Verifying Hash tables in Iris

Esben Glavind Clausen, 201206032

Master's Thesis, Computer Science
June 2017
Advisor: Lars Birkedal

AARHUS
UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

Abstract

We describe and prove specifications for two implementations of hash tables using Iris, a recent separation logic framework. The first implementation is not thread-safe and supports operations for iteration. The second implementation is a concurrent implementation. We use higher-order predicates together with the invariants in Iris to achieve very general specifications that can allow either unrestricted sharing of the tables or ownership of parts of the tables.

Acknowledgments

I would like to thank my advisor Lars Birkedal for guidance throughout the entire process of creating this thesis. In addition, I would like to thank Aleš Bizjak, Ralf Jung, and Mathias Høier for discussing and providing feedback on various parts of my work during the process.

Contents

Chapter 1

Introduction

Concurrency poses major challenges for the specification and verification of programs. Shared resources often result in complex and potentially unpredictable behavior. O'Hearn [20] proposed *separation logic* as a useful tool for reasoning about concurrent programs, as it allows local reasoning about different parts of a program that use disjoint resources. This has resulted in the development of numerous concurrent separation logics such as CAP [8], TaDA [6], and Iris [15, 14, 18] just to name a few. This report will focus on Iris, a recent separation logic framework, which offers features from several other separation logics derived from a few basic primitives. Since it is a quite recent framework that is still being developed further, only a small number of specifications have actually been proven using Iris at the time of writing and many of those specifications only describe very simple programs. In this report, we aim to provide evidence that Iris is in fact suitable for proving specifications for programs of non-trivial complexity. We do so by presenting a case study, where we present and prove specifications for implementations of hash tables. We present two different implementations: a "regular" implementation, that is only safe to use in sequential programs, and a concurrent implementation, where all operations are thread-safe. The specifications and proofs are written and verified using the Coq proof assistant[1].

1.1 Iris

Iris is a concurrent separation logic framework developed as an attempt at unifying many of the features seen in various different concurrent separation logics while only introducing a small set of primitive constructions and rules. The main idea behind Iris is that *partial commutative monoids* (or the similar *resource algebras* described in §2.3) and *invariants* form a basis powerful enough to implement many advanced features seen in different concurrent separation logics. Key features of Iris include higher-order predicates, user-defined higher-order ghost state, invariants, and language agnosticism, i.e., Iris can be instantiated for a wide range of programming languages. Many constructions,

[1] The Coq code can be found at `https://github.com/esbengc/iris-hashtable`.

including invariants and Hoare triples, are defined entirely in terms of simpler constructions. Iris has been fully specified in the Coq proof assistant [12]. In addition, Krebbers, Timany, and Birkedal [17] have extended the Coq implementation of Iris with a dedicated proof mode. This proof mode treats Iris as an embedded logic in Coq, making it possible write Iris propositions in the embedded logic and prove them interactively. It offers variants of common Coq tactics such as `intros` and `apply` which allows Iris propositions to be proven similarly to how one would normally prove statements in Coq. This proof mode has been used to prove all the specifications presented in this report.

1.2 Contributions

We present a case study, where we state and proof specifications of two hash table modules. This is done using the latest version of Iris, Iris 3.0 [18], and is verified in Coq. The first module is an implementation of "regular" hash tables, that are not safe for concurrent use. This implementation supports the common operations `create`, `lookup`, `insert`, and `remove`, as well as the following two operations for iteration: `fold` and `cascade`. The specifications for this module is based on the work of Pottier [21], who verified a hash table module written in OCaml using CFML.

The second module is an implementation of concurrent hash tables that includes the common operations `create`, `lookup`, `insert`, and `remove`. The specifications use a shared higher-order predicate parameterized with a user-provided predicate, which the content of the table must satisfy at all times as an invariant. To our knowledge, this approach has not been used before for specifying concurrent data structures.

1.3 Reader assumptions and outline

The reader of this report should be familiar with separation logic. A basic understanding of Iris is preferred as well, though not strictly required, as we will introduce the most important features later in this report. The reader should also be familiar with Coq, including a basic understanding of the type system, the specification language, and commonly used elements from the standard library.

The remainder of the report is structured as follows: Chapter 2 provides a short introduction to the most important features of Iris. This introduction is intentionally kept short and only contains the bare minimum needed to understand the specifications presented in this report. In Chapter 3, we present two hash table modules and their specifications. The definitions and specifications are presented as they are written in Coq (slightly modified in some places for readability) rather than using mathematical notation, as the Coq implementation of Iris uses quite readable notations. We do not include proofs of the specifications in the report, as everything presented has been proven interactively using Coq. Finally, in Chapter 4 we sum up our experiences with Iris and present related work.

Chapter 2

Iris

Iris [15, 14, 18] is a recent higher-order concurrent separation logic framework. We will not describe Iris in full detail here. The lecture notes provided by Birkedal et al. [2] provide a basic introduction to Iris. Readers unfamiliar with Iris, and in particular those unfamiliar with separation logic and Hoare logic in general, are encouraged to read these notes. In this chapter, we will introduce some of the features that are more specific to Iris. In particular, we introduce *ghost state* and *invariants*, which are the two primary tools used for concurrent reasoning in Iris. This chapter not meant as a comprehensive documentation of these features, but rather is meant to give a general understanding of these features and how they are used. The proof rules presented have been selected because they illustrate the ideas behind the features.

2.1 Programming language

Iris is a framework, which is not bound to any specific language, but rather can be instantiated with a wide range of different languages. For this project, we will use a call-by-value λ-calculus with recursive functions, references, compare-and-swap (CAS), and fork. The syntax for the language is shown in Figure 2.1 ($\circledcirc$ ranges over common binary operators such as $+$ and $<$). The operational semantics are quite standard and therefore omitted here. One should note, however, that the memory model used consists of a single heap shared by all threads, and that the heap can store any kind of value at

$$
\begin{aligned}
e \in \mathit{Expr} ::= {} & () \mid n \mid \mathsf{false} \mid \mathsf{true} \mid l \mid x \mid (e, e) \mid \mathsf{inl}\ e \mid \mathsf{inr}\ e \mid \mathsf{rec}\ f(x) := e \mid \\
& e \circledcirc e \mid \mathsf{fst}\ e \mid \mathsf{snd}\ e \mid \mathsf{if}\ e\ \mathsf{then}\ e\ \mathsf{else}\ e \mid \\
& \mathsf{match}\ e\ \mathsf{with}\ \mathsf{inl}\ x \Rightarrow e \mid \mathsf{inr}\ x \Rightarrow e\ \mathsf{end} \mid \\
& \mathsf{ref}\ e \mid !e \mid e \leftarrow e \mid \mathsf{CAS}(e, e, e) \mid \mathsf{fork}\ e
\end{aligned}
$$

Figure 2.1: Syntax of the language

$$\text{WP-VALUE}\ \frac{\Phi(v)}{\mathsf{wp}\, v\, \{\Phi\}} \qquad \text{WP-MONO}\ \frac{\forall x. \Phi(x) \vdash \Psi(x)}{\mathsf{wp}\, e\, \{\Phi\} \vdash \mathsf{wp}\, e\, \{\Psi\}} \qquad \text{WP-FRAME}\ \frac{P * \mathsf{wp}\, e\, \{\Phi\}}{\mathsf{wp}\, e\, \{P * \Phi\}}$$

$$\text{WP-BIND}\ \frac{K \text{ is an evaluation context} \qquad \mathsf{wp}\, e\, \{v. \mathsf{wp}\, K[v]\, \{\Phi\}\}}{\mathsf{wp}\, K[e]\, \{\Phi\}} \qquad \text{WP-STORE}\ \frac{l \mapsto v * \triangleright (l \mapsto w \mathrel{-\!\!*} \Phi())}{\mathsf{wp}\, (l \leftarrow w)\, \{\Phi\}}$$

Figure 2.2: Some rules for weakest precondition

each location. The Iris proposition $l \mapsto v$ represents a heap or part of a heap, where the value v is stored at location l.

In addition to the constructs defined in Figure 2.1, we will also use the following derived constructs: Lamdas ($\lambda x.e$), let-expressions ($\mathsf{let}\ x = e\ \mathsf{in}\ e$), sequential composition ($e; e$), and parallel composition ($e||e$). These constructs are all defined in terms of the primitive constructs in a straightforward manner.

2.2 Weakest precondition, Hoare triples, and persistent propositions

Iris uses a notion called *weakest precondition* from which Hoare triples are defined. For a term e and a predicate $\Phi : Val \to iProp$ (where *iProp* is the type of propositions in Iris), $\mathsf{wp}\, e\, \{\Phi\}$ is an Iris proposition stating that all executions of the term t are safe (i.e., they do not get stuck in an irreducible state before reducing to a value) and for all executions reducing to a value v, $\Phi(v)$ holds after the execution. The resources represented by $\mathsf{wp}\, e\, \{\Phi\}$ are, as implied by the name, the weakest precondition for this to be true, i.e., it represents exactly the resources that must be available and owned at the start of the execution for the execution to be safe and for the postcondition Φ to hold at the end. In postconditions we sometimes use the notation $x.P$ as a shorthand notation for $\lambda x.P$ and $\Phi * P$ as a shorthand for $x.\Phi(x) * P$.

To prove propositions involving weakest precondition, a set of rules are available. A few of them are shown in Figure 2.2. The rules are similar to the kind of rules one would usually see for Hoare triples, except they are specified in terms of weakest preconditions[1]. Several rules other than the ones shown are available, including rules for each reduction defined by the operational semantics of the language (the figure shows WP-STORE as one of them).

[1]One other difference from what one might usually expect in such rules is the presence of the later modality $\triangleright$. It is not easy to explain the meaning of the later modality without explaining the semantic model of the Iris logic in detail, which is beyond the scope of this report. For the purpose of this report, it suffices to know that $\triangleright P$ is a strictly weaker assertion than P and that at no point in this report does it play an important role. The reader is thus free to ignore the later modality whenever it appears.

Hoare triples are defined in terms of weakest precondition as follows:

$$\{P\}\, e\, \{\Phi\} \triangleq \Box(P \mathrel{-\!\!*} \mathsf{wp}\, e\, \{\Phi\})$$

Intuitively $\{P\}\, e\, \{\Phi\}$ states that if P holds at the beginning, then any execution of e is safe and if it terminates with a value v, $\Phi(v)$ holds after termination. For readers familiar with Hoare triples in general, it may be easier to understand weakest preconditions from how they are used in defining Hoare triples. The definition makes use of the *always* modality $\Box$. The always modality can be described as follows: The proposition $\Box P$ states that P holds without asserting exclusive ownership of any resources. We call a proposition P *persistent* if it satisfies $P \vdash \Box P$. The always modality is idempotent, which implies that all propositions of the form $\Box P$ are persistent. Persistent propositions are duplicable as they do not assert any ownership over non-duplicable resources. By using the always modality in definition of Hoare triples, we ensure that a triple can be duplicated and reused as a specification. However, it also means that a proof of a triple cannot make use of any non-persistent propositions except for the precondition of the triple itself. In other words, even if we can show $Q \vdash P \mathrel{-\!\!*} \mathsf{wp}\, e\, \{\Phi\}$, we may not be able to show $Q \vdash \{P\}\, e\, \{\Phi\}$ if Q is non-persistent, unless we can do so without using Q at all. In other words, P must always be sufficient as a precondition.

2.3 Ghost state

Iris allows users to define their own resources that can be used in proofs. These resources can be decomposed into smaller pieces, which can be recombined. While these resources have no bearing on the physical state of the programs, they can describe a logical state, which can be used in proofs. This purely logical state is often called *ghost state*. The resources described by ghost state are defined using *resource algebras* (RA). The formal definition of a resource algebra is shown in Figure 2.3. The composition operator $(\cdot)$ defines how resources are composed. The core $|{-}|$ is a generalization of a unit element, as composing an element with its core yields the same element. Unlike unit elements, the core does not have to be unique for all elements, nor is every element required to have a core. The set $\mathcal{V}$ defines a set of valid elements. An invalid element can be used to represent an invalid resource or state. In addition, we define the notion of *frame-preserving updates*:

$$a \rightsquigarrow b \triangleq \forall c^? \in M^?.a \cdot c^? \in \mathcal{V} \Rightarrow b \cdot c^? \in \mathcal{V}$$

In other words, an update is frame-preserving if it does not cause any valid compositions to become invalid.

Given a RA M, an element $a \in M$, and a ghost name γ, the proposition $\boxed{a}^{\gamma}$ asserts ownership over (a part of) a resource represented by a. The name γ belongs to a countably infinite set of ghost names. Ghost assertions under different names are independent from each other and may not even be represented by the same RA. The

A *resource algebra* (RA) is a tuple $(M, \mathcal{V} \subseteq M, |-| : M \to M^?, (\cdot) : M \times M \to M)$ satisfying:

$$\forall a, b, c.\, (a \cdot b) \cdot c = a \cdot (b \cdot c) \qquad \forall a, b.\, a \cdot b = b \cdot a$$
$$\forall a.\, |a| \in M \Rightarrow |a| \cdot a = a \qquad \forall a.\, |a| \in M \Rightarrow ||a|| = |a|$$
$$\forall a, b.\, |a| \in M \land a \preccurlyeq b \Rightarrow |b| \in M \land |a| \preccurlyeq |b| \qquad \forall a, b.\, (a \cdot b) \in \mathcal{V} \Rightarrow a \in \mathcal{V}$$

where $M^? \triangleq M \uplus \{\bot\} \qquad a^? \cdot \bot \triangleq \bot \cdot a^? \triangleq a^? \qquad a \preccurlyeq b \triangleq \exists c \in M.\, b = a \cdot c$

Figure 2.3: Resource algebras

following proof rules are available for ghost state:

RES-ALLOC
$$\frac{a \in \mathcal{V}}{\vdash \Rrightarrow \exists \gamma. \boxed{a}^\gamma}$$

RES-UPDATE
$$\frac{a \rightsquigarrow b}{\boxed{a}^\gamma \vdash \Rrightarrow \boxed{b}^\gamma}$$

RES-OP
$$\boxed{a}^\gamma * \boxed{b}^\gamma \dashv\vdash \boxed{a \cdot b}^\gamma$$

These rules allow us to allocate a new ghost resource under a new name, perform frame-preserving updates, and compose and decompose resources. The rules mention the update modality $\Rrightarrow$. The intuition of the update modality is described quite well by the rule RES-UPDATE: The proposition $\Rrightarrow P$ represents resources from which P can be obtained by performing frame-preserving updates. The following two rules state that we can perform updates before and after each step of a computation:

FUP-WP
$$\Rrightarrow_{\mathcal{E}} \mathsf{wp}\, e\, \{\Phi\} \vdash \mathsf{wp}\, e\, \{\Phi\}$$

WP-FUP
$$\mathsf{wp}\, e\, \{v. \Rrightarrow_{\mathcal{E}} \Phi(v)\} \vdash \mathsf{wp}\, e\, \{\Phi\}$$

2.4 Invariants

A very useful tool in Iris when reasoning about concurrent programs is *invariants*. An invariant is a proposition, which must be satisfied at all times during execution. The proposition $\boxed{P}^\iota$ states that the proposition P holds as an invariant. Each invariant is indexed by a name ι from an infinite set of invariant names. A key property of the proposition $\boxed{P}^\iota$ is that it is persistent, which means that it can be duplicated and thus be made available to multiple threads. The following two rules describe how invariants are used:

INV-ALLOC
$$\triangleright P \vdash \Rrightarrow_{\mathcal{E}'} \exists \iota \in \mathcal{E}. \boxed{P}^\iota$$

WP-INV
$$\frac{\triangleright P \vdash \mathsf{wp}_{\mathcal{E} \setminus \{\iota\}}\, e\, \{\triangleright P * \Phi\} \qquad \mathsf{atomic(e)} \qquad \iota \in \mathcal{E}}{\boxed{P}^\iota \vdash \mathsf{wp}_{\mathcal{E}}\, e\, \{\Phi\}}$$

where $\mathsf{atomic(e)}$ means that e reduces to a value in a single step. The later modality $\triangleright$ is there mainly to ensure that the rules are sound. The reader may feel free to ignore it as it does not play an important role anywhere in this report. In WP-INV, the weakest

preconditions are annotated with masks (denoted by $\mathcal{E}$), which is a set of invariant names indicating which invariants can be opened. The mask is usually omitted from the weakest preconditions when it is not important.

The first rule INV-ALLOC allows us to allocate a new invariant if we can show that the proposition already holds. It uses the *fancy update modality* $\Rrightarrow_{\mathcal{E}}$ which, in addition to permitting frame-preserving updates like the basic update modality $\Rrightarrow$, is also used for enabling and disabling invariants. We will not cover this modality in further detail here. It can usually be stripped away when used for proving weakest preconditions.

The proposition $\boxed{P}^{\iota}$ cannot itself be used in place of P as it does not assert any ownership. It only asserts that we know that the resources exist. The second rule WP-INV allows an invariant to be opened for a single reduction step. When doing this, we take exclusive ownership of the proposition and the associated resource, thus allowing us to use it. The proposition must be reestablished in the postcondition after this single step. The name of the invariant is removed from the mask upon opening, which ensures that multiple copies of the potentially non-duplicable proposition cannot be obtained. The restriction to atomic expressions ensures that no other threads may take a step before the invariant has been reestablished in the postcondition, thus avoiding that another thread takes ownership of the proposition before the current thread has released the ownership.

Rather than work with invariant names directly, we will often work with *namespaces* instead. A namespace $\mathcal{N}$ is a sequence of natural numbers describing an infinite set of invariant names $\mathcal{N}^{\uparrow}$. The mapping from namespaces to sets of invariant names is defined such that $\mathcal{N}_1^{\uparrow} \subseteq \mathcal{N}_2^{\uparrow}$ if and only if $\mathcal{N}_2$ is a suffix of $\mathcal{N}_1$. In addition, if $m \neq n$, then $\mathcal{N}.m^{\uparrow}$ and $\mathcal{N}.n^{\uparrow}$ are disjoint, where $\mathcal{N}.m \triangleq [m] +\!\!+ N$. This allows users to pick disjoint sets of invariant names for separate parts of a program. Each part can then further decompose the set of names if said part consists of several smaller parts. This is similar to how many programming languages support nested modules.

We overload the notation for invariants as follows:

$$\boxed{P}^{\mathcal{N}} \triangleq \exists \iota \in \mathcal{N}^{\uparrow}.\boxed{P}^{\iota}$$

We also have the following updated rules for invariants:

$$\text{INV-ALLOC}\quad \triangleright P \vdash \Rrightarrow_{\mathcal{E}'} \boxed{P}^{\mathcal{N}} \qquad \text{WP-INV}\quad \frac{\triangleright P \vdash \mathsf{wp}_{\mathcal{E}\setminus\mathcal{N}^{\uparrow}}\, e\, \{\triangleright P * \Phi\} \qquad \mathsf{atomic(e)} \qquad \mathcal{N}^{\uparrow} \subseteq \mathcal{E}}{\boxed{P}^{\mathcal{N}} \vdash \mathsf{wp}_{\mathcal{E}}\, e\, \{\Phi\}}$$

2.5 Iris in Coq

A full implementation of Iris exists in Coq[12]. Everything presented in this report is implemented in Coq using the Iris implementation as a basis. Several definitions will be presented using snippets of Coq code. We here give a brief introduction to this implementation to aid the reader in understanding these snippets.

The Coq implementation of Iris introduces `iProp Σ` as the type of Iris propositions[2]. It also introduces notations for all logical constructs. Most of these notations use Unicode characters to make them look exactly like the corresponding mathematical notations: `P ∧ Q`, `P * Q`, `P -* Q`, and `∀x,P` just to give a few examples. Many of these notations overload existing notations for regular Coq propositions. To distinguish between the different interpretations, we will adopt the following convention for all code snippets presented in this report: For regular Coq propositions of type `Prop`, we will use ASCII based notations such as `forall`, and `/\`, while for Iris propositions of type `iProp Σ` we will use Unicode based notations.

The implementation also supports the notion of *pure* propositions, i.e. Coq propositions lifted into Iris. A pure proposition is written `⌜P⌝`, where `P` is a Coq proposition, i.e. `P` has type `Prop`. A pure proposition holds if the underlying Coq proposition holds and claims no ownership of any resources. As a result, all pure propositions are persistent.

In §2.1, we presented the programming language that we will use. The Iris implementation comes bundled with a formalization of this language. The type of expressions in the programming language is `expr`, while `val` denotes values in the language. Values of type `val` can be coerced into type `expr` implicitly. The formalization introduces the proposition `l ↦ v`, which asserts that the heap stores the value `v` in location `l`. It also specifies notations for objects of type `expr` and `val`. Most of these notations are quite obvious, so we will not cover them all. There are a few things worth noting, however. The first is literals, which are written as `#l`, where `l` can have type `Z`, `bool`, or the type of heap locations `loc` or it can be the unit value `()`. Another thing worth noting is the notations `SOME e` and `NONE`. These are syntactic sugar for `InjR e` and `InjL #()` respectively. Note that these notations use all capital letters unlike Coqs `option` type, which uses the constructors `Some` and `None`. Finally, it is worth noting that some notations sometimes uses keywords prepending the letter V such as `InjRV`. This indicates that it is an expression of type `val` rather than `expr`.

A weakest precondition proposition is written as `WP e @ E {{ Φ }}`, where `e` is an expression, `E` is a mask, and `Φ` is an Iris predicate of type `val -> iProp Σ`. It can also be written as `WP e @ E {{ v, P }}` in which case the postcondition is `λ v, P`. In both cases, `@ E` can be omitted, in which case the mask $\top$ is used, i.e., the mask containing all possible invariant names. The notation for Hoare triples is `{{ P }} e @ E {{ Φ }}`. Like for weakest precondition, the mask can be omitted and the postcondition can be written as `{{ v, P }}`.

The implementation supports a different kind of triples as well. These are written as follows:

```
{{{{ P }}}} e @ E {{{ x1 .. xn , RET v ; Q }}}}
```

where `x1 .. xn` are binders (i.e., variable declarations) and `v` is a value. The scope of the variables `x1 .. xn` is `v` and `Q`. As with the other variants, the mask can be omitted. It has approximately the same meaning as writing `{{ P }} e @ E {{ u, ∃ x1 .. xn, ⌜u =`

[2]The parameter `Σ` has type `gFunctors`. The precise definition of this type is beyond the scope of this report, however, it is used for specifying which resource algebras are available for ghost states.

`v¬ * Q }}` using the other notation, though the exact definition behind the notation is significantly different. Though this notation is sometimes a bit convenient, the main reason it exists is that it is defined in a way that makes it much simpler to use as a specification when writing proofs using tactics.

Finally, there is the notation for ghost state and invariants. A ghost assertion is written `own γ a`, where `γ` has type `gname`, the type of ghost names, and `a` has the carrier type of some RA. Similarly, an invariant is written `inv N P`, where `P` is the enclosed Iris proposition and `N` has type `namespace`, the type of invariant namespaces.

Chapter 3

Hash tables in Iris

A dictionary is a finite mapping from keys to values. Hash tables are an implementation of dictionaries, where the keys are equipped with a hash function, i.e., a function from keys to integers. For the implementations used here, we adopt the convention that multiple values can be stored under the same key (this is not to be confused with other implementations of hash tables, where different keys can be mapped to the same bucket, but each bucket only contain one entry per key). Performing a lookup operation on a key with multiple associated values will return the value inserted last under that key.

Presented here are two hash table modules fully specified in Iris. The first module is for sequential use only, while the second module supports concurrent use. The sequential specification is based on the specification for hash tables in CFML by Pottier [21], Both modules support the following operations: `create`, `insert`, `lookup`, and `remove`. The `remove` operation removes an element from the table and returns the removed element. The other operations do exactly what their names suggest. The sequential module additionally supports two operations for iterating through the entries: `fold` and `cascade`. The `fold` function is a variant of the similarly named functions over lists known from functional programming. It takes a function f and a value a as arguments, and returns $f\ k_n\ x_n(f\ k_{n-1}\ x_{n-1}(\ldots(f\ k_1\ x_1\ a)))$, where the (k_i, x_i)'s are the key-value pairs stored in the table. The `cascade` function returns a *cascade* [21], which is a stateless variant of *iterators* [10] as known from many programming languages, including Java. A cascade is a nullary function (i.e., a function taking the unit value `()` as its only argument) which either returns `SOME(k,x,c)`, where `(k,x)` is a key-value pair in the table and `c` is a cascade for the remaining entries in the table, or it returns `NONE` if there are no more entries to return. For a reader familiar with iterators in Java, a cascade can be seen as the `next` method of an iterator. However, rather than advancing by updating some internal state of the iterator object, the updated "state" is returned as a new function.

3.1 Model

Since hash tables rely on hash functions that can be applied to keys, both the implementations and specifications of the hash tables are parameterized with a Coq type `key` and a function `hash` that implement the type class `Hashable` shown in Figure 3.1. The type `key` should be a set of logical keys reflecting the set of program values that are valid keys. Similarly, the function `hash` must be a hash function on the `key` type. The co-domain of the hash function is the set of natural numbers. This avoid potential ambiguities that could arise due to how the modulo operator in programming languages often have semantics different from the mathematical modulo function when applied to negative numbers. The class `Hashable` contains fields relating `key` objects to program values and defining the hash function as a program function. In particular, it defines two program functions `equalf` and `hashf`, which decide equality on keys and compute the hash function respectively. This is formally specified by the fields `equal_spec` and `hash_spec`. It also contains a field `as_key`, which is a partial function from program values to objects of type `key`. The set of valid keys is defined as the values `v` for which `as_key v = Some k`. Note that `as_key` is not required to be injective for the set of valid keys. Thus multiple distinct program values may map to the same logical key. If this is the case, then an equivalence relation is induced on the set of valid keys in the language, such that each logical key defines an equivalence class. The idea of a separate type of logical keys is different from the approach used in the CFML specification [21]. Since program values in CFML are typed, it is not necessary to introduce a separate logical key type, as the set of valid keys is already defined by the key type in the programming language. The approach used here, however, allows the valid keys to be any set expressible in Coq, rather than being restricted to the types of programming languages such as OCaml. It would be possible to avoid introducing the logical `key` type if we instead required a predicate directly stating whether a value is a valid key. For the user, reasoning about a logical key type might be just as convenient, however. In any case, if the user wishes to use a predicate defining valid keys, they can usually still do so in this model, for example by defining the type `key` as a dependent pair `{ k: val | is_key k }` if `is_key` is the name of the predicate.

Since a hash table represents of a dictionary, our specifications will rely on a predicate relating a table to an abstract model of a dictionary. For this model, we will use the finite map types from the std++ library [16]. The model type we use is `map (list val)`, where `map` can be any type for which an instance of the type class `FinMap key map` exists, i. e. `map` must be a type of finite maps with `key` as the type of keys. Since we allow multiple values to be stored under each key, we use `list val` as the co-domain for the maps. We use the convention that each inserted element will be stored at the head of the list associated with the given key. To support this, we define the following operations on the model:

- `insert_val m k x` returns `m` with the value `x` inserted at the head of the list stored at `k`.
- `remove_val m k` returns `m` with the head of the list stored under key `k` removed.

```
Class Hashable Σ `{heapG Σ} (key : Type)
               `{EqDecision Key} (hash : key -> nat) :=
  {
    equalf : val;
    hashf : val;
    as_key : val -> option key;

    equal_spec (k1 k2: key) (v1 v2: val) :
      as_key v1 = Some k1 ->
      as_key v2 = Some k2 ->
      WP equalf v1 v2 {{u, ⌜u = #(bool_decide (k1 = k2))⌝}};

    hash_spec k v :
      as_key v = Some k -> WP hashf v {{u, ⌜u = #(hash k)⌝}}
  }.
```

Figure 3.1: Definition of the class `Hashable`

If no values are stored in `m` under `k`, it returns `m` unchanged.

We also need to express when two maps are equal. The `FinMap` class is defined such that two maps are equal according to Coqs built-in equality if they agree on all entries, i.e., `m1` = `m2` if `m1 !! k` = `m2 !! k` for all `k`, where the `m !! k` syntax denotes a lookup. This still leaves us with a problem: Since our co-domain is lists of values, we can represent that a table contains no values under a key `k` by using a map `m` that either maps `k` to the empty list, or does not map it to anything at all. In the former case we would have `m !! k` = `Some []`, while in the latter case, we would have `m !! k` = `None`. Therefore they would not be considered equal even though they represent the same dictionary. Our solution to this is to only allow the latter case. We consider a map well-formed if it contains no empty lists. We define the Coq predicate `table_wf`, which states exactly that. The operations `insert_val` and `remove_val` are defined such that they preserve well-formedness.

The specification in CFML [21], uses a slightly different model. Rather than using a type specifically for representing finite maps, like we do here, that specification uses regular functions of type `key -> list A` (since CFML is typed, they use a type parameter `A` to indicate the type of values in the table). In this model, unused keys can only be modelled by mapping them to empty lists. However, it also makes it necessary to explicitly state that a map is finite. In particular, it is impossible to iterate over a function. Therefore, in order to specify operations that iterate over the entries in a table, one may need a list containing all the keys that are actually mapped, i.e., the domain of the map. Thus, it must be possible to derive the domain from the map.

There are other ways in which we could have solved the problem of having multiple representations of the same tables. One way would be to define an equivalence relation

stating exactly the kind of equivalence we would like. Another way would be to define a type of non-empty lists and use that instead of regular lists. That would eliminate the problem completely. Our reason for not choosing these approaches is that we want to use definitions and lemmas available in existing Coq libraries that relate to the existing definitions. If we define a new equivalence relation, we have to show which operations and predicates on the tables are compatible with this relation, and define new versions of those that are not. If we define a new type of lists, we cannot use any of the operations and lemmas for regular lists.

One should note that the model does not mention the program values acting as keys directly. Only the logical keys of type `key` are mentioned. This means that if a value is inserted into a table under a key, and if multiple values represents the same logical key, then the information about which key was used during the insert operation will be lost. Implementations are thus allowed to forget this information as well. For instance, an implementation could choose to store only the first key being inserted from each equivalence class and reuse that key whenever subsequent insertions are done using equivalent keys. This could be a potential optimization as it could reduce the memory used by the data structure if the keys are large in size.

3.2 Array operations

Hash tables are implemented using arrays. The language, as it is, does not support arrays in any efficient way. Ideally, we would extend the language with array operations as primitive constructs. However, if we actually extend the language, we would really be defining a new language, which is a lot of work in Coq. Due to time constraints, we will not do this. Instead, we add the necessary operations and specifications as assumptions. We add three operations: `make_array`, `array_load`, and `array_store`. We also add a predicate `array`. The proposition `array a xs` states that the value `a` is an array containing the values given by the list `xs`. The assumed specification for the operations are shown in Figure 3.2. The expression `replicate n x` creates a list containing `n` copies of `x`, the expression `xs !! n` is syntax for looking up the `n`'th element in the list `xs`, while `<[ n := x]> xs` is syntax for replacing the `n`'th element in the list `xs` with `x`. Other than that, the specifications should be self-explanatory. In addition, we also assume that load and store operations are atomic. We need atomicity for the concurrent hash tables, where we will use the `array` predicate in an invariant (recall that invariants can only be opened for atomic expression).

We remind the reader, that these specifications are added as assumptions that cannot be proven for the language as it is. The assumptions should be seen as specifications that would hold if we had actually extended the language.

3.3 Sequential hash tables

The specification presented here is based on the specification for hash tables in CFML by Pottier [21]. It uses the same ideas behind the model and the function specifications

```
make_array_spec E n v:
  WP make_array (#n, v) @ E {{arr, array arr (replicate n v)}}

array_load_spec E arr xs v n :
    xs !! n = Some v ->
    {{{▷array arr xs}}}
      array_load (arr, #n) @ E
    {{{ RET v ; array arr xs}}}

array_store_spec E arr xs v n:
    n < length xs ->
    {{{▷array arr xs}}}
      array_store (arr, #n, v) @ E
    {{{ RET #() ; array arr (<[n := v]> xs)}}}

array_load_atomic arr i:
  atomic (array_load (arr, i))

array_store_atomic arr i v:
  atomic (array_store (arr, i, v))
```

Figure 3.2: The specifications for the array operations assumed to be available

and adapts the solution for Iris. The main differences between the two stems from the fact that CFML uses typed values, while the language used here is untyped.

The interface for the sequential hash table module is is defined as a record type `table`, whose definition is shown in Figure 3.3. It contains the six available operations as program values along with their specifications (the body of the specifications have been omitted here for brevity). It also contains an abstract type `table_state` which describes the internal state of the table and two predicates `table_in_state` and `is_cascade`. The former states that a value represents a table. The latter is used in the specification for cascades. Finally it contains two statements `table_in_state_wf` and `is_cascade_persistent`. The former guarantees that the model is always well-formed as discussed in §3.1. The latter states that the predicate `is_cascade` is persistent.

3.3.1 Implementation and table predicate

A hash table is represented as a triple of references `(la, ls, lc)`, where `la` points to an array of buckets, and `ls` and `lc` points to integers containing the number of stored elements and the length of the array respectively. Each bucket is a linked list of key-value pairs, i.e., a bucket is either `NONE` or `SOME (k, x, b)` for a key `k`, a value `x`, and a bucket `b`.

We first present the definition of the `table_in_state` predicate, shown in Figure 3.4. The proposition `table_in_state m data t` states that the program value `t` is a table representing the dictionary modeled by `m` and that `data` is a representation of the current state of the table. The `data` state argument allows specification to state whether an operation changes the state of the table. Specifically, if an operation has a specification with the proposition `table_in_state m data t` as both the precondition and the postcondition, then that operation does not change the state of the table (or at least it restores the table to the state it was in at the beginning of the operation). On the other hand, if the operation may change the state of the table, the specification will state `table_in_state m data' t` in the postcondition for some new state `data'`.

The `data` argument represents the data stored in the internal array of the hash table. It has type `list bucket_data`, which instantiates the `table_state` field of the interface. The type `bucket_data` is defined as `list (val * val)`, i.e., it represents a bucket as a Coq list of key-value pairs (note that we store the concrete value used as key rather than the corresponding `key`). Let us take a look at the definition of `table_in_state` in Figure 3.4. Overall, there are two parts to the definition. The first part consists entirely of pure Coq level statements about the model `m` and the content of the array `data`. The last part states that the value `t` has the correct form, i.e., it is a triple of locations and the locations point to an array containing `bucket <$> data` (`bucket` is a function mapping a `bucket_data` object to the program value representation of that bucket and `<$>` is syntax for the monadic `fmap` operator), the population count of the dictionary, and the length of the array respectively. Going back to the first part, we see that it consists of five assertions. It should be obvious what `length data > 0` means. The other four are summarized here:

```
Structure table Σ key hash map
                `{FinMap key map, heapG Σ,
                  !Hashable Σ key hash}
               : Type :=
  { table_create : val ;
    table_insert : val ;
    table_remove : val ;
    table_lookup : val ;
    table_fold : val ;
    table_cascade : val ;

    table_state : Type ;

    table_in_state : map (list val) -> table_state -> val -> iProp Σ ;
    is_cascade : map (list val) -> val -> list (val * val) ->
                 table_state -> val -> iProp Σ ;

    table_in_state_wf m state t :
      table_in_state m state t → ⌜table_wf m⌝ ;
    is_cascade_persistent m f seq state t :
      PersistentP (is_cascade m f seq state t) ;

    table_create_spec n :  (* ... *) ;
    table_insert_spec t k x m state k' : (* ... *) ;
    table_remove_spec m state t k k' : (* ... *) ;
    table_lookup_spec m state t k k' : (* ... *) ;
    table_fold_spec m state I f t a : (* ... *) ;
    is_cascade_spec m f seq state t : (* ... *) ;
    table_cascade_spec m state t : (* ... *) ;
  }.
```

Figure 3.3: The `table` type defines the interface of the hash table module

```
Definition table_in_state m data t : iProp Σ :=
  ⌜length data > 0⌝ *
  ⌜table_wf m⌝ *
  ⌜content m data⌝ *
  ⌜no_garbage data⌝ *
  ⌜have_keys data⌝ *
  ∃ lArr lSize lCap arr,
    ⌜t = (#lArr, #lSize, #lCap)⌝ *
    array arr (bucket <$> data) *
    lArr ↦ arr *
    lSize ↦ #(population m) *
    lCap ↦ #(length data).
```

Figure 3.4: Definition of the table invariant

- The proposition `table_wf m` states that `m` is a well-formed model as explained in §3.1.
- The proposition `content m data` states that `data` contains the right values in the right buckets as required by `m`.
- The proposition `no_garbage data` states that no keys are stored in the wrong buckets.
- The proposition `have_keys data` states that `data` contains only valid keys.

Figure 3.5 shows the specifications for the four basic operations. The specifications should mostly be self-explanatory, though there are a few things worth noting. First, the argument to the `create` function is a natural number indicating the initial number of buckets. Second, the `insert` function moves the content into a new array of double size if the population exceeds twice the length of the current array. Resizing is an optimization, so this is not visible in the specification, Specifically, it is hidden within the new state `data'` introduced in the postcondition.

3.3.2 Specification of fold

The function `fold` iterates over all key-value pairs in a table applying a function provided by the user to each entry using an accumulator to pass on a value between entries. Specifically, it takes three arguments: A table `t`, an initial value `a` for the accumulator, and a function `f`, which, when applied to a key-value pair and an accumulator value, returns an updated value for the accumulator. The specification for `fold` is shown in Figure 3.6. The specification requires a user-provided predicate `I`, which is used as loop invariant. Specifically, if, at any point during iteration after visiting all entries in the sequence `seq`, the accumulator has value `a`, then the proposition `I seq a` holds. The

```
Lemma create_spec n :
  n > 0 ->
  WP create #n {{t, ∃ data, table_in_state ∅ data t}}.

Lemma insert_spec t k x m data k' :
  as_key k = Some k' ->
  {{{table_in_state m data t}}}
    insert t k x
  {{{data', RET #(); table_in_state (insert_val m k' x) data' t}}}.

Lemma lookup_spec m data t k k' :
  as_key k = Some k' ->
  {{{table_in_state m data t}}}
    lookup t k
  {{{ RET match m !! k' with
          | Some (v :: _) => SOMEV v
          | _ => NONEV end ;
      table_in_state m data t}}}.

Lemma remove_spec m data t k k' :
  as_key k = Some k' ->
  {{{table_in_state m data t}}}
    remove t k
  {{{ data', RET match m !! k' with
                 | Some (v :: _) => SOMEV v
                 | _ => NONEV end ;
      table_in_state (remove_val m k') data' t}}}.
```

Figure 3.5: Specifications for the basic operations on sequential hash tables

```
Lemma fold_spec m data I f t a :
  {{{(∀ k x seq (a' : val),
           {{⌜permitted m (seq ++ [(k,x)])⌝ *I seq a'}}
             f k x a'
           {{v, I (seq ++ [(k,x)]) v }}) *
      table_in_state m data t * I [] a}}}
    fold f t a
  {{{v seq, RET v; ⌜complete m seq⌝ *
                   table_in_state m data t *
                   I seq v}}}.
```

Figure 3.6: Specification for the `fold` function

client must prove that the invariant holds initially, i.e., `I [] a` holds for the initial accumulator `a`, and that applying `f` always updates the invariant. This requirement is expressed by the nested triple in the precondition. The fact that this specification is defined in terms of another specification makes this a higher-order specification. Note that the expected specification for `f` does not include `table_in_state` as a precondition. Thus, `f` will not have any access to the table. An alternative specification could list `table_in_state m data t` as both a precondition and a postcondition. This would give `f` read-only access to the table – or at least it would require that `f` leaves the table in the same state as it was given.

The possible sequences `seq` that `fold` may produce are described by the predicates `permitted` and `complete`. The proposition `permitted m seq`, states that `seq` is a sequence of elements that could be produced from the model `m`. In other words `seq` could be obtained by removing entries of `m` one by one and putting them into a list. A sequence permitted by the `permitted` predicate is not necessarily complete, i.e., there may be elements left in the model after removing the elements in the sequence. The user provided specification of the function `f` may assume that the sequence produced so far, including the entry currently being processed, is permitted. The proposition `complete m seq` states that, in addition to being a permitted sequence, the sequence `seq` is complete, i.e., it contains all the entries of `m`. The specification of `fold` states in the postcondition that the invariant `I seq v`, where `v` is the return value, holds for a complete sequence `seq`.

The order in which the elements are visited by `fold` is mostly unspecified. However, for any key mapped to more than one value in the model, the most recently inserted values associated with that key are visited first. In other words, for each key `k`, the internal order of the elements in `m !! k` is preserved in all permitted sequences. Other than this restriction, no guarantees are made about the order of the entries of permitted sequences.

The `fold` function is implemented using two nested loops. The outer loop iterates over the array of buckets, and for each bucket it executes the inner loop, which visits

each entry in the bucket. When proving the specification of the function, it must be shown that after each iteration of the inner or outer loop, the user provided invariant `I seq a` holds and that the sequence `seq` is permitted. Both `permitted` and `complete` are defined in terms of the `removal` predicate. The proposition `removal m seq m'` states that `m'` can be obtained from `m` by removing the elements in `seq` one at a time using `remove_val`. The proposition `permitted m seq` is defined as `exists m', removal m seq m'`, while `complete m seq` is defined as `removal m seq` $\emptyset$. Thus, whenever an entry is visited and added to the sequence, we need to come up with a model that can be obtained by removing said sequence from the original model. We do this by maintaining a simulated table. This table is initially identical to the actual table, and whenever an entry is visited, we remove that entry from the simulated table. When the function terminates, the simulated table will be empty. This is done by maintaining the invariant shown in Figure 3.7. This pure predicate holds at any point during `fold` in addition to the `table_in_state` and user defined `I` invariants. It takes quite a lot of parameters. The first `m` and `data` are the model and representation of the bucket array of the actual table. These are the only parameters to remain constant between iterations. The `m'` and `data'` are the model and representation of the simulated table. The `i` parameter is the index for the outer loop and therefor also the index for the bucket currently being visited. During the i'th iteration the state of the j'th bucket of the simulated table is as follows for each index j:

- If $j < i$, then the j'th bucket has been fully processed and is therefor empty.
- If $j = i$, then the j'th bucket is currently being processed. The sequence `bPref` contains the already visited entries from this bucket. In the simulated table this bucket contains the remaining entries.
- If $j > i$, then the j'th bucket has not been visited and its content is identical to that of j'th bucket in the actual table.

The sequence `seq` contains the entries from previously visited buckets (i. e. buckets with indices lower than i). The sequence `bPref` contains the entries from the current bucket, that have been visited. Thus `seq ++ bPref` is the sequence of entries visited so far. The body of the definition of the invariant is quite large, but it really boils down to stating that both the actual and simulated tables are in a valid state and that the content of the simulated table is as stated above. The most important part of the definition is the statement `removal m (seq ++ bPref) m'`, which allows us to show that the sequence produced so far is indeed permitted.

With this invariant at hand, the proof of the specification boils down to showing that the invariant is maintained after each iteration of the loops, which is fairly straight forward.

3.3.3 Specification of cascade

The function `cascade` returns a cascade, which can be thought of as a stateless functional variant of an iterator [10]. It is a function which, when applied to the unit value, either

```
Definition fold_loop_inv data m seq bPref data' m' i :=
  table_wf m /\
  content m data /\
  no_garbage data /\
  have_keys data /\
  removal m (seq ++ bPref) m' /\
  table_wf m' /\
  content m' data' /\
  no_garbage data' /\
  have_keys data' /\
  length data = length data' /\
  (forall j, j < i -> data' !! j = Some []) /\
  (forall b, data' !! i = Some b -> data !! i = Some (bPref ++ b)) /\
  (forall j, i < j -> data' !! j = data !! j).
```

Figure 3.7: The loop invariant maintained during `fold`

returns a key-value pair and a new cascade, or returns `NONE` if there are no entries left. The specification for `cascade` is very straightforward: If `t` is a table, then `cascade t` maintains the table invariant and returns a cascade as stated by the `is_cascade` predicate. The proposition `is_cascade m f seq data t` states that `f` is a cascade for the table `t` in state `data` representing dictionary `m` and that the sequence of entries `seq` has already been produced. The user does not need to know how `is_cascade` is defined. They only need to know the `is_cascade_spec` specification, which is shown in Figure 3.8 along with the specification for `cascade`. This specification assumes in the precondition that `f` is a cascade and that the associated table is in the same state as when `cascade` was called. The postcondition states that the table remains in that state and that one of the following two cases holds:

- The returned value is `NONE`, in which case the sequence `seq` already produced is complete.
- The returned value is `SOME (k, x, f')`, where `f'` is a new cascade for which the key-value pair `(k, x)` is added to the sequence of produced entries.

The definition uses the predicates `permitted` and `complete`, which are presented in §3.3.2.

The `is_cascade` predicate is parameterized with the state of the table `data` and the specification for the cascade requires that the table is in that same state when the cascade is used. This means that cascades are invalidated when the state of the table is changed as the specification no longer applies.

It is also worth noting that `is_cascade` is persistent, meaning it can be duplicated freely. This stems from the fact that cascades are stateless. Because of this, cascades can be shared and reused, though access to the table is needed whenever they are used.

```
Lemma cascade_spec m data t:
  {{{table_in_state m data t}}}
    cascade t
  {{{f, RET f; is_cascade m f [] data t *
               table_in_state m data t }}}.

Lemma is_cascade_spec m f seq data t:
  {{{ is_cascade m f seq data t * table_in_state m data t }}}
    f #()
    {{{v k x f' , RET v;
       table_in_state m data t *
       ((⌜v = NONEV⌝ * ⌜complete m seq⌝) ∨
        (⌜v = SOMEV ((k, x), f')⌝ *
        ⌜permitted m (seq ++ [(k, x)])⌝ *
        is_cascade m f' (seq ++ [(k, x)]) data t)) }}}.
```

Figure 3.8: Specification for cascades and the `cascade` function

Like `fold`, the cascades are implemented using two nested loops. The outer loop iterates over the array of buckets and for each bucket it executes the inner loop, which visits each entry in the bucket. However, unlike in `fold`, the loop is evaluated in a lazy fashion. Whenever the loop visits an entry, it suspends and returns, along with the entry, the rest of the loop as a continuation – a lambda which, when applied, will resume the loop where it stopped. The returned continuation acts as the next cascade. The loop visits the entries in the same order as `fold` does. Because of this, the same invariant `fold_loop_inv`(see Figure 3.7) is used to prove the `permitted` and `complete` propositions in the specification for cascades. This is done by embedding the invariant into the `is_cascade` predicate. The `is_cascade` predicate (definition shown in Figure 3.9) uses existential quantification to define the current state of the loop, i.e., the index of the current bucket `i` and the remaining elements of the current bucket `b`, that have yet to be visited. This is also where it becomes necessary to assume that the state of the table has not changed. The `fold_loop_inv` predicate is parameterized with the state `data` and in order to use the invariant to argue that the sequence produced is permitted, the table needs to be in state `data` as well. The `is_cascade` predicate makes two claims: The first claim is that the `cascade_inv` predicate holds, which is essentially just a wrapper for the `fold_loop_inv` invariant. The second claim is that when applied, the cascade has the same behavior as `cascade_next` applied to the table and the current state of the loop. The function `cascade_next` is used internally in the implementation of `cascade`. It can be thought of as a lambda lifted version of the cascades. In fact the cascades are simply implemented as lambda closures, whose bodies apply `cascade_next`. The specification for `cascade_next` is shown in Figure 3.9. The postcondition for this specification is similar to the one from `is_cascade_spec`, while the precondition uses

```
Definition cascade_inv seq m data b i :=
  exists seq' bPref data' m',
    data' !! i = Some b /\
    seq = seq' ++ bPref /\
    fold_loop_inv data m seq' bPref data' m' i.

Definition is_cascade m f seq data t : iProp Σ :=
  ∃ b i,
    ⌜cascade_inv seq m data b i⌝ *
    ∀ P Q,
      □({{P}} cascade_next (bucket b) t #i {{Q}} →
        {{P}} f #() {{Q}}).

Lemma cascade_next_spec seq m data b i t:
  cascade_inv seq m data b i ->
  {{{ table_in_state m data t }}}
    cascade_next (bucket b) t #i
    {{{v k x f' , RET v;
       table_in_state m data t *
       ((⌜v = NONEV⌝ * ⌜complete m seq⌝) ∨
        (⌜v = SOMEV ((k, x), f')⌝ *
         is_cascade m f' (seq ++ [(k, x)]) data t)) }}}.
```

Figure 3.9: Definition of `is_cascade` and specification for the internal function `cascade_next`

the weaker `cascade_inv` predicate over `is_cascade`. Thus, using this specification, it is quite easy to show the specification for cascades given that cascades have the same behavior as `cascade_next`.

3.3.4 Entry invariants

The model and the specifications for the hash tables makes no assumptions about the values stored in the table. Since the values are untyped in the language, it is possible to store any value in the table. This makes the tables very flexible, however, it also makes it difficult to say anything about the values stored in the table. Users may not always need this much flexibility, but may instead want to restrict the kind of values that a table is allowed to store. One way of doing this is to introduce a predicate that must be satisfied by each entry in the table. We here present a simple set of tools to support this. These tools only refer to the model and do not depend on the implementation or even the specifications of the hash table. Figure 3.10 shows the definition of the `table_inv` predicate. The proposition `table_inv P m` states that the predicate `P` is satisfied by each key-value pair in the table. The definition makes use of the `[* list]` syntax for

```
Definition table_inv (P : key -> val -> iProp Σ) m : iProp Σ :=
  [* list] kv ∈ all_elements m, P (kv.1) (kv.2).

Lemma table_inv_empty P : table_inv P ∅.

Lemma table_inv_insert P m k x:
  table_inv P m * P k x ⊣⊢
  table_inv P (insert_val m k x).

Lemma table_inv_remove P m k x xs:
  m !! k = Some (x :: xs) ->
  table_inv P m ⊣⊢
  P k x * table_inv P (remove_val m k).

Lemma table_inv_removal P m seq m':
  removal m seq m' ->
  table_inv P m ⊣⊢
  ([* list] kv ∈ seq, (∃ k, ⌜as_key (kv.1) = Some k⌝ * P k (kv.2))) *
  table_inv P m'.

Lemma table_inv_complete P m seq:
  complete m seq ->
  table_inv P m ⊣⊢
  [* list] kv ∈ seq, (∃ k, ⌜as_key (kv.1) = Some k⌝ * P k (kv.2)).
```

Figure 3.10: Definition of `table_inv` and related lemmas

separating conjunction of a predicate applied to each element of a list. The predicate `P` will usually at least require the values to have a certain syntactical form, which would correspond to them having a certain type in a typed language, though it is in no way limited to that. Since the predicate takes both a key and a value, it can express a relation between a value and the key under which it is stored. In addition, because `P` is an Iris predicate and `table_inv` is defined using separating conjunction, it is possible for `P` to assert ownership of some resources, in which case that same ownership will be asserted by `table_inv`. In other words, it allows the table to own resources in addition to those required by the data structure itself. For instance, `P k v` could state that `v` is a resource handle while also asserting ownership of the resource associated with `v`. In that case `table_inv P m` would state that `m` is a dictionary storing resource handles while also asserting ownership of all the resources associated with the stored handles.

In addition to defining the `table_inv` predicate, we also provide a few related lemmas, which are listed in Figure 3.10. The `table_inv_empty`, `table_inv_insert`, and `table_inv_remove` lemmas should be self explanatory. The `table_inv_removal` lemma

```
Definition test_1 : expr :=
  let: "t" := table_create tab #10 in
  table_insert tab "t" #1 #1 ;;
  table_insert tab "t" #2 #2 ;;
  let: "a" := ref #0 in
  table_fold tab (λ: "k" "x" <>, "a" <- !"a" + "x") "t" #() ;;
  !"a".
```

Figure 3.11: Sample program. Here `tab` is an object of type `table` shown in Figure 3.3.

states that `P k v` holds for each pair `(k, v)` in a possibly incomplete sequence of entries produced by one of the iteration operations. The `table_inv_complete` lemma is a corollary to `table_inv_removal` stating that the predicate holds for all entries in a complete sequence.

3.3.5 Sample client

To illustrate how the specifications for the hash table module can be used, we here present a simple client program that uses the module. The program creates a table, inserts a few numbers into the table, and finally sums up the numbers in the table using `fold`. Before we can write the program, however, we need a key type and a hash function to instantiate the table module. To keep things as simple as possible, we choose `nat`, Coqs type of natural numbers, as the key type. As hash function, we choose the identity function. Thus, the set of legal keys becomes the non-negative integer literals.

The program we will verify is shown in Figure 3.11. The specification we wish to prove is quite simple: The program should return the value 3. In the proof of this specification, we apply the specifications for `table_create`, `table_insert`, and `table_fold`. The first two are straightforward to use. For the `table_fold` operation, however, we need to come up with a loop invariant. We choose the invariant `test_1_inv m l` defined in Figure 3.12. The invariant first states that all the values in the table are integers. This is expressed using the `table_inv` predicate, presented in §3.3.4.[1] In addition, the invariant states that the location `l`, which will be instantiated by the location returned by the `ref #0` expression in the program, points to an integer that is the result of summing up all the values visited so far.

With the invariant picked, the specification of `table_fold` requires that we show that the loop invariant is updated in each iteration. In other words, we have to prove the lemma shown in Figure 3.13. Using the `table_inv_removal` lemma from Figure 3.10 to assert that the value of the currently visited entry is an integer, the proof becomes quite straight forward.

[1] Strictly speaking, it is not necessary to state this as part of the invariant as we are dealing with a persistent predicate. However, in other applications, where non-persistent instances of `table_inv` are used, it is necessary to include this in the loop invariant if the proof depends on it.

```
Definition int_table : gmap nat _ -> iProp Σ :=
  table_inv (fun _ v => ∃ (j : Z), ⌜v = #j⌝).

Fixpoint int_val_sum (seq : list val) :=
  match seq with
  | [] => Some 0
  |  #(LitInt x) :: seq => z ← int_val_sum seq ; Some (z + x)
  | _ => None
  end.

Definition test_1_inv m l seq _ : iProp Σ :=
  int_table m *
    ∃ i, ⌜int_val_sum (seq.*2) = Some i⌝
         * l ↦ #i.
```

Figure 3.12: Invariant used for the application of `table_fold`

```
Lemma test_1_inner m l k x seq a:
  permitted m (seq ++ [(k,x)]) ->
  {{test_1_inv m l seq a}}
    (λ: "k" "x" <>, #l <- !#l + "x") k x a
  {{v, test_1_inv m l (seq ++ [(k,x)]) v }}.
```

Figure 3.13: Specification of the lambda given to `table_fold` as required by `table_fold_spec`

3.4 Concurrent hash tables

The hash table module presented in §3.3 is intended to be used in sequential programs only. If a single table is used concurrently by multiple threads, race conditions may occur. If, for example, two insertions are performed concurrently on the same bucket, the updated bucket resulting from the first insertion may be overridden by that of the second insertion thus causing the first insertion to be undone. The lack of concurrent support is reflected in the specifications by the fact that all the operations require the `table_in_state m state t` proposition to hold, a proposition which in general cannot be duplicated. Thus, it is impossible for anyone except the sole owner of the proposition to perform any operations on the table.

We here present an implementation and a specification of concurrent hash tables. The implementation supports a set of basic operations: `create`, `insert`, `lookup`, and `remove`. The implementation is inspired by the `ConcurrentHashMap` class from the Java standard library [13].

Before going into detail, we have to consider what kind of specification we can hope to prove for shared tables. In the sequential case, we modeled each table as a dictionary allowing us to describe the content of the table precisely at all times. This was possible because no one except the owner had access to the table and thus the owner would know that the content of the table would not change between uses. In the case where a table is shared, this is not the case. Here, other users of the table may change the content at any time, which means that each individual user cannot in general know the content of the table when they use it. The question then is: What do we know about the content of the table from the perspective of a single thread? In general, the answer is basically nothing. If other threads can change the table arbitrarily then we never know what the table contains. If we want to know anything, then we need to restrict what other threads are allowed to do. It may, however, be impossible to choose a set of restrictions that are suitable for all applications. Instead, we let the client choose. This leads us to the following approach: The table will be described by a predicate parameterized with another predicate chosen by the client. This predicate will act as an invariant for the table in the sense that the dictionary represented by the table will always satisfy it.

3.4.1 Implementation

Like for the sequential version, it is assumed that array operations are available. Additionally, the implementation also relies on a module providing support for locks. Such modules already exist and are included with the Coq formalization of Iris. The interface and specifications for the lock modules are shown in Figure 3.14. The main idea behind the specification of the lock module is that each lock is parameterized with a proposition `R`, which must hold whenever the lock is available. Acquiring the lock will make the proposition available to the client. When releaseing the lock, the proposition must be reestablished.

The implementation does not use a single lock to protect an entire table. Rather, each bucket in a table is protected by its own lock. This ensures that different threads

```
Structure lock Σ `{!heapG Σ} := Lock {
  (* -- operations -- *)
  newlock : val;
  acquire : val;
  release : val;
  (* -- predicates -- *)
  (* name is used to associate locked with is_lock *)
  name : Type;
  is_lock (N: namespace) (γ: name) (lock: val) (R: iProp Σ) : iProp Σ;
  locked (γ: name) : iProp Σ;
  (* -- general properties -- *)
  is_lock_persistent N γ lk R : PersistentP (is_lock N γ lk R);
  locked_exclusive γ : locked γ -* locked γ -* False;
  (* -- operation specs -- *)
  newlock_spec N (R : iProp Σ) :
    {{{ R }}} newlock #() {{{ lk γ, RET lk; is_lock N γ lk R }}};
  acquire_spec N γ lk R :
    {{{ is_lock N γ lk R }}}
      acquire lk
    {{{ RET #(); locked γ * R }}};
  release_spec N γ lk R :
    {{{ is_lock N γ lk R * locked γ * R }}}
      release lk
    {{{ RET #(); True }}}
}.
```

Figure 3.14: Interface for the lock modules already included in Iris

can access different buckets concurrently without blocking. This approach is inspired by the implementation of the `ConcurrentHashMap` class from the Java standard library.

A table is represented as a pair `(a,n)`, where `a` is an array and `n` is the length of `a` represented as an integer. Each entry in the array is a pair `(lk, l)` of a lock `lk` and a reference `l` to a bucket. A bucket is a linked list of key-value pairs, just like in the non-concurrent version. For the sake of simplicity, this implementation is very minimal and does not support certain advanced features that one might expect from hash tables used in practice. In particular, the tables do not support resizing of the arrays. The arrays remain unchanged after initialization.

The implementation of the operations are mostly as one would expect. One thing worth noting is how the locks are used. In `insert` and `remove`, the lock for the appropriate bucket is acquired before the bucket is read and released after the updated bucket is written. In `lookup`, no locks are acquired. Thus, the locks are only needed to obtain write access. Simply reading is always allowed.

```
Definition is_table N P t :=
  ∃ arr refs locks,
    ⌜t = (arr, #(length refs))⌝ *
    ⌜length refs > 0⌝ *
    ⌜length refs = length locks⌝ *
    ([* list] i ↦ lr ∈ zip locks refs,
      let '(l, r) := lr in
      ∃ lname, is_lock lck (N.@(S i)) lname l (r ↦{1/2} -)) *
    inv (N.@0)
     (array arr (zip_with PairV (LitV ∘ LitLoc <$> refs) locks) *
      ∃ m data,
        ⌜table_wf m⌝ *
        ⌜content m data⌝ *
        ⌜no_garbage data⌝ *
        ⌜have_keys data⌝ *
        ⌜length data = length refs⌝ *
        P m *
        [* list] rb ∈ zip refs data,
         let '(r, b) := rb in
         r ↦{1/2} bucket b).
```

Figure 3.15: Definition of the `is_table` predicate for concurrent hash tables.

3.4.2 Table invariants and predicates

Just like the non-concurrent version, both the implementation and specification of the concurrent hash table are parameterized with a type `key` and a function `hash` implementing the class `Hashable` shown in Figure 3.1.

The proposition `is_table N P t` states that the value `t` is a table representing a dictionary, which satisfies the predicate `P` and uses the namespace `N` for invariants. The predicate `P` must have type `map (list (val * val)) -> iProp Σ`, where for `map` there exists an instance of the type class `FinMap key map`. Thus, `P` is a predicate on Coq level dictionaries. It is important to note that `is_table N P t` is persistent, which means that it can be duplicated freely.

The definition of `is_table` is shown in Figure 3.15. The predicate existentially quantifies over the array (`arr`) and the references (`refs`) and locks (`locks`) stored in the array. The predicate states that the table `t` has the correct format, that all the lists used have the same (non-zero) length, and that the locks are indeed all locks. In addition it defines an Iris invariant. The body of the invariant contains an assertion that the array contains what it is supposed to contain. In addition, it defines two variables `m` and `data` through existential quantification. At any given time the variable `m` contains the model dictionary represented by the table. The `data` variable has type `list bucket_data`, where `bucket_data` is defined as `list (val * val)`. It is thus a Coq representation of

the contents of the buckets. The existential quantifiers for these are contained within the body of the invariant to allow the table to change. If the variables were declared outside the invariant, the table would be locked into single state that could not be changed. The body of the invariants contains assertions defined using the predicates `table_wf`, `content`, `no_garbage`, and `have_keys`. These are described in §3.1 and §3.3.1. Together they state that `data` represents `m`. The invariant also contains the assertion `P m` stating that the table satisfies the user provided predicate.

Finally, the invariant contains a number of assertions of the form `r ↦{1/2} bucket b`. This syntax indicates fractional ownership [4, 3] of a heap location. Fractional ownership means that the full ownership has been split into multiple parts, each annotated with a rational number $q \in (0; 1]$, which can be recombined into the full ownership according to the following rule (expressed using mathematical notation rather than Coq notation):

$$l \overset{q_1+q_2}{\mapsto} v \dashv\vdash l \overset{q_1}{\mapsto} v * l \overset{q_2}{\mapsto} v$$

The assertion $l \overset{1}{\mapsto} v$ (i.e., when $q = 1$) represents full ownership (in which case we usually omit the annotation). Full ownership is needed to modify the value, while reading the value can be done with any fraction. For all fractions $q < 1$, the actual value does not matter. For instance, $l \overset{1/10}{\mapsto} v$ and $l \overset{9/10}{\mapsto} v$ both give exactly the same rights.

Thus, the invariant contains half ownership of each reference. The other halfs are stored behind the locks (the syntax `l ↦{q} -` asserts fractional ownership of a reference `l`, but with no knowledge of the stored value). The result is that the buckets can always be read by opening the invariant and obtaining half ownership. However, to modify a bucket, full ownership is needed, which means the lock must be acquired. In addition, acquiring the lock allows us to maintain half ownership for as long as we hold the lock. Thus, we are guaranteed that the bucket is not modified by someone else, while we hold the lock.

The predicate as described here makes use of the fact that the Iris implementation already provides fractional heap assertions. However, it could just as well not have been the case, i.e., it could have been the case that we only had exclusive ownership available. In this case, we could get the same result by using ghost state. We would need a RA supporting half ownership of buckets. One such RA could be the following:

$$\textsc{Auth}\left(\left(\textsc{Ex}\left(\text{list}\left(\text{val} \times \text{val}\right)\right)\right)^{?}\right)$$

This RA is built from several general constructions described in [18]. For a set S, $\textsc{Ex}\,(S)$ is the exclusive RA of S. The valid elements of $\textsc{Ex}\,(S)$ is the set S and the composition operator is defined such that $a \cdot b$ is never valid. For a RA M, $\textsc{Auth}(M)$ is the authoritative RA of M. The authoritative RA construction will be covered in §3.4.4. We can only construct an authoritative RA from a RA with a unit element, so we use the construction of optional RA's $M^?$ to extend the exclusive RA with a unit element. The result is a RA, where $\bullet\, l$ and $\circ\, l$ are valid elements for sequences of pairs of values l. For all l_1 and l_2, both $\bullet\, l_1 \cdot \bullet\, l_2$ and $\circ\, l_1 \cdot \circ\, l_2$ are invalid, while $\bullet\, l_1 \cdot \circ\, l_2$ is valid if and only

if $l_1 = l_2$. Both $\bullet\, l$ and $\circ\, l$ each represents half ownership and both parts are needed to update the list.

What we would do is allocate a ghost resource for each bucket and maintain the assertion $l \mapsto \text{bucket}(b) * \boxed{\circ\, b}^{\gamma}$ in the invariant for each bucket. Stored behind each lock, we would have the other half, i.e., each lock would store the proposition $\exists b. \boxed{\bullet\, b}^{\gamma}$. The result would be that we would be able to read from the heap location just by opening the invariant. However, when changing the stored value, $\boxed{\circ\, b}^{\gamma}$ will have to be updated, which can only be done if both halves are available. Therefore the lock must be acquired to allow writing.

3.4.3 Specifications

Unlike in the sequential case, we do not know exactly which dictionary the table currently represents. The table is only described using the client provided predicate `P`. Since this predicate is provided by the client, we need the client to show that the predicate is maintained as an invariant when the table is modified, and we need the client to tell what they want to know about any returned elements. The specifications are shown in Figure 3.16.

The `create` operation creates a new empty table with the number of buckets given as argument. The user thus needs to prove that the invariant holds for the empty dictionary.

The `insert` operation inserts a given element under a given key. Since it is never known exactly which dictionary the table represents, the client needs to show that the invariant is preserved by the insertion for any dictionary. For this, the precondition requires a proof of `∀ m, P m ={⊤\↑N}=* P (insert_val m k x) * Q`. The notation P `={`$\mathcal{E}$`}=*` Q means $P \mathrel{-\!\!*} \Rrightarrow_{\mathcal{E}} Q$. The update modality allows the client to perform ghost state updates as well as open invariants in the proof. We also allow the client to specify a proposition `Q` that they want returned in the postcondition. This could be any resources resulting from the proof, that should not be thrown away. It can also serve as a proof that the operation did indeed update the table. If the specification did not include this postcondition, then the specification would have an empty postcondition, which could be satisfied by letting the operation do nothing. With this postcondition, the operation needs to perform the insertion. The basic idea behind this is that when proving the specification, `Q` can only be obtained by using `∀ m, P m ={⊤\↑N}=* P (insert_val m k x) * Q`. Doing so, however, updates `P m` to `P (insert_val m k x)`, which means the table must be updated.

The `remove` operation removes and returns an element from the table if possible. For this operation, the client needs to provide two proofs: One for the case where there is something to remove, and one for the case where there isn't. Since only one of the two will be used, they are specified using regular non-separating conjunction. In the case, where something is removed, we cannot say exactly what will be returned. Instead, the user provides a predicate `Q` describing the returned element. In addition to describing the returned element, `Q` may also contain any resources leftover from the proof. In the

```
Lemma create_table_spec N P n :
  n > 0 -> {{{P ∅}}} create_table #n {{{t, RET t ; is_table N P t}}}.

Lemma table_insert_spec N P Q k k' x t:
  as_key k' = Some k ->
  {{{is_table N P t * ∀ m, P m ={⊤\↑N}=* P (insert_val m k x) * Q }}}
    table_insert t k' x
  {{{RET #(); Q}}}.

Lemma table_remove_spec N P Q Q' k k' t:
  as_key k' = Some k ->
  {{{is_table N P t *
     (∀ m,
        ⌜m !! k = None⌝ -* P m ={⊤\↑N}=* P m * Q') ∧
     ∀ m x xs,
        ⌜m !! k = Some (x :: xs)⌝ -* P m ={⊤\↑N}=*
        P (remove_val m k) * Q k x}}}
    table_remove t k'
  {{{v x, RET v; ⌜v = NONEV⌝ * Q' ∨ (⌜v = SOMEV x⌝ * Q k x)}}}.

Lemma table_lookup_spec N P Q Q' k k' t:
  as_key k' = Some k ->
  {{{is_table N P t *
     (∀ m,
        ⌜m !! k = None⌝ -* P m ={⊤\↑N}=* P m * Q') ∧
     ∀ m x xs,
       ⌜m !! k = Some (x :: xs)⌝ -* P m ={⊤\↑N}=* P m * Q k x}}}
    table_lookup t k'
  {{{v x, RET v; ⌜v = NONEV⌝ * Q' ∨ (⌜v = SOMEV x⌝ * Q k x)}}}.
```

Figure 3.16: Specifications for operations on concurrent hash tables.

other case, the client just needs to show that the invariant `P m` is maintained when `m` does not change. This is, of course, a very trivial thing to prove by itself. The purpose of requiring any proof in this case, is to allow the client to require a proof that there really was nothing to return, by letting them choose a postcondition `Q'`. Without this, the specification could be satisfied by an implementation that always returns `NONE`.

The `lookup` operation is similar to `remove`. The difference is that any found element is still kept in the table. Thus, the user must prove that `P` holds for the dictionary even after obtaining the assertion `Q` on the returned element. This usually means that `lookup` cannot be used to obtain any exclusive ownership of resources associated with elements in the table.

Entry invariants

In order to use a concurrent hash table, the client must choose a predicate describing the contents of the table. A possible choice for this predicate would be an instance of the `table_inv` predicate presented in §3.3.4. This allows the client to state something that holds for each entry in the table, such as the form of the stored values (i.e., what would be the type in a typed language). In addition, the table can take ownership of resources associated with the entries, which would make the resources shared and available to any user of the table. The `table_inv` predicate, however, says nothing about which entries it actually does contain. In particular, it never guarantees that the table contains anything at all. This cannot be avoided without adding additional constraints with regards to who can insert and remove elements, as there is always the option that anything inserted by one thread is quickly removed by another thread.

The specifications in Figure 3.16 are relatively simple to use when combined with the lemmas in Figure 3.10. The specification for `create` just requires that `table_inv P ∅` holds, which is always the case according to the lemma `table_inv_empty`. For `insert` operations, we do not need the client chosen postcondition `Q`, used by the specification, so it can just be instantiated with `True`. This means we just need to show the following:

```
∀m, table_inv P m ={⊤\↑N}=* table_inv P (insert_val m k x)
```

If the predicate `P k x` holds, then this follows from lemma `table_inv_insert`. For `remove` operation, we need to choose a predicate `Q` returned in the success case and a proposition `Q'` return in the failure case. We choose the per-entry predicate `P` for the success case, which means we have to prove

```
∀ m x xs, ⌜m !! k = Some (x :: xs)⌝ -*
          table_inv P m ={⊤\↑N}=* table_inv P (remove_val m k) * P k x
```

This follows directly from the lemma `table_inv_remove`. This also means the proposition `P k x` is given in the postcondition, if something is removed. We don't need anything in the postcondition in the failure case, so we just pick `True` as the returned proposition. This makes the proof for this case trivial. The `lookup` operation is similar to `remove`, however, since we are not removing the found element from the table, we

```
Definition test_1 : expr :=
  let: "t" := create_table #10 in
  let: "x" := ref #() in
  table_insert "t" #1 #1 ||| ("x" <- table_lookup "t" #1) ;;
  !"x".
```

Figure 3.17: Very simple program that uses the concurrent table

cannot take over any ownership claimed by the table. In other words, we need `P k x` to be duplicable.

Note that the specifications in Figure 3.16 are more complicated than what is needed in this case. In particular, we did not need the client specified postconditions for the cases where nothing was returned. In §3.4.4, we will see a more complex instantiation of the table predicate, which makes full use of all the generality offered by the specifications.

As a proof that the specifications can indeed by used, we present a very simple client program that uses a concurrent table. The program is shown in Figure 3.17. It creates a table, and then inserts something into the table while in parallel performs a lookup. The specification for this program is `WP test_1 {{v, ⌜v = SOMEV #1 \/ v = NONEV⌝}}`, and it is proven by instantiating the specifications with the predicate `table_inv (fun _ v => ⌜v = #1⌝)`, i.e., all entries in the table is the number one. While admittedly this predicate is too simple for any practical use, it shows that it is indeed possible to use the specifications in conjunction with `table_inv`. There is nothing preventing `table_inv` from being instantiated with more complex predicates in more complex programs.

3.4.4 Partial table ownership

The way the concurrent hash table module is specified, the entire table can be shared freely among any number of threads. As a consequence, none of the threads have much control over any part of the table. It may be useful for a single thread to have full control over some of the entries in the table. It is, in fact, possible to achieve this using the specifications presented above. The solution uses ghost state to describe and represent ownership of part of the table.

As explained in §2.3, ghost state is defined from a Resource Algebra (RA). We will use a construction called an *authoritative* RA, which comes bundled with Iris [18]. For a RA M with a unit element, the authoritative RA $\textsc{Auth}(M)$ allows us to express that someone has full ownership over an object represented by an element of M while others may have ownership of fragments of that object. Full ownership is expressed as $\bullet a$ and it holds that $\bullet a \cdot \bullet b$ is never valid, i.e., full ownership is exclusive. It is usually held by a single authoritative entity responsible for administrating all access to the represented object. We will therefore often call $\bullet a$ an authoritative element. Partial ownership is expressed as $\circ a$ and $\circ a \cdot \circ b = \circ (a \cdot b)$. It also holds that $\bullet a \cdot \circ b$ is valid only if a and b are valid and $b \preccurlyeq a$. In other words a partial element describes part of the object described by the single authoritative element. We also have frame preserving updates

described by the following rule:

$$\text{AUTH-UPDATE}\quad \frac{(a_1, b_1) \overset{\text{l}}{\rightsquigarrow} (a_2, b_2)}{\bullet\, a_1 \cdot \circ\, b_1 \rightsquigarrow \bullet\, a_2 \cdot \circ\, b_2}$$

Here $(a_1, b_1) \overset{\text{l}}{\rightsquigarrow} (a_2, b_2)$ denotes a local update, which is defined as follows:

$$(a_1, b_1) \overset{\text{l}}{\rightsquigarrow} (a_2, b_2) \triangleq \forall n, a_{\text{f}}^?. b_1 \in \mathcal{V} \land a_1 = b_1 \cdot a_{\text{f}}^? \Rightarrow a_2 \in \mathcal{V} \land b_2 = b_2 \cdot a_{\text{f}}^?$$

In other words, we can perform an update to both the authoritative element and a fragment, if the update does not change the represented object in any way that can be observed in any other fragments.

What we will do is choose an appropriate RA M for representing fragments of a dictionary and allocate a ghost resource represented by the RA $\textsc{Auth}(M)$. As invariant for `is_table`, we choose the authoritative element. Since the fragments are always consistent with the authoritative element, each fragment will describe part of the table and assert ownership accordingly.

What we need now is a RA that describes parts of a dictionary. We want each element to describe a set of keys and the partial table obtained by picking the entries from the whole table, where the key is contained in the set of keys. The element should provide exclusive permission to perform any action on any key contained within the set, and no access to any other keys. The set of valid elements should thus be:

$$\left\{(m, S) \in (K \overset{\text{fin}}{\rightharpoonup} \text{list}(\text{Val})) \times \wp(K) \;\middle|\; \text{dom}(m) \subseteq S\right\},$$

where K is the set of keys available. Composition should be $(m_1, S_1) \cdot (m_2, S_2) = (m_1 \cup m_2, S_1 \cup S_2)$ if S_1 and S_2 are disjoint. If they are not disjoint, the composition should be invalid. In other words, each key can only be contained by the set of one element, which means that each element provides exclusive access to its set of keys.

While the RA, as described above, is fairly simple conceptually, there are some issues that must be resolved when we define it formally. The first issue is that the composition operator must map elements with intersecting sets to something invalid. The Coq implementation of Iris requires that the composition operator is implemented as a regular Coq function, which must be computable. Since disjointness of sets is not decidable in general, we cannot define an operator, that directly decides whether the result should be valid or not. We do, however, have another option. Validity of elements does not have to be decidable in general. Thus, we do not actually need to decide whether the sets intersect, when computing the composition. We just need to make sure that intersecting sets do indeed map to something invalid, even if we cannot in general decide whether the elements are invalid. Our solution is to not just use sets, but rather multisets. We define a type of multisets using functions from the type of elements to natural numbers:

```
Record multiset (A : Type) : Type := {multiset_car : A -> nat}.
```

Compared to using general sets (defined using predicates), the only limitation to the sets that can be expressed using `multiset` is that membership of the set must be decidable. Union on `multiset`s is defined using addition in the obvious way. We also introduce the predicate `multiset_no_dup` on the `multiset` type, which state that a `multiset` contains no duplicates, i.e., the function `multiset_car` never returns a number greater than one. It is easy to see that `multiset_no_dup (X U Y)` holds if and only if `X` and `Y` are disjoint and `multiset_no_dup` holds for both `X` and `Y`. Thus, we can get what we want by requiring that valid elements satisfy `multiset_no_dup`.

Next, we need to make sure that the RA satisfies all the requirements listed in Figure 2.3. One property in particular turns out to be a little tricky:

$$\forall a, b.\, (a \cdot b) \in \mathcal{V} \Rightarrow a \in \mathcal{V}$$

In other words, if the result of a composition is valid, then both components must valid as well. With our current composition, we have the following:

$$(m, S) = (m, \emptyset) \cdot (\emptyset, S)$$

If (m, S) is valid, and $m \neq \emptyset$, then this example violates the above rule, as $(m, \emptyset)$ will not be valid (the domain of m cannot be contained within the empty set). Our solution is to specifically check whether this is a problem in the composition operator. In other words, we modify the composition operator to specifically check whether the domain of each map is contained within the corresponding set. If one of the maps do not pass the test, the composition will return a special element $\bot$, which is never valid. The reason we are able to perform this check is that the domains are finite and membership of `multiset`s is decidable.

Finally, the construction for an authoritative RA requires that our RA has a unit element. The obvious candidate would be $(\emptyset, \emptyset)$, as $\emptyset$ is generally a unit for union. However, with the change above, it turns out that this will not work. In particular $(m, S) \cdot (\emptyset, \emptyset) = \bot$ if $\mathrm{dom}(m) \notin S$. Instead, we have add a new element ϵ, which acts specifically as a unit.

We define the RA in Coq. The definitions are shown in Figure 3.18. The definitions are for the most part exactly as described above. One small exception is the `union_with` function used in the composition operator. While we do have a union (`U`) operator for finite maps available, it is not commutative when the operands have intersecting domains. We know that the composition of maps with intersecting domains is never valid. However, even though we do not care much about invalid elements, the definition of a RA requires that the operator is commutative, even for invalid elements. The function `union_with` is a generalization of the union operator, where we provide a function that decides what to do whenever a key is used in both operand maps. As mentioned above, we do not really care about invalid elements. We just need to make sure that all the requirements for a RA are satisfied, so we just pick a constant dummy value (the empty list) that we use in each case.

We can show that the following local updates hold:

```
Context {K V: Type} `{FinMap K M}.

Inductive partial_table : Type :=
| PTCar : (M (list V) * multiset K) -> partial_table
| PTBot : partial_table
| PTUnit : partial_table.

Instance partial_table_op : Op partial_table :=
  fun X Y => match X, Y with
             | PTCar (m1, d1), PTCar (m2, d2) =>
              if (decide (map_Forall (fun k _ => k ∈ d1) m1 /\
                          map_Forall (fun k _ => k ∈ d2) m2))
              then PTCar (union_with (fun _ _ => Some []) m1 m2,
                          d1 ∪ d2)
              else PTBot
             | X, PTUnit => X
             | PTUnit, Y => Y
             | _, _ => PTBot
             end.

Instance partial_table_core : PCore partial_table :=
  fun _ => Some PTUnit.

Instance partial_table_valid : Valid partial_table :=
  fun X => match X with
           | PTCar (m, d) =>
             multiset_no_dup d /\
             forall k, (is_Some (m !! k) -> k ∈ d)
           | PTUnit => True
           | PTBot => False
           end.
```

Figure 3.18: The definition of the RA `partial_table`

```
Lemma partial_table_local_update_partial_alter m1 d1 m2 d2 f k :
  k ∈ d2 ->
  (PTCar (m1, d1), PTCar (m2, d2)) ~l~>
  (PTCar (partial_alter f k m1, d1), PTCar (partial_alter f k m2, d2)).
```

Here `partial_alter f k m` is the most general operation for changing an entry in a map. The function `f` is called with the value stored under key `k` in `m`, or `None` if `k` is not bound. If `f` returns with a value, then `partial_alter` returns `m`, where `k` is mapped to said value, and if `f` returns `None`, then `partial_alter` removes the entry at `k` from `m`. Both `insert` and `delete` are implemented as specializations of `partial_alter`. In other words, this lemma states that we can make any change at any key as a local update if the key is within the multiset.

With this, we can prove derived specifications for the concurrent hash table module. These specification are shown in Figure 3.19 and are all derived from the general specifications in Figure 3.16. They use the authoritative RA constructed from the `partial_table` RA. We define two predicates `partial_inv` and `partial_own`. The former contains the authoritative element and is given to `is_table` as the invariant. The latter represents ownership of part of the table. Specifically, `partial_own γ m d` states that there exists some table `m'` associated with ghost name `γ`, and `m` contains exactly the entries of `m'`, where the keys are contained in `d`. In addition, it gives exclusive permission to perform any operation on the keys contained in `d`.

In the specification for `create`, we allocate a new ghost variable with initial state `● PTCar (∅, ⊤) · ○ PTCar (∅, ⊤)`. We then split it and give the authoritative part to the general specification, and return the (currently complete) fragment in the postcondition.

In the specification for `insert`, in order to use the general specification we need to show the following:

```
∀ m, partial_inv γ m ={⊤\↑N}=* partial_inv γ (insert_val m k x)
```

We can perform this update by supplying the `partial_own γ m d` that we have available. We also instantiate the parameter `Q` in the general specification with `partial_own γ (insert_val m k x) d`, which allows us to get the ownership back when the operation is finished.

In the specification for `remove`, we need to handle two cases: The case where there is something to remove, and the case where there is not. In the former case, we instantiate the parameter `Q` with the following:

```
(∃ xs, ⌜m !! k = Some (x :: xs)⌝) * partial_own γ (remove_val m k) d)
```

The first part ensures that the returned value is indeed the correct one, while the second part ensures that we get the ownership back. In the latter case, we instantiate `Q'` with `⌜m !! k = None⌝ * partial_own γ m d`. The first part allows us to prove that if the operation returns `NONE`, then it is because there really is nothing to return. Ignoring the second part for now, this means that we have to prove the following:

```
Definition partial_inv γ m : iProp Σ := own γ (● PTCar (m, ⊤)).
Definition partial_own γ m d : iProp Σ := own γ (◯ PTCar (m, d)).

Lemma partial_table_create_spec N n:
  n > 0 ->
  WP create_table #n
     {{t, ∃ γ, is_table N (partial_inv γ) t * partial_own γ ∅ ⊤}}.

Lemma partial_table_insert_spec N γ m d k k' x t:
  as_key k' = Some k ->
  k ∈ d ->
  {{{is_table N (partial_inv γ) t * partial_own γ m d }}}
    table_insert t k' x
  {{{RET #(); partial_own γ (insert_val m k x) d}}}.

Lemma partial_table_lookup_spec N γ m d k k' t:
  as_key k' = Some k ->
  k ∈ d ->
  {{{is_table N (partial_inv γ) t * partial_own γ m d }}}
    table_lookup t k'
  {{{RET match m !! k with
         | Some (v :: _) => SOMEV v
         | _ => NONEV end ;
     partial_own γ m d}}}.

Lemma partial_table_remove_spec N γ m d k k' t:
  as_key k' = Some k ->
  k ∈ d ->
  {{{is_table N (partial_inv γ) t * partial_own γ m d }}}
    table_remove t k'
  {{{RET match m !! k with
         | Some (v :: _) => SOMEV v
         | _ => NONEV end ;
     partial_own γ (remove_val m k) d}}}.
```

Figure 3.19: Derived specifications

```
Definition test_2 : expr :=
  let: "t" := create_table #10 in
  table_insert "t" #1 #1 ||| table_insert "t" #2 #2 ;;
  (table_lookup "t" #1,  table_lookup "t" #2).
```

Figure 3.20: Very simple program

```
∀ m', ⌜m' !! k = None⌝ -* partial_inv γ m'
      ={⊤\↑N}=* partial_inv γ m' * ⌜m !! k = None⌝
```

Even though we do not perform any ghost updates, we still need to provide `partial_own γ m d` in order to establish `m !! k = m' !! k`[2]. Since we provide this proposition, we also need to get it back, hence the second part. Also note that we use `partial_own γ m d` in when proving proofs for both the success and the failure case. The reason we can do this is that the two propositions are conjoined using regular conjunction rather than separating conjunction.

Finally, the `lookup` specification is proven in almost the exact same way as the `remove` specification. The only difference is that we do not need to perform a ghost update in the case where something is returned.

We illustrate how these derived specifications can be used through a very simple program. The program, which is shown in Figure 3.20, creates a table and performs two insertions in parallel using two different keys. We want to prove the following specification:

```
WP test_2 {{v, ⌜v = (SOMEV #1, SOMEV #2)⌝}}
```

The proof of this specification goes as follows as follows: We first use the specification `partial_table_create_spec` to obtain full ownership of the newly created table represented by the assertion `partial_own γ ∅ ⊤`. We then partition the ownership into two parts: One covering all even keys and one covering all odd keys (we could just as well have chosen any other partitioning as long as 1 and 2 become separated). The result is the following proposition:

```
partial_own γ ∅ {[n | Odd n]} * partial_own γ ∅ {[n | Even n]}
```

The next step is to verify the two parallel insertions. Since we now have two separate ownership assertions, we can give one to each thread. We give ownership of the odd keys to the first thread (the one inserting at key 1) and ownership of the even keys to the second thread. For each thread, we use `partial_table_insert_spec` to verify the insertion. This leaves us with `partial_own γ {[1 := [#1] ]} {[n | Odd n]}` in the first thread and `partial_own γ {[2 := [#2] ]} {[n | Even n]}` in the second thread. When the two threads join, we get both of these propositions back. These can

[2]We actually need to establish this in all cases, including in `insert` and the success case for `remove`. In those cases, however, we need the proposition anyway to perform the ghost update

be recombined to obtain `partial_own γ {[ 1 := [#1] ; 2 := [#2] ]} ⊤`, which we use together with `partial_table_lookup_spec` to finish the proof. Strictly speaking we do not need to recombine the two propositions in this particular case, as we can also choose to use only the partial ownerships to perform the lookups.

Chapter 4

Discussion and Perspectives

4.1 Using Iris

Iris is a quite recent framework that is still being developed further, and as such has not been used for proving specifications of non-trivial programs very often. In this project, Iris has been used for verifying non-trivial program modules. During this process, several features of Iris have been used. We used higher-order specifications as part of the specification for `fold`, a higher-order function, we used invariants as part of verifying the concurrent hash table module, and we used custom ghost state to provide ownership of part of a concurrent hash table.

The Iris logic is formalized using quite complex semantics that rely on step-indexing [1] to ensure consistency (an explanation of step-indexing is beyond the scope of this report). Nonetheless, we have been able to ignore step-indexing almost entirely when proving the specifications presented in this report. For the high-level descriptions of the specifications and proofs presented here in this report, we did not even have to mention step-indexing. While writing the actual proofs in Coq, the only signs of step-indexing, that become present, were a few occurrences of the later modality $\triangleright$ (which states that a proposition holds after one step), and in essentially all of these cases it appeared in a context where it could easily be eliminated. Some of the more interesting cases where this happened, was cases were Löb induction was used. By Löb induction we mean a proof that uses the Löb rule $\triangleright P \Rrightarrow P \vdash P$, which states that when proving a proposition, we are allowed to assume that the proposition holds after one step. In our case, it was used to prove statements about recursive programs. In particular, it was used to prove specifications for functions that iterated over an array of buckets.

Being able to do the work in a proof assistant like Coq has been quite helpful. Coq ensures that we never make any mistakes in our proofs without them being caught. This is particularly useful when the proofs involve a lot of small details, as it becomes very easy to accidentally overlook something when doing such proofs by hand. Many of the proofs used in this project have involved a lot of details, including almost all presented specifications. The drawback of using Coq is that it requires everything to be proven in perfect detail, which means that one often spends time proving small details that would

barely even be present in hand written proofs. For instance, Coq has separate types for the sets $\mathbb{N}$ and $\mathbb{Z}$ (`nat` and `Z` respectively), which means we often have to explicitly convert between the two when proving statements that involve both types.

The perhaps biggest advantage of using Coq is that it makes it easier to make changes. In general, changing a definition may cause proofs that involve this definition to become invalid until the proof has been changed accordingly. It is not always obvious which changes are needed. When doing the proofs by hand, it may be hard enough that the only way to ensure that everything has been fixed is to go through all proofs that depend on the definition in detail. When using Coq, however, this becomes unnecessary. Instead, after making the change, we let Coq process all existing proofs. Whenever Coq encounters something that is now incorrect, it will report the error. We then fix the error and repeat until Coq reports no errors. Thus we end up focusing only on what needs to be fixed and nothing else.

4.2 Related work

Similar work on specifying hash tables for sequential use was done by Pottier [21], and the specifications presented in §3.3 uses many of the same ideas used in that work. His specifications are written in CFML [5], a tool that parses an OCaml program and outputs a characteristic formula from which specifications can be proven in Coq using higher-order separation logic. His work, however, is limited to sequential program and does not mention concurrency. It is part of a larger project called Vocal, which aims to develop a library of verified OCaml data structures and algorithms.

Filliâtre and Pereira [9] present a very general way of specifying iteration. Their approach generalizes not just to data structures, but to any algorithm that produces a sequence of elements, even infinite sequences. The general idea is to describe the produced sequences using predicates *enumerated* and *completed*. The former describes the possible sequences that can be produced by iteration, while the latter describes the complete sequences that can be produced. The predicates `permitted` and `complete` used in this report can be seen as concrete instances of *enumerated* and *completed*. Additionally, they verify some concrete implementation of iterators using the tool Why3.

Krishnaswami et al. [19] present general specifications and proofs of common design patterns using separation logic, including iterators for collections. When a collection is modified, all existing iterators for that collection are invalidated. This is handled by describing both the collection and the iterators with predicates parameterized with an abstract state of the collection. The specifications for the iterator operations require that the state parameters of the collection and the iterator match. This approach is similar to the approach used in this report. In addition, they also support composite iterators, i.e., iterators constructed from other iterators using operations like *filter*, which given an iterator and a predicate (i.e., a function from elements to boolean) returns a new iterator which skips the elements from the old iterator that do not satisfy the predicate, and *map2*, which given two iterators and a binary function returns an iterator producing elements that are the result of applying the function to the elements given by the old

iterators. They prove the specifications in Coq using the Ynot extension.

Haack and Hurlin [11] propose multiple specifications for Java iterators. They propose protocols that avoids errors such as concurrent modification and going out of bounds, and use separation logic to enforce them. They handle iterators that modify the underlying collections through methods such as `iterator.remove`. Concurrent modification is avoided by letting the iterators own the underlying collection. They support fractional permissions, which allows multiple read-only iterators or a single read-write iterator to exist.

Da Rocha Pinto et al. [7] present a general abstract specification of concurrent indexes. They use the term index to describe any data structure that stores data as key-value pairs, where entries are identified by their keys. Their specifications are based on Concurrent Abstract Predicates [8], which provide a separation logic for verifying concurrent programs. Their approach is to split the index based on keys and allow threads to take ownership of individual keys. This approach is similar to what we achieved with our partial table ownership in §3.4.4, though we allow splitting into potentially infinite disjoint sets of keys, while they handle keys individually. They also allow multiple variants of shared access to a single key using fractional permissions, while our solution currently only support exclusive ownership.

Xiong et al. [22] present an abstract specification for concurrent maps in Java. They also verify that the `ConcurrentSkipListMap` class in the Java standard library satisfy this specification. The specification is based on the TaDA program logic [6], which uses the notion of atomic triples. An atomic triple is used to specify the behavior of an operation that is logically atomic, which means that to all other threads in the program it looks like all the changes made by the operation happen in a single step.

4.3 Conclusion

We have described and proven specifications for two implementations of hash tables. Both implementations are parameterized with a set of keys (which need not correspond to a data type in common programming languages) and can store multiple values for each key. The first implementation is a classic implementation for sequential use only. It automatically increases the number of buckets when the number of entries grow large and supports iteration through two higher-order operations: `fold` and `cascade`. The second implementation is a concurrent implementation. The specifications describes hash tables using a higher-order predicate, that asserts no ownership by itself and can be freely shared. We also showed how ownership of part of the table can be obtained by choosing the right parameter for the predicate.

4.4 Future work

The implementation of concurrent hash tables used here is very minimal and leaves out several features that would often be present in implementations used in actual systems. The first missing feature is automatic resizing of the array of buckets as the number

of elements grow. The challenge here is to transfer all elements in the table to a new array, while making sure that all changes performed concurrently with the transfer are preserved. Creating an implementation that does this is not trivial and verifying correctness may be even less so.

Another missing feature is operations for iteration. The question here would be what the specifications would look like. In the special case where the client has full ownership of the table, the specification would be similar to what we have for the sequential hash table, however, in the general case this is not so. In particular, it may not be certain which elements are visited if the table is modified while iteration occurs.

Both implementations use functional lists as buckets, however, other data structures (e.g. mutable lists and trees) could be used as well. Da Rocha Pinto et al. [7] argue that the type of data structure used for buckets can be abstract in the definition of a hash table. The implementation of hash tables would thus be parameterised with another type of tables. All operations on the tables would make use of the corresponding operations of the buckets provided by the table interface.

Future development of Iris may lead to the possibility of simpler specifications for concurrent hash tables than what we have shown. Xiong et al. [22] propose simple specifications that use atomic triples. Like regular Hoare triples, an atomic triple specifies a precondition, which must hold before an operation is performed, and a postcondition, which will hold after the operation. However, in addition, it states that the operation will be logically atomic, which means that to all other threads it will look like the operation was performed in a single step. In other words, the program will, for all purposes, never be in an intermediary state where neither the precondition nor the postcondition holds. Atomic triples were in fact part of Iris 1.0 [15], however, Iris 3.0 included changes that resulted in atomic triples, as defined previously, no longer working. If atomic triples return to Iris in the future, then it might be possible to prove specifications similar to those proposed by Xiong et al.

Bibliography

[1] Andrew W. Appel and David McAllester. "An Indexed Model of Recursive Types for Foundational Proof-carrying Code." In: *ACM Trans. Program. Lang. Syst.* 23.5 (Sept. 2001), pp. 657–683. ISSN: 0164-0925. DOI: 10.1145/504709.504712. URL: http://doi.acm.org/10.1145/504709.504712.

[2] Lars Birkedal et al. *Notes on Iris*. 2017.

[3] Richard Bornat et al. "Permission Accounting in Separation Logic." In: *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '05. Long Beach, California, USA: ACM, 2005, pp. 259–270. ISBN: 1-58113-830-X. DOI: 10.1145/1040305.1040327. URL: http://doi.acm.org/10.1145/1040305.1040327.

[4] John Boyland. "Checking Interference with Fractional Permissions." In: *Proceedings of the 10th International Conference on Static Analysis*. SAS'03. San Diego, CA, USA: Springer-Verlag, 2003, pp. 55–72. ISBN: 3-540-40325-6. URL: http://dl.acm.org/citation.cfm?id=1760267.1760273.

[5] Arthur Charguéraud. "Characteristic Formulae for the Verification of Imperative Programs." In: *SIGPLAN Not.* 46.9 (Sept. 2011), pp. 418–430. ISSN: 0362-1340. DOI: 10.1145/2034574.2034828. URL: http://doi.acm.org/10.1145/2034574.2034828.

[6] Pedro da Rocha Pinto, Thomas Dinsdale-Young, and Philippa Gardner. "TaDA: A Logic for Time and Data Abstraction." In: *Proceedings of the 28th European Conference on ECOOP 2014 — Object-Oriented Programming - Volume 8586*. New York, NY, USA: Springer-Verlag New York, Inc., 2014, pp. 207–231. ISBN: 978-3-662-44201-2. DOI: 10.1007/978-3-662-44202-9_9. URL: http://dx.doi.org/10.1007/978-3-662-44202-9_9.

[7] Pedro da Rocha Pinto et al. "A Simple Abstraction for Complex Concurrent Indexes." In: *SIGPLAN Not.* 46.10 (Oct. 2011), pp. 845–864. ISSN: 0362-1340. DOI: 10.1145/2076021.2048131. URL: http://doi.acm.org/10.1145/2076021.2048131.

[8] Thomas Dinsdale-Young et al. "Concurrent Abstract Predicates." In: *Proceedings of the 24th European Conference on Object-oriented Programming*. ECOOP'10. Maribor, Slovenia: Springer-Verlag, 2010, pp. 504–528. ISBN: 978-3-642-14106-5. URL: http://dl.acm.org/citation.cfm?id=1883978.1884012.

[9] Jean-Christophe Filliâtre and Mário Pereira. "A Modular Way to Reason About Iteration." In: *Proceedings of the 8th International Symposium on NASA Formal Methods - Volume 9690*. NFM 2016. Minneapolis, MN, USA: Springer-Verlag New York, Inc., 2016, pp. 322–336. ISBN: 978-3-319-40647-3. DOI: 10.1007/978-3-319-40648-0_24. URL: http://dx.doi.org/10.1007/978-3-319-40648-0_24.

[10] Erich Gamma et al. *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN: 0-201-63361-2.

[11] C. Haack and C. Hurlin. "Resource Usage Protocols for Iterators." In: *Journal of Object Technology* 8.4 (June 2009). Ed. by M. Huisman and P. Müller, pp. 55–83. URL: http://www.jot.fm/issues/issue_2009_06/article3.pdf.

[12] *Iris Project*. URL: http://iris-project.org/.

[13] *Java™Platform, Standard Edition 8 API Specification*. Oracle. URL: https://docs.oracle.com/javase/8/docs/api/.

[14] Ralf Jung et al. "Higher-order Ghost State." In: *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*. ICFP 2016. Nara, Japan: ACM, 2016, pp. 256–269. ISBN: 978-1-4503-4219-3. DOI: 10.1145/2951913.2951943. URL: http://doi.acm.org/10.1145/2951913.2951943.

[15] Ralf Jung et al. "Iris: Monoids and Invariants As an Orthogonal Basis for Concurrent Reasoning." In: *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '15. Mumbai, India: ACM, 2015, pp. 637–650. ISBN: 978-1-4503-3300-9. DOI: 10.1145/2676726.2676980. URL: http://doi.acm.org/10.1145/2676726.2676980.

[16] Robbert Krebbers. *Coq-std++ Git repository*. URL: https://gitlab.mpi-sws.org/robbertkrebbers/coq-stdpp.

[17] Robbert Krebbers, Amin Timany, and Lars Birkedal. "Interactive Proofs in Higher-order Concurrent Separation Logic." In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. POPL 2017. Paris, France: ACM, 2017, pp. 205–217. ISBN: 978-1-4503-4660-3. DOI: 10.1145/3009837.3009855. URL: http://doi.acm.org/10.1145/3009837.3009855.

[18] Robbert Krebbers et al. "The Essence of Higher-Order Concurrent Separation Logic." In: *Proceedings of ESOP 2017*. 2017.

[19] Neelakantan R. Krishnaswami et al. "Design Patterns in Separation Logic." In: *Proceedings of the 4th International Workshop on Types in Language Design and Implementation*. TLDI '09. Savannah, GA, USA: ACM, 2009, pp. 105–116. ISBN: 978-1-60558-420-1. DOI: 10.1145/1481861.1481874. URL: http://doi.acm.org/10.1145/1481861.1481874.

[20] Peter W. O'Hearn. "Resources, Concurrency, and Local Reasoning." In: *Theor. Comput. Sci.* 375.1-3 (Apr. 2007), pp. 271–307. ISSN: 0304-3975. DOI: 10.1016/j.tcs.2006.12.035. URL: http://dx.doi.org/10.1016/j.tcs.2006.12.035.

[21] François Pottier. “Verifying a hash table and its iterators in higher-order separation logic.” In: *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs (CPP 2017)*. Jan. 2017, pp. 3–16. DOI: 10.1145/3018610.3018624.

[22] Shale Xiong et al. “Abstract Specifications for Concurrent Maps.” In: *Programming Languages and Systems: 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22–29, 2017, Proceedings*. Ed. by Hongseok Yang. Berlin, Heidelberg: Springer Berlin Heidelberg, 2017, pp. 964–990. ISBN: 978-3-662-54434-1. DOI: 10.1007/978-3-662-54434-1_36. URL: http://dx.doi.org/10.1007/978-3-662-54434-1_36.