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 command 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	$\in$	AEXP	::=	$\cdots$
b	$\in$	BEXP	::=	$\cdots$
x	$\in$	VAR	::=	$\cdots$
m	$\in$	LOCK	::=	$\cdots$
d	$\in$	DECL	::=	`var` x [`guarded_by` m]$_{opt}$
v	$\in$	VAL	::=	$\mathcal{Z} \cup \{$`true`, `false`$\}$
C, D	$\in$	CMD	::=	
			|	CMD `;` CMD
			|	VAR `:=` AEXP
			|	`sync` LOCK `in` CMD
			|	`yield`
			|	`skip`
			|	`if` BEXP `then` CMD `else` CMD
			|	`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 `skip`. The program is finished when all threads are `skip`.

A context is an expression with a hole; an evaluation context $\mathcal{E}$ is a context used during evaluation: $\mathcal{E}$ = `[]` | $\mathcal{E}$ `;` 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 $\text{LS} : \text{VAR} \rightarrow 2^{\text{LOCK}}$.

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:

$$\pi : \text{LOCK} \rightarrow \{\texttt{locked}, \texttt{unlocked}\}$$

$$\sigma : \text{VAR} \rightarrow \text{VAL}$$

$$\mathcal{T} : \text{THREAD} \rightarrow \text{CMD}$$

The initial state for the program is

$$\Sigma = \langle\, \lambda m \,.\, \texttt{unlocked}, \; \lambda x \,.\, 0, \; \lambda t \,.\, C_t \rangle$$

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 $\text{VAR} \rightarrow 2^{\text{LOCK}}$. 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 `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 `sync` commands and racy accesses in the program. A synchronization structure defines the set of program points $\mathcal{S}$ where preemptions are intended to occur: the program points before and after `sync` commands and racy accesses.

A *yielding discipline* is the set of `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 $\mathcal{Y}$.

[E-SKIP]

$$\frac{\begin{array}{rcl}\mathcal{T}(t) &=& \mathcal{E}[\texttt{skip ; } C] \\ \mathcal{T}' &=& \mathcal{T}[t := \mathcal{E}[C]]\end{array}}{\langle \pi, \sigma, \mathcal{T}\rangle \longrightarrow^{\texttt{skip}} \langle \pi, \sigma, \mathcal{T}'\rangle}$$

[E-ASSIGN]

$$\frac{\begin{array}{rcl}\mathcal{T}(t) &=& \mathcal{E}[x \texttt{ := } e] \\ \sigma(e) &=& v \\ \sigma' &=& \sigma[x := v] \\ \mathcal{T}' &=& \mathcal{T}[t := \mathcal{E}[\texttt{skip}]]\end{array}}{\langle \pi, \sigma, \mathcal{T}\rangle \longrightarrow^{x \texttt{ := } e} \langle \pi, \sigma', \mathcal{T}'\rangle}$$

[E-YIELD]

$$\frac{\begin{array}{rcl}\mathcal{T}(t) &=& \mathcal{E}[\texttt{yield}] \\ \mathcal{T}' &=& \mathcal{T}[t := \mathcal{E}[\texttt{skip}]]\end{array}}{\langle \pi, \sigma, \mathcal{T}\rangle \longrightarrow^{\texttt{yield}} \langle \pi, \sigma, \mathcal{T}'\rangle}$$

[E-SYNC]

$$\frac{\begin{array}{rcl}\mathcal{T}(t) &=& \mathcal{E}[\texttt{sync } m \texttt{ in } C] \\ \pi(m) &=& \texttt{unlocked} \\ \mathcal{T}' &=& \mathcal{T}[t := \mathcal{E}[\texttt{in-sync } m \texttt{ in } C]] \\ \pi' &=& \pi[m := \texttt{locked}]\end{array}}{\langle \pi, \sigma, \mathcal{T}\rangle \longrightarrow^{\texttt{sync } m \texttt{ in } C} \langle \pi', \sigma, \mathcal{T}'\rangle}$$

[E-INSYNC]

$$\frac{\begin{array}{rcl}\mathcal{T}(t) &=& \mathcal{E}[\texttt{in-sync } m \texttt{ in skip}] \\ \pi(m) &=& \texttt{locked} \\ \mathcal{T}' &=& \mathcal{T}[t := \mathcal{E}[\texttt{skip}]] \\ \pi' &=& \pi[m := \texttt{unlocked}]\end{array}}{\langle \pi, \sigma, \mathcal{T}\rangle \longrightarrow^{\texttt{in-sync } m \texttt{ in skip}} \langle \pi', \sigma, \mathcal{T}'\rangle}$$

[E-WHILE]

$$\frac{\begin{array}{rcl}\mathcal{T}(t) &=& \mathcal{E}[\texttt{while } b \texttt{ do } C\ ] \\ \mathcal{T}' &=& \mathcal{T}[t := \mathcal{E}[\texttt{if } b \texttt{ then } (C \texttt{ ; while } b \texttt{ do } C\ ) \texttt{ else skip}]]\end{array}}{\langle \pi, \sigma, \mathcal{T}\rangle \longrightarrow^{\texttt{while } b \texttt{ do } C} \langle \pi, \sigma, \mathcal{T}'\rangle}$$

[E-IFTRUE]

$$\frac{\begin{array}{rcl}\mathcal{T}(t) &=& \mathcal{E}[\texttt{if } b \texttt{ then } C \texttt{ else } D] \\ \sigma(b) &=& \texttt{true} \\ \mathcal{T}' &=& \mathcal{T}[t := C]\end{array}}{\langle \pi, \sigma, \mathcal{T}\rangle \longrightarrow^{\texttt{if } b \texttt{ then } C \texttt{ else } D} \langle \pi, \sigma, \mathcal{T}'\rangle}$$

[E-IFFALSE]

$$\frac{\begin{array}{rcl}\mathcal{T}(t) &=& \mathcal{E}[\texttt{if } b \texttt{ then } C \texttt{ else } D] \\ \sigma(b) &=& \texttt{false} \\ \mathcal{T}' &=& \mathcal{T}[t := D]\end{array}}{\langle \pi, \sigma, \mathcal{T}\rangle \longrightarrow^{\texttt{if } b \texttt{ then } C \texttt{ else } D} \langle \pi, \sigma, \mathcal{T}'\rangle}$$

Figure 2. Evaluation Rules

3.1 Consistency

A yielding discipline is *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 *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 ε in the environment Φ, consisting of the available lock set. Specifically, $\Phi \subseteq 2^{\text{LOCK}}$.

An effect is a static approximation of program behavior:

S is the empty effect - nothing of interest happens; it is also the identity effect for sequencing;

R implies a race condition;

Y means a preemption may occur;

RY is the sequential effect of an R then Y;

YR is the sequential effect of a Y then R;

BAD is an error condition.

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

$s(\varepsilon_1, \varepsilon_2)$	R	YR	RY	S	Y
R	BAD	R	BAD	R	RY
YR	BAD	YR	BAD	YR	Y
RY	R	R	RY	RY	RY
S	R	YR	RY	S	Y
Y	YR	YR	Y	Y	Y

Figure 4. Sequential Effect Combination

The sequential combination of BAD and any other effect is still BAD. We may also flag a warning to indicate excessive yields for the following four effect combinations:

Y ; Y

Y ; YR

RY ; Y

RY ; YR

The `sync` command executes its nested command while holding some lock. The `while` command also has a nested command; this

$$\frac{\textsc{ls}(x) = \emptyset}{\Phi \vdash x \texttt{ := } v : \textsc{r}}\ \text{[T-ASSIGNRACE]} \qquad \frac{\textsc{ls}(x) \subseteq \Phi}{\Phi \vdash x \texttt{ := } v : \textsc{s}}\ \text{[T-ASSIGNSKIP]}$$

$$\frac{\Phi \cup \{m\} \vdash C : \varepsilon_c \quad \varepsilon = k(\varepsilon_c)}{\Phi \vdash \texttt{sync}\ m\ \texttt{in}\ C : \varepsilon}\ \text{[T-SYNC]} \qquad \frac{}{\Phi \vdash \texttt{yield} : \textsc{y}}\ \text{[T-YIELD]}$$

$$\frac{\Phi \vdash C_1 : \varepsilon_1 \quad \Phi \vdash C_2 : \varepsilon_2 \quad \varepsilon = s(\varepsilon_1, \varepsilon_2)}{\Phi \vdash C_1\ \texttt{;}\ C_2 : \varepsilon}\ \text{[T-SEQ]} \qquad \frac{\Phi \vdash C_1 : \varepsilon_1 \quad \Phi \vdash C_2 : \varepsilon_2 \quad \epsilon = \varepsilon_1 \sqcup \varepsilon_2}{\Phi \vdash \texttt{if}\ b\ \texttt{then}\ C_1\ \texttt{else}\ C_2 : \varepsilon}\ \text{[T-IF]}$$

$$\frac{\Phi \vdash C : \varepsilon_c \quad \varepsilon = w(\varepsilon_c)}{\Phi \vdash \texttt{while}\ b\ \texttt{do}\ C\ : \varepsilon}\ \text{[T-WHILE]} \qquad \frac{}{\Phi \vdash \texttt{skip} : \textsc{s}}\ \text{[T-SKIP]}$$

Figure 3. Effect System

ε	`sync` m `in` C $k(\varepsilon)$	`while` b `do` C $w(\varepsilon)$
S	R	S
Y	R	Y
R	R	BAD
RY	RY	RY
YR	YR	YR
BAD	BAD	BAD

Figure 5. Effect of a Synchronization Block or While Loop

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).

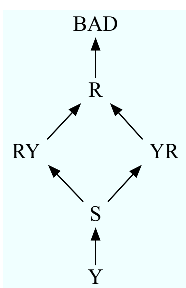

Figure 6. Joining Effects for `if` Command: $\varepsilon_1 \sqcup \varepsilon_2$

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.

5. Examples

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

y := 0;
sync m {
  x := 2;
  y := 1;
}
```

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

```
var x guarded_by m

sync m {
  x := 2;
  x := 3
}
```

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
```

```
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

[1] Cormac Flanagan and Stephen N. Freund. Type inference against races. *Sci. Comput. Program.*, 64(1):140–165, 2007.

[2] Martín Abadi, Andrew Birrell, Tim Harris, and Michael Isard. Semantics of transactional memory and automatic mutual exclusion. In *POPL*, pages 63–74. ACM, 2008.

[3] Glynn Winskel. *The formal semantics of programming languages: an introduction*. MIT Press, Cambridge, MA, USA, 1993.

[4] Daniel Marino and Todd D. Millstein. A generic type-and-effect system. In *TLDI*, pages 39–50, 2009.

[5] Martín Abadi and Gordon D. Plotkin. A model of cooperative threads. In *POPL*, pages 29–40, 2009.

[6] Cormac Flanagan and Martín Abadi. Types for safe locking. In *ESOP*, pages 91–108, 1999.