Imperial College London

Imperial College London

Department of Computing

Reasoning About Concurrent Programs

Author:
Shale XIONG

Supervisor:
Prof. Philippa GARDNER

Submitted in partial fulfillment of the requirements for the MSc degree in Advanced Computing of Imperial College London

September 3, 2015

Abstract

For concurrent programs, reasoning plays an important role, especially for those widely used modules, such as concurrent map. An specification based on abstraction can build a contract between clients and implementation. However even to give a specification for clients is not trivial, because different clients have different expectation. This project gives two specifications for concurrent map, i.e. key value style and map-based style, and argues that map-based specification is better one. Project also proves two popular implementations, B^{link}tree and skip list, and proves that those implementations are linearizable. Besides, in term of proving style, project tries to mechanise proving, which could be helpful for automatic proofs in the future.

Acknowledgements

I would like to think my supervisor Prof. Philippa Gardner for her invaluable advices and support. Thanks to Thomas Dinsdale-Young, Gian Ntzik, Azalea Raad and Julian Sutherland for the discussion and advice about specification. Thanks to Pedro Da Rocha Pinto for his great help on checking all technical details.

At last, a big great thanks to my parents, even though they does not work in academia, but their support means a lot.

Contents

Chapter 1

Introduction

Concurrent programs could lead to non-deterministic result. And concurrent algorithms are typically more complicated than sequential algorithms. Because interleaving could happen in every moment, therefore concurrent algorithms should properly take care of all possible interference. On the other hand, reasoning techniques like RGSep [33] and CAP [9] have been built and used for concurrent reasoning. Compositional reasoning, abstraction and other features are added into proof system, which simplify concurrent reasoning. Therefore for those widely used concurrent modules such as concurrent map, reasoning become more useful than traditional testing, and with new techniques reasoning become more practical.

One example of concurrent algorithm is concurrent B-tree. Many databases use B-tree to organise index, therefore performance and concurrency level of B-tree are important. Sagiv [31] presented B^{link}tree, a variant of B-tree. [6] had a proof of B^{link}tree insertion algorithm, which is a algorithm locks at most one node during insertion. All specifications and proofs are used CAP, which is a profound reasoning technique by introducing abstraction but it becomes struggling to proof linearizability. Besides they only prove B^{link}tree insertion algorithm without compression which is not a realistic implementation.

Apart from proving implementation algorithms, [6] gave abstract specifications for clients. But the specifications are key value disjoint specifications which are not good enough. To use those specifications, clients must choose to either operating on an unshared key value pair or restrain operating by stopping insertion or deletion. This unusual restraints are because CAP does not have a mechanism to describe linearizability and CAP also need to stabilise pre- and post-condition.

With respect of reasoning ability, TaDA [7] can use atomic triple to specify program state just before and after linearizability point by asserting logically atomic. Specifications by atomic triple satisfy what clients expect, linearizability. Additionally TaDA is a proof system that emphasis abstraction and strengthen abstract level reasoning instead of concrete level. Abstraction is a good way to properly describe and prove those algorithms based on complicated data structure. So for example insertion and compression among concurrent B^{link}tree can be proven in a reasonable size of description. Therefore project uses TaDA to specify concurrent map and proves more real-world implementations.

This project has given two specifications for clients using TaDA, key-value specifications and map-based specifications, where both of them explicitly assert linearizability point and can infer to old key value disjoint specifications. These two specifications are better specifications for clients than the old CAP one. In different situations one could slightly better than another, but project have not found any solid evidence that who is the best one. It is a choose and agreement between clients

and implementations that which one should be used, while this project tend to choose map-based specifications since project mainly discuss a general concurrent map which typically is provided by library.

Project has chosen more real-world implementations to show that TaDA is able to deal with complex data structures with very find-grained interference. First implementation is B^{link}tree with compression, which is a blocking algorithm where codes are extend from [6] with compression involved. This new proof is simpler than old proof because TaDA can explicitly assert a logically atomic operation whereas CAP need some proof trick to do that. Another implementation is a direct translation from `ConcurrentSkipListMap` in `java.util.concurrent` which is a non-blocking concurrent skip list algorithm. Project has shown that both two implementations satisfy new concurrent map specifications. For algorithms themselves, proofs have explicitly shown the linearizability point. While many other linearizability work typically choose simple data structure, this project has proven complex data structure.

In term of reasoning style, instead of doing some proof tricks, project has tried to prove in the simplest way which could help to understand automatic proof system in the future. All proofs have similar guard algebra in a way that no proof tricks are involved. They all use multiple level abstractions for the purpose of readable and easy-understanding proofs.

1.1 Contributions

- Present more general specifications for concurrent map by TaDA. This new specifications prove linearizability point and eliminate disjointedness restraint compared with old specifications. Given that we believe it is a better specification for clients.

- Discuss about two specifications style, key-value and map-based. Those two specifications are both suitable for clients to use and we have shown that they are equivalent. Currently we believe that map-based specifications is better one with respect of a general concurrent map module provided by some library.

- Present a simpler proof of B^{link}tree insertion algorithm and a proof for compression. Implementations of B^{link}tree depend on other lower level logically atomic operations, whereas CAP cannot specify them properly, instead the old proof used some proof tricks. However TaDA can specify them properly, therefore the proof is easier to understand.

- Have a proof of concurrent skip list insertion algorithm. Project has proven a real-world concurrent skip list, where codes are translated `ConcurrentSkipListMap` in `java.util.concurrent`.

- Proof linearizability of complex data structure. Many linearizability works prove simple examples, whereas this project has a linearizability proofs of complex data structure.

- Mechanised proving style in all proofs. Proofs of B^{link}tree and concurrent skip list have similar guard algebra, similar proving structure and similar abstraction style. All of them help us to understand how to automatically prove in the future.

1.2 Thesis Outline

Chapter 2 is technical background which consists of two parts. First part recall different proof systems and second part is about concurrent map and its implementation. **Chapter 3** discusses two specifications, key value style and map-based style, and gives our reason why we believe map-based is better one. **Chapter 4** presents proofs of insertion and compression algorithm of B^{link}tree. **Chapter 5** first translates skip list code from Java, and proves insertion algorithm. **Chapter 6** evaluates the project and **Chapter 7** overview this project again and gives the future work.

Chapter 2

Technical Backgroud

This chapter reviews sequential reasoning and concurrent reasoning techniques. §2.3 gives background of concurrent map, including implementations and some verification works. At last, chapter introduces notations that are used in thesis.

2.1 Reasoning Sequential Programs

2.1.1 Hoare Logic

Hoare Logic [15] introduced Hoare Triples, used to express a specification for programs. A Hoare Triple looks like $\{P\}\ \mathbb{C}\ \{Q\}$, where P is pre-condition, Q is post-condition and $\mathbb{C}$ is command. It asserts that if the command $\mathbb{C}$ runs under a state satisfying P, if it terminates, then the program should terminate in a state satisfying Q.

FLOYD ASSIGN
$$\frac{}{\vdash \{P\}\ \mathtt{x} := \mathbb{E}\ \{\exists x.\, (\mathtt{x} = \mathbb{E}[{}^{x}/_{\mathtt{x}}] \wedge P[{}^{x}/_{\mathtt{x}}])\}}$$

CONDITION
$$\frac{\vdash \{P \wedge \mathbb{B}\}\ \mathbb{C}_1\ \{Q\} \quad \vdash \{P \wedge \neg\mathbb{B}\}\ \mathbb{C}_2\ \{Q\}}{\vdash \{P\}\ \mathtt{if}(\mathbb{B})\ \mathbb{C}_1\ \mathtt{else}\ \mathbb{C}_2\ \{Q\}}$$

LOOP/WHILE
$$\frac{\vdash \{P \wedge \mathbb{B}\}\ \mathbb{C}\ \{P\}}{\vdash \{P\}\ \mathtt{while}(\mathbb{B})\ \mathbb{C}\ \{\neg\mathbb{B} \wedge P\}}$$

CONSEQUENCE
$$\frac{P \vdash P' \quad \vdash \{P'\}\ \mathbb{C}\ \{Q'\} \quad Q' \vdash Q}{\vdash \{P\}\ \mathbb{C}\ \{Q\}}$$

SEQUENTIAL
$$\frac{\vdash \{P\}\ \mathbb{C}_1\ \{R\} \quad \vdash \{R\}\ \mathbb{C}_2\ \{Q\}}{\vdash \{P\}\ \mathbb{C}_1;\ \mathbb{C}_2\ \{Q\}}$$

Figure 2.1: Hoare Logic axioms and rules

Hoare Logic has five axioms, *(Floyd) Assign*, *Condition*, *Loop*, *Consequence* and *Sequential*, to reason about programs only involving program variables, i.e. there is no heap manipulation. *Floyd Assign* rule is a more general rule than original assign, which is presented by Floyd [12]. To verify a loop, `while` $(\mathbb{B})\ \mathbb{C}$, the logic requires that an invariant P hold after the loop terminates given that pre-condition $P \wedge \mathbb{B}$ holds before the loop starts to execute. If the loop terminates, then

$P \wedge \neg\mathbb{B}$ holds. The rule of *Consequence* provides the ability to manipulate pre-/post-condition. It means that if a certain pre-/post-condition holds, $\{P'\}\ \mathbb{C}\ \{Q'\}$, then a weaker pre-condition P and stronger post-condition Q should also holds.

2.1.2 Separation Logic

Separation Logic [16, 30] is an extension of Hoare Logic, to provide more reasoning ability to heap resource. It introduces Separation Conjunction (called star) "$*$" to describe two disjoint head cells and notation "$\mathbb{E}_1 \mapsto \mathbb{E}_2$" to describe $\mathbb{E}_1$ points to a value $\mathbb{E}_2$. Separation Logic can prove programs involving heap manipulation, namely commands like $[\mathbb{E}_1] := \mathbb{E}_2$, $\mathtt{x} := [\mathbb{E}]$, `new` and `dispose`, where [.] denotes dereference.

New Rules

Separation Logic has two new axioms to reason about single heap cell, i.e. *Lookup* and *Mutate*, where *Lookup* is to read, and *Mutate* is to write one heap cell. Figure 2.2 presents a simple version of *Lookup* from [30], which reads one heap cell, and it requires that $\mathtt{x}$, $\mathbb{E}$ and $\mathbb{E}''$ are distinct with each others. Figure 2.2 also just presents the most simple version of *Mutate*.

LOOKUP
$$\frac{}{\vdash \{\mathtt{x} = \mathbb{E} \wedge \mathbb{E}' \mapsto \mathbb{E}''\}\ \mathtt{x} := [\mathbb{E}]\ \{\mathtt{x} = \mathbb{E}'' \wedge \mathbb{E}' \mapsto \mathbb{E}''\}}$$
where $\mathtt{x}$, $\mathbb{E}$ and $\mathbb{E}''$ are distinctest with each others.

MUTATE
$$\frac{}{\vdash \{\mathbb{E} \mapsto \_\}\ [\mathbb{E}] := \mathbb{E}'\ \{\mathbb{E} \mapsto \mathbb{E}'\}}$$
where $\mathbb{E} \mapsto \_$ denotes $\exists v.\, \mathbb{E} \mapsto v$,
additionally $\mathbb{E}$ and $\mathbb{E}'$ are distincter with each other.

FRAME
$$\frac{\{P\}\ \mathbb{C}\ \{Q\}}{\{P * R\}\ \mathbb{C}\ \{Q * R\}}$$
Where C does not influence R.

Figure 2.2: Separation Logic new axioms and rule

Instead of reasoning about single heap cell, Separation Conjunction "$*$" is binary operation for composition of separate heap cells and reasoning about them. Assertion $P * Q$ means partial heaps satisfying assertion P is separated with partial heaps satisfying assertion Q. For example, assertion $\mathtt{x} \mapsto 1 * \mathtt{y} \mapsto 1$ describes that there are two cells, one for $\mathtt{x}$ and one for $\mathtt{y}$, and these cells both store value 1. However, if it is $\mathtt{x} \mapsto 1 \wedge \mathtt{y} \mapsto 1$, in classical semantics it asserts that there is only one heap cell, and both $\mathtt{x}$ and $\mathtt{y}$ point to the same cell.

Separation Conjunction leads to another key rule for local reasoning, *Frame* rule. It says that if commands $\mathbb{C}$ does not touch R, it allows reasoning about $\mathbb{C}$ without R. *Frame* is the key to local reasoning, because a proof can frame off all irrelevant parts, and only needs to focus on those part which is accessed.

Classical Semantics And Intuitionistic Semantics

There are two semantics, classical semantics and intuitionistic semantics [16, 30, 29]. For intuitionistic semantics it satisfy *Monotonicity Condition*. If s denotes program stack with form $Var \mapsto Val$,

and h denotes heap with form $Loc \mapsto Val$, *Monotoncity Condition* means that if $s, h \models P$ and $h \sqsubseteq h'$, then $s, h' \models P$, where $h \sqsubseteq h'$ means h is a subset of h'. Intuitively, in intuitionistic semantics, $P \wedge Q$ is equivalent to $(P * \texttt{true}) \wedge (Q * \texttt{true})$ in classical semantics.

Ishtiaq and O'Hearn [16] points out that classical semantics is more expressive than intuitionistics. Therefore, typically Separation Logic use classical semantics. However, in concurrent reasoning, intuitionistics is useful for reasoning about shared heap, for instance logic like CAP [9] and TaDA [7].

2.1.3 Separation Algebra

After Separation Logic, the idea of separation conjunction $*$ is generated to other kind of resources. Calcagno et al. [4] abstract Separation Logic to Separation Algebra, and use it to model resources. In Separation Algebra, resource is defined as a cancellative, partial commutative monoid, $(\Sigma, \bullet, u)$, where u is unity and $\bullet$ is a binary operation. Thus with Separation Algebra, assertion $P * Q$ holds if and only if $h_P \bullet h_Q$ are defined, where $h_P \models P$, $h_Q \models Q$ and $h_P, h_Q \in \Sigma$.

Separation Algebra is general model that is able to model and reason about other kinds of resource, such as program variables [2]. Besides, in concurrent reasoning, permission [3, 1] is important concept to describe interference. Calcagno et al. [4] shows that permission is actually one example of Separation Algebra.

2.2 Reasoning Concurrent Programs

2.2.1 Owicki-Gries and Rely-Guarantee

Before appearance of Separation Logic there are two major concurrent reasoning techniques, Owicki-Gires and Rely-Guarantee. Owicki-Gries can have a local proof but under many restraints, however Rely-Guarantee is a compositional method.

Owicki-Gries

Owicki and Gries [23] introduced a method to reason about concurrency after Hoare Logic, it uses interference-freedom conditions and auxiliary variables to verify concurrency. Interference-freedom means that commands in one thread must not interfere with any intermediate state of other threads in a way that would break stability of assertions.

$$\text{OG-Concurrent}\quad \frac{\vdash \{P_1\}\ \mathbb{C}_1\ \{Q_1\} \quad \ldots \quad \vdash \{P_n\}\ \mathbb{C}_n\ \{Q_n\} \quad \texttt{are interference-free}}{\vdash \{P_1 \wedge \cdots \wedge P_n\}\ \mathbb{C}_1\ ||\ldots||\ \mathbb{C}_n\ \{Q_1 \wedge \cdots \wedge Q_n\}}$$

Figure 2.3: Owicki-Gries concurrent rule for reasoning concurrency

Auxiliary variables is an additional assignment to help reasoning. Typically, those auxiliary variables recording the program history. To use those auxiliary variables, one should also should programs with auxiliary variables is equivalent with original programs.

Rely-Guarantee

Rely-Guarantee introduced by Jones [17], uses rely condition and guarantee condition instead of restrict interference-free condition to reason about concurrency. It means Rely-Guarantee do not need to check that a thread does not interfered with all other threads, yet interference must be limited within rely condition and guarantee condition.

Rely expresses possible change by environment, in other word other threads, whereas guarantee expresses what operations are allowed for current thread. Thus, to reason about concurrent programs, Rely-Guarantee uses rule shown in figure 2.4 [18]. This rule says that before concurrent operations program know there is rely R and guarantee G for concurrent commands $\mathbb{C}_1$ and $\mathbb{C}_2$. During each thread, guarantee G are split into G_1 and G_2. One thread will treat guarantees from other threads as its rely. For example, to reason about $\mathbb{C}_1$, it treat guarantee G_2 as its rely.

RELY-GUARANTEE

$$\frac{\begin{array}{c} R \vee G_2, G_1 \vdash \{P\}\ \mathbb{C}_1\ \{Q_1\} \\ R \vee G_1, G_2 \vdash \{P\}\ \mathbb{C}_2\ \{Q_2\} \\ G_1 \vee G_2 \vdash G \\ Q_1 \wedge Q_2 \wedge (R \vee G) \vdash Q \end{array}}{R, G \vdash \{P\}\ \mathbb{C}_1 \mid\mid \mathbb{C}_2\ \{Q\}}$$

Figure 2.4: Rely-Guarantee rule

Rely-Guarantee is a compositional method, given that rely condition and guarantee condition are compositional. If $\mathbb{C}_1$ and $\mathbb{C}_2$ have already been proven, it can scale the proof by producing new rely R and guarantee G from the old proof, then we can proof $\mathbb{C}_1||\mathbb{C}_2$. However, even though Rely-Guarantee is a useful and compositional way, when it comes to intimate interference, specification of rely and guarantee becomes rather difficult [18]. Apart from that, Rely-Guarantee suffers the same problem as Hoare Logic, it is a global proof. It means Rely-Guarantee does not allow to chuck pre-conditions into pieces to prove each threads as locally as Owicki-Gries.

2.2.2 Separation Logic Based Reasoning

There are many new proof systems for reasoning concurrency based on Separation Logic, for example RGSep. RGSep combined Concurrent Separation Logic, an extension of Separation Logic and Rely-Guarantee. Thus, it is a compositional and local reasoning technique. Deny-Guarantee, extended from RGSep, allows reasoning about dynamic scoped concurrency by treating interference as a resource.

Concurrent Separation Logic

Reynolds [30] gave a trivial *Concurrent* rule, which requires $\mathbb{C}_1$ and $\mathbb{C}_2$ musts operating on completely separate resources. Because P_1 and P_2 assert separation heap, so two programs $\mathbb{C}_1$ and $\mathbb{C}_2$ can run concurrently without any interference controls. O'Hearn [21] extended this rule by introducing resource environment Δ in Concurrent Separation Logic where here resource is only a list of variables that must be access exclusively, which is different with §2.1.3.

Concurrent Separation Logic assumes a new command, `with` r `when` $\mathbb{B}$ `do` $\mathbb{C}$, to control concurrent operation, which $\mathbb{C}$ is critical section code, $\mathbb{B}$ is condition to entry critical section, and r is resource that are accessed. *Critical Region* is used to reason about `with` r `when` $\mathbb{B}$ `do` $\mathbb{C}$. This

Concurrent

$$\frac{\Delta_1 \vdash \{P_1\}\ \mathbb{C}_1\ \{Q_1\} \quad \Delta_2 \vdash \{P_2\}\ \mathbb{C}_2\ \{Q_2\}}{\Delta_1 \cup \Delta_2 \vdash \{P_1 * P_2\}\ \mathbb{C}_1\ ||\ \mathbb{C}_2\ \{Q_1 * Q_2\}}$$

Critical Region

$$\frac{\Delta \vdash \{(P * RI_r) \wedge \mathbb{B}\}\ \mathbb{C}\ \{Q * RI_r\}}{\Delta, r : RI_r \vdash \{P\}\ \texttt{with}\ r\ \texttt{when}\ \mathbb{B}\ \texttt{do}\ \mathbb{C}\ \{Q\}}$$

Where no other thread can modify variables free in P

Figure 2.5: Reasoning rule in Concurrent Separation Logic

rule says that inside critical session pre-condition assert that resource invariant RI_r at beginning, within critical session it may not satisfy, but it must re-construct the resource invariant RI_r before out of critical session.

Concurrent Separation Logic can proof coarse-grained programs by simply defining a resource and resource invariant. But in a scenario where large amount of threads accesses the same resource in turn, it could be difficult to find a proper resource invariant, and each thread must hold resource before finishing process.

RGSep

The idea behind RGSep [33] is simple, to combine Concurrent Separation Logic and Rely-Guarantee where Concurrent Separation Logic is local and Rely-Guarantee is compositional. To combine those two proof system, RGSep introduces a new heap model by dividing heap into global heap and local heap. Local heap is as the same as Separation Logic. But global heap, sometime called shared heap, is reasoned by a method combining Rely-Guarantee and Concurrent Separation Logic. Because of new heap model, assertions are distinguished by "box". Assertions about local heap forms P, yet assertions about shared heap is wrapped by a box $\boxed{P}$.

In term of reasoning, updating a shared state is restrained by a physical atomic command $\langle \mathbb{C} \rangle$ and interference of shared state is restrained by rely condition R and guarantee condition G. If a thread wants to update shared state, it should be actions that are allowed by guarantee $(P \rightsquigarrow Q) \subseteq G$ and post-condition Q must be stable with respect to rely R. *Parallel Composition* combining Rely-Guarantee and Concurrent Separation Logic, is used to reason about concurrency. It has the similar shape as Rely-Guarantee but allows assertions to be split as a way in Concurrent Separation Logic, and therefore reasons locally within each thread.

Update Shared Region

$$\frac{\vdash \{P\}\ \mathbb{C}\ \{Q\} \quad \vdash (P \rightsquigarrow Q) \subseteq G \quad \vdash Q\ \texttt{is stable under}\ R}{R, G \vdash \{\boxed{P}\}\ \langle \mathbb{C} \rangle\ \{\boxed{Q}\}}$$

Parallel Composition

$$\frac{R \cup G_2, G_1 \vdash \{P_1\}\ \mathbb{C}_1\ \{Q_1\} \quad R \cup G_1, G_2 \vdash \{P_2\}\ \mathbb{C}_2\ \{Q_2\}}{R, G_1 \cup G_2 \vdash \{P_1 * P_2\}\ \mathbb{C}_1\ ||\ \mathbb{C}_2\ \{, Q_1 * Q_2\}}$$

Figure 2.6: Rule in RGSep

RGSep can proof fine-grained programs, for example a fine-grained concurrent linked list. Both Rely-Guarantee and Concurrent Separation Logic are rather difficult to do that. Because Rely-Guarantee cannot give rely conditions and guarantee conditions in a local way whereas Concurrent Separation Logic does not allows any interference. Then both of them cannot proof a fine-grained program properly.

Deny-Guarantee

Deny-Guarantee [10] is another form of Rely-Guarantee, but it use permissions, or fractional permissions [3], to express interference. In Deny-Guarantee, permissions are actions bound with a fractional permission number π. If $\pi = 1$, it means that only current thread are allowed to do the updating. If $\pi = 0$, it means only environment are allowed. Whereas if π between 0 and 1, it could be either partial guarantee or partial deny, for instance $\pi = \frac{1}{2}gurd$ or $\pi = \frac{1}{2}deny$. Partial guarantee give permission to both current thread and environment, but partial deny stop both. The fractional permission number are allowed to split and merge. For example $[\mathtt{x} \mapsto 1 \rightsquigarrow \mathtt{x} \mapsto 2]_1$ means an permission that allowing updating the state of $\mathtt{x}$ from 1 to 2 with fraction 1 (only me allowed). This permission might be split into two parts connected by separation conjunction, $[\mathtt{x} \mapsto 1 \rightsquigarrow \mathtt{x} \mapsto 2]_{1/2guar} * [\mathtt{x} \mapsto 1 \rightsquigarrow \mathtt{x} \mapsto 2]_{1/2guar}$. Both of them are allowed to pass into two threads, where both threads has the knowledge that environment and threads themselves can do this action. After both threads are dead, two permissions might come back the parent thread and are allowed to be merged into the original one. Deny-Guarantee has more flexibility in term of reasoning permission, and also it supports thread forking and dying.

CoLoSL

CoLoSL [28] introduces overlapping conjunction $P \circledast Q$ to explicitly express concrete level overlapping. $P \circledast Q$ asserts that heaps h_P and h_Q are overlapped which means there exists a overlapped part h which satisfies $h_P = h'_P \uplus h$, $h_Q = h'_Q \uplus h$ and the non-overlapped part h'_P must disjoint with h'_Q.

Another contribution is flexibility of interference. Interference is a set of allowed actions, which basically describes the maximum updating ability for part of heap. In CoLoSL, *Shift* rule allows changing interference. $I \sqsubseteq^P I'$ intuitively means that I' cannot allow any new action which is not in I, besides all actions in I which relates to assertion P must be still in I'. *Merge* rule allows merging two part of heap by $P \circledast Q$ as well as their interference by $I_P \cup I_Q$.

$$\textsc{Shift}\quad \frac{I \sqsubseteq^P I'}{\vdash \boxed{P}_I \Rightarrow \boxed{P}_{I'}}$$

$$\textsc{Forget}\quad \frac{}{\vdash \boxed{P \circledast Q}_I \Rightarrow \boxed{P}_I}$$

$$\textsc{Merge}\quad \frac{I_P \texttt{ and } I_Q \texttt{ are compatible}}{\vdash \boxed{P}_{I_P} * \boxed{Q}_{I_Q} \Rightarrow \boxed{P \circledast Q}_{I_P \cup I_Q}}$$

Figure 2.7: Reasoning rules in CoLoSL

2.2.3 Reasoning in Abstraction

Except reasoning in concrete level, CAP and TaDA focus on abstraction. Abstraction gives more reasoning power by presenting a abstract specification which separate clients knowledge and implementations. This abstract specification like a contract between clients and implementations.

Abstract Predicate

Parkinson and Bierman [25] introduces abstract predicate. A abstract predicate, compared with normal predicate, can choose to reveal only part of the state of a module but encapsulate other. However for a normal predicate is a way that presents everything concrete details. For example, a abstract predicate $\texttt{list}_{abstract}(x, L)$ says that there there is a list at address x with state L, yet there is no restrain about the concrete shape about this list. This $\texttt{list}_{abstract}(x, L)$ could be interpreted as a single-linked list, double-linked list, array or other reasonable forms. However a normal predicate $\texttt{list}_{normal}(x, L)$, in Separation Logic for instance, is usually a way to present single-linked list in concrete level. Intuitively a normal predicate like a "short hand" of bunch of heap cells.

With abstract predicate, a specification become a contract between clients and the code. Clients does not need to know all the details of codes and those codes might change as long as they satisfy one interpreting of the specification.

CAP

Concurrent Abstract Predicates(CAP) [9] combines abstract predicate [25] and other concurrent reasoning techniques [21, 3, 10, 33]. CAP use abstract predicates $\alpha(\mathbb{E}_1, \mathbb{E}_2, \ldots, \mathbb{E}_n)$ to describe a module with state $\mathbb{E}_1, \mathbb{E}_2, \ldots, \mathbb{E}_n$. CAP adopts rules from RGSep [33] and simplified fractional permissions from Deny-Guarantee [10]. With abstract and rules from previous works, concurrent programs can be verified in abstract level without knowing low-level implementation.

In CAP functions could be specified by updating states of some abstract predicates. Clients only use this abstract specifications, but leave all those details to implementation. For instance a lock in abstract level, one can give a specification shown in Figure 2.8 with some restrains about abstract predicates. Clients now can use this module. In the implementation sides, those persons who implement this lock should have some interpretation about abstract predicate `isLock` and `locked`, and those interpretation should satisfy restraint that `isLock` allows duplicating but not `locked`.

$$\vdash \{\texttt{isLock(x)}\}\ \texttt{lock(x)}\ \{\texttt{isLock(x)} * \texttt{locked(x)}\}$$
$$\texttt{isLock(x)} \Longleftrightarrow \texttt{isLock(x)} * \texttt{isLock(x)}$$
$$\texttt{locked(x)} * \texttt{locked(x)} \Longrightarrow \texttt{false}$$

Figure 2.8: Abstract specification of lock by CAP

CAP generalise abstract predicate to concurrency and adopt many previous works. Therefore, it has ability to reasoning about find-grained concurrent programs in both abstract and concrete level.

Linearizability

Linearizability [14] is a correctness condition for concurrency, it requires that the history of the whole concurrency environment should be "equivalent" to a sequential history, where a sequential history is a complete order of all event under the condition that all invocations immediately return. Linearizability is a local property, because if the local history of all threads are linearizable, the whole history is linearizable. Another equivalent definition is that if all functions have been called, if there is a linearizability point in each function, it is equivalent to linearizable history, where a linearizability point is, intuitively, one atomic "jump" of program state.

TaDA

TaDA introduces atomic triple to describe logically atomic updating. This new triple asserts that the program state just before the linearizability point and just after this point.

More specific, pre-/post-condition consist of two parts, public and private. Public part is shared by threads and private part is owned by current thread. Notation $\mathbb{V}x \in X$ means that the environment is allowed to change x, but x is restrained by set X. Besides, the environment should guarantee that the pre-condition is held. Then command $\mathbb{C}$ atomically updates the public part from state $p(x)$ to $q(x, y)$ and update private part from state p_p to $q_p(x, y)$. Notation $\exists y \in Y$ means it allows implementation to choose a possible y as long as in set Y.

$$\vdash \mathbb{V}x \in X. \left\langle p_{private} \,\middle|\, p_{public}(x) \right\rangle \mathbb{C} \quad \exists y \in Y. \left\langle q_{private}(x, y) \,\middle|\, q_{public}(x, y) \right\rangle$$

Figure 2.9: Atomic triple

Let takes stack operation as example. Presume there is a stack predicate $\texttt{stack}(x, \alpha)$, where it asserts a stack at address x with abstract state α. Again presume there are functions $\texttt{pop(x)}$ and $\texttt{push(x, v)}$. In TaDA, we can use atomic triple to have strong specification. These two specifications express that the environment is allowed to change abstract state α, but environment also have responsibility to preserve the pre-condition, i.e. not dispose this stack or change the address of this stack. For `pop` function, the specification actually expresses that at beginning I do not have any local resources, but I know there is a stack in public part. Then there is a moment that this function call update the program state from pre-condition to post-condition by one step. the post-condition expresses program state just after updating. Thus these two specifications actually describe the linearizability point of a concurrent stack.

$$\begin{array}{c} \mathbb{V}\alpha. \left\langle \texttt{emp} \,\middle|\, \texttt{stack}(\texttt{x}, \alpha) \right\rangle \\ \texttt{v} := \texttt{pop}(\texttt{x}) \\ \vdash \\ \exists v \in \mathbb{N}. \left\langle \texttt{v} = v \,\middle|\, \begin{array}{c} \exists \alpha'. \texttt{stack}(\texttt{x}, \alpha') \wedge \\ \alpha = [v] +\!\!+ \alpha' \end{array} \right\rangle \end{array}$$

$$\begin{array}{c} \mathbb{V}\alpha. \left\langle \texttt{v} = v \,\middle|\, \texttt{stack}(\texttt{x}, \alpha) \right\rangle \\ \texttt{push}(\texttt{x}, \texttt{v}) \\ \vdash \\ \exists v \in \mathbb{N}. \left\langle \texttt{v} = v \,\middle|\, \exists \alpha'. \texttt{stack}(\texttt{x}, [v] +\!\!+ \alpha) \right\rangle \end{array}$$

(a) TaDA

$$\begin{array}{c} \{\exists \alpha. \texttt{stack}(\texttt{x}, \alpha)\} \\ \texttt{v} := \texttt{pop}(\texttt{x}) \\ \vdash \\ \{\exists \alpha, v. \texttt{stack}(\texttt{x}, \alpha) \wedge \texttt{v} = v\} \\ \{\exists \alpha, v. \texttt{stack}(\texttt{x}, \alpha) \wedge \texttt{v} = v\} \\ \texttt{push}(\texttt{x}, \texttt{v}) \\ \vdash \\ \{\exists \alpha, v. \texttt{stack}(\texttt{x}, \alpha) \wedge \texttt{v} = v\} \\ \texttt{stack}(\texttt{x}, \alpha) \Longrightarrow \texttt{stack}(\texttt{x}, \alpha) * \texttt{stack}(\texttt{x}, \alpha) \end{array}$$

(b) CAP

Figure 2.10: Specifications for stack

We can use CAP [9] to give specifications for the same functions. But they are weaker than TaDA specifications, because a abstract predicate in CAP requires stable until the end of function call.

2.3 Concurrent Map

Map is widely use module, they are typically provided by library with sophistic algorithm for performance. Implementations of concurrent map are designed to allow as many concurrency as possible.

2.3.1 Popular Implementation

There are many popular implementations of map or concurrent map in industry. In sequential world, C++ STL `std::map` is typically implemented by red-black trees [34]. Red-black tree is a binary tree and this tree is almost balanced, so sometime it is classified as balanced search tree. `std::unordered_map`, a new sequential map module provided by C++11, is implemented by hash map [34]. Dictionary in Python which is a container for key value pair is implemented by a hash table as well [32]. Hash map is typically implemented as a list of hash value and each hash is bound with a bucket which contains those value with the same hash. In Java, `concurrentMap` has two implementations, hash map and skip list [22]. In addition, database use B-tree and its variants to organise index to maximise performance and concurrency as respect of file system [5]. B-tree is a balanced searching tree with each node contains several key value pairs [5]. To conclude, a map is typically implemented by,

- Balanced search tree, where either relatively a simple red-black tree or B-tree.
- Hash, which the typical data structure is a list of hash and buckets.
- Skip list, with potentially the same complexity as balanced search tree but with less maintain cost [27].

2.3.2 Verification

Kung and Lehman [19] presented a concurrent balanced tree algorithm and also gave very short correctness proof by proving same global invariant of this algorithm. Pugh [26] proof a concurrent skip list algorithm by relatively formal way, however it does not proof linearizability. [11] gives a summary of examples that have been proof linearizable. However, many examples are stack, queen or set, and typically they are use relatively simple implementations, for instance list-based implementations.

Except verifying a specific algorithm, [6] had specifications of concurrent map in abstract level using CAP, and also gave proofs of three different implementations, coarse-grained linked list, trivial hash map, B^{link}Tree without compression.

2.3.3 Key-value disjoint specification

[6] presented a key-value disjointed specification by CAP. There are two abstract predicates for each possible key, $\texttt{in}(x, k, v)$ and $\texttt{out}(x, k, v)$, where x is the address of the whole map, k is key and v is current value bound with key. Both abstract predicates are marked with an extra type including `def`, `ins` and `rem`. If the abstract predicate is typed with `def`, it means current key is allowed to do both insertion and deletion but it must hold the full permission, i.e. $\texttt{in}_{\texttt{def}}(x, k, v)_1$ or $\texttt{out}_{\texttt{def}}(x, k)_1$. In this case, it means other thread cannot concurrently modify this key. In addition, whenever a thread holds full permission, it is allowed to transfer abstract predicate extra type among `def`, `ins` and `rem`.

Another two types, `ins` and `rem`, only support one way modification but with only partial permission. If predicate is typed with `ins` it only allows insertion. Similarly for `rem` but it only allows deletion. By restraining the interference, it actually increase concurrency and also guarantee predicate stable as well.

With abstract predicates `in` and `out`, [6] gave specifications for `insert`, `remove` and `search`. Figure 2.11 reviews part of specifications to help understand how the key-value disjoint specification work.

$$
\begin{array}{ccc}
\{\mathtt{in}_{\mathtt{def}}(\mathtt{x},\mathtt{k},v)_1\} & \mathtt{insert}(\mathtt{x},\mathtt{k},\mathtt{v}) & \{\mathtt{in}_{\mathtt{def}}(\mathtt{x},\mathtt{k},\mathtt{v})_1 \wedge \mathtt{ret} = v\} \\
\{\mathtt{out}_{\mathtt{def}}(\mathtt{x},\mathtt{k})_1\} & \mathtt{insert}(\mathtt{x},\mathtt{k},\mathtt{v}) & \{\mathtt{in}_{\mathtt{def}}(\mathtt{x},\mathtt{k},\mathtt{v})_1 \wedge \mathtt{ret} = \mathtt{null}\} \\
\{\mathtt{in}_{\mathtt{ins}}(\mathtt{x},\mathtt{k},v)_\pi\} & \mathtt{insert}(\mathtt{x},\mathtt{k},\mathtt{v}) & \{\mathtt{in}_{\mathtt{ins}}(\mathtt{x},\mathtt{k},\mathtt{v})_\pi \wedge \mathtt{ret} = v\} \\
\{\mathtt{out}_{\mathtt{ins}}(\mathtt{x},\mathtt{k})_\pi\} & \mathtt{insert}(\mathtt{x},\mathtt{k},\mathtt{v}) & \{\mathtt{in}_{\mathtt{ins}}(\mathtt{x},\mathtt{k},\mathtt{v})_\pi \wedge \mathtt{ret} = \mathtt{null}\} \\
\{\mathtt{in}_{\mathtt{def}}(\mathtt{x},\mathtt{k},v)_1\} & \mathtt{remove}(\mathtt{x},\mathtt{k}) & \{\mathtt{out}_{\mathtt{def}}(\mathtt{x},\mathtt{k})_1 \wedge \mathtt{ret} = v\} \\
\{\mathtt{out}_{\mathtt{def}}(\mathtt{x},\mathtt{k})_1\} & \mathtt{remove}(\mathtt{x},\mathtt{k}) & \{\mathtt{out}_{\mathtt{def}}(\mathtt{x},\mathtt{k})_1 \wedge \mathtt{ret} = \mathtt{null}\} \\
\{\mathtt{in}_{\mathtt{rem}}(\mathtt{x},\mathtt{k},v)_\pi\} & \mathtt{remove}(\mathtt{x},\mathtt{k}) & \{\mathtt{out}_{\mathtt{rem}}(\mathtt{x},\mathtt{k})_\pi \wedge \mathtt{ret} = v\} \\
\{\mathtt{out}_{\mathtt{rem}}(\mathtt{x},\mathtt{k})_\pi\} & \mathtt{remove}(\mathtt{x},\mathtt{k}) & \{\mathtt{out}_{\mathtt{rem}}(\mathtt{x},\mathtt{k})_\pi \wedge \mathtt{ret} = \mathtt{null}\}
\end{array}
$$

Figure 2.11: Review: Specifications of `insert`, `remove` and `search` by CAP

Chapter 3

Map Specification

This chapter mainly discusses what does it means for a good specification for clients side. We believe that key-value disjoint specification is not good enough for clients. Therefore we adopt the key value pair style, but build a new key value style specification with less restraints by TaDA, which means single key is no long not required to be disjointed, but can be shared. We will also show how to reduce the new key value specification to the old one.

However, we believe a trivial way to think about a concurrent index or a concurrent map is a specification based on map. Given that, a map-based specification is presented which is very similar with key value style.

We will try to argue our intuition about a map-based specification could be a better than key value style, even though it is also shown that map-based specification is equivalent to key value specification. Therefore what the meaning of a good specification could base on how this map will be used, and by who.

3.1 Key-value Specification by TaDA

We first present our new key-value specifications by TaDA and then reduce our new specification to the old CAP one. Reduction shows that whatever preview specification can do, our new specification should also be capable to prove.

new key-value specification by TaDA

TaDA [7] claims that it extends from CAP by adding the ability to proof logically atomic, i.e. similar to linearizability, which is one of the most important correctness property of concurrency. Thus we believe it should have a better specification for concurrent map by TaDA.

We start at adopting the idea of `in` and `out`, but use a new uniformed abstract predicate. $\texttt{key}(x, k, v)$ with concrete value v is similar with `in`, and $\texttt{key}(x, k, \epsilon)$ with special token ϵ is similar with `out`. However the biggest difference is our new abstract predicate `key` are allowed to shared, and allowed to do both insertion and deletion in the same time, whereas `in` and `out` must make choice between no sharing or one-way updating.

With new abstract predicates, we have simpler specifications for `insert`, `remove` and `search`. The quantity of specifications are compressed to one for each function, but the biggest change is this

$$\texttt{key}(x,k,v) \stackrel{\text{def}}{=} \exists a.\, \mathbf{Key}_a(x,k,v) * [\text{KEY}]_a$$
$$\texttt{key}(x,k,\epsilon) \stackrel{\text{def}}{=} \exists a.\, \mathbf{Key}_a(x,k,\epsilon) * [\text{KEY}]_a$$
$$[\text{KEY}] : \quad \forall v.\, v \rightsquigarrow \epsilon$$
$$\epsilon \rightsquigarrow v$$
$$[\text{KEY}] \Longrightarrow [\text{KEY}] * [\text{KEY}]$$

Figure 3.1: `key` definition and guard

new specification also shows that there is a linearizability point and what could happen in this the point.

$$\begin{array}{ll} \vdash \mathbb{\forall} v. & \langle \texttt{key(x,k,}v\texttt{)} \rangle \\ & \texttt{insert(x,k,v)} \\ & \langle \texttt{key(x,k,v)} \wedge ((v \neq \epsilon \wedge \texttt{ret} = v) \vee (v = \epsilon \wedge \texttt{ret} = \texttt{null})) \rangle \\ \vdash \mathbb{\forall} v. & \langle \texttt{key(x,k,}v\texttt{)} \rangle \\ & \texttt{remove(x,k)} \\ & \langle \texttt{key(x,k,}\epsilon\texttt{)} \wedge ((v \neq \epsilon \wedge \texttt{ret} = v) \vee (v = \epsilon \wedge \texttt{ret} = \texttt{null})) \rangle \\ \vdash \mathbb{\forall} v. & \langle \texttt{key(x,k,}v\texttt{)} \rangle \\ & \texttt{search(x,k)} \\ & \langle \texttt{key(x,k,}v\texttt{)} \wedge ((v \neq \epsilon \wedge \texttt{ret} = v) \vee (v = \epsilon \wedge \texttt{ret} = \texttt{null})) \rangle \end{array}$$

Figure 3.2: Key-value style specifications for concurrent map

In specifications, $\mathbb{\forall} v$ says that the environment is allowed to change value v, but environment must preserve a program state which satisfies pre-condition, for instance, here environment is not allowed to completely dispose map at address `x`. Yet it is allowed to remove `k`, since $\texttt{key(x,k,}\epsilon\texttt{)}$ with $v = \epsilon$ means `k` are not exist in the map at `x`.

The atomic triple $\langle P \rangle\ \mathbb{C}\ \langle Q \rangle$ means that $\mathbb{C}$ updates program state from P to Q in one atomic or logically one atomic step. Before updating or after updating, environment is allowed to change the program state. Intuitively, the atomic triple describes linearizability point of $\mathbb{C}$. For instance, specification of `insert` says that environment is allowed to change the value bound with key `k` or even delete `k`, but if this function terminates it is known that insides this function call there is a linearizability point that new key-value pair is added in or updated. Besides, after this linearizability point, even though function is not returns, environment is allowed to interfere again. Because the post-condition only asserts the exact moment after linearizability point.

Reduction to CAP key-value disjointed specification

To proof that our new specification can reduce to the old CAP one there are few tasks,

- simulate partial permission behaviour,
- simulate `def`, `ins` and `rem`

We define a relatively complex guard $[\text{KEY\_CAP}(\mathit{type}, \pi)]$ where *type* provide feature that is similar to `def`, `ins` and `rem`, and π that is similar to partial permission.

Since we need to use a new guard system to restrain interference, it is necessary to build a extra abstract layer. So we define `in` and `out` to a new region, namely $\mathbf{Key\_CAP}_a(x,k,v)$ or

$\mathbf{Key\_CAP}_a(x, k, \epsilon)$ and new guard [KEY_CAP($type, \pi$)]. This new region is directly interpreted as $\texttt{key}(x, k, v)$ or $\texttt{key}(x, k, \epsilon)$.

$$\texttt{in}_{type}(x, k, v)_\pi \stackrel{\text{def}}{=} \exists a.\, \mathbf{Key\_CAP}_a(x, k, v) * [\text{KEY\_CAP}(type, \pi)]_a$$
$$I(\mathbf{Key\_CAP}_a(x, k, v)) \stackrel{\text{def}}{=} \texttt{key}(x, k, v)$$
$$\texttt{out}_{type}(x, k)_\pi \stackrel{\text{def}}{=} \exists a.\, \mathbf{Key\_CAP}_a(x, k, \epsilon) * [\text{KEY\_CAP}(type, \pi)]_a$$
$$I(\mathbf{Key\_CAP}_a(x, k, \epsilon)) \stackrel{\text{def}}{=} \texttt{key}(x, k, \epsilon)$$

Figure 3.3: Definition and Interpretation of `in` and `out` to `key` and new guard

Now we explain guard [KEY_CAP($type, \pi$)]. The first parameter *type* has three possible values `def`, `ins` and `rem`, and the second parameter π is used to simulate the fractional permission. Therefore actions bound with guard depends on the current state of *type* and π.

$$\begin{array}{rll} [\text{KEY\_CAP}(type, \pi)] : \forall v. & v \rightsquigarrow \epsilon & \text{where } (type = \texttt{def} \wedge \pi = 1) \vee type = \texttt{rem} \\ & \epsilon \rightsquigarrow v & \text{where } (type = \texttt{def} \wedge \pi = 1) \vee type = \texttt{ins} \end{array}$$

Figure 3.4: Actions bound with [KEY_CAP($type, \pi$)]

At last by defining a proper guard algebra, it is allow to manipulate *type* and π in the same way as what can do before. The guard algebra says that if $\pi = 1$ which corresponds to full permission, *type* can transfer among three possible states. If *type* is fixed, it is allowed to split or merge π just as same mechanism as fractional permission.

$$[\text{KEY\_CAP}(\texttt{def}, 1)] \Longleftrightarrow [\text{KEY\_CAP}(\texttt{ins}, 1)] \Longleftrightarrow [\text{KEY\_CAP}(\texttt{rem}, 1)]$$
$$[\text{KEY\_CAP}(type, k, \pi_1 + \pi_2)] \Longleftrightarrow [\text{KEY\_CAP}(type, k, \pi_1)] * [\text{KEY\_CAP}(type, k, \pi_2)]$$
$$\text{where } \pi_1 + \pi_2 \leqslant 1$$

Figure 3.5: Guard algebra of [KEY_CAP($type, \pi$)]

3.2 Map-based Specification

Another new style of specifications by TaDA are based on the whole map instead of key-value, which we think is a more trivial style. If saying concurrent map, typically it is expected specifications based on the whole map, but not a key-value pairs. Another motivation is that usually computer engineers choose to synchronise a map instead of to synchronise separate key-value pair, therefore map-based specifications look like a better idea when an external synchronisation involves. The map-based specifications are similar with §3.1 in term of describing linearizability.

This abstract predicate $\texttt{map}(x, S)$ is simply defined as a region type with exactly the same parameters and a duplicatable guard. The guard [MAP] allows modifying one key value pair in a linearizability point, modifying a value, or deleting a key value pair. $S[k \mapsto v]$ denotes that if key k already exists, it re-map value to v, otherwise a new key value pair is added.

3.3 Key-Value Versus Map-based

Key-value specifications are equivalent with map-based ones, but we try to argue that map-based style is better choice when external synchronisation involves. Besides, map-based style has a clear

$$\vdash \mathbb{V}S. \quad \langle \texttt{map}(\texttt{x}, S) \rangle \; \texttt{insert}(\texttt{x}, \texttt{k}, \texttt{v}) \; \langle \texttt{map}(\texttt{x}, S[\texttt{k} \mapsto \texttt{v}]) \wedge ((\texttt{k} \in S \wedge \texttt{ret} = S(\texttt{k})) \vee (\texttt{k} \notin S \wedge \texttt{ret} = \texttt{null})) \rangle$$

$$\vdash \mathbb{V}S. \quad \langle \texttt{map}(\texttt{x}, S) \rangle \; \texttt{remove}(\texttt{x}, \texttt{k}) \; \langle \texttt{map}(\texttt{x}, S \backslash \{\texttt{k} \mapsto \_\}) \wedge ((\texttt{k} \in S \wedge \texttt{ret} = S(\texttt{k})) \vee (\texttt{k} \notin S \wedge \texttt{ret} = \texttt{null})) \rangle$$

$$\vdash \mathbb{V}S. \quad \langle \texttt{map}(\texttt{x}, S) \rangle \; \texttt{search}(\texttt{x}, \texttt{k}) \; \langle \texttt{map}(\texttt{x}, S) \wedge ((\texttt{k} \in S \wedge \texttt{ret} = S(\texttt{k})) \vee (\texttt{k} \notin S \wedge \texttt{ret} = \texttt{null})) \rangle$$

Figure 3.6: Map-based specifications for concurrent map

$$\texttt{map}(x, S) \stackrel{\text{def}}{=} \exists a. \, \mathbf{Map}_a(x, S) * [\text{MAP}]_a$$
$$[\text{MAP}] : \forall S, k, v. \quad S \rightsquigarrow S[k \mapsto v] \quad S \rightsquigarrow S \backslash \{k \mapsto v\}$$

Figure 3.7: Definition `map` and actions

boundary of map module, which is easier to understand for clients and implementations. Therefore to specify a general used map, we think map-based style is slightly better.

equivalent specification

To show key-value specifications are equivalent with map-based ones, we only need to show that we can build one from another and vice versa. We can trivially interpret region type $\mathbf{Key}(x, k, v)$ to a $\texttt{map}(x, S)$. However using `key` to interpret $\mathbf{Map}(x, S)$ need to collect all keys.

$$I(\mathbf{Key}(x, k, v)) \stackrel{\text{def}}{=} \exists S. \, \texttt{map}(x, S) \wedge ((v \neq \epsilon \wedge \{k \mapsto v\} \subseteq S) \vee (v = \epsilon \wedge k \notin S))$$
$$I(\mathbf{Map}(x, S)) \stackrel{\text{def}}{=} \circledast_{\{k \mapsto v\} \subseteq S} \texttt{key}(x, k, v) * \circledast_{k \notin S} \texttt{key}(x, k, \epsilon)$$

Figure 3.8: Build key-value abstract predicate by map-based abstract predicate and vice versa

Comparison

Even though we have shown the two styles are equivalent by building one from another, there are some case that key-value style is better idea, but we also believe in most cases, map-based style is the better one. Parallelised Map/Reduce algorithm [8] is a example that key-value style is a better specification in term of easier understanding. Let takes the problem of counting the number of occurrences of each work in large collection as a example [8]. There are many mappers, in other word we can think about there are many threads doing map operation. Mappers generate many intermediate maps, where here intermediate maps should includes words with a count number 1, i.e. $\{\texttt{apple} \mapsto 1, \texttt{computer} \mapsto 1, \dots\}$. Then there are certain number of reducers, which sums up occurrence of each word by function `reduce` (Figure 3.9). Typically each reducers only process certain number of keys, thus in this situation key-value specification is better idea.

Figure 3.9 is simple stylish proof of reducer, where `itr` is iterator who automatically goes through all separate intermediate maps and only get those pairs with key `k`. $\texttt{iterator}(L)$ is a abstract predicate and L includes those information that this iterator can access. Notice, $\texttt{iterator}(L)$

$\{\ \texttt{itr} \mapsto \texttt{iterator}(L) \wedge L = [\texttt{key}(\_, \texttt{k}, 1), \texttt{key}(\_, \texttt{k}, 1), \ldots]\ \}$
`function reduce(k, itr) {`
  `v := 0;`
  `while (itr.hasNext()) {`
$\{\ \exists L', L''.\, \texttt{itr} \mapsto \texttt{iterator}(L'') \wedge L = L' +\!\!+ L'' \wedge \texttt{v} = |L'| \wedge L'' \neq []\ \}$
    `cur := itr.next();`
$\left\{\begin{array}{l} \exists L', L''.\, \texttt{itr} \mapsto \texttt{iterator}(L'') * \texttt{cur} \mapsto \texttt{key}(\_, \texttt{k}, 1) \wedge \\ L = L' +\!\!+ L'' \wedge \texttt{v} = |L'| - 1 \wedge \texttt{last}(L') = \texttt{cur} \end{array}\right\}$
    `v := v + cur.value;`
$\left\{\begin{array}{l} \exists L', L''.\, \texttt{itr} \mapsto \texttt{iterator}(L'') * \texttt{cur} \mapsto \texttt{key}(\_, \texttt{k}, 1) \wedge \\ L = L' +\!\!+ L'' \wedge \texttt{v} = |L'| \wedge \texttt{last}(L') = \texttt{cur} \end{array}\right\}$
  `}`
$\{\ \texttt{itr} \mapsto \texttt{iterator}([]) \wedge \texttt{v} = |L|\ \}$
  `return v;`
`}`
$\{\ \texttt{itr} \mapsto \texttt{iterator}([]) \wedge \texttt{ret} = |L|\ \}$

Figure 3.9: Proof of function `reduce` for counting word occurrences

includes information from logically or even possibly physically separate map and all this iterator can observe is part of map, i.e. some key-value pairs. Therefore a key-value style is rather reasonable as respect to reason about function `reduce`.

However we think in many other cases map-based style is a better, especially in situation that map is provided as a general module to clients. First, typically clients choose to synchronise the whole map instead of separate key-value pairs. Languages, such as Java and Python, usually provide two versions of concurrent map. one is a thread-safe concurrent map such as `ConcurrentMap` in Java, which is similar with what we discuss in this thesis. Another is data consistency concurrent map such as `Collections.synchronizedMap` in Java, which is a map with some map level synchronisation. Language typically does not provide key-value level synchronisation module.

In term of synchronised map, one can think that a synchronised map is combination of lock module and map module. To use this synchronised map all operations requires to hold this lock first. We can simply build such map with the map-based specifications we have now, by wrapping the abstract predicate `map` by new region $\mathbf{WrapMap}(x, S)$ and with more restrain permission in higher level. Notice, if the second parameter of region is ϵ, i.e. $\mathbf{WrapMap}(x, \epsilon)$, it means that it is in a intermediate state, which logically cannot be observed by other threads except the thread who holds the lock. [WRAP_MAP] is bound with no action but is for the purpose of uniforming how we use guard algebra. [WRAP_MAP_UNI] is a unique guard which is bound with actions. In the rest of thesis, we typically define two guard, one is duplicatable and another is unique, and they are connected by a witness [LOCK].

With a new level of abstraction, we now are able to combine a lock module. The lock module is a general module defined in §4.1. Intuitively, this lock module is also quantify by a region id a, which build a logically relationship between lock and other module. Then the thread who successfully lock this lock, will have the witness $[\text{LOCK}]_a$, and to unlock this lock it is also need to return witness $[\text{LOCK}]_a$. At last, at the top level which is what clients can see, we mask the fact that there is lock state but only reveal S.

We now can use `synMap` to reason about functions that provide data consistency, similar to the behaviour of `Collections.synchronizedMap`. For example, assumes this synchronised map has a

$$I(\mathbf{WrapMap}_a(x,S)) \stackrel{\text{def}}{=} \texttt{map}(x,S)$$
$$I(\mathbf{WrapMap}_a(x,\epsilon)) \stackrel{\text{def}}{=} \exists S.\, \texttt{map}(x,S)$$
$$[\text{WRAP\_MAP}]: \quad \varnothing$$
$$[\text{WRAP\_MAP\_UNI}]: \quad \forall S.\, S \rightsquigarrow \epsilon \quad \epsilon \rightsquigarrow S$$
$$[\text{WRAP\_MAP}] \Longrightarrow [\text{WRAP\_MAP}] * [\text{WRAP\_MAP}]$$
$$[\text{WRAP\_MAP\_UNI}] * [\text{WRAP\_MAP\_UNI}] \Longrightarrow \texttt{false}$$
$$[\text{WRAP\_MAP}] * [\text{LOCK}] \Longrightarrow [\text{WRAP\_MAP\_UNI}] \Longrightarrow [\text{LOCK}]$$

Figure 3.10: Region $\mathbf{WrapMap}(x,S)$ and its guard

$$\texttt{synMap}(x,S) \stackrel{\text{def}}{=} \exists a, l.\, \mathbf{SynMap}_a(x,l,S) * [\text{SYN\_MAP}]_a$$
$$[\text{SYN\_MAP}]_a : \forall S.\, S \rightsquigarrow S'$$
$$I(\mathbf{SynMap}(x,l,S)) = \texttt{lock}(x.\texttt{lock}, l, a) * (\mathbf{WrapMap}_a(x.\texttt{map}, S) \vee \mathbf{WrapMap}_a(x.\texttt{map}, \epsilon)) * [\text{WRAP\_MAP}]_a$$

Figure 3.11: Synchronised map module

$\Vdash S.$
$\langle\ \texttt{synMap}(\texttt{x}, S) * \texttt{itr} \mapsto \texttt{iterator}(L) \wedge L = [(\texttt{k}_1, \texttt{v}_1), (\texttt{k}_2, \texttt{v}_2), \ldots]\ \rangle$
abstract, quantify a and l
$\langle\ \mathbf{SynMap}_a(\texttt{x}, l, S) * [\text{SYN\_MAP}]_a * \texttt{itr} \mapsto \texttt{iterator}(L)\ \rangle$
make atomic and open region
$a: \ S \rightsquigarrow S' \wedge S' = \texttt{merge}(S, L) \ \vdash$
`function putAll(x, itr) {`
$\{\ a \Rrightarrow \blacklozenge * \texttt{lock}(x.\texttt{lock}, l, a) * (\mathbf{WrapMap}_a(x.\texttt{map}, S) \vee \mathbf{WrapMap}_a(x.\texttt{map}, \epsilon)) * \ \}$
$\{\ [\text{WRAP\_MAP}]_a * \texttt{itr} \mapsto \texttt{iterator}(L)\ \}$
`lock(x.lock);`
$\{\ a \Rrightarrow \blacklozenge * \texttt{lock}(x.\texttt{lock}, 1, a) * \mathbf{WrapMap}_a(x.\texttt{map}, \epsilon) * \ \}$
$\{\ [\text{WRAP\_MAP\_UNI}]_a * \texttt{itr} \mapsto \texttt{iterator}(L)\ \}$
open region
`while (itr.hasNext()) {`
$\{\ \exists S', L', L''.\, a \Rrightarrow \blacklozenge * \texttt{map}(x.\texttt{map}, S') * \texttt{itr} \mapsto \texttt{iterator}(L'') \wedge \ \}$
$\{\ L = L' +\!\!+ L'' \wedge S' = \texttt{merge}(S, L')\ \}$
`cur := itr.next();`
$\{\ \exists S', L', L''.\, a \Rrightarrow \blacklozenge * \texttt{map}(x.\texttt{map}, S') * \texttt{itr} \mapsto \texttt{iterator}(L'') \wedge \ \}$
$\{\ L = L' +\!\!+ [\texttt{cur}] +\!\!+ L'' \wedge S' = \texttt{merge}(S, L')\ \}$
`insert(x.map, cur.key, cur.value);`
$\{\ \exists S', L', L''.\, a \Rrightarrow \blacklozenge * \texttt{map}(x.\texttt{map}, S') * \texttt{itr} \mapsto \texttt{iterator}(L'') \wedge \ \}$
$\{\ L = L' +\!\!+ [\texttt{cur}] +\!\!+ L'' \wedge S' = \texttt{merge}(S, L' +\!\!+ [\texttt{cur}])\ \}$
`}`
// use guard $[\text{WRAP\_MAP\_UNI}]_a$ update ϵ to S'
$\{\ a \Rrightarrow (S, S') * \texttt{lock}(x.\texttt{lock}, 1, a) * \mathbf{WrapMap}_a(x.\texttt{map}, S') * \ \}$
$\{\ [\text{WRAP\_MAP\_UNI}]_a * \texttt{itr} \mapsto \texttt{iterator}([]) \wedge S' = \texttt{merge}(S, L)\ \}$
`}`
$\langle\ \exists S'.\, \mathbf{SynMap}_a(\texttt{x}, l, S) * [\text{SYN\_MAP}]_a * \texttt{itr} \mapsto \texttt{iterator}(L) \wedge S' = \texttt{merge}(S, L)\ \rangle$
$\langle\ \exists S'.\, \texttt{synMap}(\texttt{x}, l, S') * \texttt{itr} \mapsto \texttt{iterator}(L) \wedge S' = \texttt{merge}(S, L)\ \rangle$

Figure 3.12: Proof of `putall`

function call `putAll`, which insert multiple key-value pairs in logically one step. For purpose of simplification, we assume `itr` store the key value pairs.

Apart from the fact that it typically synchronise at map level, for implementation, map-based specifications provide enough information about how this module should looks like, i.e. a map at address x with information S inside. But for key-value pair one, let assumes we give key-value pair abstract predicate and it corresponding specification without any other description to our implementations. Even though in the thesis we always assume that key-value pairs is part of map, but since we only give predicate and the expected specification to our implementations, thus the implementation could be mislead to other possible data structure which is not suitable for concurrent map.

To conclude, key-value specifications and map-based specifications are equivalent. In some cases key-value specifications seem to be easier to understand, such as iterators, but we believe in many other cases map-based style is a better description with a clear boundary of this module, therefore implementation get what want to know from the specification, and clients can easily understand this module and combine this module with other module with less works. So the rest of thesis choose to use map-based specifications from §3.2.

Chapter 4

B^{link}Tree Implementations and Proofs

We have chosen two ambitious examples as implementations of concurrent map. The first one is concurrent B^{link}tree [31]. One motivation is that preview work [6] assume there is no compression, therefore we would like to fulfil that work by adding compression. It is also interested to see that how TaDA works against very industry-liked codes with operations on complex data structure. We are also interested in what do abstractions mean in TaDA and what is the best practice to build the abstractions tower from concrete to the abstraction layer for client.

We will first present a general lock module which is used in many places in this thesis. After we will shortly discuss the insertion and compression algorithm, followed by predicates definition and guards. At last, we will present proofs for the main body of insertion and compression algorithms, whereas the rest part of proofs are included in appendix A.

4.1 Lock Module

First, we present a simple lock module which will be used in this thesis. Abstract predicate `lock` is defined as region and a guard [LOCK]. Guard [LOCK] allows any thread to lock or unlock, but actually only one thread can unlock it, becuase to unlock one must hold the witness [LOCKED].

In addition, our lock module is parameterise by a region id r, and the witness $[\text{LOCKED}]_r$ is under id r instead of id a. We use this mechanism to build the connection between lock module and others and also to keep the modularity of lock. Therefore other module only need to define guard algebra based on witness $[\text{LOCKED}]_r$.

$$\texttt{lock}(x, m, r) \stackrel{\text{def}}{=} \exists a.\, \mathbf{Lock}_a(x, m, r) * [\text{LOCK}]_a$$
$$[\text{LOCK}] : \forall m \in \{0, 1\}\,. \quad 1 \rightsquigarrow 0$$
$$0 \rightsquigarrow 1$$
$$I(\mathbf{Lock}_a(x, m, r)) \stackrel{\text{def}}{=} x \mapsto m \wedge (m = 1 \vee (m = 0 \wedge [\text{LOCKED}]_r))$$
$$[\text{LOCKED}] * [\text{LOCKED}] \Longrightarrow \texttt{false}$$

Figure 4.1: Lock module

4.2 Deleting Pool Module

Another general module used in this thesis is a deleting pool, which collects those heap that are logically deleted and waited for garbage collection. All those heap lets insides deleting pool are guarantee that can not be re-allocated.

DeletingPool(L) is interpreted as collection of $\texttt{dead}(x, type)$, where x should be the address and *type* is a parameter which indicates the type of x. Other module needs to choose a *type* and with abstraction every thing should work. For instance, one can have $\texttt{dead}(x, \texttt{list}) \stackrel{\text{def}}{=} \texttt{list}(x, \_)$, then it can use this module to collect logically deleted list. Guard simply allows that heap go inside but not go out which reflects the fact that they should not be re-allocated.

$$\texttt{deletingPool}(L) \stackrel{\text{def}}{=} \exists a.\, \mathbf{DeletingPool}_a(L) * [\text{DELETE\_POOL}]_a$$
$$[\text{DELETE\_POOL}] : \forall L, x.\, L \rightsquigarrow L \uplus x$$
$$I(\mathbf{DeletingPool}(L)) \stackrel{\text{def}}{=} \circledast_{x \in L} \exists type.\, \texttt{dead}(x, type)$$

Figure 4.2: Deleting pool module

4.3 B^{link}Tree

4.3.1 Data Structure

A B-tree is a balanced search tree with each node stores a certain amount of key value pairs. The number of pairs within one node potential should be between `max`/2 and `max`, where `max` is a constant number. Besides, all the information are only stored in the leaves which is at the bottom of tree and all inner nodes store indexes and pointers to their children. Typically, leaves are also linked in order, then a thread can sequentially read information which are in next leaf without search again.

Kung and Lehman [19] presented a concurrent B-tree algorithm, but it requires to lock more than one node during insertion, whereas Sagiv [31] presented a better concurrent algorithm, where insertion only needs locking at most one node. This algorithm is based on a variation of B-tree, where Sagiv [31] called B^{link}tree. B^{link}tree is a B-tree with inner nodes are also linked in order, and there is prime block which records the address of most left node in each layer.

We will only proof the `insert` algorithm and `compress` algorithm by [31] in this chapter. Since `search` is a function which only reads information, and this function are similar to partial `insert`, therefore we believe it is a trivial proof. Similarly, `delete` only deletes the key value pairs at leaf list but does not do compression, this function is similar to `search` but at the end it will delete a key value pair in a linearizability point.

4.3.2 Insertion Algorithm

Figure 4.3 has presented `insert` algorithm, which is modified from [6, 31]. The algorithm from [6] does not consider compression, thus we first re-write pseudo-code for compression from [31] and then modify the code from [6] to support compression. New `insert` has three process stages, we would like to call them,

```
function insert(x, k, v) {
  stack := newStack();
  pb := getPrimeBlock(x.pb);
  cur := root(pb);
  curn := get(cur);
  goDownAndPushStack;
  level := 1;
  m := k;
  w := v;
  while (true) {
    findNode;
    if (isSafe(curn)) {
      insertIntoSafe;
      return null;
    } else {
      pb := getPrimeBlock(x);
      if (isRoot(pb, cur)) {
        insertIntoUnsafeRoot;
        return null;
      } else {insertIntoUnsafe;}
    }
  }
}
goDownAndPushStack{
  while (isLeaf(curn) = false) {
    if (k < highKey(curn)) {
      push(stack, cur);
    }
    cur := next(curn, k);
    curn := get(cur);
  }
}
insertIntoUnsafe{
  new := new();
  newn := split(curn, m, w, new);
  put(newn, new);
  put(curn, cur);
  unlock(cur);
  w := new;
  m := lowKey(newn);
  level := level + 1;
  if (isEmpty(stack)) {
    pb := getPrimeBlock(x);
    cur := getNodeLevel(pb, level);
  } else { cur := pop(stack); }
}
```

```
findNode{
  found := false;
  lock(cur);
  while (found = false) {
    curn := get(cur);
    if (isDelete(curn)||m < lowKey(curn)) {
      unlock(cur);
      pb := getPrimeBlock(x.pb);
      cur := getNodeLevel((curn, level);
      curn := get(cur);
      lock(cur);
    } else  if (m > highKey(curn)) {
      unlock(cur);
      cur := next(curn, m);
      curn := get(cur);
      lock(cur);
    } else  if (isIn(curn, m)) {
      tv := modifyPair(curn, m, w);
      unlock(cur);
      return v;
    } else {found := true;}
  }
}
insertIntoSafe{
  addPair(curn, m, w);
  put(curn, cur);
  unlock(cur);
}
insertIntoUnsafeRoot{
  new := new();
  newn := split(curn, m, w, new);
  put(newn, new);
  lock(new);
  put(curn, cur);
  y := lowKey(curn);
  t := highKey(curn);
  u := highKey(newn);
  r := new();
  rn := newRoot(y, cur, t, new, u);
  put(rn, r);
  addRoot(pb, r);
  putPrimeBlock(x.pb);
  unlock(cur);
  unlock(new);
}
```

Figure 4.3: Concurrent B^{link}tree `insert` algorithm

- optimistic recording path,
- finding and locking a specific node, and
- modification.

First stage which is the first loop presented by `goDownAndPushStack`, searches and records path without locking any node. `goDownAndPushStack` tries to push those inner nodes which potentially leads to the leaf node containing key `k`. If there is no compression happen, all those inner nodes have been pushed into stack should be always valid. However with compression, the recorded path is not guaranteed always as a valid path. Algorithm only use this path to increase performance, because it is highly possible most part of this path still valid.

After first stage, it modifies B^{link}tree round by round until there is no more new key value pair. In each round, `findNode` finds and locks a node, where algorithm ought to add a new key value pair or modify the old value. `findNode`, instead of optimistic search, will lock node first and check whether this node is valid and is the node needed. If it is not, `findNode` will unlock current node and restart searching. In the end, if `findNode` terminates it will find the right node, and it is guaranteed to be locked, so there is no interference. `findNode` may face three possibilities,

- A node is deleted or temperately inconsistent. When `isDelete(curn)`$||$`m` $<$ `lowKey(curn)`, it means current node has been deleted or temperately inconsistent, because of compression. Therefore `findNode` has to search from left most node again, where the address of left most node should be recorded by prime block.
- A valid node but should go to next node. When `m` $<$ `lowKey(curn)` it means it should be in right of current node, thus algorithm simply read and lock the next node.
- A node including exactly the key. There is a special case for `findNode`, that exactly the same key has been find where it will only happen when it is a leaf. In this case, `findNode` will update the value into new value `v` and return the old value immediately.

After `findNode`, algorithm should hold a node which has been locked, then algorithm will check the node state where there are three possibilities, safe node, unsafe root, and unsafe node. Here "safe" means node's volume is less than the max volume max. So if the node is safe, whether it is leaf or inner node, it is guaranteed to be able to add new key value pair, without changing the structure. If the node is not safe, node will be split and generate new key value pair that should be added into their direct parent node.

The most simple case is adding safe node. `insertIntoSafe` simply adds new key value pair into node, and writes back to concrete node in one atomic file system operation.

If it is an unsafe root node, `insertIntoUnsafeRoot` splits current node and generates new root. Since current thread has already locked the old root and will only release until the end, which means before there is no interference. But algorithm does not lock the new root for the purpose to provide more concurrency. After prime block has been update, the new root become a reachable node but its two children has been lock, so there is actually no interference as well.

If it is a non-root node, `insertIntoUnsafe` splits current node and then generates new key value pair which should be added into its parent node. After splitting, it is when stack is involved. Even though it is possible that the node recorded in stack already been marked as deleted node, but in next round of loop `findNode` will deal with this situation, and what is significant for performance is that it is highly possible this node still stable. however in `insertIntoUnsafe`, it needs to deal with if it reaches a empty stack. In other words, B^{link}tree has grown up at least once. Algorithm

uses `getNodeLevel` function to get the left most node at `level`, and in next round `findNode` will search node by node.

This algorithm assumes that all nodes are stored in file system, thus functions `lock`, `put`, `get` and `new` are assumed functions that provided by file system and assume logically atomic. Apart from those functions, other functions operate on a local copy of node in memory. This algorithm also assumes there is a garbage collection.

4.3.3 Compression Algorithm

```
function compressLevel(x, level, b) {
  pb := getPrimeBlock(x.pb);
  cur := getNodeLevel(pb, level + 1);
  while (cur ≠ null) {
    lock(cur);
    curn := get(cur);
    if (isDelete(curn)) {
      unlock(cur);
      cur := getNodeLevel(pb, level + 1);
    } else if (!hasState(curn, !b)) {
      unlock(cur);
      nextNode(cur);
    } else {
      compressNode;
    }
  }
}
```

```
compressNode{
  one := firstState(curn, !b);
  if (one ≠ null) {
    lock(one); onen := get(one);
    two := nextNode(one);
    if (one = last(curn)) {
      changePointerState(curn, one, b);
      put(cur, curn);
      unlock(cur); unlock(one);
    } else { // two ≠ null
      lock(two); twon := get(two);
      if (isIn(curn, two)) {
        rearrange(onen, twon, curn);
        changePointerState(curn, one, b);
        put(cur, curn); unlock(cur);
        put(one, onen); unlock(one);
        put(two, twon); unlock(two);
      } else {
        unlock(cur); unlock(one); unlock(two);
} } } }
```

Figure 4.4: Concurrent B^{link}tree `insert` algorithm

Compression runs concurrently with insertion and deletion. compression algorithm is first presented by Sagiv [31] in an informal way. We re-write by pseudo-code where function `compressLevel(x, level, b)` ought to compress the whole list at level `level`.

`compressLevel(x, level, b)` compresses the list at level `level`, therefore if function terminates all key value pairs at list `level` + 1 will be marked with value `b`. Since in each level there is at most one compress thread, "compression-and-mark" process is guaranteed to make progress. Each loop inside `compressLevel(x, level, b)` locks a parent node `cur` at `level` + 1 and compresses two adjacent kids `one` and `two` at `level`. Compression could try to compress a node which are just split, so to make sure there is no conflict compression should check whether `two` is reachable from `cur`, and if it is not reachable compression should unlock nodes and wait the splitting finish. In addition, compression writes back parent node and two children nodes in three separate steps which might cause temperately data inconsistency. But this data inconsistency could be detected by checking the min key of second node and max key of first node, thus when searching finds inconsistency it should restart searching.

4.4 Predicate And Guard

At beginning $\mathbf{Map}(x, S)$ is interpreted as abstract presentation of B^{link}tree, i.e. a prime block and main body of tree. Prime block stores the addresses of left-most node in each layer, so the top element of prime block points to the root of B^{link}tree. `Btree` asserts a balanced tree with all data stored only in leaves but all information stored in inner nodes work as indexes to next level. Each node also has a pointer points to next node in the same level. pbL and $addrL$ are abstract state and we will explain later.

$$I(\mathbf{Map}_a(x, S)) \stackrel{\text{def}}{=} \exists pbL, addrL.\, x.\texttt{pb} \mapsto \texttt{primeBlock}(pbL) * \texttt{Btree}(pbL, addrL, S)$$

Figure 4.5: Abstract presents of B^{link}tree

Abstract predicate $x \mapsto \texttt{primeBlock}(pbL)$ is defined as a region with its guard. pbL is an abstract state, a list which includes all the addresses stored inside prime block. The first element of pbL points to left most leaf of B^{link}tree, and the last element points to current root of B^{link}tree. we choose this upside-down way to present prime block because algorithm calls the leaf side as $level = 1$. $\mathbf{PrimeBlock}(x, pbL)$ is interpreted as an abstract predicate `list`. To simplify description, we assume there is `list` module already. Prime block's guard [PRIMEBLOCK] is a duplicatable guard. Even though it is duplicatable, it only describes that threads who hold guard [PRIMEBLOCK] have ability and possibility to modify prime block, and it is also required current root is locked by current thread to really change prime block. Therefore growing or reducing root is actually sequentialised which is what algorithm requires.

Another abstract predicate, $x = \texttt{primeBlock}(pbL)$, is also defined as `list` but this resource is always local resource. Because of garbage collection, $x = \texttt{primeBlock}(pbL)$ is guarantee to be memory safe.

$$x \mapsto \texttt{primeBlock}(pbL) \stackrel{\text{def}}{=} \exists a.\, \mathbf{PrimeBlock}_a(x, pbL) * [\text{PRIMEBLOCK}]_a$$

$$I(\mathbf{PrimeBlock}_a(x, pbL)) \stackrel{\text{def}}{=} \texttt{list}(x, pbL)$$

$$\begin{aligned}[\text{PRIMEBLOCK}]: \quad & \forall pbL, new\_root.\, \texttt{last}(pbL) \mapsto \texttt{node}(1, \_, \_, \_, \_, \_) \\ & pbL \rightsquigarrow pbL +\!\!+ [new\_root] \\ & pbL \rightsquigarrow pbL \backslash [\texttt{last}(pbL)]\end{aligned}$$

$$[\text{PRIMEBLOCK}] \Longrightarrow [\text{PRIMEBLOCK}] * [\text{PRIMEBLOCK}]$$

$$x = \texttt{primeBlock}(pbL) \stackrel{\text{def}}{=} \texttt{list}(x, pbL)$$

Figure 4.6: Prime block and guard

Main body of B^{link}tree are organised as lists in each layer where at bottom it is a leaf list and above are inner node lists. We use `Btree` which is a predicate or normal predicate, to present the main body. `Btree` is recursively defined until leaf list. Intuitively, $addrL$ is a abstract state that includes all the nodes' addresses, kvL is a abstract state that includes all key value pairs in a layer, and pbL helps to find the first address for a layer. In detail, last element of pbL must be the first address of current list $addrL$, which means that left most node in a layer should be stored in prime block. It also requires all addresses stored in current layer must be the subset or equal to the next layer, in other word $kvL\lfloor_2 \subseteq addrL'$. Here, it guarantees that parent list should only point to nodes in their direct child list, i.e. cannot point to resource outside B^{link}tree or other layer of B^{link}tree. We should point out that it is subset instead of equal, because some nodes might already be added into child list but the corresponding index is not added into the direct parent yet. In this situation,

those nodes can be access through traversal from prime block but are not reachable directly from their parent node.

$$\begin{aligned}\texttt{Btree}(pbL, addrL, S) \equiv\ & \exists x, pbL', kvL, addrL'.\, x = \texttt{first}(addrL) \wedge (pbL = pbL' +\!\!+ [x] \wedge \\ & \texttt{innerList}(addrL, kvL) * \texttt{Btree}(pbL', addrL', S) \wedge kvL\lfloor_2 \subseteq addrL') \\ & \vee (pbL = [x] \wedge \texttt{leafList}(addrL, S))\end{aligned}$$

Figure 4.7: Main body of B^{link}tree

Leaf list and inner node list are defined and generalised by one region $\mathbf{BTreeList}(addrL, pbL, type)$, where the last parameter distinguish inner list or leaf list. If the last parameter equal to 0 it is a inner node list, otherwise it is a leaf list. The first parameter $addrL$ includes all addresses within this list, and second parameter kvL records all key value pairs within this list.

Guard [BTREE_LIST] allows adding new key value pair, modifying a specific key value pair, and adding a new node. Actions bound with leaf list are sightly different from those bound with inner list. For leaf list where $type = 1$, this guard allows adding a new key value pair and re-mapping a key to a new value. In addition, it allows adding a new key value pair and a new node in the same time. At last it allows removing one key value pair. However, for inner list where $type = 0$, the guard allows almost the same actions as $type = 1$, yet it cannot re-map key to a new value but allows changing key because compression could trigger changing key.

$$\begin{aligned}\texttt{innerList}(addrL, kvL) &\stackrel{\text{def}}{=} \exists a.\, \mathbf{BTreeList}_a(addrL, kvL, 0) * [\text{BTREE\_LIST}]_a \\ \texttt{leafList}(addrL, kvL) &\stackrel{\text{def}}{=} \exists a.\, \mathbf{BTreeList}_a(addrL, kvL, 1) * [\text{BTREE\_LIST}]_a \\ [\text{BTREE\_LIST}]: &\ type = 1 \wedge \forall addrL, x, kvL, k, v. \\ &\ addrL, kvL \rightsquigarrow addrL, kvL[k \mapsto v] \\ &\ addrL, kvL \rightsquigarrow addrL \uplus [x], kvL \uplus [(k, v)] \\ &\ addrL, kvL \rightsquigarrow addrL, kvL \backslash [(k, v)] \\ [\text{BTREE\_LIST}]: &\ type = 0 \wedge \forall addrL, x, kvL, k, v. \\ &\ addrL, kvL \rightsquigarrow addrL, kvL \backslash [(k, v)] \uplus [(k', v)] \\ &\ addrL, kvL \rightsquigarrow addrL, kvL \uplus [(k, v)] \\ &\ addrL, kvL \rightsquigarrow addrL \uplus [x], kvL \uplus [(k, v)] \\ &\ addrL, kvL \rightsquigarrow addrL, kvL \backslash [(k, v)]\end{aligned}$$

Figure 4.8: Definition of list and guards

There are another two abstract predicates which are used to assert pre-condition of compression function. Instead of guard [BTREE_LIST] they also include a unique guard [BTREE_LIST_UNI] with compression ability. This unique guard allows list to compress node, in other words, removing one node from this list but preserve all key value pairs. Since compression algorithm requires that there is at most one compression in each layer, thus compression ability should bind with some unique guard.

$$\begin{aligned}\texttt{innerListCompress}(addrL, kvL) &\stackrel{\text{def}}{=} \exists a.\, \mathbf{BTreeList}_a(addrL, kvL, 0) * [\text{BTREE\_LIST}]_a * \\ &\quad [\text{BTREE\_LIST\_UNI}]_a \\ \texttt{leafListCompress}(addrL, kvL) &\stackrel{\text{def}}{=} \exists a.\, \mathbf{BTreeList}_a(addrL, kvL, 1) * [\text{BTREE\_LIST}]_a * \\ &\quad [\text{BTREE\_LIST\_UNI}]_a \\ [\text{BTREE\_LIST\_UNI}]: &\ \forall addrL, kvL.\, addrL, kvL \rightsquigarrow addrL \backslash [x], kvL\end{aligned}$$

Figure 4.9: List predicates used to assert compression

Guard algebra between duplicatable guard [BTREE_LIST] and unique guard [BTREE_LIST_UNI] is rather simple. Those two guards are linked with a witness [LOCKED], and unique guard [BTREE_LIST_UNI] can reduce to witness [LOCKED]. This witness [LOCKED] presumably may be held by a specially thread or by some access control mechanism.

$$[\text{BTREE\_LIST}] \Longrightarrow [\text{BTREE\_LIST}] * [\text{BTREE\_LIST}]$$
$$[\text{BTREE\_LIST\_UNI}] * [\text{BTREE\_LIST\_UNI}] \Longrightarrow \texttt{false}$$
$$[\text{BTREE\_LIST}] * [\text{LOCKED}] \Longrightarrow [\text{BTREE\_LIST\_UNI}] \Longrightarrow [\text{LOCKED}]$$

Figure 4.10: Guard algebra for B^{link}tree list

The third level of abstract is node, where $\mathbf{BTreeList}(addrL, kvL, type)$ is interpreted as predicate `linkedNodeList` that includes all nodes in $addrL$ and each node store part of continuous key value pairs in kvL. `linkedNodeList` is equivalent to a `linkedNodeLseg` with the end pointer $end = \texttt{null}$ where `linkedNodeLseg` is a recursive predicate, that is equivalent to the first node and the rest list segment until the `emp` case.

To properly describe the relation between list and node, we use some auxiliary predicates. Auxiliary predicate `nextNode` says that if there is still another node after then nxt should point to that node, otherwise it points to the *end*. `maxKey` asserts that if there is a next node, value of mxk should equal to the first key of next node, or it equal to $+\infty$. The last auxiliary predicate `keyValue` describes a situation where compression causes temperate data inconsistency. It says that if node is unlock, $l = 0$, node's information should be consistent with abstract state kvL, where $kvmL$ is key value pairs and each pair has a extra marking field for compression. However if node is lock, $l = 1$, node's data could be inconsistent with abstract state kvL but part of them should be consistent which is guarantee to be prefix part or suffix part.

$$I(\mathbf{BTreeList}_a(addrL, kvL, type)) \stackrel{\text{def}}{=} \texttt{linkedNodeList}(addrL, kvL, type)$$
$$\texttt{linkedNodeList}(addrL, kvL, type) \equiv \texttt{linkedNodeLseg}(addrL, kvL, type, \texttt{null})$$
$$\begin{aligned}\texttt{linkedNodeLseg}(addrL, kvL, type, end) \equiv\ & (addrL = [] \wedge \texttt{emp}) \vee (\exists x, addrL', kvmL, mxk, kvL',\\ & kvL'', nxt, l.\, x \mapsto \texttt{node}(l, type, kvmL, mxk, nxt, 0) *\\ & \texttt{lseg}(addrL', kvL'') \wedge addrL = [x] +\!\!+ addrL' \wedge\\ & kvL = kvL' +\!\!+ kvL'' \wedge \texttt{maxKey} \wedge \texttt{nextNode} \wedge \texttt{keyValue})\end{aligned}$$
$$\begin{aligned}\texttt{nextNode} \equiv\ & (\texttt{first}(addrL') = \texttt{null} \wedge nxt = end) \vee\\ & (\texttt{first}(addrL') \neq \texttt{null} \wedge nxt = \texttt{first}(addrL'))\end{aligned}$$
$$\texttt{maxKey} \equiv (kvL'' \neq [] \wedge mxk = \texttt{first}(kvL'')\downarrow_1) \vee (kvL'' = [] \wedge mxk = +\infty)$$
$$\begin{aligned}\texttt{keyValue} \equiv\ & (l = 0 \wedge kvmL\downarrow_{(1,2)} = kvL) \vee (l = 1 \wedge (kvmL\downarrow_{(1,2)} = \texttt{prefix}(kvL') \vee\\ & kvmL\downarrow_{(1,2)} = \texttt{surfix}(kvL') \vee \texttt{prefix}(kvmL)\downarrow_{(1,2)} = kvL' \vee\\ & \texttt{surfix}(kvmL)\downarrow_{(1,2)} = kvL'))\end{aligned}$$
$$\begin{aligned}\texttt{prefix}(L) = L' \quad & \text{where } \exists L''.\, L = L' +\!\!+ L''\\ \texttt{surfix}(L) = L' \quad & \text{where } \exists L''.\, L = L'' +\!\!+ L'\end{aligned}$$

Figure 4.11: Interpretation of list and guards

Abstract predicate for node is defined as a lock, a region and a guard $[\text{NODE}]_a$. Guard [NODE] has no bound ability but only for the purpose of uniforming style of guard algebra. Because once the lock has been locked, we will get a witness [LOCKED], and then it is able to combine with guard [NODE] and transfer to guard [NODE_UNI] with some real updating abilities.

Another guard [NODE_UNI] is an unique guard which grants the thread some abilities to modify node. This guard allows changing delete bit only from 0 to 1, which means a node can be deleted

$$x \mapsto \texttt{node}(l, type, kvmL, mxk, nxt, delete) \stackrel{\text{def}}{=} \exists a.\, \texttt{lock}(x.\texttt{lock}, l, a) * [\text{NODE}]_a * \mathbf{Node}_a(x.\texttt{data}, type, kvmL, mxk, nxt, delete) *$$

$$[\text{NODE\_UNI}]: \quad \forall kvmL, mxk, nxt. \; kvmL, mxk, nxt \rightsquigarrow kvmL', mxk', nxt' \quad \forall delete \in \{0,1\}\,.\, 0 \rightsquigarrow 1$$

where,

#1	$mxk = mxk' \wedge nxt = nxt' \wedge kvmL' = kvmL \uplus [(k, v, m)]$
#2	$mxk = mxk' \wedge nxt = nxt' \wedge kvmL' = kvmL \backslash [(k, v, m)]$
#3	$mxk' = \texttt{first}(kvmL')\downarrow_1 \wedge nxt' \mapsto \texttt{node}(\_, \_, kvmL'', mxk, nxt, \_) \wedge kvmL' +\!\!+ kvmL'' = kvmL \uplus [(\_, \_, \_)]$
#4	$mxk' = \texttt{first}(kvmL')\downarrow_1 \wedge nxt \mapsto \texttt{node}(\_, \_, kvmL'', mxk', nxt', \_) \wedge kvmL' = kvmL +\!\!+ kvmL''$
#5	$\exists kvL_p.\, mxk = mxk' \wedge nxt = nxt' \wedge (k, x) \in kvL_p \wedge \texttt{first}(kvmL')\downarrow_1 = k_1$
#6	$\exists kvL_p.\, mxk' = k \wedge nxt = nxt' \wedge [(\_, x), (k, \_)] \subseteq kvL_p$

Figure 4.12: Node and guards

but cannot re-join. [NODE_UNI] also allows changing key value pairs, max key, and the pointer to next node in a certain way. More specifically, there are six possible scenarios. #1 allows adding one key value pair into node without splitting. #2 allows deleting a key value pair which represents the most simple deleting case. #3 allows current node to drop the tail part to next node and update the max key, only if the next node already contains the tail part of current node. This is the case where splitting happens. #4 allows that current node merges the information of its next node and updates max key as well. #5 allows that because of rearrangement, the direct parent expects current node has the key value pairs starting at key k, therefore this updating ability is to preserve the consistency. Here, quantified kvL_p presents the abstract state of parent list. At last #6 allows that parent node expects current node has pairs before key k and this action is to preserve this expectation.

Guard algebra is defined very similar as before, with one duplicatable guard and one unique guard. Those two guards is connected by a witness [LOCKED] where it is defined within `lock` module in §4.1.

$$[\text{NODE}] \Longrightarrow [\text{NODE}] * [\text{NODE}]$$
$$[\text{NODE\_UNI}] * [\text{NODE\_UNI}] \Longrightarrow \texttt{false}$$
$$[\text{NODE}] * [\text{LOCKED}] \Longrightarrow [\text{NODE\_UNI}] \Longrightarrow [\text{LOCKED}]$$

Figure 4.13: Guard algebra of node

$\mathbf{Node}_a(x, type, kvmL, mxk, nxt, delete)$ is interpreted as a very naive way, since it is a frame in file system and the algorithm assume all operations among file system are correct. There are one extra property we care which is that all keys should be in order.

$$I(\mathbf{Node}_a(x, type, kvmL, mxk, nxt, delete)) \stackrel{\text{def}}{=} x \mapsto type, kvmL, mxk, nxt, delete \wedge \bigwedge_{1 \leqslant i \leqslant |kvmL|} \texttt{at}(kvmL, i)\downarrow_1 \leqslant mxk$$
$$x = \texttt{node}(l, type, kvmL, mxk, nxt, delete) \stackrel{\text{def}}{=} x \mapsto l, type, kvmL, mxk, nxt, delete$$

Figure 4.14: Interpretation of node

Similarly, abstract predicate $x = \texttt{node}(l, type, kvmL, mxk, nxt, delete)$ is a snapshot of a node and

completely local resource. We also assume that it is automatically guaranteed to be memory safe.

4.5 Auxiliary Function Specification

The section are specifications of some auxiliary functions, we will particularly discuss some complicated specifications but leave the rest in appendix A.1. In this section, $\mathcal{P}$ denotes node's parameters except locking state, in other words $\mathcal{P} \equiv type, kvmL, mxk, nxt, delete$.

Figure 4.15 presents those functions that are logically atomic. Algorithm uses blocking style `lock` function instead of non-blocking one, thus its pre-condition says environment is allowed to keep changing locking state, but there is one point when current thread will successfully lock this node. `get` and `put` are functions that only deal with data, i.e. these functions do not influence lock state. `get` takes a snapshot of node state in a certain moment, while `put` writes back information in a linearizability point. Similar for prime block, `getPrimeBlock` takes the most recent snapshot and `putPrimeBlock` updates prime block in a linearizability point.

$$
\begin{array}{rcl}
\forall l, \mathcal{P}. \left\langle \mathtt{x} \mapsto \mathtt{node}(l, \mathcal{P}) \right\rangle & \mathtt{lock}(\mathtt{x}) & \left\langle \mathtt{x} \mapsto \mathtt{node}(1, \mathcal{P}) \right\rangle \\
\left\langle \mathtt{x} \mapsto \mathtt{node}(1, \mathcal{P}) \right\rangle & \mathtt{unlock}(\mathtt{x}) & \left\langle \mathtt{x} \mapsto \mathtt{node}(0, \mathcal{P}) \right\rangle \\
\forall l, \mathcal{P}. \left\langle \mathtt{x} \mapsto \mathtt{node}(l, \mathcal{P}) \right\rangle & \mathtt{get}(\mathtt{x}) & \left\langle \begin{array}{l} \mathtt{x} \mapsto \mathtt{node}(l, \mathcal{P}) * \\ \mathtt{ret} = \mathtt{node}(l, \mathcal{P}) \end{array} \right\rangle \\
\forall l, \mathcal{P}. \left\langle \begin{array}{l} \mathtt{x} \mapsto \mathtt{node}(l, \mathcal{P}) * \\ \mathtt{n} = \mathtt{node}(l', \mathcal{P}') \end{array} \right\rangle & \mathtt{put}(\mathtt{x}, \mathtt{n}) & \left\langle \begin{array}{l} \mathtt{x} \mapsto \mathtt{node}(l, \mathcal{P}') * \\ \mathtt{n} = \mathtt{node}(l', \mathcal{P}') \end{array} \right\rangle \\
\forall pbL. \left\langle \mathtt{x} \mapsto \mathtt{primeBlock}(pbL) \right\rangle & \mathtt{getPrimeBlock}(\mathtt{x}) & \left\langle \begin{array}{l} \mathtt{x} \mapsto \mathtt{primeBlock}(pbL) * \\ \mathtt{ret} = \mathtt{primeBlock}(pbL) \end{array} \right\rangle \\
\forall pbL. \left\langle \begin{array}{l} \mathtt{x} \mapsto \mathtt{primeBlock}(pbL) * \\ \mathtt{y} = \mathtt{primeBlock}(pbL') \end{array} \right\rangle & \mathtt{putPrimeBlock}(\mathtt{x}, \mathtt{y}) & \left\langle \begin{array}{l} \mathtt{x} \mapsto \mathtt{primeBlock}(pbL') * \\ \mathtt{y} = \mathtt{primeBlock}(pbL') \end{array} \right\rangle
\end{array}
$$

Figure 4.15: atomic specifications for auxiliary functions

There are other few functions we would like to explain more, the first one is `next` shown in figure 4.16. To simplify present specification, we imagine the max key value is also part of key value and marker pairs. Then post-condition says that if current node is inner node, $type = 0$, and key `k` is between two index values, the function will return a pointer which will go down a level. Otherwise the function will return the pointer to the direct right next node, i.e. it is the case where $type = 0 \wedge \mathtt{k} \geqslant mxk$ or $type = 1$.

$$
\vdash \left\{ \mathtt{x} = \mathtt{node}(\_, \_, \_, \_, \_, \_) \right\} \ \mathtt{next}(\mathtt{x}, \mathtt{k}) \ \left\{ \begin{array}{l} \mathtt{x} = \mathtt{node}(\_, type, kvmL, mxk, nxt, \_) \wedge \\ \exists kvmL' = kvmL +\!\!+ [(mxk, \_, \_)]. ((type = 0 \wedge \exists i. \\ \mathtt{at}(kvmL', i)\lfloor_1 \leqslant \mathtt{k} < \mathtt{at}(kvmL', i+1)\lfloor_1 \wedge \\ \mathtt{ret} = \mathtt{at}(kvmL, i)\lfloor_2) \vee (\mathtt{k} \geqslant mxk \wedge \mathtt{ret} = nxt)) \end{array} \right\}
$$

Figure 4.16: specification of $\mathtt{next}(\mathtt{x}, \mathtt{k})$

`split` shown in figure 4.17 only splits a local node copy `x` instead of splitting the shared node directly, and it also presumes there is a empty node at address `new` which should be the new directly right node. Function ought to split node into two parts and preserve the requirement that each node has at least max/2 key value pairs. After splitting node, insertion algorithm uses `put` to write back both local node copies to the real shared nodes.

In details, pre-condition of `split` requires that node is already in maximum volume and the new key `k` and value `v` are not inside $kvmL$. Post-condition says that node `x` has split half of key value

pairs to a new node `ret`, and `x`'s direct next node has been updated to `new`. This function also has duty to insert new key `k` and value `v` into either `x` or `ret`. It is assume that caller of this algorithm has the responsibility to write back local node copy `ret` to the shared node `new`.

$$\left\{\begin{array}{l}\mathtt{x} = \mathtt{node}(\_, type, kvmL, mxk, nxt, \_) \wedge \\ |kvmL| = \mathtt{max} \wedge (\mathtt{k}, \mathtt{v}, \_) \notin kvmL\end{array}\right\}$$
$$\mathtt{split}(\mathtt{x}, \mathtt{k}, \mathtt{v}, \mathtt{new})$$
$$\vdash \left\{\begin{array}{l}\mathtt{x} = \mathtt{node}(\_, type, kvmL_1, \mathtt{first}(kvmL_2)\lfloor_1, \mathtt{new}, \_) * \\ \mathtt{ret} = \mathtt{node}(0, type, kvmL_2, mxk, nxt, 0) \wedge \\ kvmL_1 +\!\!+ kvmL_2 = kvmL \uplus [(\mathtt{k}, \mathtt{v}, \mathtt{false})] \wedge \\ |kvmL_1| \geqslant \mathtt{max}/2 \wedge |kvmL_2| \geqslant \mathtt{max}/2\end{array}\right\}$$

Figure 4.17: specification of $\mathtt{split}(\mathtt{x}, \mathtt{k}, \mathtt{v}, \mathtt{new})$

Another relatively complicated function is `rearrange` shown in figure 4.18, which is used by compression. This function re-arranges two adjacent nodes and their direct parent node, yet function still only works on local copies instead of shared nodes themselves. Intuitively, it either merge two nodes into one node or move part of data from one node to another node, besides it changes the parents node as necessary.

Pre-condition says that there are local node copies, `xn`, `yn`, and `pn`. It also requires that the next node of `xn` should be inside parent node. Because this function could delete value `y` from parent node, however if `y` is reachable from `xn` but not reachable from parent node `pn` it means that other thread might during process to add `y` into parent. In this case, this function could cause conflict with adding node process, so it requires that `y` must be already added into parent node.

Post-condition asserts two possibilities. If two nodes has no enough pairs, i.e. $|kvmL_1 +\!\!+ kvmL_2| < \mathtt{max}$, function merges `yn` into `xn` and update `xn`'s max key and next node pointer. Address `y` is also deleted from parent node and corresponding key as well. However this function only marks `yn` as deleting node but does not change any information inside. Another case is that if two nodes has enough pairs, $|kvmL_1 +\!\!+ kvmL_2| \geqslant$ max, function only adjusts the information in each node to make sure each node contain at least $\mathtt{max}/2$ pairs and also changes the max key of `xn` and the key for `y` in parent node.

$$\left\{\begin{array}{l}\mathtt{xn} = \mathtt{node}(\_, \_, kvmL_1, mxk_1, \mathtt{y}, \_) * \\ \mathtt{yn} = \mathtt{node}(\_, \_, kvmL_2, mxk_2, nxt, \_) * \\ \mathtt{pn} = \mathtt{node}(\_, \_, kvmL, \_, \_, \_) \wedge (\_, \mathtt{y}, \_) \in kvmL\end{array}\right\}$$
$$\mathtt{rearrange}(\mathtt{xn}, \mathtt{yn}, \mathtt{p}, \mathtt{x}, \mathtt{y})$$
$$\vdash \left\{\begin{array}{l}\exists kvmL'_1, kvmL'_2, kvmL', mxk'_1, nxt', delete. \\ \mathtt{pn} = \mathtt{node}(\_, \_, kvmL', \_, \_, \_) * \mathtt{xn} = \mathtt{node}(\_, \_, kvmL'_1, mxk'_1, nxt', \_) * \\ \mathtt{yn} = \mathtt{node}(\_, \_, kvmL'_2, mxk_2, nxt, delete) \wedge \\ ((|kvmL_1 +\!\!+ kvmL_2| < \max \wedge kvmL'_1 = kvmL_1 +\!\!+ kvmL_2 \wedge \\ kvmL'_2 = kvmL_2 \wedge mxk_1 = mxk_2 \wedge delete = 1 \wedge nxt' = nxt \wedge \\ kvmL' = kvmL \backslash [(\_, \mathtt{y}, \_)]) \vee (|kvmL_1 +\!\!+ kvmL_2| = \max \wedge \\ kvmL_1 +\!\!+ kvmL_2 = kvmL'_1 +\!\!+ kvmL'_2 \wedge |kvmL'_1| \geqslant \max/2 \wedge \\ |kvmL'_2| \geqslant \max/2 \wedge mxk'_1 = \mathtt{first}(kvmL'_2)\lfloor_1 \wedge delete = 0 \wedge \\ nxt' = \mathtt{y} \wedge kvmL' = kvmL \backslash [(\_, \mathtt{y}, \_)] \uplus [(\mathtt{first}(kvmL'_2)\lfloor_1, \mathtt{y}, \_)]))\end{array}\right\}$$

Figure 4.18: specification for $\mathtt{rearrange}(\mathtt{xn}, \mathtt{yn}, \mathtt{p}, \mathtt{x}, \mathtt{y})$

4.6 Proof

4.6.1 Proof of insertion algorithm

We state at presenting some auxiliary predicates to simplify presentation of proofs. First to distinguish shared and local resource, we use $x^{\blacktriangle}$ to explicitly represent local resource. $x^{\bullet}$ is used to represent the state of x after updating by current thread for the purpose of a readable proof.

$$\begin{aligned} x \mapsto \texttt{node}(\_,\_,\_,\_,nxt,delete)^{\alpha}_{level} \equiv\ & x \mapsto \texttt{node}(\_,\_,\_,nxt,delete) \wedge x \in addrL_{level} \wedge \\ & nxt \in addrL_{level} \wedge delete = 0 \\ xn = \texttt{node}(\_,\_,\_,\_,nxt,\_)_{level} \equiv\ & xn = \texttt{node}(\_,\_,\_,\_,nxt,\_)^{\blacktriangle} \wedge nxt \in addrL_{level} \\ (x,xn) \mapsto \texttt{node}(l,\mathcal{P})^{\alpha}_{level} \equiv\ & x \mapsto \texttt{node}(l,\mathcal{P})^{\alpha}_{level} * xn = \texttt{node}(l,\mathcal{P})_{level} \end{aligned}$$

Figure 4.19: Node auxiliary predicate

$x \mapsto \texttt{node}(\_,\_,\_,\_,nxt,delete)_{level}$ with a number *level* as subscript means that this shared node is one node in list at *level*. The raise α could be a local resource mark, i.e. ▲ or nothing. Basically the new allocated node in file system will be local resource before it has been linked into B^{link}tree. Since xn is the local copy, $xn = \texttt{node}(\_,\_,\_,\_,nxt,\_)_{level}$ only asserts that the next node, i.e. nxt is a node in list at *level*. At the end, we use $(x,xn) \mapsto \texttt{node}(l,\mathcal{P})^{\alpha}_{level}$ to assert a shared node and its local copy. The shared node and local copy are completely disjoint resource, however algorithm logically links these two resources.

$$\texttt{untouchedList}(level) \equiv \circledast_{level \leqslant i \leqslant n} \texttt{list}(addrL_i, kvL_i, type_i)$$

Figure 4.20: List auxiliary predicate

`untouchedList` is a short-hand predicate of list from *level* to n, where n is quantified as the current length of snapshot of prime block, i.e. $n = |pbL'|$. Intuitively, algorithm firstly updates the leaf list and then goes upward, thus the parameter of this predicate *level* will keep increasing until n.

$$\begin{aligned} \texttt{stackProperty}(level) \equiv\ & (level \geqslant n \wedge \texttt{stack}(\texttt{stack},[])^{\blacktriangle}) \vee (level < n \wedge \exists skL. \\ & \texttt{stack}(\texttt{stack},skL)^{\blacktriangle} \wedge \textstyle\bigwedge_{1 \leqslant i \leqslant |kvL|}((\texttt{at}(skL,i) \mapsto \texttt{node}(\_,\_,\_,\_,\_,0) \wedge \\ & \texttt{at}(skL,i) \in addrL_{i+level}) \vee (\texttt{at}(skL,i) \mapsto \texttt{node}(\_,\_,\_,\_,\_,1)))) \end{aligned}$$

Figure 4.21: Stack auxiliary predicate

`stackProperty` asserts property of local stack used by function `insert`. Algorithm uses this stack to record a path from root to the leaf node contains key `k`, with one node at each level. Therefore during searching process, parameter *level* is going to decrease from n to 1. Basically *level* means currently inner nodes from searching path below *level* are inside stack, whereas if $level \geqslant n$ it asserts an empty stack.

An subtle property of stack is that those node been recorded could been deleted by compression. Predicate, in fact, asserts that if node is still active it should be a node from certain level, otherwise it only asserts a deleted node.

`regionState` is a loop invariant where `level` is a variable used by algorithm. If `level` = 1 loop processes a leaf and it is still before linearizability point, Otherwise algorithm is after linearizability point but works on a certain inner node to re-construct B^{link}tree.

`primeBlockProperty` is a rather complex predicate to describe the property about prime block and its local copy. *pbL* ought to be the abstract state of shared resource where the left most addresses

$$\texttt{regionState} \equiv \quad (\texttt{level} = 1 \wedge a \Rrightarrow \blacklozenge \wedge \texttt{m} = \texttt{k} \wedge \texttt{w} = \texttt{v}) \vee (\texttt{level} \neq 1 \wedge a \Rrightarrow (S, S[\texttt{k} \mapsto \texttt{v}]))$$

Figure 4.22: Region state auxiliary predicate

$$\begin{aligned}\texttt{primeBlockProperty} \equiv \quad & \exists pbL, pbL', m, addrL_{(1,|pbL|)}. \\ & \texttt{x.pb} \mapsto \texttt{primeBlock}(pbL) * \texttt{pb} = \texttt{primeBlock}(pbL')^{\blacktriangle} \wedge \\ & \textstyle\bigwedge_{1 \leqslant i \leqslant |pbL|} (addrL_i \in pbL \wedge \texttt{at}(pbL, i) \mapsto \texttt{node}(\_, \_, \_, \_, \_, 0)) \wedge \\ & m \leqslant |pbL| \wedge m \leqslant |pbL'| \wedge \textstyle\bigwedge_{1 \leqslant i \leqslant m} \texttt{at}(pbL, i) = \texttt{at}(pbL', i) \wedge \\ & \textstyle\bigwedge_{m < i \leqslant |pbL'|} \texttt{at}(pbL', i) \mapsto \texttt{node}(\_, \_, \_, \_, \_, 1) \\ addrL_{level} \in pbL \equiv \quad & \texttt{first}(addrL_{level}) = \texttt{at}(pbL, level)\end{aligned}$$

Figure 4.23: Prime block auxiliary predicate

of all lists should reachable from pbL, i.e. $addrL_i \in pbL$. In addition, pbL always points to those nodes that are not deleted which actually is equivalent with reachability, since the abstract state of list only contains valid nodes.

However after taking a local copy pbL' from pbL, pbL could change dramatically even though statistically they should still store very similar information. Therefore what are known about pbL' is that there is a value m, and all the elements in pbL' bellows this value should remain as same value as pbL. Notice it is also based on the fact that compression either preserves the first node's address of each list or deletes current root and shrinks pbL completely. Given the fact, if there are elements above value m in local copy pbL', those nodes have actually been deleted.

$$\texttt{addrLkvLProperty} \equiv \quad kvL_1 = S \wedge \textstyle\bigwedge_{2 \leqslant i \leqslant |pbL|} kvL_{i-1}\restriction_2 \subseteq addrL_i$$

Figure 4.24: $addrL$ and pbL auxiliary predicate

`addrLkvLProperty` asserts that key value pairs at bottom should be equal to the abstract state S, besides a list should contains at least partial addresses of its direct child list. Basically, they are the same property in recursive predicate `Btree`.

Now we will present proof for main body of `insert`, whereas Appendix A.2 includes the rest proofs. As respect of simplification, we might do trivial frame off and frame back. Linearizability point is in `findNode`, `insertIntoSafe`, `insertIntoUnsafeRoot` or `insertIntoUnsafe`.

abstract and quantify a | make atomic and open region | Use

$\forall S.$
$\langle\ \texttt{map}(\texttt{x}, S)\ \rangle$
$\langle\ \mathbf{Map}_a(\texttt{x}, S) * [\text{MAP}]_a\ \rangle$
$a: \ S \rightsquigarrow S[\texttt{k} \mapsto \texttt{v}] \ \vdash$
$\{\ a \Rrightarrow \blacklozenge * \exists pbL, addrL, S.\, \texttt{x.pb} \mapsto \texttt{primeBlock}(pbL) * \texttt{Btree}(pbL, addrL, S)\ \}$
`function insert(x, k, v) {`
  `stack := newStack();`
  $\forall pbL.$
  $\langle\ \texttt{x.pb} \mapsto \texttt{primeBlock}(pbL)\ \rangle$
    `pb := getPrimeBlock(x.pb);`
  $\langle\ \texttt{x.pb} \mapsto \texttt{primeBlock}(pbL) * \texttt{pb} = \texttt{primeBlock}(pbL)\ \rangle$
$\{\ a \Rrightarrow \blacklozenge * \exists pbL, pbL', addrL, S.\, \texttt{x.pb} \mapsto \texttt{primeBlock}(pbL) *$
$\texttt{Btree}(pbL, addrL, S) * \texttt{stack}(\texttt{stack}, [])^{\blacktriangle} * \texttt{pb} = \texttt{primeBlock}(pbL')^{\blacktriangle}\ \}$
  `cur := root(pb);`

abstract and quantify a

make atomic and open region

abstract, quantify r_i and open region

// quantify $n = |pbL'|$ which equals to lenth of the most recent snapshot
// of prime block $|pbL'|$. $type_1 = 1$ whereas other $type_i = 0$
$\forall addrL_{(1,n)}, kvL_{(1,n)}, pbL.$

$\langle\ a \Rightarrow \blacklozenge * \texttt{primeBlockProperty} * \texttt{leafList}(addrL_1, kvL_1) *$
$\circledast_{2 \leqslant i \leqslant n} (\texttt{innerList}(addrL_i, kvL_i)) *$
$\texttt{addrLkvLProperty} * \texttt{stackProperty}(n)\ \rangle$

$\{\ a \Rightarrow \blacklozenge * \texttt{primeBlockProperty} * \texttt{untouchedList}(1) *$
$\texttt{addrLkvLProperty} * \texttt{stackProperty}(n)\ \}$

Use
$\forall addrL_n, l, \mathcal{P}.$
$\langle\ \texttt{cur} \mapsto \texttt{node}(l, \mathcal{P}) \wedge y \in addrL_n\ \rangle$
curn := get(cur);
$\langle\ \exists type.\ \texttt{curn} = \texttt{node}(l, \mathcal{P})^{\blacktriangle}_n *$
$\texttt{cur} \mapsto \texttt{node}(l, \mathcal{P}) \wedge y \in addrL_n\ \rangle$

$\{\ a \Rightarrow \blacklozenge * \texttt{primeBlockProperty} * \texttt{untouchedList}(1) *$
$\texttt{addrLkvLProperty} * \texttt{curn} = \texttt{node}(\_, \_, \_, \_, \_, \_)^{\blacktriangle}_n * \texttt{stackProperty}(n)\ \}$

goDownAndPushStack;

$\{\ a \Rightarrow \blacklozenge * \texttt{primeBlockProperty} * \texttt{untouchedList}(1) *$
$\texttt{addrLkvLProperty} * \texttt{curn} = \texttt{node}(\_, \_, \_, \_, \_, \_)^{\blacktriangle}_1 * \texttt{stackProperty}(1)\ \}$

level := 1;
m := k;
w := v;
while (true) {
// those region **BTreeList**() with level < level has
// already been updated and been frame off

$\{\ \texttt{regionState} * \texttt{primeBlockProperty} * \texttt{untouchedList}(\texttt{level}) *$
$\texttt{addrLkvLProperty} * \texttt{stackProperty}(\texttt{level}) * \texttt{cur} \mapsto \texttt{node}(\_, \_, \_, \_, \_, \_)\ \}$

findNode;
// if function not return

$\{\ \texttt{regionState} * \texttt{primeBlockProperty} * \texttt{untouchedList}(\texttt{level}) *$
$\texttt{addrLkvLProperty} * \texttt{stackProperty}(\texttt{level}) * \exists kvmL, mxk.$
$(\texttt{cur}, \texttt{curn}) \mapsto \texttt{node}(1, \_, kvmL, mxk, \_, \_)_{\texttt{level}} \wedge$
$\texttt{first}(kvmL)\downarrow_1 \leqslant \texttt{m} < mxk\ \}$

// frame off untouched list and stack
if (isSafe(curn)) {

$\{\ \texttt{regionState} * \exists kvmL, mxk.$
$(\texttt{cur}, \texttt{curn}) \mapsto \texttt{node}(1, \_, kvmL, mxk, \_, \_)_{\texttt{level}} \wedge$
$|kvmL| < \texttt{max} \wedge \texttt{first}(kvmL)\downarrow_1 \leqslant \texttt{m} < mxk\ \}$

insertIntoSafe;
// frame back untouched list and stack

$\{\ a \Rightarrow (S, S[\texttt{k} \mapsto \texttt{v}]) * \texttt{primeBlockProperty} * \texttt{untouchedList}(\texttt{level} + 1) *$
$\texttt{addrLkvLProperty} * \texttt{stackProperty}(\texttt{level}) \wedge ((\texttt{level} = 1 \wedge \texttt{m} = \texttt{k} \wedge$
$\texttt{w} = \texttt{v}) \vee \texttt{level} \neq 1)\ \}$

return null;
// end of all rule
$\langle\ \texttt{map}(\texttt{x}, S[\texttt{k} \mapsto \texttt{v}]) \wedge \texttt{ret} = \texttt{null}\ \rangle$

4.6.2 Proof of compression algorithm

Since compression may logically delete node, those we use the deleting pool module from §4.2. Therefore we need to define a new `dead` predicate.

$$\texttt{dead}(x, \texttt{btreeNode}) \stackrel{\text{def}}{=} x \mapsto \texttt{node}(\_, \_, \_, \_, \_, \_)$$

Figure 4.25: Dead B^{link}tree node

Auxiliary predicate `listProperty` asserts that there is a list at *level* and some guards. Those guards with id a are guards from list at *level* and guard with id l are guard from list at $level + 1$. Predicate also asserts that both lists at *level* and $level + 1$ should be reachable from pbL.

$$\begin{aligned}\texttt{listProperty}(level) \equiv\ & \texttt{list}(addrL_{level}, kvL_{level}, type_{level}) * [\text{BTREE\_LIST}]_l * \\ & [\text{BTREE\_LIST}]_a * [\text{BTREE\_LIST\_UNI}]_a \wedge \\ & addrL_{level} \in pbL \wedge addrL_{level+1} \in pbL\end{aligned}$$

Figure 4.26: List in compression auxiliary predicate

We present proof of main body of `compress`, the rest proof is in appendix A.3. Specification for `compress` says that all key value pairs at `level` are preserved by this function in a logically one step. This specification can imply $\langle \texttt{map}(\texttt{x}, S)\rangle \texttt{compress}(\texttt{x}, \texttt{level}, \texttt{b}) \langle \texttt{map}(\texttt{x}, S)\rangle$, which gives guarantee to clients that compression preserves all information inside map.

abstract and quantify l and a

$\forall pbL, addrL_{\texttt{level}+1}, kvL_{\texttt{level}+1}, addrL_{\texttt{level}}, kvL_{\texttt{level}}, deadL.$

$\langle\ \texttt{x.pb} \mapsto \texttt{primeBlock}(pbL) * \texttt{innerList}(addrL_{\texttt{level}+1}, kvL_{\texttt{level}+1}) * \texttt{deletingPool}(deadL) *$
$((\texttt{level} > 1 \wedge \texttt{innerListCompress}(addrL_{\texttt{level}}, kvL_{\texttt{level}})) \vee$
$(\texttt{level} = 1 \wedge \texttt{leafListCompress}(addrL_{\texttt{level}}, kvL_{\texttt{level}})))\ \rangle$

make atomic and open region

// frame off `deletingPool`

$\langle\ \texttt{x} \mapsto \texttt{primeBlock}(pbL) * \mathbf{BTreeList}_l(addrL_{\texttt{level}+1}, kvL_{\texttt{level}+1}, 0) *$
$[\text{BTREE\_LIST}]_l * \exists type.\, \mathbf{BTreeList}_a(addrL_{\texttt{level}}, kvL_{\texttt{level}}, type) *$
$[\text{BTREE\_LIST}]_a * [\text{BTREE\_LIST\_UNI}]_a \wedge$
$((\texttt{level} > 1 \wedge type = 0)) \vee (\texttt{level} = 1 \wedge type = 1)))\ \rangle$

$a :\ kvL_{\texttt{level}} \rightsquigarrow kvL_{\texttt{level}}\ \vdash$

`function compressLevel(x, level, b) {`

// frame off list at `level` and guards

$\{\ \texttt{x.pb} \mapsto \texttt{primeBlock}(pbL) * \texttt{list}(addrL_{\texttt{level}+1}, kvL_{\texttt{level}+1}, 0)\ \}$

Use
- $\forall pbL.$
- $\langle\ \texttt{x.pb} \mapsto \texttt{primeBlock}(pbL)\ \rangle$
- `pb := getPrimeBlock(x.pb);`
- $\langle\ \texttt{x.pb} \mapsto \texttt{primeBlock}(pbL) * \texttt{pb} = \texttt{primeBlock}(pbL)\ \rangle$

$\{\ \texttt{primeBlockProperty} * \texttt{list}(addrL_{\texttt{level}+1}, kvL_{\texttt{level}+1}, 0)\ \}$

`cur := getNodeLevel(pb, level + 1);`

$\{\ \texttt{primeBlockProperty} * \texttt{cur} \mapsto \texttt{node}(\_, \_, \_, \_, \_, \_) \wedge addrL_{\texttt{level}+1} \in pbL\ \}$

`while (cur ≠ null) {`

$\{\ \texttt{primeBlockProperty} * \texttt{cur} \mapsto \texttt{node}(\_, \_, \_, \_, \_, \_) \wedge addrL_{\texttt{level}+1} \in pbL\ \}$

Use
- $\forall l, \mathcal{P}.$
- $\langle\ \texttt{cur} \mapsto \texttt{node}(l, \mathcal{P})\ \rangle$
- `lock(cur);`
- $\langle\ \texttt{cur} \mapsto \texttt{node}(1, \mathcal{P})\ \rangle$

$\{\ \texttt{primeBlockProperty} * \texttt{cur} \mapsto \texttt{node}(1, \_, \_, \_, \_, \_) \wedge addrL_{\texttt{level}+1} \in pbL\ \}$

Use
- $\langle\ \texttt{cur} \mapsto \texttt{node}(1, \mathcal{P})\ \rangle$
- `curn := get(cur);`
- $\langle\ \texttt{cur} \mapsto \texttt{node}(1, \mathcal{P}) * \texttt{curn} = \texttt{node}(1, \mathcal{P})\ \rangle$

abstract and quantify l and a

make atomic and open region

$\{$ $\texttt{primeBlockProperty} * \texttt{cur} \mapsto \texttt{node}(1,\_,\_,\_,\_,\_) * \texttt{curn} = \texttt{node}(1,\_,\_,\_,\_,\_) \wedge$
$addrL_{\texttt{level}+1} \in pbL$ $\}$

if (isDelete(curn)) {

$\{$ $\texttt{primeBlockProperty} * \texttt{cur} \mapsto \texttt{node}(1,\_,\_,\_,\_,1) * \texttt{curn} = \texttt{node}(1,\_,\_,\_,\_,1) \wedge$
$addrL_{\texttt{level}+1} \in pbL \wedge \texttt{cur} \notin addrL_{\texttt{level}+1}$ $\}$

Use $\langle$ $\texttt{cur} \mapsto \texttt{node}(1,\_,\_,\_,\_,\_)$ $\rangle$
unlock(cur);
$\langle$ $\texttt{cur} \mapsto \texttt{node}(\_,\_,\_,\_,\_,\_)$ $\rangle$

$\{$ $\texttt{primeBlockProperty} * \texttt{cur} \mapsto \texttt{node}(\_,\_,\_,\_,\_,\_) * \wedge addrL_{\texttt{level}+1} \in pbL \wedge$
$\texttt{cur} \notin addrL_{\texttt{level}+1}$ $\}$

cur := getNodeLevel(pb, level + 1);

$\{$ $\texttt{primeBlockProperty} * \texttt{cur} \mapsto \texttt{node}(\_,\_,\_,\_,\_,\_) * \wedge addrL_{\texttt{level}+1} \in pbL$ $\}$

} else if (!hasState(curn, !b)) {

$\{$ $\texttt{primeBlockProperty} * \exists kvmL.\, \texttt{cur} \mapsto \texttt{node}(1,\_,kvmL,\_,\_,\_)_{\texttt{level}+1} *$
$\texttt{curn} = \texttt{node}(1,\_,kvmL,\_,\_,\_)_{\texttt{level}+1} \wedge addrL_{\texttt{level}+1} \in pbL \wedge \bigwedge_{(\_,\_,b) \in kvmL} b = \texttt{b}$ $\}$

Use $\langle$ $\texttt{cur} \mapsto \texttt{node}(1,\_,\_,\_,\_,\_)$ $\rangle$
unlock(cur);
$\langle$ $\texttt{cur} \mapsto \texttt{node}(\_,\_,\_,\_,\_,\_)$ $\rangle$

$\{$ $\texttt{primeBlockProperty} * \texttt{cur} \mapsto \texttt{node}(\_,\_,\_,\_,\_,\_) * \texttt{curn} = \texttt{node}(\_,\_,\_,\_,\_,\_) \wedge$
$addrL_{\texttt{level}+1} \in pbL$ $\}$

nextNode(cur);

$\{$ $\texttt{primeBlockProperty} * (\texttt{cur} \mapsto \texttt{node}(\_,\_,\_,\_,\_,\_) \vee \texttt{cur} = \texttt{null}) *$
$\texttt{curn} = \texttt{node}(\_,\_,\_,\_,\_,\_) \wedge addrL_{\texttt{level}+1} \in pbL$ $\}$

} else {

// frame back list at `level` and guards,
// which presents by `listProperty`

$\{$ $\texttt{primeBlockProperty} * \texttt{listProperty}(\texttt{level}) * \exists kvmL.$
$\texttt{cur} \mapsto \texttt{node}(1,\_,kvmL,\_,\_,\_)_{\texttt{level}+1} * \texttt{curn} = \texttt{node}(1,\_,kvmL,\_,\_,\_)_{\texttt{level}+1} \wedge$
$addrL_{\texttt{level}+1} \in pbL \wedge (\_,\_,\neg\texttt{b}) \in kvmL$ $\}$

compressNode;

$\{$ $\texttt{primeBlockProperty} * \texttt{listProperty}(\texttt{level}) * (\texttt{cur} \mapsto \texttt{node}(\_,\_,\_,\_,\_,\_) \vee$
$\texttt{cur} = \texttt{null}) \wedge addrL_{\texttt{level}+1} \in pbL$ $\}$

}}}

// $kvL_{\texttt{level}+1}$, $addrL_{\texttt{level}}$ and $deadL$ has been update

$\langle$ $\dagger^1$ $\rangle$

$\langle$ $\dagger^2$ $\rangle$

$\dagger^1$ $\exists type, kvL'_{\texttt{level}+1}, addrL'_{\texttt{level}}, deadL'.\, \texttt{x} \mapsto \texttt{primeBlock}(pbL) *$
$\mathbf{BTreeList}_l(addrL_{\texttt{level}+1}, kvL'_{\texttt{level}+1}, 0) * [\text{BTREE\_LIST}]_l *$
$\mathbf{BTreeList}_a(addrL'_{\texttt{level}}, kvL_{\texttt{level}}, type) * [\text{BTREE\_LIST}]_a * [\text{BTREE\_LIST\_UNI}]_a *$
$\texttt{deletingPool}(deadL') \wedge ((\texttt{level} > 1 \wedge type = 0) \vee (\texttt{level} = 1 \wedge type = 1)) \wedge$
$deadL' = deadL \uplus (addrL_{\texttt{level}} \backslash addrL'_{\texttt{level}})$

$\dagger^2$ $\exists kvL'_{\texttt{level}+1}, addrL'_{\texttt{level}}, deadL.\, \texttt{x.pb} \mapsto \texttt{primeBlock}(pbL) *$
$\texttt{innerList}(addrL_{\texttt{level}+1}, kvL'_{\texttt{level}+1}) * \texttt{deletingPool}(deadL') *$
$((\texttt{level} > 1 \wedge \texttt{innerListCompress}(addrL'_{\texttt{level}}, kvL_{\texttt{level}})) \vee$
$(\texttt{level} = 1 \wedge \texttt{leafListCompress}(addrL'_{\texttt{level}}, kvL_{\texttt{level}}))) \wedge$
$deadL' = deadL \uplus (addrL_{\texttt{level}} \backslash addrL'_{\texttt{level}})$

Chapter 5

Skip List Implementations and Proofs

The second implmentation is concurrent skip list [27, 26, 35].This chapter chooses a realistic code which is a tranlation from Java `ConcurrentSkipListMap` [35]. Besides, we would like to find out whether there a proof pattern by looking more examples, which could be good start point for automatic proving in the future work.

5.1 Skip List

5.1.1 Data Structure

Skip List is a probability version of balanced search tree [27]. A skip list is a normal linked list and there are certain amount of index lists at top. There should be statistically $\log_2 n$ index lists, where n is the number of nodes at bottom. Each level should contain roughly half indexes as its directly child list, for instance the first level should have about $n/2$ index nodes and the second level should have $n/4$, etc. With those index lists, instead of balancing, the performance of skip list depends on a random number generator, because when adding a new node at bottom the random number generator should also give a number which represents how many index nodes it should have. In other words, this random number generator should satisfy Poisson distribution, which then can give us expected performance of skip list [24].

Given that skip List is easy to maintained but have similar performance of balanced search tree, it is one of popular implementation of map or set. Pugh [26] has a algorithm for concurrent skip list but it requires lock. `java.util.concurrent` has another algorithm [35] that only uses `CAS` and deletion marker, where the idea is adopted from [13, 20].

Before we explain algorithm, we first explains the concrete structure of `ConcurrentSkipListMap` in Java [35]. At beginning, there is a linked list with each node recording a level number and an address, which helps to find the first node in each level so as the root. We would like to call it head list and head node in the rest part of this chapter. At bottom of the whole data structure, there is a linked list to record all key value pairs and we would like to call bottom list and bottom node. At last, all index lists are also linked list with each node contains three pointers, one for the right index node, one for next level index node and one for bottom node. Here index lists does not record any concrete index value, but instead, they records a pointer to bottom node. The trick is when some thread deletes a bottom node, which will set the value to `null` by a `CAS`, then all the

index nodes who point to this bottom node are also logically deleted as well. However if index node saved a concrete index value, it may needed a multi-cas or other mechanism to maintain indexes.

5.1.2 Deletion Algorithm

Since Insertion algorithm should take care of the effect of deletion, so we also cover idea of deletion algorithm here. Deletion is rather simple operation by three `CAS`. Let take deleting a key value pair (cur, v_2) as a example, where $\rightarrow$ denotes "points to", and tuple denotes a node contains one key value pair,

- With starting state $(prev, v_1) \rightarrow (cur, v_2) \rightarrow (next, v_3)$, deletion thread `CAS` value v_2 to `null`. If `CAS` succeed, it should end up at state $(prev, v_1) \rightarrow (cur, \texttt{null}) \rightarrow (next, v_3)$ where key value pair is already logically deleted. It means all other threads should treat this key no longer exists and this node must to be deleted eventually. Besides, all index nodes who point to *cur* also know they should be deleted.
- From state $(prev, v_1) \rightarrow (cur, \texttt{null}) \rightarrow (next, v_3)$, any other thread who finds this situation should try to append a marker to *cur* by `CAS`. Then if it succeed, state should be $(prev, v_1) \rightarrow (cur, \texttt{null}) \rightarrow (marker, \_) \rightarrow (next, v_3)$, and marker also stop the next node being change, i.e. deletion or insertion.
- At last, whoever find $(prev, v_1) \rightarrow (cur, \texttt{null}) \rightarrow (marker, \_) \rightarrow (next, v_3)$ should try to delete *cur* and *marker* by `CAS`. Then the successful final state is $(prev, v_1) \rightarrow (next, v_3)$.

Except the first step, the rest two steps are not guarantee to be done by deletion, which means that the insertion and search might help finish the rest two steps.

```
function insert(x, k, v) {
  while (true) {
    restart := false;
    prev := findPredecessor(x, k);
    cur := prev.next;
    while (restart = false) {
      found := false;
      if (cur ≠ null) {
        next := cur.next; tv := cur.value;
        if (tv = null) {
          helpDelete(prev, cur, next);
          restart := true;
        } else if(isMarker(tv)||
            prev.value = null){
          restart := true;
        } else if(k > cur.key) {
          prev := cur; cur := next;
        } else if(k = cur.key) {
          b := CAS(cur.value, tv, v);
          if (b = 1) {return tv;}
          else {restart := true;}
        } else { found := true;}
      } else { found := true;}
      if (found = true) {
        new := newNode(k, v, cur);
        b := CAS(prev.next, cur, new);
        if (b = 0) {restart := true;} else {
          level := randomLevel();
          if (level > 0) {
            insertindex(x, new, level);
          }
          return null;
} } } } }
```

Figure 5.1: Concurrent Skip List Insertion

5.1.3 Insertion Algorithm

After discuss the deletion Figure 5.1 presents the pseudo-code of insertion, which is a direct translation of function `doPut` in [35]. The outside loop is a dead loop but functions will return potentially some point within this loop, and this loop is used for restarting insertion from root after following three situations,

- thread has tried to help deleting a node,

- fail to `CAS` a old value to new value `v`,
- fail to adding new node.

`helpDelete` ought to help to preserve the data structure, i.e. the last two steps described in §5.1.2. If it finds exact same key as `k`, algorithm try to modify the old value to new value by `CAS(cur.value,tv,v)`. But if `found` = `true`, it means that there is no node with the same key has been found yet a direct preview node has been found. In this situation, `CAS(prev.next,cur,new)` is used to append a new node with key `k` and value `v`, and then `insertindex` is used to add necessary index nodes. Intuitively, `insertindex` uses `CAS` to add index node in many steps, and this function has `level` linearizability points. those change, however, is not observable by client but only for purpose of performance.

Notice, this algorithm cannot work well if there is no garbage collection. In addition, our algorithm has no `break` command, but Java use many `break` commands. It is because TaDA cannot deal with break so we change the algorithm by introducing some variables such as `found`.

5.2 Predicate And Guard

$\mathbf{Map}(\mathtt{x}, S)$ is firstly interpreted as a the most abstract form of skip list that includes a head list, a index block and a bottom list. Predicate `equal` presumably says that the abstract state S are logically equivalent to $addrkvL$. $addrkvL$ is a abstract state with each elements is triple $(addr, k, v)$, where $addr$ is node address, k is key and v is value. $addrkvL$ is abstract state for bottom list so $addrkvL$ might also include node with `null` value. Predicate `inOrder` simply restrain key should be stored in order. We will explains other predicates later.

$$
\begin{aligned}
I(\mathbf{Map}(\mathtt{x}, S)) \stackrel{\text{def}}{=}\ & \exists hL, idxL, addrkvL.\, \mathtt{x.head} \mapsto \mathtt{headList}(hL) * \\
& \mathtt{indexBlock}(hL, idxL, addrkvL) * \mathtt{bottomList}(addrkvL) \wedge \\
& \mathtt{equal}(S, addrkvL) \wedge \mathtt{inOrder}(addrkvL) \\
\mathtt{equal}(S, addrkvL) \equiv\ & \exists k, v, addrkvL', S'.\, \mathtt{equal}(S', addrkvL') \wedge \\
& ((addrkvL = [(\_, \_, \mathtt{null})] +\!\!+ addrkvL' \wedge S = S') \vee \\
& (addrkvL = [(\_, k, v)] +\!\!+ addrkvL' \wedge S = S' \uplus \{k \mapsto v\})) \\
\mathtt{inOrder}(addrkvL) \equiv\ & \textstyle\bigwedge_{1 \leqslant i < |addrkvL|} \mathtt{at}(addrkvL, i)\lfloor_2 < \mathtt{at}(addrkvL, i+1)\lfloor_2
\end{aligned}
$$

Figure 5.2: Abstract predicates for skip list

`headList` is defined as a region with its duplicatable guard. This duplicatable guard allows adding or deleting one element at the top which is also the current top of skip list. $\mathbf{HeadList}(x, hL)$ is interpreted as a linked list which consist of head nodes. Head node has three fields where *right* points to the left most node of the index list at *level*, and *down* points to next head node.

Guard [BOTTOM_LIST] for bottom list allows adding a new node, modifying value of a exist node to a new value, setting value to `null` and at last deleting a node with value `null`. $\mathbf{BottomList}(addrkvL)$ is interpreted as linked list with abstract state $addrkvL$. This list is consist of bottom nodes that contain key value pair. There is a special case within this list, that it is between two elements from $addrkvL$ it could be a marker, i.e. $nxt \mapsto \mathtt{marker}(nxt')$.

Index Block is relatively complicated part of skip list. hL is the abstract state of head list. $idxL$ is the abstract state of index list at the top of index block with each element is triple $(addr, node, down)$, where $addr$ is the index node address, $node$ is the pointer points to a bottom node, and $down$ is a pointer points to another index in next level. Intuitively, by traversal the

$$x \mapsto \texttt{headList}(hL) \stackrel{\text{def}}{=} \exists a.\, \mathbf{HeadList}_a(x, hL) * [\text{HEAD\_LIST}]_a$$

$$[\text{HEAD\_LIST}]: \quad \forall hL, hL'.\, hL \rightsquigarrow hL' +\!\!+ hL \quad hL' +\!\!+ hL \rightsquigarrow hL$$

$$I(\mathbf{HeadList}(x, hL)) \stackrel{\text{def}}{=} \exists level.\, \texttt{hList}(x, hL, level)$$

$$\begin{aligned}\texttt{hList}(x, hL, level) \equiv\ & (x = \texttt{null} \wedge hL = [] \wedge level = 0) \vee (\exists right, hL', down.\\ & x \mapsto \texttt{headnode}(right, level, down) * \texttt{hList}(down, hL', level - 1) \wedge\\ & hL = [right] +\!\!+ hL')\end{aligned}$$

$$[\text{HEAD\_LIST}] \Longrightarrow [\text{HEAD\_LIST}] * [\text{HEAD\_LIST}]$$

Figure 5.3: Head list and guards

$$\texttt{bottomList}(addrkvL) \stackrel{\text{def}}{=} \exists a.\, \mathbf{BottomList}_a(addrkvL) * [\text{BOTTOM\_LIST}]_a$$

$$\begin{aligned}[\text{BOTTOM\_LIST}]: \quad & \forall addrkvL, addr, k, v, v'.\, addrkvL \rightsquigarrow addrkvL \uplus [(addr, k, v)]\\ & addrkvL \rightsquigarrow addrkvL \backslash [(addr, k, v)] \uplus [(addr, k, v')] \wedge addr \in addrkvL\downarrow_1\\ & addrkvL \rightsquigarrow addrkvL \backslash [(addr, k, v)] \uplus [(addr, k, \texttt{null})] \wedge addr \in addrkvL\downarrow_1\\ & addrkvL \rightsquigarrow addrkvL \backslash [(addr, k, \texttt{null})]\end{aligned}$$

$$I(\mathbf{BottomList}(addrkvL)) \stackrel{\text{def}}{=} \texttt{addrkvlist}(addrkvL)$$

$$\begin{aligned}\texttt{addrkvlist}(addrkvL) \equiv\ & addrkvL = [] \vee (\exists x, k, v, nxt, nxt', addrkvL'.\, x \mapsto \texttt{node}(k, v, nxt) *\\ & (nxt = \texttt{first}(addrkvL')\downarrow_1 \vee (nxt \neq \texttt{first}(addrkvL')\downarrow_1 \wedge\\ & nxt \mapsto \texttt{marker}(nxt') \wedge nxt' = \texttt{first}(addrkvL')\downarrow_1)) *\\ & \texttt{addrkvlist}(addrkvL') \wedge addrkvL = [(x, k, v)] +\!\!+ addrkvL'\end{aligned}$$

$$[\text{BOTTOM\_LIST}] \Longrightarrow [\text{BOTTOM\_LIST}] * [\text{BOTTOM\_LIST}]$$

Figure 5.4: Bottom list and guard

down pointer, all the index node should have the same *node* value. $addrkvL$ is the abstract state of bottom list.

$\texttt{indexBlock}(hL, idxL, addrkvL)$ is recursively defined as an index list $\texttt{indexList}(idxL)$ at top, and the rest index block $\texttt{indexBlock}(hL', idxL', addrkvL)$. This predicate also asserts that if there are more index list, those two adjacent lists should satisfy some property. $idxL\downarrow_2 \subseteq idxL'\downarrow_2$ means that child list should have all the indexes that what parent has. Whereas $idxL\downarrow_3 = idxL'\downarrow_1$ means that all the *down* pointers in parent list with abstract state $idexL$ must be one node in next level. With these two assertions, we have restrained the relationship between two adjacent lists. Predicate also asserts that if it reaches the case there is no more index list, it should have the information that all node in bottom list has one corresponding node in the last index list, in other word $idxL\downarrow_2 = addrkvL\downarrow_1$. At last algorithm assumes all *down* pointers in the last index list should point to `null`.

Guard for index list is also very similar with bottom list, yet index list is only allowed to add a new index node or delete a index node with side condition that *node* should point to a bottom node with value `null`.

As similar to bottom list, $\mathbf{IndexList}(idxL)$ is interpreted as a trivial linked list with each element is an . The predicate $addr \mapsto \texttt{index}(node, down, right)$ asserts that a index node with *node* pointer points to a node in bottom list, *down* pointer points to next level index node and *right* points the next index node in the same level.

$$\begin{aligned}
\texttt{indexBlock}(hL, idxL, addrkvL) \equiv\ & \exists x, hL', idxL'.\, (|hL| > 1 \wedge \texttt{indexList}(idxL) * \\
& \texttt{indexBlock}(hL', idxL', addrkvL) \wedge hL = [x] +\!\!+ hL' \wedge \\
& (x, _, _) = \texttt{first}(idxL) \wedge idxL\!\downarrow_2 \subseteq idxL'\!\downarrow_2 \wedge \\
& idxL\!\downarrow_3 = idxL'\!\downarrow_1) \vee (|hL| = 1 \wedge \texttt{indexList}(idxL) \wedge \\
& idxL\!\downarrow_2 = addrkvL\!\downarrow_1 \wedge \textstyle\bigwedge_{(_,_,down) \in idxL} down = \texttt{null})
\end{aligned}$$

$$\texttt{indexList}(idxL) \stackrel{\text{def}}{=} \exists a.\, \mathbf{IndexList}_a(idxL) * [\text{INDEX\_LIST}]_a$$

$$\begin{aligned}
[\text{INDEX_LIST}]: \quad & \forall idxL, addr, down, node.\, idxL \rightsquigarrow idxL \uplus [(addr, node, down)] \\
& idxL \rightsquigarrow idxL \backslash [(addr, node, down)] \wedge addr \in idxL\!\downarrow_1 \wedge node \mapsto \texttt{node}(_, \texttt{null}, _)
\end{aligned}$$

$$I(\mathbf{IndexList}(idxL)) \stackrel{\text{def}}{=} \texttt{idxList}(idxL)$$

$$\begin{aligned}
\texttt{idxList}(idxL) \equiv\ & idxL = [] \vee (\exists addr, node, down, idxL'. \\
& addr \mapsto \texttt{index}(node, down, \texttt{first}(idxL')\!\downarrow_1) * \texttt{idxList}(idxL') \wedge \\
& idxL = [(addr, node, down)] +\!\!+ idxL'
\end{aligned}$$

Figure 5.5: Index list and guards

5.3 Proof

For a readable proof, we introduce a auxliary predicate, $x \mapsto \texttt{node}(k, v, nxt)_{addrkvL}$, to assert a bottom node which is included in $addrkvL$. Similarly, there is $x \mapsto \texttt{marker}(nxt)_{addrkvL}$.

$$\begin{aligned}
x \mapsto \texttt{node}(k, v, nxt)_{addrkvL} &\equiv x \mapsto \texttt{node}(k, v, nxt) \wedge (v = \texttt{null} \vee (v \neq \texttt{null} \wedge \texttt{prev} \in addrkvL\!\downarrow_1)) \\
x \mapsto \texttt{marker}(nxt)_{addrkvL} &\equiv \quad x \mapsto \texttt{marker}(nxt) \wedge nxt \in addrkvL \\
x \mapsto \texttt{nodeOrMarker}(k, v, nxt)_{\alpha} &\equiv \quad x \mapsto \texttt{node}(k, v, nxt)_{\alpha} \vee (x \mapsto \texttt{marker}(nxt)_{\alpha} \wedge v = \delta) \\
x \mapsto \texttt{nodeOrMarkerOrNull}(k, v, nxt)_{\alpha} &\equiv \quad x \mapsto \texttt{nodeOrMarker}(k, v, nxt)_{\alpha} \vee x = \texttt{null}
\end{aligned}$$

Figure 5.6: Node auxiliary predicates

Two other trivial auxiliary predicates are used as short hand for x points to a bottom node, a marker or $x = \texttt{null}$. Here, α could be $addrkvL$ or nothing. For instance, it could be $x \mapsto \texttt{nodeOrMarker}(k, v, nxt)_{addrkvL}$ or $x \mapsto \texttt{nodeOrMarker}(k, v, nxt)$. In term of the marker case, the real Java algorithm choose marker as a node which the value points to itself. However, here we simplify then by assuming marker has special value δ.

We begin from presenting specifications of four auxiliary functions used in insertion. `findPredecessor(x,k)` shown in figure 5.7 will find the predecessor of key `k`, which means if this key exists return should be a node after predecessor node. The formal specification asserts that it is one logically atomic step and `ret` is a node in bottom list. But pre-condition also includes the head list and index block, the reason is in term of correctness this function need to know where are the head list and index block.

Here, we choose to use $\texttt{indexBlock}(hL, idxL_{level}, addrkvL)$ to intuitively present that the index list from 1 to *level*. One can define a better and formal predicate to describe that, but here we believe it should not be a big problem therefore we present this predicate in a very stylish way.

Besides in term of specification, one can choose to specify this function in a more abstract level, for instance `map` level. But this would be a question about what is the client of this function. If this function is for all clients, it would a good idea to specify in a more abstract level, but here it is only a private function so we choose to speficy in a lower level.

`helpDelete(prev, cur, next)` is a function that could append a marker to `cur` (#1 in figure 5.8), or if a marker exist it could concretely delete `cur` (#2 in figure 5.8). Since all those operation are

$$\vdash \quad \forall S, addrkvL, level, hL, idxL_{(1,level)}.$$
$$\langle \texttt{x.head} \mapsto \texttt{headList}(hL) * \texttt{indexBlock}(hL, idxL_{level}, addrkvL) * \texttt{addrkvlist}(addrkvL) \rangle$$
$$\texttt{findPredecessor}(\texttt{x}, \texttt{k})$$
$$\left\langle \begin{array}{c} \texttt{x.head} \mapsto \texttt{headList}(hL) * \texttt{indexBlock}(hL, idxL_{level}, addrkvL) * \\ (\texttt{addrkvlist}(addrkvL) \wedge \texttt{ret} \mapsto \texttt{node}(k, \_, \_) \wedge \texttt{ret} \in addrkvL) \wedge k < \texttt{k} \end{array} \right\rangle$$

Figure 5.7: Specification for `findPredecessor`

$$\vdash \quad \forall nxt_1, nxt_2, nxt_3.$$
$$\left\langle \begin{array}{c} \texttt{prev} \mapsto \texttt{node}(\_, \_, nxt_1) * \texttt{cur} \mapsto \texttt{node}(\_, \texttt{null}, nxt_2) * \\ \texttt{next} \mapsto \texttt{nodeOrMarkerOrNull}(\_, \_, nxt_3) \end{array} \right\rangle$$
$$\texttt{helpDelete}(\texttt{prev}, \texttt{cur}, \texttt{next})$$
$$\langle \#1 \vee \#2 \vee \#3 \rangle$$

where,

$\#1$ $\texttt{prev} \mapsto \texttt{node}(\_, \_, nxt_3) * \texttt{cur} \mapsto \texttt{node}(\_, \texttt{null}, nxt_2') * \texttt{next} \mapsto \texttt{marker}(nxt_3) \wedge$
$nxt_1 = \texttt{cur} \wedge nxt_2 = \texttt{next}$

$\#2$ $\exists newmarker.\, \texttt{prev} \mapsto \texttt{node}(\_, \_, nxt_1) * \texttt{cur} \mapsto \texttt{node}(\_, \texttt{null}, newmarker) *$
$newmarker \mapsto \texttt{marker}(nxt_2) * \texttt{next} \mapsto \texttt{nodeOrMarkerOrNull}(\_, \_, nxt_3) \wedge$
$nxt_1 = \texttt{cur} \wedge nxt_2 = \texttt{next}$

$\#3$ $\texttt{prev} \mapsto \texttt{node}(\_, \_, nxt_1) * \texttt{cur} \mapsto \texttt{node}(\_, \texttt{null}, nxt_2) *$
$\texttt{next} \mapsto \texttt{nodeOrMarkerOrNull}(\_, \_, nxt_3) \wedge (nxt_1 \neq \texttt{cur} \vee nxt_2 \neq \texttt{next})$

Figure 5.8: Specification for `helpDelete`

by `CAS` and this function only tries to update instead of guarantee to do those operation, therefore the third case is that `CAS` is fail and this function does not do any update but envoriment does (#3 in figure 5.8).

`insertindex(x,new,level)` shown in figure 5.9 is, in fact, a very complicated functions. But it is a function that is for purpoce of performance, so we specify it in a very simple way. This function creates index nodes and inserts them to index lists in several linearizability points. It could also add a new head node at top of head list and add a new level of index list at top of index block as well. In post-condition, we simply assume index block and head list has been updated to a new abstract state by quantify $level', hL', idxL'_{(1,level')}$, but more importance is that function preserve the abstract state of S.

$$\vdash \quad \forall addrkvL, level, hL, idxL_{(1,level)}.$$
$$\langle \texttt{x.head} \mapsto \texttt{headList}(hL) * \texttt{indexBlock}(hL, idxL_{level}, addrkvL) \rangle$$
$$\texttt{insertindex}(\texttt{x}, \texttt{new}, \texttt{level})$$
$$\left\langle \begin{array}{c} \exists level', hL', idxL'_{(1,level')}.\, \texttt{x.head} \mapsto \texttt{headList}(hL') * \\ \texttt{indexBlock}(hL', idxL'_{level}, addrkvL) \wedge (level' = level \vee level' = level + 1) \end{array} \right\rangle$$

Figure 5.9: Specification of `insertindex`

With those specifications of auxiliary functions and those auxiliary predicates, we have a proof for insertion algorithm. In proof, $U\&U$ is short for Use atomic and Update atomic rules.

$\forall S.$
$\langle\ \texttt{map}(\texttt{x}, S)\ \rangle$
$\quad \big| \langle\ \mathbf{Map}_a(\texttt{x}, S) * [\mathrm{MAP}]_a\ \rangle$
$\quad\quad \big| \ a :\ \ S \rightsquigarrow S[\texttt{k} \mapsto \texttt{v}]\ \ \vdash$


abstract, quantify a
make atomic and open region
abstract, and open region bottomList
function insert(x, k, v) {
while (true) {
$a \Rightarrow \blacklozenge * \exists S, hL, idxL, addrkvL.$ x.head $\mapsto$ headList(hL)*
indexBlock($hL, idxL, addrkvL$) * bottomList($addrkvL$) $\wedge$
equal($S, addrkvL$) $\wedge$ inOrder($addrkvL$)
// $\exists S.$ equal($S, addrkvL$) $\wedge$ inOrder($addrkvL$) are globally held
// [BOTTOM_LIST] are globally held
$a \Rightarrow \blacklozenge * \exists hL, idxL, addrkvL.$ x.head $\mapsto$ headList(hL)*
indexBlock($hL, idxL, addrkvL$) * addrkvlist($addrkvL$)
restart := false;
$a \Rightarrow \blacklozenge * \exists hL, idxL, addrkvL.$ x.head $\mapsto$ headList(hL)*
indexBlock($hL, idxL, addrkvL$) * addrkvlist($addrkvL$) $\wedge$
restart = false
Use
// indexBlock wraps index list from 1 to level
$\forall S, addrkvL, level, hL, idxL_{(1,level)}.$
$\langle$ x.head $\mapsto$ headList(hL) * indexBlock($hL, idxL_{level}, addrkvL$)*
addrkvlist($addrkvL$) $\rangle$
prev := findPredecessor(x, k);
// modify the data struct but it still preserve $addrkvL$ and thus S
// use $P \wedge Q$ to pressent resource overlaping
$\langle$ x.head $\mapsto$ headList(hL) * indexBlock($hL, idxL_{level}, addrkvL$)*
(addrkvlist($addrkvL$) $\wedge$ prev $\mapsto$ node($k, \_, \_)_{addrkvL}$) $\wedge$ $k <$ k $\rangle$
// frame off headList, indexBlock and part of addrkvlist
$a \Rightarrow \blacklozenge * \exists addrkvL, k, k'.$ prev $\mapsto$ node($k, \_, \_)_{addrkvL} \wedge$
$k <$ k $\wedge$ restart = false
Use
$\forall nxt, addrkvL.$
$\langle$ prev $\mapsto$ node($\_, \_, nxt)_{addrkvL}$ $\rangle$
cur := prev.next;
$\langle$ prev $\mapsto$ node($\_, \_, nxt)_{addrkvL}$*
cur $\mapsto$ nodeOrMarkerOrNull($\_, \_, \_)_{addrkvL} \wedge$ cur = nxt $\rangle$
$a \Rightarrow \blacklozenge * \exists addrkvL, k, k'.$ prev $\mapsto$ node($k, \_, \_)_{addrkvL}$*
cur $\mapsto$ nodeOrMarkerOrNull($k', \_, \_)_{addrkvL} \wedge$
$k <$ k $\wedge$ $k < k'$ $\wedge$ restart = false
while (restart = false) {
found := false;
if (cur $\neq$ null) {
$a \Rightarrow \blacklozenge * \exists addrkvL, k, k'.$ prev $\mapsto$ node($k, \_, \_)_{addrkvL}$*
cur $\mapsto$ nodeOrMarker($k', \_, \_)_{addrkvL} \wedge k <$ k $\wedge$ $k < k' \wedge$
found = false $\wedge$ restart = false
Use
$\forall nxt, addrkvL.$
$\langle$ cur $\mapsto$ node($\_, \_, nxt)_{addrkvL}$ $\rangle$
next := cur.next;
$\langle$ cur $\mapsto$ node($\_, \_, nxt)_{addrkvL}$*
next $\mapsto$ nodeOrMarkerOrNull($\_, \_, \_)_{addrkvL} \wedge$ next = nxt $\rangle$
$a \Rightarrow \blacklozenge * \exists addrkvL, k, k', k''.$
prev $\mapsto$ node($k, \_, \_)_{addrkvL}$ * cur $\mapsto$ nodeOrMarker($k', \_, \_)_{addrkvL}$*
next $\mapsto$ nodeOrMarkerOrNull($k'', \_, \_)_{addrkvL} \wedge$
$k <$ k $\wedge$ $k < k' < k'' \wedge$ found = false $\wedge$ restart = false

abstract, quantify a

make atomic and open region

abstract, and open region `bottomList`

Use
$\Vdash \forall value, addrkvL.$
$\langle$ `cur` $\mapsto$ `node`$\mathtt{OrMarker}(\_, value, \_)_{addrkvL}$ $\rangle$
`tv := cur.value;`
$\left\langle \begin{array}{l} (\mathtt{cur} \mapsto \mathtt{node}(\_, value, \_)_{addrkvL} \wedge \mathtt{tv} = value) \vee \\ (\mathtt{cur} \mapsto \mathtt{marker}(value)_{addrkvL} \wedge \mathtt{tv} = \delta) \end{array} \right\rangle$

$\left\{ \begin{array}{l} a \Mapsto \blacklozenge * \exists addrkvL, k, k', k''. \\ \mathtt{prev} \mapsto \mathtt{node}(k, \_, \_)_{addrkvL} * \mathtt{cur} \mapsto \mathtt{nodeOrMarker}(k', \_, \_)_{addrkvL} * \\ (\mathtt{next} \mapsto \mathtt{nodeOrMarker}(k'', \_, \_)_{addrkvL} \vee \mathtt{next} = \mathtt{null}) \wedge \\ k < \mathtt{k} \wedge k < k' < k'' \wedge \mathtt{found} = \mathtt{false} \wedge \mathtt{restart} = \mathtt{false} \end{array} \right\}$

// notice we also have $\mathtt{tv} = \delta \vee \mathtt{tv} \neq \delta$
// and δ is a special heap cell, which means $\delta \neq \mathtt{null}$
`if (tv = null) {`

$\left\{ \begin{array}{l} a \Mapsto \blacklozenge * \exists addrkvL, k, k', k''. \\ \mathtt{prev} \mapsto \mathtt{node}(k, \_, \_)_{addrkvL} * \mathtt{cur} \mapsto \mathtt{node}(k', \mathtt{tv}, \_)_{addrkvL} * \\ (\mathtt{next} \mapsto \mathtt{nodeOrMarker}(k'', \_, \_)_{addrkvL} \vee \mathtt{next} = \mathtt{null}) \wedge \\ k < \mathtt{k} \wedge k < k' < k'' \wedge \mathtt{tv} = \mathtt{null} \wedge \mathtt{found} = \mathtt{false} \wedge \\ \mathtt{restart} = \mathtt{false} \end{array} \right\}$

Use
$\Vdash \forall S, addrkvL, nxt_1, nxt_2, nxt_3.$
$\left\langle \begin{array}{l} \mathtt{prev} \mapsto \mathtt{node}(\_, \_, nxt_1)_{addrkvL} * \mathtt{cur} \mapsto \mathtt{node}(\_, \mathtt{null}, nxt_2)_{addrkvL} * \\ (\mathtt{next} \mapsto \mathtt{nodeOrMarker}(\_, \_, nxt_3)_{addrkvL} \vee \mathtt{next} = \mathtt{null}) \end{array} \right\rangle$
`helpDelete(prev, cur, next);`
$\langle$ $\dagger^1$ $\rangle$

$\left\{ \begin{array}{l} a \Mapsto \blacklozenge * \exists addrkvL, k, k', k'', k''', nxt. \\ \mathtt{prev} \mapsto \mathtt{node}(k, \_, \_)_{addrkvL} * \mathtt{cur} \mapsto \mathtt{node}(k', \mathtt{tv}, \_)_{addrkvL} * \\ (\mathtt{next} \mapsto \mathtt{nodeOrMarker}(k''', \_, \_)_{addrkvL} \vee \mathtt{next} = \mathtt{null}) \wedge \\ k < \mathtt{k} \wedge k < k' < k'' < k''' \wedge \mathtt{tv} = \mathtt{null} \wedge \mathtt{found} = \mathtt{false} \wedge \\ \mathtt{restart} = \mathtt{false} \end{array} \right\}$

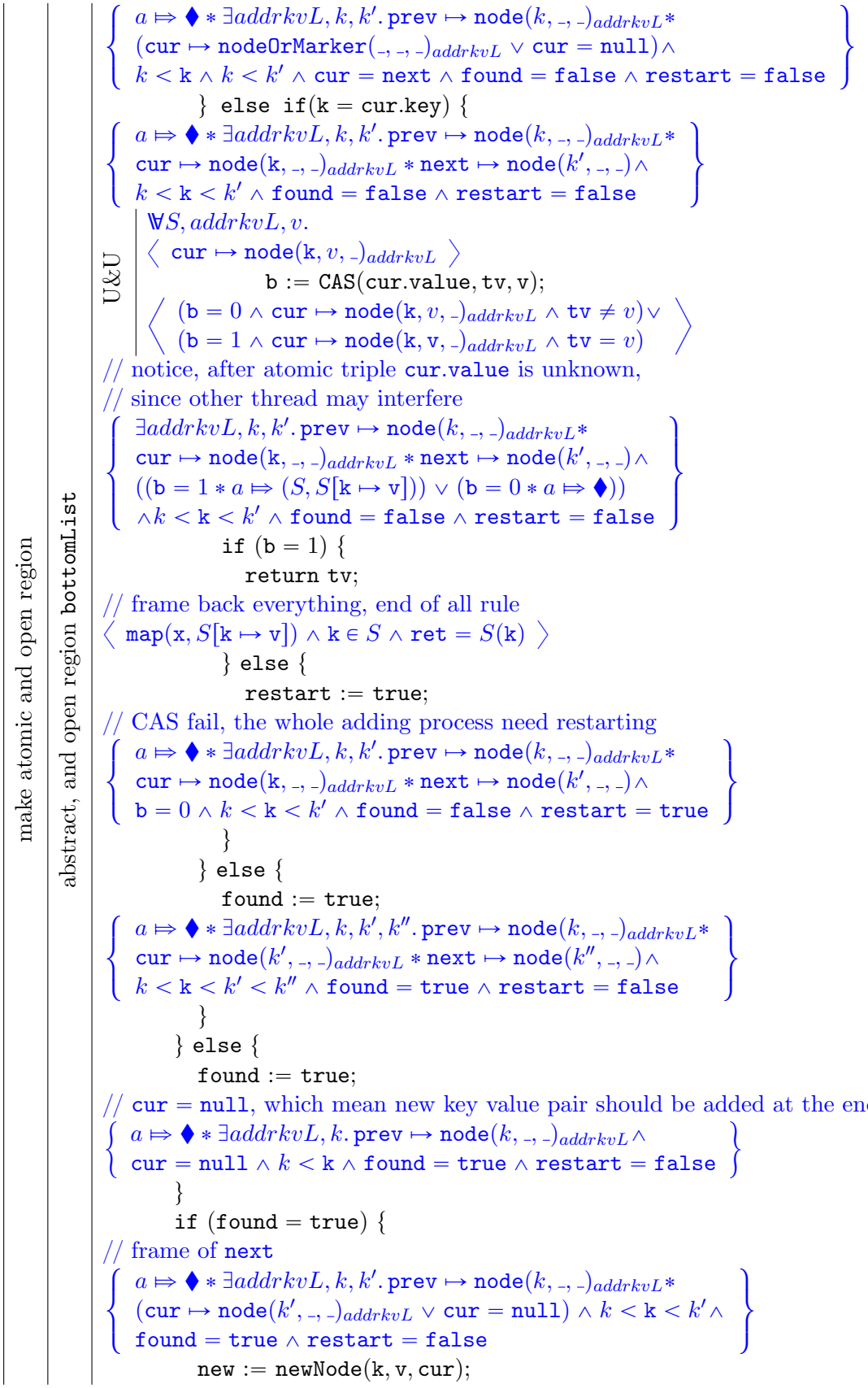
`restart := true;`
`} else if(isMarker(tv)||prev.value = null)) {`
`restart := true;`
// frame off other irrelevant part
$\left\{ \begin{array}{l} a \Mapsto \blacklozenge * \exists addrkvL. (\mathtt{prev} \mapsto \mathtt{node}(\_, \mathtt{null}, \_)_{addrkvL} \vee \\ \mathtt{cur} \mapsto \mathtt{marker}(\_)_{addrkvL}) \wedge \mathtt{found} = \mathtt{false} \wedge \mathtt{restart} = \mathtt{true} \end{array} \right\}$

`} else if(k > cur.key) {`

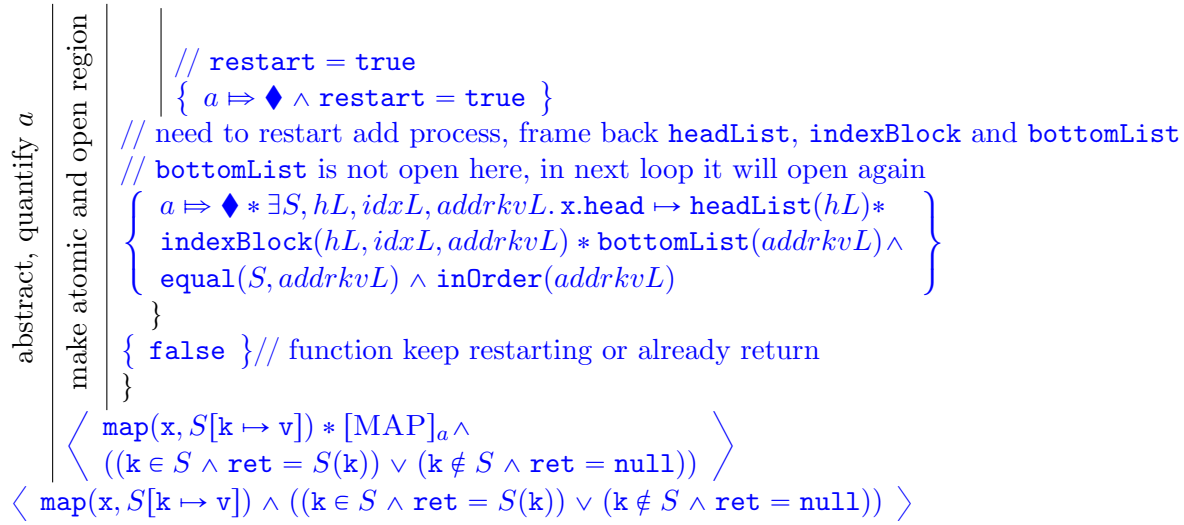
$\left\{ \begin{array}{l} a \Mapsto \blacklozenge * \exists addrkvL, k, k', k''. \\ \mathtt{prev} \mapsto \mathtt{node}(k, \_, \_)_{addrkvL} * \mathtt{cur} \mapsto \mathtt{node}(k', \_, \_)_{addrkvL} * \\ (\mathtt{next} \mapsto \mathtt{nodeOrMarker}(k'', \_, \_)_{addrkvL} \vee \mathtt{next} = \mathtt{null}) \wedge \\ k' < \mathtt{k} \wedge k < k' < k'' \wedge \mathtt{found} = \mathtt{false} \wedge \mathtt{restart} = \mathtt{false} \end{array} \right\}$

`prev := cur;`
`cur := next;`

$\dagger^1$ $(\mathtt{prev} \mapsto \mathtt{node}(\_, \_, nxt_3)_{addrkvL} * \mathtt{cur} \mapsto \mathtt{node}(\_, \mathtt{null}, nxt'_2)_{addrkvL} *$
$\mathtt{next} \mapsto \mathtt{marker}(nxt_3)_{addrkvL} \wedge nxt_1 = \mathtt{cur} \wedge nxt_2 = \mathtt{next}) \vee$
$(\exists newmarker. \mathtt{prev} \mapsto \mathtt{node}(\_, \_, nxt_1)_{addrkvL} * \mathtt{cur} \mapsto \mathtt{node}(\_, \mathtt{null}, newmarker)_{addrkvL} *$
$newmarker \mapsto \mathtt{marker}(nxt_2) * (\mathtt{next} \mapsto \mathtt{nodeOrMarker}(\_, \_, nxt_3)_{addrkvL} \vee \mathtt{next} = \mathtt{null}) \wedge$
$nxt_1 = \mathtt{cur} \wedge nxt_2 = \mathtt{next}) \vee (\mathtt{prev} \mapsto \mathtt{node}(\_, \_, nxt_1)_{addrkvL} *$
$\mathtt{cur} \mapsto \mathtt{node}(\_, \mathtt{null}, nxt_2)_{addrkvL} * (\mathtt{next} \mapsto \mathtt{nodeOrMarker}(\_, \_, nxt_3)_{addrkvL} \vee$
$\mathtt{next} = \mathtt{null}) \wedge (nxt_1 \neq \mathtt{cur} \vee nxt_2 \neq \mathtt{next}))$


abstract, quantify a
make atomic and open region
abstract, and open region bottomList
U&U
$\{\ a \Mapsto \blacklozenge * \exists addrkvL, k, k'.\ \texttt{prev} \mapsto \texttt{node}(k, \_, \_)_{addrkvL} *$
$(\texttt{cur} \mapsto \texttt{nodeOrMarker}(\_, \_, \_)_{addrkvL} \vee \texttt{cur} = \texttt{null}) \wedge$
$k < \texttt{k} \wedge k < k' \wedge \texttt{cur} = \texttt{next} \wedge \texttt{found} = \texttt{false} \wedge \texttt{restart} = \texttt{false}\ \}$
} else if(k = cur.key) {
$\{\ a \Mapsto \blacklozenge * \exists addrkvL, k, k'.\ \texttt{prev} \mapsto \texttt{node}(k, \_, \_)_{addrkvL} *$
$\texttt{cur} \mapsto \texttt{node}(\texttt{k}, \_, \_)_{addrkvL} * \texttt{next} \mapsto \texttt{node}(k', \_, \_) \wedge$
$k < \texttt{k} < k' \wedge \texttt{found} = \texttt{false} \wedge \texttt{restart} = \texttt{false}\ \}$
$\forall S, addrkvL, v.$
$\langle\ \texttt{cur} \mapsto \texttt{node}(\texttt{k}, v, \_)_{addrkvL}\ \rangle$
b := CAS(cur.value, tv, v);
$\langle\ (\texttt{b} = 0 \wedge \texttt{cur} \mapsto \texttt{node}(\texttt{k}, v, \_)_{addrkvL} \wedge \texttt{tv} \neq v) \vee$
$(\texttt{b} = 1 \wedge \texttt{cur} \mapsto \texttt{node}(\texttt{k}, \texttt{v}, \_)_{addrkvL} \wedge \texttt{tv} = v)\ \rangle$
// notice, after atomic triple cur.value is unknown,
// since other thread may interfere
$\{\ \exists addrkvL, k, k'.\ \texttt{prev} \mapsto \texttt{node}(k, \_, \_)_{addrkvL} *$
$\texttt{cur} \mapsto \texttt{node}(\texttt{k}, \_, \_)_{addrkvL} * \texttt{next} \mapsto \texttt{node}(k', \_, \_) \wedge$
$((\texttt{b} = 1 * a \Mapsto (S, S[\texttt{k} \mapsto \texttt{v}])) \vee (\texttt{b} = 0 * a \Mapsto \blacklozenge))$
$\wedge k < \texttt{k} < k' \wedge \texttt{found} = \texttt{false} \wedge \texttt{restart} = \texttt{false}\ \}$
if (b = 1) {
return tv;
// frame back everything, end of all rule
$\langle\ \texttt{map}(\texttt{x}, S[\texttt{k} \mapsto \texttt{v}]) \wedge \texttt{k} \in S \wedge \texttt{ret} = S(\texttt{k})\ \rangle$
} else {
restart := true;
// CAS fail, the whole adding process need restarting
$\{\ a \Mapsto \blacklozenge * \exists addrkvL, k, k'.\ \texttt{prev} \mapsto \texttt{node}(k, \_, \_)_{addrkvL} *$
$\texttt{cur} \mapsto \texttt{node}(\texttt{k}, \_, \_)_{addrkvL} * \texttt{next} \mapsto \texttt{node}(k', \_, \_) \wedge$
$\texttt{b} = 0 \wedge k < \texttt{k} < k' \wedge \texttt{found} = \texttt{false} \wedge \texttt{restart} = \texttt{true}\ \}$
}
} else {
found := true;
$\{\ a \Mapsto \blacklozenge * \exists addrkvL, k, k', k''.\ \texttt{prev} \mapsto \texttt{node}(k, \_, \_)_{addrkvL} *$
$\texttt{cur} \mapsto \texttt{node}(k', \_, \_)_{addrkvL} * \texttt{next} \mapsto \texttt{node}(k'', \_, \_) \wedge$
$k < \texttt{k} < k' < k'' \wedge \texttt{found} = \texttt{true} \wedge \texttt{restart} = \texttt{false}\ \}$
}
} else {
found := true;
// cur = null, which mean new key value pair should be added at the end
$\{\ a \Mapsto \blacklozenge * \exists addrkvL, k.\ \texttt{prev} \mapsto \texttt{node}(k, \_, \_)_{addrkvL} \wedge$
$\texttt{cur} = \texttt{null} \wedge k < \texttt{k} \wedge \texttt{found} = \texttt{true} \wedge \texttt{restart} = \texttt{false}\ \}$
}
if (found = true) {
// frame of next
$\{\ a \Mapsto \blacklozenge * \exists addrkvL, k, k'.\ \texttt{prev} \mapsto \texttt{node}(k, \_, \_)_{addrkvL} *$
$(\texttt{cur} \mapsto \texttt{node}(k', \_, \_)_{addrkvL} \vee \texttt{cur} = \texttt{null}) \wedge k < \texttt{k} < k' \wedge$
$\texttt{found} = \texttt{true} \wedge \texttt{restart} = \texttt{false}\ \}$
new := newNode(k, v, cur);


abstract, quantify a
make atomic and open region
abstract, and open region bottomList
U&U
Use
$\left\{ a \mapsto \blacklozenge * \exists addrkvL, k, k'.\, \texttt{prev} \mapsto \texttt{node}(k, \_, \_)_{addrkvL} * (\texttt{cur} \mapsto \texttt{node}(k', \_, \_)_{addrkvL} \vee \texttt{cur} = \texttt{null}) * \texttt{new} \mapsto \texttt{node}(\texttt{k}, \texttt{v}, \texttt{cur})^{\blacktriangle} \wedge k < \texttt{k} < k' \wedge \texttt{found} = \texttt{true} \wedge \texttt{restart} = \texttt{false} \right\}$
$\mathbb{W}S, addrkvL, nxt.$
$\langle\ \texttt{prev} \mapsto \texttt{node}(\_, \_, nxt)_{addrkvL}\ \rangle$
b := CAS(prev.next, cur, new);
$\left\langle \begin{array}{l} (\texttt{b} = 0 \wedge \texttt{prev} \mapsto \texttt{node}(\_, \_, nxt)_{addrkvL} \wedge nxt \neq \texttt{cur}) \vee \\ (\texttt{b} = 1 \wedge \texttt{prev} \mapsto \texttt{node}(\_, \_, \texttt{next})_{addrkvL} \wedge nxt = \texttt{cur}) \end{array} \right\rangle$
// notice, after atomic triple $nxt \neq$ cur is not valid,
// since other thread may reconnect cur
$\left\{ \begin{array}{l} \exists addrkvL, k, k'.\, \texttt{prev} \mapsto \texttt{node}(k, \_, \_)_{addrkvL} * (\texttt{cur} \mapsto \texttt{node}(k', \_, \_)_{addrkvL} \vee \\ \texttt{cur} = \texttt{null}) \wedge ((\texttt{b} = 0 * a \mapsto \blacklozenge * \texttt{new} \mapsto \texttt{node}(\texttt{k}, \texttt{v}, \texttt{cur})^{\blacktriangle}) \vee (\texttt{b} = 1 * \\ a \mapsto (S, S[\texttt{k} \mapsto \texttt{v}]) * \texttt{new} \mapsto \texttt{node}(\texttt{k}, \_, \_)_{adddrkvL})) \wedge \\ k < \texttt{k} < k' \wedge \texttt{found} = \texttt{true} \wedge \texttt{restart} = \texttt{false} \end{array} \right\}$
if (b = 0) {
restart := true;
// frame back next and frame off new
$\left\{ \begin{array}{l} a \mapsto \blacklozenge * \exists addrkvL, k, k', k''.\, \texttt{prev} \mapsto \texttt{node}(k, \_, \_)_{addrkvL} * \\ (\texttt{cur} \mapsto \texttt{node}(k', \_, \_)_{addrkvL} \vee \texttt{cur} = \texttt{null}) * \texttt{nextmapnode}(k'', \_, \_) \wedge \\ \texttt{b} = 0 \wedge k < k' < k'' \wedge \texttt{found} = \texttt{true} \wedge \texttt{restart} = \texttt{true} \end{array} \right\}$
} else {
// frame off other part except new
$\left\{ \begin{array}{l} a \mapsto (S, S[\texttt{k} \mapsto \texttt{v}]) * \exists addrkvL.\, \texttt{new} \mapsto \texttt{node}(\texttt{k}, \_, \_)_{adddrkvL} \wedge \\ \texttt{b} = 1 \wedge \texttt{found} = \texttt{true} \wedge \texttt{restart} = \texttt{false} \end{array} \right\}$
level := randomLevel();
if (level > 0) {
// frame back headList and indexBlock
$\left\{ \begin{array}{l} a \mapsto (S, S[\texttt{k} \mapsto \texttt{v}]) * \exists hL, idxL, addrkvL.\, \texttt{x.head} \mapsto \texttt{headList}(hL) * \\ \texttt{indexBlock}(hL, idxL, addrkvL) * \texttt{new} \mapsto \texttt{node}(\texttt{k}, \_, \_)_{adddrkvL} \end{array} \right\}$
// indexBlock is normal predicate,
// which wrap index list from 1 to level
$\mathbb{W}S, addrkvL, level, hL, idxL_{(1,level)}.$
$\langle\ \texttt{x.head} \mapsto \texttt{headList}(hL) * \texttt{indexBlock}(hL, idxL_{level}, addrkvL)\ \rangle$
insertindex(x, new, level);
// function preserve $addrkvL$ and thus S
$\left\langle \begin{array}{l} \exists level', hL', idxL'_{(1,level')}. \\ \texttt{x.head} \mapsto \texttt{headList}(hL') * \texttt{indexBlock}(hL', idxL'_{level}, addrkvL) \end{array} \right\rangle$
$\left\{ \begin{array}{l} a \mapsto (S, S[\texttt{k} \mapsto \texttt{v}]) * \exists hL, idxL, addrkvL.\, \texttt{x.head} \mapsto \texttt{headList}(hL) * \\ \texttt{indexBlock}(hL, idxL, addrkvL) * \texttt{new} \mapsto \texttt{node}(\texttt{k}, \_, \_)_{adddrkvL} \end{array} \right\}$
}
return null;
// end of all rule
$\langle\ \texttt{map}(\texttt{x}, S[\texttt{k} \mapsto \texttt{v}]) \wedge \texttt{k} \notin S \wedge \texttt{ret} = \texttt{null}\ \rangle$
}
}
}


abstract, quantify a
make atomic and open region
// restart = true
{ $a \Rightarrow \blacklozenge \wedge$ restart = true }
// need to restart add process, frame back headList, indexBlock and bottomList
// bottomList is not open here, in next loop it will open again
$a \Rightarrow \blacklozenge * \exists S, hL, idxL, addrkvL.$ x.head $\mapsto$ headList(hL)*
indexBlock($hL, idxL, addrkvL$) * bottomList($addrkvL$)$\wedge$
equal($S, addrkvL$) $\wedge$ inOrder($addrkvL$)
}
{ false }// function keep restarting or already return
}
map(x, S[k $\mapsto$ v]) * [MAP]$_a \wedge$
(k $\in S \wedge$ ret = S(k)) $\vee$ (k $\notin S \wedge$ ret = null))
map(x, S[k $\mapsto$ v]) $\wedge$ ((k $\in S \wedge$ ret = S(k)) $\vee$ (k $\notin S \wedge$ ret = null))

Chapter 6

Evaluation

This chapter discusses why it is difficult to give a good map specification for clients. After, it presents some interesting finding in proofs of implementation such as advantages of abstraction, termination property, and similar proving style.

6.1 Specification for Client

We have shown two specifications for concurrent map, and have tried to argue that map-based one is slightly better. But we have found out it is rather difficult to argue which one is a better specification. This discussions leads us to a question, what is a good specification. In some case it is trivial, for instance, a counter module that only allows increasing value. But for complicate module such as concurrent map, the story is different. One can choose to think about map as a big box, i.e. a map based specification. Others can also think about as key value pairs, which reflects the idea that minimum is better, like what separation logic does. We have believed that it is a better to present as map-based way for a map for general purposes, which typically is a module provided by library. Because it is a relatively direct way to describe, and highly possible that programmers consider them as a big box as well. However, in particular applications, for instance intermediate maps used in Map/Reduce algorithm, key value specifications win. At last, we think there could be even other form of specification based on some particular application needs, but those special forms could be built by map-based style.

There is no universal standard to specify a map module, which also causes our specification difficulty. In other research area, researchers also use formal way to model and verify, but the difference is that those works typically model and verify those objects with a very restrict standard, for instance C++, TCP/IP, etc. Therefore in those works specifications are easy tasks that reflects what standard requires. But here to argue which specification is better for a concurrent map module, what we have done right now is discussion based on people's experience. We can also cheat by arguing that we use Java map standard, and how Java describe them is map-based style instead of key value style, therefore, map based specification is better, but it seems not a very solid argument.

If we take one step back, only thinking about concurrency, thus both key value style and map-based style are able to show linearizability. Besides, §3.3 have proven that one can transfer to another one by building one more abstract layer. So those two forms of specification are both good enough to show linearizability.

To conclude, we cannot give a strong answer for what is the best specification for concurrent map.

In this thesis, we have chosen to use map-based style rather than key value pair style, but we have shown that they are equivalent, so it remains as a question for what does it means for a good specification.

6.2 Abstraction and Atomic Triple

Abstraction is a very interesting features of proofs presented at §4 §5. First, Abstraction allows building a extra layer between clients and implementation. Client should only know what a module choose to reveal, for example the concurrent map choose to reveal the information in mathematical way. By Abstraction implementation also have more freedom. Because as long as a implementation satisfy the abstract description of a module, they are free to change the implementation details. This is very similar with OO-style programs, for clients they only know those information is public, but for implementation they know every details.

Apart from the first level abstraction, multiple level abstractions at implementation, even though it is not necessary, helps to understand the algorithm better. In fact, many implementations are also organised in several layer and use other low level interfaces, for example B^{link}tree has three levels of abstraction and use low level file system interface.

TaDA can also proof linearizability point with abstraction, which is a very interesting combination. Because for many data structure, when talking linearizability for instance stack, it actually refers to a abstract stack, and discuss the linearizability upon this abstract form of stack. TaDA uses atomic triple to explicitly describe the linearizability and what happen in the linearizability point. Our practical works, proofs of two implementations, have shown that for a very complicate example, TaDA are able to proof linearizability whereas other linearizability works usually choose simple examples.

In conclusion, for the purpose of function correctness, abstraction is key to simplify specification and hiding the implementation details. For concurrency, linearizability is a very important criteria. Thus, TaDA find a way to combine abstraction and linearizability, which is very important in term of reasoning about complicate concurrent programs.

6.3 Termination For Code Used In Industry

The motivation to shortly discuss termination is there is a extension version of TaDA, which can proof the termination property of non-blocking algorithm. But it is necessary to restrain the environment. Let takes Concurrent Skip List §5 as a example. Insertion algorithm of concurrent skip list keep trials until it succeeds. If we want to use the extension version of TaDA to proof termination property, intuitively we have to restrain environment in a way that asking clients to show that there are only finite deletions. In some cases it is reasonable, but for a module that potentially run forever in servers, no one can says that deletion is finite. Whereas presumably, operations of Java concurrent skip list has a very large probability that it terminates even though this module could run in a situation that has infinite deletion.

6.4 General Proving Style

We have tried to maintain similar proving style of B^{link}tree and skip list, potentially to provides more study cases in term of automatic proof in the future. Our proof follows steps with no complex guard algebra trick,

- Use make atomic rule and claim the updating premise.
- Keep opening region until reaching a certain level of abstraction.
- In this abstraction level, each step is straightforward as long as predicates are stable.
- After each use atomic rule, it ought to stabilise predicate, which is a process that weaken predicate.

We think it is possible to have some tools to help us doing this very tedious proof. But this tools may need manually inputting definitions of region and guards.

6.5 Simple Guards

TaDA is a complex proof system, one reason is guard algebra. But in our proofs, we have only used three guards for each region, and also we have chosen not parametrise those guards,

- Duplicatable guard, bound with behaviours does not require exclusive access.
- Unique guard, bound with behaviours requires exclusive access.
- An "locked" witness, a way to connect duplicatable guard and unique guard by guard algebra, and also a way to connect a module with other access control module, for instance the general lock module presented at §4.1.

Guard is introduced by TaDA to replace partial permission. In fact we have shown how to build a partial permission by using guard in §3.1. But the intuition is that we think it is too powerful, therefore to not confuse people, we think it is a good idea to use them in a very simple way.

Besides if one wants to do automatic proof using TaDA, one big challenge is be how to deal with guard algebra. One solution could be what we have presented here that restrain how to use guard. But it currently is only our intuition based on some proof examples.

Chapter 7

Conclusion

This chapter will recap contribution of project, and discuss some interesting points and unsolved questions.

7.1 Summary

This project consists of two parts, one is to discuss what is a good concurrent map specification for clients, another is to have a deep understand of TaDA by proving more complex and realistic algorithms, and what is the connection between specifications and implementations.

There is no universal way to specify a concurrent map, therefore it is very difficult to argue which specification is better. By intuition and some experience, we have believed map-based one is better, but actually other could also argue that key value style has its own useful place. Map-based one fit with our assumptions, that it is a general used concurrent map, but we have also shown that in some particular applications, for instance Map/Reduce algorithm, key value style seems to be easier to understand. However, there is no objective evident has been found, and we think judging a specification is highly subjective, as long as specification satisfies some correctness conditions, such as linearizability. Therefore what we have done is choosing one specification, and showing other style is equivalent with the one we choose.

The second part is implementations and their proofs, where we have proven two popular implementations of concurrent map, b-tree and skip list. One interesting point is that we have shown that abstract is a good method to deal with complicate data structure and its reasoning. In both proofs, many layers of abstraction help us to understand algorithms and simplify proofs as well. For example data structures used to implement concurrent map typically is rather difficult even though the general ideas behind those data structure are typically simple. With abstract we are able to present general ideas hide the unnecessary details and only present the general shape in higher abstract level but show more details in lower abstraction level. Besides we have tried to fix proving style and guard algebra, which gives us idea for how to automatically prove in the future. Those two proofs are good study cases in term of working on a automatic proof system with TaDA.

If we only think about the algorithms themselves, we have shown that those algorithms have a linearizability point. While many other works on linearizability choose very simple examples for instance stack or queue, our works have proven very complicate examples. We have shown that TaDA are capable to proof those examples with a reasonable size of description.

7.2 Future Work

There are many unsolved questions during this project, the first one is what does it means for good specifications. Even a very common concurrent map, we have found out at least two reasonable ways to specify it. We have believed that it is a debate about presenting as "a big box" or presenting as minimum information as possible. In concrete level, minimum information seems to be a good idea, but modules from library are better present in a big box way. However how to choose one of them as the basic specification remains as a question.

Second question: is there a possible way to prove automatically. In the proofs of two implementations, we have tried to fix proving style. We have found out that it is possible to prove automatically if dealing with guard algebra and abstraction properly. And we think it is good start based on those examples to discover a method for automatic proving.

At last, the question is can we reasoning about algorithms based on distributed system. This project have used Map/Reduce as an examples, which leads us to a very straightforward question: what if this map module is implemented or used by a distributed system. Following the idea, we would like to know if it is possible to use what we have right now to reasoning about distributed system, and what assumption we need in term of reasoning. So far, we think to restrain some fail situations, such as communication fail, it may possible to use current proof system to deal with distributed system, but details remain unknown.

Bibliography

[1] Bornat, R., C. Calcagno, P. O'Hearn, and M. Parkinson (2005). Permission accounting in separation logic. In *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, New York, NY, USA, pp. 259–270. ACM.

[2] Bornat, R., C. Calcagno, and H. Yang (2006, May). Variables as resource in separation logic. *Electron. Notes Theor. Comput. Sci. 155*, 247–276.

[3] Boyland, J. (2003). Checking interference with fractional permissions. In *Proceedings of the 10th International Conference on Static Analysis*, SAS'03, Berlin, Heidelberg, pp. 55–72. Springer-Verlag.

[4] Calcagno, C., P. W. O'Hearn, and H. Yang (2007). Local action and abstract separation logic. In *Proceedings of the 22Nd Annual IEEE Symposium on Logic in Computer Science*, LICS '07, Washington, DC, USA, pp. 366–378. IEEE Computer Society.

[5] Comer, D. (1979, June). Ubiquitous B-Tree. *ACM Comput. Surv. 11*(2), 121–137.

[6] da Rocha Pinto, P., T. Dinsdale-Young, M. Dodds, P. Gardner, and M. J. Wheelhouse (2011). A simple abstraction for complex concurrent indexes. In C. V. Lopes and K. Fisher (Eds.), *OOPSLA*, pp. 845–864. ACM.

[7] da Rocha Pinto, P., T. Dinsdale-Young, and P. Gardner (2014). TaDA: A logic for time and data abstractionse. In R. Jones (Ed.), *ECOOP 2014 Object-Oriented Programming*, Volume 8586 of *Lecture Notes in Computer Science*, pp. 207–231. Springer Berlin Heidelberg.

[8] Dean, J. and S. Ghemawat (2008, January). Mapreduce: Simplified data processing on large clusters. *Commun. ACM 51*(1), 107–113.

[9] Dinsdale-Young, T., M. Dodds, P. Gardner, M. J. Parkinson, and V. Vafeiadis (2010). Concurrent abstract predicates. In *Proceedings of the 24th European Conference on Object-oriented Programming*, ECOOP'10, Berlin, Heidelberg, pp. 504–528. Springer-Verlag.

[10] Dodds, M., X. Feng, M. Parkinson, and V. Vafeiadis (2009). Deny-Guarantee reasoning. In *Proceedings of the 18th European Symposium on Programming Languages and Systems: Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009*, ESOP '09, Berlin, Heidelberg, pp. 363–377. Springer-Verlag.

[11] Dongol, B. and J. Derrick (2014). Verifying linearizability: A comparative survey. *CoRR abs/1410.6268*.

[12] Floyd, R. W. (1967). Assigning meanings to programs. *Mathematical aspects of computer science 19*(19-32), 1.

[13] Harris, T. L. (2001). A pragmatic implementation of non-blocking linked-lists. In *Proceedings*

of the 15th International Conference on Distributed Computing, DISC '01, London, UK, UK, pp. 300–314. Springer-Verlag.

[14] Herlihy, M. P. and J. M. Wing (1990, July). Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst. 12*(3), 463–492.

[15] Hoare, C. A. R. (1969, October). An axiomatic basis for computer programming. *Commun. ACM 12*(10), 576–580.

[16] Ishtiaq, S. S. and P. W. O'Hearn (2001). BI as an assertion language for mutable data structures. In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '01, New York, NY, USA, pp. 14–26. ACM.

[17] Jones, C. B. (1983). Specification and design of (parallel) programs. In *IFIP Congress*, pp. 321–332.

[18] Jones, C. B. (2003). Wanted: A compositional approach to concurrency. In A. McIver and C. Morgan (Eds.), *Programming Methodology*, pp. 5–15. New York, NY, USA: Springer-Verlag New York, Inc.

[19] Kung, H. T. and P. L. Lehman (1980, September). Concurrent manipulation of binary search trees. *ACM Trans. Database Syst. 5*(3), 354–382.

[20] Michael, M. M. (2002). High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '02, New York, NY, USA, pp. 73–82. ACM.

[21] OHearn, P. W. (2007, April). Resources, concurrency, and local reasoning. *Theor. Comput. Sci. 375*(1-3), 271–307.

[22] Oracle. Java 8 api. `https://docs.oracle.com/javase/8/docs/api/`.

[23] Owicki, S. and D. Gries (1976, December). An axiomatic proof technique for parallel programs i. *Acta Informatica 6*(4), 319–340.

[24] Papadakis, T. (1993). *Skip Lists and Probabilistic Analysis of Algorithm*. Ph. D. thesis, University of Waterloo, Canada.

[25] Parkinson, M. and G. Bierman (2005). Separation logic and abstraction. In *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, New York, NY, USA, pp. 247–258. ACM.

[26] Pugh, W. (1990a). Concurrent maintenance of skip lists. Technical report, Dept. of Computer Science, University of Maryland, College Park.

[27] Pugh, W. (1990b, June). Skip lists: A probabilistic alternative to balanced trees. *Commun. ACM 33*(6), 668–676.

[28] Raad, A., J. Villard, and G. Philippa (2015). CoLoSL: Concurrent local subjective logic. In *European Symposium on Programming*, ESOP '15.

[29] Reynolds, J. C. (2000). Intuitionistic reasoning about shared mutable data structure. In *Millennial Perspectives in Computer Science*, pp. 303–321. Palgrave.

[30] Reynolds, J. C. (2002). Separation Logic: A logic for shared mutable data structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, LICS '02, Washington, DC, USA, pp. 55–74. IEEE Computer Society.

[31] Sagiv, Y. (1985). Concurrent operations on b-trees with overtaking. In *Proceedings of the Fourth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, PODS '85, New York, NY, USA, pp. 28–37. ACM.

[32] Selivanov, Y. Python dictionary implementation. `https://hg.python.org/cpython/file/tip/Objects/dictobject.c`.

[33] Vafeiadis, V. and M. Parkinson (2007). A marriage of Rely/Guarantee and Separation Logic. In *IN 18TH CONCUR*, pp. 256–271. Springer.

[34] Webpage. C++ standard library. `http://en.cppreference.com/w/`.

[35] Webpage. Concurrentskiplistmap by openjdk-7. `http://www.docjar.com/html/api/java/util/concurrent/ConcurrentSkipListMap.java.html`.

Appendix A

Concurrent Concurrent B^{link}tree

A.1 Specification of Auxiliary Functions

The following specification of auxiliary functions are rather trivial agaist their name. Function `new` returns a new node in file system. From `lowKey` to `lastPointer` are functions operate on key value pairs, but do not touch marking bit. `hasState`, `changePointerState` and `firstState` are three functions manipulate marking bit, which are used by compression. There are also three functions for prime block and four functions for standard stack operation.

$$\{\texttt{emp}\} \quad \texttt{new}() \quad \{\texttt{ret} \mapsto \texttt{node}(0, 0, [], +\infty, \texttt{null}, 0)\}$$

$$\{\texttt{emp}\} \quad \texttt{newRoot}(\texttt{k0}, \texttt{v0}, \texttt{k1}, \texttt{v1}, \texttt{mxk}) \quad \begin{Bmatrix} \texttt{ret} = \texttt{node}(0, 0, kvmL, \texttt{mxk}, \texttt{null}, 0) \wedge \\ kvmL = [(\texttt{k0}, \texttt{v}_0, \texttt{false}), (\texttt{k1}, \texttt{v1}, \texttt{false})] \end{Bmatrix}$$

$$\{\texttt{x} = \texttt{node}(\_, \_, \_, \_, \_, \_)\} \quad \texttt{lowKey}(\texttt{x}) \quad \begin{Bmatrix} \texttt{x} = \texttt{node}(\_, \_, kvmL, \_, \_, \_) \wedge \\ \texttt{ret} = \texttt{first}(kvmL)\downarrow_1 \end{Bmatrix}$$

$$\{\texttt{x} = \texttt{node}(\_, \_, \_, \_, \_, \_)\} \quad \texttt{highKey}(\texttt{x}) \quad \{\texttt{x} = \texttt{node}(\_, \_, \_, \texttt{ret}, \_, \_)\}$$

$$\{\texttt{x} = \texttt{node}(\_, \_, \_, \_, \_, \_)\} \quad \texttt{lookUp}(\texttt{x}, \texttt{k}) \quad \begin{Bmatrix} \texttt{x} = \texttt{node}(\_, \_, kvmL, \_, \_, \_) \wedge \\ ((\texttt{k}, \texttt{ret}, \_) \in kvmL \vee \\ ((\texttt{k}, \_, \_) \notin kvmL \wedge \texttt{ret} = \texttt{null})) \end{Bmatrix}$$

$$\begin{Bmatrix} \texttt{x} = \texttt{node}(\_, \_, kvmL, \_, \_, \_) \wedge \\ (\texttt{k}, \_, \_) \notin kvmL \wedge |kvmL| < \texttt{max} \end{Bmatrix} \quad \texttt{addPair}(\texttt{x}, \texttt{k}, \texttt{v}) \quad \begin{Bmatrix} \texttt{x} = \texttt{node}(\_, \_, kvmL \uplus [(\texttt{k}, \texttt{v}, \texttt{false})], \_, \_, \_) \wedge \\ (\texttt{k}, \_, \_) \notin kvmL \wedge \texttt{ret} = \texttt{v} \end{Bmatrix}$$

$$\begin{Bmatrix} \texttt{x} = \texttt{node}(\_, \_, kvmL, \_, \_) \wedge \\ (\texttt{k}, \_, \_) \in kvmL \end{Bmatrix} \quad \texttt{removePair}(\texttt{x}, \texttt{k}) \quad \begin{Bmatrix} \texttt{x} = \texttt{node}(\_, \_, kvmL \backslash [(\texttt{k}, \texttt{ret}, \_)], \_, \_) \wedge \\ (\texttt{k}, v, \_) \in kvmL \end{Bmatrix}$$

$$\{\texttt{x} = \texttt{node}(\_, \_, \_, \_, \_, \_)\} \quad \texttt{isSafe}(\texttt{x}) \quad \begin{Bmatrix} \texttt{x} = \texttt{node}(\_, \_, kvmL, \_, \_, \_) \wedge \\ ((|kvmL| < \texttt{max} \wedge \texttt{ret} = \texttt{true}) \vee \\ (|kvmL| \geqslant \texttt{max} \wedge \texttt{ret} = \texttt{false})) \end{Bmatrix}$$

$$\{\texttt{x} = \texttt{node}(\_, \_, \_, \_, \_, \_)\} \quad \texttt{isIn}(\texttt{x}, \texttt{k}) \quad \begin{Bmatrix} \texttt{x} = \texttt{node}(\_, \_, kvmL, \_, \_, \_) \wedge \\ (((\texttt{k}, \_, \_) \in kvmL \wedge \texttt{ret} = \texttt{true}) \vee \\ ((\texttt{k}, \_, \_) \notin kvmL \wedge \texttt{ret} = \texttt{false})) \end{Bmatrix}$$

$$\{\texttt{x} = \texttt{node}(\_, \_, \_, \_, \_, \_)\} \quad \texttt{isLeaf}(\texttt{x}) \quad \begin{Bmatrix} \texttt{x} = \texttt{node}(\_, type, \_, \_, \_, \_) \wedge \\ ((type = 1 \wedge \texttt{ret} = \texttt{true}) \vee \\ (type = 0 \wedge \texttt{ret} = \texttt{false})) \end{Bmatrix}$$

$\{\mathtt{x} = \mathtt{node}(\_,\_,\_,\_,\_,\_)\}$ $\quad \mathtt{lastPointer}(\mathtt{x}) \quad$ $\begin{Bmatrix}\mathtt{x} = \mathtt{node}(\_,\_,kvmL,\_,\_,\_)\wedge\\ \mathtt{ret} = \mathtt{last}(kvmL)\downarrow_2\end{Bmatrix}$

$\{\mathtt{x} = \mathtt{node}(\_,\_,\_,\_,\_,\_)\}$ $\quad \mathtt{hasState}(\mathtt{x},\mathtt{b}) \quad$ $\begin{Bmatrix}\mathtt{x} = \mathtt{node}(\_,\_,kvmL,\_,\_,\_)\wedge\\ (((\_,\_,\mathtt{b}) \in kvmL \wedge \mathtt{ret} = \mathtt{true})\vee\\ (\bigwedge_{(\_,\_,b)\in kvmL} b = \neg\mathtt{b} \wedge \mathtt{ret} = \mathtt{false}))\end{Bmatrix}$

$\begin{Bmatrix}\mathtt{x} = \mathtt{node}(\_,\_,kvmL,\_,\_,\_)\wedge\\ (\mathtt{y},\_,\_) \in kvmL\end{Bmatrix}$ $\quad \mathtt{changePointerState}(\mathtt{x},\mathtt{y},\mathtt{b})$

$\begin{Bmatrix}\mathtt{x} = \mathtt{node}(\_,\_,kvmL\backslash[(\mathtt{y},\_,\_)] \uplus [(\mathtt{y},\_,\mathtt{b})],\_,\_,\_)\wedge\\ (\mathtt{y},\_,\_) \in kvmL\end{Bmatrix}$

$\{\mathtt{x} = \mathtt{node}(\_,\_,kvmL,\_,\_,\_)\}$ $\quad \mathtt{firstState}(\mathtt{x},\mathtt{b})$

$\begin{Bmatrix}\mathtt{x} = \mathtt{node}(\_,\_,kvmL,\_,\_,\_) \wedge \exists v_i, b_i.\\ kvmL = [(\_,v_1,b_1),\ldots,(\_,v_{|kvmL|},b_{|kvmL|})]\wedge\\ ((\bigwedge_{1\leqslant i\leqslant |kvmL|} b_i = \neg\mathtt{b} \wedge \mathtt{ret} = \mathtt{null})\vee\\ (\exists m.\, \bigwedge_{1\leqslant i\leqslant m} b_i = \neg\mathtt{b} \wedge b_{m+1} = \mathtt{b} \wedge \mathtt{ret} = v_{m+1}))\end{Bmatrix}$

$\{\mathtt{x} = \mathtt{primeBlock}(pbL)\}$ $\quad \mathtt{root}(\mathtt{x}) \quad$ $\begin{Bmatrix}\mathtt{x} = \mathtt{primeBlock}(pbL)\wedge\\ \mathtt{ret} = \mathtt{last}(pbL)\end{Bmatrix}$

$\{\mathtt{x} = \mathtt{primeBlock}(pbL)\}$ $\quad \mathtt{getNodeLevel}(\mathtt{x},\mathtt{i}) \quad$ $\begin{Bmatrix}\mathtt{x} = \mathtt{primeBlock}(pbL)\wedge\\ \mathtt{ret} = \mathtt{at}(pbL,\mathtt{i})\end{Bmatrix}$

$\begin{Bmatrix}\mathtt{x} = \mathtt{primeBlock}(pbL)*\\ \mathtt{r} \mapsto \mathtt{node}(\_,\_,\_,\_,\_,\_)\end{Bmatrix}$ $\quad \mathtt{addRoot}(\mathtt{x},\mathtt{r}) \quad$ $\begin{Bmatrix}\mathtt{x} = \mathtt{primeBlock}(pbL +\!\!+ [\mathtt{r}])*\\ \mathtt{r} \mapsto \mathtt{node}(\_,\_,\_,\_,\_,\_)\end{Bmatrix}$

$\{\mathtt{emp}\}$ $\quad \mathtt{newStack}() \quad$ $\{\mathtt{stack}(\mathtt{ret},[])\}$

$\{\mathtt{stack}(\mathtt{x},L)\}$ $\quad \mathtt{push}(\mathtt{x},\mathtt{v}) \quad$ $\{\mathtt{stack}(\mathtt{x},[\mathtt{v}] +\!\!+ L)\}$

$\{\mathtt{stack}(\mathtt{x},[v] +\!\!+ L)\}$ $\quad \mathtt{pop}(\mathtt{x}) \quad$ $\{\mathtt{stack}(\mathtt{x},L) \wedge \mathtt{ret} = v\}$

$\{\mathtt{stack}(\mathtt{x},L)\}$ $\quad \mathtt{isEmpty}(\mathtt{x}) \quad$ $\begin{Bmatrix}\mathtt{stack}(\mathtt{x},L) \wedge ((L = [] \wedge \mathtt{ret} = \mathtt{true})\vee\\ (L \neq [] \wedge \mathtt{ret} = \mathtt{false}))\end{Bmatrix}$

A.2 Proof of Concurrent B^{link}tree Insertion

A.2.1 optimistic recording path

$\begin{Bmatrix}\exists l, l', \mathcal{P}, \mathcal{P}', level.\, \mathtt{untouchedList}(1) * \mathtt{stackProperty}(level)*\\ \mathtt{cur} \mapsto \mathtt{node}(l,\mathcal{P}) * \mathtt{curn} = \mathtt{node}(l',\mathcal{P}')^{\blacktriangle}\end{Bmatrix}$

// frame off cur

goDownAndPushStack{

while (isLeaf(curn) = false) {

// node's type = 0, because it is inner node, besides, level > 1.

$\begin{Bmatrix}\exists level.\, \mathtt{untouchedList}(1) * \mathtt{stackProperty}(level)*\\ \mathtt{curn} = \mathtt{node}(\_,\_,\_,\_,\_,\_)^{\blacktriangle} \wedge level > 1\end{Bmatrix}$

if (k < highKey(curn)) {

push(stack, cur);

$\begin{Bmatrix}\exists level, mxk.\, \mathtt{untouchedList}(1) * \mathtt{stackProperty}(level-1)*\\ \mathtt{curn} = \mathtt{node}(\_,\_,\_,mxk,\_,\_)^{\blacktriangle} \wedge \mathtt{k} < mxk\end{Bmatrix}$

}

$\begin{Bmatrix}\exists level, mxk.\, \mathtt{untouchedList}(1) * \mathtt{stackProperty}(level')*\\ \mathtt{curn} = \mathtt{node}(\_,\_,\_,mxk,\_,\_)^{\blacktriangle}\end{Bmatrix}$

cur := next(curn, k);

$\{\ \exists level.\, \texttt{untouchedList}(1) * \texttt{stackProperty}(level) * \texttt{curn} = \texttt{node}(\_,\_,\_,\_,\_,\_)^{\blacktriangle}\ \}$

Use
$\forall l, \mathcal{P}.$
$\langle\ \texttt{cur} \mapsto \texttt{node}(l, \mathcal{P}) * \texttt{curn} = \_\ \rangle$
$\quad \texttt{curn} := \texttt{get}(\texttt{cur});$
$\langle\ \texttt{cur} \mapsto \texttt{node}(l, \mathcal{P}) * \texttt{curn} = \texttt{node}(l, \mathcal{P})\ \rangle$

$\{\ \exists level.\, \texttt{untouchedList}(1) * \texttt{stackProperty}(level) * \texttt{curn} = \texttt{node}(\_,\_,\_,\_,\_,\_)^{\blacktriangle}\ \}$
}
}
$\left\{\begin{array}{l} \exists l, l', \mathcal{P}, \mathcal{P}'.\, \texttt{untouchedList}(1) * \texttt{stackProperty}(0) * \\ \texttt{cur} \mapsto \texttt{node}(l, \mathcal{P}) * \texttt{curn} = \texttt{node}(l', \mathcal{P}')^{\blacktriangle} \end{array}\right\}$

A.2.2 find and lock node

// frame off stackProperty(level)
$\left\{\begin{array}{l} \texttt{regionState} * \texttt{primeBlockProperty} * \texttt{untouchedList}(\texttt{level}) * \\ \texttt{addrLkvLProperty} * \texttt{cur} \mapsto \texttt{node}(\_,\_,\_,\_,\_,\_) \end{array}\right\}$
findNode{
found := false;
$\left\{\begin{array}{l} \texttt{regionState} * \texttt{primeBlockProperty} * \texttt{untouchedList}(\texttt{level}) * \\ \texttt{addrLkvLProperty} * \texttt{cur} \mapsto \texttt{node}(\_,\_,\_,\_,\_,\_) \wedge \texttt{found} = \texttt{false} \end{array}\right\}$

Use
$\forall l, \mathcal{P}.$
$\langle\ \texttt{cur} \mapsto \texttt{node}(l, \mathcal{P})\ \rangle$
$\quad \texttt{lock}(\texttt{cur});$
$\langle\ \texttt{cur} \mapsto \texttt{node}(1, \mathcal{P})\ \rangle$

$\left\{\begin{array}{l} \texttt{regionState} * \texttt{primeBlockProperty} * \texttt{untouchedList}(\texttt{level}) * \\ \texttt{addrLkvLProperty} * \texttt{cur} \mapsto \texttt{node}(1,\_,\_,\_,\_,\_) \wedge \texttt{found} = \texttt{false} \end{array}\right\}$
while (found = false) {
$\left\{\begin{array}{l} \texttt{regionState} * \texttt{primeBlockProperty} * \texttt{untouchedList}(\texttt{level}) * \\ \texttt{addrLkvLProperty} * \texttt{cur} \mapsto \texttt{node}(1,\_,\_,\_,\_,\_) \wedge \texttt{found} = \texttt{false} \end{array}\right\}$

Use
$\langle\ \texttt{cur} \mapsto \texttt{node}(1, \mathcal{P}) * \texttt{curn} = \_\ \rangle$
$\quad \texttt{curn} := \texttt{get}(\texttt{cur});$
$\langle\ \texttt{cur} \mapsto \texttt{node}(1, \mathcal{P}) * \texttt{curn} = \texttt{node}(1, \mathcal{P})\ \rangle$

$\left\{\begin{array}{l} \texttt{regionState} * \texttt{primeBlockProperty} * \texttt{untouchedList}(\texttt{level}) * \\ \texttt{addrLkvLProperty} * (\texttt{cur}, \texttt{curn}) \mapsto \texttt{node}(1,\_,\_,\_,\_,\_) \wedge \texttt{found} = \texttt{false} \end{array}\right\}$
if (isDelete(curn)||m < lowKey(curn)) {
$\left\{\begin{array}{l} \texttt{regionState} * \texttt{primeBlockProperty} * \texttt{untouchedList}(\texttt{level}) * \\ \texttt{addrLkvLProperty} * \exists delelte, kvmL. \\ (\texttt{cur}, \texttt{curn}) \mapsto \texttt{node}(1,\_, kvmL,\_,\_, delete) \wedge \\ (\texttt{m} < \texttt{first}(kvmL)\lfloor_1 \vee delete = \texttt{true}) \wedge \texttt{found} = \texttt{false} \end{array}\right\}$

Use
$\langle\ \texttt{cur} \mapsto \texttt{node}(1,\_,\_,\_,\_,\_)\ \rangle$
$\quad \texttt{unlock}(\texttt{cur});$
$\langle\ \texttt{cur} \mapsto \texttt{node}(\_,\_,\_,\_,\_,\_)\ \rangle$

// frame off cur and curn
$\left\{\begin{array}{l} \texttt{regionState} * \texttt{primeBlockProperty} * \texttt{untouchedList}(\texttt{level}) * \\ \texttt{addrLkvLProperty} \wedge \texttt{found} = \texttt{false} \end{array}\right\}$

Use $\forall pbL.$
$\langle$ x.pb $\mapsto$ primeBlock(pbL) $*$ pb $=$ _ $\rangle$
pb := getPrimeBlock(x.pb);
$\langle$ x.pb $\mapsto$ primeBlock(pbL) $*$ pb $=$ primeBlock(pbL) $\rangle$

{ regionState $*$ primeBlockProperty $*$ untouchedList(level)$*$
addrLkvLProperty $\wedge$ found $=$ false }

cur := getNodeLevel(curn, level);

// cur has been update

{ regionState $*$ primeBlockProperty $*$ untouchedList(level)$*$
addrLkvLProperty $*$ cur $\mapsto$ node(_, _, _, _, _, _) $\wedge$ found $=$ false }

Use $\forall l, \mathcal{P}.$
$\langle$ cur $\mapsto$ node($l, \mathcal{P}$) $\rangle$
lock(cur);
$\langle$ cur $\mapsto$ node(1, $\mathcal{P}$) $\rangle$

{ regionState $*$ primeBlockProperty $*$ untouchedList(level)$*$
addrLkvLProperty $*$ cur $\mapsto$ node(1, _, _, _, _, _) $\wedge$ found $=$ false }

Use $\langle$ cur $\mapsto$ node(1, $\mathcal{P}$) $*$ curn $=$ _ $\rangle$
curn := get(cur);
$\langle$ cur $\mapsto$ node(1, $\mathcal{P}$) $*$ curn $=$ node(1, $\mathcal{P}$) $\rangle$

{ regionState $*$ primeBlockProperty $*$ untouchedList(level)$*$
addrLkvLProperty $*$ (cur, curn) $\mapsto$ node(1, _, _, _, _, _) $\wedge$ found $=$ false }

} else if (m $\geqslant$ highKey(curn)) {

{ regionState $*$ primeBlockProperty $*$ untouchedList(level)$*$
addrLkvLProperty $*$ $\exists mxk.$ (cur, curn) $\mapsto$ node(1, _, _, mxk, _, _)$_{\texttt{level}}$ $\wedge$
m $\geqslant$ mxk $\wedge$ found $=$ false }

Use $\langle$ cur $\mapsto$ node(1, _, _, _, _, _)$_{\texttt{level}}$ $\rangle$
unlock(cur);
$\langle$ cur $\mapsto$ node(_, _, _, _, _, _)$_{\texttt{level}}$ $\rangle$

{ regionState $*$ primeBlockProperty $*$ untouchedList(level)$*$
addrLkvLProperty $*$ $\exists mxk, mxk'.$ cur $\mapsto$ node(_, _, _, mxk, _, _)$*$
curn $=$ node(1, _, _, mxk', _, _)$^{\blacktriangle}$ $\wedge$ m $>$ mxk' $\wedge$ found $=$ false }

cur := next(curn, m);

// frame off curn

{ regionState $*$ primeBlockProperty $*$ untouchedList(level)$*$
addrLkvLProperty$*$. cur $\mapsto$ node(_, _, _, _, _, _) $\wedge$ found $=$ false }

Use $\forall l, \mathcal{P}.$
$\langle$ cur $\mapsto$ node($l, \mathcal{P}$) $\rangle$
lock(cur);
$\langle$ cur $\mapsto$ node(1, $\mathcal{P}$) $\rangle$

{ regionState $*$ primeBlockProperty $*$ untouchedList(level)$*$
addrLkvLProperty$*$. cur $\mapsto$ node(1, _, _, _, _, _) $\wedge$ found $=$ false }

Use $\langle$ cur $\mapsto$ node(1, $\mathcal{P}$) $*$ curn $=$ _ $\rangle$
curn := get(cur);
$\langle$ cur $\mapsto$ node(1, $\mathcal{P}$) $*$ curn $=$ node(1, $\mathcal{P}$) $\rangle$

{ regionState $*$ primeBlockProperty $*$ untouchedList(level)$*$
addrLkvLProperty $*$ (cur, curn) $\mapsto$ node(1, _, _, _, _, _) $\wedge$ found $=$ false }

} else if (isIn(curn, m)) {

$\left\{ \begin{array}{l} \texttt{regionState} * \texttt{primeBlockProperty} * \texttt{untouchedList(level)} * \\ \texttt{addrLkvLProperty} * \exists kvmL.\, (\texttt{cur}, \texttt{curn}) \mapsto \texttt{node}(1, \_, kvmL, \_, \_, \_)_{\texttt{level}} \wedge \\ (\texttt{m}, \_, \_) \in kvmL \wedge \texttt{found} = \texttt{false} \end{array} \right\}$

tv := modifyPair(curn, m, w);

$\left\{ \begin{array}{l} \texttt{regionState} * \texttt{primeBlockProperty} * \texttt{untouchedList(level)} * \\ \texttt{addrLkvLProperty} * \exists kvmL.\, \texttt{cur} \mapsto \texttt{node}(1, \_, kvmL, \_, \_, \_)_{\texttt{level}} * \\ \texttt{curn} = \texttt{node}(1, \_, kvmL \backslash [(\texttt{m}, \_, \_)] \uplus [(\texttt{m}, \texttt{w}, \_)], \_, \_, \_)^{\blacktriangle}_{\texttt{level}} \wedge \\ (\texttt{m}, \_, \_) \in kvmL \wedge \texttt{found} = \texttt{false} \end{array} \right\}$

U&U

$\forall kvL_{\texttt{level}}.$

$\left\langle \begin{array}{l} \texttt{cur} \mapsto \texttt{node}(1, \_, kvmL, \_, \_, \_)_{\texttt{level}} * \\ \texttt{curn} = \texttt{node}(1, \_, kvmL \backslash [(\texttt{m}, \_, \_)] \uplus [(\texttt{m}, \texttt{w}, \_)], \_, \_, \_)^{\blacktriangle}_{\texttt{level}} \end{array} \right\rangle$

put(cur, curn);

$\left\langle \begin{array}{l} \texttt{cur} \mapsto \texttt{node}(1, \_, kvmL \backslash [(\texttt{m}, \_, \_)] \uplus [(\texttt{m}, \texttt{w}, \_)], \_, \_, \_)_{\texttt{level}} * \\ \texttt{curn} = \texttt{node}(1, \_, kvmL \backslash [(\texttt{m}, \_, \_)] \uplus [(\texttt{m}, \texttt{w}, \_)], \_, \_, \_)^{\blacktriangle}_{\texttt{level}} \end{array} \right\rangle$

$\left\{ \begin{array}{l} a \Mapsto (S, S[\texttt{k} \mapsto \texttt{v}]) * \texttt{primeBlockProperty} * \texttt{untouchedList(level)} * \\ \texttt{addrLkvLProperty} * \exists kvmL.\, (\texttt{cur} \mapsto \texttt{node}(1, \_, kvmL, \_, \_, \_)_{\texttt{level}} \wedge \\ (\texttt{m}, \texttt{w}, \_) \in kvmL \wedge ((\texttt{level} = 1 \wedge \texttt{m} = \texttt{k} \wedge \texttt{w} = \texttt{v}) \vee \texttt{level} \neq 1) \wedge \texttt{found} = \texttt{false} \end{array} \right\}$

Use

$\left\langle \texttt{cur} \mapsto \texttt{node}(1, \_, \_, \_, \_, \_)_{\texttt{level}} \right\rangle$

unlock(cur);

$\left\langle \texttt{cur} \mapsto \texttt{node}(\_, \_, \_, \_, \_, \_)_{\texttt{level}} \right\rangle$

// frame off curn

$\left\{ \begin{array}{l} a \Mapsto (S, S[\texttt{k} \mapsto \texttt{v}]) * \texttt{primeBlockProperty} * \texttt{untouchedList(level)} * \\ \texttt{addrLkvLProperty} \wedge ((\texttt{level} = 1 \wedge \texttt{m} = \texttt{k} \wedge \texttt{w} = \texttt{v}) \vee \texttt{level} \neq 1) \wedge \texttt{found} = \texttt{false} \end{array} \right\}$

// end of all rule

return v;

$\left\langle \texttt{map}(\texttt{x}, S[\texttt{k} \mapsto \texttt{v}]) \wedge \texttt{ret} = S(\texttt{k}) \right\rangle$

} else {

$\left\{ \begin{array}{l} \texttt{regionState} * \texttt{primeBlockProperty} * \texttt{untouchedList(level)} * \\ \texttt{addrLkvLProperty} * \exists kvmL, mxk.\, (\texttt{cur}, \texttt{curn}) \mapsto \texttt{node}(1, \_, kvmL, mxk, \_, \_)_{\texttt{level}} \wedge \\ \texttt{first}(kvmL)\!\downarrow_1 \leqslant \texttt{w} < mxk \wedge \texttt{found} = \texttt{false} \end{array} \right\}$

found := true;

$\left\{ \begin{array}{l} \texttt{regionState} * \texttt{primeBlockProperty} * \texttt{untouchedList(level)} * \\ \texttt{addrLkvLProperty} * \exists kvmL, mxk.\, (\texttt{cur}, \texttt{curn}) \mapsto \texttt{node}(1, \_, kvmL, mxk, \_, \_)_{\texttt{level}} \wedge \\ \texttt{first}(kvmL)\!\downarrow_1 \leqslant \texttt{w} < mxk \wedge \texttt{found} = \texttt{true} \end{array} \right\}$

// to get the most recent copy

Use

$\forall pbL.$

$\left\langle \texttt{x.pb} \mapsto \texttt{primeBlock}(pbL) \right\rangle$

pb := getPrimeBlock(x.pb);

$\left\langle \texttt{x.pb} \mapsto \texttt{primeBlock}(pbL) * \texttt{pb} = \texttt{primeBlock}(pbL) \right\rangle$

$\left\{ \begin{array}{l} \texttt{regionState} * \texttt{primeBlockProperty} * \texttt{untouchedList(level)} * \\ \texttt{addrLkvLProperty} * \exists kvmL, mxk.\, (\texttt{cur}, \texttt{curn}) \mapsto \texttt{node}(1, \_, kvmL, mxk, \_, \_)_{\texttt{level}} \wedge \\ \texttt{first}(kvmL)\!\downarrow_1 \leqslant \texttt{w} < mxk \wedge \texttt{found} = \texttt{true} \end{array} \right\}$

}

}

$\left\{ \begin{array}{l} \texttt{regionState} * \texttt{primeBlockProperty} * \texttt{untouchedList(level)} * \\ \texttt{addrLkvLProperty} * \exists kvmL, mxk.\, (\texttt{cur}, \texttt{curn}) \mapsto \texttt{node}(1, \_, kvmL, mxk, \_, \_)_{\texttt{level}} \wedge \\ \texttt{first}(kvmL)\!\downarrow_1 \leqslant \texttt{w} < mxk \end{array} \right\}$

A.2.3 Insert into safe node

$$\left\{\begin{array}{l}\texttt{regionState} * \exists kvmL, mxk.\, (\texttt{cur}, \texttt{curn}) \mapsto \texttt{node}(1, \_, kvmL, mxk, \_, \_)_{\texttt{level}} \wedge \\ \texttt{first}(kvmL)\downarrow_1 \leqslant \texttt{m} < mxk \wedge |kvmL| < \texttt{max}\end{array}\right\}$$

insertIntoSafe{

addPair(curn, m, w);

$$\left\{\begin{array}{l}\texttt{regionState} * \exists kvmL, mxk.\, \texttt{cur} \mapsto \texttt{node}(1, \_, kvmL, mxk, \_, \_)_{\texttt{level}} * \\ \texttt{curn} = \texttt{node}(1, \_, kvmL \uplus [(\texttt{m}, \texttt{w}, \texttt{false})], mxk, \_, \_)^{\blacktriangle}_{\texttt{level}} \wedge \\ \texttt{first}(kvmL)\downarrow_1 \leqslant \texttt{m} < mxk\end{array}\right\}$$

U&U

$\forall kvL_{\texttt{level}}.$

$$\left\langle\begin{array}{l}\texttt{cur} \mapsto \texttt{node}(1, \_, kvmL, \_, \_, \_)_{\texttt{level}} * \\ \texttt{curn} = \texttt{node}(1, \_, kvmL \uplus [(\texttt{m}, \texttt{w}, \texttt{false})], \_, \_, \_)^{\blacktriangle}_{\texttt{level}}\end{array}\right\rangle$$

put(curn, cur);

$$\left\langle\begin{array}{l}\texttt{cur} \mapsto \texttt{node}(1, \_, kvmL \uplus [(\texttt{m}, \texttt{w}, \texttt{false})], \_, \_, \_)_{\texttt{level}} * \\ \texttt{curn} = \texttt{node}(1, \_, kvmL \uplus [(\texttt{m}, \texttt{w}, \texttt{false})], \_, \_, \_)^{\blacktriangle}_{\texttt{level}}\end{array}\right\rangle$$

$$\left\{\begin{array}{l}a \Mapsto (S, S[\texttt{k} \mapsto \texttt{v}]) * \exists kvmL.\, (\texttt{cur}, \texttt{curn}) \mapsto \texttt{node}(1, \_, kvmL, \_, \_, \_)_{\texttt{level}} \wedge \\ kvmL \subseteq kvL_{\texttt{level}} \wedge ((\texttt{level} = 1 \wedge S = kvL_{\texttt{level}} \wedge \texttt{m} = \texttt{k} \wedge \texttt{w} = \texttt{v}) \vee \\ \texttt{level} > 1) \wedge (\texttt{m}, \texttt{w}, \_) \in kvmL\end{array}\right\}$$

Use

$$\left\langle \texttt{cur} \mapsto \texttt{node}(1, \_, \_, \_, \_, \_)_{\texttt{level}} \right\rangle$$

unlock(cur);

$$\left\langle \texttt{cur} \mapsto \texttt{node}(\_, \_, \_, \_, \_, \_)_{\texttt{level}} \right\rangle$$

// frame off curn

$$\left\{\begin{array}{l}a \Mapsto (S, S[\texttt{k} \mapsto \texttt{v}]) * \texttt{cur} \mapsto \texttt{node}(\_, \_, \_, \_, \_, \_)_{\texttt{level}} \wedge \\ ((\texttt{level} = 1 \wedge S = kvL_{\texttt{level}} \wedge \texttt{m} = \texttt{k} \wedge \texttt{w} = \texttt{v}) \vee (\texttt{level} > 1))\end{array}\right\}$$

}

$$\left\{\begin{array}{l}a \Mapsto (S, S[\texttt{k} \mapsto \texttt{v}]) * \texttt{cur} \mapsto \texttt{node}(\_, \_, \_, \_, \_, \_)_{\texttt{level}} \wedge \\ ((\texttt{level} = 1 \wedge S = kvL_{\texttt{level}} \wedge \texttt{m} = \texttt{k} \wedge \texttt{w} = \texttt{v}) \vee (\texttt{level} > 1))\end{array}\right\}$$

A.2.4 Insert into unsafe root

// quantify $n = |pbL|$ which equal to
// lenth of the prime block $|pbL|$
// because the root node is locked, there is no interference for prime block

$$\left\{\begin{array}{l}\texttt{regionState} * \texttt{primeBlockProperty} * \exists kvmL, mxk. \\ (\texttt{cur}, \texttt{curn}) \mapsto \texttt{node}(1, \_, kvmL, mxk, \_, \_)_{\texttt{level}} \wedge \\ \texttt{level} = n \wedge \texttt{first}(kvmL)\downarrow_1 \leqslant \texttt{m} < mxk \wedge \texttt{last}(pbL) = \texttt{cur}\end{array}\right\}$$

insertIntoUnsafeRoot{

new := new();

$$\left\{\begin{array}{l}\texttt{regionState} * \texttt{primeBlockProperty} * \exists kvmL, mxk. \\ (\texttt{cur}, \texttt{curn}) \mapsto \texttt{node}(1, \_, kvmL, mxk, \_, \_)_{\texttt{level}} * \\ \texttt{new} \mapsto \texttt{node}(0, 0, [], +\infty, \texttt{null}, \texttt{false})^{\blacktriangle} \wedge \\ \texttt{level} = n \wedge \texttt{first}(kvmL)\downarrow_1 \leqslant \texttt{m} < mxk \wedge \texttt{last}(pbL) = \texttt{cur}\end{array}\right\}$$

newn := split(curn, m, w, new);

// new node *type* is the same as old, and not dead node

$\left\{\begin{array}{l}\texttt{regionState} * \texttt{primeBlockProperty} * \exists kvmL, kvmL', kvmL'', mxk, nxt.\\ \texttt{cur} \mapsto \texttt{node}(1, \_, kvmL, mxk, nxt, \_)_{\texttt{level}} *\\ \texttt{curn} = \texttt{node}(1, \_, kvmL', \texttt{first}(kvmL'')\downarrow_1, \texttt{new}, \_)^{\blacktriangle}_{\texttt{level}} *\\ \texttt{new} \mapsto \texttt{node}(0, 0, [], +\infty, \texttt{null}, \_)^{\blacktriangle} *\\ \texttt{newn} = \texttt{node}(0, \_, kvmL'', mxk, nxt, \_)^{\blacktriangle}_{\texttt{level}} \wedge\\ \texttt{level} = n \wedge \texttt{last}(pbL) = \texttt{cur} \wedge \texttt{first}(kvmL)\downarrow_1 \leqslant \texttt{m} < mxk \wedge\\ kvmL' +\!\!+ kvmL'' = kvmL \uplus [(\texttt{m}, \texttt{w}, \texttt{false})]\end{array}\right\}$

Use

$\left\langle\begin{array}{l}\texttt{new} \mapsto \texttt{node}(0, 0, [], \texttt{null}, \texttt{false})^{\blacktriangle} *\\ \texttt{newn} = \texttt{node}(0, type, kvmL'', mxk, nxt, \texttt{false})^{\blacktriangle}_{\texttt{level}}\end{array}\right\rangle$

$\texttt{put}(\texttt{newn}, \texttt{new});$

$\left\langle\begin{array}{l}\texttt{new} \mapsto \texttt{node}(0, type, kvmL'', mxk, nxt, \texttt{false})^{\blacktriangle} *\\ \texttt{newn} = \texttt{node}(0, type, kvmL'', mxk, nxt, \texttt{false})^{\blacktriangle}_{\texttt{level}}\end{array}\right\rangle$

$\left\{\begin{array}{l}\texttt{regionState} * \texttt{primeBlockProperty} * \exists kvmL, kvmL', kvmL'', mxk, nxt.\\ \texttt{cur} \mapsto \texttt{node}(1, \_, kvmL, mxk, nxt, \_)_{\texttt{level}} *\\ \texttt{curn} = \texttt{node}(1, \_, kvmL', \texttt{first}(kvmL'')\downarrow_1, \texttt{new}, \_)^{\blacktriangle}_{\texttt{level}} *\\ \texttt{new} \mapsto \texttt{node}(0, \_, kvmL'', mxk, nxt, \_)^{\blacktriangle} *\\ \texttt{newn} = \texttt{node}(0, \_, kvmL'', mxk, nxt, \_)^{\blacktriangle}_{\texttt{level}} \wedge\\ \texttt{level} = n \wedge \texttt{last}(pbL) = \texttt{cur} \wedge \texttt{first}(kvmL)\downarrow_1 \leqslant \texttt{m} < mxk \wedge\\ kvmL' +\!\!+ kvmL'' = kvmL \uplus [(\texttt{m}, \texttt{w}, \texttt{false})]\end{array}\right\}$

Use

$\langle\ \texttt{new} \mapsto \texttt{node}(0, \_, \_, \_, \_)^{\blacktriangle}\ \rangle$

$\texttt{lock}(\texttt{new});$

$\langle\ \texttt{new} \mapsto \texttt{node}(1, \_, \_, \_, \_)^{\blacktriangle}\ \rangle$

$\left\{\begin{array}{l}\texttt{regionState} * \texttt{primeBlockProperty} * \exists kvmL, kvmL', kvmL'', mxk, nxt.\\ \texttt{cur} \mapsto \texttt{node}(1, \_, kvmL, mxk, nxt, \_)_{\texttt{level}} *\\ \texttt{curn} = \texttt{node}(1, \_, kvmL', \texttt{first}(kvmL'')\downarrow_1, \texttt{new}, \_)^{\blacktriangle}_{\texttt{level}} *\\ \texttt{new} \mapsto \texttt{node}(1, \_, kvmL'', mxk, nxt, \_)^{\blacktriangle} *\\ \texttt{newn} = \texttt{node}(0, \_, kvmL'', mxk, nxt, \_)^{\blacktriangle}_{\texttt{level}} \wedge\\ \texttt{level} = n \wedge \texttt{last}(pbL) = \texttt{cur} \wedge \texttt{first}(kvmL)\downarrow_1 \leqslant \texttt{m} < mxk \wedge\\ kvmL' +\!\!+ kvmL'' = kvmL \uplus [(\texttt{m}, \texttt{w}, \texttt{false})]\end{array}\right\}$

U&U

$\left\langle\begin{array}{l}\texttt{cur} \mapsto \texttt{node}(1, \_, kvmL, \_, \_)_{\texttt{level}} *\\ \texttt{curn} = \texttt{node}(1, \_, kvmL', \texttt{first}(kvmL'')\downarrow_1, \texttt{new}, \_)^{\blacktriangle}_{\texttt{level}}\end{array}\right\rangle$

$\texttt{put}(\texttt{curn}, \texttt{cur});$

$\left\langle\begin{array}{l}\texttt{cur} \mapsto \texttt{node}(1, \_, kvmL', \texttt{first}(kvmL'')\downarrow_1, \texttt{new}, \_)_{\texttt{level}} *\\ \texttt{curn} = \texttt{node}(1, \_, kvmL', \texttt{first}(kvmL'')\downarrow_1, \texttt{new}, \_)^{\blacktriangle}_{\texttt{level}}\end{array}\right\rangle$

// also make new become shared resource

$\left\{\begin{array}{l}\texttt{regionState} * \texttt{primeBlockProperty} * \exists kvmL, kvmL', kvmL'', mxk, nxt.\\ (\texttt{cur}, \texttt{curn}) \mapsto \texttt{node}(1, \_, kvmL', \texttt{first}(kvmL'')\downarrow_1, \texttt{new}, \_)_{\texttt{level}} *\\ \texttt{new} \mapsto \texttt{node}(1, \_, kvmL'', mxk, nxt, \_)^{\blacktriangle} *\\ \texttt{newn} = \texttt{node}(0, \_, kvmL'', mxk, nxt, \_)^{\blacktriangle}_{\texttt{level}} \wedge\\ \texttt{level} = n \wedge \texttt{last}(pbL) = \texttt{cur} \wedge \texttt{first}(kvmL)\downarrow_1 \leqslant \texttt{m} < mxk \wedge\\ kvmL' +\!\!+ kvmL'' = kvmL \uplus [(\texttt{m}, \texttt{w}, \texttt{false})]\end{array}\right\}$

$\texttt{y} := \texttt{lowKey}(\texttt{curn});$

$\texttt{t} := \texttt{highKey}(\texttt{curn});$

$\texttt{u} := \texttt{highKey}(\texttt{newn});$

// frame off curn and newn

$$\left\{\begin{array}{l} a \Mapsto (S, S[\mathtt{k} \mapsto \mathtt{v}]) * \mathtt{primeBlockProperty} * \exists kvmL, kvmL', mxk, mxk'. \\ \mathtt{cur} \mapsto \mathtt{node}(1, \_, kvmL, mxk, \mathtt{new}, \_)_{\mathtt{level}} * \\ \mathtt{new} \mapsto \mathtt{node}(1, \_, kvmL', mxk', \_, \_)_{\mathtt{level}} \wedge \\ \mathtt{level} = n \wedge \mathtt{last}(pbL) = \mathtt{cur} \wedge (\mathtt{m}, \mathtt{w}, \_) \in (kvmL +\!\!+ kvmL') \wedge \\ ((\mathtt{level} = 1 \wedge \mathtt{m} = \mathtt{k} \wedge \mathtt{w} = \mathtt{v} \wedge S^{\bullet} = kvL^{\bullet}_{\mathtt{level}}) \vee \mathtt{level} > 1) \wedge \\ \mathtt{y} = \mathtt{first}(kvmL)\!\downarrow_1 \wedge \mathtt{t} = mxk \wedge \mathtt{u} = mxk' \end{array}\right\}$$

$\mathtt{r} := \mathtt{new}();$

$\mathtt{rn} := \mathtt{newRoot}(\mathtt{y}, \mathtt{cur}, \mathtt{t}, \mathtt{new}, \mathtt{u});$

$$\left\{\begin{array}{l} a \Mapsto (S, S[\mathtt{k} \mapsto \mathtt{v}]) * \mathtt{primeBlockProperty} * \exists kvmL, kvmL', kvmL'', mxk, mxk'. \\ \mathtt{cur} \mapsto \mathtt{node}(1, \_, kvmL, mxk, \mathtt{new}, \_)_{\mathtt{level}} * \mathtt{new} \mapsto \mathtt{node}(1, \_, kvmL', mxk' \_, \_)_{\mathtt{level}} * \\ \mathtt{r} \mapsto \mathtt{node}(0, 0, [], +\infty, \mathtt{null}, \mathtt{false})^{\blacktriangle} * \mathtt{rn} = \mathtt{node}(0, 0, kvmL'', mxk', \mathtt{null}, \mathtt{false})^{\blacktriangle} \wedge \\ \mathtt{level} = n \wedge \mathtt{last}(pbL) = \mathtt{cur} \wedge (\mathtt{m}, \mathtt{w}, \_) \in kvmL +\!\!+ kvmL' \wedge \\ ((\mathtt{level} = 1 \wedge \mathtt{m} = \mathtt{k} \wedge \mathtt{w} = \mathtt{v} \wedge S^{\bullet} = kvL^{\bullet}_{\mathtt{level}}) \vee \mathtt{level} > 1) \wedge \\ kvmL'' = [(\mathtt{first}(kvmL)\!\downarrow_1, \mathtt{cur}, \mathtt{false}), (mxk, \mathtt{new}, \mathtt{false})] \end{array}\right\}$$

Use

$$\left\langle\begin{array}{l} \mathtt{r} \mapsto \mathtt{node}(0, 0, [], \mathtt{null}, \mathtt{false})^{\blacktriangle} * \\ \mathtt{rn} = \mathtt{node}(0, 0, kvmL, mxk', \mathtt{null}, \mathtt{false})^{\blacktriangle} \wedge \\ kvmL = [(\mathtt{first}(kvmL)\!\downarrow_1, \mathtt{cur}, \mathtt{false}), (mxk, \mathtt{new}, \mathtt{false})] \end{array}\right\}$$

$\mathtt{put}(\mathtt{rn}, \mathtt{r});$

$$\left\langle\begin{array}{l} \mathtt{r} \mapsto \mathtt{node}(0, 0, kvmL, mxk', \mathtt{null}, \mathtt{false})^{\blacktriangle} * \\ \mathtt{rn} = \mathtt{node}(0, 0, kvmL, mxk', \mathtt{null}, \mathtt{false})^{\blacktriangle} \wedge \\ kvmL = [(\mathtt{first}(kvmL)\!\downarrow_1, \mathtt{cur}, \mathtt{false}), (mxk, \mathtt{new}, \mathtt{false})] \end{array}\right\}$$

// frame off `rn`. notice, currently `r` is local resource

$$\left\{\begin{array}{l} a \Mapsto (S, S[\mathtt{k} \mapsto \mathtt{v}]) * \mathtt{primeBlockProperty} * \exists kvmL, kvmL', kvmL'', mxk, mxk'. \\ \mathtt{cur} \mapsto \mathtt{node}(1, \_, kvmL, mxk, \mathtt{new}, \_)_{\mathtt{level}} * \mathtt{new} \mapsto \mathtt{node}(1, \_, kvmL', mxk' \_, \_)_{\mathtt{level}} * \\ \mathtt{r} \mapsto \mathtt{node}(0, 0, kvmL'', mxk', \mathtt{null}, \mathtt{false})^{\blacktriangle} \wedge \mathtt{level} = n \wedge \mathtt{last}(pbL) = \mathtt{cur} \wedge \\ (\mathtt{m}, \mathtt{w}, \_) \in kvmL +\!\!+ kvmL' \wedge ((\mathtt{level} = 1 \wedge \mathtt{m} = \mathtt{k} \wedge \mathtt{w} = \mathtt{v} \wedge S^{\bullet} = kvL^{\bullet}_{\mathtt{level}}) \vee \\ \mathtt{level} > 1) \wedge kvmL'' = [(\mathtt{first}(kvmL)\!\downarrow_1, \mathtt{cur}, \mathtt{false}), (mxk, \mathtt{new}, \mathtt{false})] \end{array}\right\}$$

$\mathtt{addRoot}(\mathtt{pb}, \mathtt{r});$

// $\mathtt{level} = n - 1$ instead of $\mathtt{level} = n$,
// since $n = |pbL +\!\!+ [\mathtt{r}]|$ now

$$\left\{\begin{array}{l} a \Mapsto (S, S[\mathtt{k} \mapsto \mathtt{v}]) * \mathtt{primeBlockProperty} * \exists kvmL, kvmL', kvmL'', mxk, mxk'. \\ \mathtt{cur} \mapsto \mathtt{node}(1, \_, kvmL, mxk, \mathtt{new}, \_)_{\mathtt{level}} * \mathtt{new} \mapsto \mathtt{node}(1, \_, kvmL', mxk' \_, \_)_{\mathtt{level}} * \\ \mathtt{r} \mapsto \mathtt{node}(0, 0, kvmL'', mxk', \mathtt{null}, \mathtt{false})^{\blacktriangle} \wedge \mathtt{level} = n - 1 \wedge \mathtt{last}(pbL) = \mathtt{cur} \wedge \\ (\mathtt{m}, \mathtt{w}, \_) \in kvmL +\!\!+ kvmL' \wedge ((\mathtt{level} = 1 \wedge \mathtt{m} = \mathtt{k} \wedge \mathtt{w} = \mathtt{v} \wedge S^{\bullet} = kvL^{\bullet}_{\mathtt{level}}) \vee \\ \mathtt{level} > 1) \wedge kvmL'' = [(\mathtt{first}(kvmL)\!\downarrow_1, \mathtt{cur}, \mathtt{false}), (mxk, \mathtt{new}, \mathtt{false})] \end{array}\right\}$$

Use

$$\left\langle\begin{array}{l} \mathtt{pb} = \mathtt{primeBlock}(pbL +\!\!+ [\mathtt{r}])^{\blacktriangle} * \\ \mathtt{x.pb} \mapsto \mathtt{primeBlock}(pbL) \wedge \mathtt{cur} = \mathtt{last}(pbL) \end{array}\right\}$$

$\mathtt{putPrimeBlock}(\mathtt{x.pb}, \mathtt{pb});$

$$\left\langle\begin{array}{l} \mathtt{pb} = \mathtt{primeBlock}(pbL +\!\!+ [\mathtt{r}])^{\blacktriangle} * \\ \mathtt{x.pb} \mapsto \mathtt{primeBlock}(pbL +\!\!+ [\mathtt{r}]) \wedge \mathtt{cur} = \mathtt{last}(pbL) \end{array}\right\}$$

// notice, after updating local snapshot $|pbL'|$ grow 1,
// and $|pbL|$ grow at least 1,
// because new root `r` become shared resource,
// and could be locked by other threads,
// thus, it become `primeBlockProperty`,

$$\left\{\begin{array}{l} a \Mapsto (S, S[\mathtt{k} \mapsto \mathtt{v}]) * \mathtt{primeBlockProperty} * \exists kvmL, kvmL'. \\ \mathtt{cur} \mapsto \mathtt{node}(1, \_, kvmL, \_, \mathtt{new}, \_)_{\mathtt{level}} * \mathtt{new} \mapsto \mathtt{node}(1, \_, kvmL', \_, \_, \_)_{\mathtt{level}} * \\ \mathtt{level} = n - 1 \wedge (\mathtt{w}, \mathtt{m}, \mathtt{false}) \in kvmL +\!\!+ kvmL' \wedge \\ ((\mathtt{level} = 1 \wedge \mathtt{m} = \mathtt{k} \wedge \mathtt{w} = \mathtt{v} \wedge S^{\bullet} = kvL^{\bullet}_{\mathtt{level}}) \vee \mathtt{level} > 1) \end{array}\right\}$$

Use $\left|\begin{array}{l}\langle\ \mathtt{cur} \mapsto \mathtt{node}(1,\_,\_,\_,\_,\_)_{\mathtt{level}}\ \rangle \\ \quad \mathtt{unlock}(\mathtt{cur}); \\ \langle\ \mathtt{cur} \mapsto \mathtt{node}(\_,\_,\_,\_,\_,\_)_{\mathtt{level}}\ \rangle\end{array}\right.$

$\left\{\begin{array}{l} a \Rrightarrow (S, S[\mathtt{k} \mapsto \mathtt{v}]) * \mathtt{primeBlockProperty} * \mathtt{new} \mapsto \mathtt{node}(1,\_,\_,\_,\_,\_)_{\mathtt{level}} \wedge \\ \mathtt{level} = n-1 \wedge ((\mathtt{level} = 1 \wedge \mathtt{m} = \mathtt{k} \wedge \mathtt{w} = \mathtt{v} \wedge S^{\bullet} = kvL^{\bullet}_{\mathtt{level}}) \vee \mathtt{level} > 1) \end{array}\right\}$

Use $\left|\begin{array}{l}\langle\ \mathtt{new} \mapsto \mathtt{node}(1,\_,\_,\_,\_,\_)_{\mathtt{level}}\ \rangle \\ \quad \mathtt{unlock}(\mathtt{new}); \\ \langle\ \mathtt{cur} \mapsto \mathtt{node}(\_,\_,\_,\_,\_,\_)_{\mathtt{level}}\ \rangle\end{array}\right.$

$\left\{\begin{array}{l} a \Rrightarrow (S, S[\mathtt{k} \mapsto \mathtt{v}]) * \mathtt{primeBlockProperty} \wedge \\ ((\mathtt{level} = 1 \wedge \mathtt{m} = \mathtt{k} \wedge \mathtt{w} = \mathtt{v} \wedge S^{\bullet} = kvL^{\bullet}_{\mathtt{level}}) \vee \mathtt{level} > 1) \end{array}\right\}$

}

$\left\{\begin{array}{l} a \Rrightarrow (S, S[\mathtt{k} \mapsto \mathtt{v}]) * \mathtt{primeBlockProperty} \wedge \\ ((\mathtt{level} = 1 \wedge \mathtt{m} = \mathtt{k} \wedge \mathtt{w} = \mathtt{v} \wedge S^{\bullet} = kvL^{\bullet}_{\mathtt{level}}) \vee \mathtt{level} > 1) \end{array}\right\}$

A.2.5 Insert into unsafe node

$\left\{\begin{array}{l} \mathtt{regionState} * \exists kvmL, mxk.\, (\mathtt{cur}, \mathtt{curn}) \mapsto \mathtt{node}(1,\_,kvmL,mxk,\_,\_)_{\mathtt{level}} * \\ \mathtt{stackProperty}(\mathtt{level}) \wedge \mathtt{first}(kvmL) \leqslant mxk \end{array}\right\}$

// frame off stackProperty and primeBlockProperty

insertIntoUnsafe{

  new := new();

  newn := split(curn, m, w, new);

$\left\{\begin{array}{l} \mathtt{regionState} * \exists kvmL, kvmL', kvmL'', mxk, mxk', nxt. \\ \mathtt{cur} \mapsto \mathtt{node}(1,\_,kvmL,mxk,nxt,\_)_{\mathtt{level}} * \\ \mathtt{curn} = \mathtt{node}(1,\_,kvmL',\mathtt{first}(kvmL'')\!\downarrow_1,\mathtt{new},\_)^{\blacktriangle}_{\mathtt{level}} * \\ \mathtt{new} \mapsto \mathtt{node}(0,\_,[],+\infty,\mathtt{null},\_)^{\blacktriangle} * \\ \mathtt{newn} = \mathtt{node}(0,\_,kvmL'',mxk,nxt,\_)^{\blacktriangle}_{\mathtt{level}} \wedge \\ \mathtt{first}(kvmL)\!\downarrow_1 \leqslant \mathtt{m} < mxk \wedge kvmL' +\!\!+ kvmL'' = kvmL \uplus [(\mathtt{m},\mathtt{w},\_)] \end{array}\right\}$

Use $\left|\begin{array}{l}\left\langle\begin{array}{l}\mathtt{new} \mapsto \mathtt{node}(0,0,[],\mathtt{null},\mathtt{false})^{\blacktriangle} * \\ \mathtt{newn} = \mathtt{node}(0,type,kvmL'',mxk,nxt,\mathtt{false})^{\blacktriangle}_{\mathtt{level}}\end{array}\right\rangle \\ \quad \mathtt{put}(\mathtt{newn},\mathtt{new}); \\ \left\langle\begin{array}{l}\mathtt{new} \mapsto \mathtt{node}(0,type,kvmL'',mxk,nxt,\mathtt{false})^{\blacktriangle} * \\ \mathtt{newn} = \mathtt{node}(0,type,kvmL'',mxk,nxt,\mathtt{false})^{\blacktriangle}_{\mathtt{level}}\end{array}\right\rangle\end{array}\right.$

$\left\{\begin{array}{l} \mathtt{regionState} * \exists kvmL, kvmL', kvmL'', mxk, mxk', nxt. \\ \mathtt{cur} \mapsto \mathtt{node}(1,\_,kvmL,mxk,nxt,\_)_{\mathtt{level}} * \\ \mathtt{curn} = \mathtt{node}(1,\_,kvmL',\mathtt{first}(kvmL'')\!\downarrow_1,\mathtt{new},\_)^{\blacktriangle}_{\mathtt{level}} * \\ (\mathtt{new},\mathtt{newn}) \mapsto \mathtt{node}(0,\_,kvmL'',mxk,nxt,\_)^{\blacktriangle}_{\mathtt{level}} \wedge \\ \mathtt{first}(kvmL)\!\downarrow_1 \leqslant \mathtt{m} < mxk \wedge kvmL' +\!\!+ kvmL'' = kvmL \uplus [(\mathtt{m},\mathtt{w},\_)] \end{array}\right\}$

U&U $\left|\begin{array}{l}\left\langle\begin{array}{l}\mathtt{cur} \mapsto \mathtt{node}(1,\_,kvmL,\_,\_)_{\mathtt{level}} * \\ \mathtt{curn} = \mathtt{node}(1,\_,kvmL',\mathtt{first}(kvmL'')\!\downarrow_1,\mathtt{new},\_)^{\blacktriangle}_{\mathtt{level}}\end{array}\right\rangle \\ \quad \mathtt{put}(\mathtt{curn},\mathtt{cur}); \\ \left\langle\begin{array}{l}\mathtt{cur} \mapsto \mathtt{node}(1,\_,kvmL',\mathtt{first}(kvmL'')\!\downarrow_1,\mathtt{new},\_)_{\mathtt{level}} * \\ \mathtt{curn} = \mathtt{node}(1,\_,kvmL',\mathtt{first}(kvmL'')\!\downarrow_1,\mathtt{new},\_)^{\blacktriangle}_{\mathtt{level}}\end{array}\right\rangle\end{array}\right.$

// new cannot be compress until adding into their parent node,

// new the value can change now, but it still in list.

// frame off new and newn

$\{\ a \Rrightarrow (S, S[\mathtt{k} \mapsto \mathtt{v}]) * (\mathtt{cur}, \mathtt{curn}) \mapsto \mathtt{node}(1,\_,\_,\_,\mathtt{new},\_)_{\mathtt{level}} \wedge (\_,\mathtt{new},\_) \notin kvL_{\mathtt{level}+1}\ \}$

  w := new;

$\mathtt{m} := \mathtt{highKey}(\mathtt{curn});$
$\{\ a \Rightarrow (S, S[\mathtt{k} \mapsto \mathtt{v}]) * (\mathtt{cur}, \mathtt{curn}) \mapsto \mathtt{node}(1, \_, \_, \mathtt{m}, \mathtt{w}, \_)_{\mathtt{level}} \wedge (\mathtt{m}, \mathtt{w}, \_) \notin kvL_{\mathtt{level}+1}\ \}$

Use
$\langle\ \mathtt{cur} \mapsto \mathtt{node}(1, \_, \_, \_, \_, \_)_{\mathtt{level}}\ \rangle$
$\mathtt{unlock}(\mathtt{cur});$
$\langle\ \mathtt{cur} \mapsto \mathtt{node}(\_, \_, \_, \_, \_, \_)_{\mathtt{level}}\ \rangle$

// frame off `list` currently at `level`
$\{\ a \Rightarrow (S, S[\mathtt{k} \mapsto \mathtt{v}]) \wedge (\mathtt{m}, \mathtt{w}, \_) \notin kvL_{\mathtt{level}+1}\ \}$
$\mathtt{level} := \mathtt{level} + 1;$
// frame back `list` at `level` + 1
// frame back `stackProperty` and `primeBlockProperty`
$\{\ a \Rightarrow (S, S[\mathtt{k} \mapsto \mathtt{v}]) * \mathtt{primeBlockProperty} * \mathtt{stackProperty}(\mathtt{level} - 1) *$
$\mathtt{list}(addrL_{\mathtt{level}}, kvL_{\mathtt{level}}, type_{\mathtt{level}}) \wedge \mathtt{level} > 1 \wedge (\mathtt{m}, \mathtt{w}, \_) \notin kvL_{\mathtt{level}}\ \}$
$\mathtt{if}\ (\mathtt{isEmpty}(\mathtt{stack}))\ \{$
$\{\ a \Rightarrow (S, S[\mathtt{k} \mapsto \mathtt{v}]) * \mathtt{primeBlockProperty} * \mathtt{stack}(\mathtt{stack}, []) *$
$\mathtt{list}(addrL_{\mathtt{level}}, kvL_{\mathtt{level}}, type_{\mathtt{level}}) \wedge \mathtt{level} > 1 \wedge (\mathtt{m}, \mathtt{w}, \_) \notin kvL_{\mathtt{level}}\ \}$

Use
$\forall pbL.$
$\langle\ \mathtt{x.pb} \mapsto \mathtt{primeBlock}(pbL)\ \rangle$
$\mathtt{pb} := \mathtt{getPrimeBlock}(\mathtt{x.pb});$
$\langle\ \mathtt{x.pb} \mapsto \mathtt{primeBlock}(pbL) * \mathtt{pb} = \mathtt{primeBlock}(pbL)\ \rangle$

$\{\ a \Rightarrow (S, S[\mathtt{k} \mapsto \mathtt{v}]) * \mathtt{primeBlockProperty} * \mathtt{stack}(\mathtt{stack}, []) *$
$\mathtt{list}(addrL_{\mathtt{level}}, kvL_{\mathtt{level}}, type_{\mathtt{level}}) \wedge \mathtt{level} > 1 \wedge (\mathtt{m}, \mathtt{w}, \_) \notin kvL_{\mathtt{level}}\ \}$
$\mathtt{cur} := \mathtt{getNodeLevel}(\mathtt{pb}, \mathtt{level});$
$\{\ a \Rightarrow (S, S[\mathtt{k} \mapsto \mathtt{v}]) * \mathtt{primeBlockProperty} * \mathtt{stack}(\mathtt{stack}, []) *$
$\mathtt{cur} \mapsto \mathtt{node}(\_, \_, \_, \_, \_, \_) \wedge \mathtt{level} > 1 \wedge (\mathtt{m}, \mathtt{w}, \_) \notin kvL_{\mathtt{level}}\ \}$
// weaken `stack(stack, [])` to `stackProperty(level)`
$\}\ \mathtt{else}\ \{$
$\mathtt{cur} := \mathtt{pop}(\mathtt{stack});$
$\{\ a \Rightarrow (S, S[\mathtt{k} \mapsto \mathtt{v}]) * \mathtt{primeBlockProperty} * \mathtt{stackProperty}(\mathtt{level}) *$
$\mathtt{cur} \mapsto \mathtt{node}(\_, \_, \_, \_, \_, \_) \wedge \mathtt{level} > 1 \wedge (\mathtt{m}, \mathtt{w}, \_) \notin kvL_{\mathtt{level}}\ \}$
$\}$
$\}$
// weak $a \Rightarrow (S, S[\mathtt{k} \mapsto \mathtt{v}]) \wedge \mathtt{level} > 1$ to `regionState`
$\{\ \mathtt{regionState} * \mathtt{primeBlockProperty} * \mathtt{stackProperty}(\mathtt{level}) *$
$\mathtt{cur} \mapsto \mathtt{node}(\_, \_, \_, \_, \_, \_) \wedge \mathtt{level} > 1 \wedge (\mathtt{m}, \mathtt{w}, \_) \notin kvL_{\mathtt{level}}\ \}$

A.3 Proof of Concurrent B^{link}tree Compression

$\{\ \mathtt{listProperty}(\mathtt{level}) * \exists kvmL.$
$(\mathtt{cur}, \mathtt{curn}) \mapsto \mathtt{node}(1, \_, kvmL, \_, \_, \_)_{\mathtt{level}+1} \wedge (k, \_, \neg \mathtt{b}) \in kvmL\ \}$
$\mathtt{compressNode}\{$
$\mathtt{one} := \mathtt{firstState}(\mathtt{curn}, !\mathtt{b});$
$\mathtt{if}\ (\mathtt{one} \neq \mathtt{null})\ \{$
$\{\ \mathtt{listProperty}(\mathtt{level}) * (\mathtt{cur}, \mathtt{curn}) \mapsto \mathtt{node}(1, \_, \_, \_, \_, \_)_{\mathtt{level}+1} *$
$\mathtt{one} \mapsto \mathtt{node}(\_, \_, \_, \_, \_, \_)_{\mathtt{level}}\ \}$

Use | $\forall l, \mathcal{P}.$
$\langle\ \texttt{one} \mapsto \texttt{node}(l, \mathcal{P})_{\texttt{level}}\ \rangle$
`lock(one);`
$\langle\ \texttt{one} \mapsto \texttt{node}(1, \mathcal{P})_{\texttt{level}}\ \rangle$

$\{\ \texttt{listProperty(level)} * (\texttt{cur}, \texttt{curn}) \mapsto \texttt{node}(1, \_, \_, \_, \_, \_)_{\texttt{level}+1} * \ \}$
$\{\ \texttt{one} \mapsto \texttt{node}(1, \_, \_, \_, \_, \_)_{\texttt{level}}\ \}$

Use | $\langle\ \texttt{one} \mapsto \texttt{node}(1, \mathcal{P})_{\texttt{level}} * \texttt{onen} = \_\ \rangle$
`onen := get(one);`
$\langle\ \texttt{one} \mapsto \texttt{node}(1, \mathcal{P})_{\texttt{level}} * \texttt{onen} = \texttt{node}(1, \mathcal{P})_{\texttt{level}}\ \rangle$

$\{\ \texttt{listProperty(level)} * (\texttt{cur}, \texttt{curn}) \mapsto \texttt{node}(1, \_, \_, \_, \_, \_)_{\texttt{level}+1} * \ \}$
$\{\ (\texttt{one}, \texttt{onen}) \mapsto \texttt{node}(1, \_, \_, \_, \_, \_)_{\texttt{level}}\ \}$

`two := nextNode(one);`

$\{\ \texttt{listProperty(level)} * (\texttt{cur}, \texttt{curn}) \mapsto \texttt{node}(1, \_, \_, \_, \_, \_)_{\texttt{level}+1} * \ \}$
$\{\ (\texttt{one}, \texttt{onen}) \mapsto \texttt{node}(1, \_, \_, \_, \_, \_)_{\texttt{level}} * (\texttt{two} \mapsto \texttt{node}(\_, \_, \_, \_, \_, \_)_{\texttt{level}} \vee \texttt{two} = \texttt{null})\ \}$

```
if (one = last(curn)) {
  changePointerState(curn, one, b);
```

// frame off onen and two

$\{\ \texttt{listProperty(level)} * \exists kvmL.\ \texttt{cur} \mapsto \texttt{node}(1, \_, kvmL, \_, \_, \_)_{\texttt{level}+1} *$
$\texttt{curn} = \texttt{node}(1, \_, kvmL \backslash [(k, \texttt{one}, \neg\texttt{b})] \uplus [(k, \texttt{one}, \texttt{b})], \_, \_, \_)_{\texttt{level}+1} *$
$\texttt{one} \mapsto \texttt{node}(1, \_, \_, \_, \texttt{two}, \_)_{\texttt{level}} \wedge (\_, \texttt{one}, \neg\texttt{b}) = \texttt{last}(kvmL)\ \}$

Use | $\langle\ \texttt{curn} \mapsto \texttt{node}(1, \_, kvmL, \_, \_, \_)_{\texttt{level}+1} *$
$\texttt{curn} = \texttt{node}(1, \_, kvmL \backslash [(k, \texttt{one}, \neg\texttt{b})] \uplus [(k, \texttt{one}, \texttt{b})], \_, \_, \_)_{\texttt{level}+1}\ \rangle$
`put(cur, curn);`
$\langle\ \texttt{curn} \mapsto \texttt{node}(1, \_, kvmL \backslash [(k, \texttt{one}, \neg\texttt{b})] \uplus [(k, \texttt{one}, \texttt{b})], \_, \_, \_)_{\texttt{level}+1} *$
$\texttt{curn} = \texttt{node}(1, \_, kvmL \backslash [(k, \texttt{one}, \neg\texttt{b})] \uplus [(k, \texttt{one}, \texttt{b})], \_, \_, \_)_{\texttt{level}+1}\ \rangle$

// frame off curn

$\{\ \texttt{listProperty(level)} * \exists kvmL.\ \texttt{cur} \mapsto \texttt{node}(1, \_, kvmL, \_, \_, \_)_{\texttt{level}+1} *$
$\texttt{one} \mapsto \texttt{node}(1, \_, \_, \_, \texttt{two}, \_)_{\texttt{level}} \wedge (\_, \texttt{one}, \texttt{b}) = \texttt{last}(kvmL)\ \}$

Use | $\langle\ \texttt{cur} \mapsto \texttt{node}(1, \_, \_, \_, \_, \_)_{\texttt{level}+1}\ \rangle$
`unlock(cur);`
$\langle\ \texttt{cur} \mapsto \texttt{node}(\_, \_, \_, \_, \_, \_)_{\texttt{level}+1}\ \rangle$

$\{\ \texttt{listProperty(level)} * \texttt{cur} \mapsto \texttt{node}(\_, \_, \_, \_, \_, \_) * \texttt{one} \mapsto \texttt{node}(1, \_, \_, \_, \texttt{two}, \_)_{\texttt{level}}\ \}$

Use | $\langle\ \texttt{one} \mapsto \texttt{node}(1, \_, \_, \_, \_, \_)\ \rangle$
`unlock(one);`
$\langle\ \texttt{one} \mapsto \texttt{node}(\_, \_, \_, \_, \_, \_)\ \rangle$

// frame off one

$\{\ \texttt{listProperty(level)} * \texttt{cur} \mapsto \texttt{node}(\_, \_, \_, \_, \_, \_)\ \}$

```
} else { // two ≠ null
```

$\{\ \texttt{listProperty(level)} * (\texttt{cur}, \texttt{curn}) \mapsto \texttt{node}(1, \_, \_, \_, \_, \_)_{\texttt{level}+1} * \ \}$
$\{\ (\texttt{one}, \texttt{onen}) \mapsto \texttt{node}(1, \_, \_, \_, \texttt{two}, \_)_{\texttt{level}} * \texttt{two} \mapsto \texttt{node}(\_, \_, \_, \_, \_, \_)_{\texttt{level}}\ \}$

Use | $\forall l, \mathcal{P}.$
$\langle\ \texttt{two} \mapsto \texttt{node}(l, \mathcal{P})_{\texttt{level}}\ \rangle$
`lock(two);`
$\langle\ \texttt{two} \mapsto \texttt{node}(1, \mathcal{P})_{\texttt{level}}\ \rangle$

$\{\ \texttt{listProperty(level)} * (\texttt{cur}, \texttt{curn}) \mapsto \texttt{node}(1, \_, \_, \_, \_, \_)_{\texttt{level}+1} * \ \}$
$\{\ (\texttt{one}, \texttt{onen}) \mapsto \texttt{node}(1, \_, \_, \_, \texttt{two}, \_)_{\texttt{level}} * \texttt{two} \mapsto \texttt{node}(1, \_, \_, \_, \_, \_)_{\texttt{level}}\ \}$

Use | $\langle\ \texttt{two} \mapsto \texttt{node}(1, \mathcal{P})_{\texttt{level}} * \texttt{twon} = \_\ \rangle$
`twon := get(two);`
$\langle\ \texttt{two} \mapsto \texttt{node}(1, \mathcal{P})_{\texttt{level}} * \texttt{twon} = \texttt{node}(1, \mathcal{P})_{\texttt{level}}\ \rangle$

$$\left\{\begin{array}{l}\texttt{listProperty(level)} * (\texttt{cur}, \texttt{curn}) \mapsto \texttt{node}(1, \_, \_, \_, \_, \_)_{\texttt{level}+1} * \\ (\texttt{one}, \texttt{onen}) \mapsto \texttt{node}(1, \_, \_, \_, \texttt{two}, \_)_{\texttt{level}} * (\texttt{two}, \texttt{twon}) \mapsto \texttt{node}(1, \_, \_, \_, \_, \_)_{\texttt{level}}\end{array}\right\}$$

$$\texttt{if (isIn(curn, two))} \{$$

$$\left\{\begin{array}{l}\texttt{listProperty(level)} * \exists kvmL.\, (\texttt{cur}, \texttt{curn}) \mapsto \texttt{node}(1, \_, kvmL, \_, \_, \_)_{\texttt{level}+1} * \\ (\texttt{one}, \texttt{onen}) \mapsto \texttt{node}(1, \_, \_, \_, \texttt{two}, \_)_{\texttt{level}} * (\texttt{two}, \texttt{twon}) \mapsto \texttt{node}(1, \_, \_, \_, \_, \_)_{\texttt{level}} \wedge \\ [(\_, \texttt{one}, \neg\texttt{b}), (\_, \texttt{two}, \_)] \subseteq kvmL\end{array}\right\}$$

$$\begin{array}{l}\texttt{rearrange(onen, twon, curn);} \\ \texttt{changePointerState(curn, one, b);}\end{array}$$

$$\left\{\begin{array}{l}\texttt{listProperty(level)} * \exists kvmL, kvmL', kvmL_1, kvmL_2, kvmL_1', kvmL_2', mxk_1, \\ mxk_2, mxk_1', delete, nxt_1, nxt_2.\, \texttt{cur} \mapsto \texttt{node}(1, \_, kvmL, \_, \_, \_)_{\texttt{level}+1} * \\ \texttt{curn} = \texttt{node}(1, \_, kvmL', \_, \_, \_)_{\texttt{level}+1} * \texttt{one} \mapsto \texttt{node}(1, \_, kvmL_1, mxk_1, \texttt{two}, 0)_{\texttt{level}} * \\ \texttt{onen} = \texttt{node}(1, \_, kvmL_1', mxk_1', nxt_1, 0)_{\texttt{level}} * \texttt{two} \mapsto \texttt{node}(1, \_, kvmL_2, mxk_2, nxt_2, 0)_{\texttt{level}} * \\ \texttt{twon} = \texttt{node}(1, \_, kvmL_2', mxk_2, nxt_2, delete)_{\texttt{level}} \wedge \dagger^1\end{array}\right\}$$

Use
$$\left|\begin{array}{l}\forall kvL_{\texttt{level}+1}. \\ \left\langle\begin{array}{l}\texttt{cur} \mapsto \texttt{node}(1, \_, kvmL, \_, \_, \_)_{\texttt{level}+1} * \\ \texttt{curn} = \texttt{node}(1, \_, kvmL', \_, \_, \_)_{\texttt{level}+1}\end{array}\right\rangle \\ \qquad \texttt{put(cur, curn);} \\ \left\langle\begin{array}{l}\texttt{cur} \mapsto \texttt{node}(1, \_, kvmL', \_, \_, \_)_{\texttt{level}+1} * \\ \texttt{curn} = \texttt{node}(1, \_, kvmL', \_, \_, \_)_{\texttt{level}+1}\end{array}\right\rangle\end{array}\right.$$

// frame off curn

$$\left\{\begin{array}{l}\texttt{listProperty(level)} * \exists kvmL, kvmL_1, kvmL_2, kvmL_1', kvmL_2', mxk_1, mxk_2, mxk_1', \\ delete, nxt_1, nxt_2.\, \texttt{cur} \mapsto \texttt{node}(1, \_, kvmL, \_, \_, \_)_{\texttt{level}+1} * \\ \texttt{one} \mapsto \texttt{node}(1, \_, kvmL_1, mxk_1, \texttt{two}, 0)_{\texttt{level}} * \texttt{onen} = \texttt{node}(1, \_, kvmL_1', mxk_1', nxt_1, 0)_{\texttt{level}} * \\ \texttt{two} \mapsto \texttt{node}(1, \_, kvmL_2, mxk_2, nxt_2, 0)_{\texttt{level}} * \\ \texttt{twon} = \texttt{node}(1, \_, kvmL_2', mxk_2, nxt_2, delete)_{\texttt{level}} \wedge \dagger^2\end{array}\right\}$$

Use
$$\left|\begin{array}{l}\left\langle\ \texttt{cur} \mapsto \texttt{node}(1, \_, \_, \_, \_, \_)_{\texttt{level}+1}\ \right\rangle \\ \qquad \texttt{unlock(cur)} \\ \left\langle\ \texttt{cur} \mapsto \texttt{node}(\_, \_, \_, \_, \_, \_)_{\texttt{level}+1}\ \right\rangle\end{array}\right.$$

// frame off cur

$$\left\{\begin{array}{l}\texttt{listProperty(level)} * \exists kvmL_1, kvmL_2, kvmL_1', kvmL_2', mxk_1, mxk_2, mxk_1', \\ delete, nxt_1, nxt_2.\, \texttt{one} \mapsto \texttt{node}(1, \_, kvmL_1, mxk_1, \texttt{two}, 0)_{\texttt{level}} * \\ \texttt{onen} = \texttt{node}(1, \_, kvmL_1', mxk_1', nxt_1, 0)_{\texttt{level}} * \texttt{two} \mapsto \texttt{node}(1, \_, kvmL_2, mxk_2, nxt_2, 0)_{\texttt{level}} * \\ \texttt{twon} = \texttt{node}(1, \_, kvmL_2', mxk_2, nxt_2, delete)_{\texttt{level}} \wedge \dagger^3\end{array}\right\}$$

Use
$$\left|\begin{array}{l}\forall addrL_{\texttt{level}}, kvL_{\texttt{level}}. \\ \left\langle\begin{array}{l}\texttt{one} \mapsto \texttt{node}(1, \_, kvmL_1, mxk_1, \texttt{two}, 0)_{\texttt{level}} * \\ \texttt{onen} = \texttt{node}(1, \_, kvmL_1', mxk_1', nxt_1, 0)_{\texttt{level}}\end{array}\right\rangle \\ \qquad \texttt{put(one, onen);} \\ \left\langle\begin{array}{l}\texttt{one} \mapsto \texttt{node}(1, \_, kvmL_1', mxk_1', \texttt{two}, 0)_{\texttt{level}} * \\ \texttt{onen} = \texttt{node}(1, \_, kvmL_1', mxk_1', nxt_1, 0)_{\texttt{level}}\end{array}\right\rangle\end{array}\right.$$

// frame off onen

$$\left\{\begin{array}{l}\texttt{listProperty(level)} * \exists kvmL_1, kvmL_2, kvmL_2', mxk_1, mxk_2, delete, nxt_1, nxt_2. \\ \texttt{one} \mapsto \texttt{node}(1, \_, kvmL_1, mxk_1, nxt_1, 0)_{\texttt{level}} * \texttt{two} \mapsto \texttt{node}(1, \_, kvmL_2, mxk_2, nxt_2, 0)_{\texttt{level}} * \\ \texttt{twon} = \texttt{node}(1, \_, kvmL_2', mxk_2, nxt_2, delete)_{\texttt{level}} \wedge \dagger^4\end{array}\right\}$$

Use
$$\left|\begin{array}{l}\left\langle\ \texttt{one} \mapsto \texttt{node}(1, \_, \_, \_, \_, \_)_{\texttt{level}}\ \right\rangle \\ \qquad \texttt{unlock(one)} \\ \left\langle\ \texttt{one} \mapsto \texttt{node}(\_, \_, \_, \_, \_, \_)_{\texttt{level}}\ \right\rangle\end{array}\right.$$

// frame off one

$$\left\{\begin{array}{l}\texttt{listProperty(level)} * \exists kvmL, kvmL', delete, nxt.\, \texttt{two} \mapsto \texttt{node}(1, \_, kvmL, \_, nxt, 0)_{\texttt{level}} * \\ \texttt{twon} = \texttt{node}(1, \_, kvmL', \_, nxt, delete)_{\texttt{level}} \wedge \dagger^5\end{array}\right\}$$

Use | $\forall kvL_{\texttt{level}}.$
$\left\langle \begin{array}{l} \texttt{two} \mapsto \texttt{node}(1, \_, kvmL, \_, nxt, 0)_{\texttt{level}} * \\ \texttt{twon} = \texttt{node}(1, \_, kvmL', \_, nxt, delete)_{\texttt{level}} \end{array} \right\rangle$
put(two, twon);
$\left\langle \begin{array}{l} \texttt{two} \mapsto \texttt{node}(1, kvmL', \_, nxt, delete)_{\texttt{level}} * \\ \texttt{twon} = \texttt{node}(1, \_, kvmL', mxk, \_, delete)_{\texttt{level}} \end{array} \right\rangle$

// frame off twon

$\left\{ \begin{array}{l} \texttt{listProperty}(\texttt{level}) * \exists delete.\, \texttt{two} \mapsto \texttt{node}(1, \_, \_, \_, \_, delete)_{\texttt{level}} \wedge \\ (((\texttt{two}, \_) \in kvL_{\texttt{level}+1} \wedge delete = 1) \vee ((\texttt{two}, \_) \notin kvL_{\texttt{level}+1} \wedge delete = 0)) \end{array} \right\}$

Use | $\left\langle \texttt{two} \mapsto \texttt{node}(1, \_, \_, \_, \_, \_)_{\texttt{level}} \right\rangle$
unlock(two)
$\left\langle \texttt{two} \mapsto \texttt{node}(\_, \_, \_, \_, \_, \_)_{\texttt{level}} \right\rangle$

// frame back cur

{ listProperty(level) ∗ cur ↦ node(_, _, _, _, _, _) }

} else {

// frame off listProperty

$\left\{ \begin{array}{l} \exists kvmL.\, (\texttt{cur}, \texttt{curn}) \mapsto \texttt{node}(1, \_, kvmL, \_, \_, \_)_{\texttt{level}+1} * \\ (\texttt{one}, \texttt{onen}) \mapsto \texttt{node}(1, \_, \_, \_, \texttt{two}, 0)_{\texttt{level}} * (\texttt{two}, \texttt{twon}) \mapsto \texttt{node}(1, \_, \_, \_, \_, 0)_{\texttt{level}} \wedge \\ (\_, \texttt{one}, \neg\texttt{b}) \in kvmL \wedge (\_, \texttt{two}, \_) \notin kvmL \end{array} \right\}$

Use | $\left\langle \texttt{cur} \mapsto \texttt{node}(1, \_, \_, \_, \_, \_)_{\texttt{level}+1} \right\rangle$
unlock(cur)
$\left\langle \texttt{cur} \mapsto \texttt{node}(\_, \_, \_, \_, \_, \_)_{\texttt{level}+1} \right\rangle$

// frame off cur and curn

$\left\{ \begin{array}{l} (\texttt{one}, \texttt{onen}) \mapsto \texttt{node}(1, \_, \_, \_, \texttt{two}, 0)_{\texttt{level}} * (\texttt{two}, \texttt{twon}) \mapsto \texttt{node}(1, \_, \_, \_, \_, 0)_{\texttt{level}} \wedge \\ (\_, \texttt{one}, \neg\texttt{b}) \in kvL_{\texttt{level}+1} \end{array} \right\}$

Use | $\left\langle \texttt{one} \mapsto \texttt{node}(1, \_, \_, \_, \_, \_)_{\texttt{level}} \right\rangle$
unlock(one)
$\left\langle \texttt{one} \mapsto \texttt{node}(\_, \_, \_, \_, \_, \_)_{\texttt{level}} \right\rangle$

// frame off one and onen

{ (two, twon) ↦ node(1, _, _, _, _, 0)level }

Use | $\left\langle \texttt{two} \mapsto \texttt{node}(1, \_, \_, \_, \_, \_)_{\texttt{level}} \right\rangle$
unlock(two)
$\left\langle \texttt{two} \mapsto \texttt{node}(\_, \_, \_, \_, \_, \_)_{\texttt{level}} \right\rangle$

// frame off two and twon and,
// frame back listProperty and cur

{ listProperty(level) ∗ cur ↦ node(_, _, _, _, _, _) }

}
}
}
}

{ listProperty(level) ∗ cur ↦ node(_, _, _, _, _, _) }

†1 $[(\_, \texttt{one}, \neg\texttt{b}), (\_, \texttt{two}, \_)] \subseteq kvmL \wedge ((|kvmL_1 +\!\!+ kvmL_2| < \max \wedge$
$kvmL'_1 = kvmL_1 +\!\!+ kvmL_2 \wedge kvmL'_2 = kvmL_2 \wedge mxk'_1 = mxk_2 \wedge nxt_1 = nxt \wedge$
$delete = 1 \wedge kvmL' = kvmL \backslash [(k, \texttt{one}, \neg\texttt{b}), (\_, \texttt{two}, \_)] \uplus [(k, \texttt{one}, \texttt{b})]) \vee$
$(|kvmL_1 +\!\!+ kvmL_2| \geqslant \max \wedge kvmL'_1 +\!\!+ kvmL'_2 = kvmL_1 +\!\!+ kvmL_2 \wedge$
$|kvmL'_1| \geqslant \max/2 \wedge |kvmL'_2| \geqslant \max/2 \wedge mxk'_1 = \texttt{first}(kvmL'_2)\lfloor_1 \wedge nxt_1 = \texttt{two} \wedge$
$delete = 0 \wedge kvmL' = kvmL \backslash [(k, \texttt{one}, \neg\texttt{b}), (\_, \texttt{two}, \_)] \uplus [(k, \texttt{one}, \texttt{b}), (mxk'_1, \texttt{two}, \_)]))$

$\dagger^2$ $(\_, \texttt{one}, \texttt{b}) \in kvmL \wedge ((|kvmL_1 +\!\!+ kvmL_2| < \max \wedge (\_, \texttt{two}, \_) \notin kvmL \wedge$
$kvmL'_1 = kvmL_1 +\!\!+ kvmL_2 \wedge kvmL'_2 = kvmL_2 \wedge mxk'_1 = mxk_2 \wedge nxt_1 = nxt \wedge$
$delete = 1) \vee (|kvmL_1 +\!\!+ kvmL_2| \geqslant \max \wedge (mxk'_1, \texttt{two}, \_) \in kvmL \wedge$
$kvmL'_1 +\!\!+ kvmL'_2 = kvmL_1 +\!\!+ kvmL_2 \wedge |kvmL'_1| \geqslant \max/2 \wedge$
$|kvmL'_2| \geqslant \max/2 \wedge mxk'_1 = \texttt{first}(kvmL'_2)\downarrow_1 \wedge nxt_1 = \texttt{two} \wedge delete = 0))$

$\dagger^3$ $(\_, \texttt{one}), \in kvL_{\texttt{level}+1} \wedge ((|kvmL_1 +\!\!+ kvmL_2| < \max \wedge (\_, \texttt{two}) \notin kvL_{\texttt{level}+1} \wedge$
$kvmL'_1 = kvmL_1 +\!\!+ kvmL_2 \wedge kvmL'_2 = kvmL_2 \wedge mxk'_1 = mxk_2 \wedge nxt_1 = nxt \wedge$
$delete = 1) \vee (|kvmL_1 +\!\!+ kvmL_2| \geqslant \max \wedge (mxk'_1, \texttt{two}) \in kvL_{\texttt{level}+1} \wedge$
$kvmL'_1 +\!\!+ kvmL'_2 = kvmL_1 +\!\!+ kvmL_2 \wedge |kvmL'_1| \geqslant \max/2 \wedge$
$|kvmL'_2| \geqslant \max/2 \wedge mxk'_1 = \texttt{first}(kvmL'_2)\downarrow_1 \wedge nxt_1 = \texttt{two} \wedge delete = 0))$

$\dagger^4$ $(\_, \texttt{one}), \in kvL_{\texttt{level}+1} \wedge (((\_, \texttt{two}) \notin kvL_{\texttt{level}+1} \wedge kvmL_2 \subseteq kvmL_1 \wedge kvmL'_2 = kvmL_2 \wedge$
$mxk_1 = mxk_2 \wedge nxt_1 = nxt_2 \wedge delete = 1) \vee ((mxk_1, \texttt{two}) \in kvL_{\texttt{level}+1} \wedge |kvmL_1| \geqslant \max/2 \wedge$
$|kvmL'_2| \geqslant \max/2 \wedge mxk_1 = \texttt{first}(kvmL'_2)\downarrow_1 \wedge nxt_1 = \texttt{two} \wedge delete = 0))$

$\dagger^5$ $((\_, \texttt{two}) \notin kvL_{\texttt{level}+1} \wedge kvmL' = kvmL \wedge delete = 1) \vee$
$((\_, \texttt{two}) \in kvL_{\texttt{level}+1} \wedge |kvmL'| \geqslant \max/2 \wedge \wedge delete = 0)$