 \in \langle s_1\rangle \qquad \iota_2 \in \langle s_2\rangle}{\langle s_1 \Rsh s_2\rangle = \{\texttt{fork}\sharp(\iota_1 \mid\mid \iota_2)\}}$$

Figure 3. Transformation rules

ConditionTrue

$$\frac{\Pi = \xi v : \texttt{ptr}(\rho), \xi' v' : \texttt{ptr}(\rho'), \Pi'}{\Delta; \Pi; \Phi; \hat{\Phi} \vdash v\texttt{==}v' \dashv \Delta; \rho = \rho', \Pi; \Phi; \hat{\Phi}}$$

ConditionFalse

$$\frac{\Pi = \xi v : \texttt{ptr}(\rho), \xi' v' : \texttt{ptr}(\rho'), \Pi'}{\Delta; \Pi; \Phi; \hat{\Phi} \vdash v\texttt{!=}v' \dashv \Delta; \text{not } \rho = \rho', \Pi; \Phi; \hat{\Phi}}$$

TrySuccessful

$$\frac{\Pi = \xi v : \texttt{ptr}(\rho), \Pi' \qquad \rho \notin \hat{\Phi}}{\Delta; \Pi; \Phi; \hat{\Phi} \vdash \texttt{try}(v) \dashv \Delta; \texttt{1/2}\xi v : \texttt{ptr}(\rho), \Pi'; \rho, \Phi; \{\rho\} \cup \hat{\Phi}}$$

TryFailed

$$\frac{\Pi = \xi v : \texttt{ptr}(\rho), \Pi' \qquad \rho \in \hat{\Phi}}{\Delta; \Pi; \Phi; \hat{\Phi} \vdash \texttt{try}(v) \dashv \text{failed}}$$

Fork

$$\Delta; \Pi; \Phi; \hat{\Phi} \vdash \texttt{fork} \dashv \Delta; \Pi; (\Phi \mid\mid \cdot); \hat{\Phi}$$

Release1

$$\frac{\Pi = \texttt{1/2}\xi v : \texttt{ptr}(\rho), \Pi' \qquad \rho \notin \Phi}{\Delta; \Pi; \rho, \Phi; \{\rho\} \cup \hat{\Phi} \vdash \texttt{rel}(v) \dashv \Delta; \xi v : \texttt{ptr}(\rho), \Pi'; \Phi; \hat{\Phi}}$$

Release2

$$\frac{\Pi = \xi v : \texttt{ptr}(\rho), \Pi' \qquad \rho \in \Phi}{\Delta; \Pi; \rho, \Phi; \hat{\Phi} \vdash \texttt{rel}(v) \dashv \Delta; 2\xi v : \texttt{ptr}(\rho), \Pi'; \Phi; \hat{\Phi}}$$

Figure 4. Permission type rules for actions.

Permission checking two parallel statements is then transformed into checking two traces in parallel $\iota_1 \mid\mid \iota_2$, which is written as $\lambda_1\sharp\iota_1'$ and $\lambda_2\sharp\iota_2'$ respectively. Assuming λ_i is neither $\texttt{try}(v)$ nor $\texttt{fork}$, we either check λ_1 before $\iota_1' \mid\mid \iota_2$; or check λ_2 before $\iota_1 \mid\mid \iota_2'$. After finishing all possibilities, all output environments are united as the entire output environment for checking $\iota_1 \mid\mid \iota_2$.

$$\frac{\begin{array}{c} \iota_1 = \lambda_1\sharp\iota_1' \qquad \iota_2 = \lambda_2\sharp\iota_2' \\ \Pi = \Pi_1, \Pi_2, \Pi_3 \qquad \Delta; \Pi_1; \Phi_1; \hat{\Phi} \vdash \lambda_1 \dashv \Delta'; \Pi'; \Phi'; \hat{\Phi}' \\ \Delta; \Pi_2; \Phi_2; \hat{\Phi} \vdash \lambda_2 \dashv \Delta''; \Pi''; \Phi''; \hat{\Phi}'' \\ \Delta'; \Pi', \Pi_2, \Pi_3; \Phi' \mid\mid \Phi_2; \hat{\Phi}' \vdash \iota_1' \mid\mid \iota_2 \dashv \left\{\overline{E_1}\right\} \\ \Delta''; \Pi'', \Pi_1, \Pi_3; \Phi_1 \mid\mid \Phi''; \hat{\Phi}'' \vdash \iota_1 \mid\mid \iota_2' \dashv \left\{\overline{E_2}\right\} \end{array}}{\Delta; \Pi; \Phi_1 \mid\mid \Phi_2; \hat{\Phi} \vdash \iota_1 \mid\mid \iota_2 \dashv \left\{\overline{E_1}\right\} \cup \left\{\overline{E_2}\right\}}$$

To check the interference, we split the input permissions Π into Π_1, Π_2, Π_3 and use Π_1 and Π_2 to check λ_1 and λ_2 respectively. If both actions can be checked under correspondent permissions, then there is no interference between these two actions.

There are several related lemmas.

Lemma 1: Two actions do not interfere with each other under the given environment E (written as $E[[\lambda_1 \asymp \lambda_2]]$), if $E = \Delta; \Pi; \Phi; \hat{\Phi}$ and Π can be transformed into (Π_1, Π_2, Π_3) and Π_i permission check λ_i independently (i=1,2).

Lemma 2: Two parallel traces do not interfere with each other under the given environment E (written as $E[[\iota_1 \asymp \iota_2]]$), if for any λ_1, λ_2, there exists ι_{11}, ι_{12}, ι_{21}, ι_{22} and $\overline{E'}$, such that $\iota_1 = \iota_{11}\sharp\lambda_1\sharp\iota_{12}$, $\iota_2 = \iota_{21}\sharp\lambda_2\sharp\iota_{22}$ and $E \vdash \iota_{11} \mid\mid \iota_{21} \dashv \left\{\overline{E'}\right\}$, $E'[[\lambda_1 \asymp \lambda_2]]$ for every $E' \in \left\{\overline{E'}\right\}$.

Lemma 3: Two statements in parallel $s_1 \mid\mid s_2$ is race free under the environment E (written as $E[[s_1 \asymp s_2]]$), if for any trace ι_1 and ι_2, such that $\iota_1 \in \langle s_1\rangle \wedge \iota_2 \in \langle s_3\rangle$, $E[[\iota_1 \asymp \iota_2]]$.

$$\frac{\begin{array}{c} \iota_i \in \langle s_1\rangle \qquad \iota_j' \in \langle s_2\rangle \\ \Delta; \Pi; \Phi_1 \mid\mid \Phi_2; \hat{\Phi} \vdash \iota_i \mid\mid \iota_j' \dashv \left\{\overline{\Delta_{i,j}; \Pi_{i,j}; \Phi_{i,j}; \hat{\Phi}_{i,j}}\right\} \end{array}}{\Delta; \Pi; \Phi_1 \mid\mid \Phi_2; \hat{\Phi} \vdash s_1 \mid\mid s_2 \dashv \bigcup_{i=1,j=1}^{|\langle s_1\rangle|,|\langle s_2\rangle|} \left\{\overline{\Delta_{i,j}; \Pi_{i,j}; \Phi_{i,j}; \hat{\Phi}_{i,j}}\right\}}$$

C. Deadlock Detection

Deadlocks may happen when several threads attempt to lock some resources being held by some other threads. For example:

```
synch v1 do {synch v2 do {...}}
|| synch v2 do {synch v3 do {...}}
|| synch v3 do {synch v1 do {...}}
```

It is possible that the main thread is holding the lock for v_1 and waiting for v_2, the child thread is holding the lock for v_2 and waiting for v_3 and the grandchild thread is holding the lock for v_3 and waiting for v_1. In this case, a deadlock happens and none of parallel executions can make any progress. It is easy to see that the deadlock condition can only happen between multiple $try(v)$ actions in parallel.

Lemma 4: A deadlock condition exists among some parallel actions $try(v_1)$, $try(v_2)$, ... $try(v_n)$ under the

given environment $E = (\Delta; \Pi; (\Phi_1||\Phi_2||...||\Phi_n); \hat{\Phi})$, if $\forall i \in [1..n] \exists j \in [1..n] : (i \neq j) \wedge (\Pi = \xi_i v_i : \texttt{ptr}(\rho_i), \Pi'_i) \wedge (\rho_i \in \Phi_j)$.
Detailed detection rules are omitted here.

V. CONSISTENCY

To ensure the soundness of permission type system, it is required that well-typed program can never go wrong. In this system, soundness means permission checked program will never have data races or deadlocks at runtime. To verify this two properties, we need to make the static permission type and dynamic runtime state match to each other according to its operational semantics and sequential/parallel permission type rules. This is the "consistency".

The consistency relation between a memory μ, a runtime thread information T and a permission environment $\Delta; \Pi; (\Phi_1||\Phi_2||...||\Phi_n); \hat{\Phi}$ is represented as:

$$\Delta; \Pi; (\Phi_1||\Phi_2||...||\Phi_n); \hat{\Phi} \vdash \mu; T \text{ ok}$$

The threads T is a sequence of held locks:

$$T ::= \overline{l_1}||\overline{l_2}||...||\overline{l_n}$$

In order to match the static type to runtime state, it is required to:

- map the pointer variable in permission type to absolute address in memory
- match parallel locks in environment to actual parallel threads
- match the global lock set $\hat{\Phi}$ to the actual lock status in memory

A memory μ only includes values of pointer variables, but the type system has location variables and fraction variables as well. We use a partial function ψ_1 to map from fractions into numbers between zero and ψ_2 to map from location variables into actual locations in the memory:

$$\psi_1 : F \rightharpoonup (0, 1] \qquad \psi_2 : R \rightharpoonup L$$

For fractions, we have:

$$\psi_1(1) = 1 \qquad \psi_1(q) = q \qquad \psi_1(\xi\xi') = \psi_1(\xi) * \psi_1(\xi')$$

Once $\xi = \xi'$ happens, we will get $\psi_1(\xi) = \psi_1(\xi')$. For location variable ρ, the $\psi_2(\rho)$ will be its actual location, thus for a permission $\xi v : \texttt{ptr}(\rho)$, we get $\psi_2(\rho) = \mu(v)$.

In order to track the fraction in permissions, we use ψ_3 to map from a permission bag into a set of mappings from pointer or location variables to numbers between zero and one:

$$\psi_3 : (\Pi \rightharpoonup (V \cup L \rightarrow (0, 1]))$$

First, we apply ψ_3 to single permissions and get:

$$\begin{array}{l} \psi_3(\xi v : \texttt{ptr}(\rho)) = \{v \mapsto \psi_1(\xi)\} \\ \psi_3(\xi\rho) = \{\psi_2(\rho) \mapsto \psi_1(\xi)\} \\ \psi_3(\rho{=}\rho') = \{\} \text{ if } \psi_2(\rho) = \psi_2(\rho') \end{array}$$

To apply the ψ_3 on permission bag Π, it's required to merge all the "compatible" permissions to get a "compact" permission bag first.

Definition 3: Two permissions π_1 and π_2 are compatible in a permission bag Π if either:

- $\pi_1 = \pi_2$;
- $\pi_1 = \xi_1\rho_1$, $\pi_2 = \xi_2\rho_2$ and $\Pi = (\rho_1 = \rho_2), \Pi'$ for some Π';
- $\pi_1 = \xi_1 v : \texttt{ptr}(\rho_1)$, $\pi_2 = \xi_2 v : \texttt{ptr}(\rho_2)$ and $\Pi = (\rho_1{=}\rho_2), \Pi'$ for some Π'.

Definition 4: A permission bag Π is compact if for any π_1 and π_2, such that $\Pi = \Pi', \pi_1, \Pi'', \pi_2, \Pi'''$ for some Π', Π'' and Π''', π_1 and π_2 are not compatible.

Theorem 1: Every permission bag Π can be equally transformed to a compact permission bag: $\forall \Pi, \exists \Pi' : (\Pi \equiv \Pi') \wedge \text{Compact}(\Pi')$.

Then we could apply ψ_3 to a compact permission bag:

$$\frac{\text{Compact}(\Pi) \qquad \Pi = \pi, \Pi'}{\psi_3(\Pi) = \psi_3(\pi) \cup \psi_3(\Pi')}$$

For simplicity, we do not distinguish ψ_1, ψ_2 and ψ_3. All of them are replaced by ψ. The total consistent rules is:

$$\frac{\begin{array}{c} \exists\psi.(\psi; \mu \vdash \Pi \text{ consistent}) \quad \wedge \\ (\psi; \mu; T; \Pi \vdash (\Phi_1||\Phi_2||...||\Phi_n); \hat{\Phi};\ \text{consistent}) \end{array}}{\Delta; \Pi; (\Phi_1||\Phi_2||...||\Phi_n); \hat{\Phi} \vdash \mu; T \text{ ok}}$$

which requires that the permission and lock information are consistent with correspondent runtime state. Auxiliary permission consistency rules and lock consistency rules are omitted here.

Lemma 5: Given a permission-checked program P, if a sequence of parallel statements $s_1||s_2||...||s_n$ with $(1 \leq n)$ can be permission checked with the environment $\Delta; \Pi; (\Phi_1||\Phi_2||...||\Phi_n); \hat{\Phi}$ $(\Delta; \Pi; (\Phi_1||\Phi_2||...||\Phi_n); \hat{\Phi} \vdash s_1||s_2||...||s_n \dashv \{\overline{E'}\})$ for some $\{\overline{E'}\}$ with the consistent memory μ and the thread information T $(\Delta; \Pi; (\Phi_1||\Phi_2||...||\Phi_n); \hat{\Phi} \vdash \mu; T$ ok), then either $n = 1$ and s_1 is `skip` or $s_1||s_2||...||s_n$ can be evaluated by either:

- eliminating a latest created thread:
  $$(\mu; (s_1||s_2||...||s_n)) \rightarrow (\mu; s_1||s_2||...||s_{n-1})$$
  then there exists an permission environment
  $\Delta; \Pi; (\Phi_1||\Phi_2||...||\Phi_{n-1}); \hat{\Phi}$
  which is consistent with $\mu; T \setminus \{T_n\}$ where T_n corresponds to Φ_n and also permission checks $s_1||s_2||...||s_{n-1}$ with the output environment $\{\overline{E'}\}$, such that:
  $$\Delta; \Pi; (\Phi_1||\Phi_2||...||\Phi_{n-1}); \hat{\Phi} \vdash s_1||s_2||...||s_{n-1} \dashv \{\overline{E'}\}$$
- nondeterministically picking a thread to evaluate:
  $$\begin{array}{c} (\mu; (s_1||s_2||...||s_j||...||s_n)) \rightarrow \\ \{(\mu_j; s_1||s_2||...||s'_j||...||s_n)|1 \leq j \leq n\} \end{array}$$
  then there exists a set of $\left\{\Delta_j; \Pi_j; (\Phi_{1_j}||\Phi_{2_j}||...||\Phi_{n_j}); \hat{\Phi}_j\right\}$ and $\{\sigma_j\}$, such that each is consistent with $\mu_j; T_j$ $(\Delta_j; \Pi_j; (\Phi_{1_j}||\Phi_{2_j}||...||\Phi_{n_j}); \hat{\Phi}_j \vdash \mu_j; T_j$ ok) and also permission checks the $s_1||s_2||...||s'_j||...||s_n$, that is:

$$\Delta_j; \Pi_j; (\Phi_{1_j}||\Phi_{2_j}||...||\Phi_{n_j}); \hat{\Phi}_j \vdash s_1||s_2||...||s'_j||...||s_n \dashv \{\overline{E''}\}$$

with $\overline{E'' \subseteq \sigma_j E'}$;

- creating a thread:

$$(\mu; (s_1||s_2||...||s_n)) \to (\mu; s_1||s_2||...||s_{n+1})$$

then there exists a permission environment $\Delta; \Pi; (\Phi_1||\Phi_2||...||\Phi_n||\emptyset); \hat{\Phi}$ which is consistent with $\mu; T \cup \{T_{n+1}\}$ where T_{n+1} corresponds to the new created thread and also permission checks $s_1||s_2||...||s_{n+1}$ with the output environment $\{\overline{E'}\}$, such that:

$$\Delta; \Pi; (\Phi_1||\Phi_2||...||\Phi_n||\emptyset); \hat{\Phi} \vdash s_1||s_2||...||s_{n+1} \dashv \{\overline{E'}\}$$

VI. FURTHER WORK

A. High Level Annotations

This paper is a subsequent work based on [11] and we only mention the low level permission system. In order to make the whole system work, we need to use the effect system and translate the high level annotations into the low level permissions which are ignored in this paper. We are currently designing the permission system for an object-oriented language and some high level annotations have been modelled by low level permissions, for instance, *unique*, *readonly*, *borrowed* and so on. Next, *final*, *guarded_by* and *requires* annotations may also be added.

B. Object-oriented Features

One of the most challenging features of object-oriented is polymorphism. The language in this paper only has two primitive types and a pointer type. If we add the object-oriented features, the permission type system will become more complicated. The actual runtime type may not be determined statically, therefore the conservatively approximation need to be applied. Furthermore, Boyland et al. [12] have developed a model of object invariant using "adoption", in which an invariant is established by the constructor and is always correct thereafter. Soundness for sequential programs has been proved and we hope to extend it to parallel programs.

C. Atomicity

"Race-free" is still a weak property and is not sufficient to ensure the absence of errors due to unexpected thread interactions. Instead, *atomicity* is much more stronger non-interference property. Atomic methods can be assumed to execute serially, without interleavings from other threads [13]. Flanagan suggest to add atomicity to parallel programs. To implement this using permission, some additional atomicity information or facts need to be attached into the permission environment.

VII. CONCLUSION

This paper presents a type system for a simple language with concurrency features using the fractional permissions. Permission is abstracted as one kind of semantic access right associated with every global variables or locations in the memory. Any write access is required to be granted a whole permission, which may be further split into fractions to allow read accesses. We give the permission rules to check the race conditions and deadlocks. Well-typed programs in permission system are guaranteed to be race-free and deadlock-free. At last, some consistency rules between the runtime state and the static permission environment are established.

REFERENCES

[1] Y. Zhao, "A simple permission checking for structural parallel programs," in *ICACTE '08*, Dec. 2008.

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

[3] C. Flanagan and Martín, "Object types against races," in *Conference on Concurrent Theory*, Aug. 1999.

[4] C. Boyapati, "Safejava: A unified type system for safe programming," Ph.D. dissertation, MIT, 2004.

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

[6] M. Abadi, C. Flanagan, and S. N. Freund, "Types for safe locking: Static race detection for java," *ACM Transactions on Programming Languages and Systems*, vol. 28, no. 2, pp. 207–255, Mar. 2006.

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

[8] R. Rugina and M. C. Rinard, "Pointer analysis for structured parallel programs," *ACM Transactions on Programming Languages and Systems*, vol. 25, no. 1, pp. 70–116, Jan. 2003.

[9] R. Bornat, C. Calcagno, P. O'Hearn, and M. Parkinson, "Permission accounting in separation logic," in *POPL '05*. ACM Press, 2005, pp. 259–270.

[10] L. Wang and S. D. Stoller, "Runtime analysis of atomicity for multithreaded programs," *IEEE Transactions on Software Engineering*, vol. 32, no. 2, pp. 93–110, Feb. 2006.

[11] J. Boyland, "Checking interference with fractional permissions," in *SAS '03*, ser. LNCS, vol. 2694. Springer, 2003, pp. 55–72.

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

[13] C. Flanagan and S. Qadeer, "A type and effect system for atomicity," in *PLDI '03*. ACM Press, 2003, pp. 338–349.

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 Associate Professor at Nanjing University of Sci.& Tech., China. His current research interests include program analysis and software engineering.