A Simple Abstraction for Complex Concurrent Indexes

Pedro da Rocha Pinto
Imperial College London
pmd09@doc.ic.ac.uk

Thomas Dinsdale-Young
Imperial College London
td202@doc.ic.ac.uk

Mike Dodds
University of Cambridge
mike.dodds@cl.cam.ac.uk

Philippa Gardner
Imperial College London
pg@doc.ic.ac.uk

Mark Wheelhouse
Imperial College London
mjw03@doc.ic.ac.uk

Abstract

Indexes are ubiquitous. Examples include associative arrays, dictionaries, maps and hashes used in applications such as databases, file systems and dynamic languages. Abstractly, a sequential index can be viewed as a partial function from keys to values. Values can be queried by their keys, and the index can be mutated by adding or removing mappings. Whilst appealingly simple, this abstract specification is insufficient for reasoning about indexes that are accessed concurrently.

We present an abstract specification for concurrent indexes. We verify several representative concurrent client applications using our specification, demonstrating that clients can reason abstractly without having to consider specific underlying implementations. Our specification would, however, mean nothing if it were not satisfied by standard implementations of concurrent indexes. We verify that our specification is satisfied by algorithms based on linked lists, hash tables and B^{Link} trees. The complexity of these algorithms, in particular the B^{Link} tree algorithm, can be completely hidden from the client's view by our abstract specification.

General Terms Algorithms, Concurrency, Theory, Verification.

Keywords B-Trees, Concurrent Abstract Predicates, Separation Logic.

1. Introduction

An *index* is a data structure where data is associated with identifying *keys*, through which the data can be efficiently retrieved. Indexes are ubiquitous in computer systems: they are integral to databases, caches, file systems, and even the objects of dynamic languages such as JavaScript. Concurrent systems use indexes for: *database sanitation* – to concurrently remove patients who have been cured or transferred; *graphics rendering* – to clip all objects outside horizontal or vertical bounds; *garbage collection* – to concurrently mark reachable objects; and *web applications* – to allow multiple clients to add and remove pictures and comments, for instance. A variety of implementations of indexes exist, such as skip lists, hash tables and B-trees. Different implementations offer different performance characteristics, but all exhibit the same abstract behaviour.

To a sequential client, an index can be viewed abstractly as a partial function from keys to values. A client can query or mutate the index without having to take into account the complexities of its underlying implementation. This simple, yet powerful, abstract specification largely accounts for the popularity of indexes. However, this abstraction breaks down if an index is accessed concurrently. When several threads insert, remove and query keys, clients can no longer model the whole index by a single partial function. Each client must take account of potential interference from other threads.

In this paper, we present a novel abstract specification for *concurrent indexes*, and use it to verify a number of client programs. Crucially, clients can reason abstractly using our specification without having to consider specific underlying implementations. However, we can also verify our specification against complex concurrent index implementations.

Our approach is based on concurrent abstract predicates [7], recently introduced to reason about concurrent modules. With this technology, we can view the index as *divisible*: keys are a resource which can be divided between the threads. When threads operate on disjoint keys, they can do so independently of each other. When threads operate on shared keys, concurrent abstract predicates can account for the interference caused by other threads.

OOPSLA'11, October 22–27, 2011, Portland, Oregon, USA.

Intuitive description of the approach. First, consider the *disjoint* case, where each key is manipulated by a single thread. In this case, we can verify each thread in terms of the keys it uses, and combine the results to understand the composed system. In our specification, we have the predicates $\mathsf{in}(h,k,v)$ and $\mathsf{out}(h,k)$: $\mathsf{in}(h,k,v)$ declares that the key k is mapped to value v in index h; $\mathsf{out}(h,k)$ declares that there is no mapping of k. A thread must hold one of these predicates in order to modify k. A disjointness axiom enforces that only one thread can hold such a predicate on k at any one time. We describe these predicates as *abstract*, because they do not reveal how they are implemented.

Given these predicates, we can give the following specification to `remove`:

$$\{\mathsf{in}(\mathtt{h},\mathtt{k},v)\} \quad \mathtt{remove}(\mathtt{h},\mathtt{k}) \quad \{\mathsf{out}(\mathtt{h},\mathtt{k})\}$$

With this specification, we can prove the following property of a simple client program performing parallel removes (our proof assumes that $\mathtt{k}_1 \neq \mathtt{k}_2$):

$$\begin{array}{c}
\{\mathsf{in}(\mathtt{h},\mathtt{k}_1,v_1) * \mathsf{in}(\mathtt{h},\mathtt{k}_2,v_2)\} \\
\begin{array}{c||c}
\{\mathsf{in}(\mathtt{h},\mathtt{k}_1,v_1)\} & \{\mathsf{in}(\mathtt{h},\mathtt{k}_2,v_2)\} \\
\mathtt{remove}(\mathtt{h},\mathtt{k}_1) & \mathtt{remove}(\mathtt{h},\mathtt{k}_2) \\
\{\mathsf{out}(\mathtt{h},\mathtt{k}_1)\} & \{\mathsf{out}(\mathtt{h},\mathtt{k}_2)\}
\end{array} \\
\{\mathsf{out}(\mathtt{h},\mathtt{k}_1) * \mathsf{out}(\mathtt{h},\mathtt{k}_2)\}
\end{array}$$

In this proof, we reason about the parallel threads individually. We then join the disjoint pre- and postconditions to form the overall proof. Disjointness is expressed by the separating conjunction, $*$, of concurrent separation logic [16]. The disjointness axiom requires that $\mathtt{k}_1 \neq \mathtt{k}_2$.

Now consider the shared case, where threads can interfere with each other: for example, when $\mathtt{k}_1 = \mathtt{k}_2$ in the parallel removes. We introduce the more refined predicates $\mathsf{in}_{\mathsf{def}}(h,k,v)_i$, $\mathsf{out}_{\mathsf{def}}(h,k)_i$, $\mathsf{in}_{\mathsf{rem}}(h,k,v)_i$ and $\mathsf{out}_{\mathsf{rem}}(h,k)_i$. These predicates are extended in two ways:

1. def and rem are *restrictions* on the type of interference that is allowed on the key: def prohibits any interference, while rem only permits removal of the key. All threads must agree on the type of interference for a given key.
2. The interference *permissions* $i \in (0,1]$ determine whether a thread has shared ($0 < i < 1$) or exclusive ($i = 1$) access to a key. If a thread holds shared permission, it can only perform operations that respect the interference restrictions.

Using the rem predicates, we can give the following specification for `remove`:

$$\{\mathsf{in}_{\mathsf{rem}}(\mathtt{h},\mathtt{k},v)_i\} \quad \mathtt{remove}(\mathtt{h},\mathtt{k}) \quad \{\mathsf{out}_{\mathsf{rem}}(\mathtt{h},\mathtt{k})_i\}$$

Predicates can be split and joined by permission, so for example we have the axiom:

$$\mathsf{in}_{\mathsf{rem}}(h,k,v)_{i+j} \iff \mathsf{in}_{\mathsf{rem}}(h,k,v)_i * \mathsf{in}_{\mathsf{rem}}(h,k,v)_j,$$

where the sum of permissions held by all threads cannot exceed 1. In addition, if the current thread holds exclusive permission, we have axioms to change the type of the interference restriction without violating the expectations of other threads, such as:

$$\mathsf{in}_{\mathsf{def}}(\mathtt{h},\mathtt{k},v)_1 \Leftrightarrow \mathsf{in}_{\mathsf{rem}}(\mathtt{h},\mathtt{k},v)_1.$$

Using our specification and these axioms, we can prove a natural specification for parallel remove on a shared key:

$$\begin{array}{c}
\{\mathsf{in}_{\mathsf{def}}(\mathtt{h},\mathtt{k},v)_1\} \\
\{\mathsf{in}_{\mathsf{rem}}(\mathtt{h},\mathtt{k},v)_1\} \\
\{\mathsf{in}_{\mathsf{rem}}(\mathtt{h},\mathtt{k},v)_{\frac{1}{2}} * \mathsf{in}_{\mathsf{rem}}(\mathtt{h},\mathtt{k},v)_{\frac{1}{2}}\} \\
\begin{array}{c||c}
\{\mathsf{in}_{\mathsf{rem}}(\mathtt{h},\mathtt{k},v)_{\frac{1}{2}}\} & \{\mathsf{in}_{\mathsf{rem}}(\mathtt{h},\mathtt{k},v)_{\frac{1}{2}}\} \\
\mathtt{remove}(\mathtt{h},\mathtt{k}) & \mathtt{remove}(\mathtt{h},\mathtt{k}) \\
\{\mathsf{out}_{\mathsf{rem}}(\mathtt{h},\mathtt{k})_{\frac{1}{2}}\} & \{\mathsf{out}_{\mathsf{rem}}(\mathtt{h},\mathtt{k})_{\frac{1}{2}}\}
\end{array} \\
\{\mathsf{out}_{\mathsf{rem}}(\mathtt{h},\mathtt{k})_{\frac{1}{2}} * \mathsf{out}_{\mathsf{rem}}(\mathtt{h},\mathtt{k})_{\frac{1}{2}}\} \\
\{\mathsf{out}_{\mathsf{def}}(\mathtt{h},\mathtt{k})_1\}
\end{array}$$

This specifies the strong property that, if we definitely know that key `k` has a value then, after the parallel remove, we definitely know that the value has been removed. We do not know *which* thread has performed the remove, but this fact is irrelevant to correctness.

Verifying clients and index implementations. Our concurrent index specification allows us to present a single abstract interface to clients, irrespective of the choice of underlying implementation. We demonstrate that our specification is useful by verifying several representative client programs such as function memoization, a prime number sieve and a mapping of a function onto an index.

We also verify that several concurrent index algorithms satisfy our specification: in particular, a naïve linked list algorithm with coarse-grained locking for expository purposes; a simple algorithm using a hash table linked to a set of (abstract) secondary indexes, to demonstrate the verification of a more complex implementation; and Sagiv's substantial B$^{\mathit{Link}}$ tree algorithm [20] to demonstrate the scalability of our techniques to a real-world algorithm. During verification, we found a subtle bug in the B$^{\mathit{Link}}$ tree algorithm.

We use the *concurrent abstract predicate* methodology [7] to hide low-level sharing in the implementations from clients. In particular, the underlying sharing mechanism used by the B$^{\mathit{Link}}$ tree algorithm to permit non-blocking reads is exceedingly complex. This complexity is completely hidden from the client's view by our abstract specification.

Related work. We build directly on concurrent abstract predicates (CAP) [7], which provides a logic for verifying concurrent modules based on separation logic. CAP developed from three lines of work: racy concurrent variants of separation logic such as RGSep [9, 10, 23]; sequential modular reasoning based on abstract predicates [17]; and fine-grained modular reasoning based on context logic [3, 8]. We originally used RGSep [23] to verify concurrent B-trees [4]. However, RGSep and similar approaches depend on global conditions; consequently, they cannot verify abstract specifications such as our index specification. This observation formed part of our original motivation for CAP.

Our concurrent index specification descends from the set specification verified in [7]. In that paper, we focussed on building a sound logic, and verified only simple, disjoint specifications against small implementations. As far as we are aware, our specification is the first in separation logic to allow thread-local reasoning combined with races over elements of a shared structure. We have verified our index specification against Sagiv's real-world concurrent B^{Link} tree algorithm [20][1], a substantial jump in the complexity of the verification compared with [7]. Our work is beginning to develop the idioms necessary to scale to large examples.

Others have worked on reasoning abstractly about index-like data structures for sequential clients. For example, Dillig *et al.* propose a static analysis for C-like programs which represents the abstract content of containers [6]. Kuncak *et al.* propose an analysis that represents various kinds of data by abstract sets, while proving these abstractions [14].

One of the most challenging parts of our work was verifying that the concurrent B^{Link} tree implementation satisfies our specification. Some prior work exists on verifying *sequential* B-trees. In [21], B-tree search and insert operations are verified as fault-free in a simplified sequential setting. In [15], a sequential B-tree implementation is verified in Coq as part of a relational database management system. The authors comment that the proof was difficult and in need of abstraction. They go on to state that '*verifying the correctness of high-performance, concurrent B+ trees will be a particularly challenging problem*'.

The only prior verification of a *concurrent* B-tree we are aware of is a highly-abstracted version of the algorithm modelled in process algebra [18]. It verifies a global specification, rather than allowing elements to be divided between threads. We believe that our work provides the first direct, formal verification of Sagiv's widely-used algorithm [20].

Paper structure. §2 gives technical background. §3 give the disjoint index specification, and §4 extends it to sharing. §5 discusses iteration over indexes. §6 describes verifying our specification against index implementations. §3-5 can be understood from the simple summary in §2. A complete understanding of the technicalities in §6 requires knowledge of the original CAP paper [7].

2. Separation Logic & Abstraction

This paper is based on separation logic [19], a Hoare-style program logic for reasoning *locally* about programs that manipulate resource: for example, C programs that manipulate the heap. Local reasoning focusses on the specific part of the resource that is relevant at each point in the program. This supports scalable and compositional reasoning, since disjoint resource neither impinges upon nor is affected by the behaviour of the program at that point.

Separation logic specifications have a fault-avoiding partial-correctness interpretation. Consider the following specification for a command $\mathbb{C}$ (here P, Q are assertions):

$$\{P\}\ \mathbb{C}\ \{Q\}$$

The interpretation of this specification is that (1) executing $\mathbb{C}$ in a state satisfying assertion P will result in a state satisfying assertion Q, if the command terminates; and (2) the resources represented by P are the only resources needed for $\mathbb{C}$ to execute successfully.

Other resources can be conjoined with such a specification without affecting its validity. This is expressed by the following proof rule:

$$\textsc{Frame}\quad \frac{\{P\}\ \mathbb{C}\ \{Q\}}{\{P * F\}\ \mathbb{C}\ \{Q * F\}}\quad \langle \textit{side-condition} \rangle$$

This rule allows us to extend a specification on a small resource with an unmodified *frame assertion* F, giving a larger resource. Here, '$*$' is the so-called *separating conjunction*. Combining two assertions P and F into a separating conjunction $P * F$ asserts that both resources are independent of each other. The side-condition simply states that no variable occurring free in F is modified by the program $\mathbb{C}$.

Separation logic provides straightforward reasoning about sequential programs. It also handles concurrency [16], using the following rule:

$$\textsc{Par}\quad \frac{\{P_1\}\ \mathbb{C}_1\ \{Q_1\} \qquad \{P_2\}\ \mathbb{C}_2\ \{Q_2\}}{\{P_1 * P_2\}\ \mathbb{C}_1 \parallel \mathbb{C}_2\ \{Q_1 * Q_2\}}$$

In a concurrent setting, the precondition and postcondition are interpreted as resources owned exclusively by the thread. Reasoning using Par is *thread-local*. We reason about each thread purely using the resources that are mentioned in its precondition, without requiring global reasoning about interleaving. As with sequential reasoning, locality is the key to compositional reasoning about threads.

Abstraction. Abstract specifications are a mechanism for specifying the external behaviour of a module's functions, while hiding their implementation details from clients. Resources are represented by *abstract predicates* [17]. Clients do not need to know the concrete definitions of these predicates; they can reason purely in terms of the module's operations. For example, `insert` in a set module might be specified as:

$$\{\mathsf{set}(\mathtt{x}, S)\}\quad \mathtt{insert}(\mathtt{x}, \mathtt{v})\quad \{\mathsf{set}(\mathtt{x}, S \cup \{\mathtt{v}\})\}$$

`insert` updates the abstract contents of the set at address `x` from S to $S \cup \{\mathtt{v}\}$. A client can reason about the high-level

[1] Without compression, which is beyond the scope of this paper.

behaviour of `insert` without knowing about the concrete definition of the set predicate.

Abstract predicates, however, can only represent the set as a single entity, because implementation details disrupt finer-grained abstractions. *Concurrent abstract predicates* [7], on the other hand, can achieve finer abstractions. We can break the set down into predicates representing individual elements: $\mathsf{in}(x, v)$ if v belongs to the set x; $\mathsf{out}(x, v)$ if it does not. Different threads can hold access to different set elements. When element `v` is not in the set, the command `insert` can be specified by:

$$\{\mathsf{out}(\mathtt{x}, \mathtt{v})\} \quad \mathtt{insert}(\mathtt{x}, \mathtt{v}) \quad \{\mathsf{in}(\mathtt{x}, \mathtt{v})\}$$

Concurrent abstract predicates provide a finer granularity of local reasoning, whilst still hiding implementation details from clients. We follow the concurrent abstract predicate approach in our reasoning about concurrent indexes.

3. Index Specification: Disjointness

We start by giving a simple specification which divides an index up into its constituent keys. Our specification ensures that each key is accessed by at most one thread (in §4 we discuss a refined specification that supports sharing). Our specification hides the fact that each key is part of an underlying shared data structure, allowing straightforward high-level reasoning about keys and values.

Abstractly, the state of an index can be seen as a partial function mapping keys to values[2]:

$$H\colon \mathsf{Keys} \rightharpoonup \mathsf{Vals}$$

There are three basic operations on an index – `search`, `insert` and `remove` – which operate on index `h` (with current state H) as follows:

- $\mathtt{search}(\mathtt{h}, \mathtt{k})$ looks for the key `k` in the index. It returns $H(\mathtt{k})$ if it is defined, and nil otherwise.
- $\mathtt{insert}(\mathtt{h}, \mathtt{k}, \mathtt{v})$ tries to modify H to associate the key `k` with value `v`. If $\mathtt{k} \in \mathrm{dom}(H)$ then `insert` does nothing. Otherwise it modifies the shared index to $H \uplus \{\mathtt{k} \mapsto \mathtt{v}\}$.
- $\mathtt{remove}(\mathtt{h}, \mathtt{k})$ tries to remove the value of the key `k` from the index. If $\mathtt{k} \notin \mathrm{dom}(H)$ then `remove` does nothing. Otherwise it rewrites the index to $H \setminus \{\mathtt{k}\}$.

This view of operations on the index is appealingly simple, but cannot be used for practical concurrent reasoning. This is because it depends on *global* knowledge of the underlying index H. To reason in this way, a thread would require perfect knowledge of the behaviour of other threads.

To avoid this, we give a specification that breaks the index up by key value. Our specification allows threads to hold the exclusive ownership of an individual key. Each key in the index is represented by a predicate, either in or out depending on whether the key is associated with a value or not. The predicates have this intuitive interpretation:

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

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

These predicates combine knowledge about state – whether a key is in the index – with knowledge about ownership – whether the thread is allowed to alter that key. A thread holding the predicate for a given key knows the value of the key, and can be sure that no other thread will modify it. This entangling of state with ownership is essential to our approach: each predicate is invariant under the behaviour of other threads, meaning its implementation can be abstracted.

The index operations have the following specifications with respect to these predicates:

$$\begin{array}{rll}
\{\mathsf{in}(\mathtt{h}, \mathtt{k}, v)\} & \mathtt{r} := \mathtt{search}(\mathtt{h}, \mathtt{k}) & \{\mathsf{in}(\mathtt{h}, \mathtt{k}, v) \wedge \mathtt{r} = v\} \\
\{\mathsf{out}(\mathtt{h}, \mathtt{k})\} & \mathtt{r} := \mathtt{search}(\mathtt{h}, \mathtt{k}) & \{\mathsf{out}(\mathtt{h}, \mathtt{k}) \wedge \mathtt{r} = \mathsf{nil}\} \\
\{\mathsf{in}(\mathtt{h}, \mathtt{k}, v')\} & \mathtt{insert}(\mathtt{h}, \mathtt{k}, \mathtt{v}) & \{\mathsf{in}(\mathtt{h}, \mathtt{k}, v')\} \\
\{\mathsf{out}(\mathtt{h}, \mathtt{k})\} & \mathtt{insert}(\mathtt{h}, \mathtt{k}, \mathtt{v}) & \{\mathsf{in}(\mathtt{h}, \mathtt{k}, \mathtt{v})\} \\
\{\mathsf{in}(\mathtt{h}, \mathtt{k}, v)\} & \mathtt{remove}(\mathtt{h}, \mathtt{k}) & \{\mathsf{out}(\mathtt{h}, \mathtt{k})\} \\
\{\mathsf{out}(\mathtt{h}, \mathtt{k})\} & \mathtt{remove}(\mathtt{h}, \mathtt{k}) & \{\mathsf{out}(\mathtt{h}, \mathtt{k})\}
\end{array}$$

Predicates can be composed using the separating conjunction $*$, indicating that they hold independently of each other. Note that our specification allows us to reason about an index as a collection of disjoint, independent elements, despite the fact that indexes are generally implemented as a single shared data structure.

Each predicate represents exclusive ownership of a particular key. Our specification represents this fact by exposing the following axiom:

$$\begin{pmatrix} (\mathsf{in}(h, k, v) \vee \mathsf{out}(h, k)) * \\ (\mathsf{in}(h, k, v') \vee \mathsf{out}(h, k)) \end{pmatrix} \implies \mathsf{false}$$

Given the above specifications, we can reason locally about programs that use concurrent indexes. Consider for example the following simple program:

$$\begin{array}{l}
\mathtt{r} := \mathtt{search}(\mathtt{h}, \mathtt{k}_2); \\
\mathtt{insert}(\mathtt{h}, \mathtt{k}_1, \mathtt{r}) \;\; \| \;\; \mathtt{remove}(\mathtt{h}, \mathtt{k}_2)
\end{array}$$

This program retrieves the value v associated with the key $\mathtt{k}_2$. It then concurrently associates v with the key $\mathtt{k}_1$ and removes the key $\mathtt{k}_2$. When the program completes, $\mathtt{k}_1$ will be associated with v, and $\mathtt{k}_2$ will have been removed from the index. This specification can be expressed as:

$$\{\mathsf{out}(\mathtt{h}, \mathtt{k}_1) * \mathsf{in}(\mathtt{h}, \mathtt{k}_2, v)\} \; - \; \{\mathsf{in}(\mathtt{h}, \mathtt{k}_1, v) * \mathsf{out}(\mathtt{h}, \mathtt{k}_2)\}$$

[2] Where possible, we treat the key and value sets abstractly. Implementations require certain properties of these sets, however: all require keys to be comparable for equality, hash tables require the ability to compute hashes of keys, and B-trees require a linear ordering on keys.

We can prove this specification as follows:

$$
\begin{array}{c}
\left\{\mathsf{out}(\mathtt{h},\mathtt{k}_1) * \mathsf{in}(\mathtt{h},\mathtt{k}_2,v)\right\} \\
\mathtt{r} := \mathtt{search}(\mathtt{h},\mathtt{k}_2); \\
\left\{\mathsf{out}(\mathtt{h},\mathtt{k}_1) * \mathsf{in}(\mathtt{h},\mathtt{k}_2,v) \wedge \mathtt{r} = v\right\} \\
\begin{array}{c|c}
\left\{\mathsf{out}(\mathtt{h},\mathtt{k}_1) \wedge \mathtt{r} = v\right\} & \left\{\mathsf{in}(\mathtt{h},\mathtt{k}_2,v)\right\} \\
\mathtt{insert}(\mathtt{h},\mathtt{k}_1,\mathtt{r}) & \mathtt{remove}(\mathtt{h},\mathtt{k}_2) \\
\left\{\mathsf{in}(\mathtt{h},\mathtt{k}_1,v)\right\} & \left\{\mathsf{out}(\mathtt{h},\mathtt{k}_2)\right\}
\end{array} \\
\left\{\mathsf{in}(\mathtt{h},\mathtt{k}_1,v) * \mathsf{out}(\mathtt{h},\mathtt{k}_2)\right\}
\end{array}
$$

In this proof, the `search` operation first uses the predicate $\mathsf{in}(\mathtt{h},\mathtt{k}_2,v)$ to retrieve the value v. Then, the parallel rule hands `insert` and `remove` the $\mathsf{out}(\mathtt{h},\mathtt{k}_1)$ and $\mathsf{in}(\mathtt{h},\mathtt{k}_2,v)$ predicates respectively. The postcondition of the program consists of the separating conjunction of the two thread postconditions.

3.1 Example: Map

A common operation on a concurrent index is applying a particular function to every value held in the index: *mapping* the function onto the index. We consider a simple algorithm `rangeMap` that maps function f (implemented by `f`) onto keys within a specified range. We implement `rangeMap` with a divide-and-conquer approach, which splits the key range into sub-intervals on which the map operation is recursively applied in parallel.

```
rangeMap(h, $k_1$, $k_2$) {
 if ($k_1$ = $k_2$) {
   r := search(h, $k_1$);
   if (r $\neq$ nil) {
     remove(h, $k_1$);
     r := f(r);
     insert(h, $k_1$, r);
   }
 } else {
    rangeMap(h, $k_1$, $k_1$+(($k_2$-$k_1$)/2))
    || rangeMap(h, $k_1$+(($k_2$-$k_1$)/2)+1, $k_2$)
}}
```

We specify `rangeMap` as follows, where S is a set of key-value pairs:

$$
\begin{array}{c}
\left\{\begin{array}{l}\circledast_{\mathtt{k}_1 \le i \le \mathtt{k}_2}.\, (\mathsf{out}(\mathtt{h},i) \wedge i \notin \mathsf{keys}(S)) \vee \\ \qquad\qquad (\exists v.\, \mathsf{in}(\mathtt{h},i,v) \wedge (i,v) \in S)\end{array}\right\} \\
\mathtt{rangeMap}(\mathtt{h},\mathtt{k}_1,\mathtt{k}_2) \\
\left\{\begin{array}{l}\circledast_{\mathtt{k}_1 \le i \le \mathtt{k}_2}.\, (\mathsf{out}(\mathtt{h},i) \wedge i \notin \mathsf{keys}(S)) \vee \\ \qquad\quad (\exists v.\, \mathsf{in}(\mathtt{h},i,f(v)) \wedge (i,v) \in S)\end{array}\right\}
\end{array}
$$

(Here, $\circledast$ is the iterated separating conjunction. That is, $\circledast_{x \in \{1,2,3\}}.\, P$ is equivalent to $P[1/x] * P[2/x] * P[3/x]$. The set $\mathsf{keys}(S)$ is the set of keys associated with values in S.)

In the specification, the logical variable S describes the initial state of the index (in the key range $[\mathtt{k}_1, \mathtt{k}_2]$). Assuming that S contains at most one key-value pair for each key, the key i (for $\mathtt{k}_1 \le i \le \mathtt{k}_2$) initially has value v if and only if $(i, v) \in S$. After execution of `rangeMap`, the postcondition ensures that if the key i had and initial value v, then it now has value $f(v)$, and if it had no value then it still has no value. A proof that `rangeMap` conforms to this specification is given in Figure 1.

```
$\left\{\begin{array}{l}\circledast_{\mathtt{k}_1 \le i \le \mathtt{k}_2}.\, (\mathsf{out}(\mathtt{h},i) \wedge i \notin \mathsf{keys}(S)) \vee \\ \qquad\qquad (\exists v.\, \mathsf{in}(\mathtt{h},i,v) \wedge (i,v) \in S)\end{array}\right\}$
rangeMap(h, $k_1$, $k_2$) {
 if ($k_1$ = $k_2$) {
  $\left\{\begin{array}{l}\mathtt{k}_1 = \mathtt{k}_2 \wedge ((\mathsf{out}(\mathtt{h},\mathtt{k}_1) \wedge \mathtt{k}_1 \notin \mathsf{keys}(S)) \vee \\ (\exists v.\, \mathsf{in}(\mathtt{h},\mathtt{k}_1,v) \wedge (\mathtt{k}_1,v) \in S))\end{array}\right\}$
  r := search(h, $k_1$);
  $\left\{\begin{array}{l}((\mathsf{out}(\mathtt{h},\mathtt{k}_1) \wedge \mathtt{k}_1 \notin \mathsf{keys}(S) \wedge \mathtt{r} = \mathsf{nil}) \vee \\ (\mathsf{in}(\mathtt{h},\mathtt{k}_1,\mathtt{r}) \wedge (\mathtt{k}_1,\mathtt{r}) \in S)) \wedge \mathtt{k}_1 = \mathtt{k}_2\end{array}\right\}$
  if (r $\neq$ nil) {
   $\left\{\mathsf{in}(\mathtt{h},\mathtt{k}_1,\mathtt{r}) \wedge (\mathtt{k}_1,\mathtt{r}) \in S \wedge \mathtt{k}_1 = \mathtt{k}_2\right\}$
    remove(h, $k_1$);
   $\left\{\mathsf{out}(h,\mathtt{k}_1) \wedge (\mathtt{k}_1,\mathtt{r}) \in S \wedge \mathtt{k}_1 = \mathtt{k}_2\right\}$
    r := f(r);
   $\left\{\exists v.\, \mathsf{out}(\mathtt{h},\mathtt{k}_1) \wedge (\mathtt{k}_1,v) \in S \wedge \mathtt{r} = f(v) \wedge \mathtt{k}_1 = \mathtt{k}_2\right\}$
    insert(h, $k_1$, r);
   $\left\{\exists v.\, \mathsf{in}(\mathtt{h},\mathtt{k}_1,f(v)) \wedge (\mathtt{k}_1,v) \in S \wedge \mathtt{k}_1 = \mathtt{k}_2\right\}$
  }
  $\left\{\begin{array}{l}\mathtt{k}_1 = \mathtt{k}_2 \wedge ((\mathsf{out}(\mathtt{h},\mathtt{k}_1) \wedge \mathtt{k}_1 \notin \mathsf{keys}(S)) \vee \\ (\exists v.\, \mathsf{in}(\mathtt{h},\mathtt{k}_1,f(v)) \wedge (\mathtt{k}_1,v) \in S))\end{array}\right\}$
 } else {
  $\left\{\begin{array}{l}\circledast_{\mathtt{k}_1 \le i \le \left\lfloor\frac{\mathtt{k}_1+\mathtt{k}_2}{2}\right\rfloor}.\left(\begin{array}{l}(\mathsf{out}(\mathtt{h},i) \wedge i \notin \mathsf{keys}(S)) \vee \\ (\exists v.\, \mathsf{in}(\mathtt{h},i,v) \wedge (i,v) \in S)\end{array}\right) * \\ \circledast_{\left\lfloor\frac{\mathtt{k}_1+\mathtt{k}_2}{2}\right\rfloor < i \le \mathtt{k}_2}.\left(\begin{array}{l}(\mathsf{out}(\mathtt{h},i) \wedge i \notin \mathsf{keys}(S)) \vee \\ (\exists v.\, \mathsf{in}(\mathtt{h},i,v) \wedge (i,v) \in S)\end{array}\right)\end{array}\right\}$
  // Apply the PAR rule.
  rangeMap(h, $k_1$, $k_1$+(($k_2$-$k_1$)/2))
  || rangeMap(h, $k_1$+(($k_2$-$k_1$)/2)+1, $k_2$)
  $\left\{\begin{array}{l}\circledast_{\mathtt{k}_1 \le i \le \left\lfloor\frac{\mathtt{k}_1+\mathtt{k}_2}{2}\right\rfloor}.\left(\begin{array}{l}(\mathsf{out}(\mathtt{h},i) \wedge i \notin \mathsf{keys}(S)) \vee \\ (\exists v.\, \mathsf{in}(\mathtt{h},i,f(v)) \wedge (i,v) \in S)\end{array}\right) * \\ \circledast_{\left\lfloor\frac{\mathtt{k}_1+\mathtt{k}_2}{2}\right\rfloor < i \le \mathtt{k}_2}.\left(\begin{array}{l}(\mathsf{out}(\mathtt{h},i) \wedge i \notin \mathsf{keys}(S)) \vee \\ (\exists v.\, \mathsf{in}(\mathtt{h},i,f(v)) \wedge (i,v) \in S)\end{array}\right)\end{array}\right\}$
}}
$\left\{\begin{array}{l}\circledast_{\mathtt{k}_1 \le i \le \mathtt{k}_2}.\, (\mathsf{out}(\mathtt{h},i) \wedge i \notin \mathsf{keys}(S)) \vee \\ \qquad\quad (\exists v.\, \mathsf{in}(\mathtt{h},i,f(v)) \wedge (i,v) \in S)\end{array}\right\}$
```

Figure 1. Proof for `rangeMap`.

`rangeMap` might not be considered truly typical of map operations, as it maps over a range of keys rather than the entire index. In §5, we introduce a specification for iterators, allowing all keys in an index to be enumerated. Using an iterator, we implement and verify a map function over all values in the index.

4. Index Specification: Sharing

The specification we defined in the previous section requires that each key in the index is accessed by at most one thread. However, often threads read and write to keys at the same time. In this section, we define a refined specification that allows for concurrent access to keys. As before, our specification hides implementation details and allows threads to reason locally.

Consider the following program:

$$\texttt{remove(h,k)} \;\|\; \texttt{r} := \texttt{search(h,k)} \qquad (1)$$

If we know at the start of the program that key `k` maps to some value v, we should be able to establish that there will not be a mapping from the key `k` at the end. However, we will not know the value of `r`, because we do not know at which point during the `remove` operation that the `search` operation will read the value associated with `k`.

Implementations have many different ways of handling the sharing of keys (for example using mutual exclusion locks or transactions), but at the abstract level they all behave in the same way. If a thread reads a key multiple times, the reads all return the same result, unless another thread also writes to that key.

Our refined specification is based on abstract predicates that express three facts about a given key:

1. whether there is a mapping from the key to some value in a set;
2. whether the thread holding the predicate can add or remove the value of the key in the index;
3. whether any other concurrently running threads (the *environment*) can add or remove the value of the key in the index.

These facts are related. If a key maps to a value in the index, but other threads are allowed to remove the value of the key, the current thread cannot assume the value will remain in the index. Our predicates therefore reflect the uncertainty generated by sharing in a local way.

We define the following set of predicates, parametric on key k and index h:

$\mathsf{in}_{\mathsf{def}}(h,k,v)_i$: there is a mapping from key k to value v and a thread can only modify this key if it has exclusive permission $(i = 1)$.

$\mathsf{out}_{\mathsf{def}}(h,k)_i$: there is no mapping from key k and a thread can only modify this key if it has exclusive permission $(i = 1)$.

$\mathsf{in}_{\mathsf{ins}}(h,k,S)_i$: there is a mapping from key k to a value in set S and threads can only insert values in set S at this key.

$\mathsf{out}_{\mathsf{ins}}(h,k,S)_i$: there may be a mapping from key k to a value in set S, threads can only insert values in set S at this key, and the current thread has not made such an insertion so far.

$\mathsf{in}_{\mathsf{rem}}(h,k,v)_i$: there may be a mapping from key k to value v, threads can only remove the value at this key, and the current thread has not done this so far.

$\mathsf{out}_{\mathsf{rem}}(h,k)_i$: there is no mapping from key k and threads can only remove the value at this key.

$\mathsf{unk}(h,k,S)_i$: there may be a mapping from key k to a value v in set S and threads can search, remove and insert any value in set S at this key.

$\mathsf{read}(h,k)$: there may be a mapping from key k to some value, the current thread may not change it, but other threads can make any modification.

The subscripts def, ins and rem and the fractional components $i \in (0,1]$ record the behaviours allowed by the current thread and its environment on key k.

Access to keys can be shared between threads. We represent this in our specification by splitting predicates. Our specification includes axioms which define the ways that predicates can be split and joined. For example:

$$\mathsf{in}_{\mathsf{rem}}(h,k,v)_{i+j} \iff \mathsf{in}_{\mathsf{rem}}(h,k,v)_i * \mathsf{in}_{\mathsf{rem}}(h,k,v)_j \qquad \text{if } i+j \leq 1$$

As in Boyland [2], fractional *permissions* are used to record splittings. A permission $i \in (0,1)$ records that a key is shared with other threads, while $i = 1$ records it is held exclusively by the current thread.

When a thread holds exclusive access to a key $(i = 1)$, the thread can add or remove the key freely. When a thread shares access to the key $(i \in (0,1))$, the subscripts def, ins and rem *restrict* what the thread and its environment are able to do. Subscript def specifies that no thread is able to modify the key. Subscript ins specifies that both thread and environment can insert on the key, but not remove the key, while subscript rem specifies the converse.

Modifying keys concurrently can result in different threads holding different predicates for the same key. For example, suppose a thread holds the $\mathsf{in}_{\mathsf{rem}}(h,k,v)_1$ predicate, which denotes that the key k has value v in the index. Since the permission is 1, this knowledge is assured. However, we can split this predicate into two halves, $\mathsf{in}_{\mathsf{rem}}(h,k,v)_{\frac{1}{2}}$ and $\mathsf{in}_{\mathsf{rem}}(h,k,v)_{\frac{1}{2}}$, and give each half to two sub-threads. Assume the first thread does not modify the key, but the second calls $\texttt{remove}(h,k)$, which has the following specification:

$$\{\mathsf{in}_{\mathsf{rem}}(\texttt{h},\texttt{k},v)_i\} \quad \texttt{remove(h,k)} \quad \{\mathsf{out}_{\mathsf{rem}}(\texttt{h},\texttt{k})_i\}$$

The result is uncertainty: one thread holds the $\mathsf{out}_{\mathsf{rem}}(h,k)_{\frac{1}{2}}$ predicate, stating that k is not in the index, while the other holds the $\mathsf{in}_{\mathsf{rem}}(h,k,v)_{\frac{1}{2}}$ predicate, stating that k may have associated value v. We define joining axioms that resolve this uncertainty. Since rem allows removal but not insertion, we know that once the key has been removed from the index, it stays removed. So $\mathsf{out}_{\mathsf{rem}}$ dominates $\mathsf{in}_{\mathsf{rem}}$, which is reflected in the following axiom:

$$\mathsf{in}_{\mathsf{rem}}(h,k,v)_i * \mathsf{out}_{\mathsf{rem}}(h,k)_j \implies \mathsf{out}_{\mathsf{rem}}(h,k)_{i+j} \qquad \text{if } i+j \leq 1$$

Some predicates take sets of value arguments, while others take singleton values. We use singleton values when we

		Thread		Env.	
Predicate	Perm.	Ins.	Rem.	Ins.	Rem.
$\mathsf{in}_{\mathsf{def}}$ / $\mathsf{out}_{\mathsf{def}}$	1	Yes	Yes	No	No
$\mathsf{in}_{\mathsf{def}}$ / $\mathsf{out}_{\mathsf{def}}$	i	No	No	No	No
$\mathsf{in}_{\mathsf{ins}}$ / $\mathsf{out}_{\mathsf{ins}}$	1	Yes	No	No	No
$\mathsf{in}_{\mathsf{ins}}$ / $\mathsf{out}_{\mathsf{ins}}$	i	Yes	No	Yes	No
$\mathsf{in}_{\mathsf{rem}}$ / $\mathsf{out}_{\mathsf{rem}}$	1	No	Yes	No	No
$\mathsf{in}_{\mathsf{rem}}$ / $\mathsf{out}_{\mathsf{rem}}$	i	No	Yes	No	Yes
unk	i	Yes	Yes	Yes	Yes
read	-	No	No	Yes	Yes

Figure 2. Predicates and their interference.

know a key has that value. We use a set of values when concurrent inserts are possible (that is, in the ins and unk cases), because we cannot know which thread will be the first to insert. However, if a value is inserted, it will be one of the values in the set S.

Our full specification is given in Figure 3. The choice of predicates is not arbitrary; each represents a stable combination of facts about the key k and the behaviours permitted by the thread and environment. Figure 2 shows how various combinations of fractional permissions and subscripts correspond to various behaviours. Our predicates give almost complete coverage of all possible combinations. The missing combinations are either cases where the current thread has no access to a key, or where it is only safe to conclude that a key has an unknown value, in which case we can use one of the read or unk predicates. We do not claim that our specification is definitive, just one natural choice. We expect to adapt our specification when looking at real-world applications such as the POSIX file system, the concurrent database algorithm ARIES, and `java.util.concurrent`. We believe that our specification is robust enough to be able to support such applications with minor modification.

4.1 Proving Simple Examples

Recall the program labelled (1) with which we began this section. This program satisfies the following specifications:

$$\{\mathsf{in}_{\mathsf{def}}(\mathtt{h},\mathtt{k},v)_1\} \;-\; \{\mathsf{out}_{\mathsf{def}}(\mathtt{h},\mathtt{k})_1\}$$
$$\{\mathsf{out}_{\mathsf{def}}(\mathtt{h},\mathtt{k})_1\} \;-\; \{\mathsf{out}_{\mathsf{def}}(\mathtt{h},\mathtt{k})_1\}$$

Using our abstract specifications, we can prove the first of these specifications as follows:

$$\{\mathsf{in}_{\mathsf{def}}(\mathtt{h},\mathtt{k},v)_1\}$$
$$\{\mathsf{in}_{\mathsf{def}}(\mathtt{h},\mathtt{k},v)_1 * \mathsf{read}(\mathtt{h},\mathtt{k})\}$$
$$\begin{array}{l||r}
\{\mathsf{in}_{\mathsf{def}}(\mathtt{h},\mathtt{k},v)_1\} & \{\mathsf{read}(\mathtt{h},\mathtt{k})\} \\
\mathtt{remove}(\mathtt{h},\mathtt{k}) & \mathtt{r} := \mathtt{search}(\mathtt{h},\mathtt{k}) \\
\{\mathsf{out}_{\mathsf{def}}(\mathtt{h},\mathtt{k})_1\} & \{\mathsf{read}(\mathtt{h},\mathtt{k})\}
\end{array}$$
$$\{\mathsf{out}_{\mathsf{def}}(\mathtt{h},\mathtt{k})_1 * \mathsf{read}(\mathtt{h},\mathtt{k})\}$$
$$\{\mathsf{out}_{\mathsf{def}}(\mathtt{h},\mathtt{k})_1\}$$

The proof starts with the predicate $\mathsf{in}_{\mathsf{def}}(\mathtt{h},\mathtt{k},v)_1$, which specifies that there is a mapping from key `k` to a value v in the index. The def subscript asserts that no other thread can modify the value mapped by this key. We use the following axiom to create a $\mathsf{read}(\mathtt{h},\mathtt{k})$ predicate:

$$X_i \iff X_i * \mathsf{read}(h,k)$$

This allows the right-hand thread to perform a simple `search` operation, although the postcondition establishes nothing about the result. This captures the fact that we do not know at which point during the `remove` operation the `search` operation will read the key's value. The $\mathsf{in}_{\mathsf{def}}(\mathtt{h},\mathtt{k},v)_1$ predicate allows the left-hand thread to remove the value successfully, as we know that it is the only thread changing the shared state for the key `k`. When both threads finish their execution we use the same axiom to merge $\mathsf{read}(\mathtt{h},\mathtt{k})$ back into the $\mathsf{out}_{\mathsf{def}}(\mathtt{h},\mathtt{k})_1$. We can prove the second specification in a similar fashion.

We can establish natural specifications for all the various combinations of `insert`, `remove` and `search`. For example, consider the parallel composition of two removes on the same key `k`:

$$\mathtt{remove}(\mathtt{h},\mathtt{k}) \;\|\; \mathtt{remove}(\mathtt{h},\mathtt{k})$$

Regardless of whether `k` is in the index, we definitely know that there will be no mapping from key `k` afterwards. By splitting the predicates, we can share this knowledge between the threads.

$$\{\mathsf{in}_{\mathsf{def}}(\mathtt{h},\mathtt{k},v)_1\}$$
$$\{\mathsf{in}_{\mathsf{rem}}(\mathtt{h},\mathtt{k},v)_1\}$$
$$\{\mathsf{in}_{\mathsf{rem}}(\mathtt{h},\mathtt{k},v)_{\frac{1}{2}} * \mathsf{in}_{\mathsf{rem}}(\mathtt{h},\mathtt{k},v)_{\frac{1}{2}}\}$$
$$\begin{array}{c||c}
\{\mathsf{in}_{\mathsf{rem}}(\mathtt{h},\mathtt{k},v)_{\frac{1}{2}}\} & \{\mathsf{in}_{\mathsf{rem}}(\mathtt{h},\mathtt{k},v)_{\frac{1}{2}}\} \\
\mathtt{remove}(\mathtt{h},\mathtt{k}) & \mathtt{remove}(\mathtt{h},\mathtt{k}) \\
\{\mathsf{out}_{\mathsf{rem}}(\mathtt{h},\mathtt{k})_{\frac{1}{2}}\} & \{\mathsf{out}_{\mathsf{rem}}(\mathtt{h},\mathtt{k})_{\frac{1}{2}}\}
\end{array}$$
$$\{\mathsf{out}_{\mathsf{rem}}(\mathtt{h},\mathtt{k})_{\frac{1}{2}} * \mathsf{out}_{\mathsf{rem}}(\mathtt{h},\mathtt{k})_{\frac{1}{2}}\}$$
$$\{\mathsf{out}_{\mathsf{def}}(\mathtt{h},\mathtt{k})_1\}$$

We cannot always establish the exact state of an index at all points during a program, but our specification will always allow us to be as precise as possible. For example, consider the following program:

$$\begin{array}{c}
\mathtt{remove}(\mathtt{h},\mathtt{k}) \;\|\; \mathtt{insert}(\mathtt{h},\mathtt{k},\mathtt{v}) \\
\mathtt{remove}(\mathtt{h},\mathtt{k})
\end{array}$$

When run in a state where key `k` is initially unassigned, we will not know if there is a mapping from key `k` in the index at the end of the parallel call. However, after the final `remove`

SPECIFICATIONS:

$$
\begin{array}{rll}
\{\mathsf{in}_{\mathsf{def}}(\mathtt{h},\mathtt{k},v)_i\} & \mathtt{r} := \mathtt{search}(\mathtt{h},\mathtt{k}) & \{\mathsf{in}_{\mathsf{def}}(\mathtt{h},\mathtt{k},v)_i \land \mathtt{r} = v\} \\
\{\mathsf{out}_{\mathsf{def}}(\mathtt{h},\mathtt{k})_i\} & \mathtt{r} := \mathtt{search}(\mathtt{h},\mathtt{k}) & \{\mathsf{out}_{\mathsf{def}}(\mathtt{h},\mathtt{k})_i \land \mathtt{r} = \mathsf{nil}\} \\
\{\mathsf{in}_{\mathsf{ins}}(\mathtt{h},\mathtt{k},S)_i\} & \mathtt{r} := \mathtt{search}(\mathtt{h},\mathtt{k}) & \{\mathsf{in}_{\mathsf{ins}}(\mathtt{h},\mathtt{k},S)_i \land \mathtt{r} \in S\} \\
\{\mathsf{out}_{\mathsf{ins}}(\mathtt{h},\mathtt{k},S)_i\} & \mathtt{r} := \mathtt{search}(\mathtt{h},\mathtt{k}) & \{(\mathsf{out}_{\mathsf{ins}}(\mathtt{h},\mathtt{k},S)_i \land \mathtt{r} = \mathsf{nil}) \lor (\mathsf{in}_{\mathsf{ins}}(\mathtt{h},\mathtt{k},S)_i \land \mathtt{r} \in S)\} \\
\{\mathsf{in}_{\mathsf{rem}}(\mathtt{h},\mathtt{k},v)_i\} & \mathtt{r} := \mathtt{search}(\mathtt{h},\mathtt{k}) & \{(\mathsf{in}_{\mathsf{rem}}(\mathtt{h},\mathtt{k},v)_i \land \mathtt{r} = v) \lor (\mathsf{out}_{\mathsf{rem}}(\mathtt{h},\mathtt{k})_i \land \mathtt{r} = \mathsf{nil})\} \\
\{\mathsf{out}_{\mathsf{rem}}(\mathtt{h},\mathtt{k})_i\} & \mathtt{r} := \mathtt{search}(\mathtt{h},\mathtt{k}) & \{\mathsf{out}_{\mathsf{rem}}(\mathtt{h},\mathtt{k})_i \land \mathtt{r} = \mathsf{nil}\} \\
\{\mathsf{unk}(\mathtt{h},\mathtt{k},S)_i\} & \mathtt{r} := \mathtt{search}(\mathtt{h},\mathtt{k}) & \{\mathsf{unk}(\mathtt{h},\mathtt{k},S)_i \land (\mathtt{r} \in S \lor \mathtt{r} = \mathsf{nil})\} \\
\{\mathsf{read}(\mathtt{h},\mathtt{k})\} & \mathtt{r} := \mathtt{search}(\mathtt{h},\mathtt{k}) & \{\mathsf{read}(\mathtt{h},\mathtt{k})\} \\
\\
\{\mathsf{in}_{\mathsf{def}}(\mathtt{h},\mathtt{k},v)_i\} & \mathtt{insert}(\mathtt{h},\mathtt{k},\mathtt{v}') & \{\mathsf{in}_{\mathsf{def}}(\mathtt{h},\mathtt{k},v)_i\} \\
\{\mathsf{out}_{\mathsf{def}}(\mathtt{h},\mathtt{k})_1\} & \mathtt{insert}(\mathtt{h},\mathtt{k},\mathtt{v}) & \{\mathsf{in}_{\mathsf{def}}(\mathtt{h},\mathtt{k},\mathtt{v})_1\} \\
\{(\mathsf{in}_{\mathsf{ins}}(\mathtt{h},\mathtt{k},S)_i \lor \mathsf{out}_{\mathsf{ins}}(\mathtt{h},\mathtt{k},S)_i) \land \mathtt{v} \in S\} & \mathtt{insert}(\mathtt{h},\mathtt{k},\mathtt{v}) & \{\mathsf{in}_{\mathsf{ins}}(\mathtt{h},\mathtt{k},S)_i\} \\
\{\mathsf{unk}(\mathtt{h},\mathtt{k},S)_i \land \mathtt{v} \in S\} & \mathtt{insert}(\mathtt{h},\mathtt{k},\mathtt{v}) & \{\mathsf{unk}(\mathtt{h},\mathtt{k},S)_i\} \\
\\
\{\mathsf{in}_{\mathsf{def}}(\mathtt{h},\mathtt{k},v)_1\} & \mathtt{remove}(\mathtt{h},\mathtt{k}) & \{\mathsf{out}_{\mathsf{def}}(\mathtt{h},\mathtt{k})_1\} \\
\{\mathsf{out}_{\mathsf{def}}(\mathtt{h},\mathtt{k})_i\} & \mathtt{remove}(\mathtt{h},\mathtt{k}) & \{\mathsf{out}_{\mathsf{def}}(\mathtt{h},\mathtt{k})_i\} \\
\{\mathsf{in}_{\mathsf{rem}}(\mathtt{h},\mathtt{k},v)_i \lor \mathsf{out}_{\mathsf{rem}}(\mathtt{h},\mathtt{k})_i\} & \mathtt{remove}(\mathtt{h},\mathtt{k}) & \{\mathsf{out}_{\mathsf{rem}}(\mathtt{h},\mathtt{k})_i\} \\
\{\mathsf{unk}(\mathtt{h},\mathtt{k},S)_i\} & \mathtt{remove}(\mathtt{h},\mathtt{k}) & \{\mathsf{unk}(\mathtt{h},\mathtt{k},S)_i\}
\end{array}
$$

AXIOMS:

$$
\begin{array}{rcll}
X_i * X_j & \Leftrightarrow & X_{i+j} & \text{if } i+j \leq 1 \\
\mathsf{in}_{\mathsf{ins}}(h,k,S)_i * \mathsf{out}_{\mathsf{ins}}(h,k,S)_j & \Rightarrow & \mathsf{in}_{\mathsf{ins}}(h,k,S)_{i+j} & \text{if } i+j \leq 1 \\
\mathsf{in}_{\mathsf{rem}}(h,k,v)_i * \mathsf{out}_{\mathsf{rem}}(h,k)_j & \Rightarrow & \mathsf{out}_{\mathsf{rem}}(h,k)_{i+j} & \text{if } i+j \leq 1 \\
\mathsf{in}_{\mathsf{def}}(h,k,v)_1 & \Leftrightarrow & \mathsf{in}_{\mathsf{rem}}(h,k,v)_1 & \\
\exists v \in S.\, \mathsf{in}_{\mathsf{def}}(h,k,v)_1 & \Leftrightarrow & \mathsf{in}_{\mathsf{ins}}(h,k,S)_1 & \\
\mathsf{out}_{\mathsf{def}}(h,k)_1 & \Leftrightarrow & \mathsf{out}_{\mathsf{rem}}(h,k)_1 \;\Leftrightarrow\; \mathsf{out}_{\mathsf{ins}}(h,k,S)_1 & \\
X_i & \Leftrightarrow & X_i * \mathsf{read}(h,k) & \\
\mathsf{read}(h,k) & \Leftrightarrow & \mathsf{read}(h,k) * \mathsf{read}(h,k) & \\
\mathsf{unk}(h,k,S)_1 & \Leftrightarrow & \mathsf{out}_{\mathsf{def}}(h,k)_1 \lor \exists v \in S.\, \mathsf{in}_{\mathsf{def}}(h,k,v)_1 &
\end{array}
$$

CONTRADICTION AXIOMS:

$$
\begin{array}{rcll}
X_i * X_j & \Rightarrow & \mathsf{false} & \text{if } i+j > 1 \\
\mathsf{in}_{\mathsf{def}}(h,k,v)_i * X_j & \Rightarrow & \mathsf{false} & \text{if } X \neq \mathsf{in}_{\mathsf{def}}(h,k,v) \\
\mathsf{out}_{\mathsf{def}}(h,k)_i * X_j & \Rightarrow & \mathsf{false} & \text{if } X \neq \mathsf{out}_{\mathsf{def}}(h,k) \\
(\mathsf{in}_{\mathsf{ins}}(h,k,S)_i \lor \mathsf{out}_{\mathsf{ins}}(h,k,S)_i) * X_j & \Rightarrow & \mathsf{false} & \text{if } X \neq \mathsf{in}_{\mathsf{ins}}(h,k,S) \land X \neq \mathsf{out}_{\mathsf{ins}}(h,k,S) \\
(\mathsf{in}_{\mathsf{rem}}(h,k,v)_i \lor \mathsf{out}_{\mathsf{rem}}(h,k)_i) * X_j & \Rightarrow & \mathsf{false} & \text{if } X \neq \mathsf{in}_{\mathsf{rem}}(h,k,v) \land X \neq \mathsf{out}_{\mathsf{rem}}(h,k) \\
(\mathsf{in}_{\mathsf{ins}}(h,k,S)_i * \mathsf{in}_{\mathsf{ins}}(h,k,S')_j) \lor (\mathsf{out}_{\mathsf{ins}}(h,k,S)_i * \mathsf{out}_{\mathsf{ins}}(h,k,S')_j) & \Rightarrow & \mathsf{false} & \text{if } S \neq S' \\
\mathsf{unk}(h,k,S)_i * X_j & \Rightarrow & \mathsf{false} & \text{if } X \neq \mathsf{unk}(h,k,S)
\end{array}
$$

Figure 3. Full specification for concurrent indexes. X denotes $\mathsf{in}_{\mathsf{def}}(h,k,v)$, $\mathsf{out}_{\mathsf{def}}(h,k)$, $\mathsf{in}_{\mathsf{ins}}(h,k,S)$, $\mathsf{out}_{\mathsf{ins}}(h,k,S)$, $\mathsf{in}_{\mathsf{rem}}(h,k,v)$, $\mathsf{out}_{\mathsf{rem}}(h,k)$ or $\mathsf{unk}(h,k,S)$ in the axioms.

```
$\{\exists i \in (0,1].\ \circledast_{v'}.\ \mathsf{unk}(\mathtt{memo}, v', \{f(v')\})_i\}$
memoized_f(v) {
  $\{\circledast_{v'}.\ \mathsf{unk}(\mathtt{memo}, v', \{f(v')\})_i\}$
  // frame the irrelevant values off
  $\{\mathsf{unk}(\mathtt{memo}, \mathtt{v}, \{f(\mathtt{v})\})_i\}$
  r := search(memo, v);
  $\{\mathsf{unk}(\mathtt{memo}, \mathtt{v}, \{f(\mathtt{v})\})_i \wedge (\mathtt{r} = f(\mathtt{v}) \vee \mathtt{r} = \mathsf{nil})\}$
  if (r = nil) {
    $\{\mathsf{unk}(\mathtt{memo}, \mathtt{v}, \{f(\mathtt{v})\})_i\}$
    r := f(v);
    $\{\mathsf{unk}(\mathtt{memo}, \mathtt{v}, \{f(\mathtt{v})\})_i \wedge \mathtt{r} = f(\mathtt{v})\}$
    insert(memo, v, r);
    $\{\mathsf{unk}(\mathtt{memo}, \mathtt{v}, \{f(\mathtt{v})\})_i \wedge \mathtt{r} = f(\mathtt{v})\}$
  }
  $\{\mathsf{unk}(\mathtt{memo}, \mathtt{v}, \{f(\mathtt{v})\})_i \wedge \mathtt{r} = f(\mathtt{v})\}$
  // frame the values back on
  $\{\mathtt{r} = f(\mathtt{v}) \wedge \circledast_{v'}.\ \mathsf{unk}(\mathtt{memo}, v', \{f(v')\})_i\}$
  return r;
}
$\{\mathrm{ret} = f(\mathtt{v}) \wedge \exists i \in (0,1].\ \circledast_{v'}.\ \mathsf{unk}(\mathtt{memo}, v', \{f(v')\})_i\}$
```

Figure 4. Proof outline for `memoized_f`.

operation we know that the key `k` will be unassigned.

$$\{\mathsf{out}_{\mathsf{def}}(\mathtt{h},\mathtt{k})_1\}$$
$$\{\mathsf{out}_{\mathsf{def}}(\mathtt{h},\mathtt{k})_1 \vee \mathsf{in}_{\mathsf{def}}(\mathtt{h},\mathtt{k},\mathtt{v})_1\}$$
$$\{\mathsf{unk}(\mathtt{h},\mathtt{k},\{\mathtt{v}\})_1\}$$
$$\{\mathsf{unk}(\mathtt{h},\mathtt{k},\{\mathtt{v}\})_{\frac{1}{2}} * \mathsf{unk}(\mathtt{h},\mathtt{k},\{\mathtt{v}\})_{\frac{1}{2}}\}$$
$$\begin{array}{c||c}
\{\mathsf{unk}(\mathtt{h},\mathtt{k},\{\mathtt{v}\})_{\frac{1}{2}}\} & \{\mathsf{unk}(\mathtt{h},\mathtt{k},\{\mathtt{v}\})_{\frac{1}{2}}\} \\
\mathtt{remove}(\mathtt{h},\mathtt{k}) & \mathtt{insert}(\mathtt{h},\mathtt{k},\mathtt{v}) \\
\{\mathsf{unk}(\mathtt{h},\mathtt{k},\{\mathtt{v}\})_{\frac{1}{2}}\} & \{\mathsf{unk}(\mathtt{h},\mathtt{k},\{\mathtt{v}\})_{\frac{1}{2}}\}
\end{array}$$
$$\{\mathsf{unk}(\mathtt{h},\mathtt{k},\{\mathtt{v}\})_{\frac{1}{2}} * \mathsf{unk}(\mathtt{h},\mathtt{k},\{\mathtt{v}\})_{\frac{1}{2}}\}$$
$$\{\mathsf{unk}(\mathtt{h},\mathtt{k},\{\mathtt{v}\})_1\}$$
$$\{\mathsf{out}_{\mathsf{def}}(\mathtt{h},\mathtt{k})_1 \vee \mathsf{in}_{\mathsf{def}}(\mathtt{h},\mathtt{k},\mathtt{v})_1\}$$
$$\mathtt{remove}(\mathtt{h},\mathtt{k})$$
$$\{\mathsf{out}_{\mathsf{def}}(\mathtt{h},\mathtt{k})_1\}$$

The key step in this proof is the use of the final axiom from Figure 3 to convert a complete unk predicate into the disjunction of an in and out predicate. In both cases, the `remove` operation results in an index where the key `k` is definitely unassigned.

4.2 Example: Memoization

A common client application of indexes is memoization: storing the results of expensive computations to avoid having to recompute them. Our specification can verify that a memoized function gives the same result as the original function.

Suppose that `f` is a side-effect free procedure implementing the (mathematical) function f. A memoized version of `f`, `memoized_f`, can be implemented using the index `memo` as follows:

```
$\{\exists i \in (0,1].\ \circledast_{v'}.\ \mathsf{unk}(\mathtt{memo}, v', \{f(v')\})_i\}$
evict_f() {
  while (...) {
    $\{\circledast_{v'}.\ \mathsf{unk}(\mathtt{memo}, v', \{f(v')\})_i\}$
    k := nondet();
    // frame the irrelevant values off
    $\{\mathsf{unk}(\mathtt{memo}, \mathtt{k}, \{f(\mathtt{k})\})_i\}$
    remove(memo,k);
    $\{\mathsf{unk}(\mathtt{memo}, \mathtt{k}, \{f(\mathtt{k})\})_i\}$
} }
$\{\exists i \in (0,1].\ \circledast_{v'}.\ \mathsf{unk}(\mathtt{memo}, v', \{f(v')\})_i\}$
```

Figure 5. Proof outline for `evict_f`.

```
memoized_f(v) {
  r := search(memo, v);
  if (r = nil) {
    r := f(v);
    insert(memo, v, r);
  }
  return r;
}
```

We give `memoized_f` the following specification:

$$\{\mathsf{memo}\}\ \mathtt{r} := \mathtt{memoized\_f}(\mathtt{v})\ \{\mathtt{r} = f(\mathtt{v}) \wedge \mathsf{memo}\}$$

where the abstract predicate memo is

$$\mathsf{memo} \quad \triangleq \quad \exists i \in (0,1].\ \circledast_{v'}.\ \mathsf{unk}(\mathtt{memo}, v', \{f(v')\})_i$$

The definition of memo states that, for each value v', we do not know if v' is in the index. The predicate is splittable: that is, memo $\Leftrightarrow$ memo $*$ memo. The `memoized_f` specification therefore allows calls to `f` to be replaced with `memoized_f`, even in parallel. A proof of the specification for `memoized_f` is shown in Figure 4.

Evicting memoised values. We may want to periodically *evict* memoised values from the index, for example to ensure that the number of stored values does not grow indefinitely. Using our index specification, we can show that values can be evicted in parallel with `memoised_f()`.

We model eviction by the function `evict_f`, which nondeterministically removes keys from the index:

```
evict_f() {
  while (...) {
    k := nondet();
    remove(memo,k);
} }
```

where `nondet()` returns an arbitrary key value and the Boolean assertion for `while` is not given. (A more nuanced eviction function might store timestamps along with the memoised values, and evict only old values. For simplicity, we choose not to model this.)

We give `evict_f` the following specification:

$$\{\mathsf{memo}\}\ \mathtt{evict\_f}()\ \{\mathsf{memo}\}$$

A proof of this specification is given in Figure 5. Because we can split and join memo arbitrarily, we can reason as follows:

$$\begin{array}{c}
\{\mathsf{memo}\} \\
\{\mathsf{memo} * \mathsf{memo}\} \\
\begin{array}{c||l}
\{\mathsf{memo}\} & \{\mathsf{memo}\} \\
\mathtt{evict\_f}() & \mathtt{r} := \mathtt{memoised\_f}(\mathtt{v}) \\
\{\mathsf{memo}\} & \{\mathsf{memo} \land \mathtt{r} = f(\mathtt{v})\}
\end{array} \\
\{\mathsf{memo} * (\mathsf{memo} \land \mathtt{r} = f(\mathtt{v}))\} \\
\{\mathsf{memo} \land \mathtt{r} = f(\mathtt{v})\}
\end{array}$$

Consequently, it is safe to run the memoised version of `f` in parallel with eviction from the index.

4.3 Example: The Sieve of Eratosthenes

Let us consider an example where many threads require write access to the same shared value in a concurrent index. We choose the Sieve of Eratosthenes [1, 13], an algorithm for generating all of the prime numbers up to a given maximum value `max`. The sieve is a simple algorithm, but it is representative of a class of algorithms where threads cooperatively race to delete elements of shared data. Similar behaviour occurs in databases when deleting stale records, and in rendering when removing objects outside of a clipped region.

The algorithm starts by constructing a set of integers from 2 (since 1 is not a prime number) to `max`. We use an index to represent the set of (candidate) prime numbers. A set can be viewed as an instance of an index where the set of values is a singleton (in this example, we use $\{0\}$). A key is either present, representing that it is in the set, or not: the value itself conveys no information. We assume a function `idxrange` that creates an index with mappings for keys in a specified range.

For each integer in the range $2 \mathinner{..} \lfloor\sqrt{\mathtt{max}}\rfloor$, a thread is created that removes multiples of that integer from the set. Once all threads have completed, the remaining elements of the set are exactly those with no factors in the range $2 \mathinner{..} \lfloor\sqrt{\mathtt{max}}\rfloor$ (excluding themselves), and hence exactly the prime numbers less than or equal to `max`.

The code for the implementation is given in Figure 6. The procedure `sieve` is the main sieve function, which uses the recursive `parwork` procedure to run each worker thread in parallel. The procedure `worker` is the implementation of the worker threads.

The specification for `sieve` is

$$\begin{array}{l}
\{\mathsf{emp} \land \mathtt{max} > 1\} \\
\quad \mathtt{x} := \mathtt{sieve}(\mathtt{max}) \\
\left\{\begin{array}{l}
\circledast_{i \in [2..\mathtt{max}]}.\ \mathsf{isPrime}(i) \Rightarrow \mathsf{in}_{\mathsf{def}}(\mathtt{x}, i, 0)_1 \\
\qquad\qquad \land \neg\mathsf{isPrime}(i) \Rightarrow \mathsf{out}_{\mathsf{def}}(\mathtt{x}, i)_1
\end{array}\right\}
\end{array}$$

where the predicate 'emp' denotes no resource at all, and the predicate '$\mathsf{isPrime}(i)$' holds exactly when i is prime. We also define the predicate '$\mathsf{fac}(i, v, v')$', which holds when i has a factor (distinct from itself) in the range $[v \mathinner{..} v']$:

$$\mathsf{fac}(i, v, v') \ \triangleq\ \exists j.\, v \le j \le v' \land j \ne i \land (i \bmod j) = 0$$

```
sieve(max) {
  idx := idxrange(2, max);
  parwork(2, max, idx);
  return idx;
}
parwork(v, max, idx) {          worker(v, max, idx) {
  if (v ≤ sqrt(max)) {            c := v + v;
    worker(v, max, idx)           while (c ≤ max)
    ||                              remove(idx, c);
    parwork(v+1, max, idx)          c := c + v;
} }                             } }
```

Figure 6. Prime sieve functions.

The proof that `sieve` meets its specification is given in Figure 7. This proof requires we establish the following specification for `worker`:

$$\begin{array}{l}
\{2 \le \mathtt{v} \land \circledast_{i \in [2..\mathtt{max}]}.\ \mathsf{in}_{\mathsf{rem}}(\mathtt{idx}, i, 0)_t\} \\
\quad \mathtt{worker}(\mathtt{v}, \mathtt{max}, \mathtt{idx}) \\
\left\{\begin{array}{l}
\circledast_{i \in [2..\mathtt{max}]}.\ \mathsf{fac}(i, \mathtt{v}, \mathtt{v}) \Rightarrow \mathsf{out}_{\mathsf{rem}}(\mathtt{idx}, i)_t \land \\
\qquad\qquad \neg\mathsf{fac}(i, \mathtt{v}, \mathtt{v}) \Rightarrow \mathsf{in}_{\mathsf{rem}}(\mathtt{idx}, i, 0)_t
\end{array}\right\}
\end{array}$$

This specification expresses that the worker removes all multiples of `v` from the set; any other elements will still be present unless they are removed by another thread. The fact that (for $v \le v'$)

$$\mathsf{fac}(i, v, v) \lor \mathsf{fac}(i, v+1, v') \iff \mathsf{fac}(i, v, v')$$

allows us to conclude that the `parwork` procedure eliminates exactly the set elements with factors different from themselves in the range $\mathtt{v} \mathinner{..} \mathtt{max}$. Since $p > 1$ is prime if and only if it has no factor in the range $2 \mathinner{..} \lfloor\sqrt{p}\rfloor$, for $i \in [2 \mathinner{..} \mathtt{max}]$

$$\neg\mathsf{fac}(i, 2, \lfloor\sqrt{\mathtt{max}}\rfloor) \iff \mathsf{isPrime}(i).$$

Together with the index axioms that allow rem predicates to be switched to def predicates when full permission is held, this lets us establish the postcondition of `sieve`.

5. Iterating an Index

The high-level specification discussed so far does not allow us to explore the contents of an arbitrary index. To use `search`, we must know which keys we seek. If we do not (and the set of keys is infinite), we cannot write a program that examines all the values stored in the index. To handle this case, we add imperative iterators, based loosely on those in Java. Iterators have three operations:

```
$\{\mathsf{emp} \wedge \mathtt{max} > 1\}$
sieve(max) {
  idx := idxrange(2, max);
  $\{\circledast_{i\in[2..\mathtt{max}]}.\ \mathsf{in}_{\mathsf{rem}}(\mathtt{idx}, i, 0)_1\}$
  parwork(2, max, idx);
  $\{\circledast_{i\in[2..\mathtt{max}]}.\ \mathsf{fac}(i, 2, \lfloor\sqrt{\mathtt{max}}\rfloor) \Rightarrow \mathsf{out}_{\mathsf{rem}}(\mathtt{idx}, i)_1 \wedge$
          $\neg\mathsf{fac}(i, 2, \lfloor\sqrt{\mathtt{max}}\rfloor) \Rightarrow \mathsf{in}_{\mathsf{rem}}(\mathtt{idx}, i, 0)_1\}$
  // By properties of prime numbers and
  // index axioms
  $\{\circledast_{i\in[2..\mathtt{max}]}.\ \mathsf{isPrime}(i) \Rightarrow \mathsf{in}_{\mathsf{def}}(\mathtt{idx}, i, 0)_1$
          $\wedge\ \neg\mathsf{isPrime}(i) \Rightarrow \mathsf{out}_{\mathsf{def}}(\mathtt{idx}, i)_1\}$
  return idx;
}
$\{\mathrm{ret} = \mathtt{idx} \wedge \circledast_{i\in[2..\mathtt{max}]}.\ \mathsf{isPrime}(i) \Rightarrow \mathsf{in}_{\mathsf{def}}(\mathtt{idx}, i, 0)_1$
          $\wedge\ \neg\mathsf{isPrime}(i) \Rightarrow \mathsf{out}_{\mathsf{def}}(\mathtt{idx}, i)_1\}$

$\{2 \leq \mathtt{v} \wedge \circledast_{i\in[2..\mathtt{max}]}.\ \mathsf{in}_{\mathsf{rem}}(\mathtt{idx}, i, 0)_t\}$
parwork(v, max, idx) {
  if (v ≤ sqrt(max)) {
    $\{(2 \leq \mathtt{v} \wedge \circledast_{i\in[2..\mathtt{max}]}.\ \mathsf{in}_{\mathsf{rem}}(\mathtt{idx}, i, 0)_{\frac{t}{2}}) *$
     $(2 \leq \mathtt{v} + 1 \wedge \circledast_{i\in[2..\mathtt{max}]}.\ \mathsf{in}_{\mathsf{rem}}(\mathtt{idx}, i, 0)_{\frac{t}{2}})\}$
    worker(v, max, idx) || parwork(v+1, max, idx)
    $\{(\circledast_{i\in[2..\mathtt{max}]}.\ \mathsf{fac}(i, \mathtt{v}, \mathtt{v}) \Rightarrow \mathsf{out}_{\mathsf{rem}}(\mathtt{idx}, i)_{\frac{t}{2}} \wedge$
          $\neg\mathsf{fac}(i, \mathtt{v}, \mathtt{v}) \Rightarrow \mathsf{in}_{\mathsf{rem}}(\mathtt{idx}, i, 0)_{\frac{t}{2}}) *$
     $\circledast_{i\in[2..\mathtt{max}]}.\ \mathsf{fac}(i, \mathtt{v}{+}1, \lfloor\sqrt{\mathtt{max}}\rfloor) \Rightarrow \mathsf{out}_{\mathsf{rem}}(\mathtt{idx}, i)_{\frac{t}{2}}$
          $\wedge\ \neg\mathsf{fac}(i, \mathtt{v}{+}1, \lfloor\sqrt{\mathtt{max}}\rfloor) \Rightarrow \mathsf{in}_{\mathsf{rem}}(\mathtt{idx}, i, 0)_{\frac{t}{2}}\}$
   // Using permission combination axioms
    $\{\circledast_{i\in[2..\mathtt{max}]}.\ \mathsf{fac}(i, \mathtt{v}, \lfloor\sqrt{\mathtt{max}}\rfloor) \Rightarrow \mathsf{out}_{\mathsf{rem}}(\mathtt{idx}, i)_t \wedge$
          $\neg\mathsf{fac}(i, \mathtt{v}, \lfloor\sqrt{\mathtt{max}}\rfloor) \Rightarrow \mathsf{in}_{\mathsf{rem}}(\mathtt{idx}, i, 0)_t\}$
} }
$\{\circledast_{i\in[2..\mathtt{max}]}.\ \mathsf{fac}(i, \mathtt{v}, \lfloor\sqrt{\mathtt{max}}\rfloor) \Rightarrow \mathsf{out}_{\mathsf{rem}}(\mathtt{idx}, i)_t \wedge$
          $\neg\mathsf{fac}(i, \mathtt{v}, \lfloor\sqrt{\mathtt{max}}\rfloor) \Rightarrow \mathsf{in}_{\mathsf{rem}}(\mathtt{idx}, i, 0)_t\}$

$\{2 \leq \mathtt{v} \wedge \circledast_{i\in[2..\mathtt{max}]}.\ \mathsf{in}_{\mathsf{rem}}(\mathtt{idx}, i, 0)_t\}$
worker(v, max, idx) {
 c := v + v;
 while (c ≤ max) {
   $\{\circledast_{i\in[2..(\mathtt{c}-1)]}.\ \mathsf{fac}(i, \mathtt{v}, \mathtt{v}) \Rightarrow \mathsf{out}_{\mathsf{rem}}(\mathtt{idx}, i)_t \wedge$
          $\neg\mathsf{fac}(i, \mathtt{v}, \mathtt{v}) \Rightarrow \mathsf{in}_{\mathsf{rem}}(\mathtt{idx}, i, 0)_t$
          $* \circledast_{j\in[\mathtt{c}..\mathtt{max}]}.\ \mathsf{in}_{\mathsf{rem}}(\mathtt{idx}, j, 0)_t\}$
   remove(idx, c);
   c := c + v;
}}
$\{\circledast_{i\in[2..\mathtt{max}]}.\ \mathsf{fac}(i, \mathtt{v}, \mathtt{v}) \Rightarrow \mathsf{out}_{\mathsf{rem}}(\mathtt{idx}, i)_t \wedge$
          $\neg\mathsf{fac}(i, \mathtt{v}, \mathtt{v}) \Rightarrow \mathsf{in}_{\mathsf{rem}}(\mathtt{idx}, i, 0)_t\}$
```

Figure 7. Proofs for the `sieve` and `worker` programs.

- `it` := `createIter(h)` creates a new iterator for index `h`.
- (`k`, `v`) := `next(it)` returns some key-value pair in the index for which `it` is an iterator. The returned pair will be one that has not been returned by a previous call to `next` on `it`. When all key-value pairs have been returned, the call returns (nil, nil).
- `destroyIter(it)` frees the iterator `it`.

To iterate an index, one creates a new iterator, calls `next` until it returns (nil, nil), then frees the iterator. Notice that the `next` procedure just returns *some* key-value pair, placing no order on the iteration. This keeps the iterator specification general, as many underlying implementations have no natural ordering.

As in Java, we do not allow full mutability of an index being iterated. We allow partial mutability: keys can be safely modified once they have been returned by `next`.

Iterator specification. An iterator is represented by the abstract predicate $\mathsf{iter}(it, h, S, K, i)$, which describes an iterator it, iterating over index h. The set S contains the key-value pairs that are in the index and have not yet been returned by `next`, while K is the set of keys that are not assigned in the index. The iterator has definite permission i for every key in $\mathsf{keys}(S) \cup K$.

Our specification for the three iterator operations is shown in Figure 8. Creating an iterator for an index requires definite information about the state of each key in that index, in the form of $\mathsf{in}_{\mathsf{def}}$ and $\mathsf{out}_{\mathsf{def}}$ predicates for all keys. It is not sensible for two threads to share the same iterator, as each thread will iterate over an unknown subset of the underlying index. As such, the iter predicate cannot be split for sharing between threads. However, notice that we can create multiple iterators for a single index, as `createIter` requires only fractional permission for each key.

The two specifications for `next` handle the case where the client has not yet seen all key-value pairs in the iterator (in which case, a pair is returned non-deterministically), and when it has (in which case, nil is returned for both the key and value). Destroying an iterator liberates all of the index predicates that have not been returned by `next`, including the $\mathsf{out}_{\mathsf{def}}$ predicates.

5.1 Example: a more powerful map.

In §3.1, we verified `rangeMap`, an algorithm that mapped all values in an index from a given key range through a function, replacing the values with the result. Using an iterator, we can define a concurrent map that does not require a key range, and works over all entries in an index. To avoid having to reason about function pointers, we assume the particular function f is baked into the algorithm source.

$$\left\{\circledast_{(k,v)\in S}\,\mathsf{in}_{\mathsf{def}}(\mathtt{h},k,v)_i * \circledast_{k\notin\mathsf{keys}(S)}\,\mathsf{out}_{\mathsf{def}}(\mathtt{h},k)_i\right\}\ \mathtt{it} := \mathtt{createIter(h)}\ \left\{\mathsf{iter}(\mathtt{it},\mathtt{h},S,\overline{\mathsf{keys}(S)},i)\right\}$$

$$\left\{\mathsf{iter}(\mathtt{it},h,S,K,i)\wedge S\neq\emptyset\right\}\ (\mathtt{k},\mathtt{v}) := \mathtt{next(it)}\ \left\{\begin{array}{r}(\mathtt{k},\mathtt{v})\in S\wedge\mathsf{iter}(\mathtt{it},h,S\setminus\{(\mathtt{k},\mathtt{v})\},K,i) * \\ \mathsf{in}_{\mathsf{def}}(h,\mathtt{k},\mathtt{v})_i\end{array}\right\}$$

$$\left\{\mathsf{iter}(\mathtt{it},h,\emptyset,K,i)\right\}\ (\mathtt{k},\mathtt{v}) := \mathtt{next(it)}\ \left\{\mathsf{iter}(\mathtt{it},h,\emptyset,K,i)\wedge\mathtt{k}=\mathsf{nil}\wedge\mathtt{v}=\mathsf{nil}\right\}$$

$$\left\{\mathsf{iter}(\mathtt{it},h,S,K,i)\right\}\ \mathtt{destroyIter(it)}\ \left\{\circledast_{(k,v)\in S}\,\mathsf{in}_{\mathsf{def}}(h,k,v)_i * \circledast_{k\in K}\,\mathsf{out}_{\mathsf{def}}(h,k)_i\right\}$$

Figure 8. Specification for iterators. For `createIter`, set S denotes the key-value pairs of h, $\mathsf{keys}(S)$ denotes the assigned keys of h, and $\overline{\mathsf{keys}(S)}$ denotes the unassigned keys.

```
{$\circledast_{(k,v)\in S}\,\mathsf{in}_{\mathsf{def}}(\mathtt{h},k,v)_1 \;*\; \circledast_{k\notin\mathsf{keys}(S)}\,\mathsf{out}_{\mathsf{def}}(\mathtt{h},k)_1$}
 map_f(h) {
   it := createIter(h);
  {$\mathsf{iter}(\mathtt{it},\mathtt{h},S,\overline{\mathsf{keys}(S)},1)$}
   map_worker(it, h);
  {$\mathsf{iter}(\mathtt{it},\mathtt{h},\emptyset,\overline{\mathsf{keys}(S)},1) * \circledast_{(k,v)\in S}\,\mathsf{in}_{\mathsf{def}}(\mathtt{h},k,f(v))_1$}
   destroyIter(it);
 }
{$\circledast_{(k,v)\in S}\,\mathsf{in}_{\mathsf{def}}(\mathtt{h},k,f(v))_1 \;*\; \circledast_{k\notin\mathsf{keys}(S)}\,\mathsf{out}_{\mathsf{def}}(\mathtt{h},k)_1$}

{$\mathsf{iter}(\mathtt{it},\mathtt{h},S,K,1)$}
 map_worker(it, h) {
   (k, v) := next(it);
  {$(\mathtt{k},\mathtt{v})\in S\wedge\mathsf{iter}(\mathtt{it},\mathtt{h},S\setminus\{(\mathtt{k},\mathtt{v})\},K,1) * \mathsf{in}_{\mathsf{def}}(\mathtt{h},\mathtt{k},\mathtt{v})_1$
                     $\vee\,(\mathsf{iter}(\mathtt{it},\mathtt{h},\emptyset,K,1)\wedge\mathtt{k}=\mathsf{nil}\wedge\mathtt{v}=\mathsf{nil})$}
   if (k $\neq$ nil) {
    {$(\mathtt{k},\mathtt{v})\in S\wedge\mathsf{iter}(\mathtt{it},\mathtt{h},S\setminus\{(\mathtt{k},\mathtt{v})\},K,1) * \mathsf{in}_{\mathsf{def}}(\mathtt{h},\mathtt{k},\mathtt{v})_1$}
     (
      {$(\mathtt{k},\mathtt{v})\in S\wedge\mathsf{in}_{\mathsf{def}}(\mathtt{h},\mathtt{k},\mathtt{v})_1$}
       remove(h, k); insert(h, k, f(v));
      {$(\mathtt{k},\mathtt{v})\in S\wedge\mathsf{in}_{\mathsf{def}}(\mathtt{h},\mathtt{k},f(\mathtt{v}))_1$}
     ) ||
    {$\mathsf{iter}(\mathtt{it},\mathtt{h},S\setminus(\mathtt{k},\mathtt{v}),K,1)$}
     map_worker(it, h);
    {$\mathsf{iter}(\mathtt{it},\mathtt{h},\emptyset,K,1) * \circledast_{(k',v')\in S\setminus(\mathtt{k},\mathtt{v})}\,\mathsf{in}_{\mathsf{def}}(\mathtt{h},k',f(v'))_1$}
 } }
{$\mathsf{iter}(\mathtt{it},\mathtt{h},\emptyset,K,1) * \circledast_{(k,v)\in S}\,\mathsf{in}_{\mathsf{def}}(\mathtt{h},k,f(v))_1$}
```

Figure 9. Proof outline for `map_f`.

```
map_f(h) {
 it := createIter(h);
 map_worker(it, h);
 destroyIter(it);
}
```

```
map_worker(it, h) {
 (k,v) := next(it);
 if (k ≠ nil) {
    ( remove(h, k);
      insert(h, k, f(v));)
   || map_worker(it, h);
}}
```

A proof of correctness for `map_f` is given in Figure 9.

5.2 Example: counting distinct values.

We can use an index to store discovered information, and then use iteration to summarise what has been discovered. To illustrate this, we give an algorithm which counts the number of distinct values stored in a tree. Both the tree and the secondary store is used for recording distinct values are implemented using our index specification.

Our algorithm is defined as follows:

```
count(it,k,is) {
  fetch(it,k,is);
  itr := createIter(is);
  num := 0;
  (k,v) := next(itr);
  while(k ≠ nil) {
    num := num + 1;
    remove(is,k);
    (k,v) := next(itr);
  }
  destroyIter(itr);
  return num;
}
```

```
fetch(it,k,is) {
  if (k ≠ nil) {
    (k1,k2,v) :=
          search(it,k);
    insert(is,v,k);
    (fetch(it,k1,is) ||
     fetch(it,k2,is));
  }
}
```

The function `count` calls `fetch` to construct the index `is` from the values of the tree in the index `it`. Values can appear at more than one tree node, but are only recorded once in the `is` index. `count` then iterates the index `is`, counting the number of distinct values discovered. We define a tree predicate annotated with the set of values stored in the tree:

$$\mathsf{tree}(h,k,vs) \triangleq \begin{array}{l}\exists k_1,k_2,vs_1,vs_2,v.\\ (k=\mathsf{nil}\wedge vs=\emptyset\wedge\mathsf{emp})\vee\\ \left(\begin{array}{r}\mathsf{tree}(h,k_1,vs_1)*\mathsf{tree}(h,k_2,vs_2)\\ *\,\mathsf{in}_{\mathsf{def}}(h,k,\langle k_1,k_2,v\rangle)_1\\ \wedge\, vs=vs_1\cup vs_2\cup\{v\}\end{array}\right)\end{array}$$

`fetch` and `count` satisfy the following specifications:

$$\begin{array}{l}\left\{\mathsf{tree}(\mathtt{it},\mathtt{k},vs) * \circledast_{k'}.\,\mathsf{out}_{\mathsf{ins}}(\mathtt{is},k',\mathsf{Keys})_i\right\}\\ \quad\mathtt{fetch}(\mathtt{it},\mathtt{k},\mathtt{is})\\ \left\{\begin{array}{r}\mathsf{tree}(\mathtt{it},\mathtt{k},vs) * \circledast_{k'\notin vs}.\,\mathsf{out}_{\mathsf{ins}}(\mathtt{is},k',\mathsf{Keys})_i\\ *\,\circledast_{k'\in vs}.\,\mathsf{in}_{\mathsf{ins}}(\mathtt{is},k',\mathsf{Keys})_i\end{array}\right\}\end{array}$$

$$\begin{array}{l}\left\{\mathsf{tree}(\mathtt{it},\mathtt{k},vs) * \circledast_{k'}.\,\mathsf{out}_{\mathsf{def}}(\mathtt{is},k')_1\right\}\\ \quad\mathtt{count}(\mathtt{it},\mathtt{k},\mathtt{is})\\ \left\{\mathsf{tree}(\mathtt{it},\mathtt{k},vs) * \circledast_{k'}.\,\mathsf{out}_{\mathsf{def}}(\mathtt{is},k')_1\wedge\mathrm{ret}=|vs|\right\}\end{array}$$

Figure 10 shows an outline proof of these specifications. The part of the proof associated with searching the tree is similar in structure to O'Hearn *et al*'s proof of tree disposal

```
$\{\mathsf{tree}(\mathtt{it},\mathtt{k},vs) * \circledast_{k'}.\,\mathsf{out}_{\mathsf{ins}}(\mathtt{is},k',\mathsf{ItKeys})_i\}$
 fetch(it,k,is) {
  if (k ≠ nil) {
    $\left\{\begin{array}{l}\exists k_1,k_2,vs_1,vs_2,v.\ \mathsf{tree}(\mathtt{it},k_1,vs_1) * \mathsf{tree}(\mathtt{it},k_2,vs_2) \\ * \mathsf{in}_{\mathsf{def}}(\mathtt{it},\mathtt{k},\langle k_1,k_2,v\rangle)_1 * \circledast_{k'}.\,\mathsf{out}_{\mathsf{ins}}(\mathtt{is},k',\mathsf{ItKeys})_i \\ \hfill \wedge\, vs = vs_1 \cup vs_2 \cup \{v\}\end{array}\right\}$
    (k1,k2,v) := search(it,k);
    $\left\{\begin{array}{l}\exists vs_1,vs_2.\ \mathsf{tree}(\mathtt{it},\mathtt{k1},vs_1) * \mathsf{tree}(\mathtt{it},\mathtt{k2},vs_2) \\ * \mathsf{in}_{\mathsf{def}}(\mathtt{it},\mathtt{k},\langle \mathtt{k1},\mathtt{k2},\mathtt{v}\rangle)_1 * \circledast_{k'}.\,\mathsf{out}_{\mathsf{ins}}(\mathtt{is},k',\mathsf{ItKeys})_i \\ \hfill \wedge\, vs = vs_1 \cup vs_2 \cup \{\mathtt{v}\}\end{array}\right\}$
    insert(is,v,k);
    $\left\{\begin{array}{l}\exists vs_1,vs_2.\ \mathsf{tree}(\mathtt{it},\mathtt{k1},vs_1) * \mathsf{tree}(\mathtt{it},\mathtt{k2},vs_2) \\ * \mathsf{in}_{\mathsf{def}}(\mathtt{it},\mathtt{k},\langle \mathtt{k1},\mathtt{k2},\mathtt{v}\rangle)_1 * \mathsf{in}_{\mathsf{ins}}(\mathtt{is},\mathtt{v},\mathsf{ItKeys})_{\frac{i}{2}} \\ * \circledast_{k'\neq\mathtt{v}}.\,\mathsf{out}_{\mathsf{ins}}(\mathtt{is},k',\mathsf{ItKeys})_{\frac{i}{2}} \\ * \circledast_{k'}.\,\mathsf{out}_{\mathsf{ins}}(\mathtt{is},k',\mathsf{ItKeys})_{\frac{i}{2}} \wedge vs = vs_1 \cup vs_2 \cup \{\mathtt{v}\}\end{array}\right\}$
    ( fetch(it,k1,is) || fetch(it,k2,is) );
  }
 }
$\left\{\begin{array}{l}\mathsf{tree}(\mathtt{it},\mathtt{k},vs) * \circledast_{k'\notin vs}.\,\mathsf{out}_{\mathsf{ins}}(\mathtt{is},k',\mathsf{ItKeys})_i \\ \hfill * \circledast_{k'\in vs}.\,\mathsf{in}_{\mathsf{ins}}(\mathtt{is},k',\mathsf{ItKeys})_i\end{array}\right\}$

$\{\mathsf{tree}(\mathtt{it},\mathtt{k},vs) * \circledast_{k'}.\,\mathsf{out}_{\mathsf{def}}(\mathtt{is},k')_1\}$
 count(it,k,is) {
  fetch(it,k,is);
  $\left\{\begin{array}{l}\mathsf{tree}(\mathtt{it},\mathtt{k},vs) * \circledast_{k'\notin vs}.\,\mathsf{out}_{\mathsf{def}}(\mathtt{is},k')_1 * \\ \circledast_{k'\in vs}.\,\exists v' \in \mathsf{ItKeys}.\ (k',v') \in S \wedge \mathsf{in}_{\mathsf{def}}(\mathtt{is},k',v')_1 \\ \wedge\, vs = \mathsf{keys}(S)\end{array}\right\}$
  itr := createIter(is);
  num := 0;
 $\{\mathsf{tree}(\mathtt{it},\mathtt{k},vs) * \mathsf{iter}(\mathtt{itr},\mathtt{is},S,\overline{vs},1) \wedge vs = \mathsf{keys}(S) \wedge \mathtt{num} = 0\}$
  (k,v) := next(itr);
  while(k ≠ nil) {
    num := num + 1;
    remove(is,k);
    $\left\{\begin{array}{l}\exists vs',S'.\ \mathsf{tree}(\mathtt{it},\mathtt{k},vs) * \mathsf{iter}(\mathtt{itr},\mathtt{is},S',\overline{vs},1) * \\ \circledast_{k'\in vs\setminus vs'}.\,\mathsf{out}_{\mathsf{def}}(\mathtt{is},k')_1 \wedge |vs'| + \mathtt{num} = |vs| \\ \wedge\, vs' = \mathsf{keys}(S')\end{array}\right\}$
    (k,v) := next(itr);
  }
  $\left\{\begin{array}{l}\mathsf{tree}(\mathtt{it},\mathtt{k},vs) * \mathsf{iter}(\mathtt{itr},\mathtt{is},\emptyset,\overline{vs},1) * \\ \circledast_{k'\in vs}.\,\mathsf{out}_{\mathsf{def}}(\mathtt{is},k')_1 \wedge \mathtt{num} = |vs|\end{array}\right\}$
  destroyIter(itr);
  return num;
 }
$\{\mathsf{tree}(\mathtt{it},\mathtt{k},vs) * \circledast_{k'}.\,\mathsf{out}_{\mathsf{def}}(\mathtt{is},k')_1 \wedge \mathrm{ret} = |vs|\}$
```

Figure 10. Outline proofs of `count` and `fetch`.

using concurrent separation logic [16]. The difference is that we are able to reason abstractly about concurrently inserting into the `is` index.

6. Verifying Index Implementations

In this section, we verify three quite different concurrent index implementations against our abstract specification. Note that proving implementations is an obligation on the writer of the module – clients can reason using our specification without any knowledge of such proofs. We first introduce a simple list-based implementation and show that it satisfies the disjoint specification of §3. This example is given to develop our technical approach. We then prove that a hash table implementation satisfies the sharing specification of §4. Finally, we show that our approach scales to quite complex implementations, by outlining our proof that the $\mathrm{B}^{\mathit{Link}}$ tree algorithm satisfies the sharing specification.

Approach: Concurrent Abstract Predicates. We use the techniques developed in the work on concurrent abstract predicates (CAP) [7] to prove that index implementations satisfy our specification. This approach extends separation logic with both explicit reasoning about sharing within modules, and a powerful abstraction mechanism that can hide sharing from clients.

Sharing between threads is represented in CAP by shared regions, denoted by boxed assertions of the form $\boxed{P}^r_I$. The assertion P describes the contents of the region, r is the name of the region, and I is an interface environment specifying type of mutations threads can perform on P. Assertions on shared regions behave additively under $*$, that is,

$$\boxed{P}^r_I * \boxed{Q}^r_I \;\triangleq\; \boxed{P \wedge Q}^r_I$$

A shared region can be mutated by the environment threads. This means that assertions about shared regions must be *stable*: that is, invariant under other threads' interference.

Often, different threads can perform different operations over a shared resource: for example, they may be able to mutate different keys in a shared index. To represent this behaviour, CAP introduces *capabilities*. These are resources giving a thread the ability to perform particular operations. Threads can hold both non-exclusive and exclusive capabilities. When an exclusive capability is held, no other thread can perform the associated operation.

Shared regions and capabilities can be abstracted using predicates in the manner described in §2. Each predicate represents both some information about a shared region, and some ability held by the thread to modify the shared region. If the combination of capabilities held ensures that the shared assertion is invariant, then stability need not be considered by clients, and the predicate can be treated abstractly.

In the discussion below, we assume the proof system and semantics given in [7], and only give details necessary for understanding the proof structure. The interested reader is referred to [7] for other technical details, including a proof of soundness for the CAP logic.

6.1 Linked List Implementation

To illustrate our approach, we consider a very simple index implementation which uses a linked list with a single lock protecting the entire list[3]. The code for this implementation

[3] This example is quite similar to the coarse-grained set example from [7].

```
search(h, k) {
  lock(h.lk);
  e := h.nxt;
  while (e ≠ nil) {
    if (e.key = k) {
      unlock(h.lk);
      return e.val;
    }
    e := e.nxt;
  }
  unlock(h.lk);
  return nil;
}

insert(h, k, v) {
  lock(h.lk);
  e := h.nxt;
  while (e ≠ nil) {
    if (e.key = k) {
      unlock(h.lk);
      return;
    }
    e := e.nxt;
  }
  e := makeNode(k,v,h.nxt);
  h.nxt := e;
  unlock(h.lk);
}
```

```
remove(h, k) {
  lock(h.lk);
  e := h.nxt;
  prev := h;
  while (e ≠ nil) {
    if(e.key = k) {
      prev.nxt := e.nxt;
      disposeNode(e);
      unlock(h.lk);
      return;
    }
    prev := e;
    e := e.nxt;
  }
  unlock(h.lk);
}
```

Figure 11. Linked list operations.

is given in Figure 11. In order to simplify the presentation, we only consider the disjoint specification of §3 in this section. Additional technicalities are required to handle the full sharing specification of §4. We give these technicalities in §6.3, when we verify the B*Link* tree implementation against the sharing specification.

Before performing any operation on the list, the thread first acquires the lock. The search operation traverses the list checking if an element matches the key; if so, it returns the corresponding value. The insert operation is similar to search. However, if it cannot find the key, it creates a new node and adds it to the head of the list. The remove operation searches for the key to be removed. If it finds the key, it updates the previous node in the list to point to the following node. The node, having been thus removed from the list, is then deleted.

Interpretation of abstract predicates. In order to prove that the operations of the implementation are correct with respect to our specification, we first give concrete interpretations to the abstract predicates.

We begin by defining a predicate $\mathsf{ls}(a, H)$, corresponding to list with address a and representing the index state $H : \mathsf{Keys} \rightharpoonup \mathsf{Vals}$. This is defined in terms of the inductive predicate $\mathsf{lseg}(a, b, H)$, which represents a list segment with address a and final pointer b, having key-value elements given by H. A list segment is either empty, in which case $a = b$ and $H = \emptyset$, or it consists of a node at address a whose key and value are taken from H, and whose nxt field points to a list segment of the rest of the keys and values. The definition of lseg is, in turn, defined in terms of the predicate $\mathsf{node}(a, k, v, n)$, which simply represents a node at address a whose key, val and nxt fields are k, v, and n respectively. The formal definitions of these predicates are as follows:

$$\begin{aligned}
\mathsf{node}(a,k,v,n) &\triangleq a.\mathsf{key} \mapsto k * a.\mathtt{val} \mapsto v * a.\mathtt{nxt} \mapsto n \\
\mathsf{lseg}(a,b,H) &\triangleq (a = b \land H = \emptyset) \lor \\
&\quad\ \exists k,v,n,H'.\, H = H' \uplus \{k \mapsto v\} \land \\
&\quad\ \ \mathsf{node}(a,k,v,n) * \mathsf{lseg}(n,b,H') \\
\mathsf{ls}(a,H) &\triangleq \mathsf{lseg}(a,\mathsf{nil},H)
\end{aligned}$$

Using the ls predicate, we can give a concrete interpretation to our index predicates for the linked list implementation of an index, as follows:

$$\begin{aligned}
\mathsf{in}(h,k,v) &\triangleq \exists r,l,H.\, H(k) = v \land [\textsc{Lock}(k)]^r_1 * \\
&\qquad \boxed{\mathsf{lock}(h.\mathtt{lk}, r, k) * h.\mathtt{nxt} \mapsto l * \mathsf{ls}(l, H)}^{r}_{I(r,h)} \\
\mathsf{out}(h,k) &\triangleq \exists r,l,H.\, k \notin \mathrm{dom}(H) \land [\textsc{Lock}(k)]^r_1 * \\
&\qquad \boxed{\mathsf{lock}(h.\mathtt{lk}, r, k) * h.\mathtt{nxt} \mapsto l * \mathsf{ls}(l, H)}^{r}_{I(r,h)}
\end{aligned}$$

Here, the boxed assertion describes the region r shared between all the threads that can access the list. This boxed assertion says that region r contains a lock at h.lk (we define the predicate lock below) and a pointer h.nxt to a list representing the contents of the index. The index state H is existentially quantified; the assertions only specify whether the key k is in the index, and its value, if any.

Both predicates also include the (unshared) capability resource $[\textsc{Lock}(k)]^r_1$. A thread with such a capability in its local state is able to update the contents of the corresponding region r by performing the $\textsc{Lock}(k)$ action that is defined in the interference environment $I(r, h)$ associated with the region. We will give the formal definition of $I(r, h)$ presently; intuitively, the $\textsc{Lock}(k)$ action allows a thread to acquire the lock in order to subsequently add or remove the key k. The subscript 1 in the capability denotes that it is an *exclusive* capability: no other thread can perform the action. The exclusivity of the permission ensures that the predicates are stable, since the state of key k in the index cannot be changed by any other thread.

We define the predicate $\mathsf{lock}(x, r, k)$ as follows:

$$\begin{aligned}
\mathsf{unlocked}(x,r) &\triangleq x \mapsto 0 * \bigotimes_{i \in \mathsf{Keys}} [\textsc{Mod}(i)]^r_1 \\
\mathsf{locked}(x,r,j) &\triangleq x \mapsto 1 * \bigotimes_{i \in \mathsf{Keys} \setminus \{j\}} [\textsc{Mod}(i)]^r_1 \\
\mathsf{lock}(x,r,k) &\triangleq \mathsf{unlocked}(x,r) \lor \exists j \neq k.\, \mathsf{locked}(x,r,j)
\end{aligned}$$

This lock predicate contains a shared lock bit and a collection of capabilities. Each capability $[\textsc{Mod}(k)]^r_1$ controls the ability to add or remove a particular key k from the shared list in region r. When these capabilities are in the shared region, no thread is able to modify the list; such is the case when the lock is unlocked. When the lock is locked, a single $[\textsc{Mod}(j)]^r_1$ capability is held by some thread, allowing it to perform the necessary update, but only to the key j. The $\mathsf{lock}(h.\mathtt{lk}, r, k)$ predicate ensures, that no other thread can have the $[\textsc{Mod}(k)]^r_1$ capability, and hence update key k.

Describing Interference. The meaning of the capabilities $[\textsc{Lock}(k)]^r_1$ and $[\textsc{Mod}(k)]^r_1$ is determined by the *interference environment* associated with the region r: $I(r, h)$. This defines the possible state mutations that can occur over a given shared region. The environment defines the meaning of capabilities in terms of actions, written $P \rightsquigarrow Q$. When a thread holds a capability mapped to an action $P \rightsquigarrow Q$, it is permitted to replace a part of the region matching P with a part matching Q. To perform the action, a thread may transfer resource between the region and its own local state, and may mutate it in an atomic operation.

For the linked list implementation, the interference environment $I(r, h)$ is defined as follows:

$$\textsc{Mod}(k): \begin{cases} h.\mathtt{nxt} \mapsto l * \mathsf{ls}(l, H) \\ \qquad\qquad \rightsquigarrow \; h.\mathtt{nxt} \mapsto l' * \mathsf{ls}(l', H \uplus \{k \mapsto v\}) \\ h.\mathtt{nxt} \mapsto l * \mathsf{ls}(l, H) \\ \qquad\qquad\qquad \rightsquigarrow \; h.\mathtt{nxt} \mapsto l' * \mathsf{ls}(l', H \setminus \{k\}) \end{cases}$$

$$\textsc{Lock}(k): \begin{cases} h.\mathtt{lk} \mapsto 0 * [\textsc{Mod}(k)]^r_1 & \rightsquigarrow \quad h.\mathtt{lk} \mapsto 1 \\ h.\mathtt{lk} \mapsto 1 \quad \rightsquigarrow & h.\mathtt{lk} \mapsto 0 * [\textsc{Mod}(k)]^r_1 \end{cases}$$

The definition of $\textsc{Mod}(k)$ says that a thread holding a capability $[\textsc{Mod}(k)]^r_1$ is allowed to update the list by adding or removing the key k. The definition of $\textsc{Lock}(k)$ says that the thread is allowed to set or unset the lock bit. Recall that actions replace part of the shared state, so the definition of $\textsc{Lock}(k)$ implies that a thread acquiring the lock also acquires the capability $[\textsc{Mod}(k)]^r_1$, which leaves the shared state. Similarly, when releasing the lock it must give up the capability $[\textsc{Mod}(k)]^r_1$. In this way, acquiring the lock gives a thread the ability to modify the contents of the list.

Verifying the operations. Having given concrete definitions to the index predicates, we can verify that the module's implementations of `add`, `remove` and `search` match our high-level specification. Figure 12 shows one such proof, establishing that the implementation of `insert` matches the following abstract specification:

$$\{\mathsf{out}(\mathtt{h}, \mathtt{k})\} \quad \mathtt{insert}(\mathtt{h}, \mathtt{k}, \mathtt{v}) \quad \{\mathsf{in}(\mathtt{h}, \mathtt{k}, \mathtt{v})\}$$

```
{out(h, k)}
 insert(h, k, v) {
  $\left\{ \exists r, l, H.\, \mathtt{k} \notin \mathrm{dom}(H) \wedge [\textsc{Lock}(\mathtt{k})]^r_1 * \boxed{\mathsf{lock}(\mathtt{h.lk}, r, \mathtt{k}) * \mathtt{h.nxt} \mapsto l * \mathsf{ls}(l, H)}^r_{I(r,h)} \right\}$
  lock(h.lk); // use the capability $[\textsc{Lock}(\mathtt{k})]^r_1$.
  $\left\{ \exists r, l, H.\, \mathtt{k} \notin \mathrm{dom}(H) \wedge [\textsc{Mod}(\mathtt{k})]^r_1 * [\textsc{Lock}(\mathtt{k})]^r_1 * \boxed{\mathsf{locked}(\mathtt{h.lk}, r, \mathtt{k}) * \mathtt{h.nxt} \mapsto l * \mathsf{ls}(l, H)}^r_{I(r,h)} \right\}$
  e := h.nxt;
  while (e ≠ nil) {
   $\left\{ \begin{array}{l} \exists r, l, H, H_1, H_2, k', v', n.\, \mathtt{k} \notin \mathrm{dom}(H) \wedge \\ \quad [\textsc{Mod}(\mathtt{k})]^r_1 * [\textsc{Lock}(\mathtt{k})]^r_1 * \\ \boxed{\begin{array}{l} \mathsf{locked}(\mathtt{h.lk}, r, \mathtt{k}) * \mathtt{h.nxt} \mapsto l * \\ \mathsf{lseg}(l, \mathsf{e}, H_1) * \mathsf{node}(\mathsf{e}, k', v', n) * \mathsf{ls}(n, H_2) \\ \qquad \wedge H_1 \uplus H_2 \uplus \{k' \mapsto v'\} = H \end{array}}^r_{I(r,h)} \end{array} \right\}$
   if (e.key = k) {
    {false} // this branch is for k in the set
    unlock(h.lk);
    return;
   }
   e := e.nxt;
   $\left\{ \begin{array}{l} \exists r, l, H, H_1, H_2.\, \mathtt{k} \notin \mathrm{dom}(H) \wedge \\ \quad [\textsc{Mod}(\mathtt{k})]^r_1 * [\textsc{Lock}(\mathtt{k})]^r_1 * \\ \boxed{\begin{array}{l} \mathsf{locked}(\mathtt{h.lk}, r, \mathtt{k}) * \mathtt{h.nxt} \mapsto l * \\ \mathsf{lseg}(l, \mathsf{e}, H_1) * \mathsf{ls}(\mathsf{e}, H_2) \wedge H_1 \uplus H_2 = H \end{array}}^r_{I(r,h)} \end{array} \right\}$
  }
  $\left\{ \exists r, l, H.\, \mathtt{k} \notin \mathrm{dom}(H) \wedge [\textsc{Mod}(\mathtt{k})]^r_1 * [\textsc{Lock}(\mathtt{k})]^r_1 * \boxed{\mathsf{locked}(\mathtt{h.lk}, r, \mathtt{k}) * \mathtt{h.nxt} \mapsto l * \mathsf{ls}(l, H)}^r_{I(r,h)} \right\}$
  e := makeNode(k,v,h.nxt);
  $\left\{ \begin{array}{l} \exists r, l, H.\, \mathtt{k} \notin \mathrm{dom}(H) \wedge \mathsf{node}(\mathsf{e}, \mathtt{k}, \mathtt{v}, l) * \\ \quad [\textsc{Mod}(\mathtt{k})]^r_1 * [\textsc{Lock}(\mathtt{k})]^r_1 * \\ \boxed{\mathsf{locked}(\mathtt{h.lk}, r, \mathtt{k}) * \mathtt{h.nxt} \mapsto l * \mathsf{ls}(l, H)}^r_{I(r,h)} \end{array} \right\}$
  h.nxt := e; // use the capability $[\textsc{Mod}(\mathtt{k})]^r_1$.
  $\left\{ \begin{array}{l} \exists r, l, H.\, \mathtt{k} \notin \mathrm{dom}(H) \wedge [\textsc{Mod}(\mathtt{k})]^r_1 * [\textsc{Lock}(\mathtt{k})]^r_1 * \\ \qquad \boxed{\begin{array}{l} \mathsf{locked}(\mathtt{h.lk}, r, \mathtt{k}) * \mathtt{h.nxt} \mapsto \mathsf{e} * \\ \qquad \mathsf{node}(\mathsf{e}, \mathtt{k}, \mathtt{v}, l) * \mathsf{ls}(l, H) \end{array}}^r_{I(r,h)} \end{array} \right\}$
  unlock(h.lk); // use the capability $[\textsc{Lock}(\mathtt{k})]^r_1$.
  $\left\{ \exists r, l, H.\, H(\mathtt{k}) = \mathtt{v} \wedge [\textsc{Lock}(\mathtt{k})]^r_1 * \boxed{\mathsf{lock}(\mathtt{h.lk}, r, \mathtt{k}) * \mathtt{h.nxt} \mapsto l * \mathsf{ls}(l, H)}^r_{I(r,h)} \right\}$
 }
{in(h, k, v)}
```

Figure 12. Proof outline for linked list `insert`.

In the proof, mutations of the shared state require that the thread holds a capability permitting the mutation. These points in `insert` are annotated by program comments. For example, towards the end of `insert`, the assignment `h.nxt:=e` assigns to the shared location `h.nxt`. This mutation corresponds to performing the first of the actions associated with the $[\textsc{Mod}(\mathtt{k})]^r_1$ capability, held in the local state. The action requires that initially `h.nxt` should point to a list representing some index state H, and that after the assignment it should point to a list representing the state $H \uplus \{\mathtt{k} \mapsto v\}$ for some v. By considering the predicate definitions, this is clearly the case.

It is necessary to check that the all assertions in the proof are stable. In fact, once the lock has been acquired, the only actions which can affect the shared state are $\textsc{Mod}(\mathtt{k})$ and $\textsc{Lock}(\mathtt{k})$. Since full permission to both is held in local state, no interference can happen, and so the assertions are stable.

For the pre- and postconditions, the list may be locked or unlocked, but it can only be modified with respect to keys other than k. Since no information about such keys is contained in these assertions, they are also stable.

Verifying the axioms. As well as proving the specifications for the operations, our other obligation is establishing that implementation satisfies the axioms of the abstract specification. To do this, we use the concrete definitions for the abstract predicates. For example, we prove the following axiom from the disjoint specification:

$$\mathsf{in}(h, k, v) * \mathsf{out}(h, k) \implies \mathsf{false}$$

If we expand the predicate definitions on the left-hand side of this implication, we end up with the following assertion:

$$\begin{array}{l} \exists r, l, H.\, H(k) = v \wedge [\textsc{Lock}(k)]^r_1 * \\ \quad \boxed{\mathsf{lock}(h.\mathtt{lk}, r, k) * h.\mathtt{nxt} \mapsto l * \mathsf{ls}(l, H)}^r_{I(r,h)} * \\ \exists r, l, H.\, k \notin \mathrm{dom}(H) \wedge [\textsc{Lock}(k)]^r_1 * \\ \quad \boxed{\mathsf{lock}(h.\mathtt{lk}, r, k) * h.\mathtt{nxt} \mapsto l * \mathsf{ls}(l, H)}^r_{I(r,h)} \end{array}$$

The memory location $h.\mathtt{nxt}$ cannot belong to more than one region at once, so we can infer that both existentially-quantified rs must refer to the same shared region. The capability $[\textsc{Lock}(k)]^r_1$ is exclusive, denoted by the 1 subscript. Now

$$[\textsc{Lock}(k)]^r_1 * [\textsc{Lock}(k)]^r_1 \implies \mathsf{false}$$

and so the axiom holds.

6.2 Hash Table Implementation

We now consider a second index implementation which uses a hash table. The hash table algorithm consists of a fixed-size array and a hashing function mapping from keys to offsets in the array. Each element of the array is a pointer to a secondary index storing the key-value pairs that hash to the associated array offset.

Secondary indexes are often implemented as linked lists, but in fact any kind of index implementation can be used. In this section, we assume that secondary indexes are implemented by *some* module matching our abstract specification, but do not specify which. (To avoid naming conflicts, we rename the methods and predicates of the secondary index to search′, insert′, remove′, $\mathsf{in}'_{\mathsf{def}}$, $\mathsf{in}'_{\mathsf{rem}}$, *etc.*.) We then show that the resulting hash table module also matches our abstract specification. That is, we show that we can build a concurrent index using a (different) index module.

The hash table implementations of search, insert and remove are given in Figure 13. This code assumes a pure hashing function hash which takes a key k and returns an integer $\mathsf{hash}(\mathtt{k})$ between 0 and $max - 1$, where *max* is the size of the hash table array.

```
search(h, k) {
  w := hash(k);                    remove(h, k) {
  a := [h+w];                        w := hash(k);
  return (search'(a, k));            a := [h+w];
}                                    remove'(a, k);
                                   }
insert(h, k, v) {
  w := hash(k);
  a := [h+w];
  insert'(a, k, v);
}
```

Figure 13. Hash table operations.

Although the implementation we consider here is very simple, it captures the essence of more complicated implementation's such as Java's ConcurrentHashMap, which uses resizable hash tables as a secondary index. It would be invaluable to consider such real-world implementations in detail, but this is beyond the scope of the present work.

Interpretation of abstract predicates. All of our index predicates – $\mathsf{in}_{\mathsf{ins}}$, $\mathsf{out}_{\mathsf{ins}}$, $\mathsf{in}_{\mathsf{rem}}$, and so on – consist of a shared region containing a hash table pointer, and a local predicate representing the associated secondary index. Picking an arbitrary example, we define $\mathsf{in}_{\mathsf{rem}}(h, k, v)_i$ as follows:

$$\mathsf{in}_{\mathsf{rem}}(h, k, v)_i \;\triangleq\; \exists r, h'.\, \boxed{h + \mathsf{hash}(k) \mapsto h' * \mathsf{true}}^r * \mathsf{in}'_{\mathsf{rem}}(h', k, v)_i$$

(The predicates have exactly the same form. Only the predicate pertaining to the secondary index changes.)

The shared region contains a pointer from $h + \mathsf{hash}(k)$ to the address of the secondary index, h'. The rest of the hash table array also belongs to the shared region; it is represented in the assertion by true. The array of pointers representing the hash table is read only, so the interference environment for the shared region is empty.

The secondary index is represented by the predicate $\mathsf{in}'_{\mathsf{rem}}(h', k, v)_i$. Note that this definition hides completely the implementation of the secondary index. The hash table simply knows that this element of the index can be queried according to the abstract index specification. State mutations on the secondary index are already captured by the predicate representing it, meaning that they need not be considered when verifying the hash table implementation.

Verifying the operations. A sketch-proof for the hash table implementation of search is given in Figure 14. Notice that this proof appeals to the specification of search′ when retrieving a value from the appropriate secondary index. Since there are no actions defined for the shared region, stability of our assertions is trivial.

Verifying the axioms. The axioms follow from the axioms of the secondary index. In particular, two predicates involving the same key will be defined in terms of predicates which must be on the same secondary index.

```
$\{\mathsf{in}_{\mathsf{def}}(\mathtt{h}, \mathtt{k}, v)_i\}$
 search(h, k) {
   $\left\{\exists r, h'. \boxed{(\mathtt{h} + \mathsf{hash}(\mathtt{k})) \mapsto h' * \mathsf{true}}^{r} * \mathsf{in}'_{\mathsf{def}}(h', \mathtt{k}, v)_i\right\}$
    w := hash(k);
    a := [h+w];
   $\left\{\exists r. \boxed{(\mathtt{h} + \mathsf{hash}(\mathtt{k})) \mapsto \mathsf{a} * \mathsf{true}}^{r} * \mathsf{in}'_{\mathsf{def}}(\mathsf{a}, \mathtt{k}, v)_i\right\}$
    return (search'(a, k)); // search' specification
   $\left\{\exists r, h'. \boxed{\mathtt{h} + \mathsf{hash}(\mathtt{k}) \mapsto h' * \mathsf{true}}^{r} * \mathsf{in}'_{\mathsf{def}}(h', \mathtt{k}, v)_i \wedge \mathrm{ret} = v\right\}$
 }
$\{\mathsf{in}_{\mathsf{def}}(\mathtt{h}, \mathtt{k}, v)_i \wedge \mathrm{ret} = v\}$
```

Figure 14. Proof outline for hash table search.

6.3 B^{Link} Tree Implementation

Our final index implementation is Sagiv's B^{Link} tree algorithm [20]. (Note that we only consider the algorithm without compression here.) A B^{Link} tree is a balanced search tree. An example is shown in Figure 15. The leaves of the tree contain they key-value pairs stored in the index in order. Non-leaf (or inner) nodes associate keys with pointers to nodes at the next level down, which direct the traversal of the tree. In addition, the final pointer in each node's list, the link pointer, points to the next node at that level (if it exists). The tree is accessed through a prime block which holds pointers to the first node at each level in the tree.

During inserts, nodes of the tree that are at full capacity may be split by creating a new right sibling and transferring half of the keys to the new node. This new node must then be attached to the level above, which might require further splittings. However, other operations may still need to traverse the tree before this operation is completed. A traversal in progress may therefore have to use link pointers to find the correct leaf. Since the minimum values of nodes are always preserved, and every leaf with a minimum value no less than that of a given node is reachable from that node, such traversals are always possible.

Search operations on a B^{Link} tree are lock-free, and insert and remove operations lock only one node (or two if they are modifying the root node) at a time, making this a highly concurrent implementation of an index. This index algorithm is much more complex than the list or hash table, and is therefore considerably more challenging to verify.

The full details of the B^{Link} tree implementation are too lengthy to describe in detail. We only consider search in any depth here; for full details, see the technical report [5].

Node notation. We use the notation $\mathsf{node}(l, k, p, D, k', p')$ to denote the contents of a node, where:

- l determines whether the node is locked ($l = 1$) or not ($l = 0$),
- k is the minimum key for the node (less than or equal to all keys that are reachable from it),
- p is the pointer to the left-most child for an inner node, or nil for a leaf node,
- D is a list of pairs of keys and child pointers (for inner nodes), or keys and stored values (for leaf nodes),
- k' is the maximum key for the node,
- p' is the pointer to the next sibling of the node (or nil if it is the last).

We also use $\mathsf{inner}(l, k, p, D, k', p')$ to denote an inner node with the given contents (requiring that $p \neq \mathsf{nil}$) and $\mathsf{leaf}(l, k, D, k', p')$ to denote a leaf node. In this notation, the contents of node 3 in Figure 15 would be represented as

$$\mathsf{inner}(0, -\infty, 1, [(22, 2), (38, 7)], 44, 8).$$

Interpretation of abstract predicates. All of our index predicates are defined as a shared region containing a B^{Link} tree and a collection of shared capabilities, as well as some thread-local capabilities. For example, the predicate $\mathsf{in}_{\mathsf{def}}(h, k, v)$ is defined as follows:

$$\mathsf{in}_{\mathsf{def}}(h, k, v)_i \triangleq \exists r. \boxed{\mathsf{B}_{\in}(h, k, v)}^{r}_{I(r,h)} * \mathsf{dcaps}(k, r, i)$$

The shared assertion $\mathsf{B}_{\in}(h, k, v)$ denotes a B^{Link} tree at address h containing the key-value pair (k, v). We omit the formal definition of this predicate here, which may be found in the technical report [5]. The predicate $\mathsf{dcaps}(k, r, i)$, which is defined in Figure 16, consists of capabilities associated with the current thread.

The permission subscripts of the capabilities are more complex than those we have seen so far: they are deny-guarantee permissions [9]. A guarantee permission, indicated by the subscript (g, i) for $0 < i \leq 1$ (or simply g when we do not care about the exact value of i), allows a thread to perform the associated action. If the permission is less than 1, other threads may have guarantee permissions to perform the same action – it is a non-exclusive permission. A deny permission, indicated by the subscript (d, i) (or, again, simply d), does not allow the thread to perform the associated action, but precludes the possibility that any other thread will either. Fractions of the same type may be combined by addition, and $(g, 1) = 1 = (d, 1)$ represents exclusive permission; however, a deny permission and a guarantee permission cannot be combined, since they are conflicting. (For further details, see Dodds *et al.* [9].)

The intuitive meaning of the capabilities in the dcaps predicate is as follows. The $[\textsc{Lock}]^{r}_{g}$ capability says that the current thread is allowed to lock nodes in the region r. The $[\textsc{Swap}]^{r}_{g}$ capability allows the $\mathsf{in}_{\mathsf{def}}$ predicate to be modified to represent different behaviour (for example, by converting it to $\mathsf{in}_{\mathsf{rem}}$ or unk) provided $i = 1$. The $[\textsc{Rem}(0, k)]^{r}_{(d,i)}$ capability says that neither the current thread, nor any other thread, is allowed to remove the key k from the B^{Link} tree in region r. However, if $i = 1$, then the current thread has the exclusive capability to remove key k from the tree. The

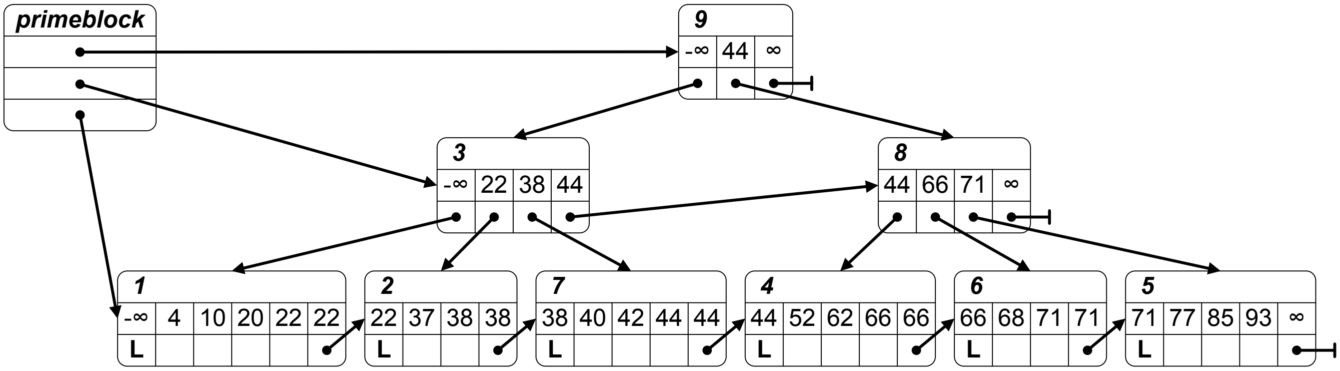

Figure 15. A B^{Link} tree.

$$\mathsf{dcaps}(k, r, i) \triangleq [\textsc{Lock}]^r_g * [\textsc{Swap}]^r_g * [\textsc{Rem}(0, k)]^r_{(d,i)} * \mathop{\circledast}_{v \in \mathsf{Vals}} [\textsc{Ins}(0, k, v)]^r_{(d,i)}$$

$$\begin{aligned}\mathsf{niceNode}(N, k, v, r, h) \triangleq \exists k_0, p_0, D, k', p'.\ &\left(k' = +\infty \vee \boxed{p' \mapsto \mathsf{node}(-, k', -, -, -, -) * \mathsf{true}}^{\,r}_{I(r,h)}\right) \wedge \\ &\left(\left(\begin{matrix} N = \mathsf{inner}(-, k_0, p_0, D, k', p') \wedge \forall (k, p) \in D. \\ \boxed{p \mapsto \mathsf{node}(-, k, -, -, -, -) * \mathsf{true}}^{\,r}_{I(r,h)} \end{matrix}\right) \vee \left(\begin{matrix} N = \mathsf{leaf}(-, k_0, D, k', p') \wedge \\ (k_0 < k \leq k' \Rightarrow (k, v) \in D) \end{matrix}\right)\right)\end{aligned}$$

Figure 16. Predicates used in the B^{Link} tree proofs.

$[\textsc{Ins}(0, k, v')]^r_{(d,i)}$ capabilities similarly restrict the ability to insert value v' at key k.

The other index predicates are defined in a similar way to $\mathsf{in}_{\mathsf{def}}$. For example, the definition of the $\mathsf{in}_{\mathsf{rem}}(h, k, v)_i$ predicate will include a $\textsc{Rem}$ capability for k with permission (g, i), so that any thread may remove the key from the tree, as well as all $\textsc{Ins}$ capabilities for k with permission (d, i), so that no thread may insert values for the key into the tree. The full definitions of the predicates may be found in the technical report [5].

Describing Interference. The interference environment, $I(r, h)$, for the B^{Link} tree implementation is markedly more complex than for the list or hash table. It involves a substantial amount of capability swapping to track changes to the shared state and to thread behaviour. Figure 17 gives a few examples of definitions in the interference environment. These definitions can be read as follows:

- $\textsc{Lock}$ allows a thread to lock a node in the B^{Link} tree. When locking, the thread acquires the exclusive capability $[\textsc{Unlock}(x)]^r_1$, allowing it to unlock the node again.
- $\textsc{Rem}(t, k)$ allows a thread to give up $[\textsc{Rem}(t, k)]^r_{(g,i)}$ and $[\textsc{Unlock}(x)]^r_1$ and acquire the exclusive capability $[\textsc{ModLR}(t, x, k, i)]^r_1$. This means that a thread which is allowed to remove the key k from the tree and holds the lock on a node x can acquire the right to remove the key k from the leaf node x (the value t is used to track capability transfer in some environments).
- $\textsc{ModLR}(t, x, k, i)$ allows a thread to remove a key-value pair $(k, -)$ from a leaf node. In doing so, the thread gives up the capability $[\textsc{ModLR}(t, x, k, i)]^r_1$ and reacquires the capability $[\textsc{Unlock}(x)]^r_1$, and, if $t = 0$, the capability $[\textsc{Rem}(k)]^r_{(g,i)}$. (We write "$-$" to indicate an unspecified, existentially quantified value.)

Full details of the interference environment may be found in the technical report [5].

Note that both $[\textsc{Rem}(0, k)]^r_{(g,i)}$ and $[\textsc{Rem}(1, k)]^r_{(g,i)}$ capabilities allow a thread to remove the key k; however, the latter requires the thread to leave a $[\textsc{Rem}(1, k)]^r_{(g,i)}$ capability behind in the shared state when it does so. This is used to implement the $\mathsf{in}_{\mathsf{rem}}$ predicate: if none of the threads with $\mathsf{in}_{\mathsf{rem}}(k, v)$ predicates remove k then between them they must still be able to produce the full $[\textsc{Rem}(1, k)]^r_1$ capability, proving that none of them did so. Thus the $\mathsf{in}_{\mathsf{rem}}(k, v)_1$ can be converted to $\mathsf{in}_{\mathsf{def}}(k, v)_1$.

Verifying the operations. We give a sketch proof in Figure 18, showing that the B^{Link} tree implementation of `search` matches the following specification:

$$\left\{\mathsf{in}_{\mathsf{def}}(\mathtt{h}, \mathtt{k}, v)_i\right\}\ \mathtt{r} := \mathtt{search}(\mathtt{h}, \mathtt{k})\ \left\{\mathsf{in}_{\mathsf{def}}(\mathtt{h}, \mathtt{k}, v)_i \wedge \mathtt{r} = v\right\}$$

The `search` operation only mutates thread-local state, so the thread does not require capabilities to perform actions. However, by owning deny permissions (d, i) on all the $\textsc{Rem}$ and $\textsc{Ins}$ capabilities for key `k`, the thread can establish that no other thread can modify the value associated with `k`.

$$
\begin{aligned}
\textsc{Lock}: \quad & x \mapsto \mathsf{node}(0, k_0, p, D, k', p') * [\textsc{Unlock}(x)]^r_1 \quad \rightsquigarrow \quad x \mapsto \mathsf{node}(1, k_0, p, D, k', p') \\
\textsc{Rem}(t,k): \quad & [\textsc{ModLR}(t,x,k,i)]^r_1 \quad \rightsquigarrow \quad [\textsc{Rem}(t,k)]^r_{(g,i)} * [\textsc{Unlock}(x)]^r_1 \\
\textsc{ModLR}(t,x,k,i): \quad & \left(\begin{array}{l} x \mapsto \mathsf{leaf}(1, k_0, D, k', p') * [\textsc{Unlock}(x)]^r_1 \\ * \left([\textsc{Rem}(t,k)]^r_{(g,i)} \wedge t = 0 \vee \mathsf{emp} \wedge t = 1 \right) \\ \wedge (k, -) \in D \end{array} \right) \quad \rightsquigarrow \quad \left(\begin{array}{l} x \mapsto \mathsf{leaf}(1, k_0, D', k', p') \\ * [\textsc{ModLR}(t,x,k,i)]^r_1 \\ \wedge D = D' \uplus (k, -) \end{array} \right)
\end{aligned}
$$

Figure 17. Example actions from the B^{Link} tree interference environment.

Thus, the assertion that the key-value pair $(\mathtt{k}, v)$ is contained in the B^{Link} tree is stable.

The proof uses the predicate $\mathsf{niceNode}(N, k, v, r, h)$, defined in Figure 16. The definition of niceNode asserts that the node descriptor N contains legitimate information about the tree. If N is an inner node, then the children and link pointers of N must all point to extant nodes in the tree, which have the minimum values specified by N – this ensures that following a pointer reaches an appropriate node. If N is a leaf node into whose range the key k falls, then the key-value pair (k, v) must be stored in N – this ensures that the search will return the correct value.

Assertions in the proof must be stable – that is, invariant under interference from other threads. The stability of niceNode is ensured by the fact that the capabilities held by the thread do not allow nodes to be removed, the minimum values of nodes to change, or key k to be changed.

A bug in the B^{Link} tree algorithm. While verifying the algorithm, we discovered a subtle bug in the original presentation [20]. The bug can occur during an `insert`, when a thread splits a tree node which itself was the result of another thread splitting the tree root. In order to insert the new node into the tree, the first thread will look in the prime block for the node's parent. However, the second thread might not yet have written a pointer to the new root, resulting in an invalid dereference. Our solution was to require that a thread splitting the current the root locks the new node. A thread trying to insert must wait until the creation of the root is complete. A detailed trace exhibiting this bug can be found in [5].

7. Conclusions

We have proposed a simple, abstract specification for reasoning about concurrent indexes. We have demonstrated the versatility of our specification, verifying a representative range of client applications ranging from common programming patterns such as memoization and map, to algorithms such as a prime number sieve. We have demonstrated that our particular choice of index specification is satisfied by three radically different concurrent implementations, based on simply linked lists, hash tables, and Sagiv's complex and highly concurrent B^{Link} trees respectively.

Relationship to linearizability. Linearizability [12] is the current de-facto correctness criterion for concurrent algo-

```
{$\mathsf{in_{def}}(\mathtt{h}, \mathtt{k}, v)_i$}
search(h, k) {
 {$\boxed{\mathsf{B}_\in(\mathtt{h}, \mathtt{k}, v)}^r_{I(r,\mathtt{h})} * \mathsf{dcaps}(\mathtt{k}, r, i)$}
  PB := getPrimeBlock(h);
  cur := root(PB);
  N := get(cur);
  {$\boxed{\mathsf{B}_\in(\mathtt{h}, \mathtt{k}, v)}^r_{I(r,\mathtt{h})} * \mathsf{dcaps}(\mathtt{k}, r, i)$
   $* \mathsf{niceNode}(\mathtt{N}, \mathtt{k}, v, r, \mathtt{h})$
   $\wedge \mathtt{N} = \mathsf{node}(-, k_0, p, D, k', p') \wedge k_0 = -\infty$}
  while(isLeaf(N) = false) {
    cur := next(N, k);
    N := get(cur);
  }
  {$\boxed{\mathsf{B}_\in(\mathtt{h}, \mathtt{k}, v)}^r_{I(r,\mathtt{h})} * \mathsf{dcaps}(\mathtt{k}, r, i)$
   $* \mathsf{niceNode}(\mathtt{N}, \mathtt{k}, v, r, \mathtt{h})$
   $\wedge \mathtt{N} = \mathsf{leaf}(-, k_0, D, k', p') \wedge k_0 < \mathtt{k}$}
  while(k > highValue(N)) {
    cur := next(N, k);
    N := get(cur);
  }
  {$\boxed{\mathsf{B}_\in(\mathtt{h}, \mathtt{k}, v)}^r_{I(r,\mathtt{h})} * \mathsf{dcaps}(\mathtt{k}, r, i)$
   $* \mathsf{niceNode}(\mathtt{N}, \mathtt{k}, v, r, \mathtt{h})$
   $\wedge \mathtt{N} = \mathsf{leaf}(-, k', D, k'', -) \wedge k' < \mathtt{k} \leq k''$}
  if(isIn(N, k)) {
    {$\boxed{\mathsf{B}_\in(\mathtt{h}, \mathtt{k}, v)}^r_{I(r,\mathtt{h})} * \mathsf{dcaps}(\mathtt{k}, r, i)$
     $* \mathsf{niceNode}(\mathtt{N}, \mathtt{k}, v, r, \mathtt{h})$
     $\wedge \mathtt{N} = \mathsf{leaf}(-, k', D, k'', -) \wedge (\mathtt{k}, v) \in D$}
    return( lookup(N, k) );
  } else {
    {$\mathsf{false}$}
    return nil;
  }
 {$\boxed{\mathsf{B}_\in(\mathtt{h}, \mathtt{k}, v)}^r_{I(r,\mathtt{h})} * \mathsf{dcaps}(\mathtt{k}, r, i) \wedge ret = v$}
}
{$\mathsf{in_{def}}(\mathtt{h}, \mathtt{k}, v)_i \wedge \mathrm{ret} = v$}
```

Figure 18. Proof outline for the B^{Link} tree `search`.

rithms. It requires that the methods of concurrent objects behave as atomic operations, thus providing a proof technique for observational refinement [11]. We could employ linearizability, or other atomicity refinement techniques such as [22], as a proof technique for verifying that implementations meet our abstract specification: an implementation that meets the sequential specification of an index and whose operations behave atomically can easily be shown to meet the concurrent specification. However, this simply shifts the proof burden; our approach is able to verify clients and implementations in a single coherent proof system.

While linearizability assures that index operations behave atomically, our abstract specification makes no such guarantee. Instead, our client proofs enforce abstract constraints on the possible interactions between threads, such as only allowing removals on a certain key. Consequently, while all linearizable indexes can be shown to implement our specification, our specification also admits implementations that are not linearizable. For instance, an index that implemented removal by performing the operation twice in succession could meet our specification, but would not be linearizable. As a more realistic example, consider the program:

$$\begin{array}{r||l} \mathtt{insert}(\mathtt{k},0) & \mathtt{insert}(\mathtt{k},1) \\ \mathtt{x} := \mathtt{search}(\mathtt{k}) & \mathtt{y} := \mathtt{search}(\mathtt{k}) \end{array}$$

Linearizability ensures that, after executing the program, the variables `x` and `y` will be equal (one or the other `insert` must come first, and a second `insert` has no effect). However, our specification does not ensure this. An implementation in which writes are cached, for instance, may satisfy our specification, but fail to provide this stronger guarantee. In practice, the strength of specifications is often traded against performance. We have shown how our approach can provide weak (§3) and strong (§4) specifications of concurrent behaviour. Our approach could therefore be seen as a flexible alternative to linearizability as a correctness criterion for concurrent programs.

Acknowledgments. Special thanks to Moshe Vardi for challenging us to verify Sagiv's concurrent B^{Link} tree algorithm, and to Adam Wright for substantial contributions to the section on iteration (§5), and invaluable discussions and feedback overall. Thanks also to Bornat, Jones, Shapiro, and many researchers at Cambridge, Imperial and Queen Mary working on separation logic, for discussions and feedback. We acknowledge funding from an EPSRC DTA (da Rocha Pinto), EPSRC programme grant EP/H008373/1 (da Rocha Pinto, Dinsdale-Young, Gardner and Wheelhouse) and EPSRC grant EP/H010815/1 (Dodds).

References

[1] BLELLOCH, G. E. Programming parallel algorithms. *Commun. ACM 39* (March 1996), 85–97.

[2] BOYLAND, J. Checking interference with fractional permissions. In *Static Analysis* (2003).

[3] CALCAGNO, C., GARDNER, P., AND ZARFATY, U. Context Logic and tree update. In *POPL* (2005), ACM.

[4] DA ROCHA PINTO, P. Reasoning about Concurrent Indexes. Master's thesis, Imperial College London, Sept. 2010.

[5] DA ROCHA PINTO, P., DINSDALE-YOUNG, T., DODDS, M., GARDNER, P., AND WHEELHOUSE, M. A simple abstraction for complex concurrent indexes. Tech. rep., Imperial College London, 2011.

[6] DILLIG, I., DILLIG, T., AND AIKEN, A. Precise reasoning for programs using containers. *SIGPLAN Not. 46* (2011).

[7] DINSDALE-YOUNG, T., DODDS, M., GARDNER, P., PARKINSON, M., AND VAFEIADIS, V. Concurrent abstract predicates. In *ECOOP* (2010).

[8] DINSDALE-YOUNG, T., GARDNER, P., AND WHEELHOUSE, M. Abstraction and Refinement for Local Reasoning. In *VSTTE* (2010).

[9] DODDS, M., FENG, X., PARKINSON, M., AND VAFEIADIS, V. Deny-guarantee reasoning. In *ESOP* (2009).

[10] FENG, X., FERREIRA, R., AND SHAO, Z. On the relationship between concurrent separation logic and assume-guarantee reasoning. In *ESOP* (2007).

[11] FILIPOVIC, I., O'HEARN, P., RINETZKY, N., AND YANG, H. Abstraction for concurrent objects. In *ESOP* (2010).

[12] HERLIHY, M. P., AND WING, J. M. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst. 12* (July 1990), 463–492.

[13] HOARE, C. A. R. Proof of a structured program: 'The sieve of Eratosthenes'. *The Computer Journal 15*, 4 (1972), 321–325.

[14] KUNCAK, V., LAM, P., ZEE, K., AND RINARD, M. C. Modular pluggable analyses for data structure consistency. *IEEE Trans. Softw. Eng. 32* (December 2006), 988–1005.

[15] MALECHA, G., MORRISETT, G., SHINNAR, A., AND WISNESKY, R. Toward a verified relational database management system. In *POPL* (2010).

[16] O'HEARN, P. W. Resources, concurrency, and local reasoning. *Theor. Comput. Sci. 375* (April 2007), 271–307.

[17] PARKINSON, M., AND BIERMAN, G. Separation logic and abstraction. In *POPL* (2005).

[18] PHILIPPOU, A., AND WALKER, D. A process-calculus analysis of concurrent operations on b-trees. *J. Comput. Syst. Sci. 62*, 1 (2001), 73–122.

[19] REYNOLDS, J. Separation logic: a logic for shared mutable data structures. In *LICS* (2002).

[20] SAGIV, Y. Concurrent operations on B*-trees with overtaking. *Journal of Computer and System Sciences 33* (October 1986).

[21] SEXTON, A., AND THIELECKE, H. Reasoning about B+ trees with operational semantics and separation logic. *ENTCS 218* (2008).

[22] TURON, A. J., AND WAND, M. A separation logic for refining concurrent objects. In *POPL* (2011).

[23] VAFEIADIS, V., AND PARKINSON, M. A marriage of rely/guarantee and separation logic. *CONCUR* (2007).