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

Simon Spies
MPI-SWS

Lennard Gäher
Saarland University

Daniel Gratzer
Aarhus University

Joseph Tassarotti
Boston College

Robbert Krebbers
Radboud University Nijmegen

Derek Dreyer
MPI-SWS

Lars Birkedal
Aarhus University

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.

1 Introduction

In the past decade, *separation logics* [47] 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, 13, 29], HOCAP [51], iCAP [50], and Iris [33–35, 38]. 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 [14] to OCaml [45] to Rust [18, 31, 32] to Scala [27].

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]:

```
let memo_rec t =
  let m = Hashtbl.create 0 in
  let rec g x =
    match Hashtbl.find_opt m x with
     Some y -> y
     None   -> let y = t g x in
               Hashtbl.add m x y; y
  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.

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

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` terminates, then so does `f_memo v`. In this case, we say that `f_memo` is a *termination-preserving refinement* of `f`. To prove this, 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 [19, 20] and Mével et al. [44] have shown how to prove termination if the user gives explicit time complexity bounds. Tassarotti et al. [53] 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. [53]'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 is missing is the *existential property* [36] (sometimes called the *existence property*):

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

(Here, $\vdash P$ denotes provability 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 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.

Unfortunately, the existential property is fundamentally incompatible with how step-indexed logics have thus far been modeled. In particular, existing step-indexed logics model propositions as predicates over a "step-index" in the form of a *natural number* n. Under this standard step-indexed 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 [33–35, 38], 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.

Transfinite Iris inherits Iris's support for safety reasoning about higher-order, concurrent programs. However, in order to get to the heart of our core "existential dilemma"—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*. Supporting step-indexed liveness reasoning for concurrent programs remains an important direction for future work.

While the idea of transfinite step-indexing is not new (see § 6 for a discussion of related work), Transfinite Iris is to our knowledge the first *program logic* to be founded on a transfinitely step-indexed model, and the first step-indexed logic to support 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 from the perspective of supporting 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 (§ 3) and termination (§ 4).
- We demonstrate the power of Transfinite Iris on a range of interesting examples, including the `memo_rec` example presented above (§ 3.4 and § 4.2).
- Establishing the soundness of Transfinite Iris required us to solve a new and challenging recursive domain equation for modeling the type of Transfinite Iris propositions. Due to space constraints, we cannot go into detail about this construction, but we describe it at a high level (§ 5). See the appendix [48] for details.
- All examples in § 3.4 and § 4.2, as well as the soundness of Transfinite Iris, are fully mechanized in Coq using the Iris Proof Mode [37, 39]. See [48] for the Coq proofs.

2 Key Idea: The Existential Property

In this section, we explain how the *existential property* is the key to enabling step-indexed logics to support proofs of termination and termination-preserving refinement. We formally define refinements (§ 2.1) and how they are proven using simulations (§ 2.2). We show why step-indexing is useful for defining such simulations, but why it falls short in the case of termination-preserving refinements (§ 2.3). We then 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.

2.2 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 t and source s expression, 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).

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

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, 22–24, 39, 40, 54–56]. 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*:

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

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 [11] (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 $t = t_0 \rightsquigarrow_{\mathrm{tgt}} \cdots \rightsquigarrow_{\mathrm{src}} t_n = b$ 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 $s = s_0 \rightsquigarrow_{\mathrm{src}} \cdots \rightsquigarrow_{\mathrm{src}} s_n$ 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, and see where it goes wrong:

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

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, 22] 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, 46]. We consider a version of our simulation relation that is defined using logical step-indexing:

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

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 ($\triangleright$) 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. The notion of provability $\vdash P$ of a step-indexed logic is the limit, *i.e.,* $\vdash P \triangleq \forall i.\, i \in P$.

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 the same definition as the explicit step-indexed relation $\preceq_i$ we have seen in § 2.3. If we no longer think of the step-indexing in terms of propositions modeled as sets of natural numbers, we may wonder, what property are step-indexed logics missing that prohibit us from proving termination-preserving refinements? The answer to this question is the *existential property*:

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

If we assume we work in a step-indexed logic that does enjoy the existential property, then we can, in fact, prove:

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

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

2.6 Termination

We show that the simulation relation ($\preceq_*$) can be repurposed to prove another liveness property: *termination*. We 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 $\vdash t \preceq_* s$ 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

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

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 $\vdash \exists(n : \mathbb{N}).\ \triangleright^n \bot$ is provable, while the conclusion $\vdash \triangleright^n \bot$ is false for any n.

To see why that is the case, we take a closer look at the proposition $\vdash \exists(n : \mathbb{N}).\ \triangleright^n \bot$. 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 $\vdash \exists(n : \mathbb{N}).\ \triangleright^n \bot$ means $\forall(i : \mathbb{N}).\ \exists(n : \mathbb{N}).\ n > i$, which holds trivially by picking the witness $n \triangleq i + 1$.

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 $\vdash \exists(n : \mathbb{N}).\ \triangleright^n \bot$ and the simulation $\vdash t_\infty \preceq_* s_{<\infty}$ from § 2.3, the problem was that the witnesses of existential quantification could depend on the step-index i. For $\vdash \exists(n : \mathbb{N}).\ \triangleright^n \bot$, given any step-index i, we could always pick a witness n greater than it; for $\vdash t_\infty \preceq_* s_{<\infty}$, 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 large enough 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, $\vdash \exists(n : \mathbb{N}).\ \triangleright^n \bot$ is no longer provable in a transfinitely step-indexed model. In return, as we show formally in § 5, such a model *does* validate the existential property, thus enabling liveness reasoning.

3 Termination-Preserving Refinement

We now put the key ideas from § 2 into action by introducing **Transfinite Iris**—a full-fledged separation logic capable of proving termination-preserving refinements of higher-order stateful programs. Transfinite Iris follows the "Iris approach" of modularly building up complex reasoning principles from simple abstractions. As such, it does not have a simulation relation ($\preceq_*$) built in as one of its primitive logical connectives. Instead, the simulation relation is defined in terms of simpler connectives. Specifically, it is defined using Hoare triples and separation-logic resources, following the approach of Turon et al. [55] for result refinements (explained below).

In this section, we do not discuss how Hoare triples and the underlying separation-logic resources are themselves defined. These connectives are based on their definition in vanilla Iris [34, 38] extended with ideas from § 2 to support termination-preserving refinements. (For a precise discussion, we refer the reader to the supplementary material [48].) Instead, we will focus on the key rules of Transfinite Iris that enable us to prove termination-preserving refinements.

Although Transfinite Iris is parametric in the source and target language, we use a specific language, called Sequential HeapLang (§ 3.1). We recap the approach of Turon et al. [55] for proving result refinements in separation logic (§ 3.2), and show how it can be extended to termination-preserving refinements in Transfinite Iris (§ 3.3). We conclude this section by proving a termination-preserving refinement for the challenging `memo_rec` example from the introduction (§ 3.4).

3.1 Sequential HeapLang

We use the sequential fragment of HeapLang, the default language in Iris's Coq implementation [30]. This language offers the features of a standard (untyped) functional programming language augmented with ML-like references:

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

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

3.2 Result Refinements in Vanilla Iris

We explain the approach of Turon et al. [55] for proving result refinements. While Turon et al. use a bespoke logic, our variant is embedded in Iris, based on work by Krebbers et al. [39]. The central connectives of this variant of Iris are:

$$\begin{aligned} P, Q \in \mathit{iProp} ::= {} & (x = y) \mid \forall x.\ P \mid \exists x.\ P \mid \triangleright P \mid P * Q \mid P \mathrel{-\!\!*} Q \\ & \mid \ell \mapsto v \mid \{P\}\ e\ \{v.\ Q\} \mid \ell \mapsto_{\mathrm{src}} v \mid \mathrm{src}(e) \mid \cdots \end{aligned}$$

Hoare triples $\{P\}\ e\ \{v.\ Q\}$ are used to reason about the target, while resources $\mathrm{src}(e)$ are used to reason about the source. Similar to the usual points-to connective $\ell \mapsto v$, which states

Basic step-indexed separation logic:

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

Result refinements in Iris *(these rules are not present in Transfinite Iris):*

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

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

Termination-preserving refinements in Transfinite Iris:

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

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

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

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

Figure 1. A selection of proof rules of Iris and Transfinite Iris.

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. 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:

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

The Hoare triple expresses that, assuming the source expression e_s is currently embedded in some evaluation context K, we can execute the target expression e_t to some ground value $v \in G$. This value v cannot just be any value, though—it has to be possible to execute the source expression e_s to this value. That is, in the proof of the Hoare triple, the assumption $\mathrm{src}(K[e_s])$ has to be transformed into $\mathrm{src}(K[v])$ by executing the source expression.

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 $\mathrm{src}(e_s)$ for the simulating source expression. A selection of rules is given in Figure 1. 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 [50], and higher-order ghost state [33]. 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. Let us consider a simple example.

Lemma 3.1. *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 $e_f \triangleq \mathsf{loop}\, f\, ()$, $e_g \triangleq \mathsf{loop}\, g\, ()$, 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 $\{\mathsf{src}(K[e_g]) * \triangleright(e_f \preceq_{\mathbb{1}} e_g)\}\, e_f\, \{\varphi\}$. By executing a pure step of the recursive function loop in the target (the recursive unfolding) using PureT, we have to, in turn, show $\{\mathsf{src}(K[e_g]) * e_f \preceq_{\mathbb{1}} e_g\}\, \mathbf{if}\, f()\, \mathbf{then}\, e_f\, \mathbf{else}\, ()\, \{\varphi\}$. 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 $\mathbf{if}\, g()\, \mathbf{then}\, e_g\, \mathbf{else}\, ()$ by rule PureS. The rest then follows by using Bind to execute $f()$ and $g()$ in the source and the target using the assumption $f() \preceq_{\mathbb{B}} g()$. Then, depending on the outcome, (1) ending the execution in both the source and the target, or (2) executing 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 $e_{\mathsf{loop}} \triangleq \mathsf{loop}\, (\lambda\, ().\, \mathsf{true})\, ()$, we can prove $e_{\mathsf{loop}} \preceq_{\mathbb{1}} \mathbf{skip}$ by Löb induction, analogously to the proof of Lemma 3.1, 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.

3.3 Termination-Preserving Refinements

We present the logical connectives of Transfinite Iris, their intuitive semantics, and their proof rules. The simulation relation $(\preceq_G)$ remains the same as the one from § 3.2. However, since the semantics and proofs rules of Transfinite Iris are different than those of Iris, the simulation relation is in fact adequate for termination-preserving refinements (Theorem 3.3). A selection of the proof rules is shown in Figure 1.

Later stripping and source steps. Let us reconsider the Iris proof of $e_{\mathsf{loop}} \preceq_{\mathbb{1}} \mathbf{skip}$ from § 3.2. There, we could prove a refinement of a diverging target in Iris, without constructing a diverging source execution. To avoid the same happening in Transfinite Iris, 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 Transfinite Iris 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 3.1 from § 3.2 in Transfinite Iris. Recall that since the semantics of Hoare triples changed, this now gives us a termination-preserving refinement.

Lemma 3.2. *If* $f() \preceq_{\mathbb{B}} g()$, *then* $\mathsf{loop}\, f\, () \preceq_{\mathbb{1}} \mathsf{loop}\, g\, ()$.

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

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

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 $\mathbf{if}\, f()\, \mathbf{then}\, e_f\, \mathbf{else}\, ()$ with PureT, and thereby switch back to the source-stepping Hoare triple. The rest of the proof is analogous to Lemma 3.1. □

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* [12].

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 Transfinite Iris's Hoare triples.[3] Thus, we can prove an adequacy result that shows the simulation really entails a termination-preserving refinement.

Theorem 3.3. *If* $\vdash e_t \preceq_G e_s$ *is derivable in Transfinite Iris, then* e_t *is 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. □

[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 [48].

$$\mathsf{Fib}\ \mathsf{fib}\ n \triangleq \textbf{if}\ n < 2\ \textbf{then}\ n\ \textbf{else}\ \mathsf{fib}(n-1) + \mathsf{fib}(n-2)$$

$$\mathsf{Slen}\ \mathsf{slen}\ s \triangleq \textbf{if}\ !s = 0\ \textbf{then}\ 0\ \textbf{else}\ \mathsf{slen}(s+1) + 1$$

$$\begin{aligned}&\mathsf{Lev}\ \mathsf{slen}\ \mathsf{lev}\ (s,t) \triangleq\\ &\quad \textbf{if}\ !s = 0\ \textbf{then}\ \mathsf{slen}\ t\ \textbf{else}\\ &\quad \textbf{if}\ !t = 0\ \textbf{then}\ \mathsf{slen}\ s\ \textbf{else}\\ &\quad \textbf{if}\ !s = !t\ \textbf{then}\ \mathsf{lev}(s+1, t+1)\ \textbf{else}\\ &\quad 1 + \max(\mathsf{lev}(s, t+1), \mathsf{lev}(s+1, s), \mathsf{lev}(s+1, t+1))\end{aligned}$$

Figure 2. Recursive templates of memoized functions.

3.4 Example Refinement: Memoization

Now that we have seen how termination-preserving refinements can be proven using Transfinite Iris, 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 consider the following definition of `memo_rec` in HeapLang:

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

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 Transfinite Iris, we proved a generic and modular specification for memo_rec, and used it to verify two applications: memoizing the Fibonacci function and memoizing the Levenshtein distance [41]. For both examples, the template code is given in Figure 2. To keep the explanation concrete, we focus here on how our specification for memo_rec is used in these applications. The details of the generic specification can be found in the supplementary material [48].

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 2. Given a template t, we can define the standard recursive version of the function r_t and the memoized version m_t as:

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

We wish to show that m_t refines r_t, in the sense that for all natural numbers n, we have that $m_t\ n \preceq_{\mathbb{N}} r_t\ n$ is provable in Transfinite Iris. When specialized to the pure template t, our specification for memo_rec has the following form:

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

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 must execute a step of the source before this assumption can be used.

To prove 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:

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

We should now prove $\forall n.\ h\ n \preceq_{\mathbb{N}} r_t\ n$. We proceed by using Löb induction in a way similar to Lemma 3.2. 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 Iris's *invariant* assertions to ensure that all values in the lookup table are the result of running r_t on the associated keys.

Note that in this proof, it is essential that we can stutter the source while taking steps in the target. These stutters allow us to execute get $\mathit{tbl}\ x$ and set $\mathit{tbl}\ x\ y$ in the memoized version without needing to execute steps in the source. These stutters are important since the steps taken for looking up (or storing) a value in *tbl* do not resemble any steps by the source. The termination of these stuttering steps can be guaranteed by an induction on the number of entries in *tbl*.

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 re-use the stored values in the lookup table during memoization. We omit the details here of how repeatability can be encoded using Iris's *persistent* propositions. Our general specification for memoization replaces the purity side condition in PureMemoRec with this repeatability condition.

To see an example of where our more general specification is useful, consider the Levenshtein function template in Figure 2, 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 2. Thus, to memoize the

Levenshtein function, we define:

$$\text{mlev} \triangleq \textbf{let}\ \text{mslen} = \text{memo\_rec}\ \text{Slen}\ \textbf{in}$$
$$\text{memo\_rec}\ (\text{Lev}\ \text{mslen})$$

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. If the strings were mutated, the memoized function might return the edit distance of the old versions of the strings. We use Iris's invariant mechanism to require that the strings cannot be mutated after memoization, in order to establish repeatability.

4 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. So if we instantiate the source with the inverse of a well-founded relation (*e.g.,* $(>)$ on ordinals), we obtain a technique for proving termination of the target.

To prove termination using Transfinite Iris, we reuse the simulation relation from § 3.3. We make use of the fact that Transfinite Iris is parametric in the source language, and instantiate it with ordinals with the order $(>)$ as the step relation. The resulting logic is a generalization of what is known as time credits [8, 44] to *transfinite* time credits. Transfinite time credits were used in an earlier non-step-indexed logic by da Rocha Pinto et al. [17], and allow for termination arguments based on *dynamic* information learned during program execution (§ 4.1). However, the logic of da Rocha Pinto et al. was restricted to first-order programs, whereas step-indexing allows us to handle higher-order programs. We demonstrate the power of our technique by mechanizing various examples, including (in just 850 lines of Coq) a strengthened version of the main theorem of Spies et al. [49]: termination of a linear language with asynchronous channels, modeling the core of promises in JavaScript (§ 4.2).

4.1 Time Credits

By picking ordinals as the source language of Transfinite Iris, we obtain slightly different logical connectives and proof rules. Instead of the resources $\ell \mapsto_{\text{src}} v$ and $\text{src}(e)$, we have a connective $\$\alpha$ referring to the ordinal source. Instead of the source stepping rules (TPPureS and TPStoreS), we have:

TSource
$$\frac{\langle \$\beta * P\rangle\ e\ \langle v.\ Q\rangle \qquad \beta < \alpha}{\{\$\alpha * \triangleright P\}\ e\ \{v.\ Q\}}$$

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.

In this setting, it falls out for free that we can use Transfinite Iris to prove termination. To prove that an expression e terminates, it suffices to prove terminates(e), which is defined as $\exists\alpha.\ \{\$\alpha\}\ e\ \{v.\text{True}\}$, in Transfinite Iris.

Theorem 4.1. *If* $\vdash$ terminates(e)*, then* e *terminates.*

Proof. We use the existential property (for the quantification over α in the definition of terminates), and then similar reasoning as in the adequacy proof of the termination-preserving refinement Theorem 3.3. □

Our approach generalizes *time credits* [8, 44], which are used traditionally for proving complexity results using 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.

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.,* $\{\$n_u\}\ u()\ \{m.\ m \in \mathbb{N}\}$.

[4] It is well-known that bounded termination is a safety property while termination (without a bound) is a liveness property [49]. 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.

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 determining the number of credits required here requires 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 decrease the ω credits to $k \cdot n_f + 1$ using TSource for the remainder of the execution.

The addition operation $\alpha \oplus \beta$ in TSplit is Hessenberg addition [28]—a well-behaved, commutative notion of addition on ordinals. Commutativity is important in order for ordinals to be used as resources in separation logic (separation logic resources need to form a partial commutative monoid).

While this example is contrived, it highlights the core problem: in compositional termination proofs, where there may not be enough information to bound the length of the execution statically, requiring finite termination bounds can quickly become infeasible. 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).

4.2 Case Studies

Reentrant event loop. We illustrate how transfinite time credits interact with other features of step-indexing. We do so with the example of a reentrant event loop:

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

An event loop consists of a stack of functions q, which can be extended through $\mathsf{addtask}\ q\ f$, and can be executed through $\mathsf{run}\ q$. 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 finite number can ultimately be added because the total number of credits available is a well-founded ordinal.

Logical relation for termination. Transfinite time credits allow us to obtain and mechanize the main result of Spies et al. [49] in Transfinite Iris: 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 Transfinite Iris we can, using our general form of transfinite time credits, simplify their logical relation (which they define and use in an appendix of 40 pages), 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, leveraging the fact that Transfinite Iris has impredicative invariants. The impredicative invariants of Transfinite Iris rely on the fact that the model of Transfinite Iris involves the solution to a recursive domain equation, see § 5.2. If one had attempted to generalize the logical relation of Spies et al. [49] without using Transfinite Iris, then it would have been non-trivial since one would have had to construct a solution to a recursive domain equation. For more details on the polymorphic extension we refer to the supplementary material [48].

5 Foundations of Transfinite Iris

Thus far we have worked with Transfinite Iris informally, developing intuitive explanations of the proof rules and high-level arguments illustrating their use. To ensure that these arguments are sound, it is necessary to show that Transfinite Iris is consistent, *i.e.,* $\vdash \mathsf{False}$ does not hold.

Just as in Iris [34], consistency follows from a model with step-indexed propositions (§ 5.1) and types (§ 5.2), which interprets the connectives and proof rules. In contrast to the model of Iris, the model of Transfinite Iris must also validate the existential property from § 2. This necessitates switching from types and propositions indexed over natural numbers to types and propositions indexed over ordinals. While this switch validates the existential property, it has subtle implications for the logic of Transfinite Iris. We show the most important changes, and prove they are inevitable for a wide class of step-indexed logics (§ 5.3).

5.1 Modeling Step-Indexed Propositions

As with Iris, the model of Transfinite Iris decomposes into two separate and largely orthogonal parts: the part dealing with step-indexing, and the part used to model ownership and resources. The change from finite Iris to Transfinite Iris affects only the part of the model handling step-indexing, so we set aside modeling of resources for the moment, and focus on step-indexing.

Ordinals. The fundamental change in the model is to allow step-indices to be drawn from a type of ordinals *Ord*. This type subsumes the natural numbers to give a richer notion of ordering. For instance, just like with natural numbers, there is a zero ordinal and, for each ordinal α, a strictly larger successor ordinal $\mathsf{s}\,\alpha > \alpha$ exists. Unlike natural numbers, for a small family of ordinals[5] $f : X \to \mathit{Ord}$, there exists a limit ordinal $\sup_{x:X} f(x)$, which is larger than each $f(x)$. For instance, we may form the first infinite ordinal ω as $\sup_{n:\mathbb{N}} \mathsf{s}^n\, 0$. This operation is the linchpin of the proof of the existential property (Theorem 5.2).

Step-indexed propositions. Propositions of Transfinite Iris are interpreted as families of propositions $P : \mathit{Ord} \to \mathit{Prop}$. To ensure that they validate rules such as Löb induction, we limit propositions to families that are *down-closed*.

Definition 5.1. The type of *step-indexed propositions SProp* consists of down-closed predicates $P : \mathit{Ord} \to \mathit{Prop}$. Explicitly, if $\beta \geq \alpha$, then $P\ \beta$ implies $P\ \alpha$.

Given two step-indexed propositions $P, Q : \mathit{SProp}$, we say that $P \vdash Q$ 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 $\Phi : X \to \mathit{SProp}$, there is a step-indexed proposition $(\exists x.\ \Phi(x))(\alpha) \triangleq \exists x.\ \Phi\ x\ \alpha$, which satisfies the standard logical rules for the existential quantifier.

In addition to the standard operations, step-indexed propositions also model the later modality ($\triangleright$):

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

This definition restricts to the more familiar interpretation of later when restricted to the natural numbers with $(\triangleright P)(\mathsf{s}\,\alpha) = P\ \alpha$ and $(\triangleright P)(0) = \mathsf{True}$. It also validates the central rules of step-indexed logics, such as Löb induction, and $P \vdash \triangleright P$.

The existential property. With the machinery of step-indexed propositions in hand, we can prove the existential property directly. The proof rests crucially on *Ord* being large enough to contain limit ordinals for small families.

Theorem 5.2 (Existential Property)**.** *Let $\Phi : X \to \mathit{SProp}$. If $\vdash \exists x.\ \Phi\ x$, then there exists $x : X$ such that $\vdash \Phi\ x$.*

Proof. We proceed by contradiction and assume that $\vdash \Phi\ x$ 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 $\vdash \exists x.\ \Phi\ x$ 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 $\alpha \geq \alpha_x$. We now define $\beta \triangleq \sup_{x:X} \alpha_x$. For each x, we have $\beta \geq \alpha_x$ by construction, hence $\Phi\ x\ \beta$ is false. The contradiction follows. □

[5]The supremum sup can be taken as long as X is smaller than the type of ordinals (think of X being a set, compared to *Ord* being a proper class).

5.2 Modeling Transfinite Iris

Thus far, we have discussed step-indexed propositions, which suffice for modeling connectives like the later modality ($\triangleright$) and validating rules like Löb induction. It is often necessary, however, to *define* propositions and other objects by induction on the step-index *e.g.,* the simulation relation ($\preceq_*$) in § 2.3. Another example is found in the model itself, which must solve a *recursive domain equation* to integrate step-indexed propositions and resources. This recursive equation is necessary to model Iris's impredicative invariants [50] and higher-order ghost state [33], but it does not have a solution if types are interpreted as naively as sets. In order to cope with such equations, the model of Transfinite Iris interprets types as *step-indexed types*. Step-indexed types appear in the model of Iris for the same reasons but, as with propositions, Transfinite Iris alters their form by 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, so that $x \overset{\alpha}{=} y$ implies $x \overset{\beta}{=} y$, if $\beta \leq \alpha$. For instance, *SProp* is an OFE, with $P \overset{\alpha}{=} Q$ if for all $\beta \leq \alpha$, $P(\beta)$ holds iff $Q(\beta)$ does. Recursive definitions can also be constructed directly in *SProp* using a variant of Banach's fixed-point theorem [26].

Theorem 5.3. *Let $f : \mathit{SProp} \to \mathit{SProp}$ be such that $f(P) \overset{\alpha}{=} f(Q)$ if $\forall \beta < \alpha, P \overset{\beta}{=} Q$. Then there exists R such that $R = f(R)$.*

This theorem can be generalized to arbitrary *COFEs*: OFEs which, like iProp, enjoy a certain completeness property.

Solving the recursive domain equation. We now return to a motivating example for step-indexed types: the recursive domain equation for propositions. To tie together step-indexing and ownership, the final step of the model interprets a Transfinite Iris proposition as an element of *iProp*: essentially monotone predicates from the step-indexed type of resources to *SProp*. In order to model *e.g.,* impredicative invariants, resources must depend on *iProp*, which in turn depends on the resources, etc. This is encoded as the *domain equation*: $\mathit{iProp} \approx F(\mathit{iProp}) \xrightarrow{\text{mon}} \mathit{SProp}$, where F describes the model of resources used in Transfinite Iris.

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 [10, 26] 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 [48].

This construction enables us to define *iProp* as the solution to the appropriate domain equation and construct the model.

As a consequence, we construct the model of Transfinite Iris and obtain the following theorem.

Theorem 5.4. *There is a model of Transfinite Iris, which ensures consistency and validates the existential property.*

5.3 Consequences of the Existential Property

With transfinite step-indexing we lose the following rule, which is present in conventional step-indexed logics like Iris:

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

This is not by accident. The existential property (regardless of the step-indexing technique used to model the logic) is fundamentally incompatible with the rule LaterExists, witnessed by the following theorem:

Theorem 5.5. *There exists no consistent logic with a sound later modality (*i.e., $\vdash \triangleright P$ *implies* $\vdash P$*), Löb induction (*i.e., $(\triangleright P \Rightarrow P) \vdash P$*), and the commuting rule LaterExists, which 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 derivable in such a logic. Using the existential property, we obtain an n at the meta-level, so that $\triangleright^n \bot$ is derivable. By soundness of the later modality, this gives that $\bot$ is derivable, which contradicts consistency.

We prove $\vdash (\exists(n : \mathbb{N}).\ \triangleright^n \bot)$ by Löb induction, so it suffices to show:

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

By the commuting rule LaterExists, this is reduced to:

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

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 $(\triangleright\triangleright^n \bot) \vdash (\triangleright^{n+1} \bot)$ is immediate. □

Given that LaterExists does not hold, it is not surprising that we also lose the rule $\triangleright(P * Q) \vdash (\triangleright P) * (\triangleright Q)$: separating conjunction $P * Q$ is defined in terms of existential quantification in the model of (Transfinite) Iris.

6 Related Work

Transfinite step-indexing. The technique of transfinite step-indexing has been used to define logical relations before, but never to define a *program logic*.

Svendsen et al. [52] use step-indexing up to ω^2 to decouple steps of computation from *logical steps* (such as unfolding an invariant), meaning to allow multiple logical steps per step of computation. They do not address liveness reasoning. In their work, Svendsen et al. already solve a recursive domain equation on COFEs for the ω^2 case. We have extended this result to arbitrary, even uncountable, ordinals.

Birkedal et al. [10] solve a recursive domain equation over arbitrary ordinals in the category of *sheaves*. In the present work, we solve recursive domain equations in the category of (transfinite) COFEs, a subcategory of sheaves. While sheaves form, in principle, a suitable model of Iris, they are not well suited for mechanization. Given that mechanization is a central aspect of Iris, sheaves instead of COFEs are an impractical choice. Thus, we have resorted to solving the recursive domain equation in the category of COFEs instead.

Spies et al. [49] use transfinite step-indexing up to ω^ω to prove termination of a linear language with higher-order channels. As explained in § 4.2, 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 far removed from ours. It focuses on type-systems that ensure liveness properties, rather than techniques for verification of liveness properties in a general language with features like general recursion and higher-order state.

Liveness properties in program logics. A number of program logics have been proposed to reason about liveness properties. Liang and Feng [42, 43] 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. [17] extend the concurrent separation logic TaDA [16] with ordinal time credits to prove program termination. In ongoing work, D'Osualdo et al. [21] go further and target more general liveness properties such as "always-eventually" properties. All these logics are *first-order*, which means they do not support reasoning about programs with higher-order state or even higher-order functions. As such, they are not expressive enough to verify, for instance, the `memo_rec` example. These logics have not been mechanized in a proof assistant.

On the other hand, in contrast to the present work, these logics *do* target concurrency. 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 Coq development [48] includes several representative case studies of concurrent *safety* reasoning, which we have ported from Iris to Transfinite Iris. In future work, we expect to take inspiration from the aforementioned logics in order to extend Transfinite Iris to handle concurrent, step-indexed *liveness* reasoning as well.

Yoshida et al. [57] and Charguéraud [15] introduce program logics capable of handling liveness reasoning, even in higher-order stateful settings. Yoshida et al. [57] verify a termination-preserving refinement of a memoized factorial function. In contrast to the present work, their logics are

not based on step-indexing, and therefore they do not offer features like Löb induction, guarded recursion, and impredicative invariants. In this paper, we have focused on enabling liveness in a step-indexed setting, allowing us to use all of the above logical features. In particular, these features allowed us to prove a generic specification for `memo_rec` and then *instantiate it* for multiple clients.

Approximations of liveness reasoning in step-indexed logics. Mével et al. [44] extend Iris with time credits to prove complexity results. As mentioned in § 2 and explained by Spies et al. [49], time credits support reasoning about *bounded* termination, a safety property. Hence, they require an explicit upper bound proved on the meta level before the verification can begin. § 4 shows that this can be inconvenient when proving termination instead of complexity, and even insufficient in cases where modularity is desired.

Tassarotti et al. [53] 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 (established at the meta level) 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. This means they cannot handle examples like `memo_rec`, where the number of steps needed to find a cached result in the lookup table can grow arbitrarily, requiring an unbounded number of stutters. In contrast to our work, Tassarotti et al. [53] do support concurrency. As mentioned above, we plan to incorporate liveness for concurrency in Transfinite Iris in future work.

Frumin et al. [25] 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, and thus much stronger than termination-preserving refinement. The simulation relation they use does not involve an existential quantifier, and therefore they can prove adequacy without the existential property or transfinite step-indexing.

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 (Ed.). 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. To appear in POPL'21.

[10] 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

[11] 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

[12] 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

[13] 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

[14] 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

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

[16] 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

[17] 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

[18] 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

[19] 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/

[20] 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

[21] 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

[22] Derek Dreyer, Amal Ahmed, and Lars Birkedal. 2009. Logical step-indexed logical relations. In *LICS*. 71–80. https://doi.org/10.1109/LICS.2009.34

[23] 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

[24] 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

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

[26] 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

[27] 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

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

[29] 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

[30] Iris project. 2020. A higher-order concurrent separation logic framework implemented and verified in the proof assistant Coq. https://iris-project.org/

[31] 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

[32] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2020. Safe systems programming in Rust: The promise and the challenge. *To appear in CACM* (2020).

[33] 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

[34] 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

[35] 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

[36] S. C. Kleene. 1945. On the interpretation of intuitionistic number theory. *Journal of Symbolic Logic* 10, 4 (1945), 109–124. https://doi.org/10.2307/2269016

[37] 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

[38] 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

[39] 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

[40] 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

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

[42] 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

[43] 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

[44] 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

[45] 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

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

[47] 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

[48] Simon Spies, Lennard Gäher, Daniel Gratzer, Joseph Tassarotti, Robbert Krebbers, Derek Dreyer, and Lars Birkedal. 2020. Transfinite Iris appendix and Coq development. https://iris-project.org/transfinite-iris/

[49] Simon Spies, Neel Krishnaswami, and Derek Dreyer. 2021. Transfinite step-indexing for termination. To appear in POPL'21.

[50] 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

[51] 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

[52] 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

[53] 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

[54] 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

[55] 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

[56] 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

[57] 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