Abstract Reasoning for Concurrent Indexes

Pedro da Rocha Pinto, Thomas Dinsdale-Young, Philippa Gardner, Mark Wheelhouse
Imperial College London
{pmd09, td202, pg, mjw03}@doc.ic.ac.uk

1. Introduction

Shared-memory concurrency plays an important role in modern computer systems: for example, in classic file systems and databases, and the evolving web and multi-core operating systems. Many of these systems make fundamental use of concurrent indexes. A concurrent index can be viewed as a mapping from key-values to data (typically pointers), which can be accessed simultaneously by many threads calling read/write operations on key-values. A user typically works with this abstract view of indexes, rather than with the intricate details of complex implementations. We could verify such implementations directly [5], but we instead choose to give a formal abstract specification of concurrent indexes, and verify that well-known implementations using B^{Link} trees [6] and hash tables satisfy this specification.

Separation logic has been used to reason about similar sequential structures [7], but we use an extension called concurrent abstract predicates, recently introduced by Dinsdale-Young and Gardner with Dodds, Parkinson and Vafeiadis [3]. This work enables us to develop abstract local reasoning about concurrent modules, providing a 'fiction of separation' [4] at the module level which is not present in the underlying implementation. For example, with concurrent indexes, we can reason separately about the concurrent access of partial value-pointer pairs despite the underlying concurrent implementation involving a complex, connected data structure such as a B^{Link} tree.

This work is not a straightforward application of concurrent abstract predicates as introduced in [3]. The concurrent B^{Link} tree algorithm is maximally fine-grained, allowing programs to concurrently access the same key-value. The original concurrent abstract predicate technology is not able to specify the equivalent fine-grained behaviour at the module level. We adapt the approach, allowing fractional permissions on abstract predicates to capture the fine-grained nature of the implementation, and providing more fine-grained predicates to specify stronger properties about programs.

2. Specifications

A concurrent index h is a mapping from key-values v to data pointers p or nothing (null) which can be accessed by multiple threads at the same time:

$$h : KeyValues \to Pointers \cup \{\mathsf{null}\}$$

There are three operations which manipulate a concurrent index:

$$r := \textsc{search}(h, v) \quad r := \textsc{insert}(h, v, p) \quad r := \textsc{delete}(h, v)$$

The search operation returns the pointer mapped by v if it exists in the index h, or null if it does not. The insert operation tries to insert a pointer p at the key-value v. If it already exists it returns false and does nothing, otherwise it extends the mapping and returns true. The delete operation tries to remove the mapping for key-value v from the index. If it does not exist in the mapping then it returns false and does nothing, otherwise it removes the value-pointer pair from the index and returns true.

We want to give specifications for these operations so that we may reason about concurrent indexes. For example, given program,

$$r_1 := \textsc{search}(h, v) \quad \| \quad r_2 := \textsc{insert}(h, v, q)$$

we should be able to prove that if the key-value v is unassigned at the start of the program, then it will be assigned the pointer q at the end of the program, but we will not know the return value of the search operation. This is because we do not know if the search will occur before or after the insert operation.

Predicate	Permission	Rely	Guarantee
$\mathsf{in}_{\mathsf{def}}$ / $\mathsf{out}_{\mathsf{def}}$	1	No actions	All actions
$\mathsf{in}_{\mathsf{def}}$ / $\mathsf{out}_{\mathsf{def}}$	i	No actions	No actions
$\mathsf{in}_{\mathsf{ins}}$ / $\mathsf{out}_{\mathsf{ins}}$	i	Only inserts	Only inserts
$\mathsf{in}_{\mathsf{del}}$ / $\mathsf{out}_{\mathsf{del}}$	i	Only deletes	Only deletes
unk	i	All actions	All actions
read		All actions	No actions

Table 1. Predicates and their interference

We define a set of concurrent abstract predicates which express if there is a mapping to a key-value or not and also describe the interference that is allowed on this key-value in the shared state. We provide the following set of abstract predicates parametric on key-value v and index h:

$\mathsf{in}_{\mathsf{def}}(h, v, p, i)$	the key-value v is definitely assigned.
$\mathsf{out}_{\mathsf{def}}(h, v, i)$	the key-value v is definitely not assigned.
$\mathsf{in}_{\mathsf{ins}}(h, v, p, i)$	the key-value v is assigned and there are no deletions in the interference environment.
$\mathsf{out}_{\mathsf{ins}}(h, v, i)$	the key-value v might not be assigned and there are no deletions in the interference environment.
$\mathsf{in}_{\mathsf{del}}(h, v, p, i)$	the key-value v might be assigned and there are no insertions in the interference environment.
$\mathsf{out}_{\mathsf{del}}(h, v, i)$	the key-value v is not assigned and there are no insertions in the interference environment.
$\mathsf{unk}(h, v, i)$	nothing is known about the key-value v.
$\mathsf{read}(h, v)$	nothing is known about the key-value v and the thread can only search.

The concurrent abstract predicates are extended with permission fractions i and j, with $0 < i, j \leq 1$, as presented by Boyland [1]. These permissions are used to record the splitting of the predicates. We use def, ins and del to denote the allowed interference on the shared state. The def predicates specify that there can only be either sequential writes or concurrent reads on the shared state. The ins (or del) predicates specify that there can only be insert (or delete) and search operations on the shared state. Finally, the unk and read predicates allow for any kind of operation on the shared state. The difference is that the read predicate only allows the current thread to perform searches, whilst the unk predicate allows the current thread to carry out any operation.

The choice of predicates may, at first, seem somewhat arbitrary, but in fact these predicates give a complete coverage of all possible thread and environment interactions. Table 1 shows how our choice of predicates corresponds to a rely-guarantee way of thinking about the environment and also how they cover the space of thread interactions. An action is some program effect that modifies the shared state. A search operation does not modify the shared state and so has no corresponding actions. Inserts add pointers to the index and deletes remove pointers from the index. The Rely corresponds to the actions that can be performed by the program environment (the other threads running concurrently), whereas the Guarantee corresponds to the actions that can be performed by the current thread.

We provide a system of axioms which allow us to change between interference environments and split predicate permissions. A

subset of these axioms, sufficient for our examples, is given below:

$$\begin{array}{rcl}
\mathsf{in}_{\mathsf{def}}(h,v,p,1) & \iff & \mathsf{in}_{\mathsf{del}}(h,v,p,1) \\
\mathsf{out}_{\mathsf{def}}(h,v,1) & \iff & \mathsf{out}_{\mathsf{del}}(h,v,1) \\
\mathsf{in}_{\mathsf{del}}(h,v,p,i) * \mathsf{in}_{\mathsf{del}}(h,v,p,j) & \iff & \mathsf{in}_{\mathsf{del}}(h,v,p,i+j) \\
\mathsf{out}_{\mathsf{del}}(h,v,i) * \mathsf{out}_{\mathsf{del}}(h,v,j) & \iff & \mathsf{out}_{\mathsf{del}}(h,v,i+j) \\
\mathsf{in}_{\mathsf{def}}(h,v,p,i) * \mathsf{read}(h,v) & \iff & \mathsf{in}_{\mathsf{def}}(h,v,p,i) \\
\mathsf{out}_{\mathsf{def}}(h,v,i) * \mathsf{read}(h,v) & \iff & \mathsf{out}_{\mathsf{def}}(h,v,i) \\
\mathsf{in}_{\mathsf{def}}(h,v,p,1) & \implies & \mathsf{unk}(h,v,1)
\end{array}$$

The first and second axioms show how, when we have full permission ($i = 1$) on a key-value, we can change how that key-value is treated. For example here we move from an environment where all inserts and deletes must occur sequentially, to an environment where many deletions may occur in parallel. The third and fourth axioms show how permissions may be split and combined so long as $i + j$ does not exceed full permission 1. The fifth and sixth axioms show how a read permission can be split off of any write permission, in this case in an environment where inserts and deletes are forced to occur sequentially. A read permission does not retain any information about whether the key-value is assigned in the index or not. Searches carried out with just a read permission will return an unknown result, however, we still know that they will not fault. The final axiom shows how any predicate, with full permission, can transition to the unknown predicate. This predicate loses all information about the contents of the index, but still allows for us to prove safety of a program. Note that once we have moved to the unknown state, we cannot get back to any other state.

With these predicates, we can now give the specifications of our index-manipulating operations. We will not give the full specifications here, but enough to provide an example of our reasoning.

$$\begin{array}{rl}
\{\mathsf{out}_{\mathsf{def}}(h,v,1)\}\ r := \textsc{insert}(h,v,p) & \left\{\begin{array}{l}\mathsf{in}_{\mathsf{def}}(h,v,p,1) \\ \wedge\ r = \mathsf{true}\end{array}\right\} \\
\{\mathsf{in}_{\mathsf{del}}(h,v,p,i)\}\ \ r := \textsc{delete}(h,v) & \left\{\begin{array}{l}\mathsf{out}_{\mathsf{del}}(h,v,i) \wedge \\ (r = \mathsf{true} \vee r = \mathsf{false})\end{array}\right\} \\
\{\mathsf{read}(h,v)\}\ \ r := \textsc{search}(h,v) & \left\{\begin{array}{l}\mathsf{read}(h,v) \wedge \\ (r = _ \vee r = \mathsf{null})\end{array}\right\}
\end{array}$$

Using these specifications, and common local Hoare reasoning rules for sequential and parallel programs, we can now present a proof of the program we gave at the beginning of this section.

$$\begin{array}{c}
\{\mathsf{out}_{\mathsf{def}}(h,v,1)\} \\
\{\mathsf{read}(h,v) * \mathsf{out}_{\mathsf{def}}(h,v,1)\} \\
\begin{array}{cc}
\{\mathsf{read}(h,v)\} & \{\mathsf{out}_{\mathsf{def}}(h,v,1)\} \\
r_1 := \textsc{search}(h,v) & r_2 := \textsc{insert}(h,v,q) \\
\left\{\begin{array}{l}(\mathsf{read}(h,v) \wedge r_1 = _) \vee \\ (\mathsf{read}(h,v) \wedge r_1 = \mathsf{null})\end{array}\right\} & \left\{\begin{array}{l}\mathsf{in}_{\mathsf{def}}(h,v,q,1) \\ \wedge\ r_2 = \mathsf{true}\end{array}\right\}
\end{array} \\
\left\{\begin{array}{l}(\mathsf{read}(h,v) \wedge r_1 = _) \vee (\mathsf{read}(h,v) \wedge r_1 = \mathsf{null}) \\ * \mathsf{in}_{\mathsf{def}}(h,v,q,1) \wedge r_2 = \mathsf{true}\end{array}\right\} \\
\{\mathsf{read}(h,v) * \mathsf{in}_{\mathsf{def}}(h,v,q,1) \wedge (r_1 = _ \vee r_1 = \mathsf{null}) \wedge r_2 = \mathsf{true}\} \\
\{\mathsf{in}_{\mathsf{def}}(h,v,q,1) \wedge (r_1 = _ \vee r_1 = \mathsf{null}) \wedge r_2 = \mathsf{true}\} \\
\{\mathsf{in}_{\mathsf{def}}(h,v,q,1)\}
\end{array}$$

We use $r = \_$ as short hand notation for $\exists p.(r = p)$. The proof starts with the predicate $\mathsf{out}_{\mathsf{def}}(h,v,1)$ which specifies that in the index h there is no pointer mapped to by the key-value v. The def interference environment specifies that that no other thread is modifying this key-value since the current thread holds full permission on this key. We can use one of our axioms to turn this predicate into $\mathsf{read}(h,v) * \mathsf{out}_{\mathsf{def}}(h,v,1)$. This allow us to use the $\mathsf{read}(h,v)$ predicate in the left hand thread, which performs a simple search operation and does not modify the shared state. This read predicate captures the fact that we do not know the return value of the search operation as we do not know in which order the search and insert will execute. With the $\mathsf{out}_{\mathsf{def}}(h,v,1)$ predicate we can prove the right hand thread in a deterministic way, as we know that it is the only thread changing the shared state for the key-value v. Finally, when both threads finish their execution, we can merge the $\mathsf{read}(h,v)$ predicate back into the $\mathsf{in}_{\mathsf{def}}(h,v,q)$ predicate with another axiom. We could provide a similar proof for the case where the key-value v is already assigned before the program begins. In this case we know the insert would do nothing and the search would return the original pointer stored in the index.

In a similar way we can provide specifications for other interactions on a concurrent index. For example, if we know that a certain key-value is assigned in the index and we run two concurrent deletes on that key-value, whilst we do not know which deletion will succeed and which will fail, we do know that the key-value will definitely no longer be assigned after. Permission splitting allows us to pass this knowledge to both threads.

$$\begin{array}{c}
\{\mathsf{in}_{\mathsf{def}}(h,v,p,1)\} \\
\{\mathsf{in}_{\mathsf{del}}(h,v,p,1)\} \\
\{\mathsf{in}_{\mathsf{del}}(h,v,p,1/2) * \mathsf{in}_{\mathsf{del}}(h,v,p,1/2)\} \\
\begin{array}{cc}
\{\mathsf{in}_{\mathsf{del}}(h,v,p,1/2)\} & \{\mathsf{in}_{\mathsf{del}}(h,v,p,1/2)\} \\
r_1 := \textsc{delete}(h,v) & r_2 := \textsc{delete}(h,v) \\
\left\{\begin{array}{l}\mathsf{out}_{\mathsf{del}}(h,v,1/2) \wedge \\ (r_1 = \mathsf{true} \vee r_1 = \mathsf{false})\end{array}\right\} & \left\{\begin{array}{l}\mathsf{out}_{\mathsf{del}}(h,v,1/2) \wedge \\ (r_2 = \mathsf{true} \vee r_2 = \mathsf{false})\end{array}\right\}
\end{array} \\
\left\{\begin{array}{l}\mathsf{out}_{\mathsf{del}}(h,v,1/2) \wedge (r_1 = \mathsf{true} \vee r_1 = \mathsf{false}) \\ * \mathsf{out}_{\mathsf{del}}(h,v,p,1/2) \wedge (r_2 = \mathsf{true} \vee r_2 = \mathsf{false})\end{array}\right\} \\
\left\{\begin{array}{l}\mathsf{out}_{\mathsf{del}}(h,v,1) \wedge (r_1 = \mathsf{true} \vee r_1 = \mathsf{false}) \\ \wedge\ (r_2 = \mathsf{true} \vee r_2 = \mathsf{false})\end{array}\right\} \\
\{\mathsf{out}_{\mathsf{del}}(h,v,1)\} \\
\{\mathsf{out}_{\mathsf{def}}(h,v,1)\}
\end{array}$$

We cannot always know the exact state of an index after some program has run, but we can show that the program is safe. For example, if we run an insert and delete command on the same key-value in parallel, we will not know if that key-value is assigned after, but will know that the program did not fault.

$$\begin{array}{c}
\{\mathsf{in}_{\mathsf{def}}(h,v,p,1)\} \\
r_1 := \textsc{delete}(h,v) \quad \| \quad r_2 := \textsc{insert}(h,v,q) \\
\{\mathsf{unk}(h,v,1)\}
\end{array}$$

Our reasoning also scales to more complex programs.

$$\begin{array}{l}
\textsc{search}(h,v,r) \\
\quad \| \\
\quad \| \\
\quad \| \\
\quad \|
\end{array}
\begin{array}{c}
\{\mathsf{out}_{\mathsf{def}}(h,v,1)\} \\
\textsc{insert}(h,v,p) \\
\textsc{search}(h,v) \quad \| \quad \textsc{search}(h,v) \\
\textsc{delete}(h,v) \\
\{\mathsf{out}_{\mathsf{def}}(h,v,1)\}
\end{array}$$

In this example we know that they key-value v is definitely unassigned at the end of the program. We could also show that whist the leftmost search has an unknown return value, both of the other two searches would return the same value p.

Our predicates address all the possible interference which can be induced on the shared state by the current thread and the environment. However, with the specifications presented, once we get to an unknown state, as in our third example, we are stuck there. This is a very strong assumption. It says that if there is a race condition where we cannot determine the contents of the index for a key-value, we can only prove safety for the program from this point onward, and we cannot prove what the index contents at that key might be. However, we should sometimes be able to recover from an unknown state because we can work out what the contents will be. For example, if all threads end with a deletion of a certain key-value, then we can be sure that this key-value will not be assigned in the index at the end of the program.

We can extend the unknown predicate from our system with tags such as $\mathsf{unk}_{\mathsf{ins}}(h,v,i)$ and $\mathsf{unk}_{\mathsf{del}}(h,v,i)$ which record the last write operation carried out by the thread. We use this knowledge to recover from an unknown state to a known state once we have full permission on that key-value again. In particular we add axioms to our reasoning system which capture this behaviour, such as:

$$\mathsf{unk}_{\mathsf{del}}(h,v,1) \implies \mathsf{out}_{\mathsf{def}}(h,v,1)$$

The aim of our specifications is that if you can intuitively deduce a property about a program then you should be able to formally prove that property with our system of predicates.

3. Implementations

Concurrent indexes are widely used in databases and filesystems. As such, there are many different index implementations, such as B^{Link} trees and hash tables. One of the biggest advantages of using the concurrent abstract predicate approach is that it is possible to verify a concrete implementation against our abstract specification. We have given a node-list implementation (a cut down version of a B^{Link} tree), and shown that it satisfies our abstract specification. We then use it to show that Sagiv's B^{Link} tree [6] and a hash table implementation also satisfy the abstract specification. To prove that an implementation is correct, we need to provide concrete interpretations of each of our abstract predicates for that implementation. Taking the node-list implementation as an example we shall briefly outline this process. (Full details can be found in [2].)

Assume we are given a predicate $\mathsf{nodeList}(h, S)$ which describes a list of nodes, each with a maximum capacity of $2k$ value-pointer pairs, stored at h containing a set of value-pointer pairs S. We give the concrete interpretation of the $\mathsf{in}_{\mathsf{def}}$ predicate as follows:

$$\begin{array}{rcl} \mathsf{in}_{\mathsf{def}}(h, v, p, i) & \equiv & \exists S. \boxed{\mathsf{nodeList}(h, S) \wedge (v, p) \in S}^{n}_{\mathcal{A}} \\ & & * \, [\textsc{Change}(v)]^{n}_{i} \end{array}$$

This describes two parts of memory. The boxed assertion talks about the shared memory in a region labelled n. The assertion says that the region n contains a node-list at h and that the pair (v, p) is contained somewhere in this list. Boxed assertions on the same region behave additively under $*$, that is $\boxed{P}^{n} * \boxed{Q}^{n} \equiv \boxed{P \wedge Q}^{n}$. The boxed assertion is parameterised by an interference environment $\mathcal{A}$ which describes the possible effects of the environment on the shared state. The second part of the concrete interpretation talks about the local (or private) memory of the current thread. In this case the local part of memory contains a token $[\textsc{Change}(v)]$ which is associated with shared region n and has permission i. This token allows the current thread to perform certain actions on the shared state in region n. In the case of this $[\textsc{Change}(v)]$ token, the thread may read the pointer at key-value v if it has permission $i > 0$. If the thread has full permission on the $[\textsc{Change}(v)]$ token, i.e. $i = 1$, then it is also allowed to modify (add or delete) the pointer at key-value v and it knows that no other thread is accessing that key-value. The other predicates have similar concrete interpretations.

The interference environment $\mathcal{A}$ is defined in terms of a set of actions $(P \rightsquigarrow Q)$ which describe how the shared state may be modified by the environment. For example, in the node-list implementation a thread may insert a new value-pointer pair into the index using the following action:

$$\begin{array}{l} x \mapsto \mathsf{node}(t_{id}, v_0, s, v_{i+1}, y) \\ \quad * \, [\textsc{Change}(v)]^{n}_{1} \wedge |s| < 2k \\ \quad * \, [\textsc{Unlock}(x)]^{l}_{1} \wedge (v, q) \notin s \end{array} \quad \rightsquigarrow \quad \begin{array}{l} x \mapsto \mathsf{node}(t_{id}, v_0, s', v_{i+1}, y) \\ \quad * \, [\textsc{Mod}(x, v)]^{n}_{1} \\ \quad \wedge \, s' = s \uplus (v, p) \end{array}$$

The left-hand side of the action describes part of the shared state before the action and the right-hand side describes how this part of the shared state n is modified by the action. This action requires that there is a node x in the shared state that has been locked by the current thread (with thread identifier t_{id}). This node contains a set of value-pointer pairs s whose key-values are greater than the node's minimum value v_0 and less than or equal to the node's maximum value v_{i+1}. The set of value-pointer pairs has less than $2k$ elements, which means there is space in this node to add another value-pointer pair. The node also contains a pointer y to the next node in the list. The action then adds the value-pointer pair (v, p) to the set s. Actions only describe how the shared state is modified, so there is some token transfer taking place here as well. In particular the $[\textsc{Mod}(x, v)]$ token is moved from the thread's local state to the shared state and the $[\textsc{Change}(v)]$ and $[\textsc{Unlock}(x)]$ tokens are moved from the shared state to the thread's local state. This token transfer corresponds to the abstract level permission and predicate system. In this example the current thread is giving up the right to modify the key-value v in the node x and it is gaining the right to change the key-value v in the set as well as the right to unlock the node x. In this implementation a thread can only obtain the the right to modify a node (i.e. gain the $[\textsc{Mod}(x, v)]$ token) if it has the cell x locked and is allowed to change if v has a mapping in the index (i.e. owns the $[\textsc{Change}(v)]$ and $[\textsc{Unlock}(x)]$ tokens). For a full description of the interference environment $\mathcal{A}$ and the actions contained within it see [2].

We can now prove that the implementations of the index operations satisfy the concrete operation specifications. So long as each of the concrete interpretations of the abstract predicates is stable, that is invariant under the actions in the interference environment $\mathcal{A}$, the abstraction of the implementation is sound. This means the implementation satisfies the abstract specification.

4. Conclusions and further work

We have given an abstract specification of concurrent indexes, and have shown that implementations using B^{Link} trees and hash tables are correct. This involved extending recent work on concurrent abstract predicates with permissions and interference information: the permissions allow us to prove safety for any possible interleaving of index operations; the interference information enables us to prove stronger properties about programs. Our reasoning requires that the abstract operations for manipulating the syntax are local, only requiring knowledge about the key-value which the operation manipulates. Notice that it does not require conditions on the implementation, such that the low-level operations are linearisable.

This work provides a first step in a larger project on verifying concurrent indexes. For example, we will reason about concurrent B^{Link} tree implementations used in databases and filesystems in order to get a better understanding of which properties of concurrent indexes are useful. We can already think of several natural extensions to the reasoning presented here. For example, currently when multiple inserts occur for the same key-value, our reasoning states that the contents of the pointer are unknown. However, we do intuitively know the set of pointers, and we could extend our specifications to include this information if it were important. Similarly, we could track the set of possible return values rather than lose this information. By focussing our attention on implementations used in practice, we will obtain an understanding of what kinds of properties are important for client programs using concurrent indexes.

Acknowledgements: We thank Mike Dodds for many interesting discussions about this work. In particular, he showed us his unpublished work on using concurrent abstract predicates to reason about the sieve of Eratosthenes, which contains predicates annotated with shared state interference.

References

[1] J. Boyland. Checking interference with fractional permissions. *Static Analysis*, 2003.

[2] P. da Rocha Pinto. Reasoning about Concurrent Indexes. Master's thesis, Imperial College London, 2010. Available at `http://www.doc.ic.ac.uk/~pmd09/publications/msc-thesis.pdf`.

[3] T. Dinsdale-Young, M. Dodds, P. Gardner, M. Parkinson, and V. Vafeiadis. Concurrent Abstract Predicates. *ECOOP*, 2010.

[4] T. Dinsdale-Young, P. Gardner, and M. Wheelhouse. Abstraction and Refinement for Local Reasoning. *VSTTE*, 2010.

[5] G. Malecha, G. Morrisett, A. Shinnar, and R. Wisnesky. Toward a verified relational database management system. *POPL*, 2010.

[6] Y. Sagiv. Concurrent operations on B*-trees with overtaking. *Journal of Computer and System Sciences*, 1986.

[7] A. Sexton and H. Thielecke. Reasoning about B+ Trees with Operational Semantics and Separation Logic. *ENTCS*, 2008.