Abstract Specifications for Concurrent Maps

Shale Xiong, Pedro da Rocha Pinto, Gian Ntzik, and Philippa Gardner

Imperial College London, UK
{sx14,pmd09,gn408,pg}@ic.ac.uk

Abstract. Despite recent advances in reasoning about concurrent data-structure libraries, the largest implementations in `java.util.concurrent` have yet to be verified. The key problem is in providing modular specifications, that can be used to prove strong functional properties of arbitrary clients, as well as proving implementations correct with respect to such specifications. In this paper, we identify a twofold solution comprising *abstract atomicity* and *generalised capabilities* in the form of partial commutative monoids. We apply this methodology to `java.util.concurrent`. We define an abstract atomic specification for a concurrent map and demonstrate how generalised capabilities allow us to reason about arbitrary clients of the specification, such as a concurrent set and a producer-consumer. Finally, we provide the first proof that the main operations of `ConcurrentSkipListMap` satisfy the abstract map specification.

1 Introduction

Concurrent programs are difficult to write correctly and to verify. When concurrent threads work with shared data, the resulting behaviour can be complex. This leads to the creation of concurrent libraries that provide modules to perform common operations, such as locks and sets. One of the most prominent libraries is the `java.util.concurrent` [17] from Java, and is part of O'Hearn's verification challenges for program logics [21].

Our aim is to specify concurrent library modules, such as those found in Java's concurrent collections library `java.util.concurrent`, to reason about clients using such specifications proving strong functional properties, and verify that implementations are correct with respect to these specifications. In order to reason about clients of such modules we require specifications that abstract their internal complex behaviour. There has been a substantial amount of work in specification of concurrent modules [6,14,15,20,22,24]. However, the largest algorithms of `java.util.concurrent` have yet to be verified. Recent work has demonstrated that two important ingredients are missing in the reasoning: *generalised capabilities* using arbitrary partial commutative monoids and *abstract atomicity* in the spirit of [3,5,16,18].

We focus our work on the concurrent skiplist map implementation and clients, such as concurrent set, and propose a twofold solution.

First, we define an abstract atomic specification for a map, following the specifications of the TaDA [5] program logic, a concurrent separation logic for fine-grained concurrency that introduced atomic triples. Atomicity is a common and useful abstraction for the operations of concurrent data-structures. Intuitively, an operation is abstractly atomic if it appears to take effect at a single instant during its execution. The implementation of the operation may take multiple steps to complete, updating the underlying representation of the data-structure several times. However, only one of these updates should effect the abstract change in the data-structure corresponding to the abstract atomic operation. The benefit of abstract atomicity is that a programmer can use such a data-structure in a concurrent setting while only being concerned with the abstract effects of its operations. Secondly, we make use of the abstract atomic specification to formally verify several clients and implementations. In particular, we verify the main operations of the `ConcurrentSkipListSet` from `java.util.concurrent` that makes use of the map internally. Moreover, we verify that the main operations of `ConcurrentSkipListMap` satisfy the abstract specification. In order to reason about clients and implementations, we make use of *generalised capabilities* in the form of guard resources; instances of partial commutative monoids that control the capability to perform atomic updates.

A key problem in verification is to provide modular specifications. A specification is modular if we can verify clients without reference to the implementation. Previous approaches have provided abstract specifications for sets [2,8] and maps [4]. The specifications allow multiple implementations to be verified against them, without needing to change the proofs for the clients. Nonetheless, they impose limitations on how clients can use the specifications.

Another major problem is the complexity of the concurrent skiplist map implementation. The only map algorithm comparable in size that has been verified, is Sagiv's B^{Link} tree algorithm [26], using concurrent abstract predicates [4]. The main difference is that we prove a stronger specification, which requires proving the atomicity of the operations.

TaDA already supports built-in modular specifications. However, for non-trivial implementations, they expose some of the underlying details in the form of opaque parameters. We extend the TaDA proof rules to allow hiding those underlying details from the specification, enabling us to provide more modular specifications. We show how these specifications and the logic can be used to reason about complex implementations, arbitrary clients, and prove further specifications that display a fiction of disjointness.

Previous works in reasoning about concurrent programs in Java [10,13], and modules from `java.util.concurrent` [1], provide modular specifications. However, those specifications restrict how clients can use the underlying modules and do not allow proving strong functional properties about clients. We provide a detailed account in §5. On the other hand, our specifications are strong enough to prove arbitrary clients and can justify the specifications in existing work. This allows us to demonstrate a more modular client reasoning from what was pre-

$$\vdash \{\mathsf{True}\}\ \texttt{makeMap}()\ \{\exists s \in \mathbb{T}_1.\, \mathsf{Map}(s, \mathsf{ret}, \emptyset)\}$$

$$\vdash \forall \mathcal{M}. \left\langle \begin{array}{c} \mathsf{Map}(s, \mathtt{x}, \mathcal{M}) \\ \wedge\ \mathtt{k} \neq 0 \end{array} \right\rangle \texttt{get}(\mathtt{x}, \mathtt{k}) \left\langle \begin{array}{c} \mathsf{Map}(s, \mathtt{x}, \mathcal{M}) \wedge \underline{\textbf{if}}\ \mathtt{k} \in \mathrm{dom}(\mathcal{M}) \\ \underline{\textbf{then}}\ \mathsf{ret} = \mathcal{M}(\mathtt{k})\ \underline{\textbf{else}}\ \mathsf{ret} = 0 \end{array} \right\rangle$$

$$\vdash \forall \mathcal{M}. \left\langle \begin{array}{c} \mathsf{Map}(s, \mathtt{x}, \mathcal{M}) \\ \wedge\ \mathtt{k} \neq 0 \wedge \mathtt{v} \neq 0 \end{array} \right\rangle \texttt{put}(\mathtt{x}, \mathtt{k}, \mathtt{v}) \left\langle \begin{array}{c} \mathsf{Map}(s, \mathtt{x}, \mathcal{M}[\mathtt{k} \mapsto \mathtt{v}]) \wedge \underline{\textbf{if}}\ \mathtt{k} \in \mathrm{dom}(\mathcal{M}) \\ \underline{\textbf{then}}\ \mathsf{ret} = \mathcal{M}(\mathtt{k})\ \underline{\textbf{else}}\ \mathsf{ret} = 0 \end{array} \right\rangle$$

$$\vdash \forall \mathcal{M}. \left\langle \begin{array}{c} \mathsf{Map}(s, \mathtt{x}, \mathcal{M}) \\ \wedge\ \mathtt{k} \neq 0 \end{array} \right\rangle \texttt{remove}(\mathtt{x}, \mathtt{k}) \left\langle \begin{array}{c} \underline{\textbf{if}}\ \mathtt{k} \notin \mathrm{dom}(\mathcal{M})\ \underline{\textbf{then}}\ \mathsf{Map}(s, \mathtt{x}, \mathcal{M}) \wedge \mathsf{ret} = 0 \\ \underline{\textbf{else}}\ \mathsf{Map}(s, \mathtt{x}, \mathcal{M} \backslash \{(\mathtt{k}, \mathcal{M}(\mathtt{k}))\}) \wedge \mathsf{ret} = \mathcal{M}(\mathtt{k}) \end{array} \right\rangle$$

Fig. 1. Specification for a concurrent map.

viously possible. Additionally, we give the first proof of the main operations of the `ConcurrentSkipListMap` implementation.

The work in this paper demonstrates that choosing the right abstraction for the specification is key in scaling the applicability of the reasoning to reason about clients. In order to prove strong functional properties of clients, we also require strong specifications that do not impose unnecessary constrains on clients. We have successfully applied the methodology to scale the reasoning to a fragment of a large realistic concurrent data-structure implementation. At the moment, all the proofs are done by hand. To scale further to examples of even larger size, will require some degree of automation.

2 Abstract Map Specification

We want to formally specify a fragment of the `ConcurrentMap` module from `java.util.concurrent`. The map module consists of a constructor `makeMap` that creates an empty map, a `get` operation that returns the current value mapped to a key, a `put` operation that changes the mapping associated with a key and a `remove` operation that removes a mapping for a key. None of the operations allows zero to be either a key or a value.

In Fig. 1 we show our abstract specification, i.e. does not contain references to the underlying implementation, for the main operations of the `ConcurrentMap` module. We introduce an abstract predicate $\mathsf{Map}(s, x, \mathcal{M})$ to represent the map as a resource. The first parameter, $s \in \mathbb{T}_1$, can be viewed as a logical address, ranging over an abstract type which is opaque to the client. The second parameter, $x \in \mathsf{Loc}$, is the address of the map object. The final parameter, $\mathcal{M}$, is a set of mappings that represent the state of the map. The `makeMap` operation is specified with a standard Hoare triple that asserts it will return a freshly allocated map object with no mappings. Note that since we use an intuitionistic logic, a precondition with True has a similar meaning to emp used in classical separation logic.

The `get` operation is specified with an *atomic* triple – the operation appears to take place atomically[1]. The significance of this for the program logic is

[1] For a detailed exposition of the atomic triples, see [5,6].

that shared resources can only be accessed by atomic operations; a non-atomic operation could potentially violate any invariant that the shared resources are expected to have.

The `get` operation returns the current mapping associated with the key `k`, or 0 if it does not exist, and leaves the map unchanged. In order to specify that the concurrent environment may change the state of the map while the operation is executing, the set of mappings $\mathcal{M}$ is bound by $\mathbb{V}$.

The `put` operation is similar to `get`, except that it inserts or replaces the mapping for the key `k` with the value `v` ($\mathcal{M}[\mathtt{k} \mapsto \mathtt{v}]$ means updating the existing `k` with value `v`, or adding a new $(\mathtt{k}, \mathtt{v})$ pair). Moreover, it returns the previous mapping associated with the key, or 0 if it does not exist. Finally, the `remove` operation removes an existing mapping associated with the key `k` and returns its previous contents, similarly to the previous operation.

We will show the specification is strong enough to reason about arbitrary clients and verify that a complex implementation satisfies it.

3 Client Reasoning

We motivate the advantages of our specification by showing three different ways to use it. The first is an implementation similar to the `ConcurrentSkipListSet` found in `java.util.concurrent`, that makes use of a map internally. It illustrates the modularity of the specification and helps us introduce some of the proof rules used in the reasoning. The second example is an alternative specification for maps focusing on each of its keys, inspired by Concurrent Abstract Predicates [8,4]. This specification is often useful for clients. The main differences, in our specification, is its atomicity and allowing dynamic control of the number of abstract predicates being used. The third example is a simplified producer-consumer that makes use of the map specification. We use it to show how atomicity can be used to improve client reasoning, allowing us to prove stronger properties about functional correctness than previous approaches [1,2,4].

3.1 Concurrent Set

We consider a concurrent set module, with the specification given in Fig. 2 that is implemented using a concurrent map. The module has two methods, the `setPut` that inserts an element in the set and returns true if the element did not previously exist, and `setRemove` that removes an element from the set and returns true if the element existed. The implementation uses a map to keep track of what elements are in the set. If an element v is in the set, then there will be a corresponding mapping with a key v in the underlying map.

In order to verify the specification using TaDA [5], we must provide an interpretation for the abstract predicate $\mathsf{Set}(s, \mathtt{x}, \mathcal{S})$. For this, we introduce shared regions. A shared region encapsulates some resources that may be shared by multiple threads, with the proviso that the can only be accessed by atomic operations. It has an abstract state which has a concrete interpretation, and

$$\vdash \mathbb{W}\mathcal{S}.\left\langle \mathsf{Set}(s, \mathtt{x}, \mathcal{S}) \right\rangle \ \mathtt{setPut}(\mathtt{x}, \mathtt{v}) \ \left\langle \begin{array}{c} \mathsf{Set}(s, \mathtt{x}, \mathcal{S} \cup \{\mathtt{v}\}) \wedge \\ \underline{\mathbf{if}}\ \mathtt{v} \notin \mathcal{S}\ \underline{\mathbf{then}}\ \mathsf{ret} = \mathtt{true}\ \underline{\mathbf{else}}\ \mathsf{ret} = \mathtt{false} \end{array} \right\rangle$$

$$\vdash \mathbb{W}\mathcal{S}.\left\langle \mathsf{Set}(s, \mathtt{x}, \mathcal{S}) \right\rangle \ \mathtt{setRemove}(\mathtt{x}, \mathtt{v}) \ \left\langle \begin{array}{c} \mathsf{Set}(s, \mathtt{x}, \mathcal{S} \setminus \{\mathtt{v}\}) \wedge \\ \underline{\mathbf{if}}\ \mathtt{v} \in \mathcal{S}\ \underline{\mathbf{then}}\ \mathsf{ret} = \mathtt{true}\ \underline{\mathbf{else}}\ \mathsf{ret} = \mathtt{false} \end{array} \right\rangle$$

Fig. 2. Specification for a concurrent set.

a protocol that determines how the state can change. A shared region is also equipped with resources called guards, which determine how threads can update the region's shared state. The guards for a region form a partial commutative monoid with the operation $\bullet$, which is lifted to $*$ in assertions.

For the set module, which is constructed over the map module, we introduce a region type **SLSet**. The region is parameterised by the logical address and physical address of the underlying map and an additional physical address that corresponds to the address of the set. The abstract state of the region corresponds to the contents of the set. There is a single type of guard resource associated with the region, the guard G. The $\bullet$ operator is defined so that the composition $\mathrm{G} \bullet \mathrm{G}$ is undefined, thus ensuring the guard is unique. The protocol for the **SLSet** region is specified by the following transition system, labelled by guards:

$$\mathrm{G} : \forall \mathcal{S}, \mathcal{S}'. \mathcal{S} \rightsquigarrow \mathcal{S}'$$

This transition allows a thread holding the guard resource G for the region to change the set contents, by changing its abstract state.

We define the interpretation of the region states as follows:

$$I(\mathbf{SLSet}_r(s', x, y, \mathcal{S})) \stackrel{\text{def}}{=} \exists \mathcal{M}. x \mapsto y * \mathsf{Map}(s', y, \mathcal{M}) \wedge \mathcal{S} = \mathrm{dom}(\mathcal{M})$$

It describes a heap cell that contains the address of the underlying map and relates the abstract state of the region to the domain of the mappings.

Now, we can give the interpretation of the abstract type and abstract predicates:

$$\mathbb{T}_2 \stackrel{\text{def}}{=} \mathsf{RId} \times \mathbb{T}_1 \times \mathsf{Loc} \qquad \mathsf{Set}((r, s', y), x, \mathcal{S}) \stackrel{\text{def}}{=} \mathbf{SLSet}_r(s', x, y, \mathcal{S}) * [\mathrm{G}]_r$$

where RId is the set of region identifiers, $\mathbb{T}_1$ is the abstract type of the map specification and Loc is the set of addresses.

It remains to prove that the implementations of the operations satisfy the specifications (given this interpretation). The proof of the `setPut` is given in Fig. 3 and `setRemove` is omitted due to its similarity.

There are two proof rules that are key to atomicity proofs. The first is the make atomic rule which allows us to prove that an operation can be seen as abstractly atomic. A slightly simplified version of this rule is as follows:

$$\frac{\begin{array}{c} \{(x, y) \mid x \in X, y \in Q(x)\} \subseteq \mathcal{T}_{\mathbf{t}}(\mathrm{G})^* \\ r : x \in X \rightsquigarrow Q(x) \vdash \left\{\exists x \in X. \mathbf{t}_r(z, x) * r \Rrightarrow \blacklozenge\right\} \ \mathbb{C} \ \left\{\exists x \in X, y \in Q(x). r \Rrightarrow (x, y)\right\} \end{array}}{\vdash \mathbb{W}x \in X. \left\langle \mathbf{t}_r(z, x) * [\mathrm{G}]_r \right\rangle \ \mathbb{C} \ \left\langle \exists y \in Q(x). \mathbf{t}_r(z, y) * [\mathrm{G}]_r \right\rangle}$$

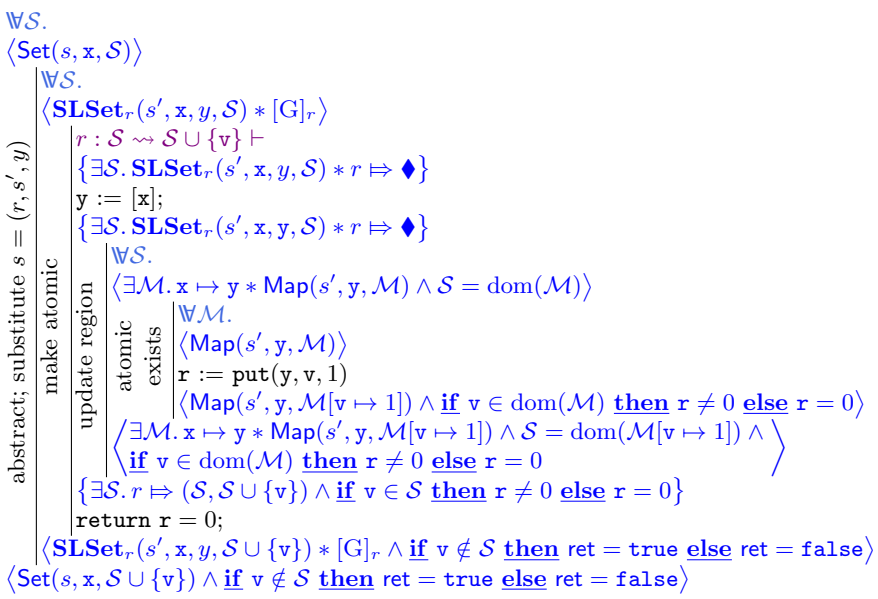


Fig. 3. Proof of correctness of `setPut` operation.

The conclusion of the rule specifies that $\mathbb{C}$ performs an atomic update, transforming the state of a region r (of type $\mathbf{t}$ and parameter z) from $x \in X$ to $y \in Q(x)$. The environment may change the state of the region before the atomic update occurs, so long as the state remains in the set X. Any update to a shared region has to be justified by a guard for that region; in the conclusion rule, this is the guard resource $[\mathrm{G}]_r$. The first premiss of the rule requires that the update from x to y is permitted by the transition system $\mathcal{T}_{\mathbf{t}}$ for region type $\mathbf{t}$ given the guard G; the superscript * indicates the reflexive-transitive closure.

The second premiss ensures that $\mathbb{C}$ actually performs the atomic update. The *atomicity context* $r : x \in X \rightsquigarrow Q(x)$ records the update that the operation must do. The atomic tracking resource $r \Mapsto \blacklozenge$ can be seen as a proxy to the guard; however, it only permits a single update to the region in accordance with the atomicity context. Moreover, until the atomic update has not been performed, the region's state is guaranteed to remain within X. When the update occurs in the region, the atomic tracking resource simultaneously changes to record the actual update performed: $r \Mapsto (x, y)$.

The second key proof rule is the update region rule, which deals with using the atomicity tracking resource to update the region. A simplified version of this rule is as follows:

$$\frac{\vdash \mathbb{V}x \in X. \left\langle I(\mathbf{t}_r(z,x)) * p(x) \right\rangle \mathbb{C} \left\langle \begin{array}{c} \exists y \in Q(x).\, I(\mathbf{t}_r(z,y)) * q_1(x,y) \\ \vee\, I(\mathbf{t}_r(z,x)) * q_2(x) \end{array} \right\rangle}{r : x \in X \rightsquigarrow Q(x) \vdash \mathbb{V}x \in X. \left\langle \begin{array}{c} \mathbf{t}_r(z,x) * p(x) \\ * \, r \Mapsto \blacklozenge \end{array} \right\rangle \mathbb{C} \left\langle \begin{array}{c} \exists y \in Q(x).\, \mathbf{t}_r(z,y) * q_1(x,y) * \\ r \Mapsto (x,y) \vee \mathbf{t}_r(z,x) * q_2(x) * r \Mapsto \blacklozenge \end{array} \right\rangle}$$

$$\{\mathsf{True}\}\ \texttt{makeMap}()\ \{\exists s.\, \mathsf{Collect}(s, \mathsf{ret}, \emptyset)\}$$
$$\vdash \forall v \in \mathbb{N}. \langle \mathsf{Key}(s, \mathtt{x}, \mathtt{k}, v) \wedge \mathtt{k} \neq 0 \rangle\ \texttt{get}(\mathtt{x}, \mathtt{k})\ \langle \mathsf{Key}(s, \mathtt{x}, \mathtt{k}, v) \wedge \mathsf{ret} = v \rangle$$
$$\vdash \forall v \in \mathbb{N}. \langle \mathsf{Key}(s, \mathtt{x}, \mathtt{k}, v) \wedge \mathtt{k} \neq 0 \wedge \mathtt{v} \neq 0 \rangle\ \texttt{put}(\mathtt{x}, \mathtt{k}, \mathtt{v})\ \langle \mathsf{Key}(s, \mathtt{x}, \mathtt{k}, \mathtt{v}) \wedge \mathsf{ret} = v \rangle$$
$$\vdash \forall v \in \mathbb{N}. \langle \mathsf{Key}(s, \mathtt{x}, \mathtt{k}, v) \wedge \mathtt{k} \neq 0 \rangle\ \texttt{remove}(\mathtt{x}, \mathtt{k})\ \langle \mathsf{Key}(s, \mathtt{x}, \mathtt{k}, 0) \wedge \mathsf{ret} = v \rangle$$
$$\mathsf{Collect}(s, x, \mathcal{S}) \iff \mathsf{Collect}(s, x, \mathcal{S} \uplus \{k\}) * \mathsf{Key}(s, x, k, 0), \text{if } k > 0$$

Fig. 4. Key-value specification for a concurrent map.

Note that if $y = x$ in the postcondition, the abstract state of the region is not changed and we can either perform the atomic update or not.

In the original TaDA, the logic allowed introducing existential quantification only for Hoare triples. A rule of the form:

$$\frac{\vdash \langle P(x) \rangle\ \mathbb{C}\ \langle Q(x) \rangle}{\vdash \langle \exists x.\, P(x) \rangle\ \mathbb{C}\ \langle \exists x.\, Q(x) \rangle}$$

is not sound in general. It assumes that the environment is able to change the value of x, while in the premiss the value cannot be changed. This meant that anything that could be potentially existential quantified, had to be exposed as parameters.

We overcome this problem, by extending the logic with the atomic exists proof rule that allows the introduction of existential quantification as follows:

$$\frac{\vdash \forall x \in X. \langle P(x) \rangle\ \mathbb{C}\ \langle Q(x) \rangle}{\vdash \langle \exists x \in X.\, P(x) \rangle\ \mathbb{C}\ \langle \exists x \in X.\, Q(x) \rangle}$$

In this case, we avoid the unsoundness by making sure that $\mathbb{C}$ tolerates changes of the value of x as long as they are contained in X. This proof rule allows us to hide the underlying set of mappings from the map module from the abstract specification of the set module, which was not previously possible.

3.2 Key-value Specification

The specifications shown before are useful for client reasoning. However, some clients work with individual mappings rather than the whole map. Concurrent Abstract Predicates [8,4] have introduced disjoint specifications where one thread can reason about individual mappings, even though they may be implemented by a single shared structure. Their specification did not allow arbitrary sharing for the same mapping. We introduce a specification in Fig. 4 that exhibits the same fiction of disjointness but combines it with atomicity to allow the client to use it without restrictions.

Our specification uses two abstract predicates: Key and Collect. Both predicates are parametrised with s and x that represent the logical and physical address of the map. The $\mathsf{Key}(s, x, k, v)$ predicate represents a mapping with key

k with value v in the map when v is not 0. The $\mathsf{Collect}(s, x, \mathcal{S})$ predicate keeps track of how many Key predicates are in use, one for each element of $\mathcal{S}$. Initially the set is empty. We have an axiom that allows constructing Key predicates by increasing the size of the set tracked by $\mathsf{Collect}$.

In order to verify the specification, we make use of a shared region with type **KVMap** that has two guards associated with it. The first guard is $\mathrm{K}(k)$ and is used to insert or remove a mapping with key k. The second guard is $\textsc{Colect}(\mathcal{S})$ and is used to track how many guards K are in use. We impose the following equivalence on guards:

$$\textsc{Collect}(\mathcal{S}) = \textsc{Collect}(\mathcal{S} \uplus \{k\}) \bullet \mathrm{K}(k)$$

where $k > 0$. This equivalence enforces that the number of guards matches the set in Collect.

We now give the transition system for the region, which requires the guard $\mathrm{K}(k)$ to make changes to the map as follows:

$$\mathrm{K}(k) : \forall v.\, \mathcal{M} \rightsquigarrow \mathcal{M}[k \mapsto v] \qquad \mathrm{K}(k) : \forall v.\, \mathcal{M} \uplus \{(k, v)\} \rightsquigarrow \mathcal{M}$$

The interpretation of the region is defined to hold the map predicate as follows:

$$I(\mathbf{KVMap}_r(s', x, \mathcal{M})) = \mathsf{Map}(s', x, \mathcal{M})$$

We define the interpretation of the abstract predicates as follows:

$$\begin{aligned}
\mathsf{Collect}((r, s'), x, \mathcal{S}) &\stackrel{\text{def}}{=} \exists \mathcal{M}.\, \mathbf{KVMap}_r(s', x, \mathcal{M}) * [\textsc{Collect}(\mathcal{S})]_r \\
&\quad \wedge \mathrm{dom}(\mathcal{M}) \subseteq \mathcal{S} \\
\mathsf{Key}((r, s'), x, k, v) &\stackrel{\text{def}}{=} \exists \mathcal{M}.\, \mathbf{KVMap}_r(s', x, \mathcal{M}) * [\mathrm{K}(k)]_r \\
&\quad \wedge \underline{\textbf{if}}\ v = 0\ \underline{\textbf{then}}\ k \notin \mathrm{dom}(\mathcal{M})\ \underline{\textbf{else}}\ (k, v) \in \mathcal{M}
\end{aligned}$$

The abstract predicate $\mathsf{Collect}((r, s'), x, \mathcal{S})$ asserts that the region contains a map and its keys are a subset of $\mathcal{S}$. Additionally, it contains the guard Collect that can be used to spawn new predicates. The abstract predicate $\mathsf{Key}((r, s'), x, k, v)$ asserts that the region contains a map and describes whether there is a mapping with key k or not, and has the corresponding guard $\mathrm{K}(k)$ that grants permission to change the mapping. The client axiom is shown in Fig. 4 follows directly from the guard equivalence and the interpretation of the abstract predicates.

We show the proof for the `get` operation in Fig. 5. The remaining proofs are similar and are shown in Appendix B.

3.3 Producer-Consumer

We now consider a simplified producer-consumer example that makes use of the map specification. The example, shown in Fig. 6, consists of a program that creates two threads, one that inserts ten elements into a map, and another that removes those elements from the map. Note that our programming language and logic supports dynamic creation of threads using the fork operation. We use the parallel composition ($\|$) in this example to simplify the presentation.

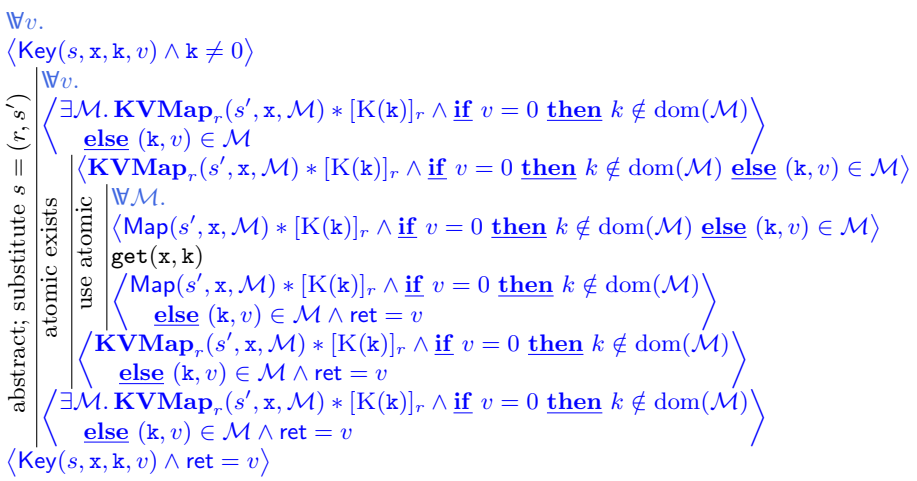


Fig. 5. Proof of correctness of `get` operation.

In order to verify the program, we introduce a region with type name **PC** that encapsulates the map shared among the threads. The region is parametrised by the logical and physical addresses of the map as before, while the abstract state records the set of mappings. Moreover, we introduce a distinguished abstract state $\circ$ to represent that the region is no longer required. The transition system for the **PC** region is as follows:

$$\begin{aligned} \textsc{Put}(\{k\}) &: \forall v, \mathcal{M}.\, \mathcal{M} \rightsquigarrow \mathcal{M} \uplus \{(k, v)\} \\ \textsc{Rem} &: \forall k, v, \mathcal{M}.\, \mathcal{M} \uplus \{(k, v)\} \rightsquigarrow \mathcal{M} \\ \textsc{Put}(\mathbb{N}_1^{10}) \bullet \textsc{Rem} &: \forall \mathcal{M}.\, \mathcal{M} \rightsquigarrow \circ \end{aligned}$$

We use the notation $\mathbb{N}_n^m$ as as shorthand for $\{k \mid k \in \mathbb{N} \wedge n \leq k \leq m\}$. There are two types of guard resources associated with **PC** regions. The guard $\textsc{Put}(\mathcal{S})$ allows a thread to insert any element with a key in $\mathcal{S}$. In order to simplify the reasoning, we allow the $\textsc{Put}$ guards to be combined according to the following equivalence:

$$\textsc{Put}(\mathcal{S}) \bullet \textsc{Put}(\mathcal{S}') = \textsc{Put}(\mathcal{S} \uplus \mathcal{S}')$$

where $\mathcal{S} \uplus \mathcal{S}' \subseteq \mathbb{N}_1^{10}$, as in this example we are only working with that range.

The second type of guard resources is $\textsc{Rem}$ that provides the capability to remove any element from the map. Here, $\bullet$ operator is also defined so that the following compositions are undefined:

$$\begin{aligned} &\textsc{Rem} \bullet \textsc{Rem} \\ &\textsc{Put}(\mathcal{S}) \bullet \textsc{Put}(\mathcal{S}') \qquad \text{if } \mathcal{S} \cap \mathcal{S}' \neq \emptyset \end{aligned}$$

This guarantees that there is only one guard of type $\textsc{Rem}$ and one guard of type $\textsc{Put}$ for each key. It remains to define the interpretation of the region states:

$$\begin{aligned} I(\mathbf{PC}_r(s, \mathtt{x}, \mathcal{M})) &\stackrel{\text{def}}{=} \mathsf{Map}(s, \mathtt{x}, \mathcal{M}) * [\textsc{Put}(\mathrm{dom}(\mathcal{M}))]_r \wedge \mathrm{dom}(\mathcal{M}) \subseteq \mathbb{N}_1^{10} \\ I(\mathbf{PC}_r(s, \mathtt{x}, \circ)) &\stackrel{\text{def}}{=} \mathsf{True} \end{aligned}$$

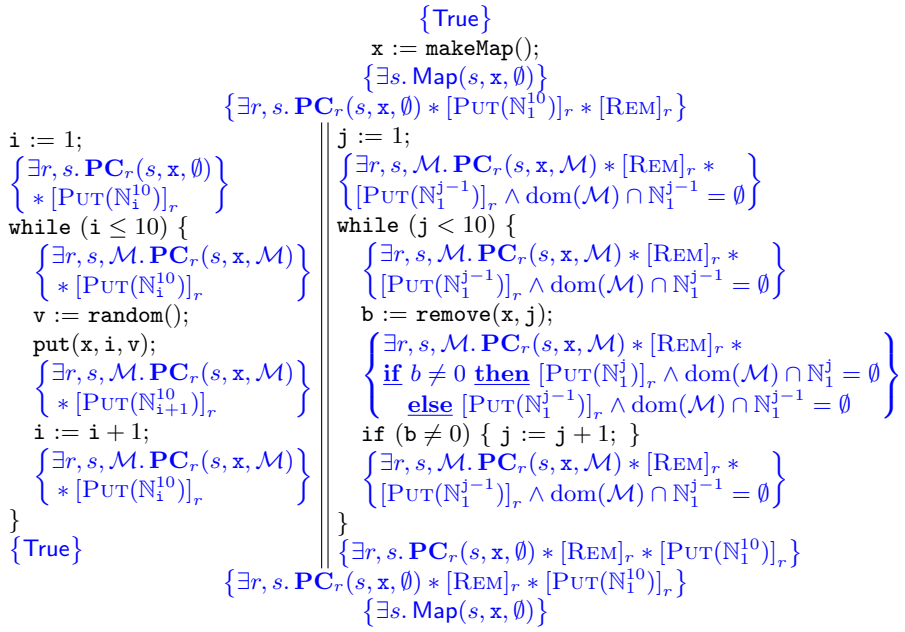


Fig. 6. Producer-consumer correctness proof.

We require that after a thread inserts an element into the map, it must also leave the corresponding Put guard that allowed it inside the region. This ensures that when a thread removes the element, it can guarantee that other threads do not insert it back by owning the guard that allowed it. Additionally, by interpreting the state $\circ$ as True, we allow a thread transitioning into the state $\circ$ to acquire the map that previously belonged to that region.

So far, we have not discussed how regions are created when doing proofs.

$$\frac{\mathrm{G} \in \mathcal{G}_{\mathbf{t}} \quad \forall r.\, p * [\mathrm{G}]_r \implies I(\mathbf{t}^{\lambda}_r(z, x)) * q(r)}{p \implies \exists r.\, \mathbf{t}^{\lambda}_r(z, x) * q(r)}$$

The conclusion of the rule specifies that we can create a new region with a freshly allocated guard. While the premiss ensures that we need to have enough resources combined with freshly allocated guard resource that is part of the region type, to entail the interpretation of a region in a particular state. In our case, we allocate the guard $\textsc{Put}(\mathbb{N}_1^{10}) \bullet \textsc{Rem}$ when we create the region. Moreover, it is important to notice that after the producer thread inserts the last element, it no longer has a guard Put and as such, we can no longer guarantee that the map is inside the region. It could be the case that the environment has transitioned the region to the state $\circ$. Therefore, when we write $\exists \mathcal{M}$ in the producer thread, it includes the $\circ$ state when there are no guard resources.

This example shows the importance of atomic specifications. In previous work, the map specifications exposed permissions which embodied decisions

about how clients should use the map [4]. In particular, when `put` and `remove` operations occurred in parallel it was not possible to prove that the map was empty at the end, shown in Appendix C,D. In contrast, our atomic specifications make no such restrictions. A client is free to use a partial commutative monoid of his choosing for the guard resources that express capabilities over the shared state. We show further examples of how to use the set specification in Appendix A, by proving a parallel sieve of Eratosthenes.

4 Concurrent Skiplist Map

In §2 we have seen an abstract specification for the `ConcurrentMap`. One of its implementations is the `ConcurrentSkipListMap`. We extract the main structure of the `ConcurrentSkipListMap` implementation and prove that the `put` and `remove` operations satisfy the atomic specification in Fig. 1.

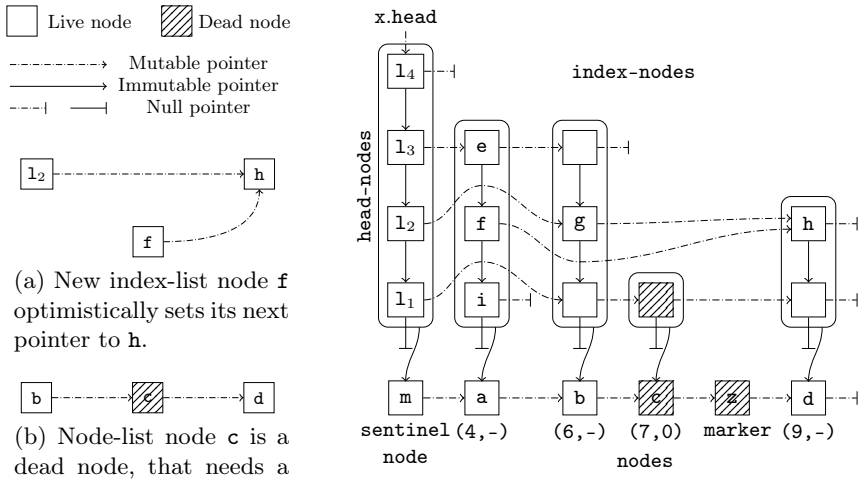
Data Structure A skiplist is built from layers of linked lists. The bottom layer is a linked list of key-value pairs, stored in key order. Each higher layer is a linked list that acts as an index to the layer below it, with approximately half the number of nodes. This arrangement results in fast search performance, comparable to that of B-trees [25].

Figure 7c depicts a snapshot of a concurrent skiplist. The linked list of the bottom layer is the *node list*. Each layer above the node list is an *index list*. The first node of the node list is always the *sentinel* node. Non-sentinel nodes store key-value pairs with immutable keys. A *dead* node is one whose value is 0. A *marker* node is one whose value is a pointer to itself. Marker nodes are used to ensure the correct removal of nodes from the skiplist, as we will explain shortly.

Node-list nodes store the key-value mappings of the map. An index-list node stores three pointers as its value: a next-pointer to the next index-list node, a down-pointer to a node in the lower index-list, and a node-pointer to a node in the node list. The key of an index-list node is the key of the node-list node it points to. All index-list nodes connected by their down-pointers should point to the same node-list node. For instance in Fig. 7c, the index-list nodes, `e`, `f` and `i` are grouped vertically because they all point to the node-list node `a`. The head node of an index list additionally stores the level of the index list.

A search for a key proceeds from the top index list left to right, until the current key is greater or equal to the target key. If the keys are equal, the key has been found. Otherwise, the search continues from the next layer down.

The `put` operation searches for the given key and if the key is found, it replaces the existing mapping by performing an atomic *compare-and-set* (`CAS`), of the existing value with the new value. If the key is not found, a new node-list node is added as the successor of the node with the greatest key that is lower than the key being added. When a node-list node is added, certain index-list nodes are added as well. Nodes are added *optimistically*, by setting the next-pointer of each new node to the successor of their predecessor, as depicted in Fig. 7a, and then performing a `CAS` on the next-pointer of the predecessor to the



(a) New index-list node `f` optimistically sets its next pointer to `h`.

(b) Node-list node `c` is a dead node, that needs a marker node appended.

(c) A snapshot of a concurrent skiplist.

Fig. 7. The structure of a concurrent skiplist.

new node. The process may restart due to interleaving from other threads. In Fig. 7a and Fig. 7c, we illustrate a case where the process of adding the index-list node `f` restarts, because it is interleaved with the addition of `g`.

To delete a key-value mapping, `remove` `CAS`s the value of its node to 0, turning it, and all index-list nodes pointing to it, into a dead node. Dead nodes are physically removed by subsequent key searches, by `CAS`ing the next-pointer of its predecessor to its successor. However, removing a node-list node is subtle, because a new node may be concurrently added to the same position of the node being removed [11]. To prevent this, a marker node is added as a successor to the node being removed first, which stops other threads from modifying the structure between the marker and its successor; then, the marker and the dead node are removed. In Fig. 7b and Fig. 7c, we illustrate deleting the node `c`.

Predicates In Table 1 we introduce predicates for the various skiplist node types, as well as reachability predicates. They are formally defined in §E.1.

To prove that the skiplist satisfies the map specification, we introduce a new region of type **SLMap**. The region is parametrised by the address x, a list of key-value pairs $\mathcal{L}$, a set of live nodes $\mathcal{N}$, a set of dead nodes $\mathcal{B}$ and an immutable sentinel node m. For example, in Fig. 7c $x = \mathtt{x}$, $\mathcal{L} = [(4,-),(6,-),(9,-)]$, $\mathcal{N} = \{\mathtt{a},\mathtt{b},\mathtt{d}\}$, $\mathtt{c} \in \mathcal{B}$ and $m = \mathtt{m}$. The interpretation of the **SLMap** region is defined in terms of the index-lists denoted by the iLists predicate and the node-list denoted by the nList predicate as follows:

$$I(\mathbf{SLMap}_r(x,\mathcal{M},\mathcal{N},\mathcal{B},m)) \stackrel{\text{def}}{=} \exists h.\, x.\mathtt{head} \mapsto h * \mathsf{iLists}(h,m,\mathcal{N},\mathcal{B}) * \mathsf{nList}(m,\mathcal{M},\mathcal{N},\mathcal{B})$$

The iLists predicate describes all the index lists and is defined below:

$$\mathcal{P} \in (\mathtt{Loc} \times \mathtt{Loc}^*)^*$$
$$\mathsf{iLists}(h,m,\mathcal{N},\mathcal{B}) \stackrel{\text{def}}{=} \exists \mathcal{P},\mathcal{H}.\, \mathcal{P}\downarrow_1 \in \mathcal{N} \uplus \mathcal{B} \wedge \mathsf{wellForm}(\mathcal{P}) * \mathsf{row}(m,\mathcal{H} +\!\!+ [h],\mathcal{P},l)$$

Table 1. Auxiliary predicates

Predicate	Meaning	Example in Fig. 7c
$\mathsf{node}(x, k, v, n)$	A node-list node at the address x with a key-value pair (k, v) and its next node n.	$\mathsf{node}(\mathtt{a}, 4, -, \mathtt{b})$
$\mathsf{index}(x, p, d, n)$	An index-list node at the address x with three pointers, to a node-list node p, to its down node d and to its next node n.	$\mathsf{index}(\mathtt{f}, \mathtt{a}, \mathtt{i}, \mathtt{h})$
$\mathsf{head}(x, l, p, d, n)$	A head node in level l at the address x with p, d and n having the same meaning as in index.	$\mathsf{head}(\mathtt{l}_2, 2, \mathtt{m}, \mathtt{l}_1, \mathtt{g})$
$\mathsf{marker}(x, n)$	A marker at the address x and its next node n.	$\mathsf{marker}(\mathtt{z}, \mathtt{d})$
$x \rightsquigarrow^* y$	The node y is reachable from x through next pointers.	$\mathtt{a} \rightsquigarrow^* \mathtt{c}$
$x \rightsquigarrow^* \mathcal{N}$	A node from $\mathcal{N}$ is reachable from the node x.	$\mathtt{m} \rightsquigarrow \{\mathtt{a}, \mathtt{b}\}$

The state $\mathcal{P}$ is a list of tuples. The first element of the tuple is an address of a node-list node and the second element is a list of addresses of index-list nodes. The wellForm predicate asserts that the order of the tuples in the list $\mathcal{P}$, is the order of the first elements of the tuples.

$$\mathsf{wellForm}(\mathcal{P}) \stackrel{\text{def}}{=} \mathcal{P} = [] \vee \exists p, k, \mathcal{I}, i, \mathcal{I}', \mathcal{P}'. \begin{pmatrix} \mathcal{P} = [(p, \mathcal{I})] +\!\!+ \mathcal{P}' \wedge \mathcal{I} = [i] +\!\!+ \mathcal{I}' \wedge \mathsf{wellForm}(\mathcal{P}') * \mathsf{node}(p, k, -, -) \wedge \\ \bigwedge_{p' \in \mathcal{P}'\downarrow_1} (\exists k'.\, \mathsf{node}(p', k', -, -) \wedge k < k') \wedge \mathsf{chain}(p, i, \mathcal{I}') \end{pmatrix}$$

$$\mathsf{chain}(p, i, \mathcal{I}) \stackrel{\text{def}}{=} (\mathcal{I} = [] \wedge i = 0) \vee (\exists i', \mathcal{I}'. \mathcal{I} = [i'] +\!\!+ \mathcal{I}' \wedge \mathsf{index}(i, p, i', -) * \mathsf{chain}(p, i', \mathcal{I}'))$$

The chain predicate asserts that the second element of each tuple in $\mathcal{P}$, denoted by $\mathcal{I}$, is a list of addresses of index-list nodes that point to the same node-list node. Intuitively, the predicate describes a grouped column in Fig. 7c.

A layer of an index-list is described by the row predicate. It is a linked list that starts with a head node followed by index nodes, defined as:

$$\mathsf{row}(m, \mathcal{H}, \mathcal{P}, l) \stackrel{\text{def}}{=} l = 0 \vee \exists i. \begin{pmatrix} \mathsf{head}(\mathcal{H}(l), l, m, \mathcal{H}(l-1), i) * \\ \mathsf{iTail}(i, getIndex(\mathcal{P}, l)) * \mathsf{row}(m, \mathcal{H}, \mathcal{P}, l-1) \end{pmatrix}$$

$$getIndex(\mathcal{P}, l) \stackrel{\text{def}}{=} \{p | \forall i.\, p = \mathcal{P}\downarrow_2 (i)(l)\}$$

$$\mathsf{iTail}(i, \mathcal{I}) \stackrel{\text{def}}{=} (\mathcal{I} = \emptyset \wedge i = 0) \vee (\exists i'.\, i, i' \in \mathcal{I} \wedge \mathsf{index}(i, -, -, i') * \mathsf{iTail}(i', \mathcal{I} \setminus \{i\}))$$

where $\mathcal{H}(l)$ returns the l-th head node in $\mathcal{H}$, and $getIndex$ returns all the addresses of the index-nodes at level l. Note that head nodes point to the sentinel node m.

The nList predicate describes the node-list of the skiplist. It starts with the sentinel node, followed by live nodes, containing the key-value pairs of the map, as well as some dead and marker nodes pending removal.

$$\mathsf{nList}(m, \mathcal{M}, \mathcal{N}, \mathcal{B}) \stackrel{\text{def}}{=} \mathsf{nTail}(m, toList(\mathcal{M}), \mathcal{N}, \mathcal{B}) \wedge \bigwedge_{n \in \mathcal{B}} (n \rightsquigarrow^* \mathcal{N} \uplus \{0\})$$

$$\mathsf{nTail}(n, \mathcal{L}, \mathcal{N}, \mathcal{B}) \stackrel{\text{def}}{=} (\mathcal{N} = \emptyset \wedge \mathcal{L} = [] \wedge n = 0) \vee \begin{pmatrix} \exists n' \in \mathcal{N} \uplus \mathcal{B}, k, v, \mathcal{L}'.\, n \in \mathcal{N} \wedge \mathcal{L} = [k, v] +\!\!+ \mathcal{L}' \wedge \\ \mathsf{node}(n, k, v, n) * \mathsf{nTail}(n', \mathcal{L}', \mathcal{N} \setminus \{n\}, \mathcal{B}) \end{pmatrix} \vee \begin{pmatrix} \exists n' \in \mathcal{N} \uplus \mathcal{B}, .\, n \in \mathcal{B} \wedge \mathsf{nTail}(n', \mathcal{L}, \mathcal{N}, \mathcal{B} \setminus \{n\}) * \\ (\exists n''.\, \mathsf{node}(n, k, v, n') \vee (\mathsf{node}(n, k, v, n'') * \mathsf{marker}(n'', n'))) \end{pmatrix}$$

The *toList* function returns a list of key-value pairs in order of ascending keys. All the dead nodes in $\mathcal{B}$ must point back to a live node or 0. The $\mathsf{node}(x, k, v, n)$ predicate asserts a node-list node at the address x that stores the key-value pair (k, v) and the next-pointer n, defined as:

$$\mathsf{node}(x, k, v, n) \stackrel{\text{def}}{=} x.\mathtt{key} \mapsto k * x.\mathtt{value} \mapsto v * x.\mathtt{next} \mapsto n$$

We use the field notation $E.\mathtt{field}$ as shorthand for $E + \mathsf{offset}(\mathtt{field})$. Here, $\mathsf{offset}(\mathtt{key}) = 0$, $\mathsf{offset}(\mathtt{value}) = 1$, and $\mathsf{offset}(\mathtt{next}) = 2$.

We associate the **SLMap** region with a single guard L, with $\mathrm{L} \bullet \mathrm{L}$ being undefined. The labelled transition system of the region comprises three transitions:

$$\begin{array}{l}
\mathrm{L} : \forall \mathcal{L}, \mathcal{N}, \mathcal{B}, k, v, v'.\, (\mathcal{M}[k \mapsto v], \mathcal{N}, \mathcal{B}) \rightsquigarrow (\mathcal{M}[k \mapsto v'], \mathcal{N}, \mathcal{B}) \\
\mathrm{L} : \forall \mathcal{L}, \mathcal{N}, \mathcal{B}, k, v, n.\, (\mathcal{M}, \mathcal{N}, \mathcal{B}) \rightsquigarrow (\mathcal{M} \uplus \{(k, v)\}, \mathcal{N} \uplus \{n\}, \mathcal{B}) \\
\mathrm{L} : \forall \mathcal{L}, \mathcal{N}, \mathcal{B}, k, v, n.\, (\mathcal{M} \uplus \{(k, v)\}, \mathcal{N} \uplus \{n\}, \mathcal{B}) \rightsquigarrow (\mathcal{M}, \mathcal{N}, \mathcal{B} \uplus \{n\})
\end{array}$$

The first replaces an old value v with a new value v', the second inserts a new key-value pair (k, v) and its corresponding new node n, and the third deletes a key k and moves the node n to the dead nodes set.

Finally, we instantiate the abstract map predicate with the region and its guard:

$$\mathsf{Map}((r, m), x, \mathcal{M}) \stackrel{\text{def}}{=} \exists \mathcal{N}, \mathcal{B}.\, \mathbf{SLMap}_r(x, \mathcal{M}, \mathcal{N}, \mathcal{B}, m) * [\mathrm{L}]_r$$

Proof of the `put` Operation Figure 8 shows a sketch proof that `put` satisfies the specification given in Fig. 1. The implementation is given in a simple imperative programming language, following the structure of the implementation found in `java.util.concurrent`. In this language, the semantics of single heap cell writes are equivalent to those of `volatile` fields in Java. We use the `outer`, `inner` and `continue` variables to emulate the control flow operators `break` and `continue`.

For brevity, we use $\top$ and $\bot$ to denote `true` and `false` respectively. We use the flow predicate for control flow within the outer loop.

$$\top \equiv \mathtt{true} \quad \bot \equiv \mathtt{false} \quad \mathsf{flow}(c, i) \equiv \mathtt{continue} = c \wedge \mathtt{inner} = i$$

The $\mathsf{wkNdReach}$ predicate used in the proof assserts that either x can reach y, or y is a dead node. Since the algorithm works on three nodes `b`, `n` and `f`, we use the predicate bnf to describe their reachability relation.

$$\begin{array}{c}
\mathsf{wkNdReach}(x, y) \stackrel{\text{def}}{=} x \rightsquigarrow^* y \vee ((y \rightsquigarrow^* \mathcal{N} \uplus \{0\}) \wedge y \in \mathcal{B}) \\
\mathsf{bnf} \equiv \mathsf{wkNdReach}(\mathtt{b}, \mathtt{n}) * \mathsf{wkNdReach}(\mathtt{n}, \mathtt{f}) \wedge (\mathtt{b.key} < \mathtt{k} \vee \mathtt{b} = m)
\end{array}$$

Note that either the node `b` is the sentinel node m, or its key is smaller then the `k` parameter of the operation. The $\mathsf{restart}$ predicate describes the cases when the algorithm needs to restart: a `CAS` fails; `n` is a dead or marker node; `b` is a dead node; `n2`, which is the second read of the successor of node `b`, is not `n`.

$$\mathsf{restart} \equiv \mathtt{cas} = \bot \vee \mathtt{n.value} = 0 \vee \mathsf{marker}(\mathtt{n}, -) \vee \mathtt{b.value} = 0 \vee \mathtt{n} \neq \mathtt{n2}$$

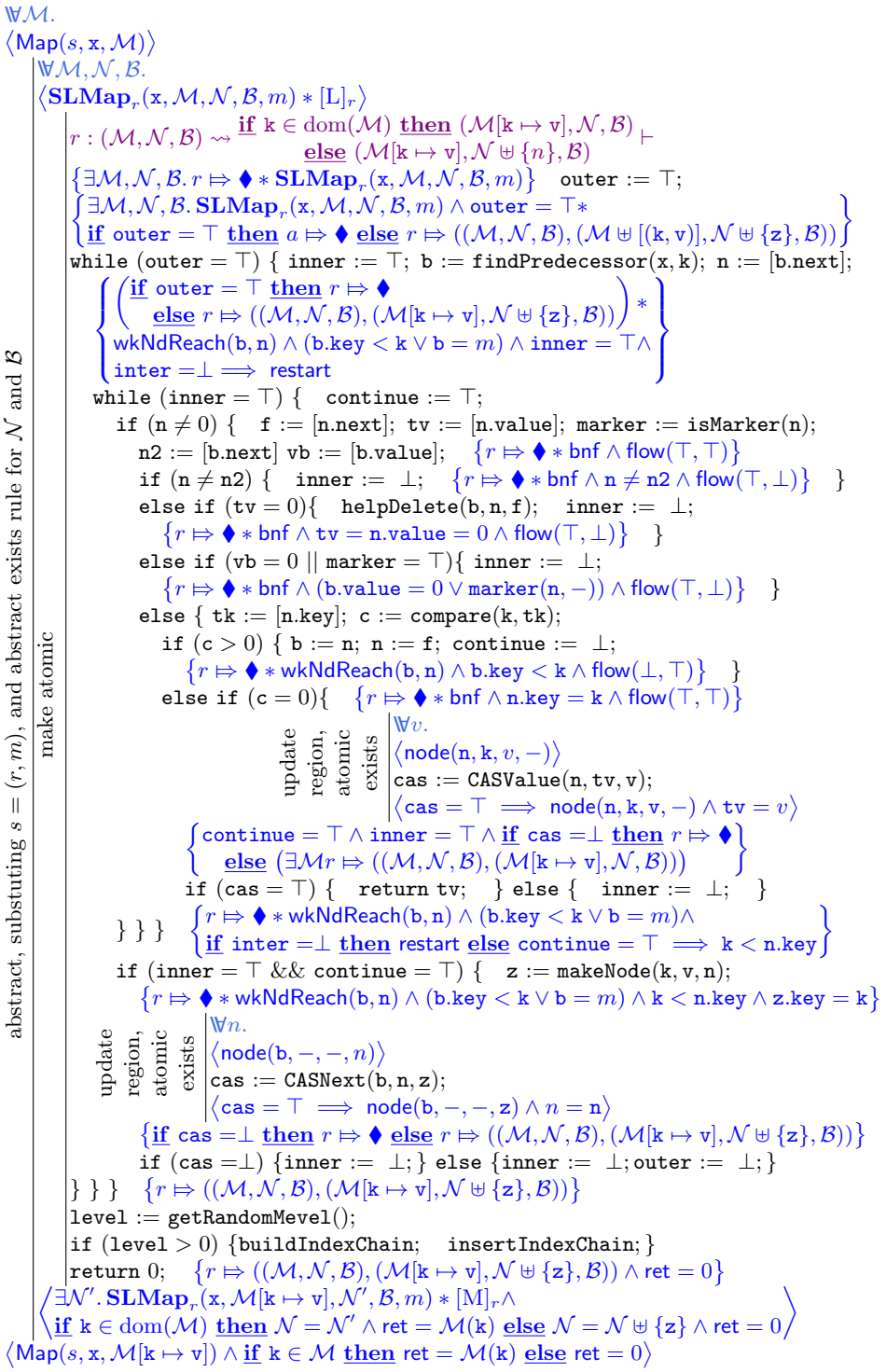


Fig. 8. Skiplist's `put` proof.

Additional specifications and proofs of auxiliary operations used in the proof of `put`, such as `CASValue` and `findPredecessor`, are given in Appendix E. We prove that the skiplist implementation of the `remove` operation satisfies the specification of Fig. 1, in Appendix E.3.

5 Related Work

We have briefly described the main challenges of concurrent reasoning in the introduction. Here, we give an account of reasoning about concurrent Java programs and `java.util.concurrent`.

Amighi *et al.* [1] used concurrent separation logic with permissions to reason about several lock modules in `java.util.concurrent` and their clients. Their approach is modular, but is not general enough to prove strong functional properties about arbitrary clients. Others have used separation logic to reason about spin locks [5,8,16,20,28] and ticket locks [8,27], but have not applied their work to `java.util.concurrent` which requires reentrant locks.

Blom *et al.* [2] reason about concurrent sets in a Java-like language, using a proof theory based on concurrent separation logic extended with histories and permissions. They specify and prove a coarse-grained concurrent set implementation using a specification that exposes histories to the client. The histories allow them to prove properties about client in a modular way. However, the concurrent set specification is restricted, limiting its applicability to the clients. There has been some work on lock-coupling list implementations of concurrent sets, such as the original work on Concurrent Abstract Predicates reasoning [8,9]. Moreover, Liang and Feng [19], and separately O'Hearn *et al.* [23], have reasoned about Heller's lazy set [12], although their techniques are not able to reason about clients. Again, this work is not aimed at `java.util.concurrent`.

In this paper, using TaDA [5], we have specified concurrent maps, have given modular proofs of functional properties of clients, and have verified the main operations of the `ConcurrentSkiplistMap` from `java.util.concurrent`. The most challenging part was the verification of the skiplist algorithm, due to its size and complexity. As far as we know, this is the first formal proof of this algorithm. In fact, there has been little reasoning about concurrent maps in general. Da Rocha Pinto *et. al.* use Concurrent Abstract Predicates [8] to develop map specifications that allow thread-local reasoning combined with races over elements of the data structure [4]. They have proved several map implementations including Sagiv's B^{Link} Tree algorithm [26]. However, their approach suffers the same shortcomings as the specifications for sets previously described. In general, this means that we cannot prove strong functional properties about a client using the map, e.g. when two threads perform concurrent insert and remove operations over the same key, we can only conclude the set of possibilities at the end of the execution. The closest specification to our work is also based on TaDA, is used to verify a linked list based map in a total-correctness setting [7]. We improve upon their specification by having a more abstract specification. This was possible with the way we extended the logic to introduce the existential quantifier

in atomic triples. Using our map specification in §2 we can reason about clients with arbitrary protocols and prove strong functional properties about clients, as shown in §3.3.

6 Conclusions and Future Work

We have presented an abstract specification for a concurrent map that captures the atomicity intended in the `java.util.concurrent` English specification [17] and is suitable for all clients. We used the specification to prove a set implementation that makes use of the map internally. Moreover, we have shown that further specifications can be built over it, such as the key-value specification that exhibits a fiction of disjointness, and proved an example that was not possible in previous work [1,2,4]. Lastly, we have given the first formal proof of the `ConcurrentSkipListMap` and shown it satisfies the atomic map specification.

So far, all the proofs using TaDA [5] have been done by hand. Despite this, our examples are comparatively substantial. Indeed, the verification of the `ConcurrentSkipListMap` is one of the most complex examples studied in the literature. We do recognise the need for mechanisation and automation. We are currently working on a semi-automated verification tool on a fragment of TaDA. This will allow us to improve the confidence of the proofs and to tackle more of the `java.util.concurrent` package.

Another avenue for future work is to adapt TaDA to handle object-oriented features from Java, such as monitors, inheritance and interfaces. Our logical abstractions seem particularly well suited to reason about the abstractions provided by object-oriented languages.

Acknowledgements. We thank Julian Sutherland for useful feedback. This research was supported by EPSRC Programme Grants EP/H008373/1 and EP/K0 08528/1, and the Dept. of Computing in Imperial College London.

References

1. Amighi, A., Blom, S., Huisman, M., Mostowski, W., Zaharieva-Stojanovski, M.: Formal Specifications for Java's Synchronisation Classes. In: Proceedings of PDP '14. pp. 725–733 (2014)
2. Blom, S., I, M., Zaharieva-Stojanovski, M.: History-based Verification of Functional Behaviour of Concurrent Programs. In: Proceedings of SEFM '15, pp. 84–98 (2015)
3. Calcagno, C., O'Hearn, P.W., Yang, H.: Local Action and Abstract Separation Logic. In: Proceedings of LICS '07. pp. 366–378 (2007)
4. da Rocha Pinto, P., Dinsdale-Young, T., Dodds, M., Gardner, P., Wheelhouse, M.J.: A Simple Abstraction for Complex Concurrent Indexes. In: Proceedings of OOPSLA '11. pp. 845–864 (2011)
5. da Rocha Pinto, P., Dinsdale-Young, T., Gardner, P.: TaDA: A Logic for Time and Data Abstraction. In: Proceedings of ECOOP '14. LNCS, vol. 8586, pp. 207–231 (2014)

6. da Rocha Pinto, P., Dinsdale-Young, T., Gardner, P.: Steps in Modular Specifications for Concurrent Modules (Invited Tutorial Paper). Electr. Notes Theor. Comput. Sci. 319, 3–18 (2015)
7. da Rocha Pinto, P., Dinsdale-Young, T., Gardner, P., Sutherland, J.: Modular Termination Verification for Non-blocking Concurrency. In: Proceedings of ESOP '16. pp. 176–201 (2016)
8. Dinsdale-Young, T., Dodds, M., Gardner, P., Parkinson, M.J., Vafeiadis, V.: Concurrent Abstract Predicates. In: Proceedings of ECOOP '10. pp. 504–528 (2010)
9. Dodds, M., Feng, X., Parkinson, M., Vafeiadis, V.: Deny-Guarantee Reasoning. In: Proceedings of ESOP '09. pp. 363–377 (2009)
10. Haack, C., Hurlin, C.: Separation Logic Contracts for a Java-like Language with Fork/Join. In: Proceedings AMAST '08. pp. 199–215 (2008)
11. Harris, T.L.: A Pragmatic Implementation of Non-blocking Linked-Lists. In: Proceedings of DISC '01. pp. 300–314 (2001)
12. Heller, S., Herlihy, M., Luchangco, V., Moir, M., Scherer, W.N., Shavit, N.: A Lazy Concurrent List-based Set Algorithm. In: Proceedings of OPODIS '05. pp. 3–16 (2006)
13. Hurlin, C.: Specification and Verification of Multithreaded Object-oriented Programs with Separation Logic. Ph.D. thesis, Université Nice Sophia Antipolis (2009)
14. Jacobs, B., Piessens, F.: Expressive Modular Fine-grained Concurrency Specification. In: Proceedings of POPL '11. pp. 271–282 (2011)
15. Jones, C.B.: Specification and Design of (Parallel) Programs. In: IFIP Congress. pp. 321–332 (1983)
16. Jung, R., Swasey, D., Sieczkowski, F., Svendsen, K., Turon, A., Birkedal, L., Dreyer, D.: Iris: Monoids and Invariants As an Orthogonal Basis for Concurrent Reasoning. In: Proceedings of POPL '15. pp. 637–650 (2015)
17. Lea, D., et al.: Java Specification Request 166: Concurrency Utilities (2004)
18. Ley-Wild, R., Nanevski, A.: Subjective Auxiliary State for Coarse-grained Concurrency. In: ACM SIGPLAN Notices. vol. 48, pp. 561–574 (2013)
19. Liang, H., Feng, X.: Modular Verification of Linearizability with Non-fixed Linearization Points. SIGPLAN Not. 48(6), 459–470 (2013)
20. Nanevski, A., Ley-Wild, R., Sergey, I., Delbianco, G.A.: Proceedings of ESOP '14, chap. Communicating State Transition Systems for Fine-Grained Concurrent Resources, pp. 290–310 (2014)
21. O'Hearn, P.W.: Scalable Specification and Reasoning: Challenges for Program Logic. In: Proceedings of VSTTE '05. LNCS, vol. 4171, pp. 116–133 (2005)
22. OHearn, P.W.: Resources, Concurrency, and Local Reasoning. Theor. Comput. Sci. 375(1-3), 271–307 (2007)
23. O'Hearn, P.W., Rinetzky, N., Vechev, M.T., Yahav, E., Yorsh, G.: Verifying Linearizability with Hindsight. In: Proceedings of PODC '10. pp. 85–94 (2010)
24. Owicki, S., Gries, D.: An axiomatic proof technique for parallel programs I. Acta Informatica 6(4), 319–340 (1976)
25. Pugh, W.: Skip Lists: A Probabilistic Alternative to Balanced Trees. Commun. ACM 33(6), 668–676 (1990)
26. Sagiv, Y.: Concurrent Operations on B-trees with Overtaking. In: Proceedings of PODS '85. pp. 28–37 (1985)
27. Sergey, I., Nanevski, A., Banerjee, A.: Mechanized Verification of Fine-grained Concurrent Programs. In: Proceedings of PLDL '15. pp. 77–87 (2015)
28. Svendsen, K., Birkedal, L.: Impredicative Concurrent Abstract Predicates. In: Proceedings of ESOP 2014. pp. 149–168. Berlin, Heidelberg (2014)

A Sieve of Eratosthenes

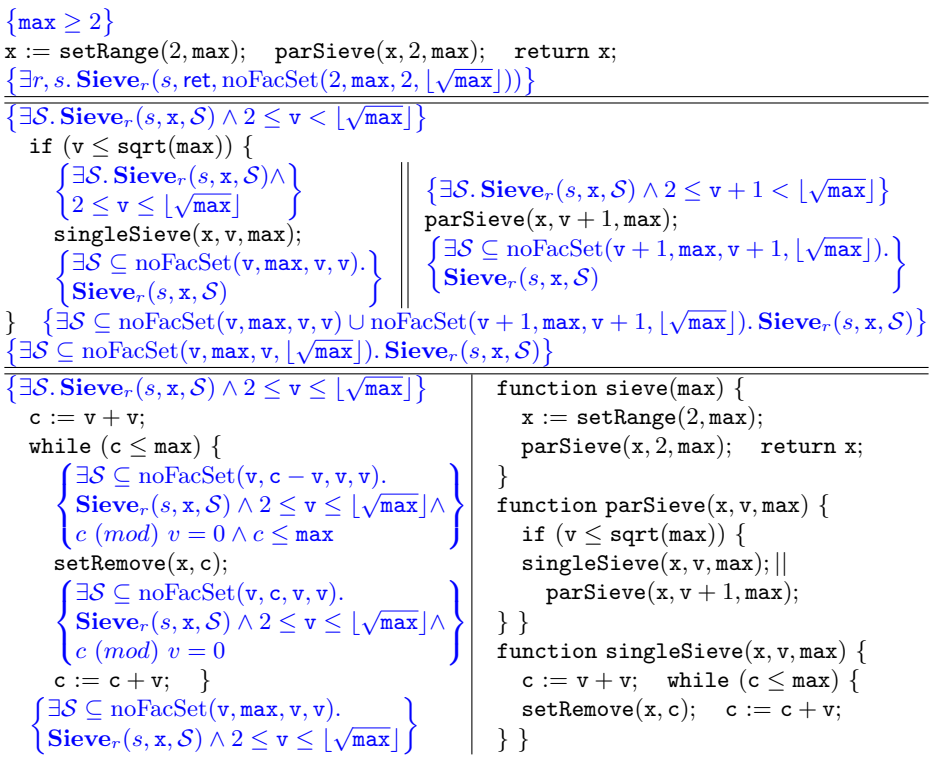


Fig. 9. Code and proofs of the sieve of Eratosthenes.

The Sieve of Eratosthenes is an algorithm to generate prime numbers between 2 and a maximum number `max`. Clients can use the set module from §3.1 to implement the sieve of Eratosthenes. The operation `sieve` in Fig. 9 takes a number `max` and returns a set including all primes numbers between 2 and `max`. The operation `singleSieve` deletes all numbers that can be factorised by the value `v`. Another operation `parSieve` calls `singleSieve` and recursively sieves the remained interval between $\mathtt{v}+1$ and `max` in parallel. The correctness of this algorithm bases on the fact that if a number cannot be factorised by any numbers smaller than its square root, this number is a prime number. The sketch proofs of these three operations are in Fig. 9.

To prove this example, the set module is wrapped by a new **Sieve** region. The unit guard **0** in the new transmission system allows to remove any member from the set. In the proof in Fig. 9, the auxiliary function $\mathrm{noFacSet}(l, u, l', u')$ returns a set of numbers between l and u that cannot be factorised by any numbers between l' and u'. Therefore, if there is no factor between 2 and the square root

$\lfloor\sqrt{u}\rfloor$, the returned set includes all prime numbers between l and u.

$$\begin{array}{rl} \mathbf{0} : \forall v.\, \mathcal{S} \rightsquigarrow \mathcal{S} \setminus \{v\} \quad I(\mathbf{Sieve}_r(s,x,\mathcal{S})) \stackrel{\text{def}}{=} \mathsf{Set}(s,x,\mathcal{S}) \\ \text{noFacSet}(l,r,l',u') & = \{l \le v \le u | \nexists l' < v' < u'.\, v' \neq v \wedge (v \bmod v') = 0\} \\ \text{noFacSet}(l,r,2,\lfloor\sqrt{u}\rfloor) & \Longleftrightarrow \{l \le v \le u | \mathtt{isPrime}(v)\} \end{array}$$

B Key-value Specification

B.1 Proofs of Key-value Specification

Given the key-value specification and definition in §3.2, we prove the key-value specification, shown in Fig. 10. We use make atomic rule to declare the atomic update, and the update atomic rule to update the abstract state of key.

B.2 Rebuild Map Specification

Predicates $\mathsf{NewMap}((r,s'),x,\mathcal{M})$ shows how to re-build a map specification by sperate key-value specification.

$$\begin{array}{l} I(\mathbf{NewMap}_r(s,x,\mathcal{M})) \stackrel{\text{def}}{=} \exists \mathcal{K}.\, \mathsf{Collect}(s,x,\mathcal{K}) \wedge \text{dom}(\mathcal{M}) \subseteq \mathcal{K} \\ \qquad\qquad \circledast_{k \in \mathcal{K}} \begin{pmatrix} \exists v.\, \mathsf{Key}(s,x,k,v) \wedge \\ \underline{\textbf{if}}\ v \neq 0\ \underline{\textbf{then}}\ (k,v) \in \mathcal{M}\ \underline{\textbf{else}}\ k \notin \text{dom}(\mathcal{M}) \end{pmatrix} \\ \mathsf{NewMap}((r,s'),x,\mathcal{M}) \stackrel{\text{def}}{=} \mathbf{NewMap}_r(s',x,\mathcal{M}) * [\text{NMAP}]_r \end{array}$$

C Restrained Key-value Specification

[4] has introduced separate key-value specification of a concurrent map, shown in Fig. 12. They describe a key by two predicates in and out that are tagged by an extra type $t = \{\mathtt{def}, \mathtt{ins}, \mathtt{rem}\}$ and a fractional permission π. The fractional permission π should be smaller or equal to 1. If a predicate is tagged by `def`, one must hold a full permission, i.e. $\pi = 1$, to do insertion or removal. If it is `rem`, one can only do removal but in partial permission, while `ins` for insertion. Therefore, their specifications of `search`, `insert` and `remove` depend on the extra type of the prediacte in pre-condition, namely `def`, `ins` and `rem`. The following specifications are a simplifier version of the original from [4], also we slightly change the behaviour of insertion for comparison. Despite specification, we also show important core axioms from [4].

$\{\mathsf{in}_{\mathsf{def}}(\mathtt{x},\mathtt{k},v)_1\}$
abstract
  // quantify a, s
  $\{\mathbf{inout}_r(s,\mathtt{x},\mathtt{k},v) * [\text{DEF}(1)]_r\}$
  use
    $\forall v.$
    $\langle \mathsf{Key}(s,\mathtt{x},\mathtt{k},v) \rangle$
    put(x, k, v);
    $\langle \mathsf{Key}(s,\mathtt{x},\mathtt{k},\mathtt{v}) \rangle$
  $\{\mathbf{inout}_r(s,\mathtt{x},\mathtt{k},\mathtt{v}) * [\text{DEF}(1)]_r \wedge \mathsf{ret} = v\}$
$\{\mathsf{in}_{\mathsf{def}}(\mathtt{x},\mathtt{k},\mathtt{v})_1 \wedge \mathsf{ret} = v\}$

Fig. 11. An example of proving specification from [4]

$\{\texttt{true}\}$ `makeMap();` $\{\exists s.\, \mathsf{Map}(s, \mathsf{ret}, \emptyset)\}$ // view shift $\{\exists s.\, \mathsf{Collect}(s, \mathsf{ret}, \emptyset)\}$

$\mathbb{\forall} v.$
$\langle \mathsf{Key}(s, \mathtt{x}, \mathtt{k}, v) \wedge \mathtt{k} \neq 0 \rangle$
abstract; substitute $s = (r, s')$
$\quad \mathbb{\forall} v.$
$\quad \left\langle \exists \mathcal{M}.\, \mathbf{KVMap}_r(s', \mathtt{x}, \mathcal{M}) * [\mathrm{K}(\mathtt{k})]_r \wedge \underline{\mathbf{if}}\ v = 0\ \underline{\mathbf{then}}\ k \notin \mathrm{dom}(\mathcal{M})\ \underline{\mathbf{else}}\ (\mathtt{k}, v) \in \mathcal{M} \right\rangle$
$\quad$ atomic exists
$\quad\quad \langle \mathbf{KVMap}_r(s', \mathtt{x}, \mathcal{M}) * [\mathrm{K}(\mathtt{k})]_r \wedge \underline{\mathbf{if}}\ v = 0\ \underline{\mathbf{then}}\ k \notin \mathrm{dom}(\mathcal{M})\ \underline{\mathbf{else}}\ (\mathtt{k}, v) \in \mathcal{M} \rangle$
$\quad\quad$ use atomic
$\quad\quad\quad \mathbb{\forall} \mathcal{M}.$
$\quad\quad\quad \langle \mathsf{Map}(s', \mathtt{x}, \mathcal{M}) * [\mathrm{K}(\mathtt{k})]_r \wedge \underline{\mathbf{if}}\ v = 0\ \underline{\mathbf{then}}\ k \notin \mathrm{dom}(\mathcal{M})\ \underline{\mathbf{else}}\ (\mathtt{k}, v) \in \mathcal{M} \rangle$
$\quad\quad\quad$ `get(x, k)`
$\quad\quad\quad \left\langle \mathsf{Map}(s', \mathtt{x}, \mathcal{M}) * [\mathrm{K}(\mathtt{k})]_r \wedge \underline{\mathbf{if}}\ v = 0\ \underline{\mathbf{then}}\ k \notin \mathrm{dom}(\mathcal{M})\ \underline{\mathbf{else}}\ (\mathtt{k}, v) \in \mathcal{M} \wedge \mathsf{ret} = v \right\rangle$
$\quad\quad \left\langle \mathbf{KVMap}_r(s', \mathtt{x}, \mathcal{M}) * [\mathrm{K}(\mathtt{k})]_r \wedge \underline{\mathbf{if}}\ v = 0\ \underline{\mathbf{then}}\ k \notin \mathrm{dom}(\mathcal{M})\ \underline{\mathbf{else}}\ (\mathtt{k}, v) \in \mathcal{M} \wedge \mathsf{ret} = v \right\rangle$
$\quad \left\langle \exists \mathcal{M}.\, \mathbf{KVMap}_r(s', \mathtt{x}, \mathcal{M}) * [\mathrm{K}(\mathtt{k})]_r \wedge \underline{\mathbf{if}}\ v = 0\ \underline{\mathbf{then}}\ k \notin \mathrm{dom}(\mathcal{M})\ \underline{\mathbf{else}}\ (\mathtt{k}, v) \in \mathcal{M} \wedge \mathsf{ret} = v \right\rangle$
$\langle \mathsf{Key}(s, \mathtt{x}, \mathtt{k}, v) \wedge \mathsf{ret} = v \rangle$

$\mathbb{\forall} v.$
$\langle \mathsf{Key}(s, \mathtt{x}, \mathtt{k}, v) \wedge \mathtt{k} \neq 0 \rangle$
abstract; substitute $s = (r, s')$
$\quad \mathbb{\forall} v.$
$\quad \left\langle \exists \mathcal{M}.\, \mathbf{KVMap}_r(s', \mathtt{x}, \mathcal{M}) * [\mathrm{K}(\mathtt{k})]_r \wedge \underline{\mathbf{if}}\ v = 0\ \underline{\mathbf{then}}\ k \notin \mathrm{dom}(\mathcal{M})\ \underline{\mathbf{else}}\ (\mathtt{k}, v) \in \mathcal{M} \right\rangle$
$\quad$ atomic exists
$\quad\quad \langle \mathbf{KVMap}_r(s', \mathtt{x}, \mathcal{M}) * [\mathrm{K}(\mathtt{k})]_r \wedge \underline{\mathbf{if}}\ v = 0\ \underline{\mathbf{then}}\ k \notin \mathrm{dom}(\mathcal{M})\ \underline{\mathbf{else}}\ (\mathtt{k}, v) \in \mathcal{M} \rangle$
$\quad\quad$ use atomic
$\quad\quad\quad \mathbb{\forall} \mathcal{M}.$
$\quad\quad\quad \langle \mathsf{Map}(s', \mathtt{x}, \mathcal{M}) * [\mathrm{K}(\mathtt{k})]_r \wedge \underline{\mathbf{if}}\ v = 0\ \underline{\mathbf{then}}\ k \notin \mathrm{dom}(\mathcal{M})\ \underline{\mathbf{else}}\ (\mathtt{k}, v) \in \mathcal{M} \rangle$
$\quad\quad\quad$ `put(x, k, v)`
$\quad\quad\quad \left\langle \mathsf{Map}(s', \mathtt{x}, \mathcal{M}[\mathtt{k} \mapsto \mathtt{v}]) * [\mathrm{K}(\mathtt{k})]_r \wedge \underline{\mathbf{if}}\ v = 0\ \underline{\mathbf{then}}\ k \notin \mathrm{dom}(\mathcal{M})\ \underline{\mathbf{else}}\ (\mathtt{k}, v) \in \mathcal{M} \wedge \mathsf{ret} = v \right\rangle$
$\quad\quad \left\langle \mathbf{KVMap}_r(s', \mathtt{x}, \mathcal{M}[\mathtt{k} \mapsto \mathtt{v}]) * [\mathrm{K}(\mathtt{k})]_r \wedge \underline{\mathbf{if}}\ v = 0\ \underline{\mathbf{then}}\ k \notin \mathrm{dom}(\mathcal{M})\ \underline{\mathbf{else}}\ (\mathtt{k}, v) \in \mathcal{M} \wedge \mathsf{ret} = v \right\rangle$
$\quad \left\langle \exists \mathcal{M}.\, \mathbf{KVMap}_r(s', \mathtt{x}, \mathcal{M}[\mathtt{k} \mapsto \mathtt{v}]) * [\mathrm{K}(\mathtt{k})]_r \wedge \underline{\mathbf{if}}\ v = 0\ \underline{\mathbf{then}}\ k \notin \mathrm{dom}(\mathcal{M})\ \underline{\mathbf{else}}\ (\mathtt{k}, v) \in \mathcal{M} \wedge \mathsf{ret} = v \right\rangle$
$\langle \mathsf{Key}(s, \mathtt{x}, \mathtt{k}, \mathtt{v}) \wedge \mathsf{ret} = v \rangle$

$\mathbb{\forall} v.$
$\langle \mathsf{Key}(s, \mathtt{x}, \mathtt{k}, v) \wedge \mathtt{k} \neq 0 \rangle$
abstract; substitute $s = (r, s')$
$\quad \mathbb{\forall} v.$
$\quad \left\langle \exists \mathcal{M}.\, \mathbf{KVMap}_r(s', \mathtt{x}, \mathcal{M}) * [\mathrm{K}(\mathtt{k})]_r \wedge \underline{\mathbf{if}}\ v = 0\ \underline{\mathbf{then}}\ k \notin \mathrm{dom}(\mathcal{M})\ \underline{\mathbf{else}}\ (\mathtt{k}, v) \in \mathcal{M} \right\rangle$
$\quad$ atomic exists
$\quad\quad \langle \mathbf{KVMap}_r(s', \mathtt{x}, \mathcal{M}) * [\mathrm{K}(\mathtt{k})]_r \wedge \underline{\mathbf{if}}\ v = 0\ \underline{\mathbf{then}}\ k \notin \mathrm{dom}(\mathcal{M})\ \underline{\mathbf{else}}\ (\mathtt{k}, v) \in \mathcal{M} \rangle$
$\quad\quad$ use atomic
$\quad\quad\quad \mathbb{\forall} \mathcal{M}.$
$\quad\quad\quad \langle \mathsf{Map}(s', \mathtt{x}, \mathcal{M}) * [\mathrm{K}(\mathtt{k})]_r \wedge \underline{\mathbf{if}}\ v = 0\ \underline{\mathbf{then}}\ k \notin \mathrm{dom}(\mathcal{M})\ \underline{\mathbf{else}}\ (\mathtt{k}, v) \in \mathcal{M} \rangle$
$\quad\quad\quad$ `remove(x, k)`
$\quad\quad\quad \left\langle \mathsf{Map}(s', \mathtt{x}, \mathcal{M} \setminus \{\mathtt{k}\}) * [\mathrm{K}(\mathtt{k})]_r \wedge \underline{\mathbf{if}}\ v = 0\ \underline{\mathbf{then}}\ k \notin \mathrm{dom}(\mathcal{M})\ \underline{\mathbf{else}}\ (\mathtt{k}, v) \in \mathcal{M} \wedge \mathsf{ret} = v \right\rangle$
$\quad\quad \left\langle \mathbf{KVMap}_r(s', \mathtt{x}, \mathcal{M} \setminus \{\mathtt{k}\}) * [\mathrm{K}(\mathtt{k})]_r \wedge \underline{\mathbf{if}}\ v = 0\ \underline{\mathbf{then}}\ k \notin \mathrm{dom}(\mathcal{M})\ \underline{\mathbf{else}}\ (\mathtt{k}, v) \in \mathcal{M} \wedge \mathsf{ret} = v \right\rangle$
$\quad \left\langle \exists \mathcal{M}.\, \mathbf{KVMap}_r(s', \mathtt{x}, \mathcal{M} \setminus \{\mathtt{k}\}) * [\mathrm{K}(\mathtt{k})]_r \wedge \underline{\mathbf{if}}\ v = 0\ \underline{\mathbf{then}}\ k \notin \mathrm{dom}(\mathcal{M})\ \underline{\mathbf{else}}\ (\mathtt{k}, v) \in \mathcal{M} \wedge \mathsf{ret} = v \right\rangle$
$\langle \mathsf{Key}(s, \mathtt{x}, \mathtt{k}, 0) \wedge \mathsf{ret} = v \rangle$

Fig. 10. Proofs of key-value specification

$$
\begin{array}{rll}
\{\mathsf{in}_{\mathtt{def}}(\mathtt{x},\mathtt{k},v)_\pi\} & \mathtt{get}(\mathtt{x},\mathtt{k}) & \{\mathsf{in}_{\mathtt{def}}(\mathtt{x},\mathtt{k},v)_\pi \wedge \mathsf{ret}=v\} \\
\{\mathsf{out}_{\mathtt{def}}(\mathtt{x},\mathtt{k})_\pi\} & \mathtt{get}(\mathtt{x},\mathtt{k}) & \{\mathsf{out}_{\mathtt{def}}(\mathtt{x},\mathtt{k})_\pi \wedge \mathsf{ret}=0\} \\
\{\mathsf{in}_{\mathtt{ins}}(\mathtt{x},\mathtt{k},\mathcal{S})_\pi\} & \mathtt{get}(\mathtt{x},\mathtt{k}) & \{\mathsf{in}_{\mathtt{ins}}(\mathtt{x},\mathtt{k},\mathcal{S})_\pi \wedge \mathsf{ret}\in\mathcal{S}\} \\
\{\mathsf{out}_{\mathtt{ins}}(\mathtt{x},\mathtt{k},\mathcal{S})_\pi\} & \mathtt{get}(\mathtt{x},\mathtt{k}) & \left\{\begin{array}{l}(\mathsf{out}_{\mathtt{ins}}(\mathtt{x},\mathtt{k},\mathcal{S})_\pi \wedge \mathsf{ret}=0)\vee\\ (\mathsf{in}_{\mathtt{ins}}(\mathtt{x},\mathtt{k},\mathcal{S})_\pi \wedge \mathsf{ret}\in\mathcal{S})\end{array}\right\} \\
\{\mathsf{in}_{\mathtt{rem}}(\mathtt{x},\mathtt{k},v)_\pi\} & \mathtt{get}(\mathtt{x},\mathtt{k}) & \left\{\begin{array}{l}(\mathsf{in}_{\mathtt{rem}}(\mathtt{x},\mathtt{k},v)_\pi \wedge \mathsf{ret}=v)\vee\\ (\mathsf{out}_{\mathtt{rem}}(\mathtt{x},\mathtt{k}) \wedge \mathsf{ret}=0)\end{array}\right\} \\
\{\mathsf{out}_{\mathtt{rem}}(\mathtt{x},\mathtt{k})_\pi\} & \mathtt{get}(\mathtt{x},\mathtt{k}) & \{\mathsf{out}_{\mathtt{rem}}(\mathtt{x},\mathtt{k})_\pi \wedge \mathsf{ret}=0\} \\
\{\mathsf{in}_{\mathtt{def}}(\mathtt{x},\mathtt{k},v)_1\} & \mathtt{put}(\mathtt{x},\mathtt{k},\mathtt{v}) & \{\mathsf{in}_{\mathtt{def}}(\mathtt{x},\mathtt{k},\mathtt{v})_1 \wedge \mathsf{ret}=v\} \\
\{\mathsf{out}_{\mathtt{def}}(\mathtt{x},\mathtt{k})_1\} & \mathtt{put}(\mathtt{x},\mathtt{k},\mathtt{v}) & \{\mathsf{in}_{\mathtt{def}}(\mathtt{x},\mathtt{k},\mathtt{v})_1 \wedge \mathsf{ret}=0\} \\
\{\mathsf{in}_{\mathtt{ins}}(\mathtt{x},\mathtt{k},\mathcal{S})_\pi \wedge \mathtt{v}\in\mathcal{S}\} & \mathtt{put}(\mathtt{x},\mathtt{k},\mathtt{v}) & \{\mathsf{in}_{\mathtt{ins}}(\mathtt{x},\mathtt{k},\mathcal{S})_\pi \wedge \mathsf{ret}\in\mathcal{S}\} \\
\{\mathsf{out}_{\mathtt{ins}}(\mathtt{x},\mathtt{k},\mathcal{S})_\pi \wedge \mathtt{v}\in\mathcal{S}\} & \mathtt{put}(\mathtt{x},\mathtt{k},\mathtt{v}) & \{\mathsf{in}_{\mathtt{ins}}(\mathtt{x},\mathtt{k},\mathcal{S})_\pi \wedge \mathsf{ret}=0\} \\
\{\mathsf{unk}(\mathtt{x},\mathtt{k})\} & \mathtt{put}(\mathtt{x},\mathtt{k},\mathtt{v}) & \{\mathsf{unk}(\mathtt{x},\mathtt{k})\} \\
\{\mathsf{in}_{\mathtt{def}}(\mathtt{x},\mathtt{k},v)_1\} & \mathtt{remove}(\mathtt{x},\mathtt{k}) & \{\mathsf{out}_{\mathtt{def}}(\mathtt{x},\mathtt{k})_1 \wedge \mathsf{ret}=v\} \\
\{\mathsf{out}_{\mathtt{def}}(\mathtt{x},\mathtt{k})_1\} & \mathtt{remove}(\mathtt{x},\mathtt{k}) & \{\mathsf{out}_{\mathtt{def}}(\mathtt{x},\mathtt{k})_1 \wedge \mathsf{ret}=0\} \\
\{\mathsf{in}_{\mathtt{rem}}(\mathtt{x},\mathtt{k},v)_\pi\} & \mathtt{remove}(\mathtt{x},\mathtt{k}) & \{\mathsf{out}_{\mathtt{rem}}(\mathtt{x},\mathtt{k})_\pi \wedge \mathsf{ret}=v\} \\
\{\mathsf{out}_{\mathtt{rem}}(\mathtt{x},\mathtt{k})_\pi\} & \mathtt{remove}(\mathtt{x},\mathtt{k}) & \{\mathsf{out}_{\mathtt{rem}}(\mathtt{x},\mathtt{k})_\pi \wedge \mathsf{ret}=0\} \\
\{\mathsf{unk}(\mathtt{x},\mathtt{k})\} & \mathtt{remove}(\mathtt{x},\mathtt{k}) & \{\mathsf{unk}(\mathtt{x},\mathtt{k})\}
\end{array}
$$

$$
\begin{array}{rcl}
X_{\pi_1} * X_{\pi_2} & \iff & X_{\pi_1+\pi_2} \\
 & & \text{where } X \text{ is any predicate, and } \pi_1+\pi_2 \le 1 \\
\exists\mathcal{S}.\, v\in\mathcal{S} \wedge \mathsf{in}_{\mathtt{def}}(x,k,v)_1 & \iff & \mathsf{in}_{\mathtt{ins}}(x,k,\mathcal{S})_1 \\
 & & \iff \mathsf{in}_{\mathtt{rem}}(x,k,v)_1 \iff \mathsf{unk}(x,k) \\
\exists\mathcal{S}.\, \mathsf{out}_{\mathtt{def}}(x,k)_1 & \iff & \mathsf{out}_{\mathtt{ins}}(x,k,\mathcal{S})_1 \iff \mathsf{out}_{\mathtt{rem}}(x,k)_1 \\
\mathsf{in}_{\mathtt{ins}}(x,k,\mathcal{S})_{\pi_1} * \mathsf{out}_{\mathtt{ins}}(x,k,\mathcal{S})_{\pi_2} & \implies & \mathsf{in}_{\mathtt{ins}}(x,k,\mathcal{S})_{\pi_1+\pi_2} \\
\mathsf{in}_{\mathtt{rem}}(x,k,v)_{\pi_1} * \mathsf{out}_{\mathtt{rem}}(x,k)_{\pi_2} & \implies & \mathsf{out}_{\mathtt{rem}}(x,k)_{\pi_1+\pi_2}
\end{array}
$$

$$
\begin{array}{rcl}
\mathsf{in}_{\mathtt{def}}(x,k,v)_\pi & \stackrel{\text{def}}{=} & \exists r,s.\, \mathbf{inout}_r(s,x,k,v) * [\textsc{Def}(\pi)]_r \wedge v\neq 0 \\
\mathsf{in}_{\mathtt{ins}}(x,k,\mathcal{S})_\pi & \stackrel{\text{def}}{=} & \exists r,s.\, \mathbf{inout}_r(s,x,k,v) * [\textsc{Spt}(\pi)]_r * [\textsc{Ins}(\pi,\mathcal{S})]_r \wedge v\in\mathcal{S} \\
\mathsf{in}_{\mathtt{rem}}(x,k,v)_\pi & \stackrel{\text{def}}{=} & \exists r,s,v'.\, \mathbf{inout}_r(s,x,k,v') * \\
 & & [\textsc{Spt}(\pi)]_r * [\textsc{Rem}(\pi,v)]_r \wedge v'\in\{v,0\} \\
\mathsf{out}_{\mathtt{def}}(x,k)_\pi & \stackrel{\text{def}}{=} & \exists r,s.\, \mathbf{inout}_r(x,k,0) * [\textsc{Def}(\pi)]_r \\
\mathsf{out}_{\mathtt{ins}}(x,k,S)_\pi & \stackrel{\text{def}}{=} & \exists r,s,v'.\, \mathbf{inout}_r(s,x,k,v') * \\
 & & [\textsc{Spt}(\pi)]_r * [\textsc{Ins}(\pi,\mathcal{S})]_r \wedge v'\in\mathcal{S}\uplus\{0\} \\
\mathsf{out}_{\mathtt{rem}}(x,k)_\pi & \stackrel{\text{def}}{=} & \exists v,r,s.\, \mathbf{inout}_r(s,x,k,0) * [\textsc{Spt}(\pi)]_r * [\textsc{Rem}(\pi,v)]_r
\end{array}
$$

$$
\begin{array}{rcl}
\textsc{Def}(1) & = & \textsc{Ins}(1,\mathcal{S}) \bullet \textsc{Spt}(1) = \textsc{Rem}(1,v) \bullet \textsc{Spt}(1) \\
\textsc{Def}(\pi_1+\pi_2) & = & \textsc{Def}(\pi_1)] \bullet \textsc{Def}(\pi_2) \\
\textsc{Spt}(\pi_1+\pi_2) & = & \textsc{Spt}(\pi_1)] \bullet \textsc{Spt}(\pi_2) \\
\textsc{Ins}(\pi_1+\pi_2,\mathcal{S}) & = & \textsc{Ins}(\pi_1,\mathcal{S}) \bullet \textsc{Ins}(\pi_2,\mathcal{S}) \\
\textsc{Rem}(\pi_1+\pi_2,v) & = & \textsc{Rem}(\pi_1,v) \bullet \textsc{Rem}(\pi_2,v)
\end{array}
$$

$$
\textsc{Def}(1) : \forall v.\, v \rightsquigarrow v' \qquad \textsc{Ins}(\pi,\mathcal{S}) : \forall v,v'\in\mathcal{S}.\, v\rightsquigarrow v' \qquad \textsc{Rem}(1,v) : v \rightsquigarrow 0
$$

$$
I(\mathbf{inout}_r(s,x,k,v)) \stackrel{\text{def}}{=} \mathsf{Key}(s,x,k,v)
$$

Fig. 12. The specification for a concurrent map from [4] and its implementation

To implement their restrained specification of a concurrent map, we use guard algebra to simulate those axioms. All in/out predicates are defined as the same type region but with different guards. These guards encode the fractional permission π and allowed actions. Some guards also record ghost states for the purpose of reasoning, e.g. $\mathcal{S}$ in $[\textsc{Ins}(\pi, \mathcal{S})]_r$.

We also need to show all the specifications are provable under the definition. They are very trivial, Fig. 11 show one example below.

D Producer and Consumer Proof by Restrained Key-value Specification

$$\{\texttt{True}\}$$
$$\texttt{x} := \texttt{makeMap}();$$
$$\left\{\circledast_{k\in\mathbb{N}} \mathsf{out}_{\mathtt{def}}(s,\texttt{x},k)_1\right\} \text{ // quantify logical identifier } s$$
$$\text{// frame off other keys, and weaken the assertion}$$
$$\left\{\circledast_{k<100} \mathsf{unk}(s,\texttt{x},k)_1\right\} \text{ // } \mathsf{unk}(s,\texttt{x},k)_1 \stackrel{\text{def}}{=} \mathsf{in}_{\mathtt{def}}(\texttt{x},k,-)_1 \vee \mathsf{out}_{\mathtt{def}}(\texttt{x},k)_1$$

$$\left\{\circledast_{k<100} \mathsf{unk}(s,\texttt{x},k)_{\frac{1}{2}}\right\} \quad \texttt{p} := 0$$
$$\texttt{while } (\texttt{p} < 100) \{ \quad \texttt{v} := \texttt{randomValue}();$$
$$\quad \texttt{insert}(\texttt{x},\texttt{p},\texttt{v}); \quad \texttt{p} := \texttt{p} + 1;$$
$$\} \quad \left\{\circledast_{k<100} \mathsf{unk}(s,\texttt{x},k)_{\frac{1}{2}}\right\}$$

$$\parallel$$

$$\left\{\circledast_{k<100} \mathsf{unk}(s,\texttt{x},k)_{\frac{1}{2}}\right\} \quad \texttt{c} := 0$$
$$\texttt{while } (\texttt{c} < 100) \{$$
$$\quad \texttt{do } \{ \texttt{t} := \texttt{remove}(\texttt{x},\texttt{p}); \} \texttt{ while}(\texttt{t} \neq 0)$$
$$\quad \texttt{c} := \texttt{c} + 1;$$
$$\} \left\{\circledast_{k<100} \mathsf{unk}(s,\texttt{x},k)_{\frac{1}{2}}\right\}$$

$$\text{// frame back other keys}$$
$$\left\{\circledast_{k<100} \mathsf{unk}(s,\texttt{x},k)_1 * \circledast_{k\geq 100} \mathsf{out}_{\mathtt{def}}(s,\texttt{x},k)_1\right\}$$

Fig. 13. Proof of producer-consumer by specification from [4]

With the specification given by [4], they are also able to prove the single producer-consumer example we have proven in §3.3. However, because of stabilisability problem, two threads can only hold unk predicates instead of a more precise one. At the end, the keys between 1 and 99 become unknown whereas they are actually not in the concurrent map. Figure 13 is the proof.

E Skiplsit

E.1 Auxiliary Predicates

The definition of *node*, *index-node* and *head-node* modules are following.

$$\begin{aligned}
\mathsf{node}(x,k,v,n) &\stackrel{\text{def}}{=} x.\texttt{key} \mapsto k * x.\texttt{value} \mapsto v * x.\texttt{next} \mapsto n \\
\mathsf{marker}(x,n) &\stackrel{\text{def}}{=} \mathsf{node}(x,-,x,n) \\
\mathsf{head}(x,l,p,d,n) &\stackrel{\text{def}}{=} \mathsf{index}(,x,p,d,n) * x.\texttt{level} \mapsto l \\
\mathsf{index}(x,p,d,n) &\stackrel{\text{def}}{=} x.\texttt{node} \mapsto p * x.\texttt{down} \mapsto d * x.\texttt{next} \mapsto n
\end{aligned}$$

We also define right-reachability and down-reachability predicates. For the simplicity, these predicates do not explicitly distinguish the type of x, yet one can easily define all different reachability predicates based on different type, and there is no difference with respect of all the proofs.

$$\begin{aligned} x \rightsquigarrow n &\stackrel{\text{def}}{=} \mathsf{index}(x,-,-,n) \vee \mathsf{head}(x,-,-,n) \vee \\ &\quad\ \mathsf{node}(x,-,-,n) \vee \mathsf{marker}(x,n) \\ x \rightsquigarrow^* n &\stackrel{\text{def}}{=} x \rightsquigarrow y \vee (\exists m.\, x \rightsquigarrow m \wedge m \rightsquigarrow^* n) \\ x \rightsquigarrow^* \mathcal{N} &\stackrel{\text{def}}{=} \exists n \in \mathcal{N}.\, x \rightsquigarrow^* n \\ x \searrow d &\stackrel{\text{def}}{=} \mathsf{index}(x,-,d,-) \vee \mathsf{head}(x,-,d,-) \\ x \searrow^* d &\stackrel{\text{def}}{=} x \searrow d \vee (\exists m.\, x \searrow m \wedge m \searrow^* d) \end{aligned}$$

E.2 Code and Specification of Auxiliary Operations

To prove `insert` and `remove`, we firstly specify auxiliary methods, shown in Fig. 14. Methods `link` and `unlink` are for index-nodes. If a node p to which an index-node `x` points is dead, these two methods will immediately return `false`, otherwise these two methods will act as `CAS`. Method `helpDelete` tries to append a marker after a node `x` or unlink a node `x`, but it does not guarantee to make progress because of concurrency. To increase performance, method `tryReduceLevel` deletes empty index list, but it still preserves the abstract state of a skiplist. Code block `buildIndexChain` builds a chain of index-nodes that point to a same node, meanwhile `insertIndexChain` inserts the chain. Each index-node is linked in by the method `link`. For clients, both `buildIndexChain` and `insertIndexChain` are opaque, so they preserve the abstract state of a skiplist.

For those methods that are easy to prove, we only give their codes (Fig. 15). All codes inlcluding those in §4 and §E.3 are translated from [17]. However, all those operations are opaque for clients.

E.3 Proof of Skiplist

Given the invariant of a skiplist in §4, we have already proven main part of `put`, and the rest are in this sections, namely `findPredecessor`, `remove`, `buildIndexChain` and `insertIndexChain`. The `get` of a skiplist is similar with `remove`, so we choose to leave it.

The method `findPredecessor` traverses all index lists from top to bottom. It also clean dead index-nodes. Once finding a dead index-node, method tries to unlink this dead index-node and restart `findPredecessor`.

In the proof of `findPredecessor`, we quantify $\mathcal{N}_l$ and $\mathcal{B}_l$ for all levels. To recall, the state $\mathcal{P}$ is a list of tuples, and each tuple have two elements. The first element is an address of a live or a dead node. The second element is a list of addresses indicating a list of index-nodes. Moreover, $\mathcal{N}$ and $\mathcal{B}$ are the sets of live nodes and dead nodes.

$$\begin{aligned} \mathcal{N}_l &= \{p | \forall i.\, p = \mathcal{P} \downarrow_2 (i)(l) \wedge \exists p.\, \mathsf{index}(i,p,-,-) \wedge p \in \mathcal{N}\} \\ \mathcal{B}_l &= \{p | \forall i.\, p = \mathcal{P} \downarrow_2 (i)(l) \wedge \exists p.\, \mathsf{index}(i,p,-,-) \wedge p \in \mathcal{B}\} \end{aligned}$$

Fig. 14. Specifications for auxiliary methods

```
function CASNext(x, cmp, val) {
  return (CAS(x.next, cmp, val));
}

function CASValue(x, cmp, val) {
  return (CAS(x.value, cmp, val));
}

function CASHead(x, cmp, val) {
  return (CAS(x.head, cmp, val));
}

function CASRight(x, cmp, val) {
  return (CAS(x.right, cmp, val));
}

function AppendMarker(x, next) {
  m := newMarker(next);
  return (CAS(x.next, next, m));
}

function link(x, cmp, val) {
  [val.right] := cmp;
  tn := [x.node];   tv := [tn.value];
  if (tv = 0) {return false;}
  return (CASRight(x, cmp, val));
}

function unlink(x, cmp) {
  tn := [x.node];   tv := [tn.value];
  if (tv ≠ 0) {return false;}
  return (CASRight(x, cmp, cmpright));
}
```

```
function helpDelete(b, n, f) {
  nn := [n.next];   nb := [b.next];
  if (f = nn&&n = nb) {
    marker := isMarker(f);
    if (f = 0 || marker = false) {
      m := newMarker(f);
      CASNext(n, f, m);
    } else {   fn := [f.next];
      CASNext(b, n, fn);   }
} }

function tryReduceLevel(x) {
  h := [x.head];
  if (h.level > 3) {   d := [h.down];
    if (d ≠ 0) {   e := [d.down];
      er := [e.right];
      dr := [d.right];   hr := [h.right];
      if (e ≠ 0 && er = 0
          && dr = 0&& hr = 0) {
        b := CASHead(x, h, d);
        if (b = true && h.right ≠ 0) {
          CASHead(x, h, d);
} } } } }

function compare(x, y) {
  if (x > y) {   return 1;   }
  else if (x = y) {   return 0;   }
  else {   return −1;   }
}

function isMarker(x) {
  tv := [x.value];
  if (tv = x) {   return true;   }
  else {   return false;   }
}
```

Fig. 15. Code of auxiliary methods.

Also, the $\mathsf{wkIdxReach}(x, y, l)$ asserts x can reach y, or y is a dead index-node.

$$\mathsf{wkIdxReach}(x, y, l) \stackrel{\text{def}}{=} x \rightsquigarrow^* y \vee ((y \rightsquigarrow^* \mathcal{N}_l \uplus \{0\}) \wedge y \in \mathcal{B}_l)$$

The prediacte qrl asserts the reachability relation between index-node $\mathtt{q}$ and $\mathtt{r}$ in level l.

$$\mathsf{qrl}(l) \equiv \mathsf{wkIdxReach}(\mathtt{q}, \mathtt{r}, l) \wedge (\mathtt{q.node} = m \vee \mathtt{k} > \mathtt{q.node.key})$$

There two assertions are not pure assertions, but because of the immutability, they are stable in the proof.

$$x.\mathtt{node} = p \text{ where } \mathsf{index}(x, p, -, -)$$
$$x.\mathtt{node.key} = k \text{ where } \exists p.\, \mathsf{index}(x, p, -, -) * \mathsf{node}(p, k, -, -)$$

$\forall \mathcal{M}, \mathcal{N}, \mathcal{B}.$
$\langle \mathbf{SLMap}_r(\mathtt{x}, \mathcal{M}, \mathcal{N}, \mathcal{B}, m) * [\mathrm{M}]_r \rangle$
$r : \mathcal{M}, \mathcal{N}, \mathcal{B} \rightsquigarrow \mathcal{M}, \mathcal{N}, \mathcal{B} \vdash$
// We assume to frame back $\mathbf{skipList}_r(s', \mathtt{x}, \mathcal{M}, \mathcal{N}, \mathcal{B}, m)$,
// explicitly apply *open* rule and frame off again.
`function findPredecessor(x, k) {`
`  while (true) {  q := [x.head];  r := [q.right];  inner := ture;`
`    // weaken to loop invariant, inner does not affect invariant`
$\{r \Mapsto \blacklozenge * \mathsf{qrl}(l)\}$ `  while (inner = true) {  continue := true;`
`    if (r ≠ 0) {` $\{r \Mapsto \blacklozenge * \mathsf{qrl}(l) \wedge \mathtt{r} \neq 0\}$
`      n := [r.node];  tk := [n.key];`
$\{r \Mapsto \blacklozenge * \mathsf{qrl}(l) * \mathsf{node}(\mathtt{n}, \mathtt{tk}, -, -) \wedge \mathtt{n} = \mathtt{r.node}\}$
`      if (n.value = 0) {`
$\{r \Mapsto \blacklozenge * \mathsf{qrl}(l) * \mathsf{node}(\mathtt{n}, \mathtt{tk}, 0, -) \wedge \mathtt{n} = \mathtt{r.node}\}$
`        lk := unlink(q, r);`
$\left\{\begin{array}{l} r \Mapsto \blacklozenge * \mathsf{qrl}(l) * \mathsf{node}(\mathtt{n}, \mathtt{tk}, 0, -) \wedge \\ \mathtt{n} = \mathtt{r.node} \wedge \underline{\mathbf{if}}\ \mathtt{lk} = \mathtt{true}\ \underline{\mathbf{then}}\ \neg(\mathtt{q} \rightsquigarrow^* \mathtt{r})\ \underline{\mathbf{else}}\ - \end{array}\right\}$
`        if (lk = false) {inner := false; }`
`        else{r := [q.right]; continue := false; }`
$\left\{\begin{array}{l} r \Mapsto \blacklozenge * \mathsf{qrl}(l) * \mathsf{node}(\mathtt{n}, \mathtt{tk}, 0, -) \wedge \mathtt{n} = \mathtt{r.node} \wedge \underline{\mathbf{if}}\ \mathtt{lk} = \mathtt{true} \\ \quad \underline{\mathbf{then}}\ \mathtt{continue} = \mathtt{false} \wedge \mathtt{inner} = \mathtt{true} \\ \quad \underline{\mathbf{else}}\ \mathtt{continue} = \mathtt{true} \wedge \mathtt{inner} = \mathtt{false} \end{array}\right\}$
`      } else {`
$\{r \Mapsto \blacklozenge * \mathsf{qrl}(l) * \mathsf{node}(\mathtt{n}, \mathtt{tk}, -, -) \wedge \mathtt{n} = \mathtt{r.node}\}$
`        c := compare(k, tk);`
`        if (c > 0) {`
$\{r \Mapsto \blacklozenge * \mathsf{qrl}(l) * \mathsf{node}(\mathtt{n}, \mathtt{tk}, -, -) \wedge \mathtt{n} = \mathtt{r.node} \wedge \mathtt{k} > \mathtt{tk}\}$
`          q := r;  r := [q.right];  continue := false;`
$\left\{\begin{array}{l} r \Mapsto \blacklozenge * \mathsf{qrl}(l) * \mathsf{node}(\mathtt{n}, \mathtt{tk}, -, -) \wedge \\ \mathtt{n} = \mathtt{p.node} \wedge \mathtt{k} > \mathtt{tk} \wedge \mathtt{continue} = \mathtt{false} \end{array}\right\}$
`      } } }`
$\left\{\begin{array}{l} r \Mapsto \blacklozenge * \mathsf{qrl}(l) \wedge \underline{\mathbf{if}}\ \mathtt{continue} = \mathtt{true} \wedge \mathtt{inner} = \mathtt{true} \\ \quad \underline{\mathbf{then}}\ (\mathtt{r} = 0 \vee \mathtt{k} \leq \mathtt{r.node.key}) \\ \quad \underline{\mathbf{else}}\ (\mathtt{r.node.value} = 0 \vee \mathtt{k} > \mathtt{r.node.key} \vee \neg(\mathtt{q} \rightsquigarrow^* \mathtt{r})) \end{array}\right\}$
`      if (inner = true && continue = true) {`
$\{r \Mapsto \blacklozenge * \mathsf{qrl}(l) \wedge (\mathtt{r} = 0 \vee \mathtt{k} \leq \mathtt{r.node.key})\}$
$\{r \Mapsto \blacklozenge * \mathsf{qrl}(l)\}$ // Weaken the assertion.
`        d := [q.down];` $\{r \Mapsto \blacklozenge * \mathsf{qrl}(l) \wedge \mathtt{q} \searrow \mathtt{d}\}$
`        if (d = 0) {  n := [q.node]  return n;`
$\{r \Mapsto ((\mathcal{M}, \mathcal{N}, \mathcal{B}), (\mathcal{M}, \mathcal{N}, \mathcal{B})) * (\mathsf{qrl}(l) \wedge \mathtt{q} \searrow 0) \wedge \mathsf{ret} = \mathtt{q.node}\}$
$\{\exists k.\ \mathsf{node}(-, \mathsf{ret}, k, -, -) \wedge (k < \mathtt{k} \vee \mathsf{ret} = m)\}$ // Weakening.
`        } else {  q := d;`
$\left\{\begin{array}{l} \exists q.\ r \Mapsto \blacklozenge * (\mathsf{wkIdxReach}(q, \mathtt{r}, l+1) \wedge q \searrow \mathtt{q}) \wedge \\ (\mathtt{k} > \mathtt{q.node.key} \vee \mathtt{q.node} = m) \end{array}\right\}$
`        r := [d.right];` $\{r \Mapsto \blacklozenge * \mathsf{qrl}(l) \wedge (\mathtt{k} > \mathtt{q.node.key} \vee \mathtt{q.node} = m)\}$
`    } } }` $\{r \Mapsto \blacklozenge * \mathsf{qrl}(l) \wedge \mathtt{inner} = \mathtt{false} \wedge \mathtt{r.node.value} = 0\}$
`  }` // while 'true' $\{\mathtt{false}\}$
`}`
make atomic, substitute $l = \mathtt{q.head.level}$
$\langle \exists k.\ (\mathbf{SLMap}_r(\mathtt{x}, \mathcal{M}, \mathcal{N}, \mathcal{B}, m) \wedge \mathsf{node}(\mathsf{ret}, k, -, -)) \wedge (k < \mathtt{k} \vee \mathsf{ret} = m) \rangle$

The `put` proof below is the fully version of the one in §4.

To recall, the predicate $\mathsf{wkNdReach}(x, y)$ assserts that either x can reach y, or y is a dead node. Since the algorithm works on three nodes `b`, `n` and `f`, we use the predicate bnf to describe their reachability relation. Note that either is node `b` the sentinel node m, or its key smaller then the `k` parameter of the method.

$$\mathsf{wkNdReach}(x, y) \equiv x \rightsquigarrow^* y \vee ((y \rightsquigarrow^* \mathcal{N} \uplus \{0\}) \wedge y \in \mathcal{B})$$
$$\mathsf{bnf} \equiv \mathsf{wkNdReach}(\mathtt{b}, \mathtt{n}) * \mathsf{wkNdReach}(\mathtt{n}, \mathtt{f}) \wedge (\mathtt{b.key} < \mathtt{k} \vee \mathtt{b} = m)$$

Similarly, some assertions are not pure assertions, but because of the immutability, they are stable in the proof.

$$x.\mathtt{key} = k \text{ where } \mathsf{node}(x, k, -, -)$$
$$(x.\mathtt{value} = 0) \equiv \mathsf{node}(x, -, 0, -)$$

$\forall \mathcal{M}. \langle \mathsf{Map}(s, \mathtt{x}, \mathcal{M}) \rangle$

abstract, substuting $s = (r, m)$, and abstract exists rule for $\mathcal{N}$ and $\mathcal{B}$

make atomic

$\langle \mathbf{SLMap}_r(\mathtt{x}, \mathcal{M}, \mathcal{N}, \mathcal{B}, m) * [\mathrm{L}]_r \rangle$

$r : (\mathcal{M}, \mathcal{N}, \mathcal{B}) \rightsquigarrow \underline{\textbf{if}}\ \mathtt{k} \in \mathrm{dom}(\mathcal{M})\ \underline{\textbf{then}}\ (\mathcal{M}[\mathtt{k} \mapsto \mathtt{v}], \mathcal{N}, \mathcal{B})$ $\underline{\textbf{else}}\ (\mathcal{M}[\mathtt{k} \mapsto \mathtt{v}], \mathcal{N} \uplus \{n\}, \mathcal{B})$ $\vdash$

```
function put(x, k, v) {
```

$\{\exists \mathcal{M}, \mathcal{N}, \mathcal{B}.\, r \Mapsto \blacklozenge * \mathbf{SLMap}_r(\mathtt{x}, \mathcal{M}, \mathcal{N}, \mathcal{B}, m)\}$ `outer := true;`

$\left\{ \begin{array}{l} \exists \mathcal{M}, \mathcal{N}, \mathcal{B}.\, \mathbf{SLMap}_r(\mathtt{x}, \mathcal{M}, \mathcal{N}, \mathcal{B}, m) \wedge \mathtt{outer} = \mathtt{true} * \\ \underline{\textbf{if}}\ \mathtt{outer} = \mathtt{true}\ \underline{\textbf{then}}\ a \Mapsto \blacklozenge \\ \quad \underline{\textbf{else}}\ r \Mapsto ((\mathcal{M}, \mathcal{N}, \mathcal{B}), (\mathcal{M}[\mathtt{k} \mapsto \mathtt{v}], \mathcal{N} \uplus \{\mathtt{z}\}, \mathcal{B})) \end{array} \right\}$

```
  while (outer = true) {   inner := true;
```

$\{\exists \mathcal{M}, \mathcal{N}, \mathcal{B}.\, \mathbf{SLMap}_r(\mathtt{x}, \mathcal{M}, \mathcal{N}, \mathcal{B}, m)\}$

atomic exists:

$\forall \mathcal{M}, \mathcal{N}, \mathcal{B}.$
$\langle \mathbf{SLMap}_r(\mathtt{x}, \mathcal{M}, \mathcal{N}, \mathcal{B}, m) \rangle$

```
b := findPredecessor(x, k);
 // Return an predecessor but not direct one
 // frame off SLMap_r(x, M, N, B, m)
```

$\langle \exists k.\, \mathsf{node}(\mathtt{b}, k, -, -) \wedge (k < \mathtt{k} \vee \mathtt{b} = m) \rangle$

$\{r \Mapsto \blacklozenge * \mathtt{b} \rightsquigarrow^* \mathcal{N} \uplus \{0\} \wedge (\mathtt{b.key} < \mathtt{k} \vee \mathtt{b} = m)\}$ `n := [b.next];`

```
 // Weaken to invariant
```

$\{r \Mapsto \blacklozenge * \mathsf{wkNdReach}(\mathtt{b}, \mathtt{n}) \wedge (\mathtt{b.key} < \mathtt{k} \vee \mathtt{b} = m)\}$

$\left\{ \begin{array}{l} \mathsf{wkNdReach}(\mathtt{b}, \mathtt{n}) \wedge (\mathtt{b.key} < \mathtt{k} \vee \mathtt{b} = m) \wedge \mathtt{inner} = \mathtt{true} * \\ \left(\begin{array}{l} \underline{\textbf{if}}\ \mathtt{outer} = \mathtt{true}\ \underline{\textbf{then}}\ r \Mapsto \blacklozenge \\ \quad \underline{\textbf{else}}\ r \Mapsto ((\mathcal{M}, \mathcal{N}, \mathcal{B}), (\mathcal{M}[\mathtt{k} \mapsto \mathtt{v}], \mathcal{N} \uplus \{\mathtt{z}\}, \mathcal{B})) \end{array} \right) \wedge \\ \underline{\textbf{if}}\ \mathtt{inter} = \mathtt{true}\ \underline{\textbf{then}}\ - \\ \quad \underline{\textbf{else}} \left(\begin{array}{l} \mathtt{cas} = \mathtt{false} \vee \mathtt{n} \neq \mathtt{n2} \vee \mathtt{n.value} = 0 \vee \\ \mathtt{b.value} = 0 \vee \mathsf{marker}(\mathtt{n}, -) \end{array} \right) \end{array} \right\}$

```
  while (inner = true) {   continue := true;
```

$\left\{ \begin{array}{l} r \Mapsto \blacklozenge * \mathsf{wkNdReach}(\mathtt{b}, \mathtt{n}) \wedge \\ (\mathtt{b.key} < \mathtt{k} \vee \mathtt{b} = m) \wedge \mathtt{continue} = \mathtt{true} \wedge \mathtt{inner} = \mathtt{true} \end{array} \right\}$

```
    if (n ≠ 0) {
      f := [n.next];   tv := [n.value];   marker := isMarker(n);
      n2 := [b.next]   vb := [b.value];
```

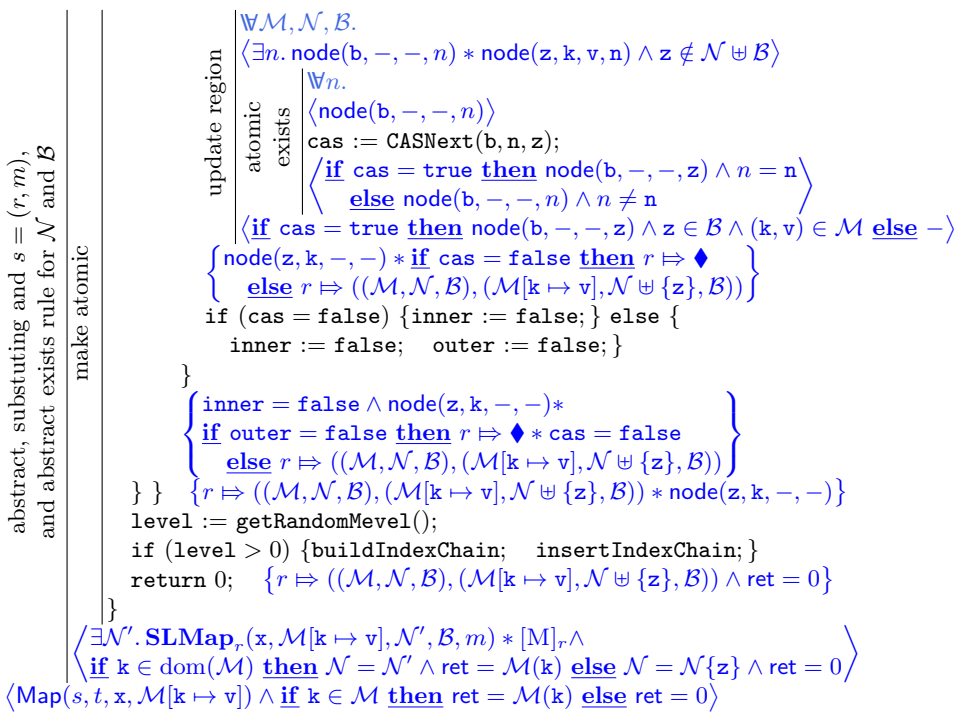
$\{r \Mapsto \blacklozenge * \mathsf{bnf} * \wedge \mathtt{continue} = \mathtt{true} \wedge \mathtt{inner} = \mathtt{true}\}$

```
      if (n ≠ n2) {   inner := false;
```

abstract, substuting $s = (r, m)$, and abstract exists rule for $\mathcal{N}$ and $\mathcal{B}$

make atomic

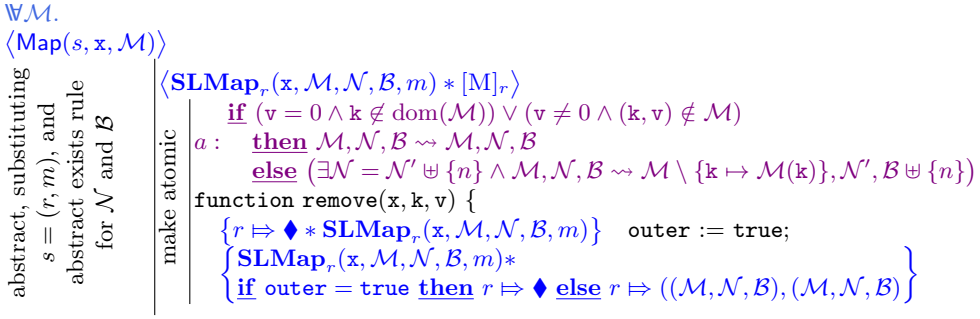
// Inconsistent read, an optimistic prediction.
$\{r \mapsto \blacklozenge * \mathsf{bnf} \wedge \mathtt{n} \neq \mathtt{n2} \wedge \mathtt{continue} = \mathtt{true} \wedge \mathtt{inner} = \mathtt{false}\}$
} else if (tv = 0){ // n is deleted.
$\{r \mapsto \blacklozenge * \mathsf{bnf} \wedge \mathtt{tv} = \mathtt{n.value} = 0 \wedge \mathtt{continue} = \mathtt{true} \wedge \mathtt{inner} = \mathtt{true}\}$
// helpDelete does not guarantee to delete n,
// but optmisitically tries once.
helpDelete(b, n, f); inner := false;
$\{r \mapsto \blacklozenge * \mathsf{bnf} \wedge \mathtt{tv} = \mathtt{n.value} = 0 \wedge \mathtt{continue} = \mathtt{true} \wedge \mathtt{inner} = \mathtt{false}\}$
} else if (vb = 0 || marker = true){ inner := false;
// The node b is deleted.
$\left\{\begin{array}{l} r \mapsto \blacklozenge * \mathsf{bnf} \wedge \left(\mathtt{b.value} = 0 \vee \mathsf{marker}(-, \mathtt{n}, -)\right) \wedge \\ \mathtt{continue} = \mathtt{true} \wedge \mathtt{inner} = \mathtt{false} \end{array}\right\}$
} else {
$\{r \mapsto \blacklozenge * \mathsf{bnf} \wedge \mathsf{node}(\mathtt{n}, -, -, -) \wedge \mathtt{continue} = \mathtt{true} \wedge \mathtt{inner} = \mathtt{true}\}$
tk := [n.key]; c := compare(k, tk);
if (c > 0) {
$\{r \mapsto \blacklozenge * \mathsf{bnf} \wedge \mathtt{n.key} < \mathtt{k} \wedge \mathtt{continue} = \mathtt{true} \wedge \mathtt{inner} = \mathtt{true}\}$
b := n; n := f; continue := false;
$\left\{\begin{array}{l} r \mapsto \blacklozenge * \mathsf{wkNdReach}(\mathtt{b}, \mathtt{n}) \wedge \\ \mathtt{b.key} < \mathtt{k} \wedge \mathtt{continue} = \mathtt{false} \wedge \mathtt{inner} = \mathtt{true} \end{array}\right\}$
} else if (c = 0){
$\{r \mapsto \blacklozenge * \mathsf{bnf} \wedge \mathtt{n.key} = \mathtt{k} \wedge \mathtt{continue} = \mathtt{true} \wedge \mathtt{inner} = \mathtt{true}\}$
update region: $\forall \mathcal{M}.$ $\langle \exists v.\, \mathsf{node}(\mathtt{n}, \mathtt{k}, v, -) \wedge (\mathtt{k}, v) \in \mathcal{M} \rangle$
atomic exists: $\forall v.$ $\langle \mathsf{node}(\mathtt{n}, \mathtt{k}, v, -) \rangle$
cas := CASValue(n, tv, v);
$\left\langle \begin{array}{l} \underline{\textbf{if}}\ \mathtt{cas} = \mathtt{true}\ \underline{\textbf{then}}\ \mathsf{node}(\mathtt{n}, \mathtt{k}, \mathtt{v}, -) \wedge \mathtt{tv} = v \\ \quad \underline{\textbf{else}}\ \mathsf{node}(\mathtt{n}, \mathtt{k}, v, -) \wedge \mathtt{tv} \neq v \end{array} \right\rangle$
$\langle \underline{\textbf{if}}\ \mathtt{cas} = \mathtt{true}\ \underline{\textbf{then}}\ \mathsf{node}(\mathtt{n}, \mathtt{k}, \mathtt{v}, -) \wedge (\mathtt{k}, \mathtt{v}) \in \mathcal{M}\ \underline{\textbf{else}}\ - \rangle$
$\left\{\begin{array}{l} \mathtt{continue} = \mathtt{true} \wedge \mathtt{inner} = \mathtt{true} \wedge \underline{\textbf{if}}\ \mathtt{cas} = \mathtt{false}\ \underline{\textbf{then}}\ r \mapsto \blacklozenge \\ \quad \underline{\textbf{else}}\ \left(\exists \mathcal{M}.\, r \mapsto ((\mathcal{M}, \mathcal{N}, \mathcal{B}), (\mathcal{M}[\mathtt{k} \mapsto \mathtt{v}], \mathcal{N}, \mathcal{B}))\right) \end{array}\right\}$
if (cas = true) { return tv;
$\{\exists \mathcal{M}.\, r \mapsto ((\mathcal{M}, \mathcal{N}, \mathcal{B}), (\mathcal{M}[\mathtt{k} \mapsto \mathtt{v}], \mathcal{N}, \mathcal{B}))\}$
} else { inner := false; }
$\{r \mapsto \blacklozenge \wedge \mathtt{n.key} = \mathtt{k} \wedge \mathtt{continue} = \mathtt{true} \wedge \mathtt{inner} = \mathtt{false}\}$
} } }
$\left\{\begin{array}{l} r \mapsto \blacklozenge * \mathsf{wkNdReach}(\mathtt{b}, \mathtt{n}) \wedge (\mathtt{b.key} < \mathtt{k} \vee \mathtt{b} = m) \wedge \\ \underline{\textbf{if}}\ \mathtt{inter} = \mathtt{true}\ \underline{\textbf{then}}\ \left(\underline{\textbf{if}}\ \mathtt{continue} = \mathtt{true}\ \underline{\textbf{then}}\ \mathtt{k} < \mathtt{n.key}\ \underline{\textbf{else}}\ -\right) \\ \quad \underline{\textbf{else}} \begin{pmatrix} \mathtt{cas} = \mathtt{false} \vee \mathtt{n} \neq \mathtt{n2} \vee \mathtt{n.value} = 0 \vee \\ \mathtt{b.value} = 0 \vee \mathsf{marker}(\mathtt{n}, -) \end{pmatrix} \end{array}\right\}$
if (inner = true && continue = true) {
$\{r \mapsto \blacklozenge * \mathsf{wkNdReach}(\mathtt{b}, \mathtt{n}) \wedge (\mathtt{b.key} < \mathtt{k} \vee \mathtt{b} = m) \wedge \mathtt{k} < \mathtt{n.key}\}$
z := makeNode(k, v, n);
$\left\{\begin{array}{l} r \mapsto \blacklozenge * \mathsf{wkNdReach}(\mathtt{b}, \mathtt{n}) * \mathsf{node}(\mathtt{z}, \mathtt{k}, \mathtt{v}, \mathtt{n}) \wedge \\ (\mathtt{b.key} < \mathtt{k} \vee \mathtt{b} = m) \wedge \mathtt{k} < \mathtt{n.key} \end{array}\right\}$



Then, the behaviour of method `remove` is different with the one in the map specification in Fig. 1. If the third parameter $\mathtt{v} = 0$, this method have the same behaviour as the one in Fig. 1. However, if $\mathtt{v} \neq 0$, it deletes a key only when the bound value is `v`.

The proof of `remove` use the wkNdReach prediacte from the proof of `put`. Apart from that, the predicate tracking asserts the state of tracking resource.

$$\mathsf{tracking} \equiv \underline{\mathbf{if}}\ \big(\mathtt{n} = 0 \vee \mathtt{k} < \mathtt{n.key} \vee (\mathtt{k} = \mathtt{n.key} \wedge \mathtt{tv} \neq \mathtt{v} \wedge \mathtt{v} \neq 0)\big)\ \underline{\mathbf{then}}\ r \Mapsto ((\mathcal{M}, \mathcal{N}, \mathcal{B}), (\mathcal{M}, \mathcal{N}, \mathcal{B}))\ \underline{\mathbf{else}}\ r \Mapsto \blacklozenge$$



abstract, substuting $s = (r, m)$, and abstract exists rule for $\mathcal{N}$ and $\mathcal{B}$

make atomic

```
while (outer = true) {   inner := true;   {SLMap_r(x, M, N, B, m)}
  b := findPredecessor(x, k);   {r ⊨ ♦ * b ⇝* N ⊎ {0}}
  n := [b.next];   {tracking * wkNdReach(b, n)}
  { tracking * wkNdReach(b, n) ∧ (b.key < k ∨ b = m) ∧ if inter = true then −
      else ( cas = false ∨ n ≠ n2 ∨ n.value = 0 ∨
             b.value = 0 ∨ marker(n, −) ) }
  while (inner = true) {   continue := true;
    if (n = 0) {   inner := false;   outer := false;
      {r ⊨ ((M, N, B), (M, N, B))}
    } else {   {tracking * wkNdReach(b, n) ∧ n ≠ 0}
      f := [n.next];   tv := [n.value];   marker := isMarker(n);
      n2 := [b.next]   vb := [b.value];
      {tracking * wkNdReach(b, n) * wkNdReach(n, f)}
      if (n ≠ n2) {   inner := false;
        {tracking * wkNdReach(b, n) * wkNdReach(n, f) ∧ n ≠ n2}
      } else if (tv = 0){   helpDelete(b, n, f);   inner := false;
        {tracking * wkNdReach(b, n) * wkNdReach(n, f) ∧ tv = n.value = 0}
      } else if (vb = 0 || marker = true){   inner := false;
        { tracking * wkNdReach(b, n) * wkNdReach(n, f) ∧ ( b = 0 ∨ marker(n, −) ) }
      } else {   tk := [n.key];   c := compare(k, tk);
        { wkNdReach(b, n) * wkNdReach(n, f) *
          if (c < 0 ∨ (c = 0 ∧ tv ≠ v ∧ v ≠ 0))
          then r ⊨ ((M, N, B), (M, N, B)) else r ⊨ ♦ }
        if (c < 0) {   inner := false;   outer := false;
          {r ⊨ ((M, N, B), (M, N, B))}
        } else if (c > 0){
          {r ⊨ ♦ * wkNdReach(b, n) * wkNdReach(n, f) ∧ k > n.key}
          b := n;   n := f;   continue := false;
          {r ⊨ ♦ * wkNdReach(b, n) ∧ k > b.key}
        } else if (v ≠ 0 && v ≠ tv){   {r ⊨ ((M, N, B), (M, N, B))}
          inner := false;   outer := false;
        } else {   {r ⊨ ♦ * node(−, n, k, −, −)}
                   update | ∀v.
                          | ⟨node(n, k, v, −)⟩
                          | cas := CASValue(n, tv, 0);
                          | ⟨ if cas = true then node(s, n, k, 0, −) ∧ v = v
                          |   else node(s, n, k, v, −) ∧ v ≠ v ⟩
          { if cas = false then r ⊨ ♦
            else ( ∃N = N' ⊎ {n} ∧
                   r ⊨ ((M, N, B), (M \ {k ↦ M(k)}, N', B ⊎ {n})) ) }
          if (cas = false) {   inner := false;   }
    } } }
    if (inner = true && continue = true) {
      // frame off r ⊨ ((M, N, B), (M \ {k ↦ M(k)}, N', B ⊎ {n}))
      {wkNdReach(b, n) * wkNdReach(n, f) ∧ n.value = 0}
```

abstract, substituting $s = (r, m)$, and
abstract exists rule for $\mathcal{N}$ and $\mathcal{B}$
make atomic

use
$\forall f.$
$\langle \mathsf{node}(s, \mathtt{n}, \mathtt{k}, 0, f) \rangle$
$\mathtt{apd} := \mathtt{appendMarker}(\mathtt{n}, \mathtt{f});$
$\left\langle \begin{array}{l} \underline{\textbf{if}}\ \mathtt{apd} = \mathtt{true} \\ \quad \underline{\textbf{then}}\ \exists z.\, \mathsf{node}(\mathtt{n}, \mathtt{k}, 0, z) * \mathsf{marker}(-, z, \mathtt{f}) \wedge f = \mathtt{f} \\ \quad \underline{\textbf{else}}\ \mathsf{node}(-, \mathtt{n}, \mathtt{k}, 0, f) \wedge f \neq \mathtt{f} \end{array} \right\rangle$
// either me or other appends a marker
$\left\{ \begin{array}{l} \exists z.\, \mathsf{wkNdReach}(\mathtt{b}, \mathtt{n}) * \\ (\mathsf{wkNdReach}(\mathtt{n}, \mathtt{f}) \wedge (\mathsf{node}(\mathtt{n}, \mathtt{k}, 0, z) * \mathsf{marker}(z, \mathtt{f}))) \end{array} \right\}$
}
if ($\mathtt{apd} = \mathtt{false}$ || $\mathtt{cas} = \mathtt{false}$) { search($\mathtt{x}, \mathtt{k}$); }
// search method has side effect
// to clean all dead index-nodes and dead nodes
else { findPredecessor($\mathtt{x}, \mathtt{k}$);
// findPredecessor method has side effect
// to clean all dead index-nodes
if ($\mathtt{x.head.right} = 0$) { tryReduceLevel($\mathtt{x}$); }
// tryReduceLevel might reduce a empty index list at top
} // All operations preserve invariant
// frame back $r \Mapsto ((\mathcal{M}, \mathcal{N}, \mathcal{B}), (\mathcal{M} \setminus \{\mathtt{k} \mapsto \mathcal{M}(\mathtt{k})\}, \mathcal{N}', \mathcal{B} \uplus \{n\})$
$\left\{ \begin{array}{l} \exists \mathcal{N}'.\, \mathcal{N} = \mathcal{N}' \uplus \{\mathtt{n}\} \wedge \\ r \Mapsto ((\mathcal{M}, \mathcal{N}, \mathcal{B}), (\mathcal{M} \setminus \{\mathtt{k} \mapsto \mathcal{M}(\mathtt{k})\}, \mathcal{N}', \mathcal{B} \uplus \{\mathtt{n}\}) \end{array} \right\}$
return tv;
$\left\{ \begin{array}{l} \exists \mathcal{N}'.\, \mathcal{N} = \mathcal{N}' \uplus \{\mathtt{n}\} \wedge \mathsf{ret} = \mathcal{M}(\mathtt{k}) \wedge \\ r \Mapsto ((\mathcal{M}, \mathcal{N}, \mathcal{B}), (\mathcal{M} \setminus \{\mathtt{k} \mapsto \mathsf{ret}\}, \mathcal{N}', \mathcal{B} \uplus \{\mathtt{n}\}) \end{array} \right\}$
} } }
$\{r \Mapsto ((\mathcal{M}, \mathcal{N}, \mathcal{B}), (\mathcal{M}, \mathcal{N}, \mathcal{B})\}$
return 0; $\{r \Mapsto ((\mathcal{M}, \mathcal{N}, \mathcal{B}), (\mathcal{M}, \mathcal{N}, \mathcal{B}) \wedge \mathsf{ret} = 0\}$
}
$\exists \mathcal{M}', \mathcal{N}', \mathcal{B}'.\, \mathbf{SLMap}_r(\mathtt{x}, \mathcal{M}', \mathcal{N}', \mathcal{B}', m) \wedge$
$\left\langle \begin{array}{l} \underline{\textbf{if}}\ (\mathtt{v} \neq 0 \wedge (\mathtt{k}, \mathtt{v}) \notin \mathcal{M}) \vee (\mathtt{v} = 0 \wedge \mathtt{k} \notin \mathcal{M} \downarrow_1) \\ \quad \underline{\textbf{then}}\ \mathcal{M} = \mathcal{M}' \wedge \mathcal{N} = \mathcal{N}' \wedge \mathcal{B} = \mathcal{B}' \wedge \mathsf{ret} = 0 \\ \quad \underline{\textbf{else}}\ \exists n.\, \mathcal{M}' = \mathcal{M} \setminus \{\mathtt{k} \mapsto \mathsf{ret}\} \wedge \mathcal{N} = \mathcal{N}' \uplus \{n\} \wedge \mathcal{B}' = \mathcal{B} \uplus \{n\} \end{array} \right\rangle$
$\left\langle \begin{array}{l} \exists \mathcal{M}'.\, \mathsf{Map}(s, \mathtt{x}, \mathcal{M}') \wedge \underline{\textbf{if}}\ (\mathtt{v} = 0 \wedge \mathtt{k} \in \mathcal{M}) \vee (\mathtt{k}, \mathtt{v}) \in \mathcal{M} \\ \quad \underline{\textbf{then}}\ \mathsf{ret} = \mathcal{M}(\mathtt{k}) \wedge \mathcal{M}' = \mathcal{M} \setminus \{\mathtt{k} \mapsto \mathsf{ret}\} \\ \quad \underline{\textbf{else}}\ \mathsf{ret} = 0 \wedge \mathcal{M} = \mathcal{M}' \end{array} \right\rangle$

Code blocks `buildIndexChain` and `insertIndexChain` preserving the invariant of a skiplist. Here we give functional correctness proofs. The proofs can imply the atomic specifications, because they do not change any observable states, $\mathcal{M}$, $\mathcal{N}$ and $\mathcal{B}$.

Predicates $\mathsf{indexChain}$ asserts a list of index-nodes at the address x. The parameter $\mathcal{R}$ is a set of next index-nodes (head-nodes) in each level and p is the node to which all index-nodes (head-nodes) point.

$$\begin{array}{rl} \mathsf{indexChain}_l(x, \mathcal{R}, p) \stackrel{\text{def}}{=} & (\mathcal{R} = [] \wedge l = 0) \vee \exists y, r, \mathcal{R}'. \\ & (\mathsf{index}(x, p, d, r) * \mathsf{indexChain}_{l-1}(d, \mathcal{R}', p) \wedge \mathcal{R} = [r] +\!\!+ \mathcal{R}') \end{array}$$

Similarly, $\mathsf{headChain}$ asserts a list of head-nodes that are supposed to point to the dummy node of a skiplist . This list of head-nodes starts from head-node at

the address x to a head-node pointing to y.

$$\mathsf{headChain}_l(x,\mathcal{R},p,y) \stackrel{\text{def}}{=} (x=y \wedge \mathcal{R}=[] \wedge l=0) \vee \exists r,d,\mathcal{R}'.\ \begin{pmatrix}\mathsf{headIndex}(x,l,p,d,r)* \\ \mathsf{headChain}_{l-1}(d,\mathcal{R}',p,y) \wedge \mathcal{R}=[r]+\!\!+\mathcal{R}'\end{pmatrix}$$

Similar to the proof of `findPredecessor`, we quantify the set of live index-nodes $\mathcal{N}_l$ and of dead index-nodes $\mathcal{B}_l$ for all level. Also, the prediacte Nil asserts a list of 0 pointers.

$$\mathsf{Nil} \equiv [0,\ldots]$$

Following is proofs in the order of `buildIndexChain` and `insertIndexChain`, we globally quntify the set of live nodes $\mathcal{N}$ and the of dead nodese $\mathcal{B}$ as well as for all level l, the set of live indexes $\mathcal{N}_l$ and the set of dead indexes $\mathcal{B}_l$.

$\{\mathtt{z} \in \mathcal{N} \uplus \mathcal{B} \wedge \mathtt{z.key} = \mathtt{k} \wedge \mathtt{level} > 0\}$
`buildIndexChain{  idx` $:= 0;$ `  h` $:=$ `[x.head];`
  // `x.head` can reach `h` or other way round
  $\{(\mathtt{x.head} \searrow^* \mathtt{h} \vee \mathtt{h} \searrow^* \mathtt{x.head}) \wedge \mathtt{z} \in \mathcal{N} \uplus \mathcal{B} \wedge \mathtt{z.key} = \mathtt{k} \wedge \mathtt{level} > 0\}$
  `max` $:=$ `[h.level];`
  `if (level` $\leq$ `max) {`
    `i` $:= 1;$ `  while (i` $\leq$ `level) {  idx` $:=$ `makeIndex(z, idx,` $0);$ `  i` $:=$ `i` $+ 1;$ `  }`
    $\left\{\begin{array}{l}\mathsf{indexChain}_{\mathtt{level}}(\mathtt{idx},\mathsf{Nil},\mathtt{z}) * (\mathtt{x.head} \searrow^* \mathtt{h} \vee \mathtt{h} \searrow^* \mathtt{x.head}) \wedge \\ \mathtt{z} \in \mathcal{N} \uplus \mathcal{B} \wedge \mathtt{z.key} = \mathtt{k} \wedge 0 < \mathtt{level} \leq \mathtt{max}\end{array}\right\}$
  `} else {  level` $:=$ `max` $+ 1;$ `  idxs` $:=$ `alloc(level` $+ 1);$ `  i` $:= 1;$
    `while (i` $\leq$ `level) {`
      `idx` $:=$ `makeIndex(z, idx,` $0);$ `  [idxs + i]` $:=$ `idx;  i` $:=$ `i` $+ 1;$ `  }`
    $\left\{\begin{array}{l}\left(\begin{array}{l}\bigwedge_{1\leq i\leq \mathtt{level}}(\exists y.\ \mathtt{idxs}+i \mapsto y* \\ \mathsf{indexChain}_i(y,\mathsf{Nil},\mathtt{z}))\end{array}\right) \wedge \mathsf{indexChain}_{\mathtt{level}}(\mathtt{idx},\mathsf{Nil},\mathtt{z}) \Big) * \\ (\mathtt{x.head} \searrow^* \mathtt{h} \vee \mathtt{h} \searrow^* \mathtt{x.head}) \wedge \\ \mathtt{z} \in \mathcal{N} \uplus \mathcal{B} \wedge \mathtt{z.key} = \mathtt{k} \wedge \mathtt{level} > \mathtt{max} > 0\end{array}\right\}$
    `outer` $:=$ `true;`
    `while (outer = true) {  h` $:=$ `[x.head];  oldLevel` $:=$ `[h.level];`
      // If `level` $\leq$ `oldLevel`, then no need for new head-nodes,
      // otherwise build new head-nodes and link in them
      `if (level` $\leq$ `oldLevel) {  outer` $:=$ `false;  }`
      `else {  newh` $:=$ `h;  oldbase` $:=$ `[h.node];  j` $:=$ `oldLevel` $+ 1;$
        `while (j` $\leq$ `level) {`
          `newh` $:=$ `makeHeadIndex(oldbase, newh, idxs[j], j);  j` $:=$ `j` $+ 1;$ `  }`
        // `newh` is a list that ends up at the old head `h`
        $\left\{\begin{array}{l}\exists y_1,\ldots,y_{\mathtt{level}}.\ \left(\begin{array}{l}\bigwedge_{1\leq i\leq \mathtt{level}}(\mathtt{idxs}+i \mapsto y_i * \mathsf{indexChain}_i(y_i,\mathsf{Nil},\mathtt{z})) \wedge \\ \mathsf{indexChain}_{\mathtt{level}}(\mathtt{idx},\mathsf{Nil},\mathtt{z})\end{array}\right) * \\ \mathsf{headIndexChain}(\mathtt{newh},[y_{\mathtt{level}},\ldots,y_{\mathtt{oldLevel}+1}],\mathtt{h},\mathtt{oldBase}) \wedge \\ \mathtt{z} \in \mathcal{N} \uplus \mathcal{B} \wedge \mathtt{z.key} = \mathtt{k} \wedge \mathtt{level} > \mathtt{oldLevel} > 0 \wedge \mathtt{oldbase} = m\end{array}\right\}$
        `cas` $:=$ `CASHead(x, h, newh);`
        $\left\{\begin{array}{l}\exists y_1,\ldots,y_{\mathtt{level}}.\ \left(\begin{array}{l}\bigwedge_{1\leq i\leq \mathtt{level}}(\mathtt{idxs}+i \mapsto y_i * \mathsf{indexChain}_i(y_i,\mathsf{Nil},\mathtt{z})) \wedge \\ \mathsf{indexChain}_{\mathtt{level}}(\mathtt{idx},\mathsf{Nil},\mathtt{z})\end{array}\right) * \\ \mathsf{headIndexChain}(\mathtt{newh},[y_{\mathtt{level}},\ldots,y_{\mathtt{oldLevel}+1}],\mathtt{h},\mathtt{oldBase}) * \\ \left(\begin{array}{l}\underline{\mathtt{if}}\ \mathtt{cas} = \mathtt{true}\ \underline{\mathtt{then}}\ \mathtt{x.head} \searrow^* \mathtt{newh} \vee \mathtt{newh} \searrow^* \mathtt{x.head} \\ \quad \underline{\mathtt{else}}\ \neg(\mathtt{x.head} \searrow^* \mathtt{newh} \vee \mathtt{newh} \searrow^* \mathtt{x.head})\end{array}\right) \wedge \\ \mathtt{z} \in \mathcal{N} \uplus \mathcal{B} \wedge \mathtt{z.key} = \mathtt{k} \wedge \mathtt{level} > \mathtt{oldLevel} > 0 \wedge \mathtt{oldbase} = m\end{array}\right\}$

```
        if (cas = true) {  h := newh;  level := oldlevel;
          idx := idxs[level];  outer := false;
           // Frame off some resource and
           // weaken to the asserts similar to another branch.
           {indexChain_level(idx, Nil, z) * (x.head ↘* h ∨ h ↘* x.head)∧
            z ∈ N ⊎ B ∧ z.key = k ∧ level > 0 ∧ outer = false}
    } } } // out of outer loop
  }
}  {indexChain_level(idx, Nil, z) * (x.head ↘* h ∨ h ↘* x.head)∧
    z ∈ N ⊎ B ∧ z.key = k ∧ level > 0}
```

```
{indexChain_level(idx, Nil, z) * (x.head ↘* h ∨ h ↘* x.head)∧
 z ∈ N ⊎ B ∧ z.key = k ∧ h.level ≥ level > 0}
insertIndexChain{  insertionLevel := level;  outer := true;
  while (outer = true) {  j := [h.level];  q := h;
    r := [q.right];  t := idx;  inner := true;
    {indexChain_insertionLevel(t, Nil, z) * (x.head ↘* q ∨ q ↘* x.head)*
     wkIdxReach(q, r, j) ∧ z ∈ N ⊎ B ∧ z.key = k ∧ level ≥ j ≥ insertionLevel}
     // Weaken to loop invariant †   {†}
          indexChain_insertionLevel(t, −, z) * (x.head ↘* q ∨ q ↘* x.head)*
     // † ≡ (wkIdxReach(q, r, j) ∨ q = 0) ∧ z ∈ N ⊎ B ∧ z.key = k∧
          level ≥ j ≥ insertionLevel
    while (inner = true) {  continue := true;
      if (q = 0 || t = 0) {  inner := false;  outer := false;  }
       // Deletion happens
      else if(r ≠ 0) {  n := [r.node];
        {indexChain_insertionLevel(t, −, z) * (x.head ↘* q ∨ q ↘* x.head)*
         wkIdxReach(q, r, j) ∧ z ∈ N ⊎ B ∧ z.key = k∧
         level ≥ j ≥ insertionLevel ∧ n = r.node}
        tv := [n.value];  if (tv = 0) {
          {indexChain_insertionLevel(t, −, z) * (x.head ↘* q ∨ q ↘* x.head)*
           wkIdxReach(q, r, j) ∧ z ∈ N ⊎ B ∧ z.key = k∧
           level ≥ j ≥ insertionLevel ∧ n = r.node ∧ n.value = 0}
          lk := unlink(q, r);
          {indexChain_insertionLevel(t, −, z) * (x.head ↘* q ∨ q ↘* x.head)*
           wkIdxReach(q, r, j) ∧ (if lk = true then r ∈ B_j else −)∧
           z ∈ N ⊎ B ∧ z.key = k∧
           level ≥ j ≥ insertionLevel ∧ n = r.node ∧ n.value = 0}
          if (lk = true) {  inner := false;  } else {
            r := [q.right];  continue := false;  }
           // Back to loop invariant   {†}
        } else {
          {indexChain_insertionLevel(t, −, z) * (x.head ↘* q ∨ q ↘* x.head)*
           wkIdxReach(q, r, j) ∧ z ∈ N ⊎ B ∧ z.key = k∧
           level ≥ j ≥ insertionLevel ∧ n = r.node}
          tk := [n.key];  c := compare(k, tk);
          if (c > 0) {  q := r;  r := [r.right];  continue := false;  }
           // Back to loop invariant   {†}
      } }
```

$\left\{\dagger \wedge \underline{\textbf{if}}\ \neg(\texttt{inner} = \texttt{true} \wedge \texttt{continue} = \texttt{true})\ \underline{\textbf{then}}\ - \atop \quad \underline{\textbf{else}}\ \texttt{q} \neq 0 \wedge \texttt{r} \neq 0 \wedge \texttt{k} \leq \texttt{n.key}\right\}$

```
if (inner = true && continue = true) {
```

$\{\dagger \wedge \texttt{q} \neq 0 \wedge \texttt{r} \neq 0 \wedge \texttt{k} \leq \texttt{n.key}\}$

```
  if (j = insertionLevel) {
```

$\{\dagger \wedge \texttt{q} \neq 0 \wedge \texttt{r} \neq 0 \wedge \texttt{k} \leq \texttt{n.key} \wedge \texttt{j} = \texttt{insertionLevel}\}$

```
    lk := link(q, r, t);
```

$$\left\{\begin{array}{l}\mathsf{indexChain}_{\texttt{insertionLevel}}(\texttt{t}, [\texttt{r}, \ldots], \texttt{z}) * (\texttt{x.head} \searrow^* \texttt{q} \vee \texttt{q} \searrow^* \texttt{x.head}) * \\ (\underline{\textbf{if}}\ \texttt{lk} = \texttt{true}\ \underline{\textbf{then}}\ \mathsf{wkIdxReach}(\texttt{q}, \texttt{t}, \texttt{j})\ \underline{\textbf{else}}\ -) * \\ \mathsf{wkIdxReach}(\texttt{t}, \texttt{r}, \texttt{j}) \wedge \texttt{z} \in \mathcal{N} \uplus \mathcal{B} \wedge \texttt{z.key} = \texttt{k} \wedge \\ \texttt{level} \geq \texttt{j} = \texttt{insertionLevel} \wedge \texttt{q} \neq 0 \wedge \texttt{r} \neq 0 \wedge \texttt{k} \leq \texttt{n.key}\end{array}\right\}$$

```
    if (lk = false) {  inner := false;  }
     // Weaken to invariant
```

$\{\dagger\}$

```
    else if(t.node.value = 0) {
      findNode(x, k);  inner := false;  outer := false;  }
     // Find out the node has been deleted
     // method findNode(x, k) try to clean.
    else if(insertionLevel = 1) {  inner := false;  outer := false;
```

$$\left\{\begin{array}{l}\mathsf{indexChain}_1(\texttt{t}, [\texttt{r}], \texttt{z}) * (\texttt{x.head} \searrow^* \texttt{q} \vee \texttt{q} \searrow^* \texttt{x.head}) * \\ \mathsf{wkIdxReach}(\texttt{q}, \texttt{t}, \texttt{j}) * \mathsf{wkIdxReach}(\texttt{t}, \texttt{r}, \texttt{j}) \wedge \\ \texttt{z} \in \mathcal{N} \uplus \mathcal{B} \wedge \texttt{z.key} = \texttt{k} \wedge \texttt{level} \geq \texttt{j} = \texttt{insertionLevel} \wedge \\ \texttt{q} \neq 0 \wedge \texttt{r} \neq 0 \wedge \exists v \neq 0.\, [\mathrm{V}(\texttt{n}, v)] \wedge \texttt{k} \leq \texttt{n.key}\end{array}\right\}$$

```
    } else {  insertionLevel := insertionLevel − 1;  }
  }
```

$$\left\{\underline{\textbf{if}}\ \neg(\texttt{inner} = \texttt{true} \wedge \texttt{continue} = \texttt{true})\ \underline{\textbf{then}}\ \dagger \vee \texttt{t.node.value} = 0 \atop \quad \underline{\textbf{else}} \begin{pmatrix}\mathsf{indexChain}_{\texttt{insertionLevel}+1}(\texttt{t}, [\texttt{r}, \ldots], \texttt{z}) * \\ (\texttt{x.head} \searrow^* \texttt{q} \vee \texttt{q} \searrow^* \texttt{x.head}) * \\ \mathsf{wkIdxReach}(\texttt{q}, \texttt{t}, \texttt{j}) * \mathsf{wkIdxReach}(\texttt{t}, \texttt{r}, \texttt{j}) \wedge \texttt{z} \in \mathcal{N} \uplus \mathcal{B} \wedge \\ \texttt{z.key} = \texttt{k} \wedge \texttt{level} \geq \texttt{j} > \texttt{insertionLevel} \wedge \texttt{q} \neq 0 \wedge \\ \texttt{r} \neq 0 \wedge \texttt{k} \leq \texttt{n.key}\end{pmatrix}\right\}$$

```
  if (inner = true && continue = true) {  j := j − 1;
```

$$\left\{\begin{array}{l}\mathsf{indexChain}_{\texttt{insertionLevel}+1}(\texttt{t}, [\texttt{r}, \ldots], \texttt{z}) * \\ (\texttt{x.head} \searrow^* \texttt{q} \vee \texttt{q} \searrow^* \texttt{x.head}) * \mathsf{wkIdxReach}(\texttt{q}, \texttt{t}, \texttt{j}+1) * \\ \mathsf{wkIdxReach}(\texttt{t}, \texttt{r}, \texttt{j}+1) \wedge \texttt{z} \in \mathcal{N} \uplus \mathcal{B} \wedge \texttt{z.key} = \texttt{k} \wedge \\ \texttt{level} \geq \texttt{j} \geq \texttt{insertionLevel} \wedge \texttt{q} \neq 0 \wedge \texttt{r} \neq 0 \wedge \texttt{k} \leq \texttt{n.key}\end{array}\right\}$$

```
    if (j ≥ insertionLevel && j < level) {  t := [t.down];  }
    q := [q.down];  r := [q.right];
     // Frame off some resource and weaken to invariant †
```

$\{\dagger\}$

```
  } }
```

$\{\dagger\}$

```
  }  // Out of inner loop, finish all process or restart process
```

$$\left\{\dagger \wedge \begin{pmatrix}\texttt{q} = 0 \vee \texttt{r} = 0 \vee \texttt{t.node.value} = 0 \vee \\ \texttt{r} \in \mathcal{B}_{\texttt{j}} \vee \exists r.\, [\mathrm{I}(\texttt{q}, r)] \vee \texttt{insertionLevel} = 1\end{pmatrix}\right\}$$

```
} // Out of outer loop, finish insertion or not necessary to continue.
}
```