Formal Verification of *Pilot*

To avoid confusion, we rename the local variables in *Pilot*. Assuming that there are a sender and a receiver running concurrently and at the beginning, $data = oldData\_s \wedge data = oldData\_r \wedge flag = oldFlag\_r \wedge cnt\_s = cnt\_r$. Local variable $newData\_s$ is an argument. We will prove that the return value of the receiver equals the initial value of $newData\_s$.

Algorithm 1: *Pilot* Sender Side Implementation

Data: Shared: $flag = 0$, $data = 0$;
Local : $newData\_s$, $oldData\_s = 0$, $cnt\_s = 0$;
Const : $hashPool$;

$newData\_s \leftarrow newData\_s$ ^ $hashPool[cnt\_s$ ++ % SIZE];
if $newData\_s = oldData\_s$ **then**
  $flag \leftarrow flag$ ^ 1;
else
  $data \leftarrow newData\_s$;
  $oldData\_s \leftarrow newData\_s$;

Algorithm 2: *Pilot* Receiver Side Implementation

Data: Shared: $flag = 0$, $data = 0$;
Local : $oldFlag\_r = 0$, $oldData\_r = 0$, $cnt\_r = 0$;
Const : $hashPool$;

while $data = oldData\_r$ **do**
  if $flag \neq oldFlag\_r$ **then**
    $oldFlag\_r \leftarrow flag$;
    break;
$oldData\_r \leftarrow data$;
return $oldData\_r$ ^ *hashPool[cnt_r ++ % SIZE]*

Some techniques in formal verification from previous works [1–3] are used to prove the correctness of *Pilot*. Concurrency can be modeled as the interleavings of atomic operations, which generate a trace [2]. The intuitive way to prove correctness is to consider all possible execution results of interleavings. To reduce the number of interleavings that we must consider, we adopt mover types to prove certain operations are left- or right-commutative with respect to concurrent operations by other threads [1] [3].

Definition 1. An operation opA is a right-mover iff

$$
\begin{aligned}
&\forall opB\ \ state0\ \ state1\ \ state2\ \ tidA\ \ tidB.\\
&tidA \neq tidB \rightarrow\\
&exec\ \ opA\ \ tidA\ \ state0\ \ state1 \rightarrow\\
&exec\ \ opB\ \ tidB\ \ state1\ \ state2 \rightarrow\\
&\exists\ \ state'.\\
&exec\ \ opB\ \ tidB\ \ state0\ \ state' \wedge\\
&exec\ \ opA\ \ tidA\ \ state'\ \ state2
\end{aligned}
$$

Here, $exec$, opA, $tidA$, $state0$, and $state1$ denotes executing opA of thread $tidA$ changes the program state from $state0$ to $state1$.

The definition means opA is a right-mover iff executing opA firstly and opB secondly transforms the program state from $state0$ to $state2$ and executing opB firstly and opA secondly also transforms the program state from $state0$ to state2. So given an execution trace, moving opA to the right of other thread's operations gives the same execution result as that of the original trace. For example, since $newData\_s \leftarrow newData\_s$ ˆ $hashPool[cnt\_s$ ++ %SIZE] in the sender operates on constant and local variables, we can treat it to be atomic, which makes it a right-mover.

Theorem 1. $newData\_s \leftarrow newData\_s$ ˆ $hashPool[cnt\_s$ ++ % SIZE] in the sender is a right-mover.

Proof. According to the definition of right-mover, we must prove this operation of the sender is right-commutative with respect to all possible concurrent operations by the receiver thread. We use ops to denote $newData\_s \leftarrow newData\_s$ ˆ $hashPool[cnt\_s$ ++ %SIZE] and use opr to denote concurrent operation of the receiver. We use the form of {P}trace{Q} to denote that the execution trace transforms the program state from P to Q.

1. Opr≡ if data=oldData_r.

$$
\begin{aligned}
&\{\\
&newData\_s = v1 \wedge cnt\_s = v2 \wedge hashPool[v2\%SIZE] = v3 \wedge\\
&data = v4 \wedge oldData\_r = v5 \wedge data = oldData\_r \wedge data = oldData\_s\\
&\}\\
&newData\_s \leftarrow newData\_s\hat{}hashPool[cnt\_s++\%SIZE]\\
&if\ \ data = oldData\_r\\
&\{\\
&newData\_s = v1\hat{}v3 \wedge cnt\_s = v2+1 \wedge hashPool[v2\%SIZE] = v3 \wedge\\
&data = v4 \wedge oldData\_r = v5 \wedge data = oldData\_r \wedge data = oldData\_s\\
&\}
\end{aligned}
$$

Then if we reorder ops to the right of opr:

$$
\begin{aligned}
&\{\\
&newData\_s = v1 \wedge cnt\_s = v2 \wedge hashPool[v2\%SIZE] = v3 \wedge\\
&data = v4 \wedge oldData\_r = v5 \wedge data = oldData\_r \wedge data = oldData\_s\\
&\}\\
&if \quad data = oldData\_r;\\
&newData\_s \leftarrow newData\_s\hat{\ } hashPool[cnt\_s + +\%SIZE]\\
&\{\\
&newData\_s = v1\hat{\ }v3 \wedge cnt\_s = v2 + 1 \wedge hashPool[v2\%SIZE] = v3 \wedge\\
&data = v4 \wedge oldData\_r = v5 \wedge data = oldData\_r \wedge data = oldData\_s\\
&\}
\end{aligned}
$$

Reordering doesn't change the execution result. So ops is right-commutative with respect to opr.

2. Opr$\equiv$ if $flag \neq oldflag\_r$.

$$
\begin{aligned}
&\{\\
&newData\_s = v1 \wedge cnt\_s = v2 \wedge hashPool[v2\%SIZE] = v3 \wedge\\
&data = v4 \wedge oldData\_r = v5 \wedge data = oldData\_r \wedge data = oldData\_s \wedge\\
&flag = oldFlag\_r\\
&\}\\
&newData\_s \leftarrow newData\_s\hat{\ } hashPool[cnt\_s + +\%SIZE]\\
&if \quad flag \neq oldflag\_r\\
&\{\\
&newData\_s = v1\hat{\ }v3 \wedge cnt\_s = v2 + 1 \wedge hashPool[v2\%SIZE] = v3 \wedge\\
&data = v4 \wedge oldData\_r = v5 \wedge data = oldData\_r \wedge data = oldData\_s \wedge\\
&flag = oldFlag\_r\\
&\}
\end{aligned}
$$

Then if we reorder ops to the right of opr:

$$
\begin{aligned}
&\{\\
&newData\_s = v1 \wedge cnt\_s = v2 \wedge hashPool[v2\%SIZE] = v3 \wedge\\
&data = v4 \wedge oldData\_r = v5 \wedge data = oldData\_r \wedge data = oldData\_s \wedge\\
&flag = oldFlag\_r\\
&\}\\
&if \quad flag \neq oldflag\_r;\\
&newData\_s \leftarrow newData\_s\hat{\ } hashPool[cnt\_s + +\%SIZE]\\
&\{\\
&newData\_s = v1\hat{\ }v3 \wedge cnt\_s = v2 + 1 \wedge hashPool[v2\%SIZE] = v3 \wedge\\
&data = v4 \wedge oldData\_r = v5 \wedge data = oldData\_r \wedge data = oldData\_s \wedge\\
&flag = oldFlag\_r\\
&\}
\end{aligned}
$$

3. Opr≡ $oldFlag\_r \leftarrow flag$. The proof is similar to the above.

4. Opr≡ $oldData\_r \leftarrow data$. The proof is similar to the above.

5. Opr≡ return $oldData\_r\hat{}hashPool[cnt\_r++\%SIZE]$. The proof is similar to the above.

In conclusion, $newData\_s \leftarrow newData\_s$ ˆ $hashPool[cnt\_s$ ++ % SIZE] is right-commutative with respect to all concurrent operations by the receiver. So it's a right-mover. □

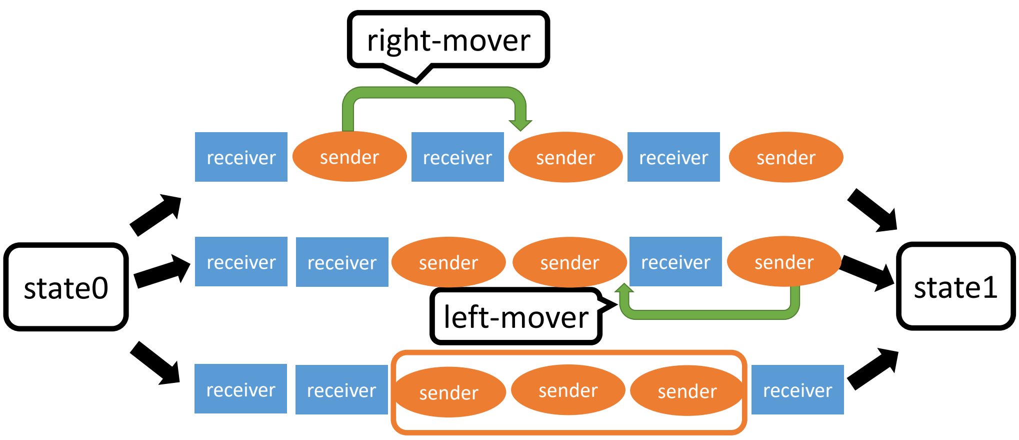
Intuitively, $newData\_s \leftarrow newData\_s$ ˆ $hashPool[cnt\_s$ ++ % SIZE] is a right-mover because it only operate on local variables and constants. Similarly, we can define left-mover.

Definition 2. An operation opA is a left-mover iff

$$\begin{aligned}
&\forall opB\ \ state0\ \ state1\ \ state2\ \ tidA\ \ tidB.\\
&tidA \neq tidB \rightarrow\\
&exec\ \ opB\ \ tidB\ \ state0\ \ state1 \rightarrow\\
&exec\ \ opA\ \ tidA\ \ state1\ \ state2 \rightarrow\\
&\exists\ \ state'.\\
&exec\ \ opA\ \ tidA\ \ state0\ \ state' \wedge\\
&exec\ \ opB\ \ tidB\ \ state'\ \ state2
\end{aligned}$$

The definition means opA is a left-mover iff executing opB firstly and opA secondly transforms the program state from $state0$ to $state2$ and executing opA firstly and opB secondly also transforms the program state from $state0$ to state2. So given an execution trace, moving opA to the left of other thread's operations gives the same execution result as that of the original trace. For example, $oldData\_s \leftarrow newData\_s$ in the sender is a left-mover. The proof is similar to that of theorem 1.

It is clear that operations on line 1 and line 2 of the sender are right-movers, and operations on line 6 is a left-mover. Because they only operate on local variables and constants. Operations on line 3 and line 5 cannot be moved, but only one of them can exist in a legal trace. So given an execution trace, we can repeatedly move operations on line 1 and line 2 of the sender to the right and move operations on line 6 of the sender to the left. Finally, the trace looks like that the execution of the sender thread is sequential.



Therefore, we can treat the sender program to be atomic, which reduces the number of interleaving we must consider. With the help of mover, the problem is simplified into proving the correctness of sequential execution trace.

Theorem 2. After running the sender and the receiver, the return value of the receiver equals to the initial value of $newData\_s$ in the sender.

Proof. The sender program can be treated to be atomic, we consider it to be a single operation and consider all possible interleaving. Because only one of $flag \leftarrow flag\hat{}1$ and $data \leftarrow newData\_s; oldData\_s \leftarrow newData\_s$ can exist in the execution trace, we use classification according to it to prove. Before running the concurrent programs, $newData\_s = v1$, $hashPool[cnt\_s{+}{+}\%SIZE] = v2$, $data = v3$, $oldData\_r = v3$, $oldData\_s = v3$.

Firstly, we consider $flag \leftarrow flag\hat{}1$ exists in the trace. In this case, "if" condition in the sender is true and v1ˆv2 = v3.

1. Sender is executed before line 1 in the receiver. Then "while" and "if" condition in the receiver are true. Thus, the return value of the receiver = v3ˆv2 = v1ˆv2ˆv2 = v1 = initial value of $newData\_s$ in the sender.

2. Sender is executed between line 1 and line 2 in the receiver. Then "while" and "if" condition in the receiver are true. Thus, the return value of the receiver = v3ˆv2 = v1ˆv2ˆv2 = v1 = initial value of $newData\_s$ in the sender.

3. Sender is executed between line 2 and line 3 in the receiver. It is impossible because "if" condition in the receiver is false.

4. Sender is executed between line 3 and line 4 in the receiver. It is impossible because "if" condition in the receiver is false.

5. Sender is executed exactly before line 5 in the receiver. It is impossible because "if" condition in the receiver is false and "while" condition in the receiver is true.

Secondly, we consider $data \leftarrow newData\_s; oldData\_s \leftarrow newData\_s$ exists in the trace. In this case, "if" condition in the sender is false. v1ˆv2 $\neq$ v3.

1. Sender is executed before line 1 in the receiver. Then "while" in the receiver is false. The return value of the receiver = v1 ˆv2 ˆv2 = v1 = initial value of $newData\_s$ in the sender.

2. The sender is executed between line 1 and line 2 in the receiver. Then "while" condition in the receiver is true and "if" condition in the receiver is false. So "while" iterates again and "while" condition in the receiver becomes false. Thus, the return value of the receiver = v1 ˆv2 ˆv2 = v1 = initial value of $newData\_s$ in the sender.

3. Sender is executed between line 2 and line 3 in the receiver. It is impossible because "if" condition in the receiver is false.

4. Sender is executed between line 3 and line 4 in the receiver. It is impossible because "if" condition in the receiver is false.

5. Sender is executed exactly before line 5 in the receiver. It is impossible because "if" condition in the receiver is false and "while" condition in the receiver is true.

In conclusion, after running the sender and the receiver, the return value of the receiver equals to the initial value of $newData\_s$ in the sender in both cases.

□

References

[1] Tej Chajed, Frans Kaashoek, Butler Lampson, and Nickolai Zeldovich. Verifying concurrent software using movers in *cspec*. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 306–322, 2018.

[2] Leslie Lamport. What good is temporal logic? In *IFIP congress*, volume 83, pages 657–668, 1983.

[3] Richard J Lipton. Reduction: A method of proving properties of parallel programs. *Communications of the ACM*, 18(12):717–721, 1975.