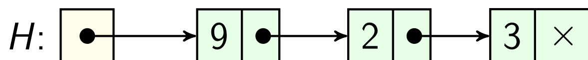
Proving Lock-Freedom Easily and Automatically

Xiao Jia[1] Wei Li[1] Viktor Vafeiadis[2]

[1] Shanghai Jiao Tong University
[2] Max Planck Institute for Software Systems (MPI-SWS)

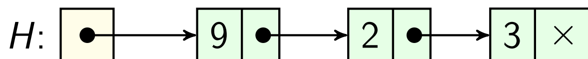
CPP'15, 14 January 2015

Blocking synchronisation



$$
\begin{array}{l}
n := \text{malloc}(\ldots); \\
n{\rightarrow}val := 7; \\
\text{lock}(); \\
\quad t := *H; \\
\quad n{\rightarrow}next := t; \\
\quad *H := n; \\
\text{unlock}();
\end{array}
$$

Non-blocking synchronisation



n := malloc(...);
$n \rightarrow val$:= 7;
while(true) {
    $t := *H$;
    $n \rightarrow next := t$;
    if($CAS(H, t, n)$) **break**;
}

What is lock-freedom?

Liveness property for concurrent libraries:

- If library operations are scheduled infinitely often, then *some* operation will terminate.

Observations:

- LF is a global property.
- LF allows starvation.
- wait-free $\subseteq$ lock-free $\subseteq$ obstruction-free
- LF is a practical compromise.

Proving lock-freedom is low-hanging fruit:

- Typically lock-freedom is really easy to show.
- But formal LF proofs are often too complex.
- Automation is not readily available.

- Concurrent library, $L = \langle C_{\text{init}}, C_1, \ldots, C_n \rangle$.
- Bounded most general client, $\text{BMGC}^{k,m}(L)$, runs C_{init} and then creates k threads executing up to m times a method from $C_1, \ldots, C_n$.

Theorem (Hoffmann, Marmar, Shao, LICS'13)

L is lock-free $\iff \forall k, m.\ \text{BMGC}^{k,m}(L)$ terminates.

Read-compute-update pattern

A common pattern in LF algorithms:

$$
\begin{array}{l}
\textbf{while}(\text{true})\ \{ \\
\quad t := X; \\
\quad \ldots; \\
\quad n := \ldots; \\
\quad \textbf{if}\ (\text{CAS}(\&X, t, n))\ \textbf{break}; \\
\}
\end{array}
$$

Read-compute-update pattern

A common pattern in LF algorithms:

$$
\begin{array}{l}
\textbf{while}(\text{true})\ \{ \\
\quad t := X; \\
\quad \ldots; \\
\quad n := \ldots; \\
\quad \textbf{if}\ (X == t)\ \{X := n;\ \textbf{break};\} \\
\}
\end{array}
$$

Verifying the read-compute-update pattern

Using a global progress counter:

$$
\begin{array}{l}
\textbf{while}(\text{true})\ \{ \\
\quad r := P; \\
\quad t := X; \\
\quad \ldots; \\
\quad n := \ldots; \\
\quad \textbf{if}\ (X == t)\ \{\langle X := n; ++P\rangle;\ \textbf{break};\} \\
\quad \textbf{assert}(P > r); \\
\}
\end{array}
$$

Using a progress flag per thread:

$$
\begin{array}{l}
\textbf{while}(\mathrm{true})\ \{ \\
\quad B_{tid} := \mathrm{false}; \\
\quad t := X; \\
\quad \ldots; \\
\quad n := \ldots; \\
\quad \textbf{if}\ (X == t)\ \{\langle X := n; \forall i.\ B_i := \mathrm{true}\rangle;\ \textbf{break}; \} \\
\quad \textbf{assert}(B_{tid}); \\
\}
\end{array}
$$

Verifying the read-compute-update pattern

Using a progress flag per thread:

$$
\begin{array}{l}
\textbf{while}(\text{true})\ \{ \\
\quad B_{tid} := *; \\
\quad t := X; \\
\quad \ldots; \\
\quad n := \ldots; \\
\quad \textbf{if}\ (X == t)\ \{\langle X := n; \forall i.\ B_i := \text{true}\rangle;\ \textbf{break};\} \\
\quad \textbf{assert}(B_{tid}); \\
\}
\end{array}
$$

Verifying the read-compute-update pattern

Using a progress flag per thread:

$$
\begin{array}{l}
\textbf{while}(\text{true})\ \{ \\
\quad B_{tid} := *; \\
\quad t := X; \\
\quad \ldots; \\
\quad n := \ldots; \\
\quad \textbf{if}\ (X == t)\ \{\langle X := n; \textbf{assume}(\forall i.\ B_i)\rangle;\ \textbf{break};\} \\
\quad \textbf{assert}(B_{tid}); \\
\}
\end{array}
$$

What else is there?

Two important extensions:

- Nested loops
- Loops that terminate for sequential reasons

What else:

- Implementation/evaluation
- Formalisation in Ott & Coq

Handling nested loops

```
while(true) {
    t1 := X1; ...;
    if(X1 == t1) { X1 := ...; break; }
    ...;
    while(true) {
        t2 := X2; ...;
        if(X2 == t2) { X2 := ...; break; }
        ...;
    }
    ...;
}
```

Handling nested loops

$$
\begin{array}{l}
\textbf{while}(\text{true})\ \{r_1 := P_1; \\
\quad \langle t_1 := X_1; {+}{+}P_2\rangle; \ldots; \\
\quad \textbf{if}(X_1 == t_1)\ \{\ \langle X_1 := \ldots; {+}{+}P_1; {+}{+}P_2\rangle; \textbf{break}; \} \\
\quad {+}{+}P_2; \ldots; \\
\quad \textbf{while}(\text{true})\ \{r_2 := P_2; \\
\qquad t_2 := X_2; \ldots; \\
\qquad \textbf{if}(X_2 == t_2)\ \{\ \langle X_2 := \ldots; {+}{+}P_2\rangle; \textbf{break}; \} \\
\qquad \ldots; \textbf{assert}(P_2 > r_2); \\
\quad \} \\
\quad {+}{+}P_2; \ldots; \textbf{assert}(P_1 > r_1); \\
\}
\end{array}
$$

Termination for sequential reasons

How about the following program?

$$i := 0;$$
$$\text{while}(i < 2)\ \{i{+}{+};\}$$

Termination for sequential reasons

How about the following program?

$$i := 0;$$
$$\text{while}(i < 2)\ \{i{+}{+};\}$$

Naive instrumented version:

$$\langle i := 0; {+}{+}P_1\rangle;$$
$$\mathbf{L}_0(\text{skip},\ p := P_1;$$
$$\text{if}(i < 2)\ i{+}{+};\ \text{else break};$$
$$\text{assert}(P_1 > p);\ )$$

Oops!

Termination for sequential reasons

How about the following program?

$$i := 0;$$
$$\text{while}(i < 2)\ \{i{+}{+};\}$$

Why does it terminate?

- Ranking function, $f(i) = 2 - i$.
- Decreases round each loop iteration.
- Loop exits if $f(i) < 0$.

Idea: use the ranking function in the instrumentation!

Termination for sequential reasons

How about the following program?

$$i := 0;$$
$$\text{while}(i < 2)\ \{i{+}{+};\}$$

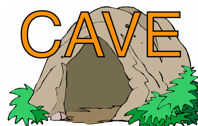
Better instrumented version:

$$\langle i := 0; {+}{+}P_1 \rangle;$$
$$\mathbf{L}_0(\text{skip},\ p := P_1; i_{\text{saved}} := i;$$
$$\text{if}(i < 2)\ i{+}{+};\ \text{else break};$$
$$\text{assert}(P_1 > p \lor 2 - i < 2 - i_{\text{saved}});\ )$$

Now provable.

Experimental results

Algorithm	LOC	Time
CAS counter	41	0.02s
Double counter	64	0.11s
Treiber stack	52	0.11s
DCAS stack	52	0.11s
Elimination stack	76	1.22s
MS queue	83	3.01s
DGLM queue	83	3.92s
MSV queue	74	2.85s



Formal set up

Minimal programming language:

$$C ::= \mathsf{skip} \mid \mathsf{break} \mid BC \mid C_1; C_2 \mid C_1 \parallel C_2 \mid C_1 \oplus C_2 \mid \mathbf{L}_n(C_1, C_2)$$

Standard operational semantics:

$$\langle C, \sigma \rangle \to \langle C', \sigma' \rangle$$

Instrumented semantics:

$$d \vdash \langle C, \sigma, P \rangle \to \langle C', \sigma', P' \rangle$$

$$d \vdash \langle C, \sigma, P \rangle \to \mathbf{abort}$$

Operational semantics: standard rules

$$\frac{(\sigma, \sigma') \in [\![BC]\!]}{d \vdash \langle BC, \sigma, P\rangle \rightarrow \langle \mathsf{skip}, \sigma', \mathsf{Cinc}(P, d)\rangle}$$

$$\frac{d \vdash \langle C_1, \sigma, P\rangle \rightarrow \langle C_1', \sigma', P'\rangle}{d \vdash \langle C_1; C_2, \sigma, P\rangle \rightarrow \langle C_1'; C_2, \sigma', P'\rangle} \quad \text{etc.}$$

where

$$\mathsf{Cinc}(P, d) \stackrel{\text{def}}{=} \lambda x. \begin{cases} P(x) & \text{if } x \leq d \\ P(x) + 1 & \text{if } x > d \end{cases}$$

Operational semantics: loops

$$\frac{}{d \vdash \langle \mathbf{L}_n(\mathsf{skip}, C), \sigma, P\rangle \rightarrow \langle \mathbf{L}_{P(d)}(C, C), \sigma, P\rangle}$$

$$\frac{P(d) \leq n}{d \vdash \langle \mathbf{L}_n(\mathsf{skip}, C), \sigma, P\rangle \rightarrow \mathbf{abort}}$$

$$\frac{d + \mathsf{ldinc}(C_1) \vdash \langle C_1, \sigma, P\rangle \rightarrow \langle C_1', \sigma', P'\rangle}{d \vdash \langle \mathbf{L}_n(C_1, C_2), \sigma, P\rangle \rightarrow \langle \mathbf{L}_n(C_1', C_2), \sigma', P'\rangle}$$

where

$$\mathsf{ldinc}(C) \stackrel{\text{def}}{=} \begin{cases} 0 & \text{if } \mathsf{lpExit}(C) \\ 1 & \text{if } \neg\mathsf{lpExit}(C) \end{cases}$$

Results

- Define configuration size, $\llparenthesis C, P \rrparenthesis^{d} : \mathbb{N} \to \mathbb{N}$.
- $\prec_k$: lexicographic order on first k elems.

Lemma (Progress)

If $d \vdash \langle C, \sigma, P \rangle \to \langle C', \sigma', P' \rangle$ *and* $d \vdash \langle C, \sigma, P \rangle \not\to$ **abort**, *then* $\llparenthesis C', P' \rrparenthesis^{d} \prec_{d + \llparenthesis C \rrparenthesis_{\text{depth}}} \llparenthesis C, P \rrparenthesis^{d}$.

Theorem (Termination)

If $0 \vdash \langle C, \sigma, \lambda x.\ 1 \rangle \not\to^{*}$ **abort**, *then* $\langle C, \sigma \rangle$ *always terminates.*

Termination metric: the size of a configuration

$$
\begin{aligned}
\langle\!|\mathsf{skip}, P|\!\rangle^{d}(m) &\stackrel{\text{def}}{=} 0 \\
\langle\!|\mathsf{break}, P|\!\rangle^{d}(m) &\stackrel{\text{def}}{=} 1 \\
\langle\!|BC, P|\!\rangle^{d}(m) &\stackrel{\text{def}}{=} 1 \\
\langle\!|C_1; C_2, P|\!\rangle^{d}(m) &\stackrel{\text{def}}{=} \begin{cases} \langle\!|C_1, P|\!\rangle^{d}(m) + 1 & \text{if } \mathsf{lpExit}(C_1) \\ \langle\!|C_1, P|\!\rangle^{d}(m) + \langle\!|C_2, P|\!\rangle^{d}(m) + 1 & \text{if } \neg\mathsf{lpExit}(C_1) \end{cases} \\
\langle\!|\mathbf{L}_n(C_1, C_2), P|\!\rangle^{d}(m) &\stackrel{\text{def}}{=} \begin{cases} \langle\!|C_1, P|\!\rangle^{d}(m) & \text{if } \mathsf{lpExit}(C_1) \\ \langle\!|C_2, P|\!\rangle^{d+1}(m) & \text{else if } m < d \\ \langle\!|C_1, P|\!\rangle^{d+1}(m) + \langle\!|C_2, P|\!\rangle^{d+1}(m) + \langle\!|C_2, P|\!\rangle^{d+1}(m) + 1 & \text{else if } n < P(d) \\ \langle\!|C_1, P|\!\rangle^{d+1}(m) + \langle\!|C_2, P|\!\rangle^{d+1}(m) & \text{otherwise} \end{cases} \\
\langle\!|C_1 \parallel C_2, P|\!\rangle^{d}(m) &\stackrel{\text{def}}{=} \langle\!|C_1, P|\!\rangle^{d}(m) + \langle\!|C_2, P|\!\rangle^{d}(m) + 1 \\
\langle\!|C_1 \oplus C_2, P|\!\rangle^{d}(m) &\stackrel{\text{def}}{=} \langle\!|C_1, P|\!\rangle^{d}(m) + \langle\!|C_2, P|\!\rangle^{d}(m) + 1
\end{aligned}
$$

Questions?