A Multithreading Language without Data races and Deadlocks

Liu Junhao, He Siqi

Motivation

- A problem in research:
- If a program can finish running in certain time after parallelized?
  - Auto parallelization?
  - Time complexity verifying?

M~~C~~ML: a Multithreading Language ~~(with Complexity Checking)~~

OU

parallel the python code:

```
for f in all_features:
        for t in previous_best:
            new_t = normalize_tuple(t + (f, ))
            if len(new_t) != len(t) + 1:
                continue
            if new_t not in new_tuples:
                new_tuples.add(new_t)
                state['t_order'][new_t] = copy.deepcopy(state['t_order'][t])
                state['t_order'][new_t].append(f)
                state['t_coverage_idx'][new_t] = (
                    state['t_coverage_idx'][t].intersection(
                        state['t_coverage_idx'][(f,)]))
                state['t_coverage'][new_t] = (
                    float(len(state['t_coverage_idx'][new_t])) /
                    coverage_data.shape[0])
                t_idx = np.array(list(state['t_idx'][t]))
                t_data = state['data'][t_idx]
                present = np.where(t_data[:, f] == 1)[0]
                state['t_idx'][new_t] = set(t_idx[present])
                idx_list = list(state['t_idx'][new_t])
                state['t_nsamples'][new_t] = float(len(idx_list))
                state['t_positives'][new_t] = np.sum(
                    state['labels'][idx_list])
```

OU

parallel the python code:

```
for f in all_features:
        for t in previous_best:
            new_t = normalize_tuple(t + (f, ))
            if len(new_t) != len(t) + 1:
                continue
            if new_t not in new_tuples:
                new_tuples.add(new_t)
```

Here's the Python code parallelized using the multiprocessing library:

python Copy code

```python
import multiprocessing as mp
import copy

def process_feature(f, previous_best, new_tuples, state, coverage_data):
    for t in previous_best:
        new_t = normalize_tuple(t + (f,))
        if len(new_t) != len(t) + 1:
            continue
        if new_t not in new_tuples:
            new_tuples.add(new_t)
            state['t_order'][new_t] = copy.deepcopy(state['t_order'][t])
            state['t_order'][new_t].append(f)
            state['t_coverage_idx'][new_t] = (
                state['t_coverage_idx'][t].intersection(
                    state['t_coverage_idx'][(f,)]))
            state['t_coverage'][new_t] = (
                float(len(state['t_coverage_idx'][new_t])) /
```

ChatGPT may fool you

OU

Will it cause problems that operating on `state` simutaneous?

Yes, there is a risk of race conditions when operating on the `state` dictionary simultaneously in multiple processes. When multiple processes modify the same object concurrently, it can lead to inconsistent or incorrect results.

Goal

- develop a data race-free multithreading language based on “fullrecon”
  - no data races
  - no dead locks
- programmers to write multithreaded code in a natural way without worrying about data races and deadlocks.

Feasibility

OCaml

API Version 5.0

< General Index

OCaml library : Thread

Module Thread

```
module Thread: sig .. end
```

Lightweight threads for Posix `1003.1c` and Win32.

Feasibility

Ownership Types for Safe Programming: Preventing Data Races and Deadlocks

Chandrasekhar Boyapati Robert Lee Martin Rinard

Laboratory for Computer Science
Massachusetts Institute of Technology

A Type System for Preventing Data Races and Deadlocks in the Java Virtual Machine Language

Pratibha Permandla Chandrasekhar Boyapati

ACM Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA), October 2001

A Parameterized Type System for Race-Free Java Programs

Chandrasekhar Boyapati Martin Rinard

Plans

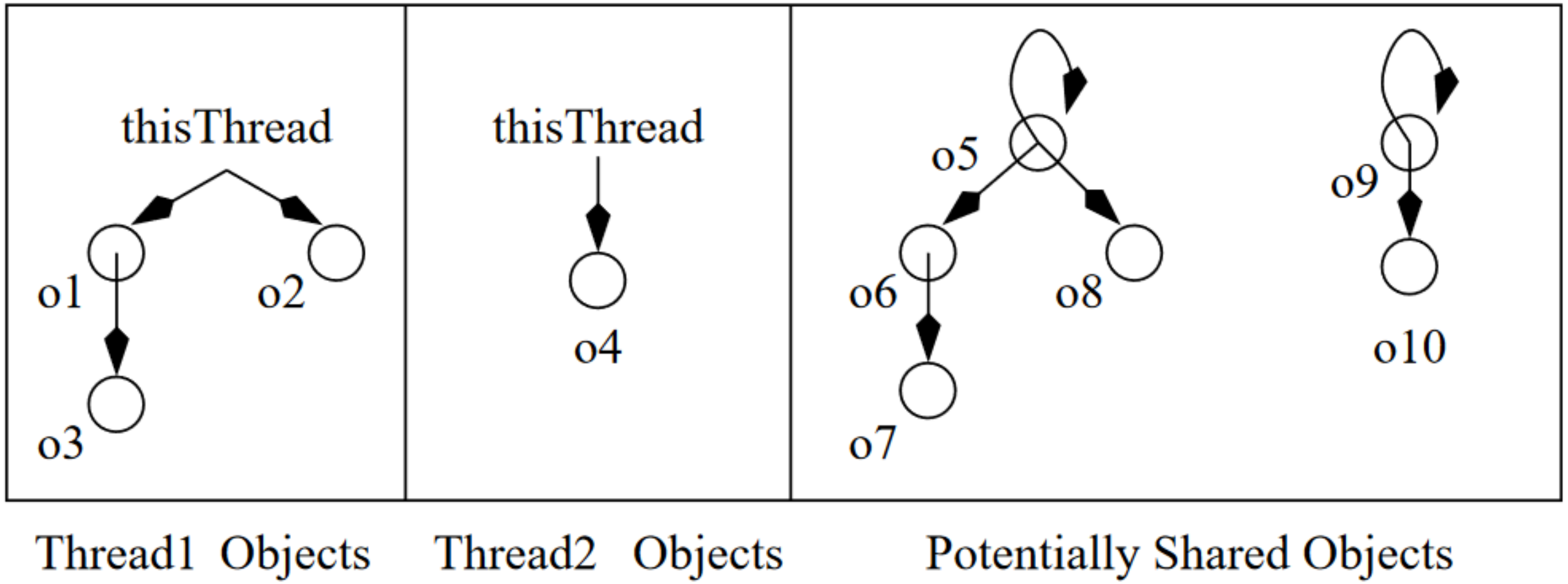
- Design and implement multithreading functionality [1]
- Define and implement lock types [2]
- Design and implement deadlock detection algorithm [2]
- Possibly include features such as exception detection for concurrent errors

- [1]Learning from Ocaml language's Parallel programming code.
- [2] Implementation will be based on the research paper "Ownership types for safe programming: preventing data races and deadlocks"

Type System to Prevent Data Races

$$
\begin{array}{rcl}
\mathit{defn} & ::= & \mathsf{class}\ cn\langle owner\ formal*\rangle\ \mathsf{extends}\ c\ body \\
c & ::= & cn\langle owner{+}\rangle \mid \mathsf{Object}\langle owner{+}\rangle \\
owner & ::= & formal \mid \mathsf{self} \mid \mathsf{thisThread} \mid e_{\mathrm{final}} \\
meth & ::= & t\ mn(arg*)\ \mathsf{accesses}\ (e_{\mathrm{final}}*)\ \{e\} \\
e_{\mathrm{final}} & ::= & e \\
formal & ::= & f \\
 & & \\
f & \in & \text{owner names}
\end{array}
$$

Figure 6: Grammar Extensions for Race-Free Java

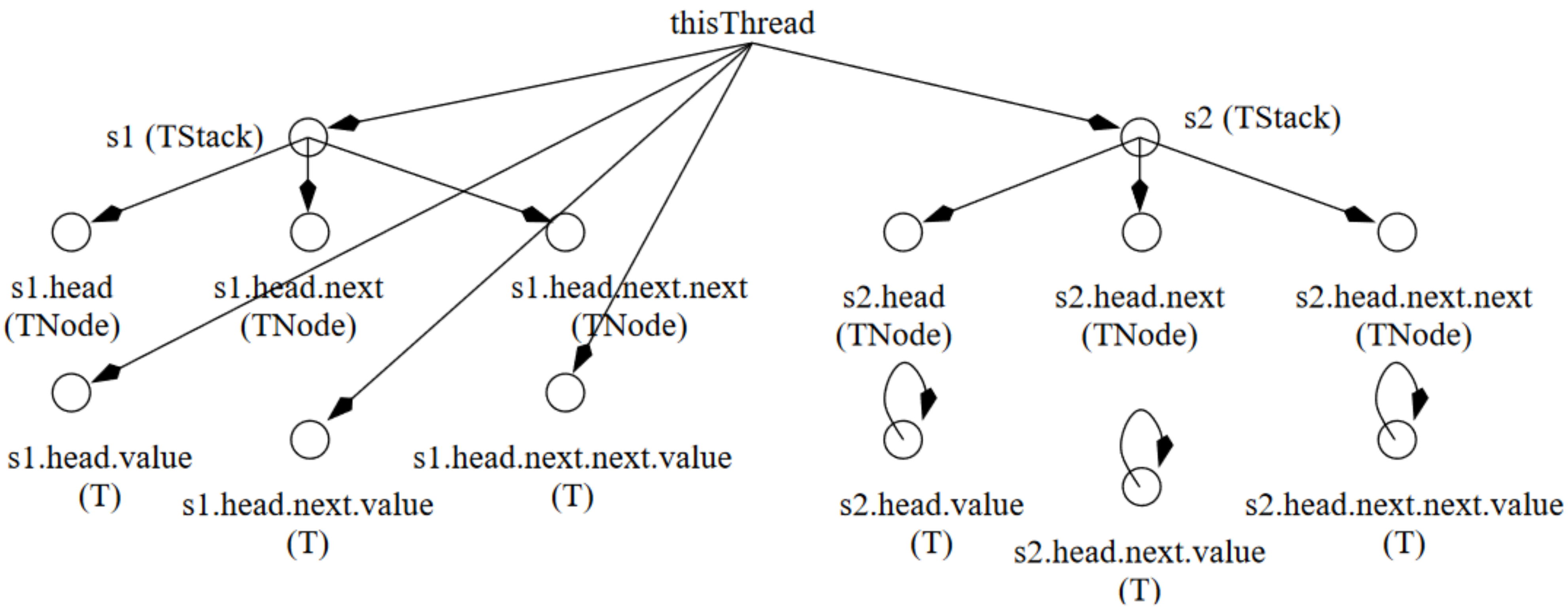

thisThread
o1
o2
o3
thisThread
o4
o5
o6
o7
o8
o9
o10
Thread1 Objects
Thread2 Objects
Potentially Shared Objects

Stack of T Objects in Race-Free Java

```
// thisOwner owns the TStack object
// TOwner owns the T objects in the stack.

class TStack<thisOwner, TOwner> {
  TNode<this, TOwner> head = null;

  T<TOwner> pop() accesses (this) {
    if (head == null) return null;
    T<TOwner> value = head.value();
    head = head.next();
    return value;
  }
  ...
}
```

```
class TNode<thisOwner, TOwner> {
  T<TOwner> value;
  TNode<thisOwner, TOwner> next;

  T<TOwner> value() accesses (this) {
    return value;
  }
  TNode<thisOwner, TOwner> next() accesses (this) {
    return next;
  }
  ...
}
class T<thisOwner> { int x=0; }

TStack<thisThread, thisThread> s1 =
  new TStack<thisThread, thisThread>;
TStack<thisThread, self>       s2 =
  new TStack<thisThread, self>;
```

Ownership Relation for TStacks s1 and s2



```
// thisOwner owns the Account object
class Account<thisOwner> {
  int balance = 0;
  int deposit(int x) requires (this) {
    this.balance = this.balance + x;
  }
}

// Acount a1 is owned by this thread, so it is thread-local
Account<thisThread> a1 = new Account<thisThread>;
a1.deposit(10);

// Account a2 owns itself, so it can be shared between threads
final Account<self> a2 = new Account<self>;
fork (a2) { synchronized (a2) in { a2.deposit(10); } }
fork (a2) { synchronized (a2) in { a2.deposit(10); } }

final Account<self> a3 = new Account<self>;
Account<a3>          a4 = new Account<a3>;
```

Type System to Prevent Deadlocks

$$
\begin{array}{rcl}
\textit{body} & ::= & \{\textit{level}*\ \textit{field}*\ \textit{meth}*\} \\
\textit{level} & ::= & \mathsf{LockLevel}\ l = \mathsf{new} \mid \mathsf{LockLevel}\ l < cn.l* > cn.l* \\
\textit{owner} & ::= & \textit{formal} \mid \mathsf{self}{:}cn.l \mid \mathsf{thisThread} \mid e_{\text{final}} \\
\textit{meth} & ::= & t\ mn(arg*)\ \mathsf{accesses}\ (e_{\text{final}}*)\ \textit{locksclause}\ \{e\} \\
\textit{locksclause} & ::= & \mathsf{locks}\ (cn.l*\ [\textit{lock}]_{\text{opt}}) \\
\textit{lock} & ::= & e_{\text{final}} \\
 & & \\
l & \in & \text{lock level names}
\end{array}
$$

Figure 9: Grammar Extensions for Deadlock-Free Java

Lock Level Properties

L1. The lock levels form a partial order.

L2. Objects that own themselves are locks. Every lock belongs to some lock level. The lock level of a lock does not change over time.

L3. The necessary and sufficient condition for a thread to acquire a new lock l is that the levels of all the locks that the thread currently holds are greater than the level of l.

L4. A thread may also acquire a lock that it already holds. The lock acquire operation is redundant in that case.

Combined Account Example in Deadlock-Free Java

```
class Account {
  int balance = 0;

  int balance()        accesses (this) { return balance; }
  void deposit(int x)  accesses (this) { balance += x; }
  void withdraw(int x) accesses (this) { balance -= x; }
}

class CombinedAccount<readonly> {
  LockLevel savingsLevel = new;
  LockLevel checkingLevel < savingsLevel;
  final Account<self:savingsLevel> savingsAccount
    = new Account;
  final Account<self:checkingLevel> checkingAccount
    = new Account;

  void transfer(int x) locks(savingsLevel) {
    synchronized (savingsAccount) {
      synchronized (checkingAccount) {
        savingsAccount.withdraw(x);
        checkingAccount.deposit(x);
  }}}
  int creditCheck() locks(savingsLevel) {
    synchronized (savingsAccount) {
      synchronized (checkingAccount) {
        return savingsAccount.balance() +
               checkingAccount.balance();
  }}}
  ...
}
```

Optional

- with complexity checking…

- Plan:
  Add a parallel complexity label to the language to automatically analyze and check complexity.

- Reference:
  Type-Based Complexity Analysis for Fork Processes

Schedule

- 7 weeks to final presentation:
  - Develop of Thread + Test ——1~2 week
  - Data Race-Free part + Test ——1.5~2.5 week
  - Deadlock-Free part + Test ~——1.5~2.5 week

Thanks for listening!