An Effect System for Checking Consistency of Synchronization and Yields

Technical Report UCSC-SOE-09-33

Jevgenia Smorgun  Jaeheon Yi
University of California at Santa Cruz
jsmorgun@ucsc.edu, jaeheon@soe.ucsc.edu

1. Introduction

Type-and-effect systems can guard against race conditions by statically enforcing a locking discipline [1]. A program’s synchronization structure enforces a program’s locking discipline. Whether or not a program’s locking discipline is enforced by its synchronization structure is a previously studied question.

A yield is a multithreading synchronization mechanism for automatic mutual exclusion (AME) [2], where multithreading is explicitly allowed at selected yield points, and excluded elsewhere. AME’s semantics have cooperative multithreading, where the yield explicitly permits preemptions to occur. We consider yields as a specification in a non-cooperative semantics, such that yields indicate program points where the programmer expects a preemption to possibly occur: a yielding discipline.

Given a program, are its synchronization structure and yielding discipline consistent with each other? We propose an effect system for this problem.

2. Concurrent IMP

Our Concurrent IMP programming language [3] consists of the following domains, including commands.

- \( e \in \text{AEXP} \) := \( \ldots \)
- \( b \in \text{BEXP} \) := \( \ldots \)
- \( x \in \text{VAR} \) := \( \ldots \)
- \( m \in \text{LOCK} \) := \( \ldots \)
- \( d \in \text{DECL} \) := \( \text{var} x \) [guarded_by \( m \)]_{opt}
- \( v \in \text{VAL} \) := \( Z \cup \{ \text{true}, \text{false} \} \)
- \( C, D \in \text{CMD} \) := \( \text{CMD} \mid \text{CMD} \)
  \hspace{1cm} | \hspace{1cm} \text{VAR} := \text{AEXP} \)
  \hspace{1cm} | \hspace{1cm} \text{sync LOCK in CMD}
  \hspace{1cm} | \hspace{1cm} \text{yield}
  \hspace{1cm} | \hspace{1cm} \text{skip}
  \hspace{1cm} | \hspace{1cm} \text{if BEXP then CMD else CMD}
  \hspace{1cm} | \hspace{1cm} \text{while BEXP do CMD}

Figure 1. Domains of Concurrent IMP

A program in IMP is a declaration of variables, a set of commands representing the thread pool, and the accompanying state. Threads finish when their command is \text{skip}. The program is finished when all threads are \text{skip}.

A context is an expression with a hole; an evaluation context \( \mathcal{E} \) is a context used during evaluation: \( \mathcal{E} = [ \cdot ] \mid \mathcal{E} ; \text{CMD} \). If \( \mathcal{E} \) is a metavariable ranging over eval contexts and we have some expression \( C \), we take \( \mathcal{E}[C] \) to mean the context \( \mathcal{E} \) with \( C \) placed in \( \mathcal{E} \)'s hole.

Every command \( C \) defines two program points, \( C^- \) and \( C^+ \), representing the points just before and after \( C \) executes.

We may query the guarding lock set for each variable from the declaration of variables by the function \( \text{LS} : \text{VAR} \to 2^\text{LOCK} \).

2.1 Evaluation Rules

We assume an interleaving semantics where the scheduling is non-cooperative; a preemption may occur after any evaluation step. Evaluation steps are atomic: when one evaluation step occurs, no evaluation step by another thread may occur simultaneously.

We represent the state space of the program as follows:

\[
\pi : \text{LOCK} \to \{ \text{locked, unlocked} \}
\]

\[
\sigma : \text{VAR} \to \text{VAL}
\]

\[
\mathcal{T} : \text{THREAD} \to \text{CMD}
\]

The initial state for the program is

\[
\Sigma = ( \lambda m . \text{unlocked}, \lambda x . 0, \lambda t . C_t )
\]

where \( C_t \) is the initial command defined in the program for each thread \( t \).

Transition rules express the effect of the command evaluation on the state (Figure 2).

3. Locking and Yielding

A locking discipline is a mapping \( \text{VAR} \to 2^\text{LOCK} \). The locking discipline of a program tells us what variables are protected by which lock, and is defined in the program’s variable declaration.

In our language, variable accesses in command \( C \) are protected by a lock \( m \) through the synchronization command \( \text{sync } m \) in \( C \). Such a command may disallow observable preemptions by other threads from occurring through an underlying mutual exclusion mechanism. A variable may not have a declared lockset; a racy access is an access to such a variable.

A program’s synchronization structure is the set of \text{sync} commands and racy accesses in the program. A synchronization structure defines the set of program points \( S \) where preemptions are intended to occur: the program points before and after \text{sync} commands and racy accesses.

A yielding discipline is the set of \text{yield} commands in the program. A yield specifies a program point where the programmer explicitly expects preemptions to possibly occur. We indicate a yielding discipline’s preemption points with \( \gamma \).
A yielding discipline is \textit{consistent} with respect to the synchronization structure if for every pair of elements \((C, D)\) in a thread’s synchronization structure such that \(C^+\) sequentially comes before \(D^-\) in the thread command, there exists a yield command between \(C^+\) and \(D^-\).

A consistent yielding discipline is easily obtained by wrapping every other command between two yield commands. A consistent yielding discipline is \textit{excessive} if removing one yield command still maintains a consistent yielding discipline.

4. Effect System for Concurrent IMP

A type-and-effect system is a type system augmented with special rules to reason about computational effects that may occur during run time [4]. Type-and-effect systems are widely used to statically check for a variety of program effects, such as memory allocation and exception throwing.

We have the following effect system to check for consistency of synchronization structure and yield discipline (Figure 3). A type-and-effect system may be straightforwardly obtained by adding in typing judgments for arithmetic and boolean expressions.

The effect judgment \(\Phi \vdash C : \varepsilon\) judges command \(C\) to have effect \(\varepsilon\) in the environment \(\Phi\), consisting of the available lock set. Specifically, \(\Phi \subseteq \mathcal{L}^{\text{LOCK}}\).

An effect is a static approximation of program behavior:

- \(\text{S}\) is the empty effect - nothing of interest happens; it is also the identity effect for sequencing;
- \(\text{R}\) implies a race condition;
- \(\text{Y}\) means a preemption may occur;
- \(\text{RY}\) is the sequential effect of an \(\text{R}\) then \(\text{Y}\);
- \(\text{YR}\) is the sequential effect of a \(\text{Y}\) then \(\text{R}\);
- \(\text{BAD}\) is an error condition.

When sequentially composing two effects via the ; command, we summarize the combined effect as listed in Figure 4.

![Figure 2. Evaluation Rules](image)

![Figure 4. Sequential Effect Combination](image)
may be executed zero or more times. We list the effect of a `sync`
command and `if` command in Figure 5.
The `if` command executes one of two nested commands. To
summarize the effect of the `if` command, we find the join (or least
upper bound) of two effects within a lattice of effects (Figure 6).

### Figure 3. Effect System

<table>
<thead>
<tr>
<th>( \varepsilon )</th>
<th><code>sync m in C</code></th>
<th><code>while b do C</code></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>S</code></td>
<td><code>R</code></td>
<td><code>S</code></td>
</tr>
<tr>
<td><code>Y</code></td>
<td><code>R</code></td>
<td><code>Y</code></td>
</tr>
<tr>
<td><code>RY</code></td>
<td><code>RY</code></td>
<td><code>RY</code></td>
</tr>
<tr>
<td><code>YR</code></td>
<td><code>YR</code></td>
<td><code>YR</code></td>
</tr>
<tr>
<td><code>BAD</code></td>
<td><code>BAD</code></td>
<td><code>BAD</code></td>
</tr>
</tbody>
</table>

### Figure 5. Effect of a Synchronization Block or While Loop

Four functions summarize effect combination:
- \( s(\varepsilon_1, \varepsilon_2) \) for the sequencing command;
- \( k(\varepsilon) \) for a nested effect within a `sync` command;
- \( w(\varepsilon) \) for a nested effect within a `while` loop;
- \( \varepsilon_1 \sqcup \varepsilon_2 \) for two nested effects within an `if` command.

#### Example

1. Unintentional races are caught by the effect system.

   ```
   var x guarded_by m
   x := 2
   ```

2. Intentional races are fine, as long as the yielding discipline is consistent. This program thread has two racy accesses on \( y \) but no intervening `yield` in between; the program effect is `BAD`.

   ```
   var x guarded_by m
   var y
   ```

3. Here is a well-synchronized program. The yielding discipline is consistent.

   ```
   var x guarded_by m
   sync m {
     x := 2;
     y := 1;
   }
   ```

4. Another well-synchronized program.

   ```
   var x guarded_by m
   var y guarded_by m
   var z guarded_by m
   sync m {
     x := 3;
     y := 2;
     z := 1;
     x := 4
   }
   ```

5. A similar program to above, but with an intentional race on \( x \) and a `yield` to indicate a race. Without the `yield`, the program’s effect is `BAD`. With the `yield`, the program’s effect is `R`.

   ```
   var x
   var y guarded_by m
   var z guarded_by m
   sync m {
     x := 3;
     yield;
     y := 2;
     z := 1;
     x := 4
   }
   ```

6. The `then` branch of the `if` command has a race, while the `else` branch doesn’t. We conservatively summarize the effect of the `if` command as `R`.

   ```
   var x
   var y guarded_by m
   var z guarded_by m
   ```
sync m { 
    if b then 
    x := 1 
    else 
    y := 2 
}

7. A while command’s effect can be summarized by sequentially composing the nested effect with itself. Since the while command executes a racy access, two consecutive racy accesses with no intervening yield is BAD.

    var x 
    while b do 
    x := 1

8. A more complicated example with two threads. The yielding discipline is excessive but consistent with the program’s synchronization structure.

    var x 
    var y guarded_by m 
    var z guarded_by n 
    sync m in { 
        sync n in { 
            while b1 do 
                x := 3; 
                yield; 
                if b2 then 
                    x := 2; 
                    yield 
                else 
                    y := 3 
                    ; 
                    yield; 
                    x := 2; 
                    yield 
            } 
        } 
    }
    sync n in { 
        z := 3 
    };
    x := 1

References