Proving Lock-Freedom Easily

Wei Li Xiao Jia Viktor Vafeiadis

March 26, 2014

Abstract

We present a new way for proving lock-freedom automatically.

1 Introduction

A non-blocking data structure should guarantee certain *liveness properties*, which are ensured by progress conditions such as obstruction-freedom, lock-freedom, and wait-freedom. The strongest of these properties is wait-freedom, which requires each thread to make progress whenever it is scheduled for a sufficient number of steps, independently of other concurrently executing threads. Such behavior is indeed desirable, but this requirement is very strong and often results in a complicated and inefficient programs. Lock-freedom requires that when the program threads are run sufficiently long, at least one of the threads make progress. This requirement ensures that the program as a whole makes progress and is never blocked. Lock-free algorithms are known for various tasks and are often efficient. Obstruction freedom is the weakest requirement, stating that progress is only guaranteed if we let one of the program threads run in isolation sufficiently long. In this paper we concentrate on lock-freedom, which provides a strong progress guarantee and allows for efficient implementations in practice.

We use Treiber's stack example to illustrate the method that can prove lock-free easily. A shared ghost variable p is introduced to the original implemnentation of Treiber's stack, which will be incremented after each *break*, as is shown in Figure 1. Without the instructions in red, it is a full implementation of Treiber's stack. To prove the lock-freedom of the program, we can first reduce the lock-freedom to termination, which has been proved by Gotsman et al[1]. If there's no assertion violations, the new program has the same termination property with the original one. So we will prove the termination of the new program and no assertions violations existed.

Consider a non-blocking data structure with operations $op_1, op_2, \ldots, op_n$. Let op be the command that nondeterministically executes one of the operations on the data structure with arbitrary parameters:

$$op = if\ (nondet())\ op_1;\ else\ if\ (nondet())\ op_2; \ldots\ else\ op_n; \qquad (1)$$

```
struct Node {
    value_t data;
    Node *next;
};
Node *s;

void push(value_t v) {
    Node *t, *x;
    x = new Node();
    x->data = v;
    loop {
        value_t p0 = p;
        t = s;
        x->next = t;
        atomic {
            assume (s == t) {
                s = x;
                p++;
                break;
            }
        }
        assert(p0 < p);
    }
}
```

```
void init() {
    s = NULL;
}

void pop() {
    Node *t, *x;
    loop {
        value_t p0 = p;
        t = s;
        assume (t == 0) break;
        x = t->next;
        atomic {
            assume(s == t) {
                s = x;
                p++;
                break;
            }
        }
        assert(p0 < p);
    }
}
```

Figure 1: Treiber's non-blocking stack with ghost variable p

We denote non-deterministic choice with $nondet()$. The definition of lock-freedom of the data structure requires that for all k the following program $C(k)$ terminates:

$$C(k) = \|_{i=1}^{k} \, op \tag{2}$$

2 Background

2.1 Programming Language

We use a basic programming language with concurrency. Commands C are given by the abstract syntax

$$\begin{aligned} cmd,\ C \ ::= \ & \mathbf{skip} \mid x = E \mid \mathbf{assume}\, B \mid C_1; C_2 \mid \mathbf{atomic}\, C \mid \mathbf{loop}\, C \mid \mathbf{inloop}^{p}_{C_2}(C_1) \\ & \mid \mathbf{break} \mid C_1 \parallel C_2 \mid C_1 \oplus C_2 \end{aligned}$$

where E ranges over arithmetic expressions, including non-deterministic choice $nondet()$, and B expresse s all the boolean expressions. The command *atomic* C executes C in one indivisible step.

2.2 Semantics

Formulas and programs are interpreted with respect to a *program state* using a small-step operational semantics.

The rules of the semantics are defined as follows. They define an evaluation judgment of the forms

$$\langle C, \sigma, p \rangle \rightarrow \langle C', \sigma', p' \rangle \quad \text{or} \quad \langle C, \sigma, p \rangle \rightarrow abort$$

where C and C' are commands and $\sigma, \sigma' \in State$. Since our logic includes a shared ghost variable to track the progress of threads, the configuration consists not only of program states, commands, but also of a natural number p initialized with 0. The judgment states the following. If we execute the command C in the state σ with p then it transforms the program state to σ' with p' and excution continues with command C', or the next computation step results in an error ($\langle C, \sigma, p\rangle \rightarrow abort$).

The $inside\_loop(C_1, C)$ defines the commands inside the loop C, where C_1 represents the commands left in the loop after executing some part of loop, and C represents the whole loop body. $inside\_loop$ is defined inductively, which means

$$inside\_loop(loop\ C_1, C) \rightarrow inside\_loop(inside\_loop(C_1, C_1), C).$$

$\boxed{\mathbf{aborts}\langle C, \sigma, p\rangle}$

$$\begin{aligned}
\mathbf{aborts}\langle C_1; C_2, \sigma, p\rangle &\equiv \mathbf{aborts}\langle C_1, \sigma, p\rangle \\
\mathbf{aborts}\langle C_1 \parallel C_2, \sigma, p\rangle &\equiv \mathbf{aborts}\langle C_1, \sigma, p\rangle \vee \mathbf{aborts}\langle C_2, \sigma, p\rangle \\
\mathbf{aborts}\langle \mathbf{inloop}_C^{p'}(\mathbf{skip}), \sigma, p\rangle &\equiv p' < p \\
\mathbf{aborts}\langle C, \sigma, p\rangle &\equiv \mathbf{False} \\
\mathbf{aborts}\langle \mathbf{atomic}\, C, \sigma, p\rangle &\equiv \mathbf{aborts}\langle C, \sigma, p\rangle
\end{aligned}$$

$$\frac{}{\langle x = E, \sigma, p\rangle \rightarrow \langle \mathbf{skip}, \sigma[x := [\![E]\!]_\sigma], p+1\rangle} \qquad \frac{\langle C_1, \sigma, p\rangle \rightarrow \langle C_1', \sigma', p'\rangle}{\langle C_1; C_2, \sigma, p\rangle \rightarrow \langle C_1'; C_2, \sigma', p'\rangle}$$

$$\frac{}{\langle \mathbf{skip}; C_2, \sigma, p\rangle \rightarrow \langle C_2, \sigma, p\rangle} \qquad \frac{[\![B]\!]_\sigma}{\langle \mathbf{assume}\, B, \sigma, p\rangle \rightarrow \langle \mathbf{skip}, \sigma, p\rangle}$$

$$\frac{\langle C_1, \sigma, p\rangle \rightarrow \langle C_1', \sigma', p'\rangle}{\langle C_1 \parallel C_2, \sigma, p\rangle \rightarrow \langle C_1' \parallel C_2, \sigma', p'\rangle} \qquad \frac{\langle C_2, \sigma, p\rangle \rightarrow \langle C_2', \sigma', p'\rangle}{\langle C_1 \parallel C_2, \sigma, p\rangle \rightarrow \langle C_1 \parallel C_2', \sigma', p'\rangle}$$

$$\frac{}{\langle \mathbf{skip} \parallel \mathbf{skip}, \sigma, p\rangle \rightarrow \langle \mathbf{skip}, \sigma, p\rangle} \qquad \frac{}{\langle \mathbf{loop}\, C, \sigma, p\rangle \rightarrow \langle \mathbf{inloop}_C^{p}(C), \sigma, p\rangle}$$

$$\frac{\langle C_1, \sigma, p\rangle \rightarrow \langle C_1', \sigma', p'\rangle}{\langle \mathbf{inloop}_C^{p_0}(C_1), \sigma, p\rangle \rightarrow \langle \mathbf{inloop}_C^{p_0}(C_1'), \sigma, p\rangle}$$

$$\frac{}{\langle \mathbf{inloop}_C^{p_0}(\mathbf{break}), \sigma, p\rangle \rightarrow \langle \mathbf{skip}, \sigma, p+1\rangle}$$

$$\frac{}{\langle \mathbf{inloop}_C^{p_0}(\mathbf{break}; C_1), \sigma, p\rangle \rightarrow \langle \mathbf{skip}, \sigma, p+1\rangle}$$

$$\frac{p_0 < p}{\langle \mathbf{inloop}_C^{p_0}(\mathbf{skip}), \sigma, p\rangle \rightarrow \langle \mathbf{inloop}_C^{p}(C), \sigma, p\rangle}$$

$$\frac{\langle C, \sigma, p\rangle \to \langle \textbf{skip}, \sigma', p'\rangle}{\langle \textbf{atomic}\ C, \sigma, p\rangle \to \langle \textbf{skip}, \sigma', p'\rangle} \qquad \frac{}{\langle C_1 \oplus C_2, \sigma, p\rangle \to \langle C_1, \sigma, p\rangle}$$

$$\frac{}{\langle C_1 \oplus C_2, \sigma, p\rangle \to \langle C_2, \sigma, p\rangle}$$

$$\frac{}{\langle x = E, \sigma, p\rangle \to \langle skip, \sigma[x \mapsto [\![E]\!]\sigma], p+1\rangle} \quad (\text{ASSIGN})$$

$$\frac{\langle C_1, \sigma, p\rangle \to \langle C_1', \sigma', p'\rangle}{\langle C_1; C_2, \sigma, p\rangle \to \langle C_1'; C_2, \sigma', p'\rangle} \quad (\text{SEQ1})$$

$$\frac{}{\langle skip; C_2, \sigma, p\rangle \to \langle C_2, \sigma, p\rangle} \quad (\text{SEQ2})$$

$$\frac{\langle C_1, \sigma, p\rangle \to abort}{\langle C_1; C_2, \sigma, p\rangle \to abort} \quad (\text{SEQABORT})$$

$$\frac{\langle C_1, \sigma, p\rangle \to \langle C_1', \sigma', p'\rangle}{\langle C_1 \parallel C_2, \sigma, p\rangle \to \langle C_1' \parallel C_2, \sigma', p'\rangle} \quad (\text{PAR1})$$

$$\frac{\langle C_2, \sigma, p\rangle \to \langle C_2', \sigma', p'\rangle}{\langle C_1 \parallel C_2, \sigma, p\rangle \to \langle C_1 \parallel C_2', \sigma', p'\rangle} \quad (\text{PAR2})$$

$$\frac{}{\langle skip \parallel skip, \sigma, p\rangle \to \langle skip, \sigma, p\rangle} \quad (\text{PAR3})$$

$$\frac{\langle C_1, \sigma, p\rangle \to abort}{\langle C_1 \parallel C_2, \sigma, p\rangle \to abort} \quad (\text{PARABORT1})$$

$$\frac{\langle C_2, \sigma, p\rangle \to abort}{\langle C_1 \parallel C_2, \sigma, p\rangle \to abort} \quad (\text{PARABORT2})$$

$$\frac{[\![B]\!]\sigma}{\langle assume(B), \sigma, p\rangle \to \langle skip, \sigma, p\rangle} \quad (\text{ASSUME})$$

$$\frac{}{\langle loop\ C, \sigma, p\rangle \to \langle inside\_loop(C, C, p), \sigma, p\rangle} \quad (\text{LOOPENTER})$$

$$\frac{\langle C_1, \sigma, p\rangle \to \langle C_1', \sigma', p'\rangle}{\langle inside\_loop(C_1, C, p_0), \sigma, p\rangle \to \langle inside\_loop(C_1', C, p_0), \sigma', p\rangle} \quad (\text{LOOPSEQ})$$

$$\frac{}{\langle inside\_loop(break, C, p_0), \sigma, p\rangle \to \langle skip, \sigma, p+1\rangle} \quad (\text{LOOPBREAK1})$$

$$\frac{}{\langle inside\_loop(break; C_1, C, p_0), \sigma, p\rangle \to \langle skip, \sigma, p+1\rangle} \quad (\text{LOOPBREAK2})$$

$$\frac{p_0 < p}{\langle \mathit{inside\_loop}(\mathit{skip}, C, p_0), \sigma, p\rangle \rightarrow \langle \mathit{inside\_loop}(C, C, p), \sigma, p\rangle} \quad (\textsc{LoopAssert1})$$

$$\frac{p_0 \geq p}{\langle \mathit{inside\_loop}(\mathit{skip}, C, p_0), \sigma, p\rangle \rightarrow \mathit{abort}} \quad (\textsc{LoopAssert2})$$

$$\frac{\langle C_1, \sigma, p\rangle \rightarrow \mathit{abort}}{\langle \mathit{inside\_loop}(C_1, C, p_0), \sigma, p\rangle \rightarrow \mathit{abort}} \quad (\textsc{LoopAbort})$$

$$\frac{\langle C, \sigma, p\rangle \rightarrow^* \langle \mathit{skip}, \sigma', p'\rangle}{\langle \mathit{atomic}\ C, \sigma, p\rangle \rightarrow \langle \mathit{skip}, \sigma', p'\rangle} \quad (\textsc{Atom})$$

$$\frac{\langle C, \sigma, p\rangle \rightarrow^* \mathit{abort}}{\langle \mathit{atomic}\ C, \sigma, p\rangle \rightarrow \mathit{abort}} \quad (\textsc{AtomAbort})$$

$$\frac{}{\langle C_1 \oplus C_2, \sigma, p\rangle \rightarrow \langle C_1, \sigma, p\rangle} \quad (\textsc{Nondet1})$$

$$\frac{}{\langle C_1 \oplus C_2, \sigma, p\rangle \rightarrow \langle C_2, \sigma, p\rangle} \quad (\textsc{Nondet2})$$

Definition 1 (Size of Commands)**.** *Let C be a command. $|C|$ is the size of command C and is inductively defined as follows.*

$$\begin{aligned}
|\mathit{skip}| &= 0 \\
|C_1; C_2| &= |C_1| + |C_2| \\
|C_1 \parallel C_2| &= |C_1| + |C_2| \\
|C_1 \oplus C_2| &= \mathit{max}(|C_1|, |C_2|) \\
|\mathit{atomic}\ C| &= |C| \\
|\mathit{assume}(B)| &= 1 \\
|\mathit{inside\_loop}(C1, C)| &= 0 \\
|\mathit{loop}\ C| &= 1 \\
|x = E| &= 1
\end{aligned}$$

In the definition of size of commands, we only defined the size of loop to be 1 and didn't define the size of the commands inside the loop. We define the size of the commands inside the loop separately.

Definition 2. *We define $\mathit{sil}(C)$ to be the size of command inside the loop and*

$$\mathit{sil}(\mathit{inside\_loop}(C_1, C, p_0)) = |C_1|$$

Definition 3 (Progress Order)**.** *Let C, C' be commands, σ, σ' be program states and let p, p' be the value of the shared ghost variable. We define $(C', \sigma', p') \prec (C, \sigma, p) \iff p' > p \vee (p' = p \wedge |C'| < |C|) \vee (p' = p \wedge |C'| = |C| \wedge \mathit{sil}(C') < \mathit{sil}(C))$.*

Lemma 2.1. *Let C, C' be commands. If $\langle C, \sigma, p\rangle \to \langle C', \sigma', p'\rangle$, then $p + |C| = p' + |C'|$.*

Proof. By inspection of the mechanism where we increment the shared variable p. Outside the loop, p is incremented in each assignment step, where the size of the command is 1. And inside the loop p will only be incremented when the loop is successfully breaked. □

Theorem 2.2. *The shared ghost variable p, which represents the success of loops and tracks the progress of threads, has an upper bound.*

Proof. Let C_0 be the whole program commands that haven't been executed with $p = 0$. From Lemma 2.1 we can derive that $0 + |C_0| = p' + |C'|$. Since p is always incremented, inspected from the operational semantics rules, and $|C'| \geq 0$, p has an upper bound $|C_0|$. □

Lemma 2.3. *If $\langle C, \sigma, p\rangle \to \langle C', \sigma', p'\rangle$ then $(C', \sigma', p') \prec (C, \sigma, p)$*

Proof. By inspection of the operational semantics rules. □

As a consequence of Lemma 2.3 and Theorem 2.2, there are no infinite chains of the form $\langle C_1, \sigma_1, p_1\rangle \to \langle C_2, \sigma_2, p_2\rangle \to \cdots$.

Theorem 2.4. *There exists no infinite chains of the form $\langle C_1, \sigma_1, p_1\rangle \to \langle C_2, \sigma_2, p_2\rangle \to \cdots$.*

The operational semantics shows that each terminal state has the form $\langle skip, \sigma, p\rangle$.

Definition 4 (Termination)**.** *We define a program C terminates from an initial state σ if not $\langle C, \sigma, p\rangle \to^* abort$.*

Since there is no infinite evaluation chains, if the program doesn't abort, it will terminate.

2.3 Specifying Lock-Freedom

3 Evaluation

4 Related Work

Gotsman et al. [1] present an automatic approach for verifying lock-freedom by reducing the lock-freedom problem to termination.

Hoffmann et al. [2] show that Gotsman et al's reduction is incorrect for implementations using thread identifiers or thread-local state. They present an alternative reduction to termination that is correct for such implementations. Also, They propose a program logic to verify lock-freedom of concurrent objects. They reason about termination quantitatively by introducing tokens, and model the environment's interference over the current thread's termination in terms of token transfer. The idea is simple and natural, but their logic has very limited support of local reasoning. The method requires the automatic generation of loop invariants and resource invariants. One needs to know the total number of tokens needed by each thread(which may have multiple while loops) and the (fixed) number of threads, to calculate the number of tokens for a thread to lose or initially own. This requirement also disallows their logic to reason about programs with infinite nondeterminism.

5 Conclusion

We have presented a much simpler way for proving lock-freedom.

References

[1] Alexey Gotsman, Byron Cook, Matthew Parkinson, and Viktor Vafeiadis. Proving that non-blocking algorithms don't block. In *ACM SIGPLAN Notices*, volume 44, pages 16–28. ACM, 2009.

[2] Jan Hoffmann, Michael Marmar, and Zhong Shao. Quantitative reasoning for proving lock-freedom. In *Logic in Computer Science (LICS), 2013 28th Annual IEEE/ACM Symposium on*, pages 124–133. IEEE, 2013.