A Simple Abstraction for Complex Concurrent Indexes

OOPSLA 2011

Imperial College London:
Pedro da Rocha Pinto, Thomas Dinsdale-Young, Philippa Gardner and Mark Wheelhouse

University of Cambridge:
Mike Dodds
Motivation

Indexes are ubiquitous in computing systems:

- Databases
- Caches
- File systems
- JavaScript Objects
- Linked Lists
- Arrays
- B-trees
- Hash Tables

And have a variety of implementations:
An index is a partial function mapping keys to values:

$$ H : Keys \rightarrow Vals $$

There are three basic operations on an index $h$:

$$ r := \text{search}(h,k) $$

$$ \text{insert}(h,k,v) $$

$$ \text{remove}(h,k) $$
This intuitive specification is not enough to reason about concurrent access to the index.

e.g

\[
\begin{align*}
    r & := \text{search}(h,k_2); \\
    \text{insert}(h,k_1,r) & \text{ || remove}(h,k_2)
\end{align*}
\]

with \( k_1 \neq k_2 \)
Concurrent Abstract Predicates:

\[ \text{in}(h, k, v) : \text{there is a mapping in the index } h \text{ from } k \text{ to } v, \text{ and only the thread holding the predicate can modify } k. \]

\[ \text{out}(h, k) : \text{there is no mapping in the index } h \text{ from } k, \text{ and only the thread holding the predicate can modify } k. \]

Axioms:

\[ \text{e.g. } \text{in}(h, k, v) \ast \text{out}(h, k) \Rightarrow \text{false} \]
Concurrent Index Specification

\[
\begin{align*}
\{ \text{in}(h,k,v) \} & \quad r := \text{search}(h,k) \quad \{ \text{in}(h,k,v) \land r = v \} \\
\{ \text{out}(h,k) \} & \quad r := \text{search}(h,k) \quad \{ \text{out}(h,k) \land r = \text{null} \} \\
\{ \text{in}(h,k,v') \} & \quad \text{insert}(h,k,v) \quad \{ \text{in}(h,k,v') \} \\
\{ \text{out}(h,k) \} & \quad \text{insert}(h,k,v) \quad \{ \text{in}(h,k,v) \} \\
\{ \text{in}(h,k,v) \} & \quad \text{remove}(h,k) \quad \{ \text{out}(h,k) \} \\
\{ \text{out}(h,k) \} & \quad \text{remove}(h,k) \quad \{ \text{out}(h,k) \} 
\end{align*}
\]
Simple Concurrent Example

\[ r \leftarrow \text{search}(h, k_2); \]
\[ \{ \text{out}(h, k_1) \ast \text{in}(h, k_2, v) \} \]

\[ \{ \text{out}(h, k_1) \land r = v \} \]
\[ \text{insert}(h, k_1, r) \]
\[ \{ \text{in}(h, k_1, v) \} \]

\[ \{ \text{in}(h, k_1, v) \} \]
\[ \text{remove}(h, k_2) \]
\[ \{ \text{out}(h, k_2) \} \]

\[ \{ \text{in}(h, k_1, v) \ast \text{out}(h, k_2) \} \]
However, we still cannot reason about the following programs:

\[
\text{remove}(h,k) \quad || \quad \text{remove}(h,k) \\
\text{insert}(h,k,v) \quad || \quad \text{remove}(h,k) \\
r := \text{search}(h,k) \quad || \quad \text{remove}(h,k)
\]

We need to account for the sharing of keys between threads.
Real-World Client Programs

Database sanitation:
remove all patients who have been cured, transferred or released

Graphics drawing:
clip all objects outside of some horizontal and vertical bounds

Garbage collection:
parallel marking in the mark/sweep algorithm

Web caching (NOSQL):
removing a picture whilst others are attaching comments to it
Extended Concurrent Abstract Predicates: \( i \in (0,1] \)

\[ \text{in}_{\text{def}} (h, k, v)_i \]

- \( \text{in}_{\text{def}} \) : the key \( k \) definitely maps to value \( v \)
- \( 0 < i \leq 1 \) : no other thread can change the value at key \( k \)
- \( i = 1 \) : this thread can change the value at key \( k \)
- \( \text{out}_{\text{def}} \) is analogous
Extended Concurrent Abstract Predicates: with $i \in (0,1]$

$$in_{rem}(h, k, v)_i$$

- $rem$ : the key $k$ might map to a value, and if it does that value is $v$

- $0 < i \leq 1$ : all threads can only remove the value at key $k$, the current thread has not done this so far

- $out_{rem}$ is analogous

- Similarly we have $out_{ins}$ and $out_{ins}$ for insert only
New specification of remove(h,k):

\[ \{ in_{def}(h, k, v)_1 \} \text{ remove}(h, k) \{ out_{def}(h, k)_1 \} \]

\[ \{ out_{def}(h, k)_i \} \text{ remove}(h, k) \{ out_{def}(h, k)_i \} \]

\[ \{ in_{rem}(h, k, v)_i \lor out_{rem}(h, k)_i \} \text{ remove}(h, k) \{ out_{rem}(h, k)_i \} \]
Concurrent remove

\[
\begin{align*}
\{ \text{in}_{\text{def}}(h, k, v)_{1} \} \\
\{ \text{in}_{\text{rem}}(h, k, v)_{1} \} \\
\{ \text{in}_{\text{rem}}(h, k, v)_{0.5} \times \text{in}_{\text{rem}}(h, k, v)_{0.5} \} \\
\{ \text{in}_{\text{rem}}(h, k, v)_{0.5} \} & \parallel \{ \text{in}_{\text{rem}}(h, k, v)_{0.5} \} \\
\text{remove}(h, k) & \parallel \text{remove}(h, k) \\
\{ \text{out}_{\text{rem}}(h, k)_{0.5} \} & \parallel \{ \text{out}_{\text{rem}}(h, k)_{0.5} \} \\
\{ \text{out}_{\text{rem}}(h, k)_{0.5} \times \text{out}_{\text{rem}}(h, k)_{0.5} \} \\
\{ \text{out}_{\text{def}}(h, k)_{1} \} 
\end{align*}
\]
Parallel Sieve of Eratosthenes
Parallel Sieve of Eratosthenes

Worker thread:

\[
\{ 2 \leq v \land \bigcirc_{2 \leq n \leq \text{max}} \text{in}_{\text{rem}}(h, n, 0)_i \} \\
\]

\[
\text{worker}(v, \text{max}, h) \\
\quad c := v + v; \\
\quad \text{while}(c \leq \text{max}) \\
\quad \quad \text{remove}(h, c); \\
\quad \quad c := c + v; \\
\]

\[
\{ \bigcirc_{2 \leq n \leq \text{max}} \overline{\text{fac}}(n, v) \Rightarrow \text{out}_{\text{rem}}(h, n)_i \land \}
\]

\[
\{ \bigcirc_{2 \leq n \leq \text{max}} \neg\text{fac}(n, v) \Rightarrow \text{in}_{\text{rem}}(h, n, 0)_i \}
\]
Combining Predicates

\[ in_{rem}(h, k, v)_i \ast in_{rem}(h, k, v)_j \Leftrightarrow in_{rem}(h, k, v)_{i+j} \quad \text{if } i + j \leq 1 \]

\[ in_{rem}(h, k, v)_i \ast out_{rem}(h, k)_j \Rightarrow out_{rem}(h, k)_{i+j} \quad \text{if } i + j \leq 1 \]
Parallel Sieve of Eratosthenes

Sieve specification:

\[
\{ \oplus_{2 \leq n \leq \text{max}} \text{in}_{\text{def}}(h, n, 0)_1 \land \text{max} > 1 \}
\]

worker(2,max,h) || worker(3,max,h) || ... || worker(m,max,h)

\[
\left\{ \begin{array}{l}
\text{isPrime}(n) \Rightarrow \text{in}_{\text{def}}(h, n, 0)_1 \land \\
\oplus_{2 \leq n \leq \text{max}} \neg\text{isPrime}(n) \Rightarrow \text{out}_{\text{def}}(h, n)_1
\end{array} \right\}
\]

where \( m = \lfloor \sqrt{\text{max}} \rfloor \)
Implementing a Concurrent Index

Our abstract concurrent index specification is sound for a number of different implementations, including:

- Linked Lists
- Arrays
- B-trees
- Hash Tables
Concurrent B-tree
B-tree remove implementation must satisfy the specification:

\[
\{ \text{in}_{def}(h, k, v)_1 \} \quad \text{remove}(h, k) \quad \{ \text{out}_{def}(h, k)_1 \}
\]

Concrete definition of \text{in}_{def}(h, k, v)_i:

\[
\text{in}_{def}(h, k, v)_i = \exists r. \quad B_{E}(h, k, v) \quad r_{I(r,h)} \quad [\text{LOCK}]_g^r \quad [\text{SWAP}]_g^r \\
\quad \ast [\text{REM}(0, k)]_{(d,i)}^r \quad \ast \quad v \in \text{vals} \quad [\text{INS}(0, k, v)]_{(d,i)}^r
\]
Concurrent B-tree

Check axioms: for example,

\[ \text{in}_{\text{def}}(h, k, v)_i \ast \text{in}_{\text{def}}(h, k, v)_j \iff \text{in}_{\text{def}}(h, k, v)_{i+j} \quad \text{if } i + j \leq 1 \]

Check stability of predicates

Check implementations satisfy abstract specifications
Concurrent B-tree

Proof of remove implementation:

```plaintext
Figure 21. Proof outline for B-tree remove (excluding
loop body).
```

```plaintext
Figure 22. Proof outline for B-tree remove (main
loop body).
```
Conclusion

Summary:

- simple abstract spec for concurrent indexes
- essence of real-world client programs
- correct implementations
  - linked lists
  - hash tables
  - concurrent B-trees
- proof structure lends itself to automation

Future work:

- Automation/Proof Assistant (Dinsdale-Young)
- java.util.concurrent (da Rocha Pinto)
- File Systems (Ntzik)