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 – also known as *associative arrays*, *dictionaries*, *maps*, or *hashes* – are abstract data-structures with myriad applications, from databases to dynamic languages. Abstractly, an index is 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. While appealingly simple, this abstract view is insufficient for reasoning about indexes that are accessed concurrently.

In this paper, we introduce an abstract specification which views an index as a divisible resource. Multiple threads can access the index concurrently, yet threads can still reason locally. We show that this specification can be used to verify a number of client applications. Our abstract specification would mean little if it were not satisfied by the implementations of concurrent indexes. We verify that our specification is satisfied by linked list, hash table and B^{Link} tree index implementations. During verification, we uncovered a subtle bug in the B^{Link} tree algorithm.

General Terms Algorithms, Concurrency, Theory, Verification.

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

1. Introduction

Indexes are ubiquitous in computer systems: they are used in the implementations of databases, caches, file systems, and even the objects of dynamic languages such as JavaScript. To a sequential client, an index can be viewed as a partial function from keys to values. The client can query the index by key, or mutate it by adding or removing mappings. A client can use an index in terms of this abstract specification, irrespective of the complexities of its implementation. The simplicity of this abstract view accounts for a large part of the popularity of indexes.

However, this simple 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 propose an abstract specification for concurrent indexes which takes account of interference between threads. Our specification allows a thread to reason locally if each key is manipulated by one thread. However, it also allows threads to share access to keys if necessary. Crucially, clients can reason entirely abstractly using our specification, without considering an index's implementation. However, we establish that our specification is satisfied by actual index implementations.

In §3 we propose a simple specification which treats the index as a resource divided up by its keys. Intuitively, if each key in an index is manipulated by a single thread then we can verify each thread in terms of the keys it uses, and combine the results to understand the composed system. In our specification, each key k can either be absent from the index h – represented by the predicate $\mathsf{out}(h, k)$ – or it can be present with some value v – represented by $\mathsf{in}(h, k, v)$. Inserting and removing a key is completely independent from operations on other keys. Disjointness is enforced by the fact that only one thread can hold the resource $\mathsf{out}(h, k)$ or $\mathsf{in}(h, k, v)$ for a particular k; that thread then has the exclusive right to read or mutate the value of the key. Using this disjoint specification, we verify a variety of clients, including a map procedure which concurrently applies a function to keys in a particular range (§3.1).

Our simplified specification assumes that each key is held by at most one thread at a time. In §4 we propose a more refined specification which allows sharing of keys between threads. Once again, keys are represented by predicates. However, our refined specification can express subtle patterns of behaviour over individual keys. For example, a thread may have the right to remove a key from the index, but other threads may also have that right. If no thread removes the key, our specification allows us to infer that the key is still present. With this specification, we can verify examples such as function memoization (§4.2) and a concurrent implementation of Eratosthenes' prime number sieve (§4.3).

We then discuss how our specification can be extended to represent iteration over indexes (§5). We show how to use this to reason about applications such as a more general version of map and a caching mechanism for a social networking website.

In order to justify our proposed specification, we have verified that it is satisfied by three concurrent index implementations: a simple linked list with coarse-grained locking (§6.1); a hash table linking to a set of secondary indexes (§6.2); and Sagiv's complicated B^{Link} tree algorithm [18] (§6.3). Our specification allows clients to reason disjointly about individual keys, even when the implementation requires underlying sharing between threads. For the B^{Link} tree algorithm in particular, the underlying sharing mechanism is exceedingly complex, so as to permit non-blocking reads. We use the *concurrent abstract predicate* methodology [6] to hide low-level sharing from clients. The benefits of our approach are illustrated by the fact that we discovered a bug in the B^{Link} algorithm.

Paper contributions:

1. An abstract specification for concurrent indexes that allows thread-local reasoning, while capturing the addition and removal of shared values. We also present an extension of this specification giving support for iteration over the keys of an index.
2. Verifications of a number of client algorithms based on indexes, including a map function, a memoisation function, and an implementation of the sieve of Eratosthenes. Our proofs demonstrate that our proposed specification allows us to reason about high-level coordination between threads.
3. Proofs that three different index implementations satisfy our proposed specification. Our implementations vary in complexity from a simple globally-locked list, to Sagiv's B^{Link} tree algorithm. Our abstract specification allows us to present an identical interface to clients, irrespective of the complexity of these implementations.

Related Work

We build most immediately on concurrent abstract predicates [6], a logic for modular verification based on separation logic. This approach developed from a line of concurrent logics, including RGSep [21] and concurrent separation logic [14]. The index specification we propose is descended from the set specification verified in [6]. However, in that paper we focussed on building a sound logic, and verified only simple specifications against naive implementations. In this paper we propose a specification which allows clients to reason straightforwardly about challenging features such as indexes and sharing, and we verify our specification against realistic implementations such as the B^{Link} tree.

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 [5]. Kuncak *et al.* propose an analysis that represents various kinds of data by set abstractions, while proving these abstractions for modules [12].

Our specification for iterators (§5) resembles the one proposed by Krishnaswami [11]. As in this work, we represent an iterator by a predicate, and we permit many iterators for a single index. Unlike Krishnaswami, our specification iterators on an index to run concurrently, and allows keys to be released once they have been iterated over. Furthermore, we can reason without the use of the separating implication (the 'magic wand') – an operator that has proved challenging for automated reasoning.

One of the most challenging parts of our work was verifying the concurrent B^{Link} tree implementation. Some prior work exists on verifying *sequential* B-trees. In [19] B-tree search and insert operations are verified as fault-free in a simplified sequential setting. In [13] a sequential B-tree implementation is verified in Coq as part of a relational database management system. The authors comment that the proof was "particularly difficult, despite previous work in this area", and 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 [16]. This paper verifies a highly-abstracted version of the algorithm modelled in process algebra, rather than C-like code. It also verifies a global specification, rather than allowing elements to be divided between threads.

2. Separation Logic & Abstraction

This paper is based on separation logic [17], 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 and 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 the frame F is modified by the program $\mathbb{C}$.

Separation logic provides straightforward reasoning about sequential programs. It also handles concurrency [14], using the following rule:

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

In a concurrent setting, the precondition and post-condition are interpreted as resources owned exclusively by the thread. Reasoning using Par is *thread-local*: each thread reasons purely about 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* [15]. 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 behaviour of insert without knowing about the concrete definition of the set predicate.

Abstract predicates can only represent the set as a single entity, because implementation details disrupt finer-grained abstractions. *Concurrent abstract predicates* [6], 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. insert might now be specified as:

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

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. In §3, §4 and §5, we work at the abstract level; in §6, we prove our abstraction is respected by implementations.

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[1]:

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

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. (In §4 we will extend this approach to allow reasoning about keys that are shared). 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 the

[1] 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.

following intuitive interpretation:

$$\begin{array}{rl} \mathsf{in}(h,k,v): & \text{there is a mapping in the index } h \text{ from } k \text{ to } v. \\ \mathsf{out}(h,k): & \text{there is no mapping in the index } h \text{ from } k. \end{array}$$

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 key. This entangling of state with ownership is essential to our approach: each predicate is invariant under the behaviour of other threads, meaning that its implementation can be abstracted.

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

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

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:

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

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:

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

We can prove this specification as follows:

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

In this proof, the $\mathtt{search}$ operation first uses the predicate $\mathsf{in}(\mathtt{h},\mathtt{k}_2,v)$ to retrieve the value v. Then the parallel rule hands $\mathtt{insert}$ and $\mathtt{remove}$ 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, k₁, k₂) {
 if (k₁ = k₂) {
   r := search(h, k₁);
   if (r ≠ nil) {
     remove(h, k₁);
     r := f(r);
     insert(h, k₁, r);
   }
 } else {
    rangeMap(h, k₁, k₁+((k₂-k₁)/2))
      || rangeMap(h, k₁+((k₂-k₁)/2)+1, k₂)
}}
```

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

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

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

```
$\left\{ \circledast_{k_1 \le i \le k_2}.\, \begin{array}{l} (\mathsf{out}(h,i) \wedge i \notin \mathsf{keys}(S)) \vee \\ \quad (\exists v.\, \mathsf{in}(h,i,v) \wedge (i,v) \in S) \end{array} \right\}$
rangeMap(h, k1, k2, v) {
 if (k1 = k2) {
  $\left\{ \begin{array}{l} k_1 = k_2 \wedge ((\mathsf{out}(h,k_1) \wedge k_1 \notin \mathsf{keys}(S)) \vee \\ (\exists v.\, \mathsf{in}(h,k_1,v) \wedge (k_1,v) \in S)) \end{array} \right\}$
  r := search(h, k1);
  $\left\{ \begin{array}{l} (\mathsf{out}(h,k_1) \wedge k_1 \notin \mathsf{keys}(S) \wedge r = \mathsf{nil}) \vee \\ (\mathsf{in}(h,k_1,r) \wedge (k_1,r) \in S) \end{array} \right\}$
  if (r ≠ nil) {
   $\{\mathsf{in}(h,k_1,r) \wedge (k_1,r) \in S\}$
   remove(h, k1);
   $\{\mathsf{out}(h,k_1) \wedge (k_1,r) \in S \wedge k_1 = k_2\}$
   r := f(r);
   $\{\exists v.\, \mathsf{out}(h,k_1) \wedge (k_1,v) \in S \wedge r = f(v)\}$
   insert(h, k1, r);
   $\{\exists v.\, \mathsf{in}(h,k_1,f(v)) \wedge (k_1,v) \in S\}$
  }
  $\left\{ \begin{array}{l} k_1 = k_2 \wedge ((\mathsf{out}(h,k_1) \wedge k_1 \notin \mathsf{keys}(S)) \vee \\ (\exists v.\, \mathsf{in}(h,k_1,f(v)) \wedge (k_1,v) \in S)) \end{array} \right\}$
 } else {
  $\left\{ \begin{array}{l} \circledast_{k_1 \le i \le \lfloor \frac{k_1+k_2}{2} \rfloor}.\, \left( \begin{array}{l} (\mathsf{out}(h,i) \wedge i \notin \mathsf{keys}(S)) \vee \\ (\exists v.\, \mathsf{in}(h,i,v) \wedge (i,v) \in S) \end{array} \right) * \\ \circledast_{\lfloor \frac{k_1+k_2}{2} \rfloor < i \le k_2}.\, \left( \begin{array}{l} (\mathsf{out}(h,i) \wedge i \notin \mathsf{keys}(S)) \vee \\ (\exists v.\, \mathsf{in}(h,i,v) \wedge (i,v) \in S) \end{array} \right) \end{array} \right\}$
  // Apply the PAR rule.
   rangeMap(h, k1, k1+((k2-k1)/2))
    || rangeMap(h, k1+((k2-k1)/2)+1, k2)
  $\left\{ \begin{array}{l} \circledast_{k_1 \le i \le \lfloor \frac{k_1+k_2}{2} \rfloor}.\, \left( \begin{array}{l} (\mathsf{out}(h,i) \wedge i \notin \mathsf{keys}(S)) \vee \\ (\exists v.\, \mathsf{in}(h,i,f(v)) \wedge (i,v) \in S) \end{array} \right) * \\ \circledast_{\lfloor \frac{k_1+k_2}{2} \rfloor < i \le k_2}.\, \left( \begin{array}{l} (\mathsf{out}(h,i) \wedge i \notin \mathsf{keys}(S)) \vee \\ (\exists v.\, \mathsf{in}(h,i,f(v)) \wedge (i,v) \in S) \end{array} \right) \end{array} \right\}$
}}
$\left\{ \circledast_{k_1 \le i \le k_2}.\, \begin{array}{l} (\mathsf{out}(h,i) \wedge i \notin \mathsf{keys}(S)) \vee \\ \quad (\exists v.\, \mathsf{in}(h,i,f(v)) \wedge (i,v) \in S) \end{array} \right\}$
```

Figure 1. Proof for rangeMap.

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.

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:

$$\mathtt{r} := \mathtt{search}(\mathtt{h},\mathtt{k}) \parallel \mathtt{remove}(\mathtt{h},\mathtt{k}) \qquad (1)$$

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 and threads can only insert values in set S at this key.

$\mathsf{in}_{\mathsf{rem}}(h, k, v)_i$: there may be a mapping from key k to value v and threads can only remove the value at this key.

$\mathsf{out}_{\mathsf{rem}}(h, k)_i$: 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 and the current thread may not change it, but other threads can make any modification.

The subscripts def, ins and rem and the fractional component $i \in (0, 1]$ 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 defining the ways that predicates can be split and joined. For example:

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

As in Boyland [2], fractional permissions are used to record splittings. A permission value $i \in (0, 1)$ 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 (when $i = 1$) the thread can add or remove the key freely. The subscripts def, ins and rem specify what a thread can do when it shares access to the key – that is, when $i \in (0, 1)$. 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, 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, denoting that the key k is associated with v in the index. 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 one each to two sub-threads. Assume the first thread does not modify the key, but the second calls $\mathtt{remove}(h, k)$, which has the following specification:

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

The result is uncertainty: one thread holds the $\mathsf{out}_{\mathsf{rem}}(h, k)_{\frac{1}{2}}$ predicate, while the other holds the $\mathsf{in}_{\mathsf{rem}}(h, k, v)_{\frac{1}{2}}$ predicate. 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 some take singleton values. We use singleton values when we know a key has that value. We use a set of values when concurrent inserts are possible (i.e. in the ins and unk environments), 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 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 a 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.

Our full specification is given in Figure 3. In the definition of the axioms, X is used to stand for $\mathsf{in}_{\mathsf{def}}(h, k, v)$, $\mathsf{out}_{\mathsf{def}}(h, k)$, $\mathsf{in}_{\mathsf{ins}}(h, k, S)$, $\mathsf{out}_{\mathsf{ins}}(h, k, S)$, $\mathsf{in}_{\mathsf{rem}}(h, k, v)$, $\mathsf{out}_{\mathsf{rem}}(h, k)$ and $\mathsf{unk}(h, k, S)$.

4.1 Proving Simple Examples

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

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

SPECIFICATIONS:

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

$$\begin{array}{rll}
\{\mathsf{in}_{\mathsf{def}}(\mathtt{h},\mathtt{k},v)_i\} & \mathtt{insert}(\mathtt{h},\mathtt{k},\mathtt{v}') & \{\mathsf{in}_{\mathsf{def}}(\mathtt{h},\mathtt{k},v)_i\} \\
\{\mathsf{out}_{\mathsf{def}}(\mathtt{h},\mathtt{k})_1\} & \mathtt{insert}(\mathtt{h},\mathtt{k},\mathtt{v}) & \{\mathsf{in}_{\mathsf{def}}(\mathtt{h},\mathtt{k},\mathtt{v})_1\} \\
\{(\mathsf{in}_{\mathsf{ins}}(\mathtt{h},\mathtt{k},S)_i \lor \mathsf{out}_{\mathsf{ins}}(\mathtt{h},\mathtt{k},S)_i) \land \mathtt{v} \in S\} & \mathtt{insert}(\mathtt{h},\mathtt{k},\mathtt{v}) & \{\mathsf{in}_{\mathsf{ins}}(\mathtt{h},\mathtt{k},S)_i\} \\
\{\mathsf{unk}(\mathtt{h},\mathtt{k},S)_i \land \mathtt{v} \in S\} & \mathtt{insert}(\mathtt{h},\mathtt{k},\mathtt{v}) & \{\mathsf{unk}(\mathtt{h},\mathtt{k},S)_i\}
\end{array}$$

$$\begin{array}{rll}
\{\mathsf{in}_{\mathsf{def}}(\mathtt{h},\mathtt{k},v)_1\} & \mathtt{remove}(\mathtt{h},\mathtt{k}) & \{\mathsf{out}_{\mathsf{def}}(\mathtt{h},\mathtt{k})_1\} \\
\{\mathsf{out}_{\mathsf{def}}(\mathtt{h},\mathtt{k})_i\} & \mathtt{remove}(\mathtt{h},\mathtt{k}) & \{\mathsf{out}_{\mathsf{def}}(\mathtt{h},\mathtt{k})_i\} \\
\{\mathsf{in}_{\mathsf{rem}}(\mathtt{h},\mathtt{k},v)_i \lor \mathsf{out}_{\mathsf{rem}}(\mathtt{h},\mathtt{k})_i\} & \mathtt{remove}(\mathtt{h},\mathtt{k}) & \{\mathsf{out}_{\mathsf{rem}}(\mathtt{h},\mathtt{k})_i\} \\
\{\mathsf{unk}(\mathtt{h},\mathtt{k},S)_i\} & \mathtt{remove}(\mathtt{h},\mathtt{k}) & \{\mathsf{unk}(\mathtt{h},\mathtt{k},S)_i\}
\end{array}$$

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.

		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.

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

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

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 $\mathtt{k}$ 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 left-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 right-hand thread to remove the value successfully, as we know that it is the only thread changing the shared state for the key $\mathtt{k}$. When both threads finish their execution we use the same axiom to merge the $\mathsf{read}(\mathtt{h},\mathtt{k})$ predicate back into the $\mathsf{out}_{\mathsf{def}}(\mathtt{h},\mathtt{k})_1$ predicate.

We can prove the second specification as follows:

$$\begin{array}{c}
\{\mathsf{out}_{\mathsf{def}}(\mathtt{h},\mathtt{k})_1\} \\
\{\mathsf{out}_{\mathsf{def}}(\mathtt{h},\mathtt{k})_{\frac{1}{2}} * \mathsf{out}_{\mathsf{def}}(\mathtt{h},\mathtt{k})_{\frac{1}{2}}\} \\
\begin{array}{r||l}
\{\mathsf{out}_{\mathsf{def}}(\mathtt{h},\mathtt{k})_{\frac{1}{2}}\} & \{\mathsf{out}_{\mathsf{def}}(\mathtt{h},\mathtt{k})_{\frac{1}{2}}\} \\
\mathtt{r} := \mathtt{search}(\mathtt{h},\mathtt{k}) & \mathtt{remove}(\mathtt{h},\mathtt{k}) \\
\{\mathsf{out}_{\mathsf{def}}(\mathtt{h},\mathtt{k})_{\frac{1}{2}} \land \mathtt{r} = \mathsf{nil}\} & \{\mathsf{out}_{\mathsf{def}}(\mathtt{h},\mathtt{k})_{\frac{1}{2}}\}
\end{array} \\
\{\mathsf{out}_{\mathsf{def}}(\mathtt{h},\mathtt{k})_{\frac{1}{2}} * \mathsf{out}_{\mathsf{def}}(\mathtt{h},\mathtt{k})_{\frac{1}{2}} \land \mathtt{r} = \mathsf{nil}\} \\
\{\mathsf{out}_{\mathsf{def}}(\mathtt{h},\mathtt{k})_1\}
\end{array}$$

Here we use the splitting axiom discussed on the previous page. Unlike the previous proof-sketch, the `remove` operation does not modify the index in this case.

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

$$\mathtt{remove}(\mathtt{h},\mathtt{k}) \parallel \mathtt{remove}(\mathtt{h},\mathtt{k})$$

In this program, we do not know which `remove` will succeed and which will fail, but we do know that there will definitely not be a mapping from key `k` afterwards. By splitting the predicates, we can communicate this knowledge between the threads.

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

We sometimes cannot establish the exact state of an index after a program has run. For example, consider the following program:

$$\mathtt{remove}(\mathtt{h},\mathtt{k}) \parallel \mathtt{insert}(\mathtt{h},\mathtt{k},\mathtt{v})$$

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. However, we can still establish that the program does not fault and that if the key is assigned, then it will have value `v`

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

We now consider a pair of more complex examples: function memoization, and the sieve of Eratosthenes.

4.2 Example: Memoization

A common application of indexes is memoization: storing the results of expensive computations to avoid having to recompute them. Our specification can be used to 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:

```
$\{\exists i \in (0,1].\, \circledast_{v'}.\, \mathsf{unk}(\mathtt{memo}, v', \{f(v')\})_i\}$
memoized_f(v) {
  $\{\circledast_{v'}.\, \mathsf{unk}(\mathtt{memo}, v', \{f(v')\})_i\}$
  // framing 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 = null) {
    $\{\mathsf{unk}(\mathtt{memo}, \mathtt{v}, \{f(\mathtt{v})\})_i\}$
    r := f(v);
    $\{\mathsf{unk}(\mathtt{memo}, \mathtt{v}, \{f(\mathtt{v})\})_i \wedge \mathtt{r} = f(\mathtt{v})\}$
    insert(memo, v, r);
    $\{\mathsf{unk}(\mathtt{memo}, \mathtt{v}, \{f(\mathtt{v})\})_i \wedge \mathtt{r} = f(\mathtt{v})\}$
  }
  $\{\mathsf{unk}(\mathtt{memo}, \mathtt{v}, \{f(\mathtt{v})\})_i \wedge \mathtt{r} = f(\mathtt{v})\}$
  // framing the other 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.

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

We give memoized_f the following specification:

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

Here memo is some splittable abstract predicate (that is, memo = memo ∗ memo). Such a specification allows calls to f to be replaced with memoized_f, even in parallel. We define memo as follows:

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

A proof of the specification for memoized_f is shown in Figure 4.

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, 10], an algorithm for generating all of the prime numbers up to a given maximum value 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.

```
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 5. Prime sieve functions.

The algorithm starts by constructing a set of integers from 2 (since 1 is not a prime number) to max. (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 5. 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

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

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

$$\mathsf{fac}(i, v, v') \quad \triangleq \quad \exists j.\, v \leq j \leq v' \wedge j \neq i \wedge (i \bmod j) = 0$$

The proof that sieve meets its specification is given in Figure 6. This proof requires we establish the following specification for worker.

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

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 $v \leq v'$)

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

allows us to conclude that the `parwork` procedure eliminates exactly the set elements with factors different from themselves in the range $\mathtt{v} \mathrel{..} \mathtt{max}$. Since $p > 1$ is prime if and only if it has a factor in the range $2 \mathrel{..} \lfloor\sqrt{p}\rfloor$, for $i \in [2 \mathrel{..} \mathtt{max}]$

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

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:

- $\mathtt{it} := \mathtt{createIter}(\mathtt{h})$ creates a new iterator for index `h`.
- $(\mathtt{k}, \mathtt{v}) := \mathtt{next}(\mathtt{it})$ 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).
- $\mathtt{destroyIter}(\mathtt{it})$ 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 the `next` procedure.

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 7. 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

```
{emp ∧ 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)) {
    {$\left(2 \leq \mathtt{v} \wedge \circledast_{i\in[2..\mathtt{max}]}.\ \mathsf{in}_{\mathsf{rem}}(\mathtt{idx}, i, 0)_{\frac{t}{2}}\right) *$
     $\left(2 \leq \mathtt{v}+1 \wedge \circledast_{i\in[2..\mathtt{max}]}.\ \mathsf{in}_{\mathsf{rem}}(\mathtt{idx}, i, 0)_{\frac{t}{2}}\right)$}
    worker(v, max, idx) ∥ parwork(v+1, max, idx)
    {$\left(\circledast_{i\in[2..\mathtt{max}]}.\ \mathsf{fac}(i, \mathtt{v}, \mathtt{v}) \Rightarrow \mathsf{out}_{\mathsf{rem}}(\mathtt{idx}, i)_{\frac{t}{2}} \wedge\right.$
          $\left.\neg\mathsf{fac}(i, \mathtt{v}, \mathtt{v}) \Rightarrow \mathsf{in}_{\mathsf{rem}}(\mathtt{idx}, i, 0)_{\frac{t}{2}}\right) *$
     $\circledast_{i\in[2..\mathtt{max}]}.\ \mathsf{fac}(i, \mathtt{v}+1, \lfloor\sqrt{\mathtt{max}}\rfloor) \Rightarrow \mathsf{out}_{\mathsf{rem}}(\mathtt{idx}, i)_{\frac{t}{2}}$
          $\wedge \neg\mathsf{fac}(i, \mathtt{v}+1, \lfloor\sqrt{\mathtt{max}}\rfloor) \Rightarrow \mathsf{in}_{\mathsf{rem}}(\mathtt{idx}, i, 0)_{\frac{t}{2}}$}
    // 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) {
   {$\circledast_{i\in[2..(\mathtt{c}-1)]}.\ \mathsf{fac}(i, \mathtt{v}, \mathtt{v}) \Rightarrow \mathsf{out}_{\mathsf{def}}(\mathtt{idx}, i)_t \wedge$
        $\neg\mathsf{fac}(i, \mathtt{v}, \mathtt{v}) \Rightarrow \mathsf{in}_{\mathsf{rem}}(\mathtt{idx}, i, 0)_t$
        $* \circledast_{j\in[\mathtt{c}..\mathtt{max}]}.\ \mathsf{in}_{\mathsf{rem}}(\mathtt{idx}, j, 0)_t$}
   remove(idx, c);
   c := c + v;
 }
}
{$\circledast_{i\in[2..\mathtt{max}]}.\ \mathsf{fac}(i, \mathtt{v}, \mathtt{v}) \Rightarrow \mathsf{out}_{\mathsf{def}}(\mathtt{idx}, i)_t \wedge$
      $\neg\mathsf{fac}(i, \mathtt{v}, \mathtt{v}) \Rightarrow \mathsf{in}_{\mathsf{rem}}(\mathtt{idx}, i, 0)_t$}
```

Figure 6. Proofs for the `sieve` and `worker` programs.

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

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

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

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

Figure 7. Specification for iterators.

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.

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.

```
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 `map_f` is given in Figure 8.

Example: website caching. Our specification does not restrict the type of value that can be stored in an index. If we store pointers to other indexes, we can create an n-dimensional index. If we view indexes as tables, we can interpret such an index structure as a rudimentary database. Such structures, sometimes called ‘NOSQL’ databases, have recently become popular [22]. Compared to standard SQL-based databases, they trade robustness for speed and conceptual simplicity. NOSQL databases are often used by large websites for caching queries to their more traditional SQL-style back-end database. Our iterator specification allows us to verify a simple NOSQL-style cache.

```
{$\circledast_{(k,v)\in S}\,\mathsf{in}_{\mathsf{def}}(\mathtt{h},k,v)_1 \;*\; \circledast_{k\notin\mathsf{keys}(S)}\,\mathsf{out}_{\mathsf{def}}(\mathtt{h},k)_1$}
 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);
  {$(\mathtt{k},\mathtt{v})\in S\wedge\mathsf{iter}(\mathtt{it},\mathtt{h},S\setminus\{(\mathtt{k},\mathtt{v})\},K,1) * \mathsf{in}_{\mathsf{def}}(\mathtt{h},\mathtt{k},\mathtt{v})$
                     $\vee\ \mathsf{iter}(\mathtt{it},\mathtt{h},\emptyset,K,1)\wedge\mathtt{k}=\mathsf{nil}\wedge\mathtt{v}=\mathsf{nil}$}
   if (k ≠ nil) {
    {$(\mathtt{k},\mathtt{v})\in S\wedge\mathsf{iter}(\mathtt{it},\mathtt{h},S\setminus\{(\mathtt{k},\mathtt{v})\},K,1) * \mathsf{in}_{\mathsf{def}}(\mathtt{h},\mathtt{k},\mathtt{v})$}
     (
      {$\mathsf{in}_{\mathsf{def}}(\mathtt{h},\mathtt{k},\mathtt{v})$}
       remove(h, k); insert(h, k, f(v));
      {$\mathsf{in}_{\mathsf{def}}(\mathtt{h},\mathtt{k},f(\mathtt{v}))$}
     ) ||
    {$\mathsf{iter}(\mathtt{it},\mathtt{h},S\setminus(\mathtt{k},\mathtt{v}),K,1)$}
     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 8. Sketch-proof for `map_f`.

Consider a Facebook-like site where users upload and comment on pictures. Each picture has a unique identifier, and each user is associated with an index. In a user index there are two keys: `pics`, mapping picture identifiers to picture data and; `cmts`, mapping pairs of user index identifiers and picture identifiers to a comment string. An instance of this database with two users, four pictures and one comment (from the second user to the first, about picture ID 3) would be:

$$
\begin{aligned}
\mathit{user}_1 &: [\texttt{pics} \mapsto P_1, \texttt{cmts} \mapsto C_1] \\
\mathit{user}_2 &: [\texttt{pics} \mapsto P_2, \texttt{cmts} \mapsto C_2] \\
P_1 &: [1 \mapsto \langle \mathit{data} \rangle, 3 \mapsto \langle \mathit{data} \rangle] \\
P_2 &: [2 \mapsto \langle \mathit{data} \rangle, 4 \mapsto \langle \mathit{data} \rangle] \\
C_1 &: [(\mathit{user}_2, 3) \mapsto \text{“Great picture!”}] \\
C_2 &: [\,]
\end{aligned}
$$

A user can add a comment on a picture by making a request to the web server. We do not model the entire server, but assume it will eventually invoke a function cmtPic to update the cache:

```
cmtPic(by, on, pID, cmt){
 commId := search(on, cmts);
 picsId := search(on, pics);
 pic := search(picsId, pID);
 if (pic ≠ nil){
   insert(commId, (by, pID), cmt);
 }
 return pic ≠ nil;
}
```

This function retrieves the comments and pictures index from the user. It then checks to see if the picture is still in the database and, if so, inserts the comment. Using our specification for iterators, we can prove the following specification:

$$
\left\{\begin{array}{l}
\mathsf{in}_{\mathsf{def}}(\texttt{on}, \texttt{cmts}, C)_i * \mathsf{in}_{\mathsf{def}}(\texttt{on}, \texttt{pics}, P)_j * \\
\mathsf{read}(P, \texttt{pID}) * \mathsf{out}_{\mathsf{ins}}(C, (\mathsf{by}, \texttt{pID}), \{\texttt{cmt}\})_k
\end{array}\right\}
$$

$\texttt{cmtPic(by, on, pID, cmt)}$

$$
\left\{\begin{array}{l}
\mathsf{in}_{\mathsf{def}}(\texttt{on}, \texttt{cmts}, C)_i * \mathsf{in}_{\mathsf{def}}(\texttt{on}, \texttt{pics}, P)_j * \\
\mathsf{read}(P, \texttt{pID}) * \left(\begin{array}{l}
(\neg \mathrm{ret} \wedge \mathsf{out}_{\mathsf{ins}}(C, (\mathsf{by}, \texttt{pID}), \{\texttt{cmt}\})_k) \\
\quad \vee (\mathrm{ret} \wedge \mathsf{in}_{\mathsf{ins}}(C, (\mathsf{by}, \texttt{pID}), \{\texttt{cmt}\})_k)
\end{array}\right)
\end{array}\right\}
$$

Users may want to delete embarrassing pictures that they have uploaded by accident. We can define a deletePic function as follows:

```
deletePic(on, pID) {
 commId := search(on, cmts);
 picsId := search(on, pics);
 remove(picsId, pID);
 it := createIter(commId);
 (k, v) := next(it);
 while (k ≠ nil) {
   (o, p) := k;
   if (p = pID){
     remove(commId, k);
   }
   (k, v) := next(it);
 }
 destroyIter(it);
}
```

To ensure user privacy, our web site should make strong guarantees that once deletion is requested, both the picture and the comments pertaining to the picture are destroyed. We can prove the following specification for deletePic:

$$
\left\{\begin{array}{l}
\mathsf{in}_{\mathsf{def}}(\texttt{on}, \texttt{cmts}, C)_i * \mathsf{in}_{\mathsf{def}}(\texttt{on}, \texttt{pics}, P)_j * \mathsf{in}_{\mathsf{def}}(P, \texttt{pID})_1 \\
\; * \circledast_{(k,v) \in S} \mathsf{in}(C, k, v)_1 * \circledast_{k \notin \mathsf{keys}(S)} \mathsf{out}(C, k)_1
\end{array}\right\}
$$

$\texttt{deletePic(on, pID)}$

$$
\left\{\begin{array}{l}
\mathsf{in}_{\mathsf{def}}(\texttt{on}, \texttt{cmts}, C)_i * \mathsf{in}_{\mathsf{def}}(\texttt{on}, \texttt{pics}, P)_j * \mathsf{out}_{\mathsf{def}}(P, \texttt{pID})_1 \\
\; * \circledast_{(k,v) \in S \setminus S'} \mathsf{in}_{\mathsf{def}}(C, k, v)_1 * \circledast_{k \notin \mathsf{keys}(S \setminus S')} \mathsf{out}_{\mathsf{def}}(C, k)_1 \\
\; \wedge S' = \{(k', v') | v' = \texttt{pID}\}
\end{array}\right\}
$$

In isolation, both of these functions perform their functions correctly. However, our specifications reveal a defect. We may have a situation where a user is attempting to delete a picture, whilst another user is adding a comment:

$$\texttt{cmtPic}(\texttt{p2}, \texttt{p1}, 3, \texttt{c}) \; || \; \texttt{deletePic}(\texttt{p1}, 3)$$

The appropriate precondition for this case is the following:

$$
\left\{\begin{array}{l}
\mathsf{in}_{\mathsf{def}}(\texttt{p1}, \texttt{cmts}, C) * \mathsf{in}_{\mathsf{def}}(\texttt{p1}, \texttt{pics}, P) * \mathsf{in}_{\mathsf{def}}(P, 3)_1 \\
\; * \circledast_{(k,v) \in S} \mathsf{in}_{\mathsf{def}}(C, k, v)_1 * \circledast_{k \notin \mathsf{keys}(S)} \mathsf{out}_{\mathsf{def}}(C, k)_1
\end{array}\right\}
$$

To ensure the correct behaviour of deletePic, we must give the deletion thread sufficient permissions to ensure any key representing a comment on picture 3 is removed. This is reflected in the pre-condition as the iterated conjunction of full permission $\mathsf{in}_{\mathsf{def}}$ and $\mathsf{out}_{\mathsf{def}}$ predicates for every key. However, the comment insertion thread requires a least a fractional $\mathsf{out}_{\mathsf{ins}}$ predicate for one key of the comments index. We cannot split the index resource to prove the parallel composition of the two procedures. The two processes are not thread-safe with respect to each other.

Revised caching. We can correct this problem with a lock, enforcing mutual exclusion on the comments table. However, this goes against the ethos of a cache, where speed of access is critical. Instead, we redesign the index structure so that rather than picture identifiers mapping to just pictures, they map to a picture along with comments about it. Deleting a picture now implicitly removes the comments associated with it; if a comment thread accesses this resource concurrently no harm occurs, as it has become disassociated from the picture and will eventually fall out of the cache. The revised code for deletePic and cmtPic is as follows:

```
deletePic2(on,pID){          cmtPic2(by,on,pID,cmt){
 picsId:=search(on,pics);     picsId:=search(on,pics);
 remove(picsId,pID);          pic:=search(picsId,pID);
}                             if (pic≠nil){
                                insert(pic,by,cmt);
                              }
                              return pic≠nil;
                             }
```

We can prove the following specifications for the revised functions:

$$\{\mathsf{in}_{\mathsf{def}}(\mathtt{on}, \mathtt{pics}, P)_i * \mathsf{in}_{\mathsf{rem}}(P, \mathtt{pID}, C)_j\}$$

```
deletePic2(on, pID)
```

$$\{\mathsf{in}_{\mathsf{def}}(\mathtt{on}, \mathtt{pics}, P)_i * \mathsf{out}_{\mathsf{rem}}(P, \mathtt{pID})_j\}$$

$$\left\{\begin{array}{l}\mathsf{in}_{\mathsf{def}}(\mathtt{on}, \mathtt{pics}, P)_i * \mathsf{in}_{\mathsf{rem}}(P, \mathtt{pID}, C)_j \\ \qquad\qquad * \mathsf{out}_{\mathsf{ins}}(C, \mathtt{by}, \{\mathtt{cmt}\})_k\end{array}\right\}$$

```
cmtPic2(by, on, pID, cmt)
```

$$\left\{\begin{array}{l}\mathsf{in}_{\mathsf{def}}(\mathtt{on}, \mathtt{pics}, P)_i * \mathsf{in}_{\mathsf{rem}}(P, \mathtt{pID}, C)_j \\ \quad * \left(\begin{array}{l}(\neg\mathrm{ret} \wedge \mathsf{out}_{\mathsf{ins}}(C, \mathtt{by}, \{\mathtt{cmt}\})_k) \\ \vee\ (\mathrm{ret} \wedge \mathsf{in}_{\mathsf{ins}}(C, \mathtt{by}, \{\mathtt{cmt}\})_k)\end{array}\right)\end{array}\right\}$$

The new `deletePic`, even when run in parallel with the new `cmtPic`, can successfully acquire the needed resource to remove the comments without requiring locks.

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 use it to develop our approach. We then prove a hash-table implementation satisfies our full specification. Finally, we show that our approach scales to quite complex implementations by outlining our proof of the B^{Link} tree algorithm.

Approach: Concurrent Abstract Predicates. We use concurrent abstract predicates (CAP) [6] 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, $\boxed{P}^r_I$. The assertion P denotes the contents of the region, r is the name of the region, and I is an environment specifying what 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 other threads, meaning that assertions about shared regions must be *stable* – 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, 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.

```
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 9. Linked list operations.

In the discussion below, we assume the proof system and semantics given in [6], and only give details necessary for understanding the proof structure. The interested reader is referred to [6] for other technical details, including a proof of soundness for the CAP logic.

6.1 Linked List Implementation

To illustrate our approach, we first consider a simple index implementation: a linked list with a single lock protecting the entire list[2]. The code for this implementation is given in Figure 9. In order to simplify the presentation, in this section we only consider the simplified specification from §3. Some additional measures are required to handle the full specification given in §4, which we take in §6.3 to verify the B^{Link} tree implementation against the full 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

[2] This example is quite similar to the coarse-grained set example from [6].

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. When it finds it, 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 $\mathsf{ls}(a, S)$, corresponding to list with first element a and key-value elements S.

$$\begin{aligned}
\mathsf{node}(a, k, v, n) &\triangleq a.\mathtt{key} \mapsto k * a.\mathtt{val} \mapsto v * a.\mathtt{nxt} \mapsto n \\
\mathsf{lseg}(a, b, S) &\triangleq \exists k, v, n.\, (k, v) \in S \wedge \\
&\qquad \mathsf{node}(a, k, v, n) * \mathsf{ls}(n, S \setminus (k, v)) \\
&\qquad \vee (a = b \wedge S = \emptyset) \\
\mathsf{ls}(a, S) &\triangleq \mathsf{lseg}(a, \mathsf{nil}, S)
\end{aligned}$$

Using the ls predicate, we can give a concrete interpretation to our index predicates for the linked list implementation of an index. For example, we give the following definition for the $\mathsf{in}(h, k, v)$ predicate:

$$\begin{aligned}
\mathsf{in}(h, k, v) \triangleq\ & \exists r, l, S.\, (k, v) \in S \wedge \\
& \boxed{\mathsf{lock}(h.\mathtt{lk}, r, k) * h.\mathtt{nxt} \mapsto l * \mathsf{ls}(l, S)}^{r}_{I(r,h)} \\
& * [\textsc{Lock}(k)]^{r}_{1}
\end{aligned}$$

($\mathsf{out}(h, k, v)$ is defined analogously by replacing $\in$ with $\notin$.)

Here the assertion surrounded by a box describes the region r shared between all the threads that can access the list. The boxed assertion says that region r contains a lock at $h.\mathtt{lk}$ (we define the predicate lock below) and a dummy next pointer $h.\mathtt{nxt}$, pointing to the main list – this is needed for in-place node removal. The set representing the content of the list is existentially quantified. However, we require that (k, v) is a member of the set.

The parts of the assertion not contained in a box are thread-local, meaning they are accessible to only the current thread. In the case of $\mathsf{in}(h, k, v)$ the local state contains the capability $[\textsc{Lock}(k)]^{r}_{1}$. This says that the current thread is allowed to acquire the lock, and to subsequently add or remove the key k from the list. The superscript r denotes that the capability is over region r, while the subscript 1 denotes that this is an *exclusive* capability. No other thread can perform this operation.

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

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

This 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. Intuitively, if the lock is held, one of the capabilities is in use by some thread. If not, all the capabilities are present.

Describing Interference. The meaning of the capabilities $[\textsc{Lock}(k)]^{r}_{1}$ and $[\textsc{Mod}(k)]^{r}_{1}$ is controlled by an *interference environment*. 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 $P \leadsto Q$. When a thread holds a capability mapped to an action ($P \leadsto Q$), it is permitted to replace a part of the region matching P with a part matching Q.

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

$$\textsc{Mod}(k) : \begin{cases}
h.\mathtt{nxt} \mapsto l * \mathsf{ls}(l, S) \wedge k \notin \mathsf{keys}(S) \\
\qquad \leadsto\ h.\mathtt{nxt} \mapsto l' * \mathsf{ls}(l', S \cup \{(k, v)\}) \\
h.\mathtt{nxt} \mapsto l * \mathsf{ls}(l, S) \\
\qquad \leadsto\ h.\mathtt{nxt} \mapsto l' * \mathsf{ls}(l', S \setminus \{(k, v)\})
\end{cases}$$

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

The definition of $\textsc{Mod}(k)$ says that a thread holding a capability $[\textsc{Mod}(k)]^{r}_{1}$ is allowed to replace the list with one with k added to or removed from the carrier set S. 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 $\textsc{Lock}(k)$ also says that a thread acquiring the lock also acquires the capability $[\textsc{Mod}(k)]^{r}_{1}$, and that 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. Once we have 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 10 shows one such proof, establishing that the implementation of `insert` matches the following abstract specification:

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

We write $-$ to indicate an unknown, existentially quantified value. 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 is allowed because the thread holds the capability $[\textsc{Mod}(\mathtt{k})]^{r}_{1}$. The definition of the capability also generates the obligation that `h.nxt` points to a list containing the same set of key-value pairs, apart from `k`. This ensures that the assertions in the proof are stable under interference from the environment. In fact, once the list is locked we know that there can be no interference from other threads, as we know all other $\textsc{Mod}$ capabilities are held by the lock.

```
$\{\mathsf{out}(\mathtt{h},\mathtt{k})\}$
insert(h, k, v) {
  $\left\{\begin{array}{l}\exists r,l,S.\,(\mathtt{k},-)\notin S\land\\ \boxed{\mathsf{lock}(\mathtt{h.lk},r,\mathtt{k}) * \mathtt{h.nxt}\mapsto l * \mathsf{ls}(l,S)}^{r}_{I(r,h)}\\ \qquad * [\textsc{Lock}(\mathtt{k})]^{r}_{1}\end{array}\right\}$
  lock(h.lk); // use the capability $[\textsc{Lock}(\mathtt{k})]^{r}_{1}$.
  $\left\{\begin{array}{l}\exists r,l,S.\,(\mathtt{k},-)\notin S\land\\ \boxed{\begin{array}{r}\mathtt{h.lk}\mapsto 1 * \circledast_{i\in\mathsf{Keys}\setminus\{\mathtt{k}\}}[\textsc{Mod}(i)]^{r}_{1}\\ * \mathtt{h.nxt}\mapsto l * \mathsf{ls}(l,S)\end{array}}^{r}_{I(r,h)}\\ \qquad * [\textsc{Mod}(\mathtt{k})]^{r}_{1} * [\textsc{Lock}(\mathtt{k})]^{r}_{1}\end{array}\right\}$
  e := h.nxt;
  while (e $\neq$ nil) {
    $\left\{\begin{array}{l}\exists r,l,S,k',v',n.\,(\mathtt{k},-)\notin S\land\\ \boxed{\begin{array}{r}\mathtt{h.lk}\mapsto 1 * \circledast_{i\in\mathsf{Keys}\setminus\{\mathtt{k}\}}[\textsc{Mod}(i)]^{r}_{1} * \mathtt{h.nxt}\mapsto l *\\ \mathsf{lseg}(l,\mathsf{e},S_1) * \mathsf{node}(\mathsf{e},k',v',n) * \mathsf{ls}(n,S_2)\\ \land S_1\uplus S_2\uplus\{(k',v')\}=S\end{array}}^{r}_{I(r,h)}\\ \qquad * [\textsc{Mod}(\mathtt{k})]^{r}_{1} * [\textsc{Lock}(\mathtt{k})]^{r}_{1}\end{array}\right\}$
    if (e.key = k) {
      $\{\mathsf{false}\}$ // this branch is for k in the set
      unlock(h.lk);
      return;
    }
    e := e.nxt;
    $\left\{\begin{array}{l}\exists r,l,S.\,(\mathtt{k},-)\notin S\land\\ \boxed{\begin{array}{r}\mathtt{h.lk}\mapsto 1 * \circledast_{i\in\mathsf{Keys}\setminus\{\mathtt{k}\}}[\textsc{Mod}(i)]^{r}_{1} * \mathtt{h.nxt}\mapsto l *\\ \mathsf{lseg}(l,\mathsf{e},S_1) * \mathsf{ls}(\mathsf{e},S_2)\land S_1\uplus S_2=S\end{array}}^{r}_{I(r,h)}\\ \qquad * [\textsc{Mod}(\mathtt{k})]^{r}_{1} * [\textsc{Lock}(\mathtt{k})]^{r}_{1}\end{array}\right\}$
  }
  $\left\{\begin{array}{l}\exists r,l,S.\,(\mathtt{k},-)\notin S\land\\ \boxed{\begin{array}{r}\mathtt{h.lk}\mapsto 1 * \circledast_{i\in\mathsf{Keys}\setminus\{\mathtt{k}\}}[\textsc{Mod}(i)]^{r}_{1} * \mathtt{h.nxt}\mapsto l\\ * \mathsf{ls}(l,S)\end{array}}^{r}_{I(r,h)}\\ \qquad * [\textsc{Mod}(\mathtt{k})]^{r}_{1} * [\textsc{Lock}(\mathtt{k})]^{r}_{1}\end{array}\right\}$
  e := makeNode(k,v,h.nxt);
  $\left\{\begin{array}{l}\exists r,l,S.\,(\mathtt{k},-)\notin S\land\\ \boxed{\begin{array}{r}\mathtt{h.lk}\mapsto 1 * \circledast_{i\in\mathsf{Keys}\setminus\{\mathtt{k}\}}[\textsc{Mod}(i)]^{r}_{1} * \mathtt{h.nxt}\mapsto l\\ * \mathsf{ls}(l,S)\end{array}}^{r}_{I(r,h)}\\ \qquad * \mathsf{node}(\mathsf{e},\mathtt{k},\mathtt{v},l) * [\textsc{Mod}(\mathtt{k})]^{r}_{1} * [\textsc{Lock}(\mathtt{k})]^{r}_{1}\end{array}\right\}$
  h.nxt := e; // use the capability $[\textsc{Mod}(\mathtt{k})]^{r}_{1}$.
  $\left\{\begin{array}{l}\exists r,l,S.\,(\mathtt{k},-)\notin S\land\\ \boxed{\begin{array}{r}\mathtt{h.lk}\mapsto 1 * \circledast_{i\in\mathsf{Keys}\setminus\{\mathtt{k}\}}[\textsc{Mod}(i)]^{r}_{1} * \mathtt{h.nxt}\mapsto \mathsf{e}\\ * \mathsf{node}(\mathsf{e},\mathtt{k},\mathtt{v},l) * \mathsf{ls}(l,S)\end{array}}^{r}_{I(r,h)}\\ \qquad * [\textsc{Mod}(\mathtt{k})]^{r}_{1} * [\textsc{Lock}(\mathtt{k})]^{r}_{1}\end{array}\right\}$
  unlock(h.lk); // use the capability $[\textsc{Lock}(\mathtt{k})]^{r}_{1}$.
  $\left\{\begin{array}{l}\exists r,l,S.\,(\mathtt{k},\mathtt{v})\in S\land\\ \boxed{\mathsf{lock}(\mathtt{h.lk},r,\mathtt{k}) * \mathtt{h.nxt}\mapsto l * \mathsf{ls}(l,S)}^{r}_{I(r,h)}\\ \qquad * [\textsc{Lock}(\mathtt{k})]^{r}_{1}\end{array}\right\}$
}
$\{\mathsf{in}(\mathtt{h},\mathtt{k},\mathtt{v})\}$
```

Figure 10. Proof outline for linked list `insert`.

Verifying the axioms. As well as proving the specifications for the operations, our other obligation in establishing that implementation satisfies the axioms of the abstract specification. To do this, we use the concrete definitions for the abstract predicates. To illustrate this, we will 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:

$$\begin{array}{l}\exists r,l,S.\,(k,v)\in S\land[\textsc{Lock}(k)]^{r}_{1} *\\ \boxed{\mathsf{lock}(h.\mathtt{lk},r,k) * h.\mathtt{nxt}\mapsto l * \mathsf{ls}(l,S)}^{r}_{I(r,h)} *\\ \exists r,l,S.\,(k,-)\notin S\land[\textsc{Lock}(k)]^{r}_{1} *\\ \boxed{\mathsf{lock}(h.\mathtt{lk},r,k) * h.\mathtt{nxt}\mapsto l * \mathsf{ls}(l,S)}^{r}_{I(r,h)}\end{array}$$

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. Consequently:

$$[\textsc{Lock}(k)]^{r}_{1} * [\textsc{Lock}(k)]^{r}_{1} \implies \mathsf{false}$$

This establishes that the axiom holds.

6.2 Hash Table Implementation

We now consider a second index implementation: 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 confusion, we rename the methods of the secondary index to `search′`, `insert′` and `remove′`.) 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) concurrent index module.

The hash table implementations of `search`, `insert` and `remove` are given in Figure 12. This code assumes a pure hashing function `hash(k)` which takes a key `k` and returns an integer between 0 and *max* − 1, where *max* is the size of the hash table array.

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

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

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

Figure 11. Hash table operations.

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

Figure 12. Proof outline for hash-table `search`.

representing the associated secondary index. Picking an arbitrary example, we define the predicate $\mathsf{in}_{\mathsf{rem}}(h, k, v)_i$ as follows:

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

(The definitions of the other 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 specifications. 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.

A sketch-proof for the hash table implementation of `search` is given in Figure 12. Notice that this proof appeals to the specification of `search` when retrieving a value from the appropriate secondary index.

6.3 B^{Link} Tree Implementation

Our final index implementation is a B^{Link} tree algorithm, based on Sagiv [18]. Search operations run 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), 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.

A B^{Link} tree is a balanced search tree. An example is shown in Figure 13. Each node in the tree contains an ordered list of key-value pairs, which at the leaves form the index represented by the tree. Non-leaf nodes map keys to pointers to the node's children. 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.

This structure ensures that every key-value pair stored in the tree can be reached by traversing from the leftmost node at any level of the tree. If the value cannot be found by following a pointer down the tree, it can be found by following the link pointer. This is important because insertion operations can create new nodes that can only be reached via the link pointers until the higher levels of the tree are later repaired by the operation.

The B^{Link} implementations of the index operations are too lengthy to go into in detail here – details can be found in [4]. In verifying the algorithm we discovered two subtle bugs (see end of section for details).

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:

$$\begin{aligned} \mathsf{in}_{\mathsf{def}}(h, k, v)_i \quad \triangleq \quad & \exists r. \boxed{\mathsf{B}_{\in}(h, k, v)}^r_{I(r,h)} * [\textsc{Lock}]^r_{(g,i)} \\ & * [\textsc{Swap}]^r_{(g,i)} * [\textsc{Rem}(0, k)]^r_{(d,i)} \\ & * \circledast_{v \in \mathsf{Vals}} [\textsc{Ins}(0, k, v)]^r_{(d,i)} \end{aligned}$$

The shared assertion $\mathsf{B}_{\in}(h, k, v)$ denotes a B^{Link} tree at address h containing the key-value pair (k, v). It is defined as follows:

$$\begin{aligned} \mathsf{B}_{\in}(h, k, v) \quad \triangleq \quad & \exists D. \mathsf{BLTree}(h, D) * \neg \exists x. \Diamond \mathsf{isNode}(x, l) \\ & \land (k, v) \in D \land \mathsf{Tokens}(h) \end{aligned}$$

The thread-local assertions in $\mathsf{in}_{\mathsf{def}}$ consists of capabilities associated with the current thread. The $[\textsc{Lock}]^r_{(g,i)}$ capability says that the current thread is allowed to lock nodes in the region r. The $[\textsc{Swap}]^r_{(g,i)}$ capability allows the the predicate to be modified to represent different behaviour when $i = 1$ (for example, by converting to $\mathsf{in}_{\mathsf{rem}}$ or unk). The $[\textsc{Rem}(0, k)]^r_{(d,i)}$ capability says that neither the current

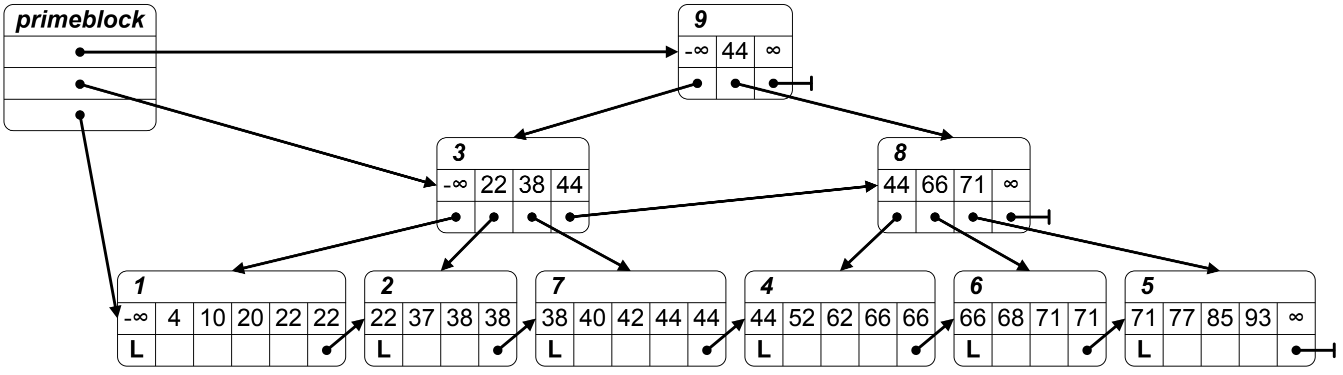

Figure 13. A B^{Link} tree.

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 $[\textsc{Ins}(0, k, v)]^r_{(d,i)}$ capabilities are similar for insertion of a value v into the tree at key k.

In the definition of $\mathsf{in}_{\mathsf{def}}$, notice that capabilities have subscripts which are not values in the interval $[0..1]$. Rather, we have permissions 1, 0, and *two* non-exclusive permissions (d, i) and (g, i) (where $i \in (0..1)$). We call the former a *deny* and the latter a *guarantee*. A deny means that the thread cannot perform the action allowed by the capability, but also that no other thread can perform it either. Conversely, the guarantee means that the thread can perform the action, but so can other threads. Deny-guarantee permissions form a lattice with $(d, 1) = 1 = (g, 1)$, $(d, i) + (d, j) = (d, i + j)$, and $(g, i) + (g, j) = (g, i + j)$. However, there is no relation between (d, i) and a (g, i) for $i \neq 1$. (For further details, see Dodds *et al.* [7].)

We define other index predicates 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. We give the full definitions of the predicates in Appendix A.

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 14 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 give the full interference environment for the B^{Link} tree implementation in Appendix A.

Note that both $[Rem(0, k)]^r_{(g,i)}$ and $[Rem(1, k)]^r_{(g,i)}$ capabilities allow a thread to remove the key k; however, the latter requires the thread to leave a $[Rem(1, k)]^r_{(g,i)}$ 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 $[Rem(1, k)]^r_1$ capability, proving that none of them did so. Thus the $\mathsf{in}_{\mathsf{rem}}(k, v)_1$ can be converted to $\mathsf{in}_{\mathsf{def}}(k, v)$.

Verifying the operations. We give a sketch proof in Figure 15, 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`. Thus, the assertion that the key-value pair $(\mathtt{k}, v)$ is contained in the B^{Link} is stable.

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

Figure 14. Example actions from the B^{Link} tree interference environment.

The proof uses the predicate $\mathsf{niceNode}(N, k, v, r, h)$, defined as follows:

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

(Here leaf and inner are predicates representing leaf and non-leaf nodes respectively. node is defined as their disjunction.)

The definition of niceNode asserts that the node descriptor N contains legitimate information about the tree. If N is a non-leaf (or *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} algorithm. While verifying the algorithm, we discovered a subtle bug in the original presentation [18]. 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 primeblock 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.

```
{$\mathsf{in_{def}}(\mathtt{h}, \mathtt{k}, v)_i$}
search(h, k) {
  {$\boxed{\mathsf{B}_\in(\mathtt{h}, \mathtt{k}, v)}^r_{I(r,\mathtt{h})} * [\textsc{Lock}]^r_{(g,i)} * [\textsc{Swap}]^r_{(g,i)} * [\textsc{Rem}(0, \mathtt{k})]^r_{(d,i)}$
   $* \circledast_{v \in \mathsf{Vals}} [\textsc{Ins}(0, \mathtt{k}, v)]^r_{(d,i)}$}
  PB := getPrimeBlock(h);
  current := root(PB);
  N := get(current);
  {$\boxed{\mathsf{B}_\in(\mathtt{h}, \mathtt{k}, v)}^r_{I(r,\mathtt{h})} * [\textsc{Lock}]^r_{(g,i)} * [\textsc{Swap}]^r_{(g,i)} * [\textsc{Rem}(0, \mathtt{k})]^r_{(d,i)}$
   $* \circledast_{v \in \mathsf{Vals}} [\textsc{Ins}(0, \mathtt{k}, v)]^r_{(d,i)} * \mathsf{niceNode}(\mathtt{N}, \mathtt{k}, v, r, \mathtt{h})$
   $\wedge \mathtt{N} = \mathsf{node}(-, k_0, p, D, k', p') \wedge k_0 = -\infty$}
  while(isLeaf(N) = false) {
    current := next(N, k);
    N := get(current);
  }
  {$\boxed{\mathsf{B}_\in(\mathtt{h}, \mathtt{k}, v)}^r_{I(r,\mathtt{h})} * [\textsc{Lock}]^r_{(g,i)} * [\textsc{Swap}]^r_{(g,i)} * [\textsc{Rem}(0, \mathtt{k})]^r_{(d,i)}$
   $* \circledast_{v \in \mathsf{Vals}} [\textsc{Ins}(0, \mathtt{k}, v)]^r_{(d,i)} * \mathsf{niceNode}(\mathtt{N}, \mathtt{k}, v, r, \mathtt{h})$
   $\wedge \mathtt{N} = \mathsf{leaf}(-, k_0, D, k', p') \wedge k_0 < \mathtt{k}$}
  while(k > highValue(N)) {
    current := next(N, k);
    N := get(current);
  }
  {$\boxed{\mathsf{B}_\in(\mathtt{h}, \mathtt{k}, v)}^r_{I(r,\mathtt{h})} * [\textsc{Lock}]^r_{(g,i)} * [\textsc{Swap}]^r_{(g,i)} * [\textsc{Rem}(0, \mathtt{k})]^r_{(d,i)}$
   $* \circledast_{v \in \mathsf{Vals}} [\textsc{Ins}(0, \mathtt{k}, v)]^r_{(d,i)} * \mathsf{niceNode}(\mathtt{N}, \mathtt{k}, v, r, \mathtt{h})$
   $\wedge \mathtt{N} = \mathsf{leaf}(-, k', D, k'', -) \wedge k' < \mathtt{k} \leq k''$}
  if(isIn(N, k)) {
    {$\boxed{\mathsf{B}_\in(\mathtt{h}, \mathtt{k}, v)}^r_{I(r,\mathtt{h})} * [\textsc{Lock}]^r_{(g,i)} * [\textsc{Swap}]^r_{(g,i)} * [\textsc{Rem}(0, \mathtt{k})]^r_{(d,i)}$
     $* \circledast_{v \in \mathsf{Vals}} [\textsc{Ins}(0, \mathtt{k}, v)]^r_{(d,i)} * \mathsf{niceNode}(\mathtt{N}, \mathtt{k}, v, r, \mathtt{h})$
     $\wedge \mathtt{N} = \mathsf{leaf}(-, k', D, k'', -) \wedge (\mathtt{k}, v) \in D$}
    return( lookup(N, k) );
  } else {
   {$\mathsf{false}$}
    return null;
  }
  {$\boxed{\mathsf{B}_\in(\mathtt{h}, \mathtt{k}, v)}^r_{I(r,\mathtt{h})} * [\textsc{Lock}]^r_{(g,i)} * [\textsc{Swap}]^r_{(g,i)} * [\textsc{Rem}(0, \mathtt{k})]^r_{(d,i)}$
   $* \circledast_{v \in \mathsf{Vals}} [\textsc{Ins}(0, \mathtt{k}, v)]^r_{(d,i)} \wedge ret = v$}
}
{$\mathsf{in_{def}}(\mathtt{h}, \mathtt{k}, v)_i \wedge \mathrm{ret} = v$}
```

Figure 15. Proof outline for B^{Link} tree `search`.

7. Conclusions

We have proposed a simple yet flexible specification for reasoning about concurrent indexes in the manner of concurrent separation logic [14]. We have shown how this specification can be used to verify a range of client applications, ranging from common programming patterns such as memoization and map, to algorithms such as a prime number sieve. These examples demonstrate the utility of our specification.

To demonstrate the relevance of our index specification, we have shown that it is satisfied by three radically different implementations: a simple linked-list, a hash table and Sagiv's complex and highly concurrent B^{Link} tree. We used concurrent abstract predicates (CAP) [6] to support highly-disjoint abstractions for implementations that involve a great deal of sharing under the hood. Any approach to reasoning about concurrent programs in a compositional fashion will naturally require some form of abstraction; our work validates the CAP approach to the problem.

Relationship to linearizability. Linearizability [9] is the current de-facto correctness criterion for concurrent algorithms. It requires that the methods of concurrent objects behave as atomic operations, thus providing a proof technique for observational refinement [8]. We could employ linearizability, or other atomicity refinement techniques such as [20], 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. Our approach could therefore be seen as an alternative correctness criterion.

Future work. Our correctness proof for the B^{Link} tree implementation is at the limit of what can be achieved by hand. We found a bug in Sagiv's presentation, but our proof is so complex that it would be hubristic to claim to have made no mistakes ourselves. In order establish certainty and to scale our approach to real-world applications we plan to develop tools for automatically checking and generating proofs.

Tools based on separation logic can now verify hundreds of thousands of lines of sequential code [3]. In contrast, verification tools for concurrency have, up to now, lacked scalability, in part due to a lack of modularly in the underlying reasoning. We have demonstrated that our approach supports strongly modular reasoning, in the sense that implementation details are completely hidden from the client's view. This abstraction mechanism offers the possibility of building truly scalable tools for verifying concurrent programs.

Acknowledgments

Special thanks to Adam Wright for substantial contributions to examples of iteration (§5) and for discussions and feedback. Thanks also to Richard Bornat, Cliff Jones, Daiva Naudžiūnienė, Gian Ntzik, Matthew Parkinson, Noam Rinetzky, Marc Shapiro, Viktor Vafeiadis and John Wickerson for useful 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., DISTEFANO, D., O'HEARN, P., AND YANG, H. Compositional shape analysis by means of bi-abduction. In *POPL* (2009).

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

[5] DILLIG, I., DILLIG, T., AND AIKEN, A. Precise reasoning for programs using containers. *SIGPLAN Not. 46* (January 2011), 187–200.

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

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

[8] FILIPOVIC, I., O'HEARN, P., RINETZKY, N., AND YANG, H. Abstraction for concurrent objects. In *ESOP* (2010), pp. 4379 – 4398. ESOP.

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

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

[11] KRISHNASWAMI, N. R. Reasoning about iterators with separation logic. In *Proceedings of the 2006 conference on Specification and verification of component-based systems* (New York, NY, USA, 2006), SAVCBS '06, ACM, pp. 83–86.

[12] 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.

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

[14] OHEARN, P. W. Resources, concurrency, and local reasoning. *Theor. Comput. Sci. 375* (April 2007), 271–307.

[15] PARKINSON, M., AND BIERMAN, G. Separation logic and abstraction. In *POPL* (New York, NY, USA, 2005), POPL '05, ACM, pp. 247–258.

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

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

[18] SAGIV, Y. Concurrent operations on B*-trees with overtaking. *Journal of Computer and System Sciences 33* (October 1986), 275–296.

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

[20] TURON, A. J., AND WAND, M. A separation logic for refining concurrent objects. In *POPL* (New York, NY, USA, 2011), ACM, pp. 247–258.

[21] VAFEIADIS, V., AND PARKINSON, M. A marriage of rely/guarantee and separation logic. *CONCUR* (2007), 256–271.

[22] XIANG, P., HOU, R., AND ZHOU, Z. Cache and consistency in NOSQL. In *ICCSIT* (2010), vol. 6, IEEE, pp. 117–120.

A. B$^{\mathit{Link}}$ Tree Implementation Details

In this appendix we provide an in depth discussion of our B$^{\mathit{Link}}$ tree index implementation, based on Sagiv's BTree algorithms, and how this implementation satisfies our abstract specification. We give concrete interpretations to each of our abstract predicates and define the interference environment for the B$^{\mathit{Link}}$ tree. Together these allow us to prove the correctness of our implementation.

B^{Link} Tree Data Structure

To begin the verification of our B$^{\mathit{Link}}$ tree implementation, we first define a series of predicates representing the concrete B$^{\mathit{Link}}$ tree data structure. There are two types of node in a B$^{\mathit{Link}}$ tree: *leaf* nodes and *inner* nodes. Leaf nodes are at the fringe of the structure and contain the key-value pairs from the abstract interface. Inner nodes make up the rest of the tree and contain key-pointer pairs that provide the search structure of the tree. We assume two basic predicates for representing these nodes in the tree: a leaf predicate, and an inner predicate.

$$x \mapsto \mathsf{leaf}(l, k_0, D, k', p') \quad x \mapsto \mathsf{inner}(l, k_0, p, D, k', p')$$

Here, x is the address of the node. The value l is the node's lock. If the node is unlocked then $l = 0$ and if the node is locked then $l = 1$. The ordered list D contains the key-value pairs (k, v) represented by the node. In each node the list D may contain up to $2K$ key-value pairs for some fixed constant K given by the implementation (K is often chosen so that a node fills a single page in memory). The values k_0 and k' are the lower and upper bound, respectively, on the keys contained in this list. So, for every key-value pair $(k, v) \in D$ we have $k_0 < k \leq k'$. The pointer p (only present in an inner node) points to the subtree which contains all of the keys which are greater than the minimum value of this node. The pointer p', known as the link pointer, points to the node's right sibling, if it exists.

We define some additional notation for handling lists. We require a notion of iterated concatenation which we denote

$$\bigcup_{i=1}^{n} D_i = D_1 :: D_2 :: ...D_n.$$

We also require an insertion operation $D \uplus (k, v)$ which adds the key-value pair (k, v) to the ordered list D in the correct place,

$$D \uplus (k, v) = D_1 :: (k, v) :: D_2$$

where $D = D_1 :: D_2$ and $D_1 = D_1' :: (k_1, v_1)$ and $D_2 = (k_2, v_2) :: D_2'$ and $k_1 < k < k_2$ (undefined otherwise).

A B$^{\mathit{Link}}$ tree is a superimposed structure made up of both a tree and several layers of linked lists. At the leaf level the linked list contains pointers to data entries, while at other levels the linked list contains pointers to nodes deeper in the tree structure. These linked lists always have at least one element, the first node has minimum value $-\infty$ and the last node has maximum value $+\infty$. Each node in these linked lists is disjoint, so we can use a separation logic predicate to define this structure precisely. Given ordered key-value list T and D, which contain the key-value pairs that point into the current level of the tree and the key-value pairs contained in the current level of the tree respectively, we can define the linked list structure for a layer of the B*Link* tree. Let $T = [(k_1, v_1), ..., (k_n, v_n)]$ then,

$$\begin{aligned}\mathsf{leafList}(T, D) \triangleq\ & \exists D_1, ..., D_n. \\ & \circledast_{i=1}^{n-1} v_i \mapsto \mathsf{leaf}(-, k_i, D_i, k_{i+1}, v_{i+1}) \\ & * v_n \mapsto \mathsf{leaf}(-, k_n, D_n, +\infty, \mathsf{nil}) \\ & \wedge k_1 = -\infty \wedge n > 0 \wedge D = \textstyle\bigcup_{i=1}^{n} D_i\end{aligned}$$

$$\begin{aligned}\mathsf{innerList}(T, D) \triangleq\ & \exists D_1, ..., D_n. \\ & \circledast_{i=1}^{n-1} v_i \mapsto \mathsf{inner}(-, k_i, v'_i, D_i, k_{i+1}, v_{i+1}) \\ & * v_n \mapsto \mathsf{inner}(-, k_n, v'_n, D_n, +\infty, \mathsf{nil}) \\ & \wedge k_1 = -\infty \wedge n > 0 \\ & \wedge D = \textstyle\bigcup_{i=1}^{n} (k_i, v'_i) \cup D_i\end{aligned}$$

We choose not to define the tree structure directly, as at some points in time the tree structure of the B*Link* tree can actually be broken by the `insert` operation. When the `insert` operation creates a new node in the tree, it is added to the linked list structure before it is given a reference in the layer above. If the `search` operation did not use the link pointers as well as the tree pointers, it would not be able to find this new node at this point in time. To capture this behaviour we instead choose to build up our tree predicate by layering our lists on top of one another. Using our linked list predicates, we can build up a predicate for the tree-like structure of the B*Link* tree.

$$\begin{aligned}\mathsf{Btree}_1(PB, x :: T, D) \triangleq\ & \exists p.\, \mathsf{leafList}(x :: T, D) \\ & \wedge x = (-\infty, p) \wedge PB = [p]\end{aligned}$$

$$\begin{aligned}\mathsf{Btree}_{n+1}(p :: PB, x :: T, D) \triangleq\ & \exists L, L'.\, \mathsf{innerList}(x :: T, L) \\ & * \mathsf{Btree}_n(PB, L', D) \\ & \wedge x = (-\infty, p) \wedge L \subseteq L'\end{aligned}$$

The prime block PB contains a list of pointers to the leftmost node at each level of the tree. The key-value list D is the concatenation of all key-value pairs at the fringe of the tree and corresponds to our abstract index view of the B*Link* tree structure.

Finally, using these predicates, we can now define a predicate for the complete B*Link* tree structure.

$$\mathsf{BLTree}(h, D) \triangleq \exists PB, n, T.\, \mathsf{Btree}_n(PB, T, D) * h \mapsto PB$$

This describes a B*Link* tree whose prime block is stored at address h and contains a set of key-value pairs D. Figure 13 shows an example of a B*Link* tree. The fringe of the tree forms a leafList that contains all of the key-value pairs mapped to by the index. Each of the other layers of the tree forms an innerList that makes up the search structure of the tree. Each list has minimum value $-\infty$ and maximum value ∞ and the primeblock points to the head of each layer's list.

Interpretation of Abstract Predicates

Now that we have a predicate describing a B*Link* tree, we can turn our attention to providing concrete interpretations of our abstract predicates. In § 6.3 we introduced the interpretation of the $\mathsf{in_{def}}(\mathsf{h}, \mathsf{k}, \mathsf{v})$ predicate. Here we go into more detail about the auxiliary predicates we used in our interpretations, and then provide the concrete interpretations of the full abstract specification.

First we define a number of predicates which will come in useful for our later definitions:

$$\begin{aligned}\Diamond P \ &\triangleq\ \mathsf{true} * P \\ \mathsf{isNode}(x, l) \ &\triangleq\ \exists k_0, p, D, k', p'. \\ & \quad\ \ x \mapsto \mathsf{node}(l, k_0, p, D, k', p') \\ \mathsf{locked}(x) \ &\triangleq\ \mathsf{isNode}(x, 1) \\ \mathsf{unlocked}(x) \ &\triangleq\ \mathsf{isNode}(x, 0) \\ \mathsf{child}(h, x) \ &\triangleq\ \exists l.\, \mathsf{isNode}(x, l) \\ & \quad\ \ \wedge \exists p, ps.\, \Diamond h \mapsto p : ps \\ & \quad\ \ \wedge x = p \\ & \quad\ \ \vee p \mapsto \mathsf{node}(-, -, -, -, -, x) \\ & \quad\ \ \vee \\ & \quad\ \ \exists y, k_0, v_0, D, k. \\ & \quad\ \ \Diamond y \mapsto \mathsf{inner}(-, k_0, v_0, D, -, -) \\ & \quad\ \ \wedge (k, x) \in (k_0, v_0) :: D \\ \mathsf{orphan}(h, x) \ &\triangleq\ \neg \mathsf{child}(h, x) \\ \mathsf{dualRoot}(h, x, y) \ &\triangleq\ \exists p, D, k, p', D'.\, h \mapsto x : xs \\ & \quad\ \ * x \mapsto \mathsf{node}(1, -\infty, p, D, k, y) \\ & \quad\ \ * y \mapsto \mathsf{node}(1, k, p', D', \infty, \mathsf{nil}) \\ \mathsf{allMods}(x) \ &\triangleq\ \forall l, t, k, v, y, i.\, \mathsf{isNode}(x, l) \\ & \quad\ \ \wedge \Diamond[\textsc{ModLR}(t, x, k, i)]_1^r \\ & \quad\ \ \wedge \Diamond[\textsc{ModLI}(t, x, k, v, i)]_1^r \\ & \quad\ \ \wedge \Diamond[\textsc{Fix}(k, x)]_1^r \\ & \quad\ \ \wedge \Diamond[\textsc{ModII}(x, k, y)]_1^r \\ & \quad\ \ \wedge \Diamond[\textsc{NewR}(x, k, y)]_1^r\end{aligned}$$

Informally, these predicates have the following meanings:

- $\Diamond P$ describes a heap where P is satisfied somewhere in the heap.
- $\mathsf{isNode}(x, l)$ describes a node x in the B*Link* tree with lock value l.
- $\mathsf{locked}(x)$ describes a locked node x in the B*Link* tree.
- $\mathsf{unlocked}(x)$ describes an unlocked node x in the B*Link* tree.
- $\mathsf{child}(h, x)$ describes a node x in the B*Link* tree at address h which is either at the root level, or has a parent in the tree's search structures; some node in the tree contains a key-value pair $(-, x)$.
- $\mathsf{orphan}(h, x)$ describes a node x in the B*Link* tree at address h which does not have a parent in the tree's search structure; it is not at the root and no node contains a key-value pair $(-, x)$.

- $\mathsf{dualRoot}(h, x, y)$ describes a B^{Link} tree at address h that currently has two nodes at its root level (so an insert operation has just split the root and is about to create a new one).
- $\mathsf{allMods}(x)$ describes the set of all modification capabilities, with excusive permission, for node x.

As we saw in § 6.3, our concrete interpretations describe the shared state with one of the following assertions:

$$\begin{aligned}\mathsf{B}_{\in}(h,k,v) &\triangleq \exists D.\, \mathsf{BLTree}(h,D) * \neg\exists x,l.\, \Diamond\mathsf{isNode}(x,l)\\ &\quad \wedge (k,v)\in D \wedge \mathsf{Tokens}(h)\\ \mathsf{B}_{\notin}(h,k) &\triangleq \exists D.\, \mathsf{BLTree}(h,D) * \neg\exists x,l.\, \Diamond\mathsf{isNode}(x,l)\\ &\quad \wedge k\notin \mathsf{keys}(D) \wedge \mathsf{Tokens}(h)\end{aligned}$$

The assertion $\mathsf{B}_{\in}(h,k,v)$ describes a B^{Link} tree at address h that contains the key-value pair (k,v). Similarly the assertion $\mathsf{B}_{\notin}(h,k,v)$ describes a B^{Link} tree at address h where the key k is unassigned. However, both assertions also describe an additional part of the shared state. The assertion $\neg\exists x,l.\, \Diamond\mathsf{isNode}(x,l)$ ensures that there are no nodes in this additional state; it consists only of capabilities. The assertion $\mathsf{Tokens}(h)$ ensures that these capabilities are consistent with the current state of the B^{Link} tree at address h.

The $\mathsf{Tokens}(h)$ predicate is quite complex and is defined in Figure 16. The predicate describes the capabilities that are in the shared state on a capability by capability basis dependent on the current state of the B^{Link} tree. The predicate is built up of the conjunction of a number of disjuncts.

The first disjunct describes if a node's $[\textsc{Unlock}(x)]^r_1$ capability is present in the shared state. If x is not a node, or if x is an unlocked node then the UNLOCK capability must be present in the shared state. If x is a locked node then this capability may be missing from the shared state. However, it is also possible that the thread that has locked the node may have acquired a MOD capability for that node, that is it is about to make some change to the node. In this case the UNLOCK capability will be present in the shared state, but so will some REM or INS capability. This may appear to allow some other thread to acquire the UNLOCK capability for this node, but recall that the node is still locked. We shall see later, when we define the interference environment, that a thread may only acquire a nodes UNLOCK capability if that node is unlocked and in doing so the thread locks the node.

The second and third disjuncts describe if we are in an action tracking state or not. If $t = 0$, then we are not tracking the actions on this key (we are in a def or unk environment) and all of the REM and INS capabilities for $t = 1$ must be in the shared state. If $t = 1$, then we are tracking the actions on this key (we are in an ins or rem environment) and all of the REM and INS capabilities for $t = 0$ must be in the shared state. We shall see later, when we define the interference environment, that when we are tracking the actions on a key, threads leave behind some fraction of their REM or INS capabilities after performing a modification action. This allows us to track if a value has been inserted or removed from a given key value and return to a def state.

The fourth and final disjunct describes which of the modification capabilities are present in the shared state for each node in the B^{Link} tree. It is always the case that either all of the modification capabilities are in the shared state, or one such capability is missing. If one of the modification capabilities is missing then the node must be locked and the locking thread must have placed the UNLOCK capability and some other capability, describing the action it is about to perform on that node (e.g. REM or INS). This represents a thread that has locked the node and is about to make some update to that node. Due to the locking, it is only ever possible for at most one thread to be in this state, hence why at most one modification capability is ever missing for any given node.

We define the concrete interpretations of our abstract predicates in Figure 17. Each case describes the current state of the shared B^{Link} tree, as well as which capabilities are known to be in the shared and thread local state. For example, the definition of $\mathsf{in}_{\mathsf{def}}(h,k,v)_i$ states that the key-value pair (k,v) must be stored in the tree. Notice that this definition also gives the thread deny permission (d,i) on all REM and INS capabilities for k. When $i \in (0..1)$ no thread is able to modify the value of k in the tree, and when $i = 1$ only the current thread may modify the value of k in the tree, so this assertion is self-stable. The thread also has the $[\textsc{Lock}]^r_{(g,i)}$ capability, which allows it to lock nodes in the tree, and the $[\textsc{Swap}]^r_{(g,i)}$ capability, which allows it to change between tracking actions or not (by swapping $t = 0$ and $t = 1$ capabilities).

Some of the other definitions make more complicated assertions about the shared state. Take, for example, the definition of the $\mathsf{in}_{\mathsf{rem}}(h,k,v)_i$ predicate. Recall from our abstract specification that this predicate states that key k was assigned value v, but that any thread can remove this value. We track which actions have occurred so far by using the $t = 1$ capabilities. If a thread removes the value for the key, then it must leave some $[\textsc{Rem}(1,k)]^r_{(g,i')}$ capability in the shared state. The uncertainty about the current assignment of k is represented by the disjunction in the shared state. In the first case no thread has yet removed the key from the tree, since there is no REM capability for that k in the shared state. In the second case some thread has just acquired the modification capability $[\textsc{ModLR}(1,x,k,i')]^r_1$ allowing it to remove the key from the tree, but it has yet to perform this action, so the key is still currently assigned. In the last case some thread has removed the key from the tree and left part of its REM capability in the shared state to signify this.

The other predicates are defined in similar ways.

We can now verify that our interpretations satisfy the axioms from §4 for our abstract specification. For example we can verify,

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

$$
\begin{array}{l}
\mathsf{Tokens}(h) \triangleq \\
\quad \forall x.\ \left(\begin{array}{l} \neg\exists l.\ \Diamond\mathsf{isNode}(x,l) \\ \ \wedge \Diamond[\textsc{Unlock}(x)]^r_1 \end{array}\right) \vee \left(\begin{array}{l} \Diamond\mathsf{unlocked}(x) \\ \ \wedge \Diamond[\textsc{Unlock}(x)]^r_1 \end{array}\right) \vee \left(\begin{array}{l} \Diamond\mathsf{locked}(x) \\ \ \wedge \neg\Diamond[\textsc{Unlock}(x)]^r_1 \end{array}\right) \\
\qquad\qquad \vee \left(\begin{array}{l} \Diamond\mathsf{locked}(x) \wedge \Diamond[\textsc{Unlock}(x)]^r_1 \\ \ \wedge \exists k,v,i,t.\ \left(\begin{array}{l} [\textsc{Rem}(t,k)]^r_i \\ \ \wedge \neg\Diamond[\textsc{ModLR}(t,x,k,i)]^r_1 \end{array}\right) \vee \left(\begin{array}{l} [\textsc{Ins}(t,k,v)]^r_i \\ \ \wedge \neg\Diamond[\textsc{ModLI}(t,x,k,v,i)]^r_1 \end{array}\right) \end{array}\right) \\
\qquad \wedge \\
\quad \forall k.\ \Diamond[\textsc{Rem}(0,k)]^r_1 \vee \Diamond[\textsc{Rem}(1,k)]^r_1 \\
\qquad \wedge \\
\quad \forall k,v.\ \Diamond[\textsc{Ins}(0,k,v)]^r_1 \vee \Diamond[\textsc{Ins}(1,k,v)]^r_1 \\
\qquad \wedge \\
\quad \forall x.\ \mathsf{allMods}(x) \vee \exists t,k,i.\ \Diamond[\textsc{Rem}(t,k)]^r_i \wedge \Diamond[\textsc{Unlock}(x)]^r_1 \wedge ([\textsc{ModLR}(t,x,k,i)]^r_1 \mathrel{-\!\!*} \mathsf{allMods}(x)) \\
\qquad\qquad\qquad\qquad \vee \exists t,k,v,i.\ \Diamond[\textsc{Ins}(t,k,v)]^r_i \wedge \Diamond[\textsc{Unlock}(x)]^r_1 \wedge ([\textsc{ModLI}(t,x,k,v,i)]^r_1 \mathrel{-\!\!*} \mathsf{allMods}(x)) \\
\qquad\qquad\qquad\qquad \vee \exists k,y.\ \mathsf{orphan}(h,x) \wedge \Diamond[\textsc{ModII}(y,k,x)]^r_1 \wedge ([\textsc{Fix}(k,x)]^r_1 \mathrel{-\!\!*} \mathsf{allMods}(x)) \\
\qquad\qquad\qquad\qquad \vee \exists k,y.\ \mathsf{orphan}(h,y) \wedge \Diamond[\textsc{Fix}(k,y)]^r_1 \wedge [\textsc{Unlock}(x)]^r_1 \wedge ([\textsc{ModII}(x,k,y)]^r_1 \mathrel{-\!\!*} \mathsf{allMods}(x)) \\
\qquad\qquad\qquad\qquad \vee \exists k,y.\ \mathsf{dualRoot}(h,x,y) \wedge \Diamond[\textsc{Unlock}(x)]^r_1 \wedge \Diamond[\textsc{Unlock}(y)]^r_1 \wedge ([\textsc{NewR}(x,k,y)]^r_1 \mathrel{-\!\!*} \mathsf{allMods}(x))
\end{array}
$$

Figure 16. Definition of the $\mathsf{Tokens}(h)$ predicate.

$$
\mathsf{in}_{\mathsf{def}}(h,k,v)_i \triangleq \exists r.\ \boxed{\mathsf{B}_{\in}(h,k,v)}^{r}_{I(r,h)} * [\textsc{Lock}]^r_{(g,i)} * [\textsc{Swap}]^r_{(g,i)} * [\textsc{Rem}(0,k)]^r_{(d,i)} * \circledast_{v\in\mathsf{Vals}}[\textsc{Ins}(0,k,v)]^r_{(d,i)}
$$

$$
\mathsf{out}_{\mathsf{def}}(h,k)_i \triangleq \exists r.\ \boxed{\mathsf{B}_{\notin}(h,k)}^{r}_{I(r,h)} * [\textsc{Lock}]^r_{(g,i)} * [\textsc{Swap}]^r_{(g,i)} * [\textsc{Rem}(0,k)]^r_{(d,i)} * \circledast_{v\in\mathsf{Vals}}[\textsc{Ins}(0,k,v)]^r_{(d,i)}
$$

$$
\begin{aligned}
\mathsf{in}_{\mathsf{ins}}(h,k,S)_i \triangleq{}& \exists v\in S, r, i', i''.\ \boxed{\mathsf{B}_{\in}(h,k,v) \wedge \Diamond[\textsc{Ins}(1,k,v)]^r_{i''}}^{r}_{I(r,h)} * [\textsc{Lock}]^r_{(g,i)} * [\textsc{Swap}]^r_{(g,i)} * [\textsc{Rem}(1,k)]^r_{(d,i)} \\
& * \circledast_{v\in S}[\textsc{Ins}(1,k,v)]^r_{(g,i')} * \circledast_{v\notin S}[\textsc{Ins}(1,k,v)]^r_{(d,i)} \wedge (i' = i \vee i' + i'' = i)
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{out}_{\mathsf{ins}}(h,k,S)_i \triangleq{}& \exists v\in S, r, i'.\ \boxed{\begin{array}{l} \mathsf{B}_{\notin}(h,k) \wedge \neg\Diamond[\textsc{Ins}(1,k,v)]^r_{(g,i')} \\ \ \vee \mathsf{B}_{\notin}(h,k) \wedge \Diamond[\textsc{Ins}(1,k,v)]^r_{(g,i')} \wedge \Diamond[\textsc{Unlock}(x)]^r_1 \wedge \neg\Diamond[\textsc{ModLI}(1,x,k,v,i')]^r_1 \\ \ \vee \mathsf{B}_{\in}(h,k,v) \wedge \Diamond[\textsc{Ins}(1,k,v)]^r_{(g,i')} \wedge \Diamond[\textsc{ModLI}(1,x,k,v,i')]^r_1 \end{array}}^{r}_{I(r,h)} \\
& * [\textsc{Lock}]^r_{(g,i)} * [\textsc{Swap}]^r_{(g,i)} * [\textsc{Rem}(1,k)]^r_{(d,i)} * \circledast_{v\in S}[\textsc{Ins}(1,k,v)]^r_{(g,i)} \\
& * \circledast_{v\notin S}[\textsc{Ins}(1,k,v)]^r_{(d,i)} \wedge i' > 0
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{in}_{\mathsf{rem}}(h,k,v)_i \triangleq{}& \exists r, i'.\ \boxed{\begin{array}{l} \mathsf{B}_{\in}(h,k,v) \wedge \neg\Diamond[\textsc{Rem}(1,k)]^r_{(g,i')} \\ \ \vee \mathsf{B}_{\in}(h,k,v) \wedge \Diamond[\textsc{Rem}(1,k)]^r_{(g,i')} \wedge \Diamond[\textsc{Unlock}(x)]^r_1 \wedge \neg\Diamond[\textsc{ModLR}(1,x,k,i')]^r_1 \\ \ \vee \mathsf{B}_{\notin}(h,k) \wedge \Diamond[\textsc{Rem}(1,k)]^r_{(g,i')} \wedge \Diamond[\textsc{ModLR}(1,x,k,i')]^r_1 \end{array}}^{r}_{I(r,h)} \\
& * [\textsc{Lock}]^r_{(g,i)} * [\textsc{Swap}]^r_{(g,i)} * [\textsc{Rem}(1,k)]^r_{(g,i)} * \circledast_{v\in\mathsf{Vals}}[\textsc{Ins}(1,k,v)]^r_{(d,i)} \wedge i' > 0
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{out}_{\mathsf{rem}}(h,k)_i \triangleq{}& \exists r, i', i''.\ \boxed{\mathsf{B}_{\notin}(h,k) \wedge \Diamond[\textsc{Rem}(1,k)]^r_{(g,i'')}}^{r}_{I(r,h)} * [\textsc{Lock}]^r_{(g,i)} * [\textsc{Swap}]^r_{(g,i)} * [\textsc{Rem}(1,k)]^r_{(g,i')} \\
& * \circledast_{v\in\mathsf{Vals}}[\textsc{Ins}(1,k,v)]^r_{(d,i)} \wedge (i' = i \vee i' + i'' = i)
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{unk}(h,k,S)_i \triangleq{}& \exists v\in S, r.\ \boxed{\mathsf{B}_{\in}(h,k,v) \vee \mathsf{B}_{\notin}(h,k)}^{r}_{I(r,h)} * [\textsc{Lock}]^r_{(g,i)} * [\textsc{Swap}]^r_{(g,i)} * [\textsc{Rem}(0,k)]^r_{(g,i)} \\
& * \circledast_{v\in S}[\textsc{Ins}(0,k,v)]^r_{(g,i)} * \circledast_{v\notin S}[\textsc{Ins}(0,k,v)]^r_{(d,i)}
\end{aligned}
$$

$$
\mathsf{read}(h,k) \triangleq \exists v, r.\ \boxed{\mathsf{B}_{\in}(h,k,v) \vee \mathsf{B}_{\notin}(h,k)}^{r}_{I(r,h)}
$$

Figure 17. Concrete predicate interpretations for the B*Link* tree implementation.

since the assertion on the shared state from the $\mathsf{out}_{\mathsf{rem}}$ predicate collapses the disjunction in the shared state from the $\mathsf{in}_{\mathsf{rem}}$ predicate into just one matching case, and the thread local capabilities sum together as expected.

Describing Interference

We model the possible interference on the shared state by an interference environment $I(r, h)$. The interference environment is made up of a set of actions that can be performed by the current thread, and other threads, so long as they posses sufficient resources and capabilities for the actions.

First, we introduce some additional predicates which will help us describe the actions in our interference environment. We have a node predicate for when we want to talk about a node of arbitrary type (a leaf node or an inner node).

$$\begin{aligned} x \mapsto \mathsf{node}(l, k_0, p, D, k', p') \triangleq{} & \\ & (p = \mathsf{nil} \wedge x \mapsto \mathsf{leaf}(l, k_0, D, k', p')) \\ & \vee (p \neq \mathsf{nil} \wedge x \mapsto \mathsf{inner}(l, k_0, p, D, k', p')) \end{aligned}$$

We also have a root predicate which describes if a node is the root of the B$^{\mathit{Link}}$ tree or not.

$$\mathsf{root}(h, x) \triangleq \exists xs.\, \Diamond h \mapsto \mathit{PB} \wedge \mathit{PB} = x :: xs$$

When the insertion operations tries to split a node (when adding a pair to full node) it is important to know if that node is the root or not. If the root is split, then a new root needs to be created and the prime block updated accordingly.

Finally, when describing the insertion action for inner nodes we require a notion of a list of nodes up to some point.

$$\begin{aligned} \mathsf{nodeList}(p, N, p') \triangleq{} & (N = [\,] \wedge p = p' \wedge \mathsf{emp}) \\ & \vee \left(\begin{array}{l} \exists l, k_0, p_0, D, k_1, p_1, N'. \\ N = (l, k_0, p_0, D, k_1, p_1) :: N' \\ \wedge\, p \mapsto \mathsf{node}(l, k_0, p_0, D, k_1, p_1) \\ *\, \mathsf{nodeList}(p_1, N', p') \end{array} \right) \end{aligned}$$

The actions that make up the interference environment for the B$^{\mathit{Link}}$ tree implementation are given in Figure 18. The $\textsc{Lock}$ and $\textsc{Unlock}(x)$ actions control the locking and unlocking of nodes in the tree. The $\textsc{Ins}(t, k, v)$, $\textsc{Rem}(t, k)$ and $\textsc{Fix}(k, y)$ actions allow a thread to gain the modification tokens for a node that they have locked. The $\textsc{Swap}$ action allows a thread with full permission for some key change if we are tracking actions for that key. The $\textsc{ModLI}(t, x, k, v, i)$ action allows a thread to insert a key-value pair (k, v) into some leaf node x. If this node was full, then the thread is given the $[\textsc{Fix}(k, y)]^r_1$ capability so that it may repair the search structure of the tree. The $\textsc{ModLR}(t, x, k, i)$ action allows a thread to remove a key-value pair $(k, -)$ from some leaf node x. Notice that there is no way for a thread to remove key-value pairs from inner nodes. The $\textsc{ModII}(x, k, y)$ action allows a thread to insert a key-value pair (k, y) into some inner node x. This action is used to repair the search structure of the tree after a node has been split. The $\textsc{NewR}(x, k, y)$ action allows a thread to create a new root, and update the prime block accordingly, after the old root has been split. This action can only be used if the thread has previously split the old root and thus acquired the $[\textsc{New}(x, k, y)]^r_1$ capability.

Verifying the Operations

Our B$^{\mathit{Link}}$ tree implementation uses a language which includes a set of heap update commands, which directly modify nodes in the shared heap, and a set of store update commands, which work with nodes but do not manipulate the shared state. We assume that variables in a thread's local store can contain integer, pointer, Boolean, stack and node content information.

The heap update commands are:

```
lock(x)
unlock(x)
x := new()
N := get(x)
put(N, x)
PB := getPrimeBlock(h)
putPrimeBlock(h, PB)
```

Since these commands update the shared state, it is necessary that they each behave atomically so that they do not interfere with one another.

The store update commands are:

```
k := lowValue(N)
k := highValue(N)
x := next(N, k)
x := lookup(N, k)
addPair(N, k, v)
removePair(N, k)
M := rearrange(N, k, v, x)
x := root(PB)
N := newRoot(k', p, k, v, k'')
addRoot(PB, x)
x := getNodeLevel(PB, i)

b := isSafe(N)
b := isIn(N, k)
b := isLeaf(N)
b := isRoot(PB, x)

stack := newStack()
push(stack, x)
x := pop(stack)
b := isEmpty(stack)
```

The store update commands only modify the local store of a thread, so it is not necessary for these commands to be atomic.

We assume that these commands satisfy the specifications given in Figure 19 and Figure 20. Our proof of the `search` operation given in §6.3 Figure 15 then follows. The other

$$\textsc{Lock}: \quad x \mapsto \mathsf{node}(0, k_0, p, D, k', p') * [\textsc{Unlock}(x)]^r_1 \quad \rightsquigarrow \quad x \mapsto \mathsf{node}(1, k_0, p, D, k', p')$$

$$\textsc{Unlock}(x): \quad x \mapsto \mathsf{node}(1, k_0, p, D, k', p') \quad \rightsquigarrow \quad x \mapsto \mathsf{node}(0, k_0, p, D, k', p') * [\textsc{Unlock}(x)]^r_1$$

$$\textsc{Ins}(t, k, v): \quad [\textsc{ModLI}(t, x, k, v, i)]^r_1 \quad \rightsquigarrow \quad [\textsc{Ins}(t, k, v)]^r_{(g,i)} * [\textsc{Unlock}(x)]^r_1$$

$$\textsc{Rem}(t, k): \quad [\textsc{ModLR}(t, x, k, i)]^r_1 \quad \rightsquigarrow \quad [\textsc{Rem}(t, k)]^r_{(g,i)} * [\textsc{Unlock}(x)]^r_1$$

$$\textsc{Fix}(k, y): \quad [\textsc{ModII}(x, k, y)]^r_1 \quad \rightsquigarrow \quad [\textsc{Fix}(k, y)]^r_1 * [\textsc{Unlock}(x)]^r_1$$

$$\textsc{Swap}: \begin{cases} [\textsc{Rem}(1, k)]^r_1 * \circledast_{v \in \mathsf{Vals}}[\textsc{Ins}(1, k, v)]^r_1 \quad \rightsquigarrow \quad [\textsc{Rem}(0, k)]^r_1 * \circledast_{v \in \mathsf{Vals}}[\textsc{Ins}(0, k, v)]^r_1 \\ \left(\begin{array}{l} [\textsc{Rem}(0, k)]^r_1 * [\textsc{Rem}(1, k)]^r_i \\ * \circledast_{v \in \mathsf{Vals}}[\textsc{Ins}(0, k, v)]^r_1 \\ * \circledast_{v \in \mathsf{Vals}}[\textsc{Ins}(1, k, v)]^r_{i_v} \end{array}\right) \quad \rightsquigarrow \quad [\textsc{Rem}(1, k)]^r_1 * \circledast_{v \in \mathsf{Vals}}[\textsc{Ins}(1, k, v)]^r_1 \end{cases}$$

$$\textsc{ModLI}(t, x, k, v, i): \begin{cases} \left(\begin{array}{l} x \mapsto \mathsf{leaf}(1, k_0, D, k', p') * [\textsc{Unlock}(x)]^r_1 \\ * \Big([\textsc{Ins}(t, k, v)]^r_{(g,i)} \wedge t = 0 \vee \mathsf{emp} \wedge t = 1\Big) \end{array}\right) \quad \rightsquigarrow \quad \left(\begin{array}{l} x \mapsto \mathsf{leaf}(1, k_0, D', k', p') \\ * [\textsc{ModLI}(t, x, k, v, i)]^r_1 \\ \wedge D' = D \uplus (k, v) \end{array}\right) \\ \left(\begin{array}{l} x \mapsto \mathsf{leaf}(1, k_0, D, k', p') * [\textsc{Unlock}(x)]^r_1 \\ * \Big([\textsc{Ins}(t, k, v)]^r_{(g,i)} \wedge t = 0 \vee \mathsf{emp} \wedge t = 1\Big) \\ * [\textsc{Fix}(k, y)]^r_1 \wedge |D| = 2K \wedge \neg\mathsf{root}(h, x) \end{array}\right) \quad \rightsquigarrow \quad \left(\begin{array}{l} x \mapsto \mathsf{leaf}(1, k_0, D_1, k_1, y) \\ * y \mapsto \mathsf{leaf}(0, k_1, D_2, k', p') \\ * [\textsc{ModLI}(t, x, k, v, i)]^r_1 \\ \wedge D_1 :: D_2 = D \uplus (k, v) \end{array}\right) \\ \left(\begin{array}{l} x \mapsto \mathsf{leaf}(1, k_0, D, k', p') \\ * [\textsc{NewR}(x, k, y)]^r_1 \\ * \Big([\textsc{Ins}(t, k, v)]^r_{(g,i)} \wedge t = 0 \vee \mathsf{emp} \wedge t = 1\Big) \\ \wedge |D| = 2K \wedge \mathsf{root}(h, x) \end{array}\right) \quad \rightsquigarrow \quad \left(\begin{array}{l} x \mapsto \mathsf{leaf}(1, k_0, D_1, k_1, y) \\ * y \mapsto \mathsf{leaf}(1, k_1, D_2, k', p') \\ * [\textsc{ModLI}(t, x, k, v, i)]^r_1 \\ \wedge D_1 :: D_2 = D \uplus (k, v) \end{array}\right) \end{cases}$$

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

$$\textsc{ModII}(x, k, y): \begin{cases} \left(\begin{array}{l} x \mapsto \mathsf{inner}(1, k_0, p, D, k', p') * [\textsc{Unlock}(x)]^r_1 \\ * y \mapsto \mathsf{node}(l, k, p_y, D_y, k'_y, p'_y) \\ * \mathsf{nodeList}(p_1, N, y) \\ \wedge (k_0, p) :: D = D_1 :: (k_1, p_1) :: (k_2, p_2) :: D_2 \\ \wedge k_1 < k < k_2 \end{array}\right) \quad \rightsquigarrow \quad \left(\begin{array}{l} x \mapsto \mathsf{inner}(1, k_0, D', k', p') \\ * y \mapsto \mathsf{node}(l, k, p_y, D_y, k'_y, p'_y) \\ * \mathsf{nodeList}(p_1, N, y) \\ * [\textsc{ModII}(x, k, y)]^r_1 \\ \wedge D' = D \uplus (k, y) \end{array}\right) \\ \left(\begin{array}{l} x \mapsto \mathsf{inner}(1, k_0, p, D, k', p') * [\textsc{Unlock}(x)]^r_1 \\ * y \mapsto \mathsf{node}(l, k, p_y, D_y, k'_y, p'_y) \\ * \mathsf{nodeList}(p_1, N, y) * [\textsc{Fix}(k_z, z)] \\ \wedge (k_0, p) :: D = D_1 :: (k_1, p_1) :: (k_2, p_2) :: D_2 \\ \wedge k_1 < k < k_2 \wedge |D| = 2K \wedge \neg\mathsf{root}(h, x) \end{array}\right) \quad \rightsquigarrow \quad \left(\begin{array}{l} x \mapsto \mathsf{inner}(1, k_0, p, D'_1, k_z, z) \\ * z \mapsto \mathsf{inner}(0, k_z, p_z, D'_2, k', p') \\ * y \mapsto \mathsf{node}(l, k, p_y, D_y, k'_y, p'_y) \\ * \mathsf{nodeList}(p_1, N, y) \\ * [\textsc{ModII}(x, k, y)]^r_1 \\ \wedge D'_1 :: (k_z, p_z) :: D'_2 = D \uplus (k, v) \end{array}\right) \\ \left(\begin{array}{l} x \mapsto \mathsf{inner}(1, k_0, p, D, k', p') \\ * y \mapsto \mathsf{node}(l, k, p_y, D_y, k'_y, p'_y) \\ * \mathsf{nodeList}(p_1, N, y) * [\textsc{NewR}(x, k_z, z)]^r_1 \\ \wedge (k_0, p) :: D = D_1 :: (k_1, p_1) :: (k_2, p_2) :: D_2 \\ \wedge k_1 < k < k_2 \wedge |D| = 2K \wedge \mathsf{root}(h, x) \end{array}\right) \quad \rightsquigarrow \quad \left(\begin{array}{l} x \mapsto \mathsf{inner}(1, k_0, p, D'_1, k_z, z) \\ * z \mapsto \mathsf{inner}(1, k_z, p_z, D'_2, k', p') \\ * y \mapsto \mathsf{node}(l, k, p_y, D_y, k'_y, p'_y) \\ * \mathsf{nodeList}(p_1, N, y) \\ * [\textsc{ModII}(x, k, y)]^r_1 \\ \wedge D'_1 :: (k_z, p_z) :: D'_2 = D \uplus (k, v) \end{array}\right) \end{cases}$$

$$\textsc{NewR}(x, k, y): \left(\begin{array}{l} x \mapsto \mathsf{node}(1, -\infty, p_0, D_1, k, y) \\ * y \mapsto \mathsf{node}(1, k, p, D_2, \infty, \mathsf{nil}) \\ * [\textsc{Unlock}(x)]^r_1 * [\textsc{Unlock}(y)]^r_1 \\ * h \mapsto PB \wedge PB = x :: xs \end{array}\right) \quad \rightsquigarrow \quad \left(\begin{array}{l} x \mapsto \mathsf{node}(1, -\infty, p_0, D_1, k, y) \\ * y \mapsto \mathsf{node}(1, k, p, D_2, \infty, \mathsf{nil}) \\ * z \mapsto \mathsf{inner}(0, -\infty, x, [(k, y)], \infty, \mathsf{nil}) \\ * [\textsc{NewR}(x, k, y)]^r_1 \\ * h \mapsto z :: PB \end{array}\right)$$

Figure 18. The interference environment for the B^{Link} tree implementation.

$\{\mathtt{x} \mapsto \mathsf{node}(0, k_0, p, D, k', p')\}$	$\mathtt{lock}(\mathtt{x})$	$\{\mathtt{x} \mapsto \mathsf{node}(1, k_0, p, D, k', p')\}$
$\{\mathtt{x} \mapsto \mathsf{node}(1, k_0, p, D, k', p')\}$	$\mathtt{unlock}(\mathtt{x})$	$\{\mathtt{x} \mapsto \mathsf{node}(0, k_0, p, D, k', p')\}$
$\{\mathsf{emp}\}$	$\mathtt{x} := \mathtt{new}()$	$\{\mathtt{x} \mapsto \mathsf{node}(0, 0, \mathsf{nil}, [\,], 0, \mathsf{nil})\}$
$\{\mathtt{x} \mapsto \mathsf{node}(l, k_0, p, D, k', p')\}$	$\mathtt{N} := \mathtt{get}(\mathtt{x})$	$\left\{\begin{array}{l}\mathtt{x} \mapsto \mathsf{node}(l, k_0, p, D, k', p') \\ \wedge\, \mathtt{N} = \mathsf{node}(l, k_0, p, D, k', p')\end{array}\right\}$
$\left\{\begin{array}{l}\mathtt{x} \mapsto \mathsf{node}(-, -, -, -, -, -) \\ \wedge\, \mathtt{N} = \mathsf{node}(l, k_0, p, D, k', p')\end{array}\right\}$	$\mathtt{put}(\mathtt{N}, \mathtt{x})$	$\left\{\begin{array}{l}\mathtt{x} \mapsto \mathsf{node}(l, k_0, p, D, k', p') \\ \wedge\, \mathtt{N} = \mathsf{node}(l, k_0, p, D, k', p')\end{array}\right\}$
$\{\mathtt{h} \mapsto \mathit{stack}\}$	$\mathtt{PB} := \mathtt{getPrimeBlock}(\mathtt{h})$	$\{\mathtt{h} \mapsto \mathit{stack} \wedge \mathsf{PB} = \mathit{stack}\}$
$\{\mathtt{h} \mapsto - \wedge \mathsf{PB} = \mathit{stack}\}$	$\mathtt{putPrimeBlock}(\mathtt{h}, \mathtt{PB})$	$\{\mathtt{h} \mapsto \mathit{stack} \wedge \mathsf{PB} = \mathit{stack}\}$

Figure 19. Specification of the heap update commands.

implementations and cases can all be proven in a similar style.

$\{\mathsf{emp} \wedge \mathtt{N} = \mathsf{node}(l, k_0, p, D, k', p')\}$	$\mathtt{k} := \mathtt{lowValue}(\mathtt{N})$	$\{\mathsf{emp} \wedge \mathtt{N} = \mathsf{node}(l, k_0, p, D, k', p') \wedge \mathtt{k} = k_0\}$
$\{\mathsf{emp} \wedge \mathtt{N} = \mathsf{node}(l, k_0, p, D, k', p')\}$	$\mathtt{k} := \mathtt{highValue}(\mathtt{N})$	$\{\mathsf{emp} \wedge \mathtt{N} = \mathsf{node}(l, k_0, p, D, k', p') \wedge \mathtt{k} = k'\}$
$\left\{\begin{array}{l}\mathsf{emp} \wedge \mathtt{N} = \mathsf{inner}(l, k_0, v_0, D, k_{n+1}, p') \\ \wedge D = [(k_1, v_1), \ldots, (k_n, v_n)] \\ \wedge k_i < \mathtt{k} \leq k_{i+1}\end{array}\right\}$	$\mathtt{x} := \mathtt{next}(\mathtt{N}, \mathtt{k})$	$\left\{\begin{array}{l}\mathsf{emp} \wedge \mathtt{N} = \mathsf{inner}(l, k_0, v_0, D, k_{n+1}, p') \\ \wedge \mathtt{x} = v_i\end{array}\right\}$
$\left\{\begin{array}{l}\mathsf{emp} \wedge \mathtt{N} = \mathsf{node}(l, k_0, p, D, k', p') \\ \wedge \mathtt{k} > k'\end{array}\right\}$	$\mathtt{x} := \mathtt{next}(\mathtt{N}, \mathtt{k})$	$\left\{\begin{array}{l}\mathsf{emp} \wedge \mathtt{N} = \mathsf{node}(l, k_0, p, D, k', p') \\ \wedge \mathtt{x} = p'\end{array}\right\}$
$\left\{\begin{array}{l}\mathsf{emp} \wedge \mathtt{N} = \mathsf{leaf}(l, k_0, D, k', p') \\ \wedge (\mathtt{k}, v) \in D\end{array}\right\}$	$\mathtt{x} := \mathtt{lookup}(\mathtt{N}, \mathtt{k})$	$\left\{\begin{array}{l}\mathsf{emp} \wedge \mathtt{N} = \mathsf{leaf}(l, k_0, D, k', p') \\ \wedge \mathtt{x} = v\end{array}\right\}$
$\left\{\begin{array}{l}\mathsf{emp} \wedge \mathtt{N} = \mathsf{node}(l, k_0, p, D, k', p') \\ \wedge \|D\| < 2K \wedge \mathtt{k} \notin \mathsf{keys}(D)\end{array}\right\}$	$\mathtt{addPair}(\mathtt{N}, \mathtt{k}, \mathtt{v})$	$\left\{\begin{array}{l}\mathsf{emp} \wedge \mathtt{N} = \mathsf{node}(l, k_0, p, D', k', p') \\ \wedge D' = D \uplus (\mathtt{k}, \mathtt{v})\end{array}\right\}$
$\left\{\begin{array}{l}\mathsf{emp} \wedge \mathtt{N} = \mathsf{node}(l, k_0, p, D, k', v') \\ \wedge (\mathtt{k}, -) \in D\end{array}\right\}$	$\mathtt{removePair}(\mathtt{N}, \mathtt{k})$	$\left\{\begin{array}{l}\mathsf{emp} \wedge \mathtt{N} = \mathsf{node}(l, k_0, p, D', k', p') \\ \wedge D = D' \uplus (\mathtt{k}, -)\end{array}\right\}$
$\left\{\begin{array}{l}\mathsf{emp} \wedge N = \mathsf{leaf}(l, k_0, D, k', p') \\ \wedge k_0 < k \leq k' \wedge \|D\| = 2K\end{array}\right\}$	$\mathtt{M} := \mathtt{rearrange}(\mathtt{N}, \mathtt{k}, \mathtt{v}, \mathtt{x})$	$\left\{\begin{array}{l}\mathsf{emp} \wedge N = \mathsf{leaf}(l, k_0, D_1, k'', x) \\ \wedge M = \mathsf{leaf}(0, k'', D_2, k', p') \\ \wedge D_1 :: D_2 = D \uplus (k, v)\end{array}\right\}$
$\left\{\begin{array}{l}\mathsf{emp} \wedge \mathtt{N} = \mathsf{inner}(l, k_0, p, D, k', p') \\ \wedge k_0 < \mathtt{k} < k' \wedge \|D\| = 2K\end{array}\right\}$	$\mathtt{M} := \mathtt{rearrange}(\mathtt{N}, \mathtt{k}, \mathtt{v}, \mathtt{x})$	$\left\{\begin{array}{l}\mathsf{emp} \wedge \mathtt{N} = \mathsf{inner}(l, k_0, p, D_1, k'', \mathtt{x}) \\ \wedge \mathtt{M} = \mathsf{inner}(0, k'', p'', D_2, k', p') \\ \wedge D_1 :: (k'', p'') :: D_2 = D \uplus (\mathtt{k}, \mathtt{v})\end{array}\right\}$
$\{\mathsf{emp} \wedge \mathsf{PB} = p : ps\}$	$\mathtt{x} := \mathtt{root}(\mathtt{PB})$	$\{\mathsf{emp} \wedge \mathsf{PB} = p : ps \wedge \mathtt{x} = p\}$
$\{\mathsf{emp}\}$	$\mathtt{N} := \mathtt{newRoot}(\mathtt{k}', \mathtt{p}, \mathtt{k}, \mathtt{v}, \mathtt{k}'')$	$\{\mathsf{emp} \wedge \mathtt{N} = \mathsf{inner}(0, \mathtt{k}', \mathtt{p}, [(\mathtt{k}, \mathtt{v})], \mathtt{k}'', \mathsf{nil})\}$
$\{\mathsf{emp} \wedge \mathsf{PB} = xs\}$	$\mathtt{addRoot}(\mathtt{PB}, \mathtt{x})$	$\{\mathsf{emp} \wedge \mathsf{PB} = \mathtt{x} : xs\}$
$\{\mathsf{emp} \wedge \mathsf{PB} = [x_n, \ldots, x_1] \wedge 1 \leq \mathtt{i} \leq n\}$	$\mathtt{x} := \mathtt{getNodeLevel}(\mathtt{PB}, \mathtt{i})$	$\{\mathsf{emp} \wedge \mathsf{PB} = [x_n, \ldots, x_1] \wedge \mathtt{x} = x_\mathtt{i}\}$
$\left\{\begin{array}{l}\mathsf{emp} \wedge \mathtt{N} = \mathsf{node}(l, k_0, p, D, k', p') \\ \wedge D = [(k_1, v_1), \ldots, (k_n, v_n)] \\ \wedge n < 2K\end{array}\right\}$	$\mathtt{b} := \mathtt{isSafe}(\mathtt{N})$	$\left\{\begin{array}{l}\mathsf{emp} \wedge \mathtt{N} = \mathsf{node}(l, k_0, p, D, k', p') \\ \wedge D = [(k_1, v_1), \ldots, (k_n, v_n)] \\ \wedge n < 2K \wedge \mathtt{b} = \mathsf{tt}\end{array}\right\}$
$\left\{\begin{array}{l}\mathsf{emp} \wedge \mathtt{N} = \mathsf{node}(l, k_0, p, D, k', p') \\ \wedge D = [(k_1, v_1), \ldots, (k_n, v_n)] \\ \wedge n = 2K\end{array}\right\}$	$\mathtt{b} := \mathtt{isSafe}(\mathtt{N})$	$\left\{\begin{array}{l}\mathsf{emp} \wedge \mathtt{N} = \mathsf{node}(l, k_0, p, D, k', p') \\ \wedge D = [(k_1, v_1), \ldots, (k_n, v_n)] \\ \wedge n = 2K \wedge \mathtt{b} = \mathsf{ff}\end{array}\right\}$
$\left\{\begin{array}{l}\mathsf{emp} \wedge \mathtt{N} = \mathsf{node}(l, k_0, p, D, k', p') \\ \wedge (\mathtt{k}, v) \in D\end{array}\right\}$	$\mathtt{b} := \mathtt{isIn}(\mathtt{N}, \mathtt{k})$	$\left\{\begin{array}{l}\mathsf{emp} \wedge \mathtt{N} = \mathsf{node}(l, k_0, p, D, k', p') \\ \wedge \mathtt{b} = \mathsf{tt}\end{array}\right\}$
$\left\{\begin{array}{l}\mathsf{emp} \wedge \mathtt{N} = \mathsf{node}(l, k_0, p, D, k', p') \\ \wedge \mathtt{k} \notin \mathsf{keys}(D)\end{array}\right\}$	$\mathtt{b} := \mathtt{isIn}(\mathtt{N}, \mathtt{k})$	$\left\{\begin{array}{l}\mathsf{emp} \wedge \mathtt{N} = \mathsf{node}(l, k_0, p, D, k', p') \\ \wedge \mathtt{b} = \mathsf{ff}\end{array}\right\}$
$\{\mathsf{emp} \wedge \mathtt{N} = \mathsf{leaf}(l, k_0, D, k', p')\}$	$\mathtt{b} := \mathtt{isLeaf}(\mathtt{N})$	$\{\mathsf{emp} \wedge \mathtt{N} = \mathsf{leaf}(l, k_0, D, k', p') \wedge \mathtt{b} = \mathsf{tt}\}$
$\{\mathsf{emp} \wedge \mathtt{N} = \mathsf{inner}(l, k_0, p, D, k', p')\}$	$\mathtt{b} := \mathtt{isLeaf}(\mathtt{N})$	$\{\mathsf{emp} \wedge \mathtt{N} = \mathsf{inner}(l, k_0, p, D, k', p') \wedge \mathtt{b} = \mathsf{ff}\}$
$\{\mathsf{emp} \wedge \mathsf{PB} = \mathtt{x} : xs\}$	$\mathtt{b} := \mathtt{isRoot}(\mathtt{PB}, \mathtt{x})$	$\{\mathsf{emp} \wedge \mathsf{PB} = \mathtt{x} : xs \wedge \mathtt{b} = \mathsf{tt}\}$
$\{\mathsf{emp} \wedge \mathsf{PB} = y : ys \wedge \mathtt{x} \neq y\}$	$\mathtt{b} := \mathtt{isRoot}(\mathtt{PB}, \mathtt{x})$	$\{\mathsf{emp} \wedge \mathsf{PB} = y : ys \wedge \mathtt{b} = \mathsf{ff}\}$
$\{\mathsf{emp}\}$	$\mathtt{stack} := \mathtt{newStack}()$	$\{\mathsf{emp} \wedge \mathtt{stack} = [\,]\}$
$\{\mathsf{emp} \wedge \mathtt{stack} = xs\}$	$\mathtt{push}(\mathtt{stack}, \mathtt{x})$	$\{\mathsf{emp} \wedge \mathtt{stack} = \mathtt{x} : xs\}$
$\{\mathsf{emp} \wedge \mathtt{stack} = y : ys\}$	$\mathtt{x} := \mathtt{pop}(\mathtt{stack})$	$\{\mathsf{emp} \wedge \mathtt{stack} = ys \wedge \mathtt{x} = y\}$
$\{\mathsf{emp} \wedge \mathtt{stack} = [\,]\}$	$\mathtt{b} := \mathtt{isEmpty}(\mathtt{stack})$	$\{\mathsf{emp} \wedge \mathtt{stack} = [\,] \wedge \mathtt{b} = \mathsf{tt}\}$
$\{\mathsf{emp} \wedge \mathtt{stack} = x : xs\}$	$\mathtt{b} := \mathtt{isEmpty}(\mathtt{stack})$	$\{\mathsf{emp} \wedge \mathtt{stack} = x : xs \wedge \mathtt{b} = \mathsf{ff}\}$

Figure 20. Specification of the store update commands.