Transfinite Iris: Resolving an Existential Dilemma of Step-Indexed Separation Logic

Simon Spies
MPI-SWS and
Saarland University
Germany
spies@mpi-sws.org

Lennard Gäher
MPI-SWS and
Saarland University
Germany
gaeher@mpi-sws.org

Daniel Gratzer
Aarhus University
Denmark
gratzer@cs.au.dk

Joseph Tassarotti
Boston College
USA
tassarot@bc.edu

Robbert Krebbers
Radboud University Nijmegen
The Netherlands
mail@robbertkrebbers.nl

Derek Dreyer
MPI-SWS
Germany
dreyer@mpi-sws.org

Lars Birkedal
Aarhus University
Denmark
birkedal@cs.au.dk

Abstract

Step-indexed separation logic has proven to be a powerful tool for modular reasoning about higher-order stateful programs. However, it has only been used to reason about safety properties, never liveness properties. In this paper, we observe that the inability of step-indexed separation logic to support liveness properties stems fundamentally from its failure to validate the *existential property*, connecting the meaning of existential quantification inside and outside the logic. We show how to validate the existential property—and thus enable liveness reasoning—by moving from finite step-indices (natural numbers) to *transfinite* step-indices (ordinals). Concretely, we transform the Coq-based step-indexed logic Iris to **Transfinite Iris**, and demonstrate its effectiveness in proving termination and termination-preserving refinement for higher-order stateful programs.

CCS Concepts: • **Theory of computation → Separation logic**; **Hoare logic**.

Keywords: Separation logic, Iris, liveness properties, step-indexing, transfinite, ordinals

ACM Reference Format:
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 *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '21), June 20–25, 2021, Virtual, Canada.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3453483.3454031

PLDI '21, June 20–25, 2021, Virtual, Canada

ACM ISBN 978-1-4503-8391-2/21/06.
https://doi.org/10.1145/3453483.3454031

1 Introduction

In the past decade, *separation logics* [51] have emerged as an essential tool for verifying complex stateful programs. Of particular note are the so-called *step-indexed* separation logics, including VST [5, 15, 31], HOCAP [55], iCAP [54], and Iris [35–37, 41]. The distinguishing feature of these step-indexed separation logics is their ability to reason modularly about programs—and their ability to build semantic models of programming languages—with "cyclic" features like recursive types and higher-order state (pointers to higher-order objects). Step-indexing has proven indispensable in a variety of major verification efforts, in languages ranging from C [5] to Go [16] to OCaml [48] to Rust [20, 33, 34] to Scala [29].

Unfortunately, all the existing step-indexed separation logics suffer from a shared Achilles heel: they support reasoning about *safety* properties ("bad things never happen"), but not *liveness* properties ("good things eventually happen"). There is a simple intuitive explanation for this limitation: the whole idea of step-indexing is to give semantics to a program based only on its finitary behavior (*i.e.,* the finite prefixes of its traces), and safety properties are precisely those properties of a program that can be determined from examining its finitary behavior. In contrast, determining whether a program satisfies a liveness property fundamentally requires examination of its infinite traces.

Nevertheless, as we will show in this paper, it *is* in fact possible to equip step-indexed separation logics with support for liveness reasoning. Specifically, we will show how to transform the step-indexed separation logic *Iris* into a new logic **Transfinite Iris** that (unlike Iris) supports the verification of two essential liveness properties—*termination* and *termination-preserving refinement*—in the presence of higher-order state. In order to do so, we need to revisit the most basic foundations of step-indexed separation logics, because it turns out that the root of the problem concerns the very notion of what a "step-index" is. But before we get there, let us begin with a concrete example to illustrate the kind of properties we are interested in proving.

A motivating example. Consider the following example, a combinator for *recursive memoization* written in OCaml[1]:

```
1  let memo_rec t =
2    let m = Hashtbl.create 0 in
3    let rec g x =
4      match Hashtbl.find_opt m x with
5      | Some y -> y
6      | None   -> let y = t g x in
7                  Hashtbl.add m x y; y
8    in g
```

Recursive memoization is a well-known optimization technique for recursive functions: results of recursive calls are cached and retrieved whenever those calls are executed again. To memoize a recursive function `f: σ -> τ`, the combinator

```
memo_rec: ((σ -> τ) -> (σ -> τ)) -> (σ -> τ)
```

is applied to a *template* `t: (σ -> τ) -> (σ -> τ)` of `f`. In addition to the argument of type σ, the template takes an argument of type `σ -> τ` to use for making recursive calls. The recursive function `let rec f x = e` is memoized by:

```
let t = fun f x -> e
let f_memo = memo_rec t
```

The implementation of `memo_rec` uses a map `m` (here a hash map of initial size 0) to store results. The resulting function `f_memo` behaves like `f`, except for one key difference: it retrieves entries from the map `m` if they have been computed previously, and stores results in `m` after it has computed them.

Now, consider what is required to *verify* `memo_rec`. As a combinator, `memo_rec` is written in a generic fashion (*i.e.,* parametric over the template `t`) and does not impose many restrictions on the template `t`, and hence the original function `f`. For example, if the original function `f` diverges on argument `x`, so will the memoized version `f_memo`. Moreover, the original function `f` does not need to be verified itself, or even have a known specification for `memo_rec` to be of use. In short, the correct behavior of `f_memo` is *relative* to the possible behavior of `f`. One way to establish this formally is by showing the memoized function `f_memo` to be a *refinement* of the original function `f`, meaning that the behaviors exhibited by `f_memo` are *contained within* those exhibited by `f`.

On the one hand, due to the presence of higher-order state (the type τ of values stored in the hash table is arbitrary), step-indexed separation logics are one of the only tools available for proving this type of refinement. On the other hand, there is an important caveat: the refinement these logics support merely establishes that *if* `f_memo v` *terminates* with a result `r`, then `f v` terminates with a related result (we call this a *result refinement*). This result refinement says nothing, however, about what happens if `f_memo v` *diverges* (*i.e.,* does not terminate). For example, if we were to replace `t g x` with `g x` in line 6 of the definition of `memo_rec`, then the resulting function returned by `memo_rec` would still be a refinement of `f` according to the theorem provable in existing step-indexed logics—yet it would diverge on every input!

What we would really like to prove is a stronger theorem, stating additionally that if `f_memo v` has a non-terminating execution, then so does `f v`—or equivalently, if `f v` always terminates, then so does `f_memo v`. In this case, we say that `f_memo` is a *termination-preserving refinement* of `f`. Proving this refinement, however, requires examining infinite traces of `f_memo`'s behavior—in other words, *liveness* reasoning.

There has been some prior work on proving *approximations* of liveness reasoning within existing step-indexed logics. In particular, Dockins and Hobor [21, 22] and Mével et al. [47] have shown how to prove termination if the user gives explicit time complexity bounds. Tassarotti et al. [58] have shown how to prove termination-preserving refinement under some restrictions: the original (source) program must only exhibit bounded nondeterminism, and internally, the refinement proof can only rely on bounded stuttering (with the bound chosen up front).

In all of the above, however, the restrictions effectively serve to turn the property being proven from a liveness property into a safety property. For example, although "e terminates" is a liveness property, "e terminates in n steps" (where n is an explicit user-specified bound) is a safety property, since its validity can be determined after examining only the first n steps of e's execution. Moreover, the restrictions of Tassarotti et al. [58]'s approach render it insufficient to prove termination-preserving refinement for even a seemingly simple example like `memo_rec`, since the number of steps required for the hash table lookup in `memo_rec` is unbounded.

Transfinite Iris and the existential property. In this paper, we show how step-indexed separation logics can be transformed to support *true* liveness reasoning, albeit with a fundamental change to how they are modeled.

Our first step is to identify the key property that existing step-indexed separation logics fail to satisfy, but that would enable liveness reasoning if it held. We observe that what these logics are missing is the *existential property*:

$$\text{If } \models \exists x : X.\, \Phi\, x, \text{ then } \models \Phi\, x \text{ for some } x : X.$$

Here, $\models P$ means P is true in the step-indexed logic. The existential property ensures that existential quantification *inside the step-indexed logic* corresponds to existential quantification *outside the step-indexed logic* (*i.e.,* at the meta-level).

Intuitively, the existential property is useful for liveness reasoning because, when we do liveness reasoning inside a step-indexed logic, we often need to prove propositions that are existentially quantified. For example, when we prove a termination-preserving refinement, we will end up needing to show that for any steps of execution in the target program (*e.g.,* `memo_rec t`), there *exist* some corresponding steps in the source program (*e.g.,* the original recursive function `f`). The existential choice of source steps is made *inside* the proof in the step-indexed logic, but ultimately in order to establish

[1]For this example, we use OCaml syntax. In principle, any higher-order stateful language would suffice, including Java, Python, and JavaScript.

the termination-preserving refinement, we need to be able to hoist that existential choice out to the meta-level. The existential property enables us to do just that.

In the models of *non*-step-indexed program logics, the existential property often holds *by definition* of $\exists x : X. \Phi\, x$. Unfortunately, the existential property is fundamentally incompatible with how step-indexed logics have thus far been modeled. In particular, most existing step-indexed logics model propositions as predicates over a "step-index" in the form of a *natural number n*. In this standard model, the existential property is demonstrably false (§2.7).

To validate the existential property, and thereby enable liveness reasoning, we thus propose to change the underlying notion of "step-index" from a finite one to a *transfinite* one: from natural numbers to *ordinals*. Concretely, we do this for one of the most widely used step-indexed logics, Iris [35–37, 41], resulting in a new logic we call **Transfinite Iris**. We use Transfinite Iris to establish two key liveness properties—*termination* and *termination-preserving refinement*—and apply it to a range of interesting examples.

In order to get to the heart of our core "existential dilemma" (*i.e.,* how the existential property can enable liveness reasoning for step-indexed logics, and how to provide a semantic foundation for it), we focus our attention in this paper on one of the primary *raisons d'être* of step-indexed separation logics: reasoning about *sequential, higher-order stateful programs*. Of course, one of the original motivations for Iris was to support *safety reasoning about concurrent programs*, and Transfinite Iris inherits that support. But we leave step-indexed *liveness reasoning about concurrent programs* as an important direction for future work.

While the idea of transfinite step-indexing is not new (see §8 for a discussion of related work), Transfinite Iris is to our knowledge the first step-indexed program logic that satisfies the existential property, thereby supporting true liveness reasoning.

Contributions. We make the following contributions:

- We identify the *existential property* as the key property missing from existing step-indexed separation logics to support liveness reasoning, and we show that this property is fundamentally inconsistent with standard step-indexed models (§2).
- Based on the Iris logic, we develop a new logic, Transfinite Iris, which supports verification of termination-preserving refinement (§4) and termination (§5).
- We demonstrate the power of Transfinite Iris on a range of interesting examples, including the `memo_rec` example presented above (§4.3 and §5.2).
- As part of the foundation of Transfinite Iris, we solve a new and challenging recursive domain equation for modeling the type of Transfinite Iris propositions. We describe the construction at a high level in §6, and elaborate on the details in the appendix [52].
- All examples in §4.3 and §5.2, as well as Transfinite Iris, are fully mechanized in Coq using the Iris Proof Mode [40, 42]. See [52] for the Coq proofs.

2 Key Idea: The Existential Property

In this section, we explain how the *existential property* is key to enabling step-indexed logics to prove termination and termination-preserving refinement. We first define refinements (§2.1) and explain how they are proven using simulations (§2.2). We then show why step-indexing is useful for defining such simulations but falls short in the case of termination-preserving refinements (§2.3). Next, we show how step-indexing is internalized in a logic (§2.4) and how a step-indexed logic with the existential property enables proofs of termination-preserving refinements (§2.5) and termination (§2.6). Finally, we describe how the existential property is justified by transfinite step-indexing (§2.7).

2.1 Refinements

Intuitively, a refinement between a target program t and a source program s expresses that all observable behaviors of the target t are also valid behaviors of the source s. To make this notion precise (and to distill the difference from prior work), we fix an abstract and simplified setting. We assume a source language S and a target language T. We assume programs in both languages are expressions, equipped with a small-step operational semantics. We write $s \rightsquigarrow_{\text{src}} s'$ for a step of the source language, and $t \rightsquigarrow_{\text{tgt}} t'$ for a step of the target language. We assume (for simplicity) that the only values in both languages are Booleans, denoted by b.

To clarify what a refinement is in this abstract setting, we have to specify what the "observable behaviors" of a program should be. In the simplest case, the only observable behavior is the result of the evaluation of a program. In this case, the corresponding refinement between target t and source s, which we dub a *result refinement*, is given by:

> For all b, if t evaluates to b, then s evaluates to b.

Here, "evaluates to b" means there exists an execution that ends in the Boolean value b.

For a termination-preserving refinement, we additionally consider divergence (*i.e.,* non-termination) as an observable behavior. Formally, a *termination-preserving refinement* between target t and source s is given by:

(1) For all b, if t evaluates to b, then s evaluates to b.
(2) If t diverges, then s diverges.

Here, "diverges" means there exists a divergent execution. This refinement, as the name suggests, preserves termination[2] from the source s to the target t.

[2]That is, "if s terminates on all execution paths, then t terminates on all execution paths" is (classically) equivalent to "if t diverges, then s diverges".

2.2 Proving Refinements using Simulations

A well-known technique to prove refinements is to (1) give a small-step simulation ($\preceq$) between target and source expressions, and (2) show that the simulation is *adequate*, *i.e.,* for every target and source expressions t and s, the simulation $t \preceq s$ implies the desired refinement between t and s.

For example, we can prove a termination-preserving refinement by establishing that s simulates t in lock-step, as captured by the following coinductively-defined relation:

$$t \preceq s \triangleq_{\text{coind}} (\exists(b : \mathbb{B}).\, t = s = b) \vee \begin{pmatrix}(\exists t'.\, t \rightsquigarrow_{\text{tgt}} t') \wedge \\ \forall t'.\, t \rightsquigarrow_{\text{tgt}} t' \Rightarrow \exists s'.\, s \rightsquigarrow_{\text{src}} s' \wedge t' \preceq s'\end{pmatrix}$$

In the definition of $t \preceq s$, either both sides have reached the same result b, or the target t can take some step (to avoid cases where the target is a Boolean different from the source), and every step of the target ($t \rightsquigarrow_{\text{tgt}} t'$) can be replayed in the source ($s \rightsquigarrow_{\text{src}} s'$) such that the resulting expressions t' and s' are again in the simulation.

2.3 Step-Indexed Simulations

While the coinductively-defined simulation relation from §2.2 is intuitive and suffices to prove refinements of first-order programs, it falls short when considering programming languages with "cyclic" features like recursive types and higher-order state (pointers to higher-order objects).

Step-indexing [1, 2, 6] has proved to be a very fruitful technique for defining simulation relations for such languages, as shown by the abundance of work on step-indexed techniques for proving result refinements, *e.g.,* [3, 24–26, 42, 43, 59–61]. The key idea of step-indexing is to stratify the simulation relation into a family of approximations ($\preceq_i$), indexed by a natural number i, called the *step-index*:

$$t \preceq_0 s \triangleq \mathsf{True}$$
$$t \preceq_{i+1} s \triangleq (\exists(b : \mathbb{B}).\, t = s = b) \vee \begin{pmatrix}(\exists t'.\, t \rightsquigarrow_{\text{tgt}} t') \wedge \\ \forall t'.\, t \rightsquigarrow_{\text{tgt}} t' \Rightarrow \exists s'.\, s \rightsquigarrow_{\text{src}} s' \wedge t' \preceq_i s'\end{pmatrix}$$

The above definition is structurally recursive on the natural number i. The simulation relation $t \preceq s$ is then redefined as the limit of the approximations, *i.e.,* $t \preceq s \triangleq \forall i.\, t \preceq_i s$.

While the simplified form of step-indexed simulation relation given above does not show it, the step-index i is commonly used as "fuel" to incorporate additional information into the simulation relation related to the unfolding level of recursive types [1], or recursively-defined worlds [12] (which are used to model higher-order references).

Step-indexing works well for result refinements, since it suffices to only inspect finite prefixes of the evaluation (the kind of reasoning step-indexing was designed for):

Lemma 2.1. *If $t \preceq s$, then t is a result refinement of s, i.e., if t evaluates to b, then s evaluates to b.*

Proof Sketch. Let $t = t_0 \rightsquigarrow_{\text{tgt}} \cdots \rightsquigarrow_{\text{tgt}} t_n = b$ be the execution of t to b. Recall that $t \preceq s$ means $\forall i.\, t \preceq_i s$. We pick $i \triangleq n + 1$ and extract an execution $s = s_0 \rightsquigarrow_{\text{src}} \cdots \rightsquigarrow_{\text{src}} s_n$ such that $t_n \preceq_1 s_n$ from $t \preceq_{n+1} s$ by unrolling the definition of $(\preceq_i)$ n times. Since $t_n = b$, the expression t_n can no longer take steps. Consequently, in the definition of $t_n \preceq_1 s_n$, only the first clause can be true. We obtain $s_n = b$. □

Unfortunately, unlike the coinductively-defined simulation relation from §2.2, the step-indexed simulation relation is *not* adequate for termination-preserving refinements. Let us try to prove that it implies a termination-preserving refinement:

If $t \preceq s$ and t diverges, then s diverges.

and see where the argument goes wrong. If we attempt a proof similar to the proof of Lemma 2.1, we are stuck when we try to determine a sufficient value of the step-index i. We have seen that for any natural number i, we can extract a finite trace of the source (of length i) from $t \preceq_i s$. However, which execution we obtain this way can depend on the step-index i, meaning for each step-index i there could be a different (finite) execution of the source.

For example, consider the case where the target t_∞ is an infinite loop, and the source $s_{<\infty}$ non-deterministically picks a natural number n, and then executes n steps before terminating. For every i, we can find a trace of i steps where $s_{<\infty}$ simulates t_∞, but there is no divergent execution of $s_{<\infty}$. Thus, we cannot extract one coherent, infinite execution of $s_{<\infty}$ from the step-indexed simulation $t_\infty \preceq s_{<\infty}$.

2.4 Logical Step-Indexed Simulations

To show how we will remedy the inability of step-indexing to prove termination-preserving refinements, we will take a look at the logical approach to step-indexing [7, 24] as employed in step-indexed logics like Iris. The logical approach to step-indexing does away with explicit indexing by natural numbers, and instead internalizes step-indexing using the "later" modality ($\triangleright$) [7, 49]. We consider a version of our simulation relation that is defined using logical step-indexing:

$$t \preceq_* s \triangleq (\exists(b : \mathbb{B}).\, t = s = b) \vee \begin{pmatrix}(\exists t'.\, t \rightsquigarrow_{\text{tgt}} t') \wedge \\ \forall t'.\, t \rightsquigarrow_{\text{tgt}} t' \Rightarrow \exists s'.\, s \rightsquigarrow_{\text{src}} s' \wedge \triangleright(t' \preceq_* s')\end{pmatrix}$$

The simulation relation $\preceq_*$ is no longer defined in ordinary mathematics, but rather *within* a step-indexed logic like Iris. One can think of the propositions of a step-indexed logic as sets of natural numbers. All logical connectives are lifted in the expected ways to such sets, while the later modality is defined as $\triangleright P \triangleq \{i \in \mathbb{N} \mid (i = 0) \vee (i - 1) \in P\}$, *i.e.,* it decrements the step-index. P is true in the step-indexed logic, written $\models P$, if it contains every step-index: $\models P \triangleq \forall i.\, i \in P$.

2.5 The Existential Property

In step-indexed logics like Iris, one cannot prove that the simulation relation $\preceq_*$ implies a termination-preserving refinement. After all, when expanding the definition of $\preceq_*$ into the step-indexed model, we end up with essentially the same definition as the explicit step-indexed relation $\preceq_i$ we have seen in §2.3. If, however, instead of thinking of step-indexed propositions as sets of natural numbers, we adopt a higher-level perspective, we may rightly wonder: what property are step-indexed logics missing that prohibits us from proving termination-preserving refinements? The answer to this question is the *existential property*:

$$\text{If } \models \exists x : X.\, \Phi\, x, \text{ then } \models \Phi\, x \text{ for some } x : X.$$

If we imagine we were working in a step-indexed logic that enjoyed the existential property, then we could, in fact, prove:

Lemma 2.2. *If $\models t \preceq_* s$ and t diverges, then s diverges.*

Proof Sketch. Let $t = t_0 \rightsquigarrow_{\text{tgt}} t_1 \rightsquigarrow_{\text{tgt}} \cdots$ be an infinite target execution. By coinduction, we will construct an infinite source execution $s = s_0 \rightsquigarrow_{\text{src}} s_1 \rightsquigarrow_{\text{src}} \cdots$. Initially, we know $\models t \preceq_* s$. With $t = t_0 \rightsquigarrow_{\text{tgt}} t_1$, we can use $\models t \preceq_* s$ to obtain $\models \exists s'.\, s \rightsquigarrow_{\text{src}} s' \land \triangleright(t_1 \preceq_* s')$. With the existential property, we obtain an s_1 such that $s \rightsquigarrow_{\text{src}} s_1$ and $\models \triangleright(t_1 \preceq_* s_1)$. Step-indexed logics enjoy the rule $\models \triangleright P$ implies $\models P$, allowing us to strip off the later modality. Thus, we obtain $\models t_1 \preceq_* s_1$. We obtain $s_2, s_3, \ldots$ by coinduction for $\models t_1 \preceq_* s_1$. □

2.6 Termination

The simulation relation ($\preceq_*$) can be repurposed to prove another liveness property: *termination*. In particular, observe that the source language in our simulation does not have to be a programming language—it merely has to be a transition system. If we instantiate it with a relation that always terminates, *e.g.*, the order ($>$) on natural numbers or on ordinals, we are guaranteed termination of the target.

Lemma 2.3. *If $\models t \preceq_* s$ for some s, and the source relation ($\rightsquigarrow_{src}$) is the inverse of a well-founded relation, then t terminates along all execution paths.*

2.7 Justifying the Existential Property

Of course, we cannot simply postulate the existential property: we have to justify that it is a sound extension of existing step-indexed logics like Iris. Unfortunately, it is not. To demonstrate this, we instantiate the existential property

$$\text{If } \models \exists x : X.\, \Phi\, x, \text{ then } \models \Phi\, x \text{ for some } x : X.$$

with the type $X \triangleq \mathbb{N}$ and the predicate $\Phi(n) \triangleq \triangleright^n \bot$, where $\triangleright^n$ is short for n iterated later modalities. In Iris, the proposition in the premise $\models \exists n : \mathbb{N}.\, \triangleright^n \bot$ is provable, while the conclusion $\models \triangleright^n \bot$ is false for any n.

To see why that is the case, we take a closer look at the proposition $\exists n : \mathbb{N}.\, \triangleright^n \bot$. Intuitively, this proposition means that eventually the step-index runs out. More precisely, if we drop down to the step-indexed model, we see that $\models \exists n : \mathbb{N}.\, \triangleright^n \bot$ means $\forall i : \mathbb{N}.\, \exists n : \mathbb{N}.\, n > i$, which holds trivially by picking the witness $n \triangleq i + 1$. In contrast, $\models \triangleright^n \bot$ is false for any $n : \mathbb{N}$ because it is false at all step-indices larger than n.

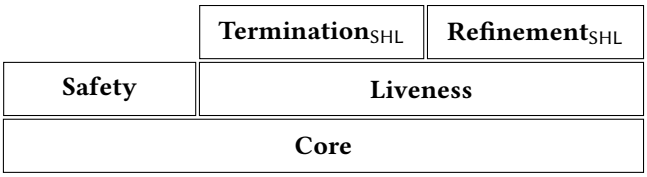

Figure 1. Roadmap of Transfinite Iris

Transfinite step-indexing to the rescue. How can we transform Iris (to Transfinite Iris) so that it will enjoy the existential property? The fundamental modification we make is to move from *finite* step-indexing with natural numbers to *transfinite* step-indexing with ordinals.

To explain how transfinite step-indexing validates the existential property, we consider what went wrong in the case of step-indexing with natural numbers. Both in the above example $\models \exists n : \mathbb{N}.\, \triangleright^n \bot$ and the simulation $\models t_\infty \preceq_* s_{<\infty}$ from §2.3, the problem was that the witnesses of existential quantification could depend on the step-index i. For $\models \exists n : \mathbb{N}.\, \triangleright^n \bot$, given any step-index i, we could always pick a witness n greater than i; for $\models t_\infty \preceq_* s_{<\infty}$, given any step-index i, we could always pick an execution of $s_{<\infty}$ that takes longer than i steps to terminate. However, if we use ordinals as step-indices, then this is no longer the case. For example, there is no $n : \mathbb{N}$ that is larger than ω, which is (by definition) the first ordinal larger than all natural numbers. As a result, $\models \exists n : \mathbb{N}.\, \triangleright^n \bot$ is no longer provable in a transfinitely step-indexed model. In return, as we show formally in §6, a model with sufficiently large ordinals *does* validate the existential property, thus enabling liveness reasoning.

3 Roadmap of Transfinite Iris

We now put the key ideas from §2 into action by introducing **Transfinite Iris**—a step-indexed separation logic framework capable of proving safety, termination, and termination-preserving refinements of higher-order stateful programs. We start in this section with a tour of the components of the framework (depicted in Figure 1). Along the way, we highlight which aspects of Transfinite Iris are original and which are inherited from Iris.

We start with the **core logic** of Transfinite Iris. Like Iris, the core of Transfinite Iris is a step-indexed logic of bunched implications [50] with resources and invariants. In layman's terms this means that the core logic of Transfinite Iris has the typical connectives of step-indexed logic (*e.g.*, the later modality) and separation logic (*e.g.*, separating conjunction), but no connectives typically found in program logics yet (*e.g.*, Hoare triples). To be precise, the core logic has all the

connectives of Iris's base logic and, additionally, Iris's impredicative invariants [35, 54]. It does not, however, have exactly the same rules and properties as Iris's base logic: in particular, the core logic enjoys the existential property, which Iris does not, but in exchange it loses two of Iris's commuting rules which are in conflict with the existential property (see §7).

While, on the surface, the core logic differs only marginally from Iris, it differs radically on the inside! In Transfinite Iris, to validate the existential property, we must change the ambient step-indexed model used to define the core logic. This change constitutes the titular difference from Iris: in the model, we use *transfinite* step-indexing (*i.e.,* with ordinals), whereas Iris's model is based on finite step-indexing (*i.e.,* with natural numbers). We postpone further discussion of the details of this change until §6.

To enable program verification, we extend the core logic with two program logics: one for proving *safety properties* and one for proving *liveness properties*. The **safety logic** is inherited from Iris with only small modifications. (As explained in §7, we need to account for the loss of the two commuting rules in the core logic.) Due to the similarity to Iris, we do not discuss safety to a large extent in the paper.

The **liveness logic** is a new contribution of Transfinite Iris. It is a generic program logic with constructs for proving termination preservation from source to target. Below, we consider two instantiations with Sequential HeapLang (SHL)—the sequential fragment of Iris's example language HeapLang. In §4, we choose SHL as both source and target to obtain $\textbf{Refinement}_{\mathsf{SHL}}$, a program logic for proving termination-preserving refinements. Then, in §5, we choose ordinals as the source and SHL as the target to obtain $\textbf{Termination}_{\mathsf{SHL}}$, a program logic for proving termination.

4 Termination-Preserving Refinement

In this section, we introduce $\textbf{Refinement}_{\mathsf{SHL}}$—Transfinite Iris's program logic for proving termination-preserving refinements in Sequential HeapLang (see Figure 2). More precisely, we first review the canonical approach for internalizing result refinements in separation logic pioneered by Turon et al. [60] (§4.1), then explain how $\textbf{Refinement}_{\mathsf{SHL}}$ goes beyond this approach to handle termination-preserving refinements (§4.2), and finally apply $\textbf{Refinement}_{\mathsf{SHL}}$ to verify the challenging `memo_rec` example from the introduction (§4.3).

4.1 Result Refinements in Iris

Transfinite Iris follows the "Iris approach" of modularly building up complex reasoning principles from simple abstractions. As such, $\textbf{Refinement}_{\mathsf{SHL}}$ does not have a simulation relation such as $(\preceq_*)$ from §2.4 built in as a primitive logical connective. Instead, its simulation relation is defined in terms of simpler connectives—Hoare triples and separation-logic

$$
\begin{aligned}
v \in \mathit{Val} &::= () \mid b \mid z \mid \ell \mid (v_1, v_2) \mid \mathbf{rec}\, f\, x.\, e \mid \cdots \\
e \in \mathit{Expr} &::= v \mid x \mid e_1\, e_2 \mid \mathbf{ref}(e) \mid !e \mid e_1 := e_2 \mid \cdots \\
K \in \mathit{Ctx} &::= \cdot \mid K\, v \mid e\, K \mid \mathbf{ref}(K) \mid !K \mid \cdots \\
h \in \mathit{Heap} &::= \cdot \mid \ell \mapsto v, h
\end{aligned}
$$

where b ranges over Booleans, z ranges over integers, and ℓ ranges over heap locations.

Figure 2. Sequential HeapLang (SHL)

resources—following the approach of Turon et al. [60]. Below, we recall the approach of Turon et al. for proving result refinements in the context of finite Iris, and then explain how we extend the approach to prove termination-preserving refinements in §4.2. (We do not discuss how Hoare triples and resources are themselves defined; for such a discussion, we refer the reader to the supplementary material [52].)

While Turon et al. use a bespoke logic, we consider a variant embedded in Iris, based on work by Krebbers et al. [42]. The central connectives of this variant are:

$$P, Q \in \mathit{iProp} ::= \cdots \mid \ell \mapsto v \mid \{P\}\ e\ \{v.\, Q\} \mid \ell \mapsto_{\mathrm{src}} v \mid \mathsf{src}(e)$$

Hoare triples $\{P\}\ e\ \{v.\, Q\}$ (where "$v.$" is a binder for the result of executing e) are used to reason about the target, while resources $\mathsf{src}(e)$ are used to reason about the source. Similar to the usual points-to connective $\ell \mapsto v$, which states that location ℓ holds value v in the target heap, there is a points-to connective $\ell \mapsto_{\mathrm{src}} v$ for the source heap. The expressions and values are drawn from Sequential HeapLang (see Figure 2)—the sequential fragment of HeapLang, the default language in Iris's mechanization [32]. SHL offers the features of a standard (untyped) functional programming language augmented with ML-like references.

We define the analog of $(\preceq_*)$ from §2, generalized to arbitrary ground types G—*e.g.,* unit ($\mathbb{1}$), Booleans ($\mathbb{B}$), natural numbers ($\mathbb{N}$)—as:

$$e_t \preceq_G e_s \triangleq \forall K.\, \{\mathsf{src}(K[e_s])\}\ e_t\ \{v.\, \mathsf{src}(K[v]) * v \in G\}$$

Recall that when proving a refinement between e_t and e_s, we need to show that for every execution of e_t there is a corresponding execution of e_s. Following Turon et al. [60] we use the Hoare triple $\{P\}\ e_t\ \{v.\, Q\}$ to reason about *every* execution of e_t, and the separation logic resource $\mathsf{src}(e_s)$ to reason about the existence of *some* execution of e_s. Intuitively, $\mathsf{src}(e_s)$ says that the program on the RHS of the refinement is currently e_s. The proof rules allow one to transform $\mathsf{src}(e_s)$ into $\mathsf{src}(e_s')$ if and only if e_s reduces to e_s'.

Hence, $\{\mathsf{src}(e_s)\}\ e_t\ \{v.\, \mathsf{src}(v) * v \in G\}$ says that if the RHS initially is e_s, then for every execution of e_t that results in v, we can reduce the RHS to the same value v. Additionally, the definition of the refinement $e_t \preceq_G e_s$ quantifies over all evaluation contexts K to make the judgment compositional; at the top level we take K to be the empty context.

Basic step-indexed separation logic:

$$\textsc{Value}\quad \{\mathsf{True}\}\ v\ \{w.\ v = w\} \qquad \textsc{Frame}\quad \frac{\{P\}\ e_t\ \{v.\ Q\}}{\{P * R\}\ e_t\ \{v.\ Q * R\}} \qquad \textsc{Bind}\quad \frac{\{P\}\ e_t\ \{v.\ Q\} \qquad \forall v.\ \{Q\}\ K[v]\ \{w.\ R\}}{\{P\}\ K[e_t]\ \{w.\ R\}} \qquad \textsc{Löb}\quad (\triangleright P \Rightarrow P) \Rightarrow P$$

***Result refinements in Iris** (these rules are not present in **Refinement**$_{SHL}$):*

$$\textsc{PureT}\quad \frac{\{P\}\ e_t'\ \{v.\ Q\} \qquad e_t \rightsquigarrow e_t'}{\{\triangleright P\}\ e_t\ \{v.\ Q\}} \qquad \textsc{PureS}\quad \frac{\{\mathsf{src}(K[e_s']) * P\}\ e_t\ \{v.\ Q\} \qquad e_s \rightsquigarrow e_s'}{\{\mathsf{src}(K[e_s]) * P\}\ e_t\ \{v.\ Q\}}$$

$$\textsc{StoreT}\quad \{\ell \mapsto v_1 * \triangleright P\}\ \ell := v_2\ \{w.\ (w = ()) * \ell \mapsto v_2 * P\} \qquad \textsc{StoreS}\quad \frac{\{\ell \mapsto_{\mathrm{src}} v_2 * \mathsf{src}(K[()])\}\ e_t\ \{v.\ Q\}}{\{\ell \mapsto_{\mathrm{src}} v_1 * \mathsf{src}(K[\ell := v_2])\}\ e_t\ \{v.\ Q\}}$$

Termination-preserving refinements in Refinement$_{SHL}$:

$$\textsc{TPPureT}\quad \frac{\{P\}\ e_t'\ \{v.\ Q\} \qquad e_t \rightsquigarrow e_t'}{\langle P\rangle\ e_t\ \langle v.\ Q\rangle} \qquad \textsc{TPStoreT}\quad \langle \ell \mapsto v_1\rangle\ \ell := v_2\ \langle w.\ (w = ()) * \ell \mapsto v_2\rangle$$

$$\textsc{TPPureS}\quad \frac{\langle \mathsf{src}(K[e_s']) * P\rangle\ e_t\ \langle v.\ Q\rangle \qquad e_s \rightsquigarrow e_s' \qquad e_t \notin \mathit{Val}}{\{\mathsf{src}(K[e_s]) * \triangleright P\}\ e_t\ \{v.\ Q\}} \qquad \textsc{TPStoreS}\quad \frac{\langle \ell \mapsto_{\mathrm{src}} v_2 * \mathsf{src}(K[()]) * P\rangle\ e_t\ \langle v.\ Q\rangle \qquad e_t \notin \mathit{Val}}{\{\ell \mapsto_{\mathrm{src}} v_1 * \mathsf{src}(K[\ell := v_2]) * \triangleright P\}\ e_t\ \{v.\ Q\}}$$

$$\textsc{TPStutterT}\quad \frac{\langle P\rangle\ e_t\ \langle v.\ Q\rangle \qquad e_t \notin \mathit{Val}}{\{P\}\ e_t\ \{v.\ Q\}} \qquad \textsc{TPStutterSPure}\quad \frac{\{\mathsf{src}(e_s') * P\}\ e_t\ \{v.\ Q\} \qquad e_s \rightsquigarrow e_s' \qquad e_t \notin \mathit{Val}}{\{\mathsf{src}(e_s) * P\}\ e_t\ \{v.\ Q\}}$$

$$\textsc{TPStutterSStore}\quad \frac{\{\ell \mapsto_{\mathrm{src}} v_2 * \mathsf{src}(K[()]) * P\}\ e_t\ \{v.\ Q\} \qquad e_t \notin \mathit{Val}}{\{\ell \mapsto_{\mathrm{src}} v_1 * \mathsf{src}(K[\ell := v_2]) * P\}\ e_t\ \{v.\ Q\}}$$

Figure 3. A selection of proof rules of Iris and **Refinement**$_{\mathrm{SHL}}$.

Proof rules. We have two kinds of rules: the usual separation logic rules for Hoare triples (which apply to the target e_t), and the rules for interacting with the resource $\mathsf{src}(e_s)$ for the simulating source expression. A selection of rules is given in Figure 3. As the standard Hoare rules, we have rules for values Value, framing Frame, reasoning about composite expressions Bind, and executing target steps (*i.e.,* the expression in the triple). Among the rules for executing target steps are the rule StoreT for executing a store operation, and PureT for executing *pure* steps, denoted by $(\rightsquigarrow)$. Pure steps are steps that do not affect the heap, *e.g.,* reducing an **if**, a **match**, or a function application $(\mathbf{rec}\ f\ x.\ e)\ v$. As the additional *source rules*, we have StoreS for the store operation in the source, and PureS for pure steps in the source.

Löb induction. Above, we have not yet explained the occurrence of the later modality $(\triangleright)$ in the rules StoreT and PureT. The later modality is used to interact with the various logical mechanisms of Iris based on step-indexing, such as Löb induction, impredicative invariants [54], and higher-order ghost state [35]. In this section we focus on proofs about recursive programs through Löb induction. Using Löb induction, we can prove a proposition P in Iris by assuming $\triangleright P$ (read "P holds later") and then proving P. From this rule, we can derive a reasoning principle in Iris for reasoning about (recursive) programs:

$$\textsc{Hoare-Löb}\quad \frac{\forall \vec{x}.\ \{P * \triangleright (\forall \vec{x}.\ \{P\}\ e\ \{v.\ Q\})\}\ e\ \{v.\ Q\}}{\forall \vec{x}.\ \{P\}\ e\ \{v.\ Q\}}$$

To prove a (universally quantified) Hoare triple, we can assume (in the precondition) that the Hoare triple already holds later. As we will see shortly, the combination of Löb induction and rules like StoreT and PureT is incompatible with proving termination-preserving refinements, so these rules will need to be changed. First, however, let us consider a simple example of using Hoare-Löb:

Lemma 4.1. *If* $f() \preceq_{\mathbb{B}} g()$, *then* $\mathsf{loop}\, f\ () \preceq_{\mathbb{1}} \mathsf{loop}\, g\ ()$, *where* $\mathsf{loop} \triangleq \mathbf{rec}\ \mathsf{loop}\ f x.\ \mathbf{if}\ f()\ \mathbf{then}\ \mathsf{loop}\ f\ x\ \mathbf{else}\ ()$.

Proof Sketch. We abbreviate $e_{\mathrm{f}} \triangleq \mathsf{loop}\, f\ ()$, $e_{\mathrm{g}} \triangleq \mathsf{loop}\, g\ ()$, and $\varphi\ v \triangleq (\mathsf{src}(K[v]) * v \in \mathbb{1})$. Recall that $(\preceq_G)$ is defined in terms of Hoare triples; hence by Hoare-Löb, we have to show $\{\mathsf{src}(K[e_{\mathrm{g}}]) * \triangleright(e_{\mathrm{f}} \preceq_{\mathbb{1}} e_{\mathrm{g}})\}\ e_{\mathrm{f}}\ \{\varphi\}$. By executing a pure step of the recursive function loop in the target (the recursive unfolding) using PureT, we have to, in turn, show $\{\mathsf{src}(K[e_{\mathrm{g}}]) * e_{\mathrm{f}} \preceq_{\mathbb{1}} e_{\mathrm{g}}\}\ \mathbf{if}\ f()\ \mathbf{then}\ e_{\mathrm{f}}\ \mathbf{else}\ ()\ \{\varphi\}$. Note that the later modality $(\triangleright)$ has been stripped from the precondition. Similarly, we can execute the loop function in the source for one step to $\mathbf{if}\ g()\ \mathbf{then}\ e_{\mathrm{g}}\ \mathbf{else}\ ()$ by rule PureS. The

rest then follows by using Bind to execute $f()$ and $g()$ in the source and the target using the assumption $f() \preceq_{\mathbb{B}} g()$. Depending on the outcome, we either (1) end the execution in both the source and the target, or (2) execute the respective recursive occurrence of e_f (resp. e_g) using $e_f \preceq_{\mathbb{1}} e_g$. □

Problem. The sketched approach to refinements in Iris works well for proving result refinements, but is not adequate for terminating-preserving refinements. For example, defining $e_{\text{loop}} \triangleq \text{loop}\,(\lambda\,().\,\text{true})\,()$, we can prove $e_{\text{loop}} \preceq_{\mathbb{1}} \textbf{skip}$ by Löb induction, analogously to the proof of Lemma 4.1, except omitting any source steps (uses of PureS). This is not a termination-preserving refinement, as the target always diverges while the source immediately terminates.

4.2 Termination-Preserving Refinements

We present the logical connectives of **Refinement**$_{\text{SHL}}$, their intuitive semantics, and their proof rules. The simulation relation ($\preceq_G$) remains the same as the one from §4.1. However, since the semantics and proofs rules of **Refinement**$_{\text{SHL}}$ are different than those of Iris, the simulation relation is in fact adequate for termination-preserving refinements (Theorem 4.3). A selection of the proof rules is shown in Figure 3.

Later stripping and source steps. Let us reconsider the argument for why $e_{\text{loop}} \preceq_{\mathbb{1}} \textbf{skip}$ with the rules discussed in §4.1, as alluded to above. There, we could prove a refinement of a diverging target in Iris, without constructing a diverging source execution. To avoid the same happening in **Refinement**$_{\text{SHL}}$, we identify the core problem that allowed the target to diverge: the interplay of Löb induction and the target stepping rules. Specifically, using Löb induction, we assumed the goal under a later modality ($\triangleright$). Once we perform a target step (using PureT or StoreT), we can strip off the later, regardless of whether we have already performed a source step (using PureS or StoreS).

To avoid this issue, we ensure that stripping-off a later requires *both* a target *and* a source step. This is achieved in **Refinement**$_{\text{SHL}}$ by having two notions of Hoare triples. The *source-stepping* Hoare triple $\{P\}\ e\ \{v.\ Q\}$ allows us to perform a step in the source, strip off a later, and continue with the target (TPPureS, TPStoreS). The *target-stepping* Hoare triple $\langle P\rangle\ e\ \langle v.\ Q\rangle$ allows us to perform a step in the target, and continue with the source (TPPureT, TPStoreT).

To strip off a later, we always need a roundtrip between both Hoare triples, thereby necessitating a step in both the target and source. Let us see these Hoare triples in action by reproving Lemma 4.1 from §4.1 in **Refinement**$_{\text{SHL}}$. Recall that since the semantics of Hoare triples changed, this now gives us a termination-preserving refinement.

Lemma 4.2. *If* $f() \preceq_{\mathbb{B}} g()$, *then* $\text{loop}\,f\,() \preceq_{\mathbb{1}} \text{loop}\,g\,()$.

Proof Sketch. We abbreviate $e_f \triangleq \text{loop}\,f\,()$, $e_g \triangleq \text{loop}\,g\,()$, and $\varphi\ v \triangleq (\text{src}(K[v]) * v \in \mathbb{1})$. By Hoare-Löb, we should show $\{\text{src}(K[e_g]) * \triangleright(e_f \preceq_{\mathbb{1}} e_g)\}\ e_f\ \{\varphi\}$. By taking a source step (unfolding the loop) using TPPureS, we have to show:

$$\langle \text{src}(K[\textbf{if}\ g()\ \textbf{then}\ e_g\ \textbf{else}\ ()]) * e_f \preceq_{\mathbb{1}} e_g\rangle\ e_f\ \langle\varphi\rangle$$

Note that the later modality ($\triangleright$) has been stripped from the precondition, and that we have switched to the target-stepping Hoare triple. Similarly, we can execute the loop in the target for one step to $\textbf{if}\ f()\ \textbf{then}\ e_f\ \textbf{else}\ ()$ with PureT, and thereby switch back to the source-stepping Hoare triple. The rest of the proof is analogous to Lemma 4.1. □

Stuttering. So far we have only considered lock-step simulation, *i.e.,* simulations where there is a one-to-one correspondence between target and source steps. For programs like the challenging `memo_rec` example from §1 such a simulation is too restrictive—we need *stuttering* [14].

The most interesting form is *source stuttering*—advancing the target without advancing the source. This is enabled by rule TPStutterT, which allows us to switch from the source-stepping Hoare triple $\{P\}\ e\ \{v.\ Q\}$ to the target-stepping Hoare triple $\langle P\rangle\ e\ \langle v.\ Q\rangle$ without performing a source step. As a consequence of the source stutter, this rule does not allow us to strip off a later modality ($\triangleright$).

Stuttering of the target is allowed by the rules TPStutterSStore and TPStutterSPure. They allow us to perform multiple (but, at least one) source steps before switching from the source-stepping Hoare triple $\{P\}\ e\ \{v.\ Q\}$ to the target-stepping Hoare triple $\langle P\rangle\ e\ \langle v.\ Q\rangle$.

Adequacy. The model of the simulation $e_t \preceq_G e_s$ resembles the simulation from §2.4 if one considers the unfolding of Hoare triples of **Refinement**$_{\text{SHL}}$.[3] Thus, we can prove an adequacy result that shows the simulation really entails a termination-preserving refinement.

Theorem 4.3. *If* $\models e_t \preceq_G e_s$, *then* e_t *is in fact a termination-preserving refinement of* e_s.

Proof Sketch. This proof is similar to that of Lemma 2.2. To prove termination preservation, we construct an infinite execution using Transfinite Iris's existential property. □

4.3 Example Refinement: Memoization

Now that we have seen how termination-preserving refinements can be proven in **Refinement**$_{\text{SHL}}$, we return to the motivating `memo_rec` example from §1. Recall that when `memo_rec` is used to memoize a recursive function, the results of recursive calls are cached in a lookup table and reused. We define `memo_rec` in Sequential HeapLang as:

$$\begin{aligned}\text{memo\_rec} \triangleq \lambda\,t.\ &\textbf{let}\ \mathit{tbl} = \text{map}()\ \textbf{in}\\ &\textbf{rec}\ g\ x.\ \textbf{match}\ \text{get}\ \mathit{tbl}\ x\ \textbf{with}\\ &\quad \mid \text{None} \Rightarrow\ \textbf{let}\ y = t\ g\ x\ \textbf{in}\ \text{set}\ \mathit{tbl}\ x\ y;\ y\\ &\quad \mid \text{Some}\ y \Rightarrow\ y\end{aligned}$$

[3] Technically, it moves away from a lock-step simulation, and incorporates the stuttering to justify what we have just seen above. Details on this change are given in the supplementary material [52].

$$\mathsf{Fib}\ \mathsf{fib}\ n \triangleq \textbf{if}\ n < 2\ \textbf{then}\ n\ \textbf{else}\ \mathsf{fib}(n-1) + \mathsf{fib}(n-2)$$
$$\mathsf{Slen}\ \mathsf{slen}\ s \triangleq \textbf{if}\ !s = 0\ \textbf{then}\ 0\ \textbf{else}\ \mathsf{slen}(s+1) + 1$$
$$\begin{aligned}&\mathsf{Lev}\ \mathsf{slen}\ \mathsf{lev}\ (s,t) \triangleq\\ &\quad \textbf{if}\ !s = 0\ \textbf{then}\ \mathsf{slen}\ t\ \textbf{else}\\ &\quad \textbf{if}\ !t = 0\ \textbf{then}\ \mathsf{slen}\ s\ \textbf{else}\\ &\quad \textbf{if}\ !s = !t\ \textbf{then}\ \mathsf{lev}(s+1, t+1)\ \textbf{else}\\ &\quad 1 + \min(\mathsf{lev}(s, t+1), \mathsf{lev}(s+1, s), \mathsf{lev}(s+1, t+1))\end{aligned}$$

Figure 4. Recursive templates of memoized functions.

Here, t is the *template* of the function we want to memoize. The function map creates a mutable lookup table, which is then accessed using the functions get and set.

In **Refinement**$_{\mathsf{SHL}}$, we proved a generic and modular specification for memo_rec, and used it to verify memoization of two examples: Fibonacci and Levenshtein distance [44]. For both examples, the template code is given in Figure 4. To keep the explanation concrete, we focus here on how our specification for memo_rec is used in these examples. The details of the generic specification can be found in the supplementary material [52].

Pure templates. We start by considering the memoization of *pure* templates for functions of type $\mathbb{N} \to \mathbb{N}$, such as the Fib template for Fibonacci, shown in Figure 4. Given a template t, we define the standard recursive version of the function r_t and the memoized version m_t as:

$$r_t \triangleq \textbf{rec}\ g\ n.\ t\ g\ n \qquad m_t \triangleq \mathsf{memo\_rec}\ t$$

We wish to show that m_t refines r_t, in the sense that for all natural numbers n, $m_t\ n \preceq_{\mathbb{N}} r_t\ n$. When specialized to a pure template t, our specification for memo_rec is

$$\frac{\forall g.\ \triangleright(\forall n.\ g\ n \preceq_{\mathbb{N}} r_t\ n) \Rightarrow \forall n.\ t\ g\ n \preceq_{\mathbb{N}} r_t\ n \qquad r_t \text{ is pure}}{\forall n.\ m_t\ n \preceq_{\mathbb{N}} r_t\ n}\ \textsc{PureMemoRec}$$

The premise requires that when the template t is applied to a function g that is assumed to refine r_t, the result must continue to refine r_t. The assumption about g is guarded by a later modality ($\triangleright$), so a proof of the premise must execute a step of the source ($r_t\ n$) before this assumption can be used. Typically, this is trivial since $r_t\ n \rightsquigarrow t\ r_t\ n$, and for safe, pure templates like Fib, the residual goal ($t\ g\ n \preceq_{\mathbb{N}} t\ r_t\ n$) is provable by a simple lock-step simulation.

Proof Sketch for PureMemoRec. We first derive standard Hoare triples for the operations of the lookup table. After allocating a lookup table *tbl*, the expression m_t reduces to:

$$\begin{aligned}h \triangleq\ &\textbf{rec}\ g\ x.\ \textbf{match}\ \mathsf{get}\ tbl\ x\ \textbf{with}\\ &\mid \mathsf{None} \Rightarrow\ \textbf{let}\ y = t\ g\ x\ \textbf{in}\ \mathsf{set}\ tbl\ x\ y;\ y\\ &\mid \mathsf{Some}\ y \Rightarrow\ y\end{aligned}$$

We must now prove $\forall n.\ h\ n \preceq_{\mathbb{N}} r_t\ n$. We proceed by using Löb induction in a way similar to Lemma 4.2. The induction hypothesis from Löb is $\triangleright(\forall n.\ h\ n \preceq_{\mathbb{N}} r_t\ n)$, which matches the left side of the implication in the premise of PureMemoRec. Given an argument n, the proof case splits on whether the result has already been stored in the lookup table. In case it has not, `memo_rec` calls $t\ h\ n$. Applying the premise of PureMemoRec to the induction hypothesis, we know $t\ h\ n \preceq_{\mathbb{N}} r_t\ n$. The resulting value is then stored in the lookup table. In case n is found in the lookup table, we must argue that the stored value is equal to the result of $r_t\ n$. To do so, we use an invariant assertion to ensure that all values in the lookup table are the result of running r_t on the associated keys. □

Stateful templates. Memoization can also be applied to templates that use state, so long as they execute in a *repeatable* fashion. That is, when the function is run multiple times, it must return the same value. This ensures that it is correct to reuse the stored values in the lookup table during memoization. Our general specification for memoization replaces the purity side condition in PureMemoRec with an encoding of this repeatability condition in Transfinite Iris.

To see an example of where our more general specification is useful, consider the Levenshtein function template in Figure 4, which computes the edit distance between two strings. We parameterize the template by a function `slen` used for computing the length of strings. The input strings are stored on the heap as null-terminated arrays, as in languages like C, and the function takes pointers to these strings. Because the length of the same substring is computed multiple times, the Levenshtein function can be optimized further by additionally memoizing the string length function, using the template Slen from Figure 4. Thus, to memoize the Levenshtein function, we define:

$$\begin{aligned}\mathsf{mlev} \triangleq\ &\textbf{let}\ \mathsf{mslen} = \mathsf{memo\_rec}\ \mathsf{Slen}\ \textbf{in}\\ &\mathsf{memo\_rec}\ (\mathsf{Lev}\ \mathsf{mslen})\end{aligned}$$

The resulting function performs nested memoization: it memoizes the length function slen inside of the template Lev.

This use of the Lev template is not pure: it not only reads from heap allocated state (the string), but also accesses and modifies the internally allocated memo table used in mslen. Nevertheless, the template is repeatable, so long as the input strings are not modified after memoizing.

5 Termination

Recall from §2.6 that termination and termination-preserving refinement are closely related. In the latter, if the source always terminates, then the target always terminates. Thus, we obtain a termination proof technique if the source reduction relation is the inverse of a well-founded relation (*e.g.*, $(>)$ on ordinals).

Below, we instantiate Transfinite Iris's liveness logic with ordinals as the source and SHL as the target, and obtain

Termination$_{\text{SHL}}$—a program logic for proving termination. **Termination**$_{\text{SHL}}$ generalizes what is known as time credits [8, 47] to *transfinite* time credits, which allow for termination arguments based on *dynamic* information learned during program execution (§5.1). Although transfinite time credits were already used in an earlier logic by da Rocha Pinto et al. [19], that logic is *not step-indexed*, thus restricting its applicability to first-order programs. In contrast, with **Termination**$_{\text{SHL}}$, step-indexing allows us to handle higher-order programs as well. We demonstrate the power of our technique by mechanizing various examples, including (in just 850 lines of Coq) a stronger version of the main theorem of Spies et al. [53]: termination of a linear language with asynchronous channels, modeling the core of promises in JavaScript (§5.2).

5.1 Time Credits

Compared to **Refinement**$_{\text{SHL}}$, we obtain slightly different logical connectives and proof rules in **Termination**$_{\text{SHL}}$ by picking ordinals as the source language. First, instead of the resources $\ell \mapsto_{\text{src}} v$ and $\text{src}(e)$, we have a connective $\$\alpha$ referring to the ordinal source. Second, instead of the source stepping rules (TPPureS and TPStoreS), we have:

$$\textsc{TSource}\ \frac{\langle \$\beta * P\rangle\ e\ \langle v.\ Q\rangle \qquad \beta < \alpha}{\{\$\alpha * \triangleright P\}\ e\ \{v.\ Q\}} \qquad \textsc{TSplit}\ \ \$(\alpha \oplus \beta) \Leftrightarrow \$\alpha * \$\beta$$

The rule TSource corresponds to steps in the source language, and the rule TSplit is a new rule, which we explain below.

With **Termination**$_{\text{SHL}}$, we can prove that an expression e terminates safely (*i.e.,* in a value), by proving $\text{terminates}(e) \triangleq \exists \alpha.\ \{\$\alpha\}\ e\ \{v.\text{True}\}$.

Theorem 5.1. *If* $\models \text{terminates}(e)$*, then* e *terminates safely.*

Proof. We use the existential property (for the quantification over α in the definition of terminates), and then use similar reasoning as in the adequacy proof of the termination-preserving refinement Theorem 4.3. □

The ordinal source $\$\alpha$ generalizes *time credits* [8, 47], which are traditionally used for proving complexity results with separation logic. Traditional time credits enable one to prove the safety property of *bounded termination.* That is, they enable one to verify that a program "terminates in n steps of computation", where the bound n has to be fixed up-front. What we obtain in this paper, by using ordinals, are *transfinite time credits.* Transfinite time credits go beyond bounded termination: they allow us to prove the liveness property of *termination*[4] for examples where it is non-trivial (if not impossible) to determine sufficient finite bounds.

[4]It is well-known that bounded termination is a safety property while termination (without a bound) is a liveness property [53]. Bounded termination can be falsified by exhibiting an execution which does not terminate within the given bound, a finite prefix. For termination, this is not the case: termination can only be falsified with an *infinite* execution.

To illustrate why this is useful, suppose we have a function f that returns a natural number, and we want to prove that $e_{\text{two}} \triangleq f() + f()$ terminates. If n_f is the maximum number of steps it takes to compute $f()$, then we know that it takes (roughly) $2 \cdot n_f$ steps for e_{two} to terminate. With both traditional and transfinite time credits, this amounts to proving $\{\$(2 \cdot n_f)\}\ e_{\text{two}}\ \{v.\ \text{True}\}$, where (ignoring stuttering) one time credit has to be spent for every step of e_{two}.

To prove this triple modularly, we make use of the distinguishing feature of time credits that makes them an ideal fit for separation logic: time credits can be split and combined. That is, we have (as an instance of TSplit for the finite case) $\$(n+m) \Leftrightarrow \$n * \$m$. We use this rule to factorize the termination proof of e_{two}: we first prove termination of f as $\{\$n_f\}\ f()\ \{m.\ m \in \mathbb{N}\}$, and then use this triple twice to prove the termination of e_{two}.

Transfinite credits. Now, consider a small generalization of the example, proving termination of:

$$\textbf{let}\ k = u\ ()\ \textbf{in let}\ a = \textbf{ref}(0)\ \textbf{in}$$
$$\textbf{for}\ i\ \textbf{in}\ 0, \ldots, k-1\ \textbf{do}\ a := !a + f\ ()$$

Here, u is a function returning a natural number k. We compute the sum of k-times executing f, and store that in a. To verify this program compositionally, we only assume that the function u, when given enough time credits n_u, returns a natural number, *i.e.,* $\{\$n_u\}\ u()\ \{m.\ m \in \mathbb{N}\}$.

In this setting, finite time credits are no longer sufficient. The number of steps it takes to execute the whole program depends on the output of $u\ ()$. The problem is that the number of credits required here depends on dynamic information—it depends on the execution of the program.

With *transfinite* time credits, *i.e.,* ordinals, we can statically abstract over this dynamic information. That is, we can show that $\$(\omega \oplus n_u)$ credits are enough to prove termination of the whole program. Given $\$(\omega \oplus n_u)$ time credits, we can spend $\$n_u$ on the execution of $u\ ()$ with TSplit. After obtaining the result k, we can use TSource to decrease the ω credits to $k \cdot n_f + 1$, which are sufficient for the remainder of the execution.

The addition operation $\alpha \oplus \beta$ in TSplit is Hessenberg addition [30]—a well-behaved, commutative notion of addition on ordinals. Commutativity is essential for us to be able to use ordinals as separation logic resources, since the latter are required to form a partial commutative monoid.

While this example is contrived, it highlights the core problem: in compositional termination proofs, there may not be enough information available to bound the length of the execution statically. With transfinite termination bounds, we can pick the termination bound *dynamically* based on information that is only learned *during* the execution (*e.g.,* the value of k in the above example).

5.2 Case Studies

Reentrant event loop. Using a reentrant event loop as an example, we illustrate how transfinite time credits interact with other features of step-indexing. An event loop consists of a stack of functions q, which can be extended with addtask $q\ f$, and can be executed through run q.

$$\begin{aligned}\mathsf{mkloop}() &\triangleq \mathsf{stack}()\\ \mathsf{addtask}\ q\ f &\triangleq \mathsf{push}\ q\ f\\ \mathsf{run}\ q &\triangleq \textbf{match}\ \mathsf{pop}\ q\ \textbf{with}\\ &\quad \mid \mathsf{None} \Rightarrow ()\\ &\quad \mid \mathsf{Some}\ f \Rightarrow f\ ();\ \mathsf{run}\ q\end{aligned}$$

Importantly, the event loop is *reentrant*: a function f that is added can extend the event loop with new functions when executed. As such, proving termination of run is subtle: there is no intrinsic termination measure since the size of the stack q can increase before it eventually decreases.

To ensure termination, the precondition for addtask consumes a constant c credits, which are logically transferred to the event loop stack. Then, to prove termination of run, we exploit the step-indexing underlying Transfinite Iris by using Löb induction, which does not require an intrinsic termination measure. Instead, with Löb induction, we obtain an assumption justifying the termination of a recursive execution of run guarded by a later modality ($\triangleright$). This later is then removed when using TSource, which requires spending a time credit. Here, we spend the time credits that were deposited by the calls to addtask. The intuition is that even though extra jobs may be added while run executes, only a bounded number can ultimately be added because the total number of credits available is an ordinal.

Logical relation for termination. Transfinite time credits allow us to obtain and mechanize the main result of Spies et al. [53] in **Termination**$_{\mathsf{SHL}}$: termination of a linear language with asynchronous channels, modeling the core of promises in JavaScript. For their termination proof, they introduce a transfinitely step-indexed logical relation which, internally, uses a bespoke form of transfinite time credits and transfinite step-indexing. In **Termination**$_{\mathsf{SHL}}$ we can, using our general form of transfinite time credits, simplify their logical relation (which they define and use in a 40-page appendix), and mechanize their result in 500 lines of Coq. With an additional 350 lines of Coq, we generalize their result to a language with impredicative polymorphism as well.

Our generalization to impredicative polymorphism relies crucially on (Transfinite) Iris's *impredicative invariants* [35, 54]—*i.e.,* invariants on shared state which can mention arbitrary logical assertions (including other invariant assertions). To explain in a bit more detail how we leverage impredicative invariants and, moreover, why it would be non-trivial to generalize the logical relation of Spies et al. [53] without using Transfinite Iris, let us briefly review how logical relations are typically formalized in Iris. A type τ is interpreted as an Iris predicate, *i.e.,* an element of $\mathit{Val} \to \mathit{iProp}$. For example, the ML-style reference type $\mathbf{ref}(\tau)$ can be interpreted as a predicate on memory locations ℓ that asserts that there is an *invariant* on ℓ, which in turn asserts that (at all times) the value v stored at ℓ must satisfy the predicate associated with τ. The key point then is that, because of polymorphic types, τ could for instance be a type variable, modeled as an *arbitrary* unknown Iris predicate, and thus it is crucial that the invariant in the model of $\mathbf{ref}(\tau)$ is impredicative.

To the best of our knowledge, there is no way to avoid impredicative invariants here—this is related to the well-known "type-world circularity", see, *e.g.,* Ahmed's earlier work on concrete step-indexed models of such type systems [1]. Hence, if one tries to generalize the logical relation of Spies et al. [53] without using Transfinite Iris, one has to solve the type-world circularity "by hand," meaning one essentially has to solve the kind of recursive domain equation underlying Transfinite Iris (see §6.2). Here, we instead leveraged that Transfinite Iris has impredicative invariants and then defined the interpretation of types along the lines of what is sketched above. For more details on the polymorphic extension, we refer to the supplementary material [52].

6 Foundations of Transfinite Iris

Thus far we have worked with Transfinite Iris informally—outlining its structure (§3), intuitively explaining the rules of its program logics (§4.1 and §5.1), and illustrating their use with examples (§4.3 and §5.2). To ensure that all of this is meaningful, we now turn to the foundations of Transfinite Iris (see Figure 1): the *core logic*.

As with Iris, the core of Transfinite Iris factors into two separate and largely orthogonal parts: the part dealing with step-indexing, and the part dealing with ownership and resources. For now, we set aside the latter and focus on the step-indexing part—we focus on *step-indexed logic*.

6.1 Transfinitely Step-Indexed Logic

We have already encountered step-indexed logic as the logical approach to step-indexing in §2.4. In contrast to the logic in §2.4 (*i.e.,* the one underlying Iris), we want the one of Transfinite Iris to validate the existential property. To this end, we switch from indexing propositions by natural numbers to indexing them by ordinals.

Ordinals. We write *Ord* for the type of ordinals. As with natural numbers, there is a zero ordinal 0 and, for each ordinal α, a strictly larger successor ordinal s $\alpha > \alpha$. Going beyond natural numbers: for each small family of ordinals $f : X \to \mathit{Ord}$, there exists an ordinal $\sup_{x:X} f\ x$ which is larger than $f\ x$ for all $x \in X$.[5] For instance, we obtain the first

[5]The supremum sup can be taken as long as X is a small type; think of X as being a set, compared to *Ord* being a proper class.

infinite ordinal ω as $\sup_{n:\mathbb{N}} \mathsf{s}^n\, 0$. The additional supremum operation is the key to obtaining the existential property.

Step-indexed propositions. In this work, step-indexed propositions are families of ordinary propositions, indexed by ordinals, with the essential "down-closure" property that truth at one step-index implies truth at all lower step-indices:

Definition 6.1. The type of *step-indexed propositions SProp* consists of families $P : \mathit{Ord} \to \mathit{Prop}$ which are down-closed, meaning if $P\,\alpha$ and $\beta \le \alpha$, then $P\,\beta$.

Given two step-indexed propositions $P, Q : \mathit{SProp}$, we say that $P \models Q$ iff $P\,\alpha$ implies $Q\,\alpha$ for all $\alpha : \mathit{Ord}$.

The familiar logical connectives (conjunction, disjunction, implication, etc.) lift to step-indexed propositions and satisfy the standard rules of intuitionistic higher-order logic. For instance, given $\Phi : X \to \mathit{SProp}$, the step-indexed proposition $(\exists x : X.\, \Phi\, x)\,\alpha \triangleq \exists x : X.\, \Phi\, x\,\alpha$ satisfies the standard logical rules for existential quantification.

In addition to the standard connectives, we equip step-indexed propositions with a later modality $\triangleright P$:

$$(\triangleright P)\,\alpha \triangleq \forall \beta < \alpha.\, P\,\beta$$

This definition restricts to the usual model of $\triangleright P$ (see §2.4) when applied to natural numbers. It also validates the central rules of step-indexed logics, such as Löb induction, as well as $P \models \triangleright P$ (the latter following directly from down-closure).

The existential property. With step-indexed propositions in hand, we proceed to prove the existential property.

Theorem 6.2 (Existential Property)**.** *Fix a small type X. If $\models \exists x : X.\, \Phi\, x$, then $\models \Phi\, x$ for some $x : X$.*

Proof. We proceed by contradiction and assume that $\models \Phi\, x$ is false for each $x : X$. Therefore, for each x there is $\alpha_x : \mathit{Ord}$ such that $\Phi\, x\, \alpha_x$ is false. From $\models \exists x.\, \Phi\, x$ we know that for each β there is some x such that $\Phi\, x\, \beta$. To obtain a contradiction, we will construct an ordinal β, where $\Phi\, x\, \beta$ is false for each x. First, we observe that $\Phi\, x$ is down-closed, so $\Phi\, x\, \alpha$ is false for any $\alpha \ge \alpha_x$. We now define $\beta \triangleq \sup_{x:X} \alpha_x$. For each x, we have $\beta \ge \alpha_x$ by construction, hence $\Phi\, x\, \beta$ is false. Hence, we have reached a contradiction. □

The proof rests crucially on *Ord* containing suprema of small families (*i.e.,* families over small types). In the mechanization [52], where our definition of ordinals is based on the work of Kirst and Smolka [39], the notion of "small types" is made precise by Coq's universe hierarchy; *SProp* satisfies the existential property for any type belonging to a universe smaller than the universe *Ord* occupies.

6.2 Modeling Transfinite Iris

So far, we have only discussed step-indexed propositions and left out how resources and ownership fit into the picture. We fill this gap by defining *iProp*, the type of propositions in the core logic (and by extension the program logics) of Transfinite Iris. *iProp* is the type of monotone, step-indexed predicates over resources. More precisely, *iProp* is the solution to the recursive domain equation:

$$\mathit{iProp} \approx F(\mathit{iProp}) \xrightarrow{\text{mon}} \mathit{SProp}$$

where F returns the resources used in Transfinite Iris. The equation is recursive because resources can depend on *iProp*, which in turn depends on the resources, etc. To name two prime examples, such circular dependencies arise for both impredicative invariants [54] and higher-order ghost state [35].

Unfortunately, this domain equation does not have a solution for arbitrary types. In order to solve such equations, we use *step-indexed types* in Transfinite Iris.

Step-indexed types. Step-indexed types already appear in Iris for the same reasons, but Transfinite Iris alters their form by (again) drawing indices from ordinals rather than from natural numbers. Specifically, we define a step-indexed type to be an *ordered family of equivalences* (OFE): a pair of a type X and a family of equivalence relations $(\overset{\alpha}{=})$. These equivalence relations must become increasingly coarse as α decreases (i.e., if $x \overset{\alpha}{=} y$ and $\beta \le \alpha$, then $x \overset{\beta}{=} y$). For instance, *SProp* is an OFE with $P \overset{\alpha}{=} Q \triangleq \forall \beta \le \alpha.\, P\,\beta \mathit{\ iff\ } Q\,\beta$. Recursive definitions can be constructed directly in *SProp* using a variant of Banach's fixed-point theorem [28]:

Theorem 6.3. *Let $f : \mathit{SProp} \to \mathit{SProp}$ such that $f\,P \overset{\alpha}{=} f\,Q$ if $\forall \beta < \alpha.\, P \overset{\beta}{=} Q$. Then there exists an R such that $R = f\,R$.*

This theorem can be generalized to arbitrary *COFEs* (OFEs which, like *SProp*, enjoy a certain completeness property).

Solving the recursive domain equation. We now return to the recursive domain equation for *iProp*. It is well-known that domain equations can be solved over *finite* step-indexed types [4]. However, while transfinite variants of step-indexed types have been considered before, it was unclear whether the construction of solutions to domain equations [11, 28] could be adapted to the transfinite setting. In order to complete the model, we have therefore defined a novel construction for solving domain equations, extending the existence of solutions to the transfinite case. For further details, see the supplementary material [52].

With *iProp* as the solution to the above domain equation, it is straightforward to redefine the remaining connectives of Iris's base logic handling resources and ownership. We obtain the following theorem.

Theorem 6.4. *Transfinite Iris is consistent and enjoys the existential property.*

Note that the construction of the step-indexed model in Coq relies on extensionality axioms (*i.e.,* propositional extensionality and functional extensionality) and classical axioms (*i.e.,* the axiom of choice and excluded middle). The soundness of Coq under this combination of axioms is a consequence of the model of CiC in ZFC [62].

7 The Loss of Commuting Rules

While our use of transfinite step-indexing validates the existential property, there is a price to pay, namely that it *invalidates* the following commuting rule from Iris:

$$\textsc{LaterExists}\quad \frac{X \text{ inhabited}}{\triangleright(\exists x : X.\, \Phi(x)) \models \exists x : X.\, \triangleright \Phi(x)}$$

This is not by accident. Regardless of the step-indexing technique used, the existential property is fundamentally incompatible with the rule LaterExists. This incompatibility is witnessed by the following theorem:

Theorem 7.1. *There exists no consistent logic with a sound later modality (*i.e., $\models \triangleright P$ *implies* $\models P$*), Löb induction (*i.e., $(\triangleright P \Rightarrow P) \models P$*), and the commuting rule LaterExists, which also enjoys the existential property.*

Proof. By way of contradiction, assume there is a logic satisfying these properties. As we will prove below, the proposition $\exists n : \mathbb{N}.\ \triangleright^n \bot$ is provable in such a logic. Using the existential property, we then obtain an n at the meta-level such that $\models \triangleright^n \bot$. By soundness of the later modality, this entails $\models \bot$, which contradicts consistency.

We prove $\models (\exists n : \mathbb{N}.\ \triangleright^n \bot)$ by Löb induction, so it suffices to show $(\triangleright \exists n : \mathbb{N}.\ \triangleright^n \bot) \models (\exists n : \mathbb{N}.\ \triangleright^n \bot)$. By the commuting rule LaterExists, this is reduced to:

$$(\exists n : \mathbb{N}.\ \triangleright \triangleright^n \bot) \models (\exists n : \mathbb{N}.\ \triangleright^n \bot)$$

We eliminate the existential quantifier in the assumption to obtain a witness n, and then instantiate the existential quantifier in the conclusion by picking $n + 1$. The remaining claim $(\triangleright \triangleright^n \bot) \models (\triangleright^{n+1} \bot)$ is immediate. □

Given that LaterExists does not hold, it is not surprising that we also lose the rule $\triangleright(P * Q) \models (\triangleright P) * (\triangleright Q)$, since in the model of the core logic, the separating conjunction connective $P * Q$ is defined by *existentially* quantifying over the split of resources between P and Q.

Avoiding the commuting rules. The absence of the commuting rules is of course a point of concern because it means that there are Iris proofs which do not hold verbatim in Transfinite Iris. That said, the fact that some existing Iris proofs may no longer be valid in Transfinite Iris does not mean that they cannot be fixed!

In an initial experiment, we tweaked the (safety) program logic of Iris to avoid the use of the commuting rules—obtaining the safety program logic of Transfinite Iris. Using this experiment, we were able to adapt various existing Iris proofs with only minor changes (for details see [52]). In particular, we were still able to verify the barrier of Jung et al. [35]: the flagship example for the introduction of higher-order ghost state in Iris. Besides these safety examples, our liveness examples from §4.3 and §5.2 avoid the broken commuting rules as well.

Nevertheless, there are Iris developments which use the commuting rules for proofs where our tweaks do not apply, *e.g.,* the type system of RustBelt [33]. Recovering every Iris development is beyond the scope of this paper and left to future work.

8 Related Work

We compare with (1) approaches approximating liveness reasoning in step-indexed logics, (2) existing separation logics for proving liveness properties, and (3) approaches using transfinitely step-indexed models.

Approximated liveness in step-indexed logics. In the absence of the existential property, liveness properties have only been approximated by safety properties in step-indexed logics. That is, all of the approaches below [21, 22, 27, 47, 58] have imposed restrictions on the liveness properties which effectively render them into safety properties (e.g., termination, but with a fixed upper bound on the number of steps).

Mével et al. [47] extend Iris with time credits to prove complexity results, *i.e.,* termination with a fixed upper bound.

Dockins and Hobor [21, 22], similarly, introduce a logic for proving termination with a fixed upper bound.

Tassarotti et al. [58] extend Iris with support for proving concurrent termination-preserving refinements. Because they use finite instead of transfinite step-indexing, they obtain the existential property only for quantification over finite sets. Compared to our work, this means their termination-preserving simulation is weaker—the source is restricted to bounded non-determinism, and can only stutter for a fixed number of times between target steps. With these restrictions, their termination-preserving simulation becomes a safety property, because non-simulation can be determined by examining a finite prefix of execution. These restrictions are too strong for examples like `memo_rec`, where an unbounded number of stutters is required because the number of steps needed to find a cached result in the lookup table can grow arbitrarily. In contrast to our work, Tassarotti et al. do support concurrency.

Frumin et al. [27] define a logic in Iris to prove the security property *timing-sensitive non-interference*. While this property ensures termination preservation, it is expressed as a lock-step program equivalence, meaning it imposes even stronger restrictions than the work of Tassarotti et al. [58].

Liveness in separation logics. In the literature, a number of separation logics for liveness reasoning have been introduced [17, 19, 23, 45, 46, 63]: Liang and Feng [45, 46] develop the program logic LiLi, a "rely-guarantee style program logic for verifying linearizability and progress together for concurrent objects under fair scheduling". da Rocha Pinto et al. [19] extend the concurrent separation logic TaDA [18] with ordinal time credits to prove program termination. In ongoing work, D'Osualdo et al. [23] go further and target more general liveness properties such as "always-eventually"

properties. Yoshida et al. [63] and Charguéraud [17] introduce program logics capable of handling liveness reasoning, even in higher-order stateful settings. In particular, Yoshida et al. [63] verify a termination-preserving refinement of a memoized factorial function.

The fundamental difference to our work is that all of these logics are *not* step-indexed. In this paper, we have focused on enabling liveness in a *step-indexed setting*, allowing us to use features like Löb induction, guarded recursion, and impredicative invariants. It was precisely the combination of these features that allowed us to prove a generic specification for `memo_rec` and then instantiate it for multiple clients. To the best of our knowledge, verifying `memo_rec` generically is not possible in the above logics.

Some of these logics [19, 23, 45, 46] are, however, capable of proving liveness properties of *concurrent* programs—a point of future work for Transfinite Iris. In this paper, to focus on dealing with the "existential dilemma" of step-indexed separation logic, we have explored liveness reasoning in the sequential setting first. That said, Transfinite Iris is compatible with concurrency—our safety program logic includes several representative case studies of concurrent *safety* reasoning, which we have ported from Iris to Transfinite Iris.

Transfinite step-indexing. In the literature, transfinite step-indexing has already been used for two purposes: modeling a separation logic for safety reasoning [56] and constructing logical relations for a single language [9–11, 13, 53, 57].

First, we discuss the relationship to the only separation logic using transfinite step-indexing—SLR by Svendsen et al. [56], a logic for the promising weak memory model [38]. SLR is restricted to proving *safety properties*. In their work, transfinite indices up to ω^2 are used to handle hypothetical steps of computation called certification steps, which occur between actual steps in the promising model to justify speculative weak memory behavior. However, since their indices only go up to ω^2, they do not suffice to obtain the existential property for quantification over infinite sets. In future work, it would be interesting to explore whether SLR can be encoded into Transfinite Iris.

Second, we discuss the relationship to logical relations defined using transfinite step-indexing. Birkedal et al. [10] use step-indexing up to ω_1 (the least uncountable ordinal) to give a logical-relations model for reasoning about must-contextual equivalence for a language with countable nondeterminism. (The inductively-defined must-convergence predicate for the language with countable nondeterminism has ω_1 as closure ordinal.) Bizjak et al. [13] show how to define the logical-relations model for countable nondeterminism in a step-indexed logic, namely in the internal logic of the topos of sheaves over ω_1. We conjecture that their step-indexed logic enjoys the existential property, restricted to countable types, and that they use it implicitly for their adequacy result (see Lemma 4 in [13]).

Svendsen et al. [57] use step-indexing up to ω^2 to decouple steps of computation from *logical steps* (such as unfolding an invariant)—*i.e.,* to allow multiple logical steps per physical step of computation. They do not address liveness reasoning. In their work, Svendsen et al. already solve a recursive domain equation (RDE) in COFEs for the ω^2 case. We have extended this result to arbitrary (uncountable) ordinals, which are needed for the existential property.

Birkedal et al. [11] solve RDEs in the more general category of *sheaves*. For Transfinite Iris, we considered simply working with solutions to RDEs in sheaves, but decided it was impractical as it would have led to problems with mechanization of Transfinite Iris—a central concern for its deployment in practice. In particular, were we to use sheaves, functions in Transfinite Iris could no longer be encoded as Coq functions with a property of non-expansiveness, as they are in the original Iris, but would instead have to be represented as (transfinite) sequences of functions. In the present work, we thus instead solve a RDE in the category of (transfinite) *COFEs*, a subcategory of sheaves. (Note that this is a novel result in relation to Birkedal et al. [11], because their RDE solutions are not guaranteed to be COFEs.) In contrast to sheaves, COFEs are well suited to mechanization (in Coq).

Spies et al. [53] use transfinite step-indexing up to ω^ω to prove termination of a linear language with higher-order channels. As explained in §5.2, the present work subsumes, extends, and mechanizes their work, in the process reducing the size of their proof significantly.

Bahr et al. [9] define a statically-typed, pure functional language, and use transfinite step-indexing up to $\omega \cdot 2$ to show that well-typed programs enjoy a certain liveness property. Aside from the use of transfinite step-indexing, this work is quite different from ours. It focuses on type systems that ensure liveness properties, rather than techniques for verification of liveness properties in a general-purpose language with features like general recursion and higher-order state.

Acknowledgments

We wish to thank Ralf Jung and Amin Timany for feedback and helpful discussions. This research was supported in part by a European Research Council (ERC) Consolidator Grant for the project "RustBelt", funded under the European Union's Horizon 2020 Framework Programme (grant agreement no. 683289), in part by the Saarbrücken Graduate School of Computer Science, in part by the International Max Planck Research School on Trustworthy Computing (IMPRS-TRUST), in part by a Villum Investigator grant (no. 25804), Center for Basic Research in Program Verification (CPV), from the VILLUM Foundation, in part by the Dutch Research Council (NWO), project 016.Veni.192.259, and in part by generous gifts from Oracle Labs and Google.

References

[1] Amal Ahmed. 2004. *Semantics of types for mutable state.* Ph.D. Dissertation. Princeton University.

[2] Amal Ahmed, Andrew W. Appel, and Roberto Virga. 2002. A stratified semantics of general references. In *LICS*. 75–86. https://doi.org/10.1109/LICS.2002.1029818

[3] Amal Ahmed, Derek Dreyer, and Andreas Rossberg. 2009. State-dependent representation independence. In *POPL*. 340–353. https://doi.org/10.1145/1480881.1480925

[4] Pierre America and Jan Rutten. 1989. Solving reflexive domain equations in a category of complete metric spaces. *JCSS* 39, 3 (1989), 343–375. https://doi.org/10.1016/0022-0000(89)90027-5

[5] Andrew W. Appel, Robert Dockins, Aquinas Hobor, Lennart Beringer, Josiah Dodds, Gordon Stewart, Sandrine Blazy, and Xavier Leroy. 2014. *Program Logics for Certified Compilers.* Cambridge University Press.

[6] Andrew W. Appel and David A. McAllester. 2001. An indexed model of recursive types for foundational proof-carrying code. *TOPLAS* 23, 5 (2001), 657–683. https://doi.org/10.1145/504709.504712

[7] Andrew W. Appel, Paul-André Melliès, Christopher D. Richards, and Jérôme Vouillon. 2007. A very modal model of a modern, major, general type system. In *POPL*. 109–122. https://doi.org/10.1145/1190216.1190235

[8] Robert Atkey. 2010. Amortised resource analysis with separation logic. In *ESOP (LNCS, Vol. 6012)*. 85–103. https://doi.org/10.1007/978-3-642-11957-6_6

[9] Patrick Bahr, Christian Uldal Graulund, and Rasmus Ejlers Møgelberg. 2021. Diamonds are not forever: liveness in reactive programming with guarded recursion. *PACMPL* 5, POPL (2021), 1–28. https://doi.org/10.1145/3434283

[10] Lars Birkedal, Ales Bizjak, and Jan Schwinghammer. 2013. Step-indexed relational reasoning for countable nondeterminism. *LMCS* 9, 4 (2013). https://doi.org/10.2168/LMCS-9(4:4)2013

[11] Lars Birkedal, Rasmus Ejlers Møgelberg, Jan Schwinghammer, and Kristian Støvring. 2011. First steps in synthetic guarded domain theory: Step-indexing in the topos of trees. In *LICS*. 55–64. https://doi.org/10.1109/LICS.2011.16

[12] Lars Birkedal, Bernhard Reus, Jan Schwinghammer, Kristian Støvring, Jacob Thamsborg, and Hongseok Yang. 2011. Step-indexed Kripke models over recursive worlds. In *POPL*. 119–132. https://doi.org/10.1145/1926385.1926401

[13] Ales Bizjak, Lars Birkedal, and Marino Miculan. 2014. A model of countable nondeterminism in guarded type theory. In *RTA-TLCA (LNCS, Vol. 8560)*. 108–123. https://doi.org/10.1007/978-3-319-08918-8_8

[14] Michael C. Browne, Edmund M. Clarke, and Orna Grumberg. 1988. Characterizing finite Kripke structures in propositional temporal logic. *TCS* 59 (1988), 115–131. https://doi.org/10.1016/0304-3975(88)90098-9

[15] Qinxiang Cao, Lennart Beringer, Samuel Gruetter, Josiah Dodds, and Andrew W. Appel. 2018. VST-Floyd: A separation logic tool to verify correctness of C programs. *JAR* 61, 1-4 (2018), 367–422. https://doi.org/10.1007/s10817-018-9457-5

[16] Tej Chajed, Joseph Tassarotti, M. Frans Kaashoek, and Nickolai Zeldovich. 2019. Verifying concurrent, crash-safe systems with Perennial. In *SOSP*. 243–258. https://doi.org/10.1145/3341301.3359632

[17] Arthur Charguéraud. 2011. Characteristic formulae for the verification of imperative programs. In *ICFP*. 418–430. https://doi.org/10.1145/2034773.2034828

[18] Pedro da Rocha Pinto, Thomas Dinsdale-Young, and Philippa Gardner. 2014. TaDA: A logic for time and data abstraction. In *ECOOP (LNCS, Vol. 8586)*. 207–231. https://doi.org/10.1007/978-3-662-44202-9_9

[19] Pedro da Rocha Pinto, Thomas Dinsdale-Young, Philippa Gardner, and Julian Sutherland. 2016. Modular termination verification for non-blocking concurrency. In *ESOP (LNCS, Vol. 9632)*. 176–201. https://doi.org/10.1007/978-3-662-49498-1_8

[20] Hoang-Hai Dang, Jacques-Henri Jourdan, Jan-Oliver Kaiser, and Derek Dreyer. 2020. RustBelt meets relaxed memory. *PACMPL* 4, POPL (2020), 34:1–34:29. https://doi.org/10.1145/3371102

[21] Robert Dockins and Aquinas Hobor. 2010. A theory of termination via indirection. In *Modelling, Controlling and Reasoning About State (Dagstuhl Seminar Proceedings, Vol. 10351)*. http://drops.dagstuhl.de/opus/volltexte/2010/2805/

[22] Robert Dockins and Aquinas Hobor. 2012. Time bounds for general function pointers. 286 (2012), 139–155. https://doi.org/10.1016/j.entcs.2012.08.010

[23] Emanuele D'Osualdo, Azadeh Farzan, Philippa Gardner, and Julian Sutherland. 2019. TaDA Live: Compositional reasoning for termination of fine-grained concurrent programs. http://arxiv.org/abs/1901.05750

[24] Derek Dreyer, Amal Ahmed, and Lars Birkedal. 2011. Logical Step-Indexed Logical Relations. *LMCS* 7, 2:16 (June 2011), 1–37. https://doi.org/10.2168/LMCS-7(2:16)2011

[25] Derek Dreyer, Georg Neis, Andreas Rossberg, and Lars Birkedal. 2010. A relational modal logic for higher-order stateful ADTs. In *POPL*. 185–198. https://doi.org/10.1145/1706299.1706323

[26] Dan Frumin, Robbert Krebbers, and Lars Birkedal. 2018. ReLoC: A mechanised relational logic for fine-grained concurrency. In *LICS*. 442–451. https://doi.org/10.1145/3209108.3209174

[27] Dan Frumin, Robbert Krebbers, and Lars Birkedal. 2021. Compositional non-interference for fine-grained concurrent programs. To appear in S&P'21.

[28] Pietro Di Gianantonio and Marino Miculan. 2004. Unifying recursive and co-recursive definitions in sheaf categories. In *FOSSACS (LNCS, Vol. 2987)*. 136–150. https://doi.org/10.1007/978-3-540-24727-2_11

[29] Paolo G. Giarrusso, Léo Stefanesco, Amin Timany, Lars Birkedal, and Robbert Krebbers. 2020. Scala step-by-step: Soundness for DOT with step-indexed logical relations in Iris. *PACMPL* 4, ICFP (2020), 114:1–114:29. https://doi.org/10.1145/3408996

[30] Gerhard Hessenberg. 1906. *Grundbegriffe der Mengenlehre.* Vol. 1. Vandenhoeck & Ruprecht.

[31] Aquinas Hobor, Andrew W. Appel, and Francesco Zappa Nardelli. 2008. Oracle semantics for concurrent separation logic. In *ESOP (LNCS, Vol. 4960)*. 353–367. https://doi.org/10.1007/978-3-540-78739-6_27

[32] Iris project. 2021. A higher-order concurrent separation logic framework implemented and verified in the proof assistant Coq. https://iris-project.org/

[33] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2018. RustBelt: Securing the foundations of the Rust programming language. *PACMPL* 2, POPL (2018), 66:1–66:34. https://doi.org/10.1145/3158154

[34] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2021. Safe systems programming in Rust. *CACM* 64, 4 (2021), 144–152. https://doi.org/10.1145/3418295

[35] Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. 2016. Higher-order ghost state. In *ICFP*. 256–269. https://doi.org/10.1145/2951913.2951943

[36] 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. *JFP* 28 (2018), e20. https://doi.org/10.1017/S0956796818000151

[37] 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 *POPL*. 637–650. https://doi.org/10.1145/2676726.2676980

[38] Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer. 2017. A promising semantics for relaxed-memory concurrency. In *POPL*. 175–189. https://doi.org/10.1145/3009837.3009850

[39] Dominik Kirst and Gert Smolka. 2018. Large Model Constructions for Second-Order ZF in Dependent Type Theory. In *CPP*. 228–239. https://doi.org/10.1145/3167095

[40] Robbert Krebbers, Jacques-Henri Jourdan, Ralf Jung, Joseph Tassarotti, Jan-Oliver Kaiser, Amin Timany, Arthur Charguéraud, and Derek Dreyer. 2018. MoSeL: A general, extensible modal framework for interactive proofs in separation logic. *PACMPL* 2, ICFP (2018), 77:1–77:30. https://doi.org/10.1145/3236772

[41] Robbert Krebbers, Ralf Jung, Ales Bizjak, Jacques-Henri Jourdan, Derek Dreyer, and Lars Birkedal. 2017. The essence of higher-order concurrent separation logic. In *ESOP (LNCS, Vol. 10201)*. 696–723. https://doi.org/10.1007/978-3-662-54434-1_26

[42] Robbert Krebbers, Amin Timany, and Lars Birkedal. 2017. Interactive proofs in higher-order concurrent separation logic. In *POPL*. 205–217. https://doi.org/10.1145/3093333.3009855

[43] Morten Krogh-Jespersen, Kasper Svendsen, and Lars Birkedal. 2017. A relational model of types-and-effects in higher-order concurrent separation logic. In *POPL*. 218–231. https://doi.org/10.1145/3093333.3009877

[44] Vladimir Iosifovich Levenshtein. 1965. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady* 10 (1965), 707–710.

[45] Hongjin Liang and Xinyu Feng. 2016. A program logic for concurrent objects under fair scheduling. In *POPL*. 385–399. https://doi.org/10.1145/2837614.2837635

[46] Hongjin Liang and Xinyu Feng. 2018. Progress of concurrent objects with partial methods. *PACMPL* 2, POPL (2018), 20:1–20:31. https://doi.org/10.1145/3158108

[47] Glen Mével, Jacques-Henri Jourdan, and François Pottier. 2019. Time credits and time receipts in Iris. In *ESOP (LNCS, Vol. 11423)*. 3–29. https://doi.org/10.1007/978-3-030-17184-1_1

[48] Glen Mével, Jacques-Henri Jourdan, and François Pottier. 2020. Cosmo: A concurrent separation logic for Multicore OCaml. *PACMPL* 4, ICFP (2020), 96:1–96:29. https://doi.org/10.1145/3408978

[49] Hiroshi Nakano. 2000. A modality for recursion. In *LICS*. 255–266. https://doi.org/10.1109/LICS.2000.855774

[50] Peter W. O'Hearn and David J. Pym. 1999. The logic of bunched implications. *Bulletin of Symbolic Logic* 5, 2 (1999), 215–244. https://doi.org/10.2307/421090

[51] Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. 2001. Local reasoning about programs that alter data structures. In *CSL (LNCS, Vol. 2142)*. 1–19. https://doi.org/10.1007/3-540-44802-0_1

[52] Simon Spies, Lennard Gäher, Daniel Gratzer, Joseph Tassarotti, Robbert Krebbers, Derek Dreyer, and Lars Birkedal. 2021. Transfinite Iris appendix and Coq development. https://iris-project.org/transfinite-iris/

[53] Simon Spies, Neel Krishnaswami, and Derek Dreyer. 2021. Transfinite step-indexing for termination. *PACMPL* 5, POPL (2021), 1–29. https://doi.org/10.1145/3434294

[54] Kasper Svendsen and Lars Birkedal. 2014. Impredicative concurrent abstract predicates. In *ESOP (LNCS, Vol. 8410)*. 149–168. https://doi.org/10.1007/978-3-642-54833-8_9

[55] Kasper Svendsen, Lars Birkedal, and Matthew J. Parkinson. 2013. Modular reasoning about separation of concurrent data structures. In *ESOP (LNCS, Vol. 7792)*. 169–188. https://doi.org/10.1007/978-3-642-37036-6_11

[56] Kasper Svendsen, Jean Pichon-Pharabod, Marko Doko, Ori Lahav, and Viktor Vafeiadis. 2018. A separation logic for a promising semantics. In *ESOP (LNCS, Vol. 10801)*. 357–384. https://doi.org/10.1007/978-3-319-89884-1_13

[57] Kasper Svendsen, Filip Sieczkowski, and Lars Birkedal. 2016. Transfinite step-indexing: Decoupling concrete and logical steps. In *ESOP (LNCS, Vol. 9632)*. 727–751. https://doi.org/10.1007/978-3-662-49498-1_28

[58] Joseph Tassarotti, Ralf Jung, and Robert Harper. 2017. A higher-order logic for concurrent termination-preserving refinement. In *ESOP (LNCS, Vol. 10201)*. 909–936. https://doi.org/10.1007/978-3-662-54434-1_34

[59] 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. *PACMPL* 2, POPL (2018), 64:1–64:28. https://doi.org/10.1145/3158152

[60] Aaron Turon, Derek Dreyer, and Lars Birkedal. 2013. Unifying refinement and Hoare-style reasoning in a logic for higher-order concurrency. In *ICFP*. 377–390. https://doi.org/10.1145/2500365.2500600

[61] Aaron Joseph Turon, Jacob Thamsborg, Amal Ahmed, Lars Birkedal, and Derek Dreyer. 2013. Logical relations for fine-grained concurrency. In *POPL*. 343–356. https://doi.org/10.1145/2429069.2429111

[62] Benjamin Werner. 1997. Sets in types, types in sets. In *TACS (LNCS, Vol. 1281)*. 530–346. https://doi.org/10.1007/BFb0014566

[63] Nobuko Yoshida, Kohei Honda, and Martin Berger. 2007. Logical reasoning for higher-order functions with local state. In *FOSSACS (LNCS, Vol. 4423)*. 361–377. https://doi.org/10.1007/978-3-540-71389-0_26