The Nextgen Modality: A Modality for Non-Frame-Preserving Updates in Separation Logic

Simon Friis Vindum
Department of Computer Science
Aarhus University
Denmark
vindum@cs.au.dk

Aïna Linn Georges
Aarhus University
Denmark
ageorges@cs.au.dk

Lars Birkedal
Department of Computer Science
Aarhus University
Denmark
birkedal@cs.au.dk

Abstract

As separation logic is a logic of resources, the way in which resources can soundly change and be updated is a fundamental aspect. Such changes have typically been restricted to certain *local* or *frame-preserving* updates. However, recently we have seen separation logics where the restriction to frame-preserving updates seems to be a hindrance towards achieving the ideal program reasoning rules. In this, paper we propose a novel *nextgen* modality that enables reasoning across *generations* where each generational change can update resources in ways that are non-local and non-frame-preserving. We implement the idea as an extension to the Iris base logic, which enriches Iris with an entirely new capability: the ability to make non-frame-preserving updates to ghost state. We show that two existing Iris modalities are special cases of the nextgen modality. Our "extension" can thus also be seen as a generalization and simplification of the Iris base logic. To demonstrate the utility of the nextgen modality we use it to construct a separation logic for a programming language with explicit stack allocation and with a return operation that clears entire stack frames. The nextgen modality is used to great effect in the reasoning rule for return, where a modular and practical reasoning rule is otherwise out of reach. This is the first separation logic for a high-level programming language with stack allocation. We sketch ideas for future work in other domains where we think the nextgen modality can be useful.

CCS Concepts: • **Theory of computation → Separation logic**; **Logic and verification**.

Keywords: separation logic, Iris, Coq, program logic, modality

ACM Reference Format:
Simon Friis Vindum, Aïna Linn Georges, and Lars Birkedal. 2025. The Nextgen Modality: A Modality for Non-Frame-Preserving Updates in Separation Logic. In *Proceedings of the 14th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP '25), January 20–21, 2025, Denver, CO, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3703595.3705876

This work is licensed under a Creative Commons Attribution 4.0 International License.

CPP '25, January 20–21, 2025, Denver, CO, USA
© 2025 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1347-7/25/01
https://doi.org/10.1145/3703595.3705876

1 Introduction

Separation logic is a logic for reasoning about ownership over resources. A crucial aspect of separation logic is the ability to perform updates to resources that are *frame-preserving*. As a quintessential example of a frame-preserving update, consider the separation logic proof rule for assignment:

$$\{\ell \hookrightarrow v\}\, \ell \leftarrow w \,\{\ell \hookrightarrow w\}.$$

Here, the difference between the points-to assertion in the pre- and postcondition means that the heap resource is *updated*. The update is sound because the points-to assertion implies *exclusive* ownership over the location ℓ. This ensures that changing this part of the heap resource is guaranteed to not interfere with any other assertions. Thus, any other assertion, or frame, P that is valid in combination with $\ell \hookrightarrow v$ (meaning that $P * \ell \hookrightarrow v$ is not false) is also valid in combination with $\ell \hookrightarrow w$. This "locality" of the update is required in order for the assignment rule to combine soundly with the *frame rule* for Hoare triples.

In the separation logic Iris [8, 9], which we use in this paper, updates to resources that satisfy this property of being sound in combination with the frame rule are called *frame-preserving updates*. Such updates are local in the sense that the changes they allow one to perform are closely related to the resources and knowledge one owns locally. This makes them well-suited for reasoning about programming language features that make similarly local changes to the physical state. For instance, writing to a reference, as above, which affects a small and precisely delineated fragment of the physical state.

In Iris, resource updates are limited to those that are frame-preserving. But updates that preserve the frame in this manner are not a natural fit for program execution steps that are less localized and that make sweeping changes to larger parts of the physical state. Examples of such non-local execution steps that can not easily be expressed as frame-preserving updates include:

Crashes in a setting with durable storage If one wishes to verify crash-safety in a setting with durable storage,

how the state of a machine changes at a crash can be represented as a *crash-step* in the operational semantics. At such a step many locations might be lost. For instance, *all* locations residing in volatile memory are lost. This makes the crash step non-local. Recently, we have seen several separation logics for reasoning about crashes and durable storage [2, 4, 5, 19]. They have all had to work around the absence of non-frame-preserving updates. To this end Chajed et al. [5] introduced a *post-crash modality*, a variant of which was also used in Vindum and Birkedal [19].

Garbage collection step In a language with garbage collection, for which the operational semantics explicitly models the action of the garbage collector, a step corresponding to garbage collection would reclaim a potentially large number of locations in memory. If one wishes to reason about the garbage collection step, without having gathered all the resources for all the locations that the garbage collector might need to collect, then the update to the logical resources is not frame-preserving. Indeed, in recent work on separation logics in the presence of garbage collection, this has been worked around by having the logics maintain a global account of all the resources that could be reclaimed by the garbage collector, for instance in the form of an explicit correspondence between physical addresses and logical addresses [7, 13, 14]

Function call returns with stack deallocation In a language with stack-allocated values, returning from a function invalidates the locations in the entire stack frame corresponding to the function call. This example is explained in greater detail later, as we use it as a case study in this paper.

There are also cases where the desire to make non-frame-preserving updates arises without stemming from the operational semantics. For example, in the work on temporary read-only permissions by Charguéraud and Pottier [6]. Mediated by a *read-only modality* their work allows one to go from a normal (read-write) points-to assertion, to a freely shareable read-only points-to assertion, and then later recover the full points-to assertion again. Here, the last move from read-only back to read-write is not frame preserving: A read-only point-to assertion is valid in a frame consisting of itself (since it is freely duplicable) but a normal points-to assertion is not valid in the same frame (since the normal points-to assertion implies exclusive access to a location). Hence, is this case the desired non-frame-preserving update is not due to the operational semantics invalidating resources, but instead to revert a previous resource update purely at the logic level. Charguéraud and Pottier were able to allow this non-frame-preserving update in their logic by explicitly building it in to the model of the logic and proving it sound in the model.

We hasten to emphasize that for the above examples, it is of course not the case that creating a logic for a particular language or verifying programs with a particular characteristic is impossible without the ability to make non-frame-preserving updates. By using sophisticated resources, advanced invariants, or straight-up workarounds, it is often possible to make do without the ability to make non-frame-preserving updates. Rather, it is the case that certain *specific* desirable program rules and verification approaches are not possible since they can not be encoded as frame-preserving updates. One good example of this is the previously mentioned work by Charguéraud and Pottier on temporary read-only permissions. It is not that their approach makes it possible to verify entirely new classes of programs as temporary read-only points-to predicates can also be achieved with the bookkeeping overhead of fractional permissions. Rather, the benefit of their approach is that it is simple and elegant, and to achieve the particular rules in their program logic, non-frame-preserving updates are necessary.

In this paper, we present a novel modality that facilitates making changes to resources in ways that are *not* frame-preserving. The modality is called the **nextgen** modality since it supports reasoning about what happens "in the next generation," after a non-frame-preserving update. The modality makes it possible to change resources as described by any (well-behaved) *transformation* function chosen by the user of the logic. We develop the modality as an extension of Iris. Usually, new features for Iris are developed *within* the logic. But since Iris, at the fundamental level of its *base logic* provides no means for expressing the kind of non-local updates that we are interested in, we have to extend the base logic itself. This is a very powerful and general extension to the logic, which can then serve as the building block for creating more specialized constructs for individual applications. We remark that the Iris base logic has been relatively stable since 2017 [10, 18] (except for experiments with transfinite versions of Iris [15]) and find it noteworthy that this is one of the instances where the base logic needs changing.

Of course, we can not in a single paper develop entire program logics for all the motivating examples above. Instead, we choose to focus on one of them, namely a program logic for a language with stack-allocated values. The main purpose of this case study is to demonstrate how to use our nextgen modality, but it is also a contribution in its own right. In the program logic, we use the nextgen modality to account for the way in which returns invalidate the call stack frame. The result is the first separation logic that supports reasoning about a high-level language with stack allocation and where, in the operational semantics, returning from a function invalidates the call stack frame of the returning function.

In summary, the contributions of the paper are as follows:

- We extend the Iris base logic with a new modality, the nextgen modality. This modality makes it possible to make non-frame-preserving changes to resources in Iris which was previously not possible. We have extended the Iris implementation in Coq to include the new modality and also adapted the Iris Proof Mode [11] to include support for the nextgen modality.
- We develop a program logic for StackLang, a language where the physical state contains a call stack and values can be allocated on the stack. Returning from a function clears the call stack from the returning function from the physical state. By using the nextgen modality in the proof rule for returns we arrive at a proof rule that is simple and easy to use. We have formalized the new program logic and examples using it in Coq in our extended version of the Iris implementation.

Our Coq mechanization is available online at https://github.com/logsem/iris-nextgen and as an artifact accompanying this paper [20].

The rest of the paper proceeds as follows. In Section 2 we give the necessary Iris background to explain our contributions, and we describe the most closely related work, the aforementioned post-crash modality. Section 3 introduces the *basic* nextgen modality, its rules in the logic, and its model. We explain how the nextgen modality is an improvement compared to the post-crash modality and show how the nextgen modality generalizes the persistently and the plainly modalities in Iris. Section 4 describes the operational semantics of StackLang and the program logic we construct for it. In Section 5 we compare against related work not covered earlier in the paper and discuss future work.

2 Background and Related Work

We first cover a bit of Iris background and the most closely related work. The background material includes some aspects of Iris that are perhaps not part of the typical Iris user's repertoire of Iris features, but that, nevertheless, are important in order to explain our contributions and situate them in comparison to the related work.

2.1 Iris Background

A central feature of Iris is its support for user-defined ghost state. Users of the logic can define and choose their own *resource algebras* (RAs) to capture the behavior of their desired ghost state. With much flexibility, one can mix and match RAs and use many of them in the logic. For any RA A and element $a \in A$, the proposition $\boxed{a}^{\gamma}$ asserts ownership over a at some *ghost location* distinguished by a *ghost name* $\gamma \in \mathit{GName}$. We do not recall the full definition of what an RA is, but it includes an associative and commutative monoidal operation, which gives meaning to separating conjunction and ownership, cf. the GHOST-OP bi-entailment in Figure 1. As not all combinations are meaningful, a validity predicate $\mathcal{V} : A \to \mathit{Prop}$ identifies the valid elements. The logic maintains the property that only valid elements can be owned, cf. GHOST-VALID. A partial function called the core $|-|$ extracts from elements their *duplicable* part. That is, for every $a \in A$, its core $|a|$ is the duplicable part of a, meaning in particular that $|a| = |a| \cdot |a|$. The persistently modality $\Box$ removes all non-duplicable resources by applying the core operation to all resources, cf. GHOST-PERSISTENTLY. Elements of an RA are ordered w.r.t. an *extension order*: $a \preccurlyeq b \triangleq \exists c.\, a \cdot c = b$. A resource a can be updated to another resource b via a frame-preserving update denoted $a \rightsquigarrow b$ and defined as:

$$a \rightsquigarrow b \triangleq \forall c. \mathcal{V}(a \cdot c) \Rightarrow \mathcal{V}(b \cdot c)$$

This definition matches the intuition we gave in the introduction: a resource can be updated as long as it remains valid in combination with any frame with which it was also valid before. Frame-preserving updates are internalized into the logic through the update modality $\Rrightarrow$ and the rule GHOST-UPDATE.

Both the ability to use several RAs and the ghost ownership assertion $\boxed{a}^{\gamma}$ are not present in the Iris *base logic*, but are provided by constructions that are *defined* within the base logic. Instead, the Iris base logic is parameterized over just a single "global" RA M and thus, a user of the base logic can in fact pick only a single RA to instantiate the logic with. In the base logic, the assertion $\mathsf{Own}\,(a)$, where $a \in M$, denotes ownership over elements of the single resource RA M.

We now recall the constructions that make it possible to use several RAs and named ghost ownership assertions on top of the base logic. First, one chooses a sequence of all the RAs that are to be used in the logic: $M_1, \ldots, M_n$, where n is the number of RAs. Then, the single global resource algebra M is chosen to be a "resource algebra of resource algebras" in the following way:

$$M \triangleq \prod_{i \in I} \mathit{GName} \xrightarrow{\mathsf{fin}} M_i \tag{1}$$

This construction has two levels. The first level is a product indexed by the number of RAs $I = \{1, \ldots, n\}$. This is such that multiple different RAs can be used. The next level is a finite map over ghost names. This is such that multiple independent instances of the same RA can be used. The set M is itself an RA whose operation simply combines the two layers point-wise. The familiar ghost ownership proposition is now defined in terms of the basic ownership Own assertion:

$$\boxed{a : M_i}^{\gamma} \triangleq \mathsf{Own}\,(\lambda j.\, \text{if } i = j \text{ then } \{\gamma \mapsto a\} \text{ else } \emptyset)$$

We emphasize that the full path to a ghost location consists of both an index $i \in I$ and a ghost name γ. The notation for ownership at ghost locations, however, usually leaves out the i as it can be inferred from the type of the element at the location.

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

GHOST-VALID
$\boxed{a}^{\gamma} \vdash \mathcal{V}(a)$

GHOST-PERSISTENTLY
$\boxed{a}^{\gamma} \vdash \Box \boxed{|a|}^{\gamma}$

GHOST-UPDATE
$$\frac{a \rightsquigarrow b}{\boxed{a}^{\gamma} \vdash \dot{\Rightarrow} \boxed{b}^{\gamma}}$$

OWN-OP
$\mathsf{Own}\,(a) * \mathsf{Own}\,(b) \dashv\vdash \mathsf{Own}\,(a \cdot b)$

OWN-VALID
$\mathsf{Own}\,(a) \vdash \mathcal{V}(a)$

OWN-PERSISTENTLY
$\mathsf{Own}\,(a) \vdash \Box\, \mathsf{Own}\,(|a|)$

Figure 1. A few of the Iris rules related to ghost state.

For modularity, proofs carried out in Iris do not specify exactly what the sequence of available RAs should be. Instead, they require that M has the form above, and that indices exist in the sequence of RAs that contain the RAs necessary for the given proof. For instance, if a proof requires an RA A then the proof will simply assume that there exists an $i \in I$ such that $M_i = A$. A proof with such a requirement can modularly be combined with other proofs making similar constraints. Only to obtain a closed "final" proof does one need to fully determine M, and at this point one can do so while ensuring that it contains all the RAs required by sub-proofs.

A program logic constructed inside of Iris usually relies on certain *global ghost names* for ghost locations that contain ghost state used by the program logic itself. We use $\vec{\gamma}$ to refer to such a collection of global ghost names. For instance, an Iris-based program logic for a programming language with a heap keeps ghost state for the heap at a global ghost name. In other words, there would be a ghost name inside $\vec{\gamma}$ specifically for the heap ghost state which we could write as $\vec{\gamma}.\mathrm{heap}$. Since points-to assertions are modeled using this ghost state they make use of the global ghost name. Global ghost names are usually left implicit both on paper and in Coq, but we could write a points-to assertion like this

$$\ell \hookrightarrow^{\vec{\gamma}.\mathrm{heap}} v$$

to make explicit the ghost name it makes use of. That is, points-to assertions are in fact parameterized over the global ghost name that they use. Similarly, as the concrete values of the global ghost names do not matter, proofs and the program logic itself are parameterized over the collection of the global ghost names. When proofs are carried out in Coq the global ghost names are assumed as an implicit context parameter. On paper one should imagine that there is an implicit "$\forall \vec{\gamma}$." in the beginning of proofs, making the names $\vec{\gamma}$ always "in scope".

2.2 Perennial's Post-Crash Modality

The work most closely related to ours is the *post-crash modality* by Tej Chajed and Joseph Tassarotti [3].[1] They developed the modality specifically for reasoning about crashes in the Perennial program logic [1, 5], but the idea behind their modality can also be applied more generally to reason about the kind of non-local resource changes we have described. We continue to use the name post-crash modality, but emphasize that the modality is not only applicable to reasoning about crashes.

Perennial is a program logic for proving crash-safety in a setting with volatile memory and a durable disk. The post-crash modality, $\Diamond$, is used to express the way in which resources change due to a system crash. As an example, the modality discards resources that correspond to the parts of the physical state that reside in volatile memory and preserves resources that correspond to the parts of the physical state that reside on the durable disk. As mentioned in the introduction, the change to the physical state that occurs at a crash can not be expressed as a frame-preserving update to the ghost state for the physical state.

The key idea of the post-crash modality is to forgo updating the existing ghost state and instead allocate *new* ghost locations. That is, instead of updating a resource $\boxed{a}^{\gamma}$ to $\boxed{b}^{\gamma}$, which would require there to be a frame-preserving update $a \rightsquigarrow b$, a *new* ghost ownership assertion $\boxed{b}^{\gamma'}$, for a new ghost name γ', is allocated instead. The resources at the new ghost locations need not be frame-preserving updates of the earlier existing resources. Thus, this approach side-steps the issue of not being able to make non-frame-preserving changes to ghost state. Since the new ghost locations have no inherent relation to the old ghost locations, some relationship between the two must be explicitly established, and it is required that one immediately stop using the old ghost name γ and switch to the new ghost name γ'. At a crash, new ghost assertions are allocated for the ghost state used internally in the program logic. Since allocating new ghost assertions results in new ghost names, this has the effect that the global ghost names that the proof is parameterized over are now obsolete as they refer to ghost locations prior to the crash. The post-crash modality then mediates between ghost state for the old global ghost names and ghost state for the new global ghost names.

To do this, the modality does not take an assertion of type iProp as argument, but instead has the type

$$\Diamond : (\mathsf{GlobalGnames} \to \mathsf{iProp}) \to \mathsf{iProp},$$

[1] While crucial to the workings of Perennial, the post-crash modality is unfortunately not described in any of the published papers about Perennial. We therefore directly cite the Coq mechanization of Perennial where the modality appears.

where GlobalGnames is a record of all the ghost names used by the program logic in question (originally, Perennial). The modality is then defined by:

$$[\![\Diamond P]\!] \triangleq \forall \sigma, \sigma', \vec{\gamma}'. R(\sigma, \sigma', \vec{\gamma}, \vec{\gamma}') \twoheadrightarrow P(\vec{\gamma}') * R(\sigma, \sigma', \vec{\gamma}, \vec{\gamma}').$$

The R above is part of the definition of the post-crash modality. It relates the global ghost names and physical state before the crash, σ and $\vec{\gamma}$, with the physical state after the crash σ' and the new global ghost names $\vec{\gamma}'$.

When used, the post-crash modality is usually given an argument of the form $\lambda \vec{\gamma}'. Q$ where Q has to use the ghost names in $\vec{\gamma}'$ for its global ghost names and *not* use the old global ghost names $\vec{\gamma}$. Following this, rules for the post-crash modality are of the form $P \vdash \Diamond(\lambda \vec{\gamma}'. Q)$. When proving soundness of such rules one ends up with goals of the form

$$P * R(\sigma, \sigma', \vec{\gamma}, \vec{\gamma}') \twoheadrightarrow Q(\vec{\gamma}') * R(\sigma, \sigma', \vec{\gamma}, \vec{\gamma}').$$

For instance, to prove soundness of the rule

$$\ell \hookrightarrow^{\vec{\gamma}.\mathrm{heap}} v \vdash \Diamond(\lambda \vec{\gamma}'. \ell \hookrightarrow^{\vec{\gamma}'.\mathrm{heap}} v)$$

for points-to assertions, one would have to prove

$$\ell \hookrightarrow^{\vec{\gamma}.\mathrm{heap}} v * R(\sigma, \sigma', \vec{\gamma}, \vec{\gamma}') \twoheadrightarrow \ell \hookrightarrow^{\vec{\gamma}'.\mathrm{heap}} v * R(\sigma, \sigma', \vec{\gamma}').$$

The crux of the proof is to turn the "old" points-to assertion into the "new" points-to assertion. Making this possible is the purpose of the R resource. It serves as a catalyst to make this transition possible, without being consumed itself. In our particular example with points-to assertions, R could be defined as

$$R(\sigma, \sigma', \vec{\gamma}, \vec{\gamma}') \triangleq \ell \hookrightarrow^{\vec{\gamma}.\mathrm{heap}} v \vee \ell \hookrightarrow^{\vec{\gamma}'.\mathrm{heap}} v.$$

Here the disjunction facilitates the "exchange" from the old points-to predicate to the new points-to predicate. More generally, R is defined such that all relevant resources can be "exchanged" from old to new in this manner.

2.3 Limitations of the Post-Crash Modality

We now describe some of the limitations and problematic aspects of the above approach. Later on, we will show how our new nextgen modality addresses these shortcomings.

Poor interaction with the $\Box$ modality. The following rule is not possible to prove for the post-crash modality

$$\Box \Diamond P \vdash \Diamond \Box P.$$

Unfolding the model of the post-crash modality we see that this amounts to proving

$$\Box(\forall \sigma, \sigma', \vec{\gamma}'. R(\sigma, \sigma', \vec{\gamma}') \twoheadrightarrow P(\vec{\gamma}') * R(\sigma, \sigma', \vec{\gamma}')) \vdash$$
$$\forall \sigma, \sigma', \vec{\gamma}'. R(\sigma, \sigma', \vec{\gamma}') \twoheadrightarrow \Box P(\vec{\gamma}') * R(\sigma, \sigma', \vec{\gamma}')$$

We need to be able to show that P holds persistently, but we only know that a wand implying P holds persistently. Since we do not have R persistently, when we apply the wand to R we do not get P persistently. Thus, the lemma can not be proven.

The persistently modality plays a crucial role in Iris and the Iris proof-mode (IPM) for Coq [11]. The IPM keeps a so-called *persistent context* which consists of propositions that hold under the $\Box$ modality. When introducing the post-crash modality (using the `iModIntro` tactic) the IPM requires the rule

$$\frac{P \vdash \Diamond Q}{\Box P \vdash \Diamond \Box Q}$$

in order to be able to transform the persistent context. However, for reasons similar to the above, we can not prove this rule. This makes the post-crash modality more challenging and cumbersome to use in practice in Coq.

Invariants and the post-crash modality. A key feature in Iris is *invariants*. An invariant is denoted $\boxed{I}^{\iota}$ and means that the assertion I is an invariant that a program maintains at every step of execution (the ι is not important for our purposes). It is not clear how invariants that contain ghost state, that uses global ghost names which are changed by the post-crash modality, can work with the post-crash modality as the invariant assertion is constant. At the very least, such invariants would have to be parameterized by the global collection of ghost names for the post-crash modality to be able to update them, and, as such, details of the post-crash modality would leak into invariants. In Section 4.2.4 we give an example of using our nextgen modality together with invariants, where the rules only have natural and necessary changes compared to normal Iris invariants.

No interaction with custom ghost state. As we have seen, the R resource in the model of the post-crash modality facilitates an exchange between old resources and new resources. This means that knowledge of certain global ghost names and resources are baked-in or hard-coded into the definition of the post-crash modality. The implication of this is that the reach of the modality can not extend to user-defined ghost state. Specifically, for an RA A and a ghost location γ, unknown to the definition of $\Diamond$, no rule of the form

$$\boxed{a}^{\gamma} \vdash \Diamond \boxed{b}^{\gamma}$$

where $a \neq b$, can exist. In other words, the only such rule is the one where $a = b$, meaning that the $\Diamond$ modality can have no interaction with user-defined ghost state. This means that it is not possible to use the post-crash modality to give logically atomic specifications for user-defined durable concurrent data structures under a weak consistency model (such as the one in [19]) whose specification relies on user-defined ghost state.

Not principled. While the post-crash modality cleverly works around the limitations in Iris for updating ghost state, we find that the mechanism it uses is not as principled as one could want. As we have shown, the workings of the post-crash modality rely on changing otherwise globally fixed ghost names. This can be confusing, both on paper and

in Coq. For instance, it means that two points-to assertions that are notationally the same, can in fact be different as they "invisibly" use two different ghost names. The post-crash modality does not remove or otherwise invalidate old resources; it is up to the user of the modality to carefully apply lemmas that translate old resources, while also making sure that no old resources are still used. In Coq the modality relies on creating multiple instances of a type class that contains the global ghost names. Having multiple instances of the same type class can be considered an unidiomatic use of type classes and relies on intricacies regarding which instance of a type class Coq uses when multiple are in scope.

3 The Basic Nextgen Modality

In this section we introduce the *basic* nextgen modality. Here the word "basic" means that the modality is a low-level addition to the Iris *base logic*. It is a minimal extension to Iris that enables one to express non-frame-preserving updates. The basic nextgen modality then facilitates the definition of higher-level nextgen modalities for specific purposes. This approach follows the Iris tradition of keeping the base logic minimal and simple, while defining more complex notions *inside* the logic. Technically, the nextgen modality can be seen as a *family* of modalities in that it is parameterized by a so-called generational transformation (defined below), and we show that the nextgen modality encompasses two existing Iris modalities, namely the persistently and the plain modalities.

The basic nextgen modality is written $\dot{\Rightarrow}^t P$ where $P : \mathsf{iProp}$ is an assertion and $t : M \to M$ is a function on the global RA. The dot above the symbol indicates that this is the *basic* nextgen modality. We call the function t a *generational transformation* or sometimes just a *transformation*. The assertion $\dot{\Rightarrow}^t P$ should be read "given a generational transformation described by t then P holds in the next generation".

In order for the modality to be sensible, the generational transformation needs to satisfy a few basic properties. Whenever we write $\dot{\Rightarrow}^t$, we assume that t ranges over functions with these properties. The requirements for t are given in the following definition.

Definition 3.1 (Generational transformation)**.** Given a resource algebra A, a generational transformation is a function $t : A \to A$ that satisfies the following conditions.

1. It is monotone with respect to the inclusion order of the resource algebra.
$$\forall x, y.\, x \preccurlyeq y \Rightarrow t(x) \preccurlyeq t(y)$$
2. It preserves validity of elements.
$$\forall x.\, \mathcal{V}(x) \Rightarrow \mathcal{V}(t(x))$$
3. It is non-expansive with respect to the ordered family of equivalences (OFE) for A.
$$\forall n, x, y.\, x \overset{n}{=} y \Rightarrow t(x) \overset{n}{=} t(y)$$

The first two conditions should seem reasonable. The first condition is necessary as the model of Iris uses monotone predicates over RAs and as we see in Section 3.4 this condition ensures that the meaning of $\dot{\Rightarrow}^t P$ is monotone in the model. The second condition is necessary as Iris maintains the property that the owned resources are always valid, hence the generational transformation needs to maintain validity. The third condition pertains to an aspect of RAs that we have not described, namely that they contain an OFE or a "step indexed equality" [8]. We include the condition here for completeness and for readers who are familiar with OFEs.

3.1 Rules

Figure 2 shows a selection of rules for the basic nextgen modality. The first rule, BNG-OWN is the nextgen modality's raison d'être. It states that the transformation t is applied to owned ghost state. As we are at the level of the base logic this rule concerns the Own assertion and not ghost locations.

The following rules in the figure state that the nextgen modality is monotone (BNG-MONO), commutes with conjunction (BNG-CONJ), disjunction (BNG-DISJ), the later modality (BNG-LATER), exist (BNG-EXISTS), and forall (BNG-FORALL). Together these rules ensures that the basic nextgen modality is well-behaved and convenient to work with. However, not all rules of this form that we would want hold without making further requirements on the transformation. If we look at the next rule BNG-SEP, we see that it states an additional demand on t. In Definition 3.1 we defined the *essential* properties that a transformation must possess in order to work at all. In practice, useful transformations satisfy more properties than those, and in those cases more rules are sound. For the rule BNG-SEP the requirement is that the transformation commutes with the monoid operation of the RA. If this is the case, then two assertions under a nextgen modality can be combined under one nextgen modality. The same is the case for the two rules BNG-PERS and BNG-IDEMP. If the transformation commutes with the core of the RA (when it is defined) then the modality commutes with the persistently modality. And, if the transformation is idempotent then so is the modality (BNG-TRANS holds for all transformations though). In all of these cases, the rule for the nextgen modality quite directly reflects the property of the transformation.

Note that BNG-SEP only holds in one direction. We emphasize that the direction that *does* hold is the important direction. For instance, this direction is used by the Iris proof mode when introducing the modality.

Since the nextgen modality modifies resources, it has no effect on propositions that do not rely on resources. In Iris, such propositions are described with the *plainly modality* $\blacksquare P$, which means that P holds without using any resources. The rule BNG-PLAINLY states that the nextgen modality has no effect in the presence of the plainly modality. If we rephrase this lemma a bit we get what we call the soundness rule for

BNG-OWN
$\mathsf{Own}\,(a) \vdash \Rightarrow^t \mathsf{Own}\,(t(a))$

BNG-MONO
$$\frac{P \vdash Q}{\Rightarrow^t P \vdash \Rightarrow^t Q}$$

BNG-CONJ
$\Rightarrow^t P \land \Rightarrow^t Q \dashv\vdash \Rightarrow^t (P \land Q)$

BNG-DISJ
$\Rightarrow^t P \lor \Rightarrow^t Q \dashv\vdash \Rightarrow^t (P \lor Q)$

BNG-LATER
$\triangleright \Rightarrow^t P \dashv\vdash \Rightarrow^t \triangleright P$

BNG-EXISTS
$\Rightarrow^t \exists x.\, P \dashv\vdash \exists x.\, \Rightarrow^t P$

BNG-FORALL
$\Rightarrow^t \forall x.\, P \dashv\vdash \forall x.\, \Rightarrow^t P$

BNG-SEP
$$\frac{\forall x, y.\, t(x \cdot y) = t(x) \cdot t(y)}{\Rightarrow^t P * \Rightarrow^t Q \vdash \Rightarrow^t (P * Q)}$$

BNG-PERS
$$\frac{\forall x.\, t(|x|) = |t(x)|}{\Box \Rightarrow^t P \dashv\vdash \Rightarrow^t \Box P}$$

BNG-TRANS
$\Rightarrow^{t_1} \Rightarrow^{t_2} P \dashv\vdash \Rightarrow^{t_2 \circ t_1} P$

BNG-IDEMP
$$\frac{\forall x.\, t(t(x)) = t(x)}{\Rightarrow^t \Rightarrow^t P \vdash \Rightarrow^t P}$$

BNG-PLAINLY
$\Rightarrow^t \blacksquare P \dashv\vdash \blacksquare P$

Figure 2. Rules for the basic nextgen modality.

the nextgen modality:

BNG-SOUND
$$\frac{\vdash \Rightarrow^t P \qquad \mathrm{plain}(P)}{\vdash P}$$

This states that if an assertion P is plain (meaning that $P \vdash \blacksquare P$) and can be derived under the nextgen modality, then the basic nextgen modality can be eliminated. A consequence of this rule is that results shown under the nextgen modality also has meaning outside of the nextgen modality, which is crucial when one wishes to prove an overall soundness or adequacy result for a program logic that makes use of the basic nextgen modality.

Just as important as the rules that do hold, is the one that does not. The following frame rule is *not* sound

$$Q * \Rightarrow^t P \not\vdash \Rightarrow^t (Q * P).$$

If Q holds in the current generation and P holds in the next generation then it is not necessarily sensible to move Q unchanged into the next generation. The equivalent rule for the update modality holds and is crucial for that modality's purpose. For the nextgen modality the opposite is the case: invalidating the frame rule is clearly *necessary* to arrive at a modality that can express non-frame-preserving changes to ghost state. Another rule that, quite naturally, is not sound is commutativity between the basic nextgen modality and the update modality:

$$\Rightarrow^t \models\!\!\Rightarrow P \not\dashv\vdash \models\!\!\Rightarrow \Rightarrow^t P.$$

As we explain in Section 4.2.1 this has an impact on the way adequacy is proven for program logics that use the nextgen modality.

3.2 Comparison to the Post-Crash Modality

In a nutshell, the difference between the post-crash modality and the nextgen modality is that where the post-crash modality works *around* the limitation that Iris does not allow for non-frame-preserving updates, the nextgen modality addresses the problem head-on by lifting the limitation through an extension of the Iris base logic. Thence the nextgen modality addresses the limitations we identified for the post-crash modality. In particular, since the nextgen modality modifies existing resources, it does *not* rely on changing ghost names and hence it does not incur the problems associated with that. We are able to prove all the expected rules, including BNG-PERS that does not hold for the post-crash modality.

3.3 Special Cases of the Basic Nextgen Modality

We now show that the persistently and the plainly modalities are special cases of the basic nextgen modality and thus our "extension" of the Iris base logic is perhaps better referred to as a "generalization and simplification" of the Iris base logic.

The persistently modality can be defined by taking the transformation to be the persistent core of the global RA:

$$\Rightarrow^{|-|} P \dashv\vdash \Box P \tag{2}$$

and the plainly modality can be defined by taking the transformation to be the constant function that returns the unit element of the global RA.

$$\Rightarrow^{\lambda a.\varepsilon} P \dashv\vdash \blacksquare P \tag{3}$$

The equivalences (2) and (3) are both easy to prove using the semantics of the basic nextgen modality, which we present in the following section.

3.4 Model

We now explain the semantics of the basic nextgen modality in the model of Iris.

To simplify the presentation and to focus on the interesting parts, we pretend that the semantic domain of Iris propositions is simply monotone predicates over resources:

$$[\![\mathsf{iProp}]\!] \triangleq M \xrightarrow{\mathsf{mon}} \mathit{Prop}.$$

The gap between this simplified definition and the full model of Iris is largely orthogonal to the semantics of the nextgen modality. We ignore the recursive domain equation arising from higher-order ghost state and step indices for the later modality. The benefit is that this simplifies the presentation and makes it easier to understand for readers who are not

familiar with the particularities of the model of Iris, but who might be familiar with the more widely used predicates-over-resources model of separation logic. Our mechanization of the nextgen modality in Coq, of course, uses the "full" model of Iris, and we refer readers interested in all the details to the accompanying Coq formalization.

The model of the nextgen modality is exactly what one would expect from its behavior in the logic:

$$[\![\dot{\leadsto}^t P]\!] \triangleq \lambda x.\, [\![P]\!](t(x))$$

In order for this definition to be well-defined it must be monotone.

Lemma 3.2. *If $x \preccurlyeq y$ then $[\![P]\!](t(x))$ implies $[\![P]\!](t(y))$.*

Proof. Since $[\![P]\!]$ is monotone it suffices to show that $t(x) \preccurlyeq t(y)$. This follows from condition 1 of Definition 3.1. □

With this model all the rules that we have seen are sound.

3.5 Generational Resource Algebras

When using the nextgen modality with particular resources, one usually picks the type of resources and the transformations for it in unison. We use the term *generational RA* to mean an RA together with transformation function over it or a set of such functions. For many of the existing RAs in Iris there are obvious transformation functions that one could use with them. As an example, for the well known authoritative RA $Auth(A)$ and a transformation $t : A \to A$, there is a transformation t_A that applies t to both the authoritative element and fragments such that

$$t_A(\bullet a) \triangleq \bullet(t_A(a)) \qquad t_A(\circ b) \triangleq \circ(t_A(b)).$$

This transformation is part of the generational RA that we use in Section 4.

Just like Iris contains a library of RA constructions that one can combine for concrete proofs, one can imagine a similar library of constructions for generational RAs. Our Coq mechanization contains a few such building blocks.

3.6 A Transformation for Ghost Locations

So far, we have seen the basic nextgen modality that applies a transformation to owned elements of the global RA. As described in Section 2.1, Iris is usually instantiated with a global RA of a particular shape. To arrive at higher-level nextgen modalities, the first step is to use transformation functions that preserve this shape:

$$TM \triangleq \prod_{i \in I} GName \xrightarrow{\text{fin}} (M_i \to M_i)$$

This definition is equal to the global RA in Equation (1) except that the type of the "leafs" is changed from M_i to $M_i \to M_i$. From a map of transformations $tm \in TM$, we can construct a transformation on the global RA in the natural way by applying the appropriate transformations.

For any $tm \in TM$, we then obtain the following rules for the basic nextgen modality and ownership of an $a : M_i$ at a ghost location γ.

$$\frac{\gamma \in \mathrm{dom}(tm(i))}{\boxed{a}^\gamma \vdash \dot{\leadsto}^{t_{tm}} \boxed{tm(i,\gamma)(a)}^\gamma} \qquad \frac{\gamma \notin \mathrm{dom}(tm(i))}{\boxed{a}^\gamma \vdash \dot{\leadsto}^{t_{tm}} \boxed{a}^\gamma}$$

This construction provides the foundation for building higher-level nextgen modalities.

A simpler variant of this construction is one where the map has the form $\prod_{i \in I}(M_i \to M_i)$. That is, where the transformation is given only per type of RA and not per type of RA *and* ghost name. Which variant to use depends on the circumstances; in the next section we see an example of using the simpler one.

3.7 Mechanization in Coq

As mentioned earlier we have mechanized the nextgen modality in Coq. The development contains the definition of the basic nextgen modality and its rules. Through type class instances the nextgen modality is integrated into the Iris Proof Mode such that it works as seamlessly as existing modalities.

Despite the nextgen modality being an extension to the base logic, we do not need to fork or modify the existing Iris Coq development. Due to the way Iris is mechanized one can define new constructs in terms of the model as long as the semantic domain is unchanged.

The mechanization also contains a number of generational transformations for common RAs and the transformation for ghost locations from the previous section.

4 Case Study of the Nextgen Modality

The basic nextgen modality lays the foundation for expressing non-frame-preserving updates. However, thus far, we have left out *how* a concrete instance of a nextgen modality is defined. In this section, we present a case study of the nextgen modality: a program logic for a language, called StackLang, whose operational semantics exhibits non-local changes to its physical state. We first present StackLang, and then we give a concrete definition of a nextgen modality, by defining the right generational transform function t. Finally, we use the nextgen modality to define a program logic for StackLang.

StackLang is a language with a high-level representation of a call stack, where stack frames (henceforth referred to as stack regions) are pushed upon function calls, and popped upon function returns. Stack region locations are referenced via special stack references, which are indexed by a region offset. These stack references are *temporary*, and have a lifetime inherently linked to their stack region. Expressing stack allocation at a higher level thus allows programmers to allocate temporary data without having to manage it, which has advantages both in languages with explicit memory management, and in languages where memory is managed by a

garbage collector (for example, Lorenzen et al. use a locality mode to express stack allocation, in order to reduce garbage collection triggers).

Furthermore, StackLang supports a version of call/cc, where continuations enclose a specific stack region, and whose invocation pop all stack regions above it.

A sound program logic for StackLang must thus account for the deallocation of stack regions. A naïve approach may simply require the program logic rule for function returns to depend on the relevant ghost state in the precondition. Such a rule would define a precondition containing *all* ghost state fragments that would get deallocated by the return expression. Unfortunately, this approach counteracts the benefits of local reasoning typically granted by separation logic, and would make it especially difficult to describe modular specifications of continuations. Instead, our goal will be to construct a program logic with a rule for function returns that does not directly depend on fragments from the stack region. Thus, while other approaches exist, we seek a program logic that does not require a lot of bookkeeping, or any instrumentation of the language itself.

4.1 Syntax and Semantics of StackLang

Figure 3 defines the syntax of StackLang values, expressions, and evaluation contexts. Values include continuations, *i.e.,* suspended evaluation contexts labelled with an index i, specifying which stack region the continuation belongs to.

Similarly, we label function closures $\lambda^{\mu}\, k, x.e$ and locations ℓ^{μ} with a locality tag μ, which specifies their *lifetime*. A heap tag means the function or location has a permanent lifetime, while a $\mathsf{stack}(i)$ tag means the function or location has the same lifetime as stack region i. The index i in locality $\mathsf{stack}(i)$ is *relative* to the top of the stack. For instance, $\ell^{\mathsf{stack}(0)}$ refers to a stack-allocated location in the topmost stack region. Likewise, a continuation with index i refers to the i^{th} stack region from the top, and invoking it will thus deallocate the i most recent regions.

New locations are allocated using $\mathsf{halloc}(e)$, which allocates locations with a heap tag, and $\mathsf{salloc}(e)$, which allocates locations with a $\mathsf{stack}(0)$ tag. The remaining values and expressions are defined as in a typical lambda calculus with references, where x is a variable, n stands for any natural number, and $\oplus$ is shorthand for binary operators. Finally, evaluation contexts define a left-to-right and call-by-value evaluation strategy.

The small-step operational semantics of StackLang is defined using two step relations. Each relation is defined over configurations (h, s, e), where h is the heap, s is an ordered list of stack regions, and e is an expression. The head of the stack describes the state of the topmost stack region, and function calls appends a new empty region to the head of the list, while function returns remove a specified number of regions from the list. We refer to the heap and stack pair (h, s) as the store.

The first step relation $\rightarrow_K$ defines steps taken under some evaluation context K. Figure 4 shows two example steps; stack allocation, and function calls. Stack allocation allocates a fresh location ℓ in the topmost stack region, and returns $\ell^{\mathsf{stack}(0)}$. Function calls push an empty region to the stack, and reduce to a return expression surrounding the body of the function. Note that since the locality of locations, closures and continuations is relative, parameters and return values are shifted to accurately reflect their new relative position.[2]

The second step relation $\rightarrow$ is built on top of $\rightarrow_K$, and defines the operational semantics of StackLang:

$$\textsc{ctx-bind}\quad \frac{(h, s, e) \rightarrow_K (h', s', e')}{(h, s, K[e]) \rightarrow (h', s', K[e'])}$$

$$\textsc{ctx-ret}\quad \frac{i \leq \mathit{length}(s) \qquad \mathit{shift}(v, -i) = v'}{(h, s, \mathsf{return}(\mathsf{cont}^i(K))(v)) \rightarrow (h, \mathit{pop}^i(s), K[v'])}$$

The step for $\mathsf{return}(\mathsf{cont}^i(K))(v)$ shifts v by $-i$, pops the top i regions from s, and is considered stuck whenever the continuation points to a stack region that does not exist.

4.2 A Program Logic for StackLang

The first step towards a program logic for StackLang is to interpret the store in terms of Iris predicates. To that end, we define three resource algebras, which allow us to model the following three separation logic predicates with associated intuitive meanings:

$\ell \mapsto v$	heap location ℓ points to value v
$\boxed{k}\, \ell \mapsto v$	stack location ℓ of region index k points to v
$\exists\, m$	the stack is currently made up of m regions

The most pertinent resource algebra is that of the stack: a ghost map, gmapView $(\mathbb{N} \times \mathsf{Location})$ Value, mapping stack region indices and location pairs to values.

Next, we define a nextgen modality that deallocates the relevant stack resources, while leaving unrelated resources intact. We use the construction outlined in Section 3.6 to build a map of transformations, which together form a transformation on the global state. We only define a non-trivial transformation function for the stack resource algebra; for all other resource algebras we use the identity transformation.

For the stack resource algebra, we define a family of transformations, SCut^n, where n is a region index. Intuitively, SCut^n filters out all the elements with an index of at least n. Concretely, $\mathsf{SCut}^n(m) = m'$ where $\mathit{dom}(m') \subseteq \mathit{dom}(m)$ and $\forall (k, \ell) \in \mathit{dom}(m), (k < n \land m'(k, \ell) = m(k, \ell)) \lor (k \geq n \land (k, \ell) \notin \mathit{dom}(m'))$.

[2] Shifting values is handled by $\mathit{shift}(v, i)$, a partial function that shifts any stack location or continuation by the integer i. If the shift would put an index below zero, it fails, *i.e.,* it is undefined.

$$
\begin{array}{llll}
\text{Index} & i & \triangleq & \mathbb{N} \\
\text{LocalityTag} & \mu & ::= & \mathsf{heap} \mid \mathsf{stack}(i) \\
\text{Value} & v & ::= & \mathsf{true} \mid \mathsf{false} \mid n \mid () \mid \lambda^{\mu}\, k, x.e \mid \ell^{\mu} \mid \mathsf{cont}^{i}(K) \mid (v, v) \\
\text{Expression} & e & ::= & x \mid \mathsf{true} \mid \mathsf{false} \mid n \mid () \mid \lambda^{\mu}\, k, x.e \mid \ell^{\mu} \mid \mathsf{cont}^{i}(K) \mid e \oplus e \mid (e, e) \mid \pi_{\{1,2\}} e \mid e(e) \mid \mathsf{return}(e)(e) \mid \\
 & & & \mathsf{let}\ x := e\ \mathsf{in}\ e \mid \mathsf{if}\ e\ \mathsf{then}\ e\ \mathsf{else}\ e \mid \mathsf{salloc}(e) \mid \mathsf{halloc}(e) \mid !e \mid e \leftarrow e \\
\text{Evaluation Context} & K & ::= & \cdot \mid K \oplus e \mid v \oplus K \mid (K, e) \mid (v, K) \mid \pi_{\{1,2\}} K \mid \\
 & & & K(e) \mid v(K) \mid \mathsf{return}(K)(e) \mid \mathsf{return}(v)(K) \mid \mathsf{let}\ x := K\ \mathsf{in}\ e \mid \mathsf{let}\ x := v\ \mathsf{in}\ K \mid \mathsf{if}\ K\ \mathsf{then}\ e\ \mathsf{else}\ e \mid \\
 & & & \mathsf{salloc}(K) \mid \mathsf{halloc}(K) \mid !K \mid K \leftarrow e \mid v \leftarrow K
\end{array}
$$

Figure 3. StackLang syntax

$$
\frac{s[0] = f \qquad \ell \notin dom(f) \qquad s' = s[0 := f \uplus \{\ell \mapsto v\}]}{(h, s, \mathsf{salloc}(v)) \to_K (h, s', \ell^{\mathsf{stack}(0)})}
\qquad
\frac{shift(v, 1) = v' \qquad k = \mathsf{cont}^{1}(K)}{(h, s, (\lambda^{\mathsf{heap}}\, f, x.e)(v)) \to_K (h, \emptyset +\!\!+ s, \mathsf{return}(k)(e[k/f][v'/x]))}
$$

Figure 4. StackLang inner step relation, excerpt

We denote the combined transformation function on the global resource algebra by ICut^n. In detail:

$$\mathsf{ICut}^n = T^{tm\{i := \mathsf{SCut}^n\}}$$

where $i \in I$ is the globally scoped ID of the stack resource algebra. By picking ICut^n as the generational transformation function, we can then define a nextgen modality for stack region deallocation:

$$\dot{\leadsto}^n \quad \triangleq \quad \leadsto^{\mathsf{ICut}^n}$$

and using the definition of ICut^n, we prove the following rules:

CUT-HEAP-INTRO
$\ell \mapsto v \vdash \dot{\leadsto}^n\, \ell \mapsto v$

CUT-STACK-INTRO
$$\frac{k < n}{\boxed{k}\, \ell \mapsto v \vdash \dot{\leadsto}^n\, \boxed{k}\, \ell \mapsto v}$$

CUT-STACK-INTRO-EMP
$$\frac{k \geq n}{\boxed{k}\, \ell \mapsto v \vdash \dot{\leadsto}^n\, \top}$$

CUT-SIZE-INTRO
$\boxminus\, m \vdash \dot{\leadsto}^n\, \boxminus\, m$

Note that the introduction rule for the stack points-to predicate requires that the resources points to a stack region below the cut n. If the region is at or above n, the fragment is deallocated, as expressed by the trivial rule CUT-STACK-INTRO-EMP.

4.2.1 Weakest precondition.

We define a program logic for StackLang by using a variant of Iris weakest preconditions, denoted $\mathsf{wp}\ e\ \{\Phi\}_n$, where n is a region index. The key idea behind n is that it dictates a lower bound to the nextgen modalities triggered by the weakest precondition.

The definition is similar to the one used in Iris for a single-threaded language: If e is a value, then the postcondition Φ holds for that value (under a fancy update modality). If e is not a value, then given any state σ satisfying the state interpretation (the authoritative view of the resource algebras used for the physical state), the expression e must be reducible, and for any configuration it steps to, we have the semantic interpretation of the new state, and a weakest precondition of the new expression. In the special case where the expression is a function call return, the state interpretation and weakest precondition of the reduced expression is guarded by a nextgen modality, reflecting that a stack deallocation has occurred. Additionally, the cutoff of that nextgen modality is guaranteed to be greater than or equal to the lower bound n. In section 4.2.2 we will see why it is useful to maintain such a lower bound.[3]

Since we are using a new definition of Iris weakest preconditions, it is important to prove that it is sound. To this end, we prove the following adequacy theorem.[4]

Theorem 4.1 (Adequacy of the nextgen weakest precondition)**.** *Let Φ be a first-order pure predicate. Assume:*

$$\vdash \mathsf{wp}\ e\ \{\Phi\}_n \text{ and } (\sigma, e) \to (\sigma_2, e_2),$$

then the following two facts hold:

1. *either (σ_2, e_2) is reducible, or e_2 is a value*
2. *if e_2 is a value, then $\Phi(e_2)$ holds*

The proof, which can be found in the accompanying Coq formalization, is more intricate than the standard Iris one since the nextgen modality does not commute with the fancy update modality. An interesting line of future work is to investigate alternative strategies for proving adequacy, such as the one employed in Transfinite Iris [15], and for Later Credits [16].

4.2.2 The frame rule and the independence modality.

Since the nextgen modality is, by design, not frame preserving, one major consequence of including it in the weakest

[3] Here we present a version of the weakest precondition definition that is tailored specifically to StackLang. In the Coq mechanization, we define a version that parametrizes over an arbitrary programming language.
[4] Here again tailored to StackLang, but proved for a general single-threaded language in the Coq mechanization, without support for later credits.

$$\mathsf{wp}\ e\ \{\Phi\}_n \quad \triangleq \quad \begin{cases} \Rrightarrow_\top \Phi(e) & \text{if } e \text{ is a value} \\ \forall \sigma, \mathsf{stateInterp}(\sigma) \twoheadrightarrow {}_\top\Rrightarrow_\emptyset\ e \text{ is reducible} & \text{if } e = K[\mathsf{return}(\mathsf{cont}^i(K'))(v)] \\ \quad \wedge \triangleright \forall e_2, \sigma_2, (\sigma, e) \rightarrow (\sigma_2, e_2) \twoheadrightarrow {}_\emptyset\Rrightarrow_\top n \leq \mathit{StackLength}(\sigma) - i * \\ \qquad (\mathrel{\dot{\leadsto}}^{\mathit{StackLength}(\sigma)-i}\ \mathsf{stateInterp}(\sigma_2)) * (\mathrel{\dot{\leadsto}}^{\mathit{StackLength}(\sigma)-i}\ \mathsf{wp}\ e_2\ \{\Phi\}_n) & \\ \forall \sigma, \mathsf{stateInterp}(\sigma) \twoheadrightarrow {}_\top\Rrightarrow_\emptyset\ e \text{ is reducible} & \text{otherwise} \\ \quad \wedge \triangleright \forall e_2, \sigma_2, (\sigma, e) \rightarrow (\sigma_2, e_2) \twoheadrightarrow {}_\emptyset\Rrightarrow_\top \mathsf{stateInterp}(\sigma_2) * \mathsf{wp}\ e_2\ \{\Phi\}_n & \end{cases}$$

Figure 5. Weakest Precondition

precondition is that we lose the frame rule. This is necessary to prevent the following clearly unsound implication:

$$\boxed{1}\,\ell \mapsto v\ *\ \mathsf{wp}\ \mathsf{return}(\mathsf{cont}^1(\cdot))(42)\ \{\top\}_1 \vdash \mathsf{wp}\ \mathsf{return}(\mathsf{cont}^1(\cdot))(42)\ \{\boxed{1}\,\ell \mapsto v\}_1$$

However, losing the frame rule altogether seems much too restrictive. There are, after all, many resources which ought not to be affected by any occurrence of the nextgen modality. For example, it should be possible to use the frame rule for heap resources, or stack resources that reside below the chosen cutoff.

To that end, we define a new *independence modality*, denoted $\boxplus^n$, and use it to express that a proposition is safe to frame away. Intuitively, the independence modality expresses that a proposition is independent of any nextgen modality at or above some lower bound n. Formally, it is defined as follows:

$$\boxplus^n P \quad \triangleq \quad P \wedge \forall n' \geq n, \mathrel{\dot{\leadsto}}^{n'} P$$

This modality gives rise to the following rules:

$$\textsc{ind-intro}\ \frac{\forall n', n \leq n' \Rightarrow P \vdash \mathrel{\dot{\leadsto}}^{n'} P}{P \vdash \boxplus^n P} \qquad \textsc{cut-ind-intro}\ \frac{n \leq n'}{\boxplus^n P \vdash \mathrel{\dot{\leadsto}}^{n'} \boxplus^n P}$$

$$\textsc{ind-elim}\ \boxplus^n P \vdash P \qquad \textsc{ind-weaken}\ \frac{n \leq n'}{\boxplus^n P \vdash \boxplus^{n'} P} \qquad \textsc{ind-heap-intro}\ \ell \mapsto v \vdash \boxplus^n \ell \mapsto v$$

$$\textsc{ind-stack-intro}\ \boxed{k}\,\ell \mapsto v \vdash \boxplus^{k+1} \boxed{k}\,\ell \mapsto v \qquad \textsc{ind-size-intro}\ \boxminus m \vdash \boxplus^n \boxminus m$$

Using the independence modality, we prove the following frame rule:

$$\textsc{Wp-Frame}\ \frac{\boxplus^n R \qquad \mathsf{wp}\ e\ \{\Phi\}_n}{\mathsf{wp}\ e\ \{\lambda v, \Phi(v) * R\}_n}$$

As with the definition of weakest precondition, we here present a version of the independence modality tailored to StackLang. In the Coq mechanization we implement a generalized definition of the independence modality, together with proof support for the modality and the new frame rule.

4.2.3 Program logic rules.

Using the new definition of weakest precondition we prove soundness of proof rules for each reduction step. In this subsection, we will focus on the rules for stack allocation and function returns. We refer to the Coq mechanization for the full program logic. Since invoking a continuation discards the current surrounding evaluation context, the typical bind-rule for Iris weakest preconditions does not hold for StackLang expressions.

For reasoning about programs that do not use non-local flow — for which the bind rule holds — we follow [17] and define a derived notion of weakest preconditions called the context-local weakest precondition:

$$\mathsf{clwp}\ e\ \{\Phi\}_n \quad \triangleq \quad \forall K\ \Psi, (\boxplus^n \forall v, \Phi(v) \twoheadrightarrow \mathsf{wp}\ K[v]\ \{\Psi\}_n) \twoheadrightarrow \mathsf{wp}\ K[e]\ \{\Psi\}_n$$

Note the use of the independence modality to express that the continuation cannot depend on stack locations above the cutoff n.

We then prove the following rule for stack allocation:

$$\textsc{ClSalloc}\ \frac{\triangleright(\forall \ell, \boxminus m * \boxed{m-1}\,\ell \mapsto v \twoheadrightarrow \mathsf{clwp}\ \ell^{\mathsf{stack}(0)}\ \{\Phi\}_n) \qquad \boxminus m \qquad 0 < m}{\mathsf{clwp}\ \mathsf{salloc}(v)\ \{\Phi\}_n}$$

The rule for stack allocation allocates a new stack points-to predicate, whose region index is determined using the stack size resource.

Since function returns use non-local flow, we define its program logic rule using the base weakest precondition.

$$\textsc{Return}\ \frac{\triangleright(\boxminus m - i \twoheadrightarrow \mathrel{\dot{\leadsto}}^{(m-i)} \mathsf{wp}\ K'[\mathit{shift}(v, -i)]\ \{\Phi\}_n) \qquad \boxminus m \qquad i \leq m \qquad n \leq m - i}{\mathsf{wp}\ K[\mathsf{return}(\mathsf{cont}^i(K'))(v)]\ \{\Phi\}_n}$$

The rule for functions returns must not only decrease the stack size resource, it must somehow handle the *deallocation* of a number of stack regions, which may now be non-empty. In other words, the rule for return is only sound if it handles the deallocation of *all* stack points-to predicates associated to popped regions. Luckily, this is exactly expressed by the nextgen modality for stack region deallocation! As a result, it suffices to guard the continuing weakest precondition with $\mathrel{\dot{\leadsto}}^{(m-i)}$, which states that the next weakest precondition

cannot depend on any points-to predicates for stack locations above $m - i$. Finally, since the weakest precondition imposes a lower bound n to occurrences of a nextgen modality, the rule requires a side condition to guarantee that $n \leq m - i$.

4.2.4 Custom ghost state and invariants. Thus far, we have presented the StackLang nextgen modality with respect to the StackLang ghost state. However, we have yet to show how to take advantage of the full expressive power of the Iris logic. Notably, we have not yet discussed how to use custom ghost state or invariants.

Since we have defined the $\dot{\mapsto}^n$ modality to apply the identity transformation on any non-stack resource, any custom ghost state can easily introduce the modality. In contrast, more interesting questions arise when we consider the interaction between the nextgen modality and Iris invariants. Since an Iris invariant is guaranteed to hold at every step of a program's execution, how can it enclose stack allocated resources that might disappear at function returns? Clearly, it would not be sound for such invariants to outlive the stack values they correspond to. One possible sound solution would be to only allow invariants that do not enclose any stack points-to predicates. However, such a limitation would disallow interesting use-cases of Iris invariants, such as defining a *temporary* invariant that holds until a region has ended.

The ideal solution would be invariants that can contain stack points-to predicates and whose lifetime matches that of the enclosed resources. This is precisely what we achieve in the following variant of Iris invariants, defined using the independence modality presented in Section 4.2.2.

Our variant of Iris invariants, denoted $\boxed{P}^{\mathcal{N},n}$, is parameterized by a region index n. One can think of the index as the lifetime of the invariant. Invariants are allocated as follows:

$$\frac{\text{INV-ALLOC}}{P \vdash \Rightarrow^n P \qquad \triangleright P}{\Rrightarrow_{\mathcal{E}} \boxed{P}^{\mathcal{N},n}}$$

When allocating an invariant for n, one must prove that the body of the invariant is unaffected by any nextgen modality that discards stack regions at or above n. This condition is precisely expressed by the independence modality.

The interaction between invariants and the nextgen and independence modalities now depends on the lifetime of the invariant:

$$\frac{\text{CUT-INV-INTRO}}{n \leq n'}{\boxed{P}^{\mathcal{N},n} \vdash \dot{\mapsto}^{n'} \boxed{P}^{\mathcal{N},n}} \qquad \text{IND-INV-INTRO}\ \ \boxed{P}^{\mathcal{N},n} \vdash \Rightarrow^n \boxed{P}^{\mathcal{N},n}$$

With this new invariant construction, it is possible to allocate invariants that enclose stack points-to predicates, and prove specifications of programs that may depend on them. As such, not only can we define invariants that are not impacted by $\dot{\mapsto}^n$, we can also define invariants that may themselves be deallocated by a particular instance of $\dot{\mapsto}^n$. The invariants exist for as long as it would be sound for them to do so, and are removed by the nextgen modality accordingly. This new definition of invariants displays the flexibility of the nextgen modality, which allows us to define arbitrary transformations over ghost state, including the ghost state of invariants themselves.

We leave out the technical details of the definition of the new invariants, and refer to the Coq mechanization for those.[5] The key idea is to apply a transformation to invariants, which mimics the transformation function for stack points-to predicates, by indexing invariants by stack regions. The requirements over P when allocating invariants is carried over to the definition of the so-called world satisfaction relation, which is the internal Iris definition which tracks and stores all invariants, and then used to prove the soundness of the nextgen introduction rules for invariants.

5 Related and Future Work

As mentioned, the post-crash modality from Perennial is the work most closely related to the nextgen modality. We have already compared the two earlier in Section 3. To the best of our knowledge there is no other work that gives a general mechanism for performing non-frame-preserving updates in separation logic.

We think that there is much exciting future work to be done, and hope that we have just scratched the surface of the usefulness of the nextgen modality. Exploring the motivating examples that we sketched in the introduction is one possible avenue for future work. We are currently exploring the application of the nextgen modality to a concurrent setting with crashes and durable storage, and our current results seem very promising. One interesting challenge in this setting is that under a weak persistency model, crashes are non-deterministic and there is thus not a fixed transformation that can be applied to ghost state at a crash.

We also think our nextgen modality can be used as a foundation to implement temporary read-only points-to predicates in Iris in the style of [6]. Our initial investigation into this seems to indicate that defining the resources for this and the nextgen modality itself is quite straightforward. However, defining a weakest precondition that validates the expected proof rules seems quite tricky. In particular, the "framed sequencing rule" of *op. cit.* is non-trivial to prove for a weakest precondition that contains a nextgen modality. We think solving this hurdle is very exciting future work, as read-only point-to predicates bring many benefits that Iris users are currently missing.

[5] The technical details behind the definition of nextgen invariants are slightly more involved and include additional ghost state to remember the lifetime n, and a transformation over this ghost state that alters it in lockstep with SCut^n. We have also generalized it to work with any arbitrary indexing type and order.

Turning to work related to our program logic for StackLang, we first remark that the program logic rules for StackLang make explicit use of evaluation contexts because returns may discard the current evaluation context. This style of proof rules is inspired by Timany and Birkedal's work on a program logic for programs with continuations [17].

We are not aware of previous separation logics that explicitly account for deallocation of stack frames. The most closely related work is the work of Timany et al. [18] for reasoning about encapsulation of local state in a sequential programming language with a state monad and a Haskell-style polymorphically-typed runST construct. Timany et al. define a logical relation of the type system with runST in Iris and use it to show that runST encapsulates computations with local state and that such computations use regions allocated in a stack-like manner. A key point of *op. cit.* is that the operational semantics of the language is a standard operational semantics with a global heap, capturing how the language would be implemented in reality, whereas the logical relation allows one to reason *as if* regions were stack-allocated physically. This is achieved by a clever use of ghost state, which tracks the virtual stack of regions and connects it to the physical memory. To account for virtual deallocation of regions (popping the virtual stack of regions), Timany et al. essentially mark regions as dead in the ghost state and a key step in their proof of soundness of the logical relations model of the type system is then to show that the type system guarantees that one does not try to access a region that is dead. Thus, Timany et al. manage to account for virtual deallocation using only frame-preserving updates, but it comes at the expense of having a global ghost resource that is threaded around in the reasoning, rather than having more modular local points-to predicates, and it is not clear how this approach would scale to a concurrent language, since it does not seem possible to share the global ghost resource among several threads. In contrast, the nextgen modality allows for more modular local reasoning and scales to concurrent languages (cf. our current explorations of the nextgen modality to a concurrent setting with crashes and durable storage mentioned above).

Acknowledgments

This work was supported in part by a Villum Investigator grant (no. 25804), Center for Basic Research in Program Verification (CPV), from the VILLUM Foundation. This work was co-funded by the European Union (ERC, CHORDS, 101096090). Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Research Council. Neither the European Union nor the granting authority can be held responsible for them.

A Appendix

A.1 Operational Semantics

Figure 6 displays a larger selection of the step relation $\rightarrow_K$.

A.2 Program Logic Rules Excerpt

Figure 7 displays a selection of program logic rules for the base weakest precondition.

A.3 Example

The following program displays various features of StackLang.

$$(\emptyset, [\emptyset], (\lambda^{\text{heap}}\ k, x.!x{+}!(\text{salloc}(41)))(\text{halloc}(1))) \tag{4}$$

$$\rightarrow(\{\ell_1 \mapsto 1\}, [\emptyset], (\lambda^{\text{heap}}\ k, x.!x{+}!(\text{salloc}(41)))(\ell_1^{\text{heap}})) \tag{5}$$

$$\rightarrow(\{\ell_1 \mapsto 1\}, [\emptyset;\emptyset], \text{return}(\text{cont}^1(\cdot))(!\ell_1^{\text{heap}}{+}!\text{salloc}(41))) \tag{6}$$

$$\rightarrow(\{\ell_1 \mapsto 1\}, [\emptyset;\emptyset], \text{return}(\text{cont}^1(\cdot))(1{+}!\text{salloc}(41))) \tag{7}$$

$$\rightarrow(\{\ell_1 \mapsto 1\}, [\{\ell_2 \mapsto 41\};\emptyset], \text{return}(\text{cont}^1(\cdot))(1{+}!\ell_2^{\text{stack}(0)})) \tag{8}$$

$$\rightarrow(\{\ell_1 \mapsto 1\}, [\{\ell_2 \mapsto 41\};\emptyset], \text{return}(\text{cont}^1(\cdot))(1+41)) \tag{9}$$

$$\rightarrow(\{\ell_1 \mapsto 1\}, [\{\ell_2 \mapsto 41\};\emptyset], \text{return}(\text{cont}^1(\cdot))(42)) \tag{10}$$

$$\rightarrow(\{\ell_1 \mapsto 1\}, [\emptyset], 42) \tag{11}$$

Note how the function call appends an empty region to the head of the stack (line 6), which gets subsequently popped when the function returns (line 10). We will use this example in Section 4.2 when introducing the program logic for StackLang.

Let's use these rules to prove a specification of the previously presented example program. The program starts executing in a configuration with a stack of size 1. Our goal is to show the following specification:

$$\exists 1 \vdash \text{wp}\ (\lambda^{\text{heap}}\ k, x.!x{+}!(\text{salloc}(41)))(\text{halloc}(1))\ \{\lambda v, v = 42\}$$

we prove the specification by applying the program logic rules given in fig. 7. The first expression to execute is halloc(1). We thus begin by applying the rule for heap allocation Halloc, and the new goal becomes:

$$\exists 1 \vdash \triangleright\ (\ell_1 \mapsto 1 \mathrel{-\!\!*} \text{wp}\ (\lambda^{\text{heap}}\ k, x.!x{+}!(\text{salloc}(41)))(\ell^{\text{heap}})\ \{\lambda v, v = 42\})$$

We introduce the new points-to predicate into the context. Next, we apply the rules for call, stack allocation, load and binary operations to reach the following context and goal:

$$\exists 2 * \ell_1 \mapsto 1 * \boxed{1}\ell_2 \mapsto 41 \vdash \text{wp}\ \text{return}(\text{cont}^1(\cdot))(42)\ \{\lambda v, v = 42\}$$

At this point, we must apply the rule for return. Since the continuation has offset 1, this will decrease the stack by 1. After applying the rule for return, we are left with the following goal:

$$\exists 1 * \ell_1 \mapsto 1 * \boxed{1}\ell_2 \mapsto 41 \vdash \dot{\leadsto}^1\ \text{wp}\ 42\ \{\lambda v, v = 42\}$$

$$(h, s, \text{let } x := v \text{ in } e) \to_K (h, s, e[v/x]) \qquad (h, s, \pi_1(e_1, e_2)) \to_K (h, s, e_1) \qquad (h, s, \pi_2(e_1, e_2)) \to_K (h, s, e_2) \qquad \frac{v_1 \oplus v_2 = v}{(h, s, v_1 \oplus v_2) \to_K (h, s, v)}$$

$$(h, s, \text{if true then } e_1 \text{ else } e_2) \to_K (h, s, e_1) \qquad (h, s, \text{if false then } e_1 \text{ else } e_2) \to_K (h, s, e_2) \qquad \frac{\text{heap} \sqsubseteq v \qquad \ell \notin dom(h)}{(h, s, \text{halloc}(v)) \to_K (h \uplus \{\ell \mapsto v\}, s, \ell^{\text{heap}})}$$

$$\frac{s[0] = f \qquad \ell \notin dom(f) \qquad s' = s[0 := f \uplus \{\ell \mapsto v\}]}{(h, s, \text{salloc}(v)) \to_K (h, s', \ell^{\text{stack}(0)})} \qquad \frac{s[i](\ell) = v \qquad shift(v, i) = v'}{(h, s, !\ell^{\text{stack}(i)}) \to_K (h, s, v')} \qquad \frac{h(\ell) = v}{(h, s, !\ell^{\text{heap}}) \to_K (h, s, v)}$$

$$\frac{\text{heap} \sqsubseteq v \qquad \ell \in dom(h)}{(h, s, \ell^{\text{heap}} \leftarrow v) \to_K (h \uplus \ell \mapsto v, s, ())} \qquad \frac{\text{stack}(i) \sqsubseteq v \qquad s[i] = f \qquad \ell \in dom(f) \qquad shift(v, -i) = v' \qquad s' = s[i := f \uplus \{\ell \mapsto v'\}]}{(h, s, \ell^{\text{stack}(i)} \leftarrow v) \to_K (h, s', ())}$$

$$\frac{shift(v, 1) = v' \qquad k = \text{cont}^1(K)}{(h, s, (\lambda^{\text{heap}}\ f, x.e)(v)) \to_K (h, \emptyset +\!\!+ s, \text{return}(k)(e[k/f][v'/x]))}$$

$$\frac{shift(v, 1) = v' \qquad shift(e, i + 1) = e'}{(h, s, (\lambda^{\text{stack}(i)}\ k, x.e)(v)) \to_K (h, \emptyset +\!\!+ s, \text{return}(\text{cont}^1(K))(e'[\text{cont}^1(K)/k][v'/x]))}$$

Figure 6. StackLang inner step relation

Stk-Load
$$\frac{\triangleright(\boxed{n}\ \ell \mapsto v * \exists m \twoheadrightarrow \text{wp } K[shift(v, i)]\ \{\Phi\}_n) \qquad \boxed{n}\ \ell \mapsto v \qquad \exists m \qquad n = m - i - 1}{\text{wp } K[!\ell^{\text{stack}(i)}]\ \{\Phi\}_n}$$

Heap-Load
$$\frac{\triangleright(\ell \mapsto v \twoheadrightarrow \text{wp } K[v]\ \{\Phi\}_n) \qquad \ell \mapsto v}{\text{wp } K[!\ell^{\text{heap}}]\ \{\Phi\}_n}$$

Salloc
$$\frac{\triangleright(\exists m * \boxed{m-1}\ \ell \mapsto v \twoheadrightarrow \text{wp } K[\ell^{\text{stack}(0)}]\ \{\Phi\}_n) \qquad \exists m \qquad 0 < m}{\text{wp } K[\text{salloc}(v)]\ \{\Phi\}_n}$$

Halloc
$$\frac{\triangleright(\ell \mapsto v \twoheadrightarrow \text{wp } K[\ell^{\text{heap}}]\ \{\Phi\}_n)}{\text{wp } K[\text{halloc}(v)]\ \{\Phi\}_n}$$

Call-Global
$$\frac{\triangleright(\exists m + 1 \twoheadrightarrow \text{wp } K[\text{return}(\text{cont}^1(K))(e[\text{cont}^1(K)/k][shift(v, 1)/x])]\ \{\Phi\}_n) \qquad \exists m}{\text{wp } K[\lambda^{\text{heap}}\ k, x.e(v)]\ \{\Phi\}_n}$$

Return
$$\frac{\triangleright(\exists m - i \twoheadrightarrow \text{⇝}^{(m-i)}\ \text{wp } K'[shift(v, -i)]\ \{\Phi\}_n) \qquad \exists m \qquad i \le m \qquad n \le m - i}{\text{wp } K[\text{return}(\text{cont}^i(K'))(v)]\ \{\Phi\}_n}$$

Figure 7. Excerpt of the Program Logic Rules for StackLang

Crucially, the new goal is guarded by the ⇝^1 modality. The only way to introduce it is by applying monotonicity of the nextgen modality (BNG-MONO). Therefore, the next step is to introduce ⇝^1 in front of all the relevant predicates in the context, and discard those which don't have such an introduction rule. We can apply CUT-HEAP-INTRO and CUT-SIZE-INTRO and introduce the modality in front of the heap points-to predicate and the stack size resource. However, we can't apply the introduction rule for the stack points-to predicate (CUT-STACK-INTRO), since the region index of $\boxed{1}\ \ell_2 \mapsto 41$ is not stricly smaller than 1. In fact, the whole purpose of ⇝^1 is exactly to deallocate such points-to predicates. As such, we discard it, and apply BNG-SEP to get the following goal:

$$\text{⇝}^1(\exists 1 \ *\ \ell_1 \mapsto 1) \vdash \text{⇝}^1 \text{wp } 42\ \{\lambda v, v = 42\}$$

We can then finally conclude by applying monotonicity, and prove the post-condition.

While the above proof sketch manually applies the introduction rule for the nextgen modality, the rule for commuting over separation conjuction, and monotonicity; each of these steps are automated when using our mechanization in Coq. Due to the integration of the nextgen modality with the Iris Proof Mode, described in Section 3.7, introducing the nextgen modality is handled by a single tactic, leading to a seamless experience when using the program logic for StackLang in Coq.

References

[1] Tej Chajed. 2022. *Verifying a concurrent, crash-safe file system with sequential reasoning*. Ph. D. Dissertation. Machetutes Institute of Technology.

[2] Tej Chajed, Haogang Chen, Adam Chlipala, M. Frans Kaashoek, Nickolai Zeldovich, and Daniel Ziegler. 2017. Certifying a file system using crash hoare logic: correctness in the presence of crashes. *Commun. ACM* 60, 4 (2017), 75–84. https://doi.org/10.1145/3051092

[3] Tej Chajed, Joseph Tassarotti, and contributors. 2022. *Post-crash modality in Perennial's Coq Mechanization*. https://github.com/mit-pdos/perennial/blob/master/src/goose_lang/crash_modality.v Permanent link for the version available at the time of writing: https://github.com/mit-pdos/perennial/blob/ec3f44007d88b6ba28d1392807b444751588ed39/src/goose_lang/crash_modality.v.

[4] Tej Chajed, Joseph Tassarotti, M. Frans Kaashoek, and Nickolai Zeldovich. 2019. Verifying concurrent, crash-safe systems with Perennial. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*, Tim Brecht and Carey Williamson (Eds.). ACM, 243–258. https://doi.org/10.1145/3341301.3359632

[5] Tej Chajed, Joseph Tassarotti, Mark Theng, Ralf Jung, M. Frans Kaashoek, and Nickolai Zeldovich. 2021. GoJournal: a verified, concurrent, crash-safe journaling system. In *15th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2021, July 14-16, 2021*, Angela Demke Brown and Jay R. Lorch (Eds.). USENIX Association, 423–439. https://www.usenix.org/conference/osdi21/presentation/chajed

[6] Arthur Charguéraud and François Pottier. 2017. Temporary Read-Only Permissions for Separation Logic. 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 (Lecture Notes in Computer Science, Vol. 10201)*, Hongseok Yang (Ed.). Springer, 260–286. https://doi.org/10.1007/978-3-662-54434-1_10

[7] Armaël Guéneau, Johannes Hostert, Simon Spies, Michael Sammler, Lars Birkedal, and Derek Dreyer. 2023. Melocoton: A Program Logic for Verified Interoperability Between OCaml and C. *Proc. ACM Program. Lang.* OOPSLA (2023).

[8] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *J. Funct. Program.* 28 (2018), e20. https://doi.org/10.1017/S0956796818000151

[9] Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. 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 2015, Mumbai, India, January 15-17, 2015*, Sriram K. Rajamani and David Walker (Eds.). ACM, 637–650. https://doi.org/10.1145/2676726.2676980

[10] Robbert Krebbers, Ralf Jung, Ales Bizjak, Jacques-Henri Jourdan, Derek Dreyer, and Lars Birkedal. 2017. The Essence of Higher-Order Concurrent Separation Logic. 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 (Lecture Notes in Computer Science, Vol. 10201)*, Hongseok Yang (Ed.). Springer, 696–723. https://doi.org/10.1007/978-3-662-54434-1_26

[11] Robbert Krebbers, Amin Timany, and Lars Birkedal. 2017. 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, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 205–217. https://doi.org/10.1145/3009837.3009855

[12] Anton Lorenzen, Leo White, Stephen Dolan, Richard A. Eisenberg, and Sam Lindley. 2024. Oxidizing OCaml with Modal Memory Management. *Proc. ACM Program. Lang.* 8, ICFP, 485–514. https://doi.org/10.1145/3674642

[13] Jean-Marie Madiot and François Pottier. 2022. A separation logic for heap space under garbage collection. *Proc. ACM Program. Lang.* 6, POPL (2022), 1–28. https://doi.org/10.1145/3498672

[14] Alexandre Moine, Arthur Charguéraud, and François Pottier. 2023. A High-Level Separation Logic for Heap Space under Garbage Collection. *Proc. ACM Program. Lang.* 7, POPL (2023), 718–747. https://doi.org/10.1145/3571218

[15] Simon Spies, Lennard Gäher, Daniel Gratzer, Joseph Tassarotti, Robbert Krebbers, Derek Dreyer, and Lars Birkedal. 2021. Transfinite Iris: resolving an existential dilemma of step-indexed separation logic. In *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, Stephen N. Freund and Eran Yahav (Eds.). ACM, 80–95. https://doi.org/10.1145/3453483.3454031

[16] Simon Spies, Lennard Gäher, Joseph Tassarotti, Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. 2022. Later credits: resourceful reasoning for the later modality. *Proc. ACM Program. Lang.* 6, ICFP (2022), 283–311. https://doi.org/10.1145/3547631

[17] Amin Timany and Lars Birkedal. 2019. Mechanized relational verification of concurrent programs with continuations. *Proc. ACM Program. Lang.* 3, ICFP (2019), 105:1–105:28. https://doi.org/10.1145/3341709

[18] Amin Timany, Léo Stefanesco, Morten Krogh-Jespersen, and Lars Birkedal. 2018. A logical relation for monadic encapsulation of state: proving contextual equivalences in the presence of runST. *Proc. ACM Program. Lang.* 2, POPL (2018), 64:1–64:28. https://doi.org/10.1145/3158152

[19] Simon Vindum and Lars Birkedal. 2023. Spirea: A Mechanized Concurrent Separation Logic for Weak Persistent Memory. *Proc. ACM Program. Lang.* OOPSLA (2023).

[20] Simon Friis Vindum, Aïna Linn Georges, and Lars Birkedal. 2024. Artifact for The Nextgen Modality: A Modality for Non-Frame-Preserving Updates in Separation Logic. https://doi.org/10.5281/zenodo.14285837

Received 2024-09-16; accepted 2024-11-19