Semantics For Concurrent Separation Logic Using Relaxed Memory Models

Aaron Gostein, Stephen Brookes

Introduction

Concurrent Programming is a highly important topic in Computer Science as it allows for the possibility of simultaneously running multiple tasks, which has a wide variety of applications.

In logic for programming languages, we write statements of the form $\{P\}c\{Q\}$ to mean that if the program c is run from a state with property P, then if c terminates the final state will have property Q.

Concurrent Programming Is Tricky

Consider the statement
$\{x = 0\}\ x := x + 1 \,\|\, x := x + 1\ \{x = 2\}$
where $\|$ denotes parallel composition.

Is this statement valid?
Consider the possible order of memory actions:

1. *Read* $x = 0$ in left program
2. *Read* $x = 0$ in right program
3. *Write* $x := 0 + 1 = 1$ in left program
4. *Write* $x := 0 + 1 = 1$ in right program

With the above interleaving of the parallel threads, we get unintended behavior.

In order to properly deal with such race conditions, Professor Brookes from CMU along with Peter O'Hearn developed Concurrent Separation Logic (CSL) in the early 2000s, which is a logic for concurrent programs that use shared memory. In CSL, every provable program is ***race-free***. However, an inherent assumption in their work is that the memory model used by the computer architecture is ***sequentially consistent*** (SC), meaning that there is a total order on the reads and writes performed by all threads of the program. In this work, we make progress towards a proof that the semantics for CSL used earlier can be extended to show that CSL is sound for all reasonable memory models.

Weak Memory

In the original paper introducing CSL semantics, Brookes used ***action traces*** (sequences of memory actions) to model the executions of programs. Traces correspond directly with total orders, which works for SC because in SC, every pair of memory actions is ordered. However, as SC is quite a rigid guarantee, modern computer architectures provide more relaxed memory models for the sake of performance improvement. Such relaxed memory models include:

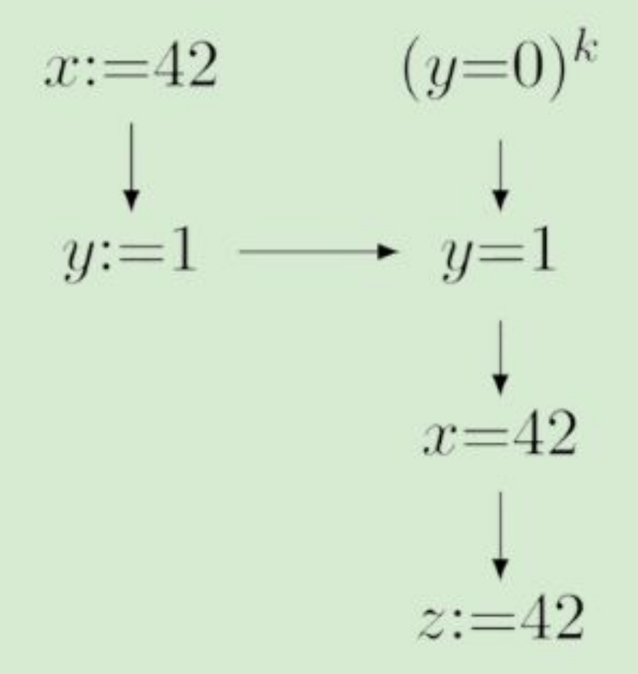
- Total Store Order (TSO):
  Each pair of **writes** is ordered
- Partial Store Order (PSO):
  Each pair of **writes to the same variable** is ordered

We say that TSO can ***relax*** (change the order of) pairs of memory actions of the form $(\mathbf{Write}(x),\ \mathbf{Read}(y))$, where x and y are different variables. PSO relaxes these pairs, and also $(\mathbf{Write}(x),\ \mathbf{Write}(y))$. TSO is guaranteed to order pairs of the form $(\mathbf{Write}(x),\ \mathbf{Write}(x))$ and $(\mathbf{Write}(x),\ \mathbf{Write}(y))$, whereas PSO is only guaranteed to order $(\mathbf{Write}(x),\ \mathbf{Write}(x))$.

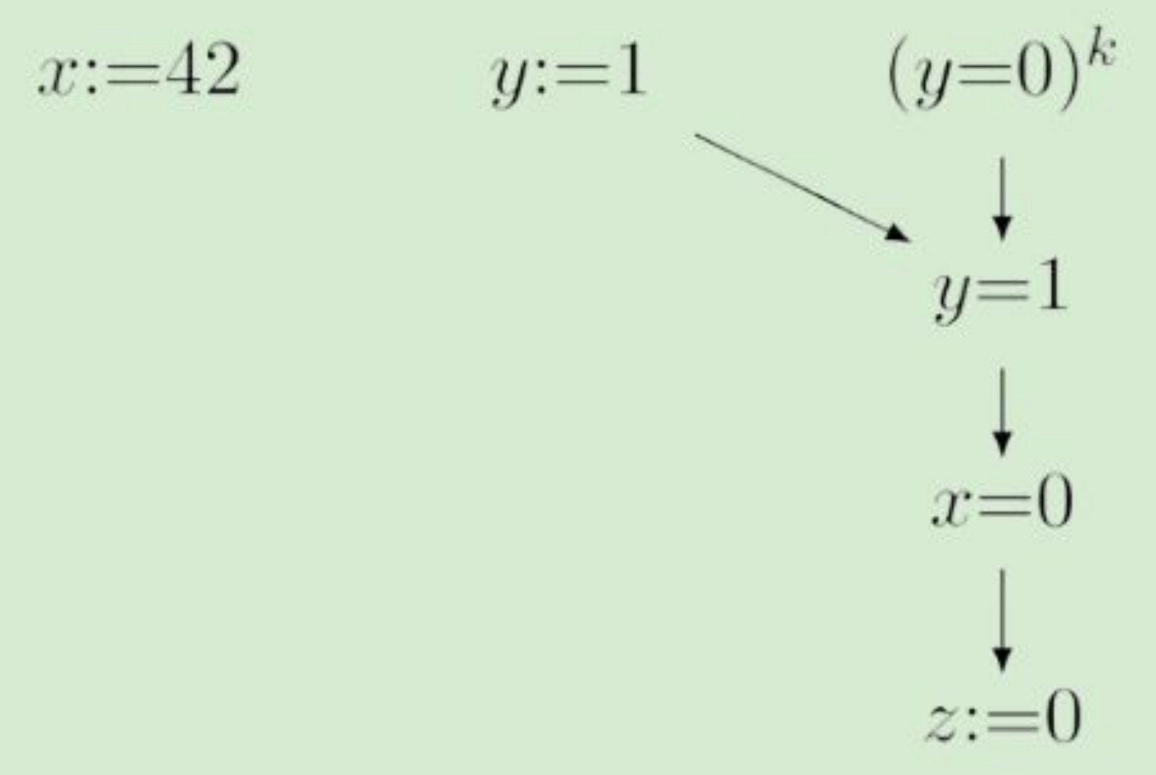
The upshot is that execution traces no longer suffice to reason about programs in these weaker memory models. To solve this issue, we have been developing a semantics using ***partially-ordered multisets*** (pomsets) of memory actions instead of traces. For example, consider the following message passing program:

$$(x := 42;\ y := 1) \| (\mathbf{while}\ y = 0\ \mathbf{do\ skip};\ z := x)$$

Under TSO, the pomsets of this program executable from the state $\{x : 0;\ y : 0;\ z : 0\}$ have the form:


$$\begin{tikzcd}
x{:=}42 \arrow[d] & (y{=}0)^k \arrow[d] \\
y{:=}1 \arrow[r] & y{=}1 \arrow[d] \\
 & x{=}42 \arrow[d] \\
 & z{:=}42
\end{tikzcd}$$

whereas under PSO, executable pomsets from this state include the below, which leaves $z = 0$ in the final state since the order of $x := 42$ and $y := 1$ is relaxed as they are writes to different variables.


$$\begin{tikzcd}
x{:=}42 & y{:=}1 \arrow[dr] & (y{=}0)^k \arrow[d] \\
 & & y{=}1 \arrow[d] \\
 & & x{=}0 \arrow[d] \\
 & & z{:=}0
\end{tikzcd}$$

Theoretical Results

We were able to show that CSL is still valid using these weaker memory models in the case of finite pomsets (which correspond to terminating programs). The techniques used for this proof constitute an adaptation of the ***Szpilrajn extension theorem***, which states that every partial order can be extended to a total order.
The core of this argument is to take pairs of memory actions in a pomset that must be ordered according to the memory model, and ordering them one by one until there are none left. This works in the case of a finite pomset because every time a new pairs is added to the ordering, the number of pairs that still need to be added decreases. However, with infinite pomsets, a more technical proof is required.