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 issue lies in the development of modular specifications, which provide clear logical boundaries between clients and implementations. A solution is to use recent advances in fine-grained concurrency reasoning, in particular the introduction of *abstract atomicity* to concurrent separation logic reasoning. We present two specifications of concurrent maps, both providing the clear boundaries we seek. We show that these specifications are equivalent, in that they can be built from each other. We show how we can verify client programs, such as a concurrent set and a producer-consumer client. We also give a substantial first proof that the main operations of `ConcurrentSkipListMap` in `java.util.concurrent` satisfy the map specification. This work demonstrates that we now have the technology to verify the largest implementations in `java.util.concurrent`.

1 Introduction

We study reasoning about fine-grained concurrent data-structure libraries, with the aim of developing modular specifications which provide a clear logical boundary between client programs and implementations. We refer to a specification as modular if it enables us to verify client programs without exposing the details of the underlying implementation. Implementations should be provably correct with respect to such specifications. Specifications should be general enough to allow for the verification of strong functional properties of arbitrary clients. Such a balance has been difficult to achieve.

There has been substantial recent work on the modular specification of concurrent libraries and the verification of their clients and implementations using concurrent separation logics: see for example [7]. However, we have only just reached a stage where specifications are fully modular. In particular, an important step has been the introduction of *abstract atomicity* to concurrent separation-logic reasoning [6,16,26,17,27,20]. We revisit the concurrent map example, a pivotal example in the original development of concurrent abstract predicates [10,5]. We demonstrate significant improvement with this work, by presenting two specifications of concurrent maps: one based on the whole map data structure; and the other on key-value pairs. These specifications are more general, yielding better functional properties of client programs and simpler proofs

that implementations meet the specifications. In particular, we are able to give a substantial first proof that the main operations of `ConcurrentSkipListMap` in `java.util.concurrent` satisfy the map specification. This work demonstrates that we now have the reasoning techniques required to verify the largest implementations in `java.util.concurrent`.

The specification and verification of concurrent maps has been fundamental to the development of abstract fine-grained concurrent reasoning. At one point, it was known how to verify Sagiv's B^{Link} tree algorithm [24] in RGSep [28], but it was not known how that reasoning could be lifted to an abstract specification. In an effort to answer this question, concurrent abstract predicates and the associated CAP reasoning were introduced [10], and a specification of concurrent maps using key-value pairs presented [5]. At the time, it was quite an achievement that such a complex algorithm could be proven correct with respect to a simple abstract specification. However, it was also known that the specification had substantial limitations. It was not general enough, in that it had specific protocol tags embedded in the specification to manage interaction between threads. This meant that, although CAP reasoning was sufficiently expressive to demonstrate the memory safety of all clients, it could only verify functional correctness properties for some. It also meant that, although implementations such as Sagiv's B^{Link} tree were indeed proven correct with respect to the specification, the proofs were complex due to the explosion of cases caused by the protocol tags. The key point is that CAP reasoning was not able to establish the correct specification boundary.

In this paper, we present two modular specifications for concurrent maps, which we believe provide clear logical boundaries for verifying implementations and clients. First, we present an abstract concurrent map specification where the focus is on the entire map data structure; such a specification is particularly suitable for verifying implementations. Using this map specification we verify an implementation of an abstract concurrent set specification, similar to `ConcurrentSkipListSet` from `java.util.concurrent`. Second, we present an alternate concurrent map specification, in which the focus is on key-value pairs, rather than on the entire map data structure. This type of specification is more appropriate for clients who only require access to some of the key-value pairs of the map. These two specifications are equivalent in the sense that the key-value specification can be built as a client of the map specification, and vice versa; they present two different views of the same data structure.

We demonstrate how to build the CAP specification from our more general key-value specification in the technical report [29], immediately inheriting all of the client examples given in [5]. We verify a functional correctness property of a simple producer-consumer client using the key-value specification, which cannot be verified using the CAP specification. We also verify that the main operations of `ConcurrentSkipListMap` from `java.util.concurrent` satisfy the map specification. As far as we are aware, this is the largest verified example of an algorithm of `java.util.concurrent` in the literature. Despite this, the proof is comparatively simpler than the CAP-style proof Sagiv's B^{Link} tree algorithm.

This is because the verification of our specification is decoupled from how the map is used by concurrent clients.

The strength of our results is due to advances in fine-grained concurrent reasoning, in particular the introduction of *abstract atomicity* made popular with the work on linearisability [15] and recently integrated into the reasoning of concurrent separation logics [6,16,26,17,27,20]. Atomicity is a common and useful abstraction for 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. We use TaDA [6], a program logic which captures abstract atomicity by introducing the notion of an *atomic triple*. These triples generalise the concept of linearisability, in the sense that they specify operations to be atomic with respect to interference restrictions on the data abstraction, in contrast to unrestricted interference on the module boundary given by linearisability. The outcome is the clear specification boundary that we seek. In fact, whilst studying the concurrent map example, we realised that we could extend TaDA with additional proof rules to provide more modular specifications than before.

The results in this paper demonstrate that choosing the right abstraction for the specification is essential for scalable reasoning about concurrent data structures in general, and concurrent maps in particular. For concurrent maps, we have demonstrated that the right abstraction is feasible using the TaDA program logic. With TaDA, we have introduced two general specifications of concurrent maps, which do not impose unnecessary constraints on the client reasoning. We have verified the main operations of `ConcurrentSkipListMap`, a large real-world concurrent data structure algorithm given in `java.util.concurrent`.

2 Abstract Map Specification

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

Figure 1 shows our abstract specification for the main operations of the `ConcurrentMap` module. This specification is abstract in the sense that it does not contain any references to the underlying implementation. To represent the map as a resource, we introduce an abstract predicate $\mathsf{Map}(s, x, \mathcal{M})$. The first parameter of the predicate, $s \in \mathbb{T}_1$, ranges over an abstract type $\mathbb{T}_1$. It captures invariant implementation-specific information about the map, which does not

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

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

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

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

Fig. 1. Specification for a concurrent map.

change during the execution of the operation. To the client, the type is opaque; the implementation realises the type appropriately. The second parameter, $x \in \mathsf{Loc}$, represents the physical address of the map object. The last parameter, $\mathcal{M}$, contains a set of mappings that represent the abstract state of the map.

The `makeMap` operation is specified with a standard Hoare triple that asserts that it will return a freshly allocated map object with no mappings. We should note here that, since TaDA is an intuitionistic logic, the precondition True has the same behaviour as the emp of classical separation logic. Also, ret is a special variable that holds the return value of an operation.

The `get`, `put` and `remove` operations are specified with an *atomic triple*, with the intended meaning that these operations appear to take place atomically, at a distinct point in time. TaDA is designed so 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 atomic triple, introduced by TaDA [6], specifies an atomic operation:

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

the intuitive meaning of which we will explain shortly. Prior to that, we need to introduce the notion of a *linearisation point*: a linearisation point of an operation is the instant at which that operation appears to take effect [15]. The precondition $P(x)$ and postcondition $Q(x)$ specify the program state before and after the linearisation point of the program $\mathbb{C}$. The pseudo-quantification $\forall x \in X$ specifies that the environment is allowed to freely change the value of x before the linearisation point of $\mathbb{C}$, as long as $x \in X$ and $P(x)$ continues to hold. At and only at the linearisation point, current program $\mathbb{C}$ changes the program state from $P(x)$ to $Q(x)$. Afterwards there are no guarantees as to the truthfulness of $Q(x)$ because the environment can interfere with no constraint[1]. For example, the specification for `get` in Fig. 1 means that immediately after the linearisation point, `get` preserves the state of the mappings and returns the current value associated with the key `k`, or 0 if the mapping does not exist. Before the linearisation point, the environment can change the value of the mappings, but cannot deallocate the structure because the precondition mandates its existence; after

[1] For a detailed exposition of the atomic triples, readers should refer to [6,7].

the linearisation point, the environment can interfere fully and even deallocate the structure.

The `put` operation is similar to `get`, except that it inserts or replaces the mapping for the key `k` with the value `v` ($\mathcal{M}[\mathtt{k} \mapsto \mathtt{v}]$ denotes the update of the existing key `k` with value `v`, or the addition of a new $(\mathtt{k}, \mathtt{v})$ pair, depending on the mapping previously existed). The operation returns the previous mapping associated with the key, or 0 if that mapping did not exist. Finally, the `remove` operation removes an existing mapping associated with the key `k` and returns its previous contents, similarly to the `put` operation.

We show that our specification is strong enough to reason about arbitrary clients, e.g. a concurrent set and producer-consumer, in §3, and verify that a complex skiplist implementation satisfies it in §4. Moreover, we derive an alternative key-value specification which provides a fiction of disjointness and can also be used for client reasoning.

3 Client Reasoning

We illustrate the advantages of our specification by showing three different ways of using it. The first example is an implementation of a concurrent set module, similar to the `ConcurrentSkipListSet` found in `java.util.concurrent`, that makes use of a map internally and illustrates the modularity of our specification. The second example is a new key-value specification that incorporates atomicity and allows dynamic control of the number of abstract key-value predicates being used. It is motivated by the fact that some clients find it preferable to work with individual key-value pairs rather than the whole map and is inspired by the original key-value specification for concurrent maps presented in Concurrent Abstract Predicates (CAP) [5,10]. The third example is a simplified producer-consumer scenario that makes use of the key-value specification. We use it to show how abstract atomicity can be used to improve client reasoning, allowing us to prove stronger properties about functional correctness than previous approaches [1,2,5].

In this section, when necessary, we will elaborate on certain important notions of TaDA. The reader can refer to the full set of TaDA rules in [4], which also includes the two additional rules with which we have extended TaDA.

3.1 Concurrent Set

We consider a concurrent set module, with the specification given in Fig. 2. This module consists of two methods: `setPut`, which inserts an element in the set and returns true if the element did not previously exist and false if it did; and `setRemove`, which removes an element from the set and returns true if the element previously existed in the set and false otherwise. The implementation internally uses a concurrent map to keep track of the elements that are in the set. If an element e is in the set, then there exists a corresponding mapping with the key e in the underlying map.

$$\vdash \mathbb{V}\mathcal{S}.\langle \mathsf{Set}(s, \mathtt{x}, \mathcal{S})\rangle\ \mathtt{setPut}(\mathtt{x}, \mathtt{e})\ \langle \mathsf{Set}(s, \mathtt{x}, \mathcal{S} \cup \{\mathtt{e}\}) \wedge \mathsf{ret} = (\mathtt{e} \notin \mathcal{S})\rangle$$
$$\vdash \mathbb{V}\mathcal{S}.\langle \mathsf{Set}(s, \mathtt{x}, \mathcal{S})\rangle\ \mathtt{setRemove}(\mathtt{x}, \mathtt{e})\ \langle \mathsf{Set}(s, \mathtt{x}, \mathcal{S} \setminus \{\mathtt{e}\}) \wedge \mathsf{ret} = (\mathtt{e} \in \mathcal{S})\rangle$$

Fig. 2. Specification for a concurrent set.

Shared Regions, Transition Systems and Guards In order to verify the specification using TaDA [6], we must provide an interpretation for the abstract predicate $\mathsf{Set}(s, \mathtt{x}, \mathcal{S})$. For this, we will introduce the *shared regions*, *transition systems* and *guards* of TaDA.

A *shared region* encapsulates resources that may be shared by multiple threads, with the proviso that they can only be accessed by atomic operations. It has an abstract state with a concrete interpretation denoted by $I(-)$. Each region is identified by a unique region identifier and adheres to a region type.

For the concurrent set module, we introduce a region type **SLSet**. A region of this type is parameterised by the physical address x of the underlying map and an additional physical address y, which corresponds to the address of the set. It also carries the parameter s, which captures the invariant information of the underlying map implementation. Lastly, the abstract state $\mathcal{S}$ corresponds to the contents of the set. The interpretation of **SLSet** is as follows:

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

Therefore, we have that **SLSet** encapsulates a heap cell that contains the address of the underlying map as well as the underlying map itself, and also relates the abstract state of the region to the domain of the mappings. The subscript r is the unique region identifier.

A shared region is associated with abstract resources, called guards, and a labelled transition system, where labels are guards, that defines how the region can be updated. In TaDA, the assertion $[\mathrm{G}]_r$ denotes a guard named G for the region r. The guards for a region form a partial commutative monoid (PCM), to which we refer to as a *guard algebra*, with a composition operation $\bullet$, which is lifted to $*$ in TaDA assertions. For the **SLSet** region, we introduce two types of guards, G and $\mathbf{0}$. The guard algebra for the set module is as follows:

$$\mathrm{G} \bullet \mathbf{0} = \mathbf{0} \bullet \mathrm{G} = \mathrm{G} \qquad \mathbf{0} \bullet \mathbf{0} = \mathbf{0} \qquad \mathrm{G} \bullet \mathrm{G} \text{ is undefined}$$

where the guard $\mathbf{0}$ is the unit of this PCM. This guard algebra ensures that the guard G is unique. The type of the region, $\mathbf{t}$, defines the transition system and the guard algebra associated with the region.

The labelled transition system associates certain actions with certain guards; these actions determine how a thread that holds this guard can update the shared state of the region. An action is a pair consisting of a pre- and a post-condition. The labelled transition system for the set region is as follows:

$$\mathrm{G} : \forall \mathcal{S}, e.\, \mathcal{S} \rightsquigarrow \mathcal{S} \cup \{e\} \qquad \mathrm{G} : \forall \mathcal{S}, e.\, \mathcal{S} \rightsquigarrow \mathcal{S} \setminus \{e\}$$

The guard G allows a thread to change its abstract state $\mathcal{S}$, effectively changing the contents of the set by either adding or removing an element e. The $\mathbf{0}$

guard has no bound action. Note that in the following discussion, if there is no ambiguity, each region implicitly has a unit guard **0** with no bound action.

Given the region and its guards, we are now ready to give the interpretation of the abstract type and our abstract predicate:

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

where RId is the set of region identifiers, $\mathbb{T}_1$ is the abstract type of the concurrent map specification and Loc is the set of physical addresses. For the first parameter of Set, s, we have that $s \in \mathbb{T}_2$. Recall that abstract types $\mathbb{T}_1$ and $\mathbb{T}_2$ encapsulate invariant, implementation-specific information and are opaque for clients.

Given this interpretation of the set module, it remains to prove that the implementations of the operations, which use a concurrent map, satisfy the set specifications. We present the proof of the `setPut` in Fig. 3, whereas we omit the proof for `setRemove` due to its similarity and it can be found in technical report [29]. The proof begins by substituting the abstract predicate $\mathsf{Set}(s, \mathtt{x}, \mathcal{S})$ with its interpretation, where we also substitute the logical parameter $s \in \mathbb{T}_2$ with its interpretation as the triple (r, s', y), where r is the identifier of the **SLSet** region, s' is the abstract logical parameter of the underlying Map predicate, and y is the physical address of the underlying Map predicate.

The remaining proof rules used will be explained shortly. We also show a further example of how to use the set specification in the technical report [29], by proving a parallel sieve of Eratosthenes.

Proof Rules There are four key rules that are used in Fig. 3: *make atomic*, *update region*, and *atomicity weakening*. The first rule that we will describe is *make atomic*, which allows us to prove that program $\mathbb{C}$ can be seen as abstractly atomic. A simplified version of this rule is as follows:

$$\frac{\{(x, y) \mid x \in X, y \in f(x)\} \subseteq \mathcal{T}_{\mathbf{t}}(\mathrm{G})^* \qquad r : x \in X \rightsquigarrow f(x) \vdash \left\{\exists x \in X.\, \mathbf{t}_r(\vec{z}, x) * r \Rightarrow \blacklozenge\right\}\ \mathbb{C}\ \left\{\exists x \in X, y \in f(x).\, r \Rightarrow (x, y)\right\}}{\vdash \mathbb{V} x \in X. \left\langle \mathbf{t}_r(\vec{z}, x) * [\mathrm{G}]_r \right\rangle\ \mathbb{C}\ \left\langle \exists y \in f(x).\, \mathbf{t}_r(\vec{z}, y) * [\mathrm{G}]_r \right\rangle}$$

This rule establishes that $\mathbb{C}$ atomically updates the region r, from a state $x \in X$ to a state $y \in Q(x)$. To do so, it requires the guard G for the region, which must permit the update according to the appropriate transition system $\mathcal{T}_{\mathbf{t}}(\mathrm{G})^*$, where $\mathbf{t}$ is the region type; this is established by the first premiss. Here, the region type $\mathbf{t}$ is **SLSet**, the guard G is our guard G, and the transition system is $\mathcal{T}_{\mathbf{t}}(\mathrm{G}) = \{\mathcal{S}, \mathcal{S} \cup \{e\}\} \cup \{\mathcal{S}, \mathcal{S} \setminus \{e\}\}$. The * denotes reflexive-transitive closure.

We use $\mathbf{t}_a(\vec{z}, x)$ to represent a region with region type $\mathbf{t}$, identifier a, parameters $\vec{z}$ and abstract state x. In our example, the **SLSet** region is parametrised with the physical address of the underlying map, the physical address of the set, the invariant implementation-specific information, and the abstract state, which corresponds to the contents of the set. The second premiss introduces two notations. The first, $r : x \in X \rightsquigarrow f(x)$, is called the *atomicity context*. The atomicity context records the abstract atomic action that is to be performed. The second, $r \Rightarrow -$, is the *atomic tracking resource*. The atomic tracking resource indicates

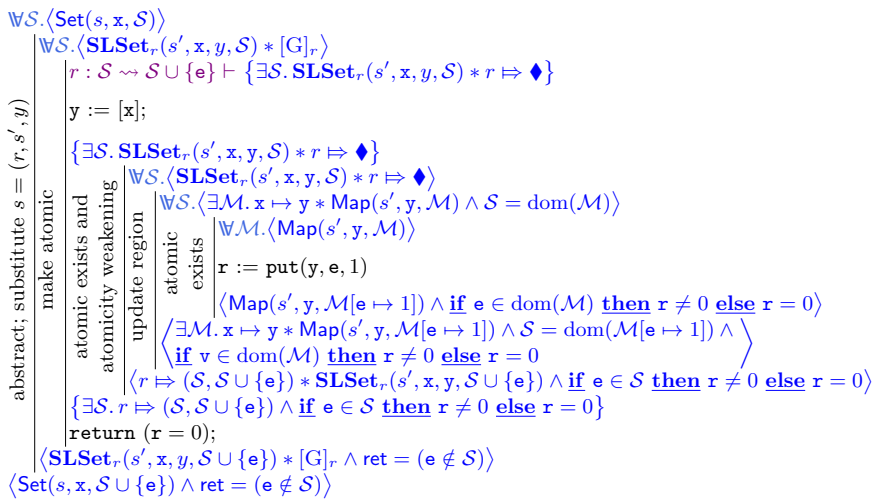


Fig. 3. Proof of correctness of the `setPut` operation.

whether or not the atomic update has occurred (the $r \Mapsto \blacklozenge$ indicates it has not) and, if so, the state of the shared region immediately before and after (the $r \Mapsto (x, y)$). The resource $r \Mapsto \blacklozenge$ also plays two special roles that are normally filled by guards. Firstly, it limits the interference on region r: the environment may only update the state so long as it remains in the set X, as specified by the atomicity context. Secondly, it confers permission for the thread to update the region from state $x \in X$ to any state $y \in f(x)$; in doing so, the thread also updates $r \Mapsto \blacklozenge$ to $r \Mapsto (x, y)$. This permission is expressed by the *update region* rule (see below), and ensures that the atomic update happens only once.

In essence, the second premiss is capturing the notion of atomicity (with respect to the abstraction in the conclusion) and expressing it as a proof obligation. Specifically, the region must be in the state x for some $x \in X$, which may be changed by the environment, until at some point the thread updates it to some $y \in f(x)$. The atomic tracking resource bears witness to this.

In the proof shown in Fig. 3, we apply the *make atomic* rule to declare an atomic update. The local variable `y` holds the address of the underlying map. Then, at the linearisation point, i.e. the `put` operation, we apply the update region rule to update the tracking resource from $r \Mapsto \blacklozenge$ to $r \Mapsto (\mathcal{S}, \mathcal{S} \cup \{\mathtt{e}\})$. The slightly simplified version of the *update region* rule is as follows:

$$\frac{\vdash \mathbb{W}x \in X. \left\langle I(\mathbf{t}_r(\vec{z}, x)) * P(x) \right\rangle \ \mathbb{C}\ \left\langle \begin{array}{c} \exists y \in f(x).\, I(\mathbf{t}_r(\vec{z}, y)) * Q_1(x, y) \\ \vee\ I(\mathbf{t}_r(\vec{z}, x)) * Q_2(x) \end{array} \right\rangle}{r : x \in X \rightsquigarrow f(x) \vdash \mathbb{W}x \in X. \left\langle \begin{array}{c} \mathbf{t}_r(\vec{z}, x) * P(x) \\ * \, r \Mapsto \blacklozenge \end{array} \right\rangle \ \mathbb{C}\ \left\langle \begin{array}{l} \exists y \in f(x).\, \mathbf{t}_r(\vec{z}, y) * Q_1(x, y) * \\ r \Mapsto (x, y) \vee \mathbf{t}_r(\vec{z}, x) * Q_2(x) * r \Mapsto \blacklozenge \end{array} \right\rangle}$$

In the conclusion, the first disjunct of the postcondition specifies that an atomic update has occurred, and that $\mathbb{C}$ updates the abstract state of the region r (of type $\mathbf{t}$ and parameters $\vec{z}$) from x to y and also $P(x)$ to $Q_1(x, y)$, and in doing so consumes the tracking resource. As the atomic update might affect

resources contained in $P(x)$, the postcondition $Q_1(x, y)$ is parametrised by y. By having $y = x$, we can also allow for the abstract state of the region to remain unchanged and not perform the atomic update, which is useful when reasoning about an atomic read. The second disjunct of the postcondition specifies that $\mathbb{C}$ only change the local resource to $Q_2(x)$ and that the atomic action has not taken place. Note that the tracking resource is only allowed to change only once from $\blacklozenge$ to (x, y), at the instant in which the atomic update takes effect.

TaDA allows eliminating existential quantification only for Hoare triples. However, a rule of following form would not be sound:

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

The conclusion allows the environment to change the value of x because of the quantification, while in the premiss the value cannot be changed by the environment. This means that anything that could be potentially existentially quantified has to be exposed as a parameter and bounded by the pseudo-quantification $\mathbb{W}$. We overcome this problem by extending the logic with the *atomic exists* proof rule that allows the elimination of existential quantification, as follows:

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

In this rule, $\mathbb{C}$ tolerates changes of the value of x as long as they are contained in X by lifting the existential quantification to pseudo-quantification. This proof rule allows us to hide the underlying states from the abstract specification of a module, while maintaining the soundness of TaDA. We apply this rule both before and after the update region rule in Fig. 3.

We now show how standard Hoare triples are derived from atomic Hoare triples. In fact, the proof in Fig. 3 implicitly performs this step when applying the *update region* rule. In TaDA, this is captured by the *atomicity weakening* rule, a simplified form of which is given below:

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

The Hoare triple in the conclusion states that the program $\mathbb{C}$ updates the state from $P(x)$ to $Q(x)$ without any interference from the environment on x, since $\forall x \in X$ in the context fixes the value of x during execution. Note that in TaDA, assertions are implicitly required to be stable, which means that the assertions must hold with respect to a protocol. Recall the atomic specification of `put` in Fig. 1: at that level, no protocol for how the concurrent map is used is defined, and therefore the precondition and postcondition assertions are trivially stable. However, when applying the *update region* rule in Fig. 3, we are using the concurrent map as part of the protocol defined for the **SLSet** region, so we must keep assertions stable with respect to that protocol. For example, at the point of the *update region*, we implicitly weaken the atomic postcondition before transitioning to the non-atomic postcondition, to keep the postcondition stable.

3.2 Key-value Specification

The concurrent map specification in Fig. 1 is given in terms of all the key-value mappings present in the map and we have used this specification to verify a concurrent set implementation. However, for some clients it may be preferable to work with individual key-value pairs rather than the whole map. Inspired by the key-value specification using Concurrent Abstract Predicates (CAP) [10,5], we introduce a new key-value specification. This new specification is atomic, in contrast to the CAP-stype specification. We show how to build a key-value specification from the map specification and vice versa, meaning that these two specifications are, in fact, equivalent and simply represent two different ways of thinking about a concurrent map.

CAP-style Approach Concurrent map specifications in terms of individual key-value pairs were first introduced using CAP [10,5]. To allow sharing between threads, concurrent abstract predicates associate key-value pairs with fractional permissions [3] and special tags $\{\texttt{def}, \texttt{ins}, \texttt{rem}, \texttt{unk}\}$ that define the protocol via which multiple threads access shared key-value pairs. Each operation is specified for all protocol tags. For example, consider the following CAP-style specification of `remove` (the specifications for `get` and `put` are in the technical report [29]):

$$\{\mathsf{CAPKey}_{\texttt{def}}(\texttt{x},\texttt{k},v)_1\}\ \texttt{remove}(\texttt{x},\texttt{k})\ \{\mathsf{CAPKey}_{\texttt{def}}(\texttt{x},\texttt{k},0)_1 \wedge \mathsf{ret} = v\}$$
$$\{\mathsf{CAPKey}_{\texttt{rem}}(\texttt{x},\texttt{k},v)_\pi\}\ \texttt{remove}(\texttt{x},\texttt{k})\ \{\mathsf{CAPKey}_{\texttt{rem}}(\texttt{x},\texttt{k},0)_\pi \wedge \mathsf{ret} \in \{v,0\}\}$$
$$\{\mathsf{CAPKey}_{\texttt{unk}}(\texttt{x},\texttt{k})_\pi\}\ \texttt{remove}(\texttt{x},\texttt{k})\ \{\mathsf{CAPKey}_{\texttt{unk}}(\texttt{x},\texttt{k})_\pi\}$$

The concurrent abstract predicate CAPKey is used as a key-value pair resource for the map with address `x`. The tag determines what the current thread and the environment can do with this resource in accordance to Table 1. The subscript π is a fractional permission, in the range $(0, 1]$, controlling how the resource is shared between threads: when $\pi = 1$, the current thread fully owns the key-value pair resource, whereas when $\pi < 1$, the resource is shared. In the `def` case, if $v \neq 0$, $\mathsf{CAPKey}_{\texttt{def}}(\texttt{x},\texttt{k},v)_1$ asserts the existence of the key-value pair $(\texttt{k}, v)$; otherwise, if $v = 0$ the key `key` is not mapped to a value. In this case, only the current thread is manipulating this key-value pair with full permission $\pi = 1$. In the `ins` case, $\mathsf{CAPKey}_{\texttt{ins}}(\texttt{x},\texttt{k},v)_\pi$ asserts that the current thread and the environment are allowed to insert the value v for key `key`. Consequently, the specification prohibits threads from performing a `remove` in this case. In the `rem` case, $\mathsf{CAPKey}_{\texttt{rem}}(\texttt{x},\texttt{k},v)_\pi$ asserts that the both the current thread and the environment are allowed to remove the key `key` from the map. Consequently, in the precondition of the specification, the key may or may not exist in the map. However, it certainly does not exist in the postcondition. Finally, in the `unk` case, $\mathsf{CAPKey}_{\texttt{unk}}(\texttt{x},\texttt{k})_\pi$ asserts that both the thread and the environment are arbitrarily manipulating the key `key`. This case allows arbitrary interference and thus we know nothing about the existence of the key or its value. Specifications with `unk` are overly weak and only allow memory-safety proofs.

There are two significant drawbacks in the CAP approach. First, the specification hard-codes specific protocols by which threads access the shared resource,

Table 1. Sharing protocols for key-value pairs of a concurrent map in CAP.

	Thread		Environment			Thread		Environment	
	put	remove	put	remove		put	remove	put	remove
def	✓	✓	✗	✗	ins	✓	✗	✓	✗
unk	✓	✓	✓	✓	rem	✗	✓	✗	✓

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

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

i.e. the tag and the fractional permission. This limits the verification of functional correctness properties to only those clients that fit the hard-coded protocols; the producer-consumer example in §3.3 does not. Second, each operation has to be separately verified per each protocol tag, drastically increasing the proof effort.

Atomic Key-value Specification We introduce our key-value based specification in Fig. 4. This specification allows us to provide a functional correctness proof of the producer-consumer example shown in §3.3. It can derive the CAP-style specification, which is in the technical report [29].

The specification uses two abstract predicates: Key and Collect. Both predicates are parametrised with s and x. The former abstracts implementation-defined invariant information, and the latter is the physical address identifying the map. When v is not 0, the predicate $\mathsf{Key}(s, x, k, v)$ represents a mapping from key k to value v in the map. When $v = 0$, it states that the key k is not in the map. The predicate $\mathsf{Collect}(s, x, \mathcal{S})$ keeps track of how many Key predicates are in use: one for each element of $\mathcal{S}$. Initially, the set $\mathcal{S}$ is empty. The axiom at the bottom of Fig. 4 allows constructing new Key predicates by increasing the size of the set tracked by Collect.

In order to verify the specification with respect to the map specification in Fig. 1, we introduce a shared region with type **KVMap**. The interpretation of the region is defined to be a façade of the concurrent map predicate as follows:

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

We associate the region with guards $\mathrm{K}(k)$ and $\textsc{Collect}(\mathcal{S})$. The guard $\mathrm{K}(k)$ grants the capability to insert and remove mappings with the key k according to the following transition system defined for the region:

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

The guard $\textsc{Collect}(\mathcal{S})$ is used to track how many guards K are in use. We impose the following guard algebra:

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

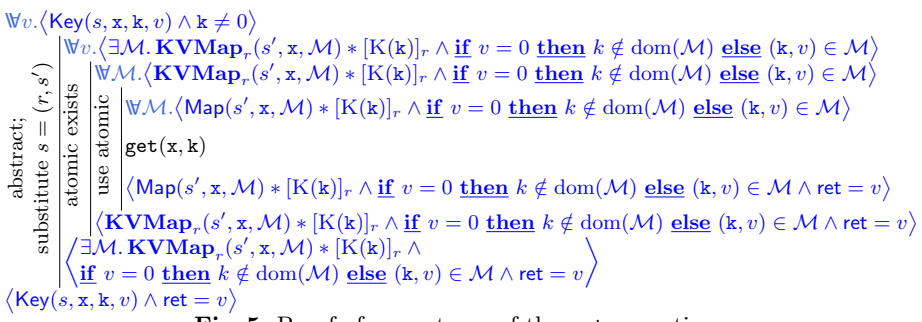


Fig. 5. Proof of correctness of the `get` operation.

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

Now we define the interpretation of the abstract predicates as follows:

$$\begin{aligned}
\mathbb{T}_3 &\stackrel{\text{def}}{=} \mathsf{RId} \times \mathbb{T}_1 \\
\mathsf{Collect}((r,s'),x,\mathcal{S}) &\stackrel{\text{def}}{=} \exists \mathcal{M}.\, \mathbf{KVMap}_r(s',x,\mathcal{M}) * [\textsc{Collect}(\mathcal{S})]_r \wedge \mathrm{dom}(\mathcal{M}) \subseteq \mathcal{S} \\
\mathsf{Key}((r,s'),x,k,v) &\stackrel{\text{def}}{=} \exists \mathcal{M}.\, \mathbf{KVMap}_r(s',x,\mathcal{M}) * [\mathrm{K}(k)]_r \\
&\quad \wedge \underline{\textbf{if}}\ v = 0\ \underline{\textbf{then}}\ k \notin \mathrm{dom}(\mathcal{M})\ \underline{\textbf{else}}\ (k,v) \in \mathcal{M}
\end{aligned}$$

The abstract predicate $\mathsf{Collect}((r,s'),x,\mathcal{S})$ is defined so that the **KVMap** region contains a map $\mathcal{M}$, the keys of which are a subset of $\mathcal{S}$. Through the guard $\textsc{Collect}(\mathcal{S})$, the set $\mathcal{S}$ tracks the keys that are currently in use. The abstract predicate $\mathsf{Key}((r,s'),x,k,v)$ is defined such that the **KVMap** region contains a map $\mathcal{M}$, in which a mapping for key k either exists or not, depending on the value of v, with the guard $\mathrm{K}(k)$, granting the capability to modify the mapping. The client axiom shown in Fig. 4 follows directly from the guard equivalence and the interpretation of the abstract predicates.

In Fig. 5 we prove that the `get` operation satisfies its key-value specification given in Fig. 4. The proof in Fig. 5 begins by substituting the abstract predicate $\mathsf{Key}(s,x,k,v)$ with its interpretation, where we also substitute the logical parameter $s \in \mathbb{T}_3$ with its interpretation as the pair (r,s'), where r is the identifier of the **KVMap** region, and s' is the abstract logical parameter of the underlying Map predicate. From there we use the atomic exists rule to eliminate the existential quantification of the abstract map $\mathcal{M}$ to a pseudo-quantification which allows $\mathcal{M}$ to be changed by the environment up to the point where `get` atomically takes effect. To justify the atomic read on the **KVMap** region, we make use of the *use atomic* TaDA proof rule discussed below.

Proof rule The *use atomic* proof rule of TaDA allows us to justify an atomic update on the abstract state of a shared region by an atomic update on the region's interpretation. A simplified form of this rule is as follows:

$$\frac{\{(x,y) \mid x \in X, y \in f(x)\} \subseteq \mathcal{T}_{\mathbf{t}}(\mathrm{G})^* \qquad \vdash \mathbb{V}x \in X.\langle I(\mathbf{t}_r(\vec{z},x)) * [\mathrm{G}]_r\rangle\ \mathbb{C}\ \langle \exists y \in f(x).\, I(\mathbf{t}_r(\vec{z},y)) * [\mathrm{G}]_r\rangle}{\vdash \mathbb{V}x \in X.\langle \mathbf{t}_r(\vec{z},x) * [\mathrm{G}]_r\rangle\ \mathbb{C}\ \langle \exists y \in f(x).\, \mathbf{t}_r(\vec{z},y) * [\mathrm{G}]_r\rangle}$$

Similarly to the *make atomic* rule, in order to justify the atomic update on region r of type $\mathtt{t}$ from abstract state x to abstract state $y \in f(x)$, the precondition is required to include a guard G for which this update is allowed by the region's transition system according to the first premiss.

We use the *use atomic* rule in the last step in the proof in Fig. 5 to justify the atomic read of the key `k` by the atomic map specification of the `get` operation in Fig. 1. The key-value based specifications for the rest of the concurrent map operations are proven similarly.

Rebuilding the Atomic Map Specification So far, we have started with a concurrent map specification in terms of the entire map, from which we then justified a key-value based specification. We now show that the opposite direction, of starting from a key-value based specification and then deriving a whole-map specification, is also possible. First, we introduce a region type **NewMap**. We interpret this region as all the key-values pairs in use:

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

where $\circledast$ denotes the iteration of $*$. The abstract state $\mathcal{M}$ includes all key-value mappings that exist in the concurrent map, as well as keys that do not exist but are in use in the form of Key predicates. Recall that if the fourth parameter of the Key predicate is 0, the corresponding key does not exist in the concurrent map. Also, those Key are allocated through $\mathsf{Collect}$, their key fields are in the set $\mathcal{K}$, which is the reason of $\mathrm{dom}(\mathcal{M}) \subseteq \mathcal{K}$.

The labelled transition system for this region is defined as follows:

$$\textsc{Nmap} : \forall \mathcal{M}, k, v.\, \mathcal{M} \rightsquigarrow \mathcal{M}[k \mapsto v]$$
$$\textsc{Nmap} : \forall \mathcal{M}, k.\, \mathcal{M} \rightsquigarrow \mathcal{M} \setminus \{(k, \mathcal{M}(k))\}$$

where the guard algebra is defined so that the composition $\textsc{Nmap} \bullet \textsc{Nmap}$ is undefined. This guarantees the uniqueness of the $\textsc{Nmap}$ guard.

We can now give an alternative implementation of the abstract predicate Map in terms of the **NewMap** region and the $\textsc{Nmap}$ guard:

$$\mathsf{Map}((r,s'),x,\mathcal{M}) \stackrel{\text{def}}{=} \mathbf{NewMap}_r(s',x,\mathcal{M}) * [\textsc{Nmap}]_r$$

By starting from the key-value based concurrent map specification in Fig. 4 we can justify the whole-map specification in Fig. 1, similarly to the proof in Fig. 5.

We have shown that the whole-map specification and key-value based specification are equivalent in the sense that they specify the same structure in two different ways. Clients are free to pick the specification that suits them best.

3.3 Producer-Consumer

We now consider a simplified producer-consumer example that uses the key-value specification. Sergey *et al.* [25] proved a producer-consumer example, but here

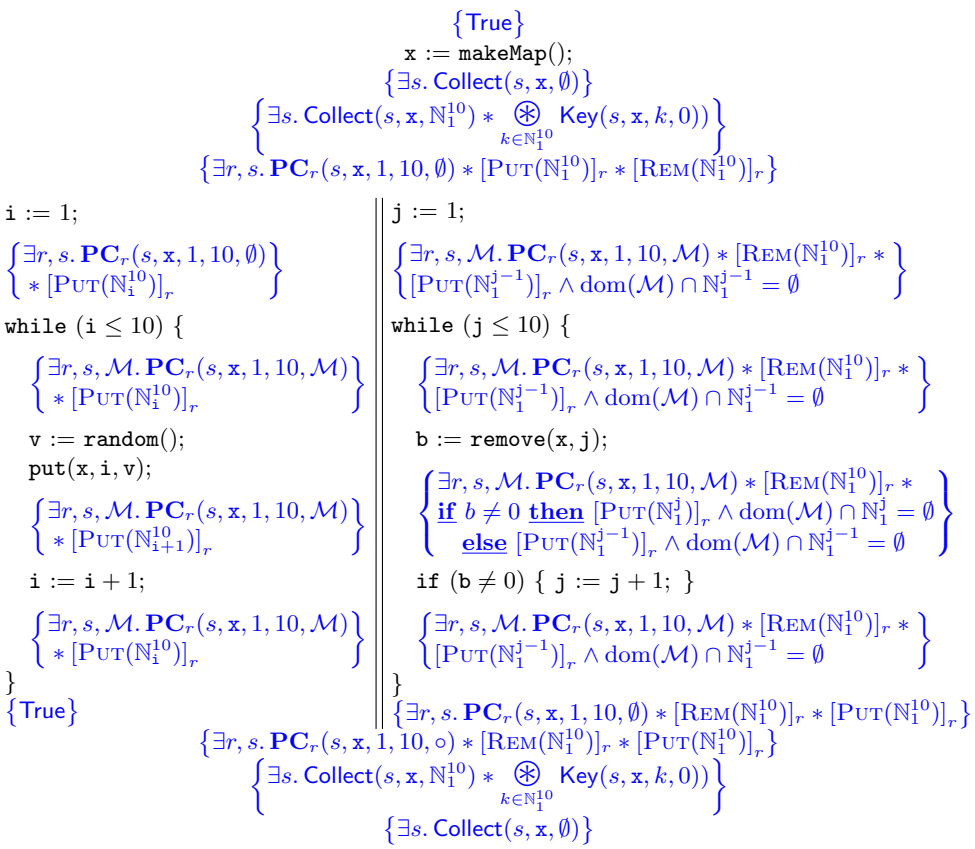


Fig. 6. Producer-consumer proof.

we only mean to use the producer-consumer to explain the improvement of our specification for concurrent map. 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. By using TaDA's guards and shared regions, we can prove that, at the end, the map is empty. We contrast this with the CAP-style approach from [5], where we cannot know anything about the state of the map after both threads finish processing. Note that our programming language and TaDA support dynamic creation of threads. We use the parallel composition ($\|$) in this example to simplify the presentation.

Shared Region, Transition System and Guards In order to verify the program, we introduce a region type **PC** that encapsulates the shared key-value mappings ranging from l to u. This region is parametrised by the address of the map x and mappings $\mathcal{M}$. Moreover, we introduce a distinguished abstract state $\circ$ to represent the state where the region is no longer required. This is the state we reach after both threads return. Before showing the interpretation of

the region, we define the transition system for the **PC** region, as follows:

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

where $\mathbb{N}_l^u$ denotes $\{k \mid k \in \mathbb{N} \wedge l \leq k \leq u\}$. The guard $\textsc{Put}(\mathcal{S})$ allows a thread to insert any element whose key is in $\mathcal{S}$ to the map, while $\textsc{Rem}(\mathbb{N}_l^u)$ allows the removal of a key-value mapping from the map. The composition $\textsc{Put}(\mathbb{N}_l^u) \bullet \textsc{Rem}(\mathbb{N}_l^u)$ allows the map to transition from a state $\mathcal{M}$ to $\circ$, where the region is no longer needed.

The guard algebra allows Put guards to be combined according to the following equivalence:

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

Additionally, the $\bullet$ operator does not allow the following compositions:

$$\textsc{Rem}(-) \bullet \textsc{Rem}(-) \qquad \textsc{Put}(\mathcal{S}) \bullet \textsc{Put}(\mathcal{S}') \text{ if } \mathcal{S} \cap \mathcal{S}' \neq \emptyset$$

This guarantees that there is only one $\textsc{Rem}(-)$ guard and one guard of type Put for each key.

We can now define the interpretation of the region states:

$$\begin{aligned} I(\mathbf{PC}_r(s,x,l,u,\mathcal{M})) &\stackrel{\text{def}}{=} ([\textsc{Put}(\mathrm{dom}(\mathcal{M}))]_r \wedge \mathrm{dom}(\mathcal{M}) \subseteq \mathbb{N}_l^u) * \\ &\qquad \circledast_{k \in \mathbb{N}_l^u} (\mathsf{Key}(s,x,k,v) \wedge \underline{\textbf{if}}\ k \in \mathrm{dom}(\mathcal{M})\ \underline{\textbf{then}}\ v \neq 0\ \underline{\textbf{else}}\ v = 0) \\ I(\mathbf{PC}_r(s,x,l,u,\circ)) &\stackrel{\text{def}}{=} \mathsf{True} \end{aligned}$$

where l and u are immutable parameters and indicate the range of keys. The **PC** region requires 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 allows the insertion. 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.

After the constructing the map with `makeMap`, the main thread owns the keys in the range 1 to 10 used subsequently by the producer and consumer threads. To allow for the keys to be shared, we create an instance of the **PC** region encapsulating those keys, which is done by the *create region* rule. Given that the producer and consumer work on key range from 1 to 10, the **PC** region is parametrised by 1 and 10.

Proof Rule In the original TaDA, the allocation of shared regions was deferred to the semantic model. In contrast, we introduce a new proof rule that enables the creation of a shared region in the logic, using view shifts [9]. The rule to create a new shared region, of type $\mathbf{t}$ at state x, is as follows:

$$\frac{\mathrm{G} \in \mathcal{G}_{\mathbf{t}} \quad \forall r.\, P * [\mathrm{G}]_r \preceq I(\mathbf{t}_r(z,x)) * Q(r)}{P \preceq \exists r.\, \mathbf{t}_r(z,x) * Q(r)}$$

The first premiss ensures that the guard G must be part of the guard algebra of the region type $\mathbf{t}$. The second premiss states that if the guard G is compatible with P for a region identifier r, then we must be able to satisfy the interpretation of the region at state x and some frame $Q(r)$. The guard G may be split to satisfy the interpretation of the region at state x and $Q(r)$. All the other resources, such as heap resources or guards from other regions, required to satisfy the interpretation must be already in P. Note that the premiss must hold for any fresh region identifier r, as we do not decide which particular identifier will be used to create the region. The conclusion of the rule specifies that we can allocate a new region and the required guard G, where r is a fresh region identifier. Note that forgetting a region can be done by logical implication, because TaDA is an intuitionistic logic. We can see the use of creating a region and forgetting a region at the beginning and the end of the proof shown in Fig. 6.

In the proof shown in Fig. 6, after creating the region, we split the $[\textsc{Put}]_r$ and $[\textsc{Rem}]_r$ guards between the producer and consumer threads, respectively, using the standard parallel composition rule of concurrent separation logic [21]. We implicitly use the *use atomic* rule for the `put` and `remove` operations in each thread. The left thread, after performing the `put` operation, must release the corresponding guard $[\textsc{Put}(\{\mathtt{i}\})]_r$ back into the region, so that the invariant of the region is maintained. The right thread, if the `remove` operation has successfully removed the key `j` from the map, takes the corresponding $[\textsc{Put}(\{\mathtt{j}\})]_r$ from the region. This describes the ownership transfer of key-value pairs from the left thread to the right. At the end, we transfer the state of **PC** from $\emptyset$ to $\circ$ and forget this region. Therefore, the Key predicates for the keys in the example range have value 0, meaning that they do not exist in the map.

Contrast the producer-consumer proof with one which uses the CAP-style approach previously summarised in §3.2. According to Table 1, the only protocol tag that allows threads to both insert and remove keys from the map is `unk`. However, the specifications of the map operations for this protocol in the CAP-approach are too weak, losing all information about the value associated with the key or even its existence in the map:

$$\begin{array}{ccc} \{\mathsf{CAPKey}_{\mathtt{unk}}(\mathtt{x},\mathtt{k})_\pi\} & \mathtt{put}(\mathtt{x},\mathtt{k},\mathtt{v}) & \{\mathsf{CAPKey}_{\mathtt{unk}}(\mathtt{x},\mathtt{k})_\pi\} \\ \{\mathsf{CAPKey}_{\mathtt{unk}}(\mathtt{x},\mathtt{k})_\pi\} & \mathtt{remove}(\mathtt{x},\mathtt{k}) & \{\mathsf{CAPKey}_{\mathtt{unk}}(\mathtt{x},\mathtt{k})_\pi\} \end{array}$$

Thus, the postcondition derived can only establish memory safety.

By hard-coding predetermined protocols according to which multiple threads can access a data structure, CAP-style specifications restrict how clients can concurrently access the data structure whilst retaining strong information about the shared state. It is possible to amend the original specification with a protocol that fits a particular client, such as the producer-consumer example. However, this would break modularity as it requires the same implementation to be re-proved with respect to each additional protocol. In effect, if we want to reason about strong functional properties of arbitrary clients with the CAP-style approach, we are unable to completely forget the implementation details by using a general abstract specification for the data structure.

In contrast, atomic specifications, such as those in Fig. 4, are decoupled from the clients' use cases. Clients are free to define the protocols according to which the data structure is accessed, by selecting the guards, guard algebras and state transition systems. Therefore, atomic triples in the style of TaDA lead to general specifications and modular reasoning.

4 Implementation: Concurrent Skiplist

In §2, we have introduced the abstract specification for the `ConcurrentMap`. One of its implementations is the `ConcurrentSkipListMap` of `java.util.concurrent`. We extract the main structure of the `ConcurrentSkipListMap` implementation (OpenJDK 8) and prove that its operations satisfy the atomic specification previously shown in Fig. 1. This is one of the largest examples available in the `java.util.concurrent` library and highlights the scalability of modern concurrent separation logics.

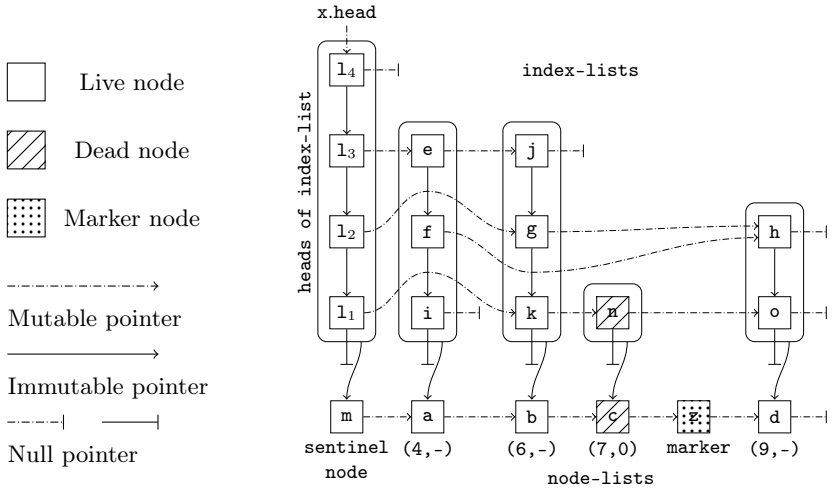
Data Structure A skiplist is built from layers of linked lists. The bottom layer is a linked list of key-value pairs, stored in key order. Each higher layer is a linked list that acts as an index to the layer below it, with approximately half the number of nodes of the lower layer. This results in obtaining a fast search performance, comparable to that of B-trees [23].

Figure 7a depicts a snapshot of the Java concurrent skiplist. The linked list at the bottom layer is the *node-list*. The first node of the node-list is always the *sentinel node*; for example, the node `m` in Fig. 7a. Nodes after the sentinel node store key-value pairs of the map in key order, where keys are immutable. A *dead node* is a node whose value is 0 and a *marker node*, or a *marker*, is a node whose value is a pointer to itself.[2] Markers are used to ensure the correct removal of dead nodes from the node-list, as we will explain shortly.

Each layer above the node-list is an *index-list*. An index-list node stores three pointers as its value: (a) a right-pointer to the next index-list node, (b) an immutable down-pointer to a node in the lower layer (the down-pointers from the lowest layer point to `null`) and (c) an immutable node-pointer to a node from the node-list. The key of each index-list node is the key of the node-list node to which it points, and is also immutable. All index-list nodes connected by their down-pointers should point to the same node-list node. For instance, in Fig. 7a, the index-list nodes `e`, `f`, and `i` are grouped vertically because they all point to the node-list node `a`. The first node of an index-list, also called the *head node*, additionally stores the level of the index-list, which is an immutable field. All head nodes point to the sentinel node, as shown in Fig. 7a.

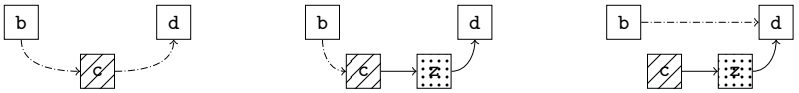
Algorithm A search for a key, the `get` operation, proceeds from the top index-list and from its left to right, until the current key is the greatest key lower than

[2] In Java, the value of a non-marker node is a pointer to its concrete value. For simplicity, here we assume that non-marker nodes directly store concrete values and there is no clash between concrete values and pointers.



(a) A snapshot of a concurrent skiplist.

(b) Link a new node `a`: set the next pointer of new node `a` to `b` and then link it into the node-list by `CAS`, and similarly for linking a new index-list node.



(c) Unlink node `c`: append a marker `z` after node `c` by `CAS`, then unlink the dead node and the marker also by `CAS`. The links between `c` and `z`, and `z` and `d` are immutable.

Fig. 7. Concurrent skiplist.

or equal to the target key. If the key is equal, the key has been found. Otherwise, the search moves down to the lower layer through the down-pointer. The search stops if the key has been found or if all layers have been traversed.

The `put` operation searches for the given key first. If the key is found, it replaces the associated value with the new value by performing an atomic *compare-and-set* (`CAS`). Otherwise, if the key is not found, a new node is added to the node-list as the successor of the node with the greatest key lower than the new node key. To do so, a new node with the key-value pair is created, and its next-pointer field is set to its predecessor's successor. Then, this new node is linked into the node-list by a `CAS`. This is shown in Fig. 7b.

Adding a node-list node also involves adding some index-list nodes as well, in order to maintain the performance. A column of index-list nodes is created first, and then they are linked into the corresponding layer from top to bottom by the process shown in Fig. 7b. Due to interference from the environment, a

Table 2. Auxiliary predicates.

Predicate	Meaning	Example in Fig. 7a
$\mathsf{node}(x, k, v, n)$	A node-list node at the address x with a key-value pair (k, v) and its next node n.	$\mathsf{node}(\mathtt{a}, 4, \_, \mathtt{b})$
$\mathsf{index}(x, p, d, r)$	An index-list node at the address x with three pointers, a node-pointer p, a down-pointer d and a right-pointer r.	$\mathsf{index}(\mathtt{f}, \mathtt{a}, \mathtt{i}, \mathtt{h})$
$\mathsf{head}(x, l, p, d, r)$	A head node at level l at the address x with p, d and r 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 node at the address x and its next node n.	$\mathsf{marker}(\mathtt{z}, \mathtt{d})$
$x \rightsquigarrow^* y$	The node x reaches y in the same layer.	$\mathtt{a} \rightsquigarrow^* \mathtt{c}$
$x \rightsquigarrow^* \mathcal{N}$	The node x reaches one node in the set $\mathcal{N}$.	$\mathtt{m} \rightsquigarrow \{\mathtt{a}, \mathtt{b}\}$

situation such as that for the `f`, `g` and `h` nodes in Fig. 7a may appear. In this case, the right-pointer of `f` is first set to `g`, and the `CAS` is performed again.

To delete a key-value mapping, the `remove` operation `CAS` the value to 0, turning the node, and all index-list nodes pointing to it, into dead nodes. Dead nodes are unlinked by subsequent key searches, and then left for the garbage collector. To unlink a dead index-list node, the right-pointer of its predecessor is `CAS`sed to its successor. However, unlinking a dead node-list node is subtle, because a new node may be concurrently added to the position after the dead node [13]. To prevent this, a marker node is added by `CAS` as the successor to the dead node first, which stops other threads from modifying the structure between the marker and its successor. Then, the marker and the dead node are unlinked by doing another `CAS`. Figure 7c illustrates unlinking the node-list node `c`.

Shared Region, Transition System and Guards In Table 2, we introduce predicates for the various skiplist node types and two reachability predicates. They are formally defined in the technical report [29].

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

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

where the `.head` denotes the offset 0.

The predicate iLists describes the index-lists and it is defined as follows:

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

where $\downarrow_i$ denotes the i-th projection and $+\!\!+$ denotes list concatenation. The $\mathcal{P}$ is a list of tuples. For each tuple, the first element is an address of a node-list node

and the second element is a list of addresses of index-list nodes. The predicate $\mathsf{wellForm}(\mathcal{P})$ asserts that each tuple in $\mathcal{P}$ is a grouped column and the node it points to in Fig. 7a. It is also asserts that all tuples are in key order. In the end, this gives us $\mathcal{P} = [(\mathtt{a}, [\mathtt{i}, \mathtt{f}, \mathtt{e}]), (\mathtt{b}, [\mathtt{k}, \mathtt{g}, \mathtt{j}]), (\mathtt{c}, [\mathtt{n}]), (\mathtt{d}, [\mathtt{o}, \mathtt{h}])]$ in Fig. 7a. The predicate rows describes the concrete heap structure of all index-lists. Note that TaDA is intuitionistic logic, so $P \wedge Q$ can assert overlapping resources.

The predicate $\mathsf{wellForm}$ and an auxiliary predicate chain are defined as follows:

$$\begin{aligned}
\mathsf{wellForm}(\mathcal{P}) &\stackrel{\text{def}}{=} \mathcal{P} = [] \vee \exists p, k, \mathcal{I}, i, \mathcal{I}', \mathcal{P}'. \\
&\quad \begin{pmatrix} \mathcal{P} = [(p, \mathcal{D})] +\!\!+ \mathcal{P}' \wedge \mathsf{wellForm}(\mathcal{P}') \wedge \mathsf{node}(p, k, _, _) \wedge \mathsf{chain}(p, d, \mathcal{D}') \wedge \\ \bigwedge_{p' \in \mathcal{P}'\downarrow_1} (\exists k'.\, \mathsf{node}(p', k', _, _) \wedge k < k') \wedge \mathcal{D} = \mathcal{D}' +\!\!+ [d] \end{pmatrix} \\
\mathsf{chain}(p, d, \mathcal{D}) &\stackrel{\text{def}}{=} (\mathcal{D} = [] \wedge d = 0) \vee \\
&\quad (\exists d', \mathcal{D}'.\, \mathcal{D} = \mathcal{D}' +\!\!+ [d'] \wedge \mathsf{index}(d, p, d', _) * \mathsf{chain}(p, d', \mathcal{D}'))
\end{aligned}$$

where $\bigwedge$ denotes iteration of conjunct. The predicate $\mathsf{wellForm}$ describes that the order of the tuples in the list $\mathcal{P}$ is the order of the first elements, by asserting that the key of the node p smaller than others in $\mathcal{P}'$ (the first conjunct in the third line). The predicate chain asserts that the second element of each tuple in $\mathcal{P}$, denoted by $\mathcal{D}$, represents a linked list of index-list nodes that point to the same node-list node p.

The predicate rows describes the index-list at level l and all below:

$$\begin{aligned}
\mathsf{rows}(m, \mathcal{H}, \mathcal{P}, l) &\stackrel{\text{def}}{=} (l = 0 \wedge \mathcal{H}(0) = 0) \vee \\
&\quad \begin{pmatrix} \exists i, \mathcal{I}.\, \mathsf{head}(\mathcal{H}(l), l, m, \mathcal{H}(l-1), i) * \\ \mathcal{P}\downarrow_2\downarrow_l = [i] +\!\!+ \mathcal{I} \wedge \mathsf{iTail}(i, \mathcal{I}) * \mathsf{rows}(m, \mathcal{H}, \mathcal{P}, l-1) \end{pmatrix} \\
\mathsf{iTail}(i, \mathcal{I}) &\stackrel{\text{def}}{=} (\mathcal{I} = \emptyset \wedge i = 0) \vee \exists i', i'', \mathcal{I}'. \\
&\quad (\mathcal{I} = [i'] +\!\!+ \mathcal{I}' \wedge i'' \in \mathcal{I} \uplus \{0\} \wedge \mathsf{index}(i, _, _, i'') * \mathsf{iTail}(i', \mathcal{I}'))
\end{aligned}$$

where $\mathcal{H}$ represents a list of head nodes, and $\mathcal{H}(l)$ denotes the l-th element of the list. An index-list is a linked list that starts with a head node followed by the rest index-list nodes. Note that the head node, described by the predicate head, must point to the sentinel node m. The notation $\mathcal{P}\downarrow_2$ denotes the second projection of $\mathcal{P}$ that is a list in which each element is a list of index-list nodes. Therefore, $\mathcal{P}\downarrow_2\downarrow_l$ denotes a list of index-nodes at level l, e.g. in Fig. 7a, $\mathcal{P}\downarrow_2\downarrow_2 = [\mathtt{f}, \mathtt{g}, \mathtt{h}]$. Due to the situation shown in Fig. 7a, the predicate iTail asserts a quasi list starting with i, where each node points to an index-list note with larger key, but this quasi list is not strictly ordered.

The predicate nList describes the node-list of the skiplist:

$$\begin{aligned}
\mathsf{nList}(m, \mathcal{M}, \mathcal{N}, \mathcal{B}) &\stackrel{\text{def}}{=} \exists n, \mathcal{L}.\, \mathsf{node}(m, _, _, n) * \mathit{toList}(\mathcal{M}) = [n] +\!\!+ \mathcal{L} \wedge \\
&\qquad \mathsf{nTail}(n, \mathcal{L}, \mathcal{N}, \mathcal{B}) \wedge \bigwedge_{n \in \mathcal{B}} (n \rightsquigarrow^* \mathcal{N} \uplus \{0\}) \\
\mathit{toList}(\mathcal{M}) &\stackrel{\text{def}}{=} \begin{cases} [] & \mathcal{M} = \emptyset \\ [(k, \mathcal{M}(k))] +\!\!+ \mathit{toList}(\mathcal{M} \setminus \{(k, \mathcal{M}(k))\}) & \mathit{otherwise} \\ \qquad \text{where } k = \min(\mathrm{dom}(\mathcal{M})) & \end{cases}
\end{aligned}$$

The node-list starts with the sentinel node m, followed by the rest described by the predicate nTail. Note that the tail also includes some dead nodes and markers

pending unlinking. The function *toList* converts the partial function from keys to values, to a list of key-value pairs in the order of ascending keys. The last conjunct of the predicate nList asserts that all dead nodes must eventually reach a live node or null. The predicate nTail is defined as follows:

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

where the second disjunct asserts a live node and the third disjunct asserts a dead node and a potentially marker.

Finally, The predicate $\mathsf{node}(x,k,v,n)$ asserts a node-list node at the address x that stores the key-value pair (k,v) and the next-pointer n. The predicate $\mathsf{marker}(x,n)$ asserts a marker. They are interpreted as concrete heap cells:

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

where the field notation $E.\mathtt{field}$ is 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:

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

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

Given the new region, guards and transition system, we instantiate the abstract map predicate and the abstract type:

$$\mathbb{T}_1 \stackrel{\text{def}}{=} \mathsf{RId} \times \mathsf{Loc} \qquad \mathsf{Map}((r,m),x,\mathcal{M}) \stackrel{\text{def}}{=} \exists \mathcal{N},\mathcal{B}.\, \mathbf{SLMap}_r(x,\mathcal{M},\mathcal{N},\mathcal{B},m) * [\mathrm{L}]_r$$

Code of the `put` Operation The implementation, shown in Fig. 8, is given in a simple imperative programming language, following the structure of the `ConcurrentSkiplistMap` implementation found in `java.util.concurrent` (OpenJDK 8). In the Java implementation, shared fields are declared as either `final` field, i.e. immutable field, or `volatile` field, i.e. no caching or reordering, so that there is no weak memory behaviour[3]. In the language we use, the semantics of single heap cell read and write are equivalent to those of `volatile` fields in Java. We use the `outer`, `inner` and `continue` variables to emulate the control

[3] More detail in JSR 133 (Java Memory Model)

flow commands `break` and `continue`. We introduce variables to record intermediate values to eliminate the side-effects of boolean expressions. This algorithm assumes a garbage-collector as it only unlinks dead nodes.

The `put` operation has two layer of loops. The inner loop traverses the node-list until finding a node with the target key `k`, or the processor where a new node will be added. If it observes any unexpected state due to the interference from the environment, for example dead nodes, the inner loop stops and jumps to the beginning of the outer loop. If a new node has been successfully added, it jumps to the end outer loop and then adds index-list nodes to preserve the performance. The code and proofs of `buildIndexChain` and `insertIndexChain` are present in the technical report [29]. These operations, which are opaque to clients, preserve the invariant of the skiplist.

Proof of the `put` Operation We present the sketch proof that `put` satisfies the map specification in Fig. 1. A detail version is in the technical report [29]. For brevity, we use $\top$ and $\bot$ to denote the boolean value `true` and `false` respectively. We use the predicate flow to describe the control flow within the outer loop.

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

The predicate $\mathsf{wkNdReach}$ used in the sketch proof asserts that either x reaches 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 relations.

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

The node `b` is the sentinel node m, or its key is smaller than the target key `k`.

The predicate $\mathsf{restart}$ describes the cases when the algorithm needs to restart: (a) `CAS` fails; (b) node `n` is dead or a marker; (c) node `b` is dead; (d) `n2`, which is the second read of the successor of node `b`, is inconsistent with `n`.

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

Additional specifications and proofs of auxiliary operations used in the proof of `put`, such as `CASValue` and `findPredecessor`, are given in the technical report [29]. We also prove that the skiplist implementation of the `remove` operation satisfies the specification of Fig. 1 in the same report.

5 Related Work

There is much recent work on modular specification of concurrent libraries using concurrent separation logic [6,16,26,17,27,20]. Here we focus on related work which uses separation logic to either reason about concurrent Java programs and `java.util.concurrent`, or reason about concurrent sets and maps.

Amighi *et al.* [1] used concurrent separation logic with permissions to reason about several lock modules in `java.util.concurrent` and their clients. Their

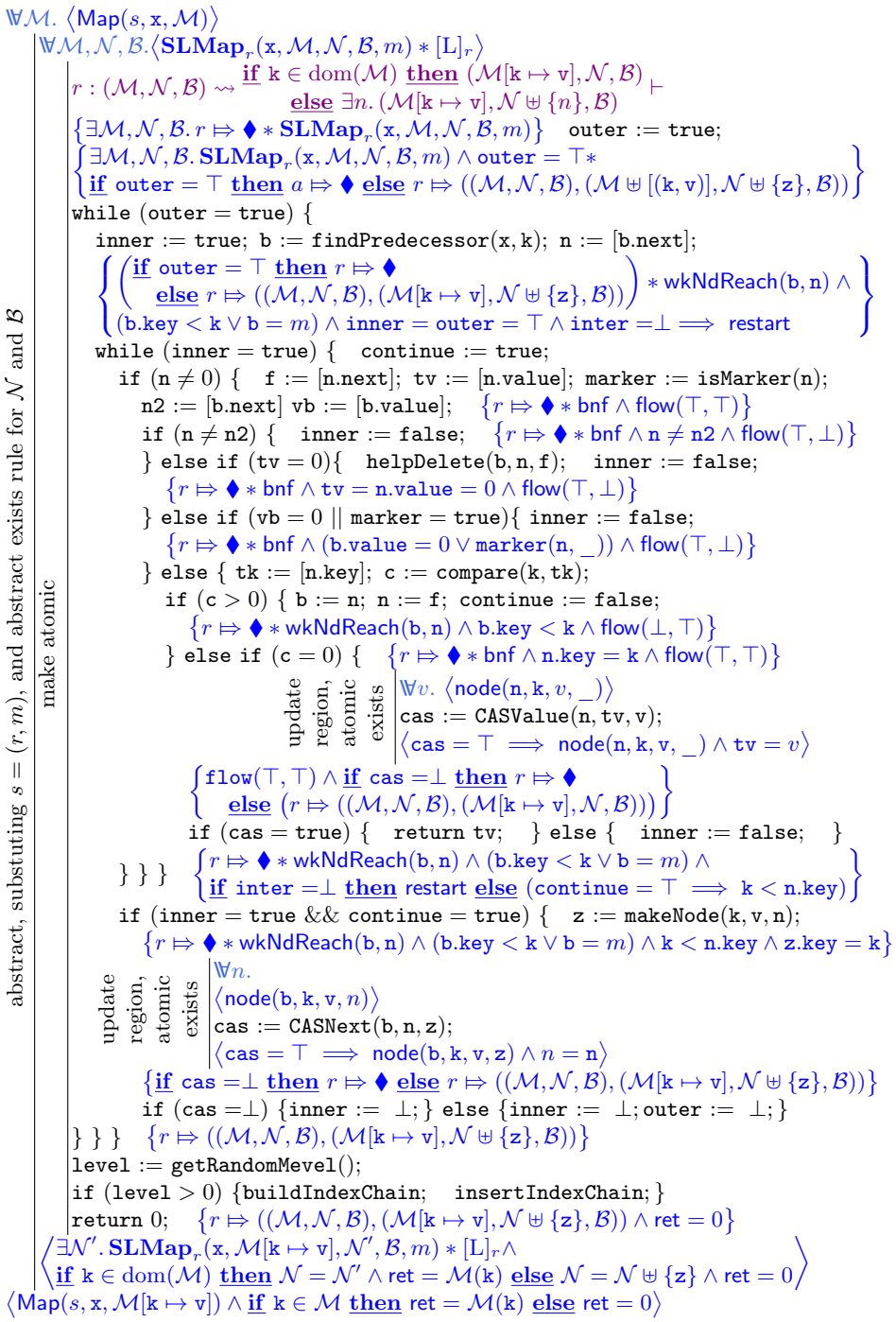


Fig. 8. Skiplist's `put` proof.

approach is modular, but is not general enough to prove strong functional properties about arbitrary clients.

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 abstracts the values in the set, losing information about the exact state of the set, which limits 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 [10,12]. Liang and Feng [19] and separately O'Hearn *et al.* [22] have reasoned about Heller's lazy set [14], and the latter is not able to reason about clients. Jacobs *et al.* [16] has given an atomic specification of a concurrent set using higher-order reasoning and proved that a simple course-grained linked-list satisfies the specification. None of this work is aimed at `java.util.concurrent`.

In this paper, using TaDA [6], 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 with separation logic about concurrent maps. Da Rocha Pinto *et. al.* use CAP [10] to develop map specifications that allow thread-local reasoning combined with races over elements of the data structure [5]. They prove several map implementations, including Sagiv's B^{Link} Tree algorithm [24]. They 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, they can only conclude the set of possibilities at the end of the execution. The closest specification to our work is a map specification using Total-TaDA [8], which was used to verify an implementation of a map based on a linked list in a total-correctness setting. We improve upon this specification by moving to a more abstract specification, which was possible due to the way in which we extended TaDA.

6 Conclusions

We have given an abstract map specification that captures the atomicity intended in the `java.util.concurrent` English specification [18]. We have used it to specify a concurrent set module that makes use of the map internally. We have shown how to build a key-value specification on top of the map specification, to present clients with an alternative view of the data structure, and demonstrated that these two specifications are equivalent. We have verified a functional correctness property for a simple producer-consumer client. Lastly, we have given the first formal proof of the `ConcurrentSkipListMap` and shown that it satisfies the atomic map specification.

These results are substantially stronger than related results found in previous work [1,2,5]. They are possible due to the abstract atomicity given by the TaDA atomic triples. So far, all of the proofs using TaDA [6] have been done by hand, but despite this, our examples are comparatively substantial. Indeed, the verification of the `ConcurrentSkipListMap` is one of the most complex examples studied in the literature. This example, we believe, is at the limit of what is possible to be done with hand-written proofs.

6.1 Future Work

Tool support. We recognise the need for mechanisation and automation in order to increase the level of confidence in our proofs. We are currently working on extending Caper [11] for a fragment of TaDA. This will allow us to provide machine-checked proofs and tackle more modules of the `java.util.concurrent` package.

Object-Oriented languages. 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 such programming language features.

Acknowledgements. We thank Andrea Cerone, Petar Maksimović, Julian Sutherland and the anonymous reviewers for useful feedback. This research was supported by the EPSRC Programme Grants EP/H008373/1 and EP/K008528/1, and the Department 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., Huisman, M., Zaharieva-Stojanovski, M.: History-based verification of functional behaviour of concurrent programs. In: Proceedings of SEFM '15. pp. 84–98. LNCS (2015)
3. Bornat, R., Calcagno, C., O'Hearn, P., Parkinson, M.: Permission accounting in separation logic. In: Proceedings of POPL '05. pp. 259–270 (2005)
4. da Rocha Pinto, P.: Reasoning with Time and Data Abstractions. Ph.D. thesis, Imperial College London (2017)
5. 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)
6. 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)
7. 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)

8. 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)
9. Dinsdale-Young, T., Birkedal, L., Gardner, P., Parkinson, M., Yang, H.: Views: compositional reasoning for concurrent programs. In: POPL. pp. 287–300 (2013)
10. Dinsdale-Young, T., Dodds, M., Gardner, P., Parkinson, M.J., Vafeiadis, V.: Concurrent abstract predicates. In: Proceedings of ECOOP '10. pp. 504–528 (2010)
11. Dinsdale-Young, T., da Rocha Pinto, P., Andersen, K.J., Birkedal, L.: Caper: Automatic verification for fine-grained concurrency. In: Proceedings of ESOP '17 (2017)
12. Dodds, M., Feng, X., Parkinson, M., Vafeiadis, V.: Deny-Guarantee reasoning. In: Proceedings of ESOP '09. pp. 363–377 (2009)
13. Harris, T.L.: A pragmatic implementation of non-blocking linked-lists. In: Proceedings of DISC '01. pp. 300–314 (2001)
14. 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)
15. Herlihy, M.P., Wing, J.M.: Linearizability: A correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. 12(3), 463–492 (1990)
16. Jacobs, B., Piessens, F.: Expressive modular fine-grained concurrency specification. In: Proceedings of POPL '11. pp. 271–282 (2011)
17. 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)
18. Lea, D., et al.: Java specification request 166: Concurrency utilities (2004)
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.: Communicating state transition systems for fine-grained concurrent resources. In: Proceedings of ESOP '14. pp. 290–310 (2014)
21. OHearn, P.W.: Resources, concurrency, and local reasoning. Theor. Comput. Sci. 375(1-3), 271–307 (2007)
22. 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)
23. Pugh, W.: Skip lists: A probabilistic alternative to balanced trees. Commun. ACM 33(6), 668–676 (1990)
24. Sagiv, Y.: Concurrent operations on b-trees with overtaking. In: Proceedings of PODS '85. pp. 28–37 (1985)
25. Sergey, I., Nanevski, A., Banerjee, A.: Specifying and verifying concurrent algorithms with histories and subjectivity. In: Proceedings of ESOP '15. pp. 333–358 (2015)
26. Svendsen, K., Birkedal, L.: Impredicative concurrent abstract predicates. In: Proceedings of ESOP '14. pp. 149–168. Berlin, Heidelberg (2014)
27. Turon, A., Dreyer, D., Birkedal, L.: Unifying refinement and hoare-style reasoning in a logic for higher-order concurrency. In: Proceedings of ICFP '13. pp. 377–390 (2013)
28. Vafeiadis, V., Parkinson, M.: A marriage of Rely/Guarantee and Separation Logic. In: Proceedings of CONCUR '07. pp. 256–271 (2007)
29. Xiong, S., da Rocha Pinto, P., Ntzik, G., Gardner, P.: Abstract specifications for concurrent maps (extended version). Tech. Rep. 2017/1, Department of Computing, Imperial College London (2017), https://www.doc.ic.ac.uk/research/technicalreports/2017/#1