Transfinite Step-Indexing for Termination

SIMON SPIES, MPI-SWS and Saarland University, Germany, and University of Cambridge, UK
NEEL KRISHNASWAMI, University of Cambridge, UK
DEREK DREYER, MPI-SWS, Germany

Step-indexed logical relations are an extremely useful technique for building operational-semantics-based models and program logics for realistic, richly-typed programming languages. They have proven to be indispensable for modeling features like *higher-order state*, which many languages support but which were difficult to accommodate using traditional denotational models. However, the conventional wisdom is that, because they only support reasoning about finite traces of computation, (unary) step-indexed models are only good for proving *safety* properties like "well-typed programs don't go wrong". There has consequently been very little work on using step-indexing to establish *liveness* properties, in particular termination.

In this paper, we show that step-indexing can in fact be used to prove termination of well-typed programs—even in the presence of dynamically-allocated, shared, mutable, higher-order state—so long as one's type system enforces disciplined use of such state. Specifically, we consider a language with asynchronous channels, inspired by promises in JavaScript, in which higher-order state is used to implement communication, and linearity is used to ensure termination. The key to our approach is to generalize from natural number step-indexing to *transfinite step-indexing*, which enables us to compute termination bounds for program expressions in a compositional way. Although transfinite step-indexing has been proposed previously, we are the first to apply this technique to reasoning about termination in the presence of higher-order state.

CCS Concepts: • **Theory of computation → Program reasoning**; *Control primitives*; *Program semantics*.

Additional Key Words and Phrases: termination, transfinite step-indexing, higher-order state, linear types, ordinals, channels, asynchronous computation, asynchronous programming, logical relations

ACM Reference Format:
Simon Spies, Neel Krishnaswami, and Derek Dreyer. 2021. Transfinite Step-Indexing for Termination. *Proc. ACM Program. Lang.* 5, POPL, Article 13 (January 2021), 29 pages. https://doi.org/10.1145/3434294

1 INTRODUCTION

Logical relations are a powerful tool for proving properties about the behavior of programs in a compositional, type-directed way. Starting with Tait [1967], who used logical relations to prove termination in the simply-typed λ-calculus, the power of logical relations has steadily expanded over time, as they have been applied to increasingly complex languages and type systems. For instance, Girard et al. [1989] showed how to extend Tait's method to prove termination of System F (the polymorphic λ-calculus), and Reynolds [1983] generalized Girard's method to support relational reasoning about parametricity. More recently, the technique of *step-indexing*, due originally to Appel and McAllester [2001] and subsequently refined by Ahmed [2004], showed how logical relations

Authors' addresses: Simon Spies, MPI-SWS and Saarland University, Saarland Informatics Campus, Germany, and University of Cambridge, UK, spies@mpi-sws.org; Neel Krishnaswami, University of Cambridge, UK, nk480@cl.cam.ac.uk; Derek Dreyer, MPI-SWS, Saarland Informatics Campus, Germany, dreyer@mpi-sws.org.

2475-1421/2021/1-ART13
https://doi.org/10.1145/3434294

could scale to handle features that had long proved awkward or impossible for denotational methods to handle, notably recursive types and *higher-order state* (*i.e.*, mutable references of any type).

In recent years, considerable effort has been devoted to defining increasingly expressive step-indexed logical relations and step-indexed program logics [Ahmed et al. 2005; Morrisett et al. 2005; Ahmed et al. 2010; Svendsen and Birkedal 2014; Jung et al. 2015, 2018a]. In all of the work cited here, the step-indexed logical relations were used to prove *safety* properties such as "well-typed programs don't go wrong". In contrast, little work has been devoted to the use of step-indexed logical relations for proving *liveness* properties such as *termination*.

On the one hand, this is not surprising: the reason step-indexing was introduced in the first place was to account for features like recursive types and higher-order state, which in general *break* termination. In particular, with unrestricted higher-order state, one can implement recursive functions by backpatching and thus introduce non-termination, as witnessed by Landin's knot [1964][1]:

$$\mathsf{let}\, r = \mathsf{ref}\ (\lambda u : \mathbb{1}.u)\ \mathsf{in}\ (r := \lambda u.\, !r\ u); !r\ ()$$

The expression first creates a new reference r, initially storing the identity function. Subsequently, with $r := \lambda u.\, !r\ u$, the reference is updated to a function which calls the function stored at reference r. Thereafter, if the function stored at reference r is called, it triggers an infinite execution, repeatedly calling itself. Seeing as step-indexed logical relations are compatible with higher-order state, it is reasonable to ask how they could possibly be used to establish that a language is terminating.

On the other hand, higher-order state in practice is often used in terminating ways. In particular, it is often used to implement concurrency on top of a sequential substrate. In Concurrent ML, for instance, concurrency is implemented by channels whose implementation is based on higher-order state and continuations. This implementation strategy is very widely used: runtimes for JavaScript use it to implement the promise abstraction [Friedman and Wise 1976] to support writing asynchronous, non-blocking programs. It is also essentially a terminating abstraction: programs will only loop forever if a programmer writes a loop in client code. Following the approach of Yoshida et al. [2004], the termination of such an abstraction can be ensured statically using a *linear* type system. Specifically, in this paper, we consider a language λ_{CHAN}, which implements channels using higher-order state and continuations (akin to promises in JavaScript), and which uses linearity to enforce disciplined use of higher-order state.

Our main result is that, despite the fact that λ_{CHAN}'s implementation of channels relies crucially on dynamically-allocated, shared, mutable, higher-order state, we can nevertheless use a novel form of step-indexed logical relation to establish compositionally that all well-typed programs in λ_{CHAN} terminate.

The starting point for our approach is the observation, due to Dockins and Hobor [2010, 2012] and Mével et al. [2019], that safe termination of programs (*i.e.*, termination in a value) can be established in a step-indexed program logic if an upper bound n on the maximum length of an execution is provided. Intuitively, this is because "termination within n steps" is no longer a liveness property but rather a safety property: it can be falsified by examining only finite traces. Of course, the question then becomes: how can one determine the right bound n? In the above-cited work, this question was passed on to the user of the logic, who was expected to provide at least a partial answer. In this paper, however, we are considering a linear type system that intrinsically guarantees termination without requiring the programmer to provide any explicit resource bounds. We must therefore develop a way to infer, compositionally, how each expression in a program contributes to the termination bound of the whole program.

[1] In the following, we use Standard ML syntax for unrestricted higher-order references. That is, we use ref for creating new references, ! for reading from a reference, and := for updating a reference.

$$
\begin{array}{lllll}
\text{Values} & v & ::= & \ell \mid () \mid n \mid b \mid \lambda x.e \mid (v_1, v_2) \\
\text{Expressions} & e & ::= & x \mid \ell \mid () \mid n \mid b \mid e_1; e_2 \mid \lambda x.e \mid e_1\ e_2 \mid e_1 \dotplus e_2 \mid \mathsf{iter}(e, e_0, x.e_S) \\
& & \mid & (e_1, e_2) \mid \mathsf{let}\ (x, y) = e_1\ \mathsf{in}\ e_2 \mid \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \\
& & \mid & \mathsf{let}\ (x, y) = \mathsf{chan}()\ \mathsf{in}\ e \mid \mathsf{get}(e_1, e_2) \mid \mathsf{put}(e_1, e_2) \\
\text{Types} & A, B & ::= & \mathbb{1} \mid \mathbb{B} \mid \mathbb{N} \mid A \otimes B \mid A \multimap B \mid \mathsf{Get}\, A \mid \mathsf{Put}\, A \\
\text{Type Contexts} & \Gamma, \Delta & ::= & \cdot \mid \Gamma, x : A \\
\text{Heap Values} & hv & ::= & \mathsf{E} \mid \mathsf{V}(v) \mid \mathsf{C}(v) \\
\text{Heaps} & h & ::= & \cdot \mid \ell \mapsto hv, h
\end{array}
$$

Fig. 1. Syntax of λ_{CHAN}

There are several technical elements to our solution, but the most interesting and important one is to generalize from natural number step-indexing to *transfinite step-indexing*. By employing ordinals in our step-indices, we can compute compositional termination bounds for functions, even though we do not know how their arguments will be instantiated or the heaps under which they will be executed. This idea of transfinite step-indexing is not new: it has been previously proposed by Schwinghammer et al. [2013] for defining logical relations for may and must equivalence in a pure λ-calculus with countable nondeterminism. It was subsequently used by Svendsen et al. [2016] in a setting with higher-order state to allow a finite number of decreases of the step-index with each step of computation, but not to prove termination. To our knowledge, we are the first to propose transfinite step-indexing for proving termination in the presence of higher-order state.

The rest of the paper is structured as follows. In Section 2, we introduce the language λ_{CHAN} for programming with linearly-typed channels, and we motivate its design with a series of examples. In Section 3, we explain the key idea of a transfinitely step-indexed logical relation and how we use it to prove termination for λ_{CHAN}. In Section 4, we spell out the full details of our logical relation and give proof sketches for some of our main results. In Section 5, we compare with related work, and in Section 6, we conclude with a discussion of future work. The accompanying technical report contains detailed proofs of all of the theorems stated in the paper [Spies et al. 2021].

2 ASYNCHRONOUS CHANNELS

In the present work, we consider the language λ_{CHAN} given in Figure 1, an extension of the simply-typed λ-calculus with an implementation of asynchronous channels. A fresh channel can be created with $\mathsf{let}\ (x, y) = \mathsf{chan}()\ \mathsf{in}\ e$, where x is a handle for receiving values over the new channel and y is a handle for sending values. The operation put can be used to send values and the operation get to receive values. For example, assuming a function print, the following expression creates a fresh channel, registers the continuation $\lambda n.\mathsf{print}(n)$ for the channel, and thereafter sends the value 42 over the channel, causing $\mathsf{print}(42)$ to be executed:

$$\mathsf{let}\ (x, y) = \mathsf{chan}()\ \mathsf{in}\ \mathsf{get}(x, \lambda n.\mathsf{print}(n)); \mathsf{put}(y, 42)$$

We assign expressions a type A in the linear type system $\Gamma \vdash e : A$, defined in Figure 2. Besides the base types $\mathbb{1}, \mathbb{B}, \mathbb{N}$, we have linear pairs $A \otimes B$ and linear functions $A \multimap B$. The type $\mathsf{Get}\, A$ is used for the receive handle of a channel, indicating that the channel will transfer a value of type A. Similarly, the type $\mathsf{Put}\, A$ is used for the send handle. We allow values of arbitrary types A to be transferred through channels, including channel handles themselves and functions possibly capturing channel handles. The context Γ is a linear context without an ordering of the variables. We write Γ, Δ for the disjoint union of the contexts Γ and Δ.

As an example, we consider forwarding a value from one channel to another — a straightforward operation using get and put. For instance, if $c_{\text{get}} : \mathsf{Get}\, \mathbb{N}$ and $d_{\text{put}} : \mathsf{Put}\, \mathbb{N}$ are channel handles, then

$$\boxed{\Gamma \vdash e : A}$$

$$\frac{}{x : A \vdash x : A} \qquad \frac{\Gamma \vdash e : B}{\Gamma, x : A \vdash e : B} \qquad \frac{}{\cdot \vdash () : \mathbb{1}} \qquad \frac{\Gamma \vdash e_1 : \mathbb{1} \qquad \Delta \vdash e_2 : A}{\Gamma, \Delta \vdash e_1; e_2 : A} \qquad \frac{}{\cdot \vdash b : \mathbb{B}}$$

$$\frac{\Gamma \vdash e : \mathbb{B} \qquad \Delta \vdash e_1 : A \qquad \Delta \vdash e_2 : A}{\Gamma, \Delta \vdash \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : A} \qquad \frac{}{\cdot \vdash n : \mathbb{N}} \qquad \frac{\Gamma \vdash e_1 : \mathbb{N} \qquad \Delta \vdash e_2 : \mathbb{N}}{\Gamma, \Delta \vdash e_1 \dot{+} e_2 : \mathbb{N}}$$

$$\frac{\Gamma \vdash e : \mathbb{N} \qquad \Delta \vdash e_0 : A \qquad x : A \vdash e_S : A}{\Gamma, \Delta \vdash \mathsf{iter}(e, e_0, x.e_S) : A} \qquad \frac{\Gamma \vdash e_1 : A_1 \qquad \Delta \vdash e_2 : A_2}{\Gamma, \Delta \vdash (e_1, e_2) : A_1 \otimes A_2}$$

$$\frac{\Gamma \vdash e_1 : A_1 \otimes A_2 \qquad \Delta, x : A_1, y : A_2 \vdash e_2 : B}{\Gamma, \Delta \vdash \mathsf{let}\ (x, y) = e_1\ \mathsf{in}\ e_2 : B} \qquad \frac{\Gamma, x : A \vdash e : B}{\Gamma \vdash \lambda x.e : A \multimap B} \qquad \frac{\Gamma \vdash e_1 : A \multimap B \qquad \Delta \vdash e_2 : A}{\Gamma, \Delta \vdash e_1\ e_2 : B}$$

$$\frac{\Gamma \vdash e_1 : \mathsf{Get}\, A \qquad \Delta \vdash e_2 : A \multimap \mathbb{1}}{\Gamma, \Delta \vdash \mathsf{get}(e_1, e_2) : \mathbb{1}} \qquad \frac{\Gamma \vdash e_1 : \mathsf{Put}\, A \qquad \Delta \vdash e_2 : A}{\Gamma, \Delta \vdash \mathsf{put}(e_1, e_2) : \mathbb{1}} \qquad \frac{\Gamma, x : \mathsf{Get}\, A, y : \mathsf{Put}\, A \vdash e : B}{\Gamma \vdash \mathsf{let}\ (x, y) = \mathsf{chan}()\ \mathsf{in}\ e : B}$$

Fig. 2. Typing Rules of λ_{CHAN}

we can forward the value from c to d with $\mathsf{get}(c_{\mathrm{get}}, \lambda x.\mathsf{put}(d_{\mathrm{put}}, x))$. More generally, we define:

$$\mathsf{forward} : \mathsf{Get}\, A \otimes \mathsf{Put}\, A \multimap \mathbb{1}$$
$$\mathsf{forward} \triangleq \lambda(c_{\mathrm{get}}, d_{\mathrm{put}}).\ \mathsf{get}(c_{\mathrm{get}}, \lambda a : A.\ \mathsf{put}(d_{\mathrm{put}}, a))$$

As done above, when giving examples, we use syntactic sugar to ease readability. We use n-ary tuples and pattern matching syntax which can easily be derived from pairs and their elimination operation $\mathsf{let}\ (x, y) = e_1\ \mathsf{in}\ e_2$. Further, we write $\mathsf{let}\ x = e\ \mathsf{in}\ e'$ for $(\lambda x.e')\ e$ where convenient. For the sake of readability, we add type annotations $e : A$ in some cases.

We include booleans and natural numbers in the language as representatives of other algebraic data types such as options, sums, and lists. For the purposes of this work, natural numbers already cause the kind of problems that arise by the inclusion of, for example, lists. Put differently, without natural numbers termination can be reduced to a term size argument, considering that the language is linear. However, with the inclusion of natural numbers a size argument no longer applies and the computational expressiveness is significantly increased. For example, it becomes possible to implement duplication of natural numbers, multiplication, and exponentiation (see Section 2.3). To define such functions, the language contains the iter construct, which enables recursion on natural numbers in a generic fashion.

We equip the language with a single-threaded, heap-based operational semantics, defined in Figure 3. Operationally, each channel is represented by a single location ℓ in the heap, storing either nothing E, a value $\mathsf{V}(v)$, or a continuation $\mathsf{C}(\lambda x.e)$. Initially, the heap stores the empty heap value E. If a value is sent over channel ℓ with $\mathsf{put}(\ell, v)$, then the value is stored in memory as $\mathsf{V}(v)$. If subsequently $\mathsf{get}(\ell, \lambda x.e)$ is executed, then the continuation $\lambda x.e$ is invoked with argument v and the state in the heap is restored to E. If from the initial state $\mathsf{get}(\ell, \lambda x.e)$ is executed, then the continuation is stored in the heap as $\mathsf{C}(\lambda x.e)$ and invoked with argument v once a corresponding $\mathsf{put}(\ell, v)$ is called. Besides the operations on channels, the operational semantics allows standard, pure reductions for the simply typed λ-calculus. Reductions are allowed to occur in any evaluation context K, making it a call-by-value, left-to-right operational semantics.

$$\begin{array}{rrcl} \text{Evaluation Contexts} & K & ::= & \cdot \mid K; e \mid K\ e' \mid v\ K \mid (K, e) \mid (v, K) \mid \mathsf{let}\ (x, y) = K\ \mathsf{in}\ e' \\ & & \mid & K \dot{+} e \mid v \dot{+} K \mid \mathsf{iter}(K, e, x.e') \mid \mathsf{iter}(v, K, x.e) \\ & & \mid & \mathsf{if}\ K\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \mid \mathsf{get}(K, e') \mid \mathsf{get}(v, K) \mid \mathsf{put}(K, e') \mid \mathsf{put}(v, K) \end{array}$$

$$\frac{e \rightsquigarrow_{\mathrm{p}} e'}{(e, h) \rightsquigarrow (e', h)} \qquad \frac{(e, h) \rightsquigarrow_{\mathrm{c}} (e', h')}{(e, h) \rightsquigarrow (e', h')} \qquad \frac{(e, h) \rightsquigarrow (e', h')}{(K[e], h) \rightsquigarrow (K[e'], h')}$$

Pure Reduction

$$\begin{array}{ll} (); e \rightsquigarrow_{\mathrm{p}} e & \mathsf{let}\ (x, y) = (v_1, v_2)\ \mathsf{in}\ e \rightsquigarrow_{\mathrm{p}} e[v_1/x, v_2/y] \\ (\lambda x.e)\ v \rightsquigarrow_{\mathrm{p}} e[v/x] & n \dot{+} m \rightsquigarrow_{\mathrm{p}} n + m \\ \mathsf{if}\ \mathsf{true}\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \rightsquigarrow_{\mathrm{p}} e_1 & \mathsf{iter}(0, v, x.e) \rightsquigarrow_{\mathrm{p}} v \\ \mathsf{if}\ \mathsf{false}\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \rightsquigarrow_{\mathrm{p}} e_2 & \mathsf{iter}(n+1, v, x.e) \rightsquigarrow_{\mathrm{p}} \mathsf{iter}(n, e[v/x], x.e) \end{array}$$

Channel Reduction

$$\begin{array}{rl} (\mathsf{let}\ (x, y) = \mathsf{chan}()\ \mathsf{in}\ e, h) \rightsquigarrow_{\mathrm{c}} (e[\ell/x, \ell/y], h[\ell \mapsto \mathsf{E}]) & \text{if } \ell \notin \mathrm{dom}\, h \\ (\mathsf{get}(\ell, \lambda x.e), h) \rightsquigarrow_{\mathrm{c}} ((), h[\ell \mapsto \mathsf{C}(\lambda x.e)]) & \text{if } h\ell = \mathsf{E} \\ (\mathsf{get}(\ell, \lambda x.e), h) \rightsquigarrow_{\mathrm{c}} (e[v/x], h[\ell \mapsto \mathsf{E}]) & \text{if } h\ell = \mathsf{V}(v) \\ (\mathsf{put}(\ell, v), h) \rightsquigarrow_{\mathrm{c}} ((), h[\ell \mapsto \mathsf{V}(v)]) & \text{if } h\ell = \mathsf{E} \\ (\mathsf{put}(\ell, v), h) \rightsquigarrow_{\mathrm{c}} (e[v/x], h[\ell \mapsto \mathsf{E}]) & \text{if } h\ell = \mathsf{C}(\lambda x.e) \end{array}$$

Fig. 3. Operational Semantics of λ_{CHAN}

We write $h[\ell \mapsto hv]$ for the heap which returns hv for argument ℓ and $h(\ell')$ for any argument $\ell' \neq \ell$. Analogously, we use the notation to update finite and infinite maps with a new binding in the remainder of this work. We assume substitution is capture avoiding, write $e[v/x]$ for the single-point substitution replacing x with v in e, and $e[\theta]$ for the parallel substitution replacing each free variable x in e with θx.

2.1 Asynchronous Programming using Asynchronous Channels

In this and the following subsection, we explore how the asynchronous channels of λ_{CHAN} relate to asynchronous programming encountered in practice. To that end, we first motivate how, conceptually, channels enable a structured approach to asynchronous programming by generalizing continuation-passing style (CPS). We then describe the connection between λ_{CHAN}'s channels and *promises*, an abstraction used for asynchronous programming in JavaScript.

In asynchronous programming, a typical function one might encounter is input.onKeyPress : $(\mathsf{char} \multimap \mathbb{1}) \multimap \mathbb{1}$. It can be used to register an event handler $\mathsf{char} \multimap \mathbb{1}$ for the event of a key press[2]. Its type suggests that it is the CPS transformation $\mathsf{CPS}\, A \triangleq (A \multimap \mathbb{1}) \multimap \mathbb{1}$ of a value of type char, a character. From the CPS perspective, the event handler $\mathsf{char} \multimap \mathbb{1}$ corresponds to the continuation which will eventually be provided to run the computation.

In the literature, it is widely accepted that there is a close connection between asynchronous programming and continuation passing style which has led to the introduction of monadic APIs

[2]The *linear* function types are fitting here since, typically, the event handler is only executed once as it may have side effects. We use types like char and option $\mathbb{N}$ for illustrative purposes even though they are not contained in the type system.

for interacting with asynchronous computations in CPS [Claessen 1999]:

$$\mathsf{return} : A \multimap \mathsf{CPS}\, A \qquad \mathsf{bind} : \mathsf{CPS}\, A \otimes (A \multimap \mathsf{CPS}\, B) \multimap \mathsf{CPS}\, B$$
$$\mathsf{return} \triangleq \lambda a.\lambda f.f\, a \qquad \mathsf{bind} \triangleq \lambda(f, g).\lambda h.f\, (\lambda a : A.g\, a\, h)$$

The monad abstraction allows us to compositionally build up event handlers. For example, given a function toNum : char $\multimap$ option $\mathbb{N}$, we can additionally parse the character of the key that was pressed with:

$$\mathsf{map}(\mathsf{input.onKeyPress}, \mathsf{toNum}) : \mathsf{CPS}(\mathsf{option}\ \mathbb{N})$$

where map is the usual map on monads given by $\mathsf{map}(m, f) \triangleq \mathsf{bind}(m, \lambda x.\mathsf{return}(f\, x))$. By chaining calls to map and bind, increasingly complex event handlers can be constructed.

Unfortunately, with the CPS monad there is no way to provide the result of a computation *from the outside*. For an instance of the monad $m : \mathsf{CPS}\, A$ the functions map and bind can only specify how we want to *use the result*. We cannot use them, or any other function, to *provide the result* of type A. The instance $m : \mathsf{CPS}\, A$ already encapsulates the computation required to produce the result of type A. In this sense, the CPS monad represents a *one-directional communication*: we *first* provide an encapsulated computation and *subsequently*, we specify how to use the result.

Asynchronous channels extend this mechanism by giving programmers fine-grained control over when and how to provide values. They enable *bi-directional communication*. The operation put can be used to return the result of an asynchronous computation. The operation get can be used to register an event handler similar to what is offered by bind in the CPS monad. In fact, Get A forms a similar monad with:

$$\mathsf{return} : A \multimap \mathsf{Get}\, A \qquad \mathsf{bind} : \mathsf{Get}\, A \otimes (A \multimap \mathsf{Get}\, B) \multimap \mathsf{Get}\, B$$
$$\mathsf{return} \triangleq \lambda a.\mathsf{let}\ (a_{\mathrm{get}}, a_{\mathrm{put}}) = \mathsf{chan}()\ \mathsf{in}\ \mathsf{put}(a_{\mathrm{put}}, a); a_{\mathrm{get}} \qquad \mathsf{bind} \triangleq \lambda(a_{\mathrm{get}}, f).\mathsf{let}\ (b_{\mathrm{get}}, b_{\mathrm{put}}) = \mathsf{chan}()\ \mathsf{in}\ \mathsf{get}(a_{\mathrm{get}}, \lambda a.\mathsf{get}(f\, a, \lambda b.\mathsf{put}(b_{\mathrm{put}}, b))); b_{\mathrm{get}}$$

From the outside of the *channel monad* Get A, the put operation can be used at an arbitrary, different point in the program to trigger the execution of (a potential chain of) continuations by providing a value. For example, given $c_{\mathrm{get}} : \mathsf{Get}\ \mathsf{char}$ we can map toNum over an asynchronously computed character with $\mathsf{map}(c_{\mathrm{get}}, \mathsf{toNum}) : \mathsf{Get}\ (\mathsf{option}\ \mathbb{N})$, as before. At an entirely different point in the program, we can decide that the character will be provided upon a key press with $\mathsf{input.onKeyPress}(\lambda x.\mathsf{put}(c_{\mathrm{put}}, x))$, even *after* a continuation was registered.

The above example showcases a more general pattern of how we can connect the CPS monad with the channel monad. If we have an encapsulated computation $f : \mathsf{CPS}\, A$ and a sending handle $a_{\mathrm{put}} : \mathsf{Put}\, A$, then we can use the sending handle to execute the computation and store the result in the channel with $f\ (\lambda a.\mathsf{put}(a_{\mathrm{put}}, a))$. The resulting value can then be accessed at a different point in the program using $a_{\mathrm{get}} : \mathsf{Get}\, A$. Building upon these insights, we obtain the following translations:

$$\mathsf{exec} : \mathsf{CPS}\, A \multimap \mathsf{Get}\, A \qquad \mathsf{cps} : \mathsf{Get}\, A \multimap \mathsf{CPS}\, A$$
$$\mathsf{exec} = \lambda f.\mathsf{let}\ (a_{\mathrm{get}}, a_{\mathrm{put}}) = \mathsf{chan}()\ \mathsf{in}\ f\ (\lambda a.\mathsf{put}(a_{\mathrm{put}}, a)); a_{\mathrm{get}} \qquad \mathsf{cps} = \lambda a_{\mathrm{get}}.\lambda g.\mathsf{get}(a_{\mathrm{get}}, g)$$

These translations between the CPS monad and the channel monad do not imply the two approaches are interchangeable. Not only do both approaches differ in their handling of communication (one-directional vs bi-directional), they also differ in their expressive power. For example, in the absence of channels the CPS monad does not admit a function $\mathsf{split} : \mathsf{CPS}(A \otimes B) \multimap \mathsf{CPS}\, A \otimes \mathsf{CPS}\, B$ which splits the computation of a pair into two computations of the individual components. The reason is that, due to the linearity restriction, the computation may only be executed once but to

implement such a splitting both the left and the right hand side would have to evaluate the computation. In the context of asynchronous programming, linearity is a sensible restriction: evaluating a computation twice is not only inefficient, especially in a setting with I/O operations, it is also dangerous as the computation may be stateful, leading to undesired side effects.

In contrast, the channel monad allows for splitting a computation of a pair $A \otimes B$ into two computations without violating linearity:

$$\begin{aligned}
&\mathsf{split} : \mathsf{Get}(A \otimes B) \multimap \mathsf{Get}\, A \otimes \mathsf{Get}\, B \\
&\mathsf{split} = \lambda c_{\mathrm{get}}.\mathsf{let}\ \left(a_{\mathrm{put}}, a_{\mathrm{get}}\right) = \mathsf{chan}()\ \mathsf{in} \\
&\qquad\qquad \mathsf{let}\ \left(b_{\mathrm{put}}, b_{\mathrm{get}}\right) = \mathsf{chan}()\ \mathsf{in} \\
&\qquad\qquad\quad \mathsf{get}(c_{\mathrm{get}}, \lambda(a, b).\mathsf{put}(a_{\mathrm{put}}, a); \mathsf{put}(b_{\mathrm{put}}, b)); \left(a_{\mathrm{get}}, b_{\mathrm{get}}\right)
\end{aligned}$$

2.2 Asynchronous Channels and Promises

In JavaScript, asynchronous programming is facilitated by the mechanism of *promises*: delayed results for which continuations can be registered. Much like encapsulated computations in CPS, promises are commonly used in a linear fashion to ensure predictable handling of effects. Below, we show how the channels of λ_{CHAN} can be understood as a statically typed abstraction over promises (which are dynamically typed in JavaScript), where the type system of λ_{CHAN} enforces the discipline of linear usage. In particular, we demonstrate how channels can be used to encode the core of promises, and then show how promises in conjunction with higher-order state can be used to encode channels.

We encode a promise transferring a value of type A as an instance of the channel monad $\mathsf{Get}\, A$ defined in Section 2.1. To construct a promise for the result of some computation, one provides the CPS transformation of the computation. That is, one provides a function which takes as its argument a continuation to be used for *resolving the promise*. For instance, we can create a promise which is immediately resolved with value 42 with `new Promise((resolve)=> {resolve(42)})`. Here, the notation `(x) => {s}` is JavaScript arrow syntax for an anonymous JavaScript function `function (x){s}`. In the context of the present work, such functions can be read as $\lambda x.(s; ())$. To encode the operation for creating new promises in λ_{CHAN}, we use the function exec. That is, given an encapsulated computation $m : \mathsf{CPS}\, A$, we obtain a promise as $\mathsf{exec}(m) : \mathsf{Get}\, A$.

Promises offer the function `Promise.resolve(v)` as a short hand for creating a promise which is immediately resolved with the value `v`. We encode `Promise.resolve` using the return operation of the monad $\mathsf{Get}\, A$.

Given a promise `p`, one may register a function `f` to be executed once the promise is resolved with `p.then(f)`. The function `f` may, in turn, return another delayed computation in the form of a promise. Fittingly, we encode `p.then(f)` as $\mathsf{bind}(p, f)$ with the bind operation of the channel monad.

Now for the reverse direction. To encode channels using promises, we need to make the `resolve` handle available outside of the scope of the function passed to the promise constructor. We do so via JavaScript's support for higher-order state:

```
1  function chan(){
2      let res = () => {}
3      let promise = new Promise((resolve) => { res = resolve })
4      return { get: (f) => { promise.then(f) }, put: res }
5  }
```

The function first creates a higher-order reference `res` storing dummy value `() => {}`. Then, when the new promise `promise` is created, `res` is backpatched with `promise`'s `resolve` handle. For the get

operation, we return `(f) => { promise.then(f) }`, and for the put operation the captured `resolve`-handle. Here, `{ get: ..., put: ... }` creates a new object with the fields `get` and `put`.

Above, we have focused on encoding the *core* of promises in a linear setting. In JavaScript, promises have additional behavior which we do not cover in the present work. First, we do not account for additional behavior that is orthogonal to termination. For instance, promises offer functionality for handling errors (called "rejecting a promise"), they flatten nested promises returned in the `then` operation, and they allow resolving a promise multiple times (which has no effect). Second, we do not consider uses of promises that violate linearity, such as providing multiple continuations which are all executed with the value the promise is resolved with. Violations of linearity can be used to introduce non-termination à la Landin's knot[3].

2.3 Additional Features of λ_{CHAN}

So far, we have focused solely on channels since they are the most interesting and distinctive feature of λ_{CHAN}. However, it is difficult to write many interesting programs with linear channels alone. Hence, we have included additional constructs in λ_{CHAN} for interacting with natural numbers and booleans, which help to make λ_{CHAN} an expressive enough language to be worth studying.

One important aspect of λ_{CHAN}, which is reflective of asynchronous programming in practice, is that, unlike channels, natural numbers and booleans are not subject to the linearity restriction. In fact, the linearity restriction does not really apply to values of any *ground* type:

$$G ::= \mathbb{1} \mid \mathbb{B} \mid \mathbb{N} \mid G_1 \otimes G_2$$

Values of ground type can be duplicated by deconstructing and reconstructing them again. For example, with the iter operation, we can define a duplication function for natural numbers $\mathsf{dupl}(n) \triangleq \mathsf{iter}(n, (0,0), (x,y).(x+1, y+1))$. With the linearity restriction lifted, we can define all kinds of primitive recursive functions using iteration. For example, we can define multiplication and exponentiation:

$$\mathsf{mult}(m,n) \triangleq \mathsf{iter}(m, 0, x.x+n) \qquad \mathsf{exp}(m,n) \triangleq \mathsf{iter}(m, 1, x.\mathsf{mult}(x,n))$$

Natural numbers and booleans also enable us to encode a common idiomatic use of JavaScript's `for`-loops. More precisely, loops of the form `for (let i = 0; i < n; i++){ s }`, where `s` does not modify `i`, can be encoded as $\mathsf{iter}(n, 0, i.(s; i+1))$. If the `for`-loop uses state over ground types, we can use arguments of the iteration to encode how the values change over time. For example, consider the following JavaScript program which stores the nth Fibonacci number in `a`:

```
let a = 0, b = 1;
for (let i = 0; i < n; i++){
    let tmp = b;
    b = a + b;
    a = tmp
}
```

We can encode this with iteration as $\mathsf{let}\ (a,b) = \mathsf{iter}(n, (0,1), (a,b).(b, a+b))\ \mathsf{in}\ a$.

3 KEY IDEAS

Operationally, asynchronous channels behave like unrestricted higher-order references. For a single channel, represented by a location ℓ in the heap, the operational behavior is best described by Figure 4. In particular, values can be stored in the heap and retrieved again as an argument to a

[3]The λ_{CHAN} version of Landin's knot is shown at the beginning of Section 3; it can be ported to a non-terminating JavaScript program via the encoding of chan given above.

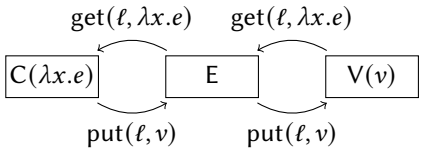


Fig. 4. Unrestricted use

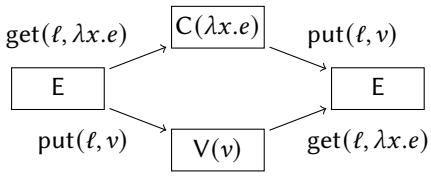


Fig. 5. Linear use

continuation using the operations get and put. Thus, we can construct the equivalent of Landin's knot on channels as:

$$
\begin{aligned}
e_{\text{knot}} \triangleq\ & \text{let } (c_{\text{get}}, c_{\text{put}}) = \text{chan}() \text{ in} \\
& \text{let } f = \lambda u : \mathbb{1}.\text{get}(c_{\text{get}}, \lambda f.\text{put}(c_{\text{put}}, f); f\, u) \text{ in} \\
& \quad \text{put}(c_{\text{put}}, f); f\ ()
\end{aligned}
$$

The function f first retrieves itself from the channel with get. Subsequently, it stores itself again in the channel with put and executes itself. We can trigger an infinite execution by first storing f in the channel with put and thereafter executing f.

If we impose linearity as done by the type system $\Gamma \vdash e : A$, the situation changes drastically: *Every closed, well-typed program* $\vdash e : A$ *terminates.* Under the restriction of linearity, each channel evolves according to Figure 5. There is a clear notion of progress associated with each operation which makes it impossible to create cycles. In particular, there can be at most one put and one get which prevents the construction of e_{knot}. To the reader, it may be quite intuitive at this point that the restrictions imposed by the type system are strong enough that well-typed programs terminate. In fact, there is not even the duplication operation which can usually be found in linear type systems [Morrisett et al. 2005; Brunel and Madet 2012]. Nevertheless, existing techniques for proving termination for programs with higher-order state do not apply.

The issue is that λ_{CHAN} uses dynamic features such as *implicit sharing* (or aliasing), *dynamic allocation of references*, and *dynamic dependencies between references. Implicit sharing* means that access to a location is shared between multiple values—in our case, between a get handle Get A and a put handle Put A, which can be used to transmit values through the location independently. *Dynamic allocation of references* means that references can be allocated dynamically during a program's execution, using in our case the iter construct. (In this context, "dynamic" means that the number of references allocated may depend dynamically on program values like inputs to functions and is not statically determined.) *Dynamic dependencies between references* means that dependencies between channels can similarly be created dynamically, depending on runtime values. For examples of these features and a comparison to existing techniques, we refer the reader to Section 5.

If we were only interested in showing safety, meaning "well-typed programs don't go wrong", instead of termination, ordinary step-indexed logical relations would be a perfect fit. In the literature, they have been used successfully to prove safety of languages with all of these features and many more [Ahmed 2004; Jung et al. 2018a]. In the present work, we use *transfinite* step-indexing and linearity of the type system to define a step-indexed logical relation ensuring termination. To motivate and explain the key ideas and the difference to existing step-indexed logical relations, we first review how logical relations and step-indexed logical relations are traditionally defined.

3.1 A Logical Relations Primer

First, we consider a logical relation for the simply typed λ-calculus ensuring termination for each closed expression of type $a, b ::= \mathbb{1} \mid a \to b$. For each type a, we give an interpretation of the

type for values (the value relation $\mathcal{V}[\![a]\!]$) and an interpretation of the type for expressions (the expression relation $\mathcal{E}[\![a]\!]$):

$$\begin{aligned}\mathcal{V}[\![\mathbb{1}]\!] &\triangleq \{()\}\\ \mathcal{V}[\![a \to b]\!] &\triangleq \left\{\lambda x.e \;\middle|\; \forall v \in \mathcal{V}[\![a]\!].e[v/x] \in \mathcal{E}[\![b]\!]\right\}\\ \mathcal{E}[\![a]\!] &\triangleq \left\{e \;\middle|\; \exists v.e \rightsquigarrow^* v \text{ and } v \in \mathcal{V}[\![a]\!]\right\}\end{aligned}$$

Intuitively, the value relation $\mathcal{V}[\![a]\!]$ contains a value v if v behaves like a value of type a, and the expression relation $\mathcal{E}[\![a]\!]$ contains an expression e if e terminates in a value in $\mathcal{V}[\![a]\!]$. The only value of the unit type $\mathbb{1}$ is $()$ and the function type $a \to b$ only contains a function $\lambda x.e$ if the function applied to values $v \in \mathcal{V}[\![a]\!]$ behaves like an expression of type b.

Given the type interpretations, the proof that every closed, well-typed expression terminates proceeds in two steps. First, we need to connect the type interpretations which only contain closed terms to the type system, denoted by $\Gamma \vdash e : a$ below, which also covers open terms. To this end, we define a semantic interpretation of the typing judgment of the language $\Gamma \vDash e : a \triangleq \forall \theta \in \mathcal{G}[\![\Gamma]\!].\ e[\theta] \in \mathcal{E}[\![a]\!]$, where $\mathcal{G}[\![\Gamma]\!] \triangleq \left\{\theta \;\middle|\; \forall x : a \in \Gamma.\ \theta x \in \mathcal{V}[\![a]\!]\right\}$ contains closing substitutions inserting values from the type interpretations. Next, we show soundness of the syntactic type system with respect to the semantic typing, meaning if $\Gamma \vdash e : a$, then $\Gamma \vDash e : a$. The soundness proof usually proceeds by induction on $\Gamma \vdash e : a$, showing in each case that the respective typing rule is validated in the semantic model, the logical relation, meaning we can replace occurrences of the syntactic notion $\vdash$ by the semantic notion $\vDash$. Soundness lets us deduce termination of well-typed programs. For instance, in the case of $\vdash e : a$, we obtain $\vDash e : a$ and thus $e \rightsquigarrow^* v$ for some value v by unfolding the definitions.

Now, consider extending the language with ML-style higher-order references $a, b ::= \cdots \mid a\ \text{ref}$. The extension by unrestricted higher-order references means it no longer makes sense to prove termination. In particular, Landin's knot [1964], the diverging expression from Section 1, becomes typeable in the extended language. We are forced to change the desired semantic property of the logical relation to *safety*: that is, if an expression e with heap h reduces in some number of steps n to an expression e' with heap h' which can no longer be reduced, written $(e, h) \rightsquigarrow^n (e', h') \not\rightsquigarrow$, then the resulting expression e' is a value.

To ensure safety, we need to modify the logical relation to additionally account for the heap. For example, the expression $!\ell\ ()$ is safe if the value stored at location ℓ in the heap is the function $\lambda x.x$, but not if the value stored there is $()$, since the expression $()\ ()$ does not reduce to a value. We account for values in the heap in the form of *invariants*, $\ell : a$, which may be understood as type assignments to the locations in the heap. In the logical relation, we interpret invariants in an additional relation, the heap typing relation $\mathcal{H}[\![\Phi]\!]$. Intuitively, the heap typing relation $\mathcal{H}[\![\Phi]\!]$ contains a heap h, if for each invariant $\ell : a \in \Phi$ the value at location ℓ is contained in $\mathcal{V}[\![a]\!]$. For references, we ensure in the value relation $\mathcal{V}[\![a\ \text{ref}]\!]$ that the invariant map Φ contains the right invariant. To this end, we relate each value and expression additionally with an invariant map Φ, representing all the invariants currently governing the heap. In the expression relation $\mathcal{E}[\![a]\!]$, we need only prove safety under heaps that are well-typed given the current invariants. The resulting logical relation is depicted in Figure 6.

Note that, in the definition of the type interpretations in Figure 6, to select expressions and invariant maps that have the desired form, we use set comprehensions. In particular, the set $\mathcal{V}[\![\mathbb{1}]\!]_i$ contains $((), \Phi)$ for any invariant map Φ, and the set $\mathcal{HV}[\![\ell : a]\!]_0$ contains any pair of a value and an invariant map.

Unfortunately, for our new heap-aware logical relation, we can no longer define it recursively on the structure of types, as we did for the simple logical relation given earlier in the section. The

$$\mathcal{V}[\![\mathbb{1}]\!]_i \triangleq \{((), \Phi)\} \qquad \mathcal{V}[\![a \text{ ref}]\!]_i \triangleq \{(\ell, \Phi) \mid \ell : a \in \Phi\}$$

$$\mathcal{V}[\![a \to b]\!]_i \triangleq \left\{(\lambda x.e, \Phi) \;\middle|\; \forall j \le i, \Phi' \supseteq \Phi, v.(v, \Phi') \in \mathcal{V}[\![a]\!]_j \Rightarrow (e[v/x], \Phi') \in \mathcal{E}[\![b]\!]_j\right\}$$

$$\mathcal{H}[\![\Phi]\!]_i \triangleq \left\{h \;\middle|\; \operatorname{dom} h = \operatorname{dom} \Phi \text{ and } \forall \ell : a \in \Phi.(h\ell, \Phi) \in \mathcal{HV}[\![\ell : a]\!]_i\right\}$$

$$\mathcal{HV}[\![\ell : a]\!]_{i+1} \triangleq \left\{(v, \Phi) \;\middle|\; (v, \Phi) \in \mathcal{V}[\![a]\!]_i\right\} \qquad \mathcal{HV}[\![\ell : a]\!]_0 \triangleq \{(v, \Phi)\}$$

$$\mathcal{E}[\![a]\!]_i \triangleq \left\{(e, \Phi) \;\middle|\; \begin{array}{l} \forall j \le i, \Phi' \supseteq \Phi, n \le j, h, h', e'.\, h \in \mathcal{H}[\![\Phi']\!]_j \text{ and } (e, h) \rightsquigarrow^n (e', h') \not\rightsquigarrow \Rightarrow \\ \exists \Phi'' \supseteq \Phi', v.\, e' = v \text{ and } h' \in \mathcal{H}[\![\Phi'']\!]_{j-n} \text{ and } (v, \Phi'') \in \mathcal{V}[\![a]\!]_{j-n} \end{array}\right\}$$

Fig. 6. Step-indexed logical relation with higher-order references

reason is as follows. In the expression relation $\mathcal{E}[\![\mathbb{1}]\!]$, we refer to the heap typing relation $\mathcal{H}[\![\Phi]\!]$ for all invariant maps Φ, including $\{\ell : \mathbb{1} \to \mathbb{1}\}$. Thus, unfolding the definition of the heap typing relation, the expression relation $\mathcal{E}[\![\mathbb{1}]\!]$ indirectly depends on the value relation $\mathcal{V}[\![\mathbb{1} \to \mathbb{1}]\!]$ in a negative (contravariant) position. Since $\mathcal{V}[\![\mathbb{1} \to \mathbb{1}]\!]$ depends on $\mathcal{E}[\![\mathbb{1}]\!]$, we encounter a circularity.

To resolve this circularity, Appel and McAllester [2001] introduced (and Ahmed [2004] refined) the technique of *step-indexing* as a way of stratifying the definition. A natural number, the step-index, is added in every relation (see the i and j in Figure 6) to ensure that recursive occurrences of the logical relations are either at a smaller step-index or at the same step-index but at a structurally smaller type. In particular, in the heap typing relation the step-index is decreased, thus avoiding the circularity described above.

For an expression at step-index i, the logical relation provides guarantees about its operational behavior for at most i steps of computation. Specifically, at step-index i the expression relation ensures that the expression does not "get stuck", *i.e.*, terminate in a non-value, in the next i steps.

Why is the step-index tied to the execution steps in this way? At step-index 0, we have not defined any relations of smaller step-index yet. As a consequence, we cannot use the logical relation to constrain the shape or behavior of values contained in the heap typing relation $\mathcal{HV}[\![\ell : a]\!]_0$ — we allow *any* pair of a value and an invariant map. We do not have to provide any guarantees about the value in the heap at step-index 0 *because* the step-indices are tied to the execution steps. Specifically, it takes one step to load a value from the heap with $!\ell$ and at step-index 0 we only provide guarantees about executions of length 0. At larger step-indices than 0, this step of $!\ell$ is used to decrement the step-index such that it matches the one used by the value in the heap.

A consequence of relating step-indices to execution steps is that closure under smaller step-indices becomes important. That is, if $(e, \Phi) \in \mathcal{E}[\![a]\!]_i$ and $j \le i$, then $(e, \Phi) \in \mathcal{E}[\![a]\!]_j$, and similarly for values. The reason this property is important is that as the execution proceeds and thereby the step-index decreases, values stored for example in the heap should remain usable even if they were of a larger step-index when they were inserted into the heap. To obtain closure under smaller step-indices, we explicitly require it in the definitions of $\mathcal{E}[\![a]\!]_i$ and $\mathcal{V}[\![a \to b]\!]_i$ with "$\forall j \le i$". For the logical relation at other types, the property follows by induction.

Finally, having defined our step-indexed logical relations so that they ensure safety for executions of length i, we can ensure that terms are safe for executions of arbitrary length by requiring that they are contained in the logical relation *for all step-indices* i. Thus, we define the semantic typing $\Gamma \vDash e : a \triangleq \forall i. \forall (\theta, \Phi) \in \mathcal{G}[\![\Gamma]\!]_i.(e[\theta], \Phi) \in \mathcal{E}[\![a]\!]_i$ where the context interpretation is given as $\mathcal{G}[\![\Gamma]\!]_i \triangleq \left\{(\theta, \Phi) \;\middle|\; \forall x : a \in \Gamma.\, (\theta x, \Phi) \in \mathcal{V}[\![a]\!]_i\right\}$.

3.2 Step-Indexed Termination

We will now proceed to explore how a step-indexed logical relation can be used not only for proving safety of programs with higher-order state but also for proving termination. Before we do that, let us step back and examine more generally what step-indexing achieves.

In general terms, let us say that we wish to prove a property $P(e)$ concerning the execution of a program e. With step-indexing, as we have already seen, the property becomes stratified by a step index i into a family of properties $P_i(e)$, where each $P_i(e)$ only considers the first i steps of e's computation. Instead of directly proving $P(e)$, step-indexing will allow us to prove $\forall i.P_i(e)$. This is good enough for establishing *safety* properties, because a safety property of e is precisely one which can be determined from just looking at finite prefixes of e's execution. Or put another way, a safety property of e is one that can be falsified by exhibiting *some* finite prefix of e's execution trace for which the property does not hold.

Unfortunately, termination—the property we are interested in proving for λ_{CHAN} programs—is not a safety property but rather a *liveness* property: it says that "eventually something good happens (the program terminates with a value)". There is no way to stratify "termination of e" into a step-indexed family of termination predicates $T_i(e)$ because there is no way (in general) to establish that e does not terminate by only examining a finite prefix of e's execution. How, then, can we use a step-indexed logical relation to prove termination of λ_{CHAN} programs?

"Termination with a resource bound" as a safety property. As a first step toward answering this question, let us take a page from prior work by Dockins and Hobor [2010, 2012] and Mével et al. [2019]. Suppose that somehow we know an upper bound n on the number of steps it will take e to terminate. The bound n here can be thought of as a kind of countable *resource* that we are given up front—Mével et al. [2019] call this resource "time credits". We can then imagine executing e on a machine that consumes one resource unit at every step of computation. On such a machine, e is safe to execute with initial resource n if and only if e evaluates to a value in at most n steps.

As this description suggests, by changing the property we wish to establish from "e terminates" to "e terminates in at most n steps", we can change a liveness property into a safety property! In particular, unlike "e terminates", the latter property can be determined by simply examining the first n steps of e's execution trace. Thus, we can recast this latter property using a family of propositions $T_i(e, n)$, defined (very roughly) as follows:

$$T_i(e, n) \triangleq \text{"if } i \geq n\text{, then } e \text{ terminates in at most } n \text{ steps"}$$

For $i < n$, the property $T_i(e, n)$ holds trivially, but for $i \geq n$, it is equivalent to "e terminates in at most n steps". Thus, a term e terminates in at most n steps if and only if $\forall i.T_i(e, n)$. Moreover, the property $T_i(e, n)$ satisfies the essential criterion of step-indexed relations: it can be determined by only looking at the first i steps of e's execution.

Dockins and Hobor [2010, 2012] and Mével et al. [2019] used step-indexed relations in the manner of $T_i(e, n)$ to build models of logics for proving that programs adhere to resource bounds. However, in their work, the user of the logic needed to supply the bound n explicitly in their logical assertions. In contrast, we want to prove termination for λ_{CHAN}, in which the type system relies on linearity to ensure termination but does not mention any resource bounds explicitly. How can we do it?

Computing the resource bound compositionally. Intuitively, the idea is to change the logical relation $\mathcal{E}[\![A]\!]_i$ so that (like $T_i(e, n)$ above) it is a predicate on both a term e and its resource bound n, and to change the definition of semantic typing ($\Gamma \vDash e : A$) so that it asserts the existence of *some* bound n on the number of steps e will take to terminate. Restricting attention for the moment to the simplified case of closed terms, the definition of $\cdot \vDash e : A$ will thus look *somewhat like* the

following (changes underlined):

$$\cdot \vDash e : A \;\approx\; \underline{\exists n.}\forall i.\forall \Phi.(e, \Phi, \underline{n}) \in \mathcal{E}[\![A]\!]_i$$

As part of the soundness proof (*i.e.*, proving that syntactic typing implies semantic typing), we must then show how to automatically compute the witness for the existentially-quantified bound n.

Computing this bound turns out to be rather subtle because it must be done *compositionally*. For example, suppose we were to just naively choose the bound n for a term e based on how many steps e itself will take to terminate. We would then run into trouble in proving semantic soundness for λ_{CHAN}, where we have to establish (among other things) that $\cdot \vDash f : A \multimap B$ and $\cdot \vDash e : A$ implies $\cdot \vDash f\ e : B$. To establish this, we must compute a resource bound n for $f\ e$ solely from the resource bounds n_f and n_e already computed for f and e, without knowing anything about what those terms are. But this is impossible: f and e could both be values, in which case n_f and n_e could both be 0, while $f\ e$ could take an arbitrarily long time to execute. Similarly, for an expression such as $\mathsf{put}(\ell, v)$ which potentially executes an unknown continuation in the heap, the execution can take arbitrarily long.

Therefore, rather than compute a bound for e based only on the resources needed for e itself to terminate, we will instead compute e's *local contribution* to the resource bound of any program it is a part of. And thanks to the linear nature of the λ_{CHAN} type system, we will be able to compute a resource bound for a whole λ_{CHAN} program as a predictable combination of the local contributions of its subexpressions.

Thus far, we have treated resources as synonymous with natural numbers, but we will see shortly that natural numbers are not good enough. In the subsections that follow, we will motivate by example what form resources should take and how resource bounds can be computed compositionally.

3.3 Towards a Transfinite Model of Resources

At first glance, natural numbers seem like a good choice for modeling the resource that is consumed at every execution step. For example, consider the expression:

$$e_{\text{single}} \triangleq \mathsf{let}\ (c_{\text{get}}, c_{\text{put}}) = \mathsf{chan}()\ \mathsf{in}\ \mathsf{get}(c_{\text{get}}, \lambda x.x); \mathsf{put}(c_{\text{put}}, ())$$

It creates a single fresh channel and uses it to send the unit value $()$. The expression terminates in 4 execution steps and correspondingly consumes 4 units of resource during its execution.

Counting "serious" steps. Frustratingly, natural numbers are already not expressive enough to cover termination of simple, pure expressions if resources are consumed for every step. For instance, consider the function:

$$e_{\text{dupl}} \triangleq \lambda m.\mathsf{iter}(m, (0,0), (m_1, m_2).(m_1 \dot{+} 1, m_2 \dot{+} 1))$$

which duplicates its argument, a natural number m. If resources are consumed for every step, the number of steps cannot be bounded by a single natural number n as the number of steps depends on the argument m.

The expression e_{dupl} is already contained in the simply typed λ-calculus where every expression terminates. Intuitively, the addition of a heap should not impact the termination guarantees one obtains for *pure* computations in the simply typed λ-calculus. Luckily, Dreyer et al. [2010, 2011] established that actually we only need to count "serious" steps: computation steps that are connected with the feature that is motivating our use of step-indexing in the first place. In particular, their step-indexed relations only counted steps related to folding/unfolding at recursive types and manipulation of the heap. We will adopt a similar, relaxed approach.

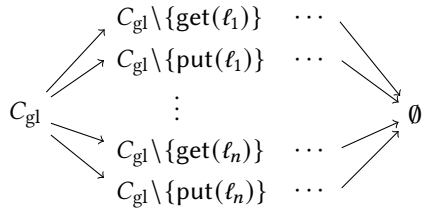


Fig. 7. Decrease of the global capability set

To explain the relaxed approach we will adopt, let us first consider a simplified setting: proving termination of expressions where the channels $\ell_1, \ldots, \ell_n$ have already been allocated and no fresh channels will be allocated. Under this assumption, we can associate a *capability* with each channel operation. We associate $\text{put}(\ell)$ with a put on channel ℓ and $\text{get}(\ell)$ with a get on channel ℓ. In our simplified setting, the set $C_{\text{gl}} = \{\text{put}(\ell_i), \text{get}(\ell_i) \mid i = 1, \ldots, n\}$ contains all the capabilities that can be used during the execution.

As a first approximation, we consider *finite sets of capabilities* as resources instead of natural numbers. Instead of a natural number n, one may think of a capability set C as the resource consumed by executing the corresponding operations in C. Under the simplifying assumption, the global set of capabilities C_{gl} then evolves according to Figure 7: it decreases according to the well-founded relation $C \subsetneq C'$. Given that the relation $C \subsetneq C'$ is well-founded, finite capability sets are suitable as step-indices for the stratification of circularities in the logical relation. The intuition is that at step-index C, the logical relation guarantees termination of all expressions which only use operations corresponding to the capabilities in C. For example, at step-index $\{\text{put}(\ell_1), \text{get}(\ell_2)\}$, the relation will be sufficiently expressive to guarantee termination of $\text{get}(\ell_2, \lambda x.x); \text{put}(\ell_1, 42)$ consuming $\{\text{put}(\ell_1), \text{get}(\ell_2)\}$ and $\text{put}(\ell_1, 0)$ consuming $\{\text{put}(\ell_1)\}$ but will not provide guarantees about the termination of $\text{put}(\ell_3, 0)$ as $\{\text{put}(\ell_3)\}$, the resource it consumes, is not contained in the step-index.

Transfinite step-indexing. Even if we tweak the logical relation to only consume resources for operations which manipulate the heap, it is still insufficient for guaranteeing termination of *all* λ_{CHAN} expressions. In general, expressions allocate fresh channels which invalidates the assumption that we can fix a set of channels $\ell_1, \ldots, \ell_n$ before the execution. Allocating a fresh channel ℓ' increases the resource by $\text{get}(\ell')$ and $\text{put}(\ell')$, if we use capability sets as resources. Unfortunately, adding capabilities to a capability set makes it unusable as a step-index: step-indices may only decrease during the execution.

The solution is to additionally bound the number of fresh channels that are allocated by an ordinal α. Whenever an expression allocates a fresh channel ℓ', it consumes a fragment of the ordinal α and obtains the resources $\text{get}(\ell')$ and $\text{put}(\ell')$. To understand why we use an ordinal to bound the number of channels that are allocated instead of a natural number, consider the following expression:

$$e_{\text{dynamic}} \triangleq \lambda n.\text{iter}(n, (), \_.e_{\text{single}})$$

The function performs a dynamic number of heap allocations which cannot be determined before the argument n of the function is known. The ordinal ω is large enough to bound the number of channels that are allocated by e_{dynamic}, regardless of the argument. In this setting, the ordinal ω^2 may then be understood as the right to finitely often pick a finite number of channels to allocate, for example in a nested iteration.

Formally, we use pairs of ordinals and capability sets (α, C) as resources and by extension as step-indices. For step-indices, we order them lexicographically, meaning $(\alpha, C) < (\alpha', C') \triangleq \alpha < \alpha' \vee (\alpha = \alpha' \wedge C \subsetneq C')$, to obtain a well-founded relation. A decrease in the ordinal indicates that fresh channels have been allocated, and a decrease in the capability set indicates that the channel operations have been executed. The lexicographic ordering allows us to add fresh capabilities to C whenever we decrease α. We use the well-founded partial order to stratify the circularities in the definition of the logical relation — similar to the case of natural numbers and finite capability sets. Fittingly, we also denote the pairs of ordinal and capability set (α, C) by i, j, and k in the following, and use them as indices for the type interpretations.

3.4 Computing Resource Bounds

Now that we have the right notion of resource, we can define step-indexed type interpretations $\mathcal{E}[\![A]\!]_i$ and $\mathcal{V}[\![A]\!]_i$, which can be used to to obtain termination guarantees if we know the right resource bound. For the resource bound, due to our transfinite resource model, we no longer have to bound the number of steps the program will take — it suffices to bound the number of channels it will allocate. Since we do not want to put the burden of providing this bound on the user, the question remains how we can actually determine a sufficient resource bound statically.

To understand how we determine such a bound, consider the following example:

$$e_{42} \triangleq \mathsf{let}\ (c_{\mathrm{get}}, c_{\mathrm{put}}) = \mathsf{chan}()\ \mathsf{in}\ \mathsf{get}(c_{\mathrm{get}}, \lambda n.\mathsf{iter}(n \dot{+} 1, (), \_.e_{\mathrm{single}})); \mathsf{put}(c_{\mathrm{put}}, 41)$$

The expression e_{42} allocates 42 channels when executed. To determine that indeed 42 channels are allocated, one needs to symbolically execute the program, including simulating the heap. Clearly, we cannot just execute expressions to statically determine their resource consumption. Instead, we are going to "compute" compositionally a sufficient bound on the number of channels that are allocated during the proof of semantic soundness.

Before explaining how exactly we compute the bound, we first show how we define the semantic typing judgment:

$$\Gamma \vDash e : A \triangleq \exists \alpha. \forall i. \forall (\theta, R_\theta) \in \mathcal{G}[\![\Gamma]\!]_i. (e[\theta], R_\theta \oplus R_{\mathrm{ord}(\alpha)}) \in \mathcal{E}[\![A]\!]_i$$

where $\mathcal{G}[\![\Gamma]\!]_i$ is the context interpretation, relating closing substitutions θ to the resources that the values inside them need in order to terminate safely. Technically, the resources we use in the logical relation also incorporate invariants to ensure that the expression terminates *safely*. For now, we will ignore that detail and just think of them as a pair of an ordinal and a set of capabilities. In this sense, the resource $R_{\mathrm{ord}(\alpha)}$ corresponds to the pair $(\alpha, \emptyset)$.

Using ordinals, it is almost trivial to compute the number of channels that are allocated during the proof of semantic soundness. For $\mathsf{let}\ (x, y) = \mathsf{chan}()\ \mathsf{in}\ e$, we account for the channel allocation by picking $\alpha \triangleq \alpha_e \oplus 1$ where α_e is an upper bound on the number of channels that are allocated by executing e. Here, we write $\alpha \oplus 1$ for the *natural addition* [Hessenberg 1906] of α and 1 (see below). For sequential composition $e_1; e_2$, we pick $\alpha \triangleq \alpha_1 \oplus \alpha_2$ where α_1 is an upper bound for e_1 and α_2 an upper bound for e_2. Similarly, we pick $\alpha \triangleq \alpha_1 \oplus \alpha_2$ for $\mathsf{get}(e_1, e_2)$ and $\mathsf{put}(e_1, e_2)$.

For variables, we pick the bound $\alpha \triangleq 0$. The bound 0 is sufficient since in the closing substitution θ every value comes equipped with its own resource, following the approach of compositionally distributing the contributions of each expression. For unit, Booleans, and natural numbers, we also pick the bound $\alpha \triangleq 0$.

For functions $\lambda x.e$, we pick $\alpha \triangleq \alpha_e$ where α_e is an upper bound for the number of channels that are allocated by e. Picking $\alpha \triangleq \alpha_e$ works because the ordinal α_e was chosen independently of the the context interpretation and thus is a sufficient upper bound regardless of which value is inserted for x. For function application $e_1\ e_2$, we pick $\alpha \triangleq \alpha_1 \oplus \alpha_2$ where α_1 is an upper bound for e_1 and α_2

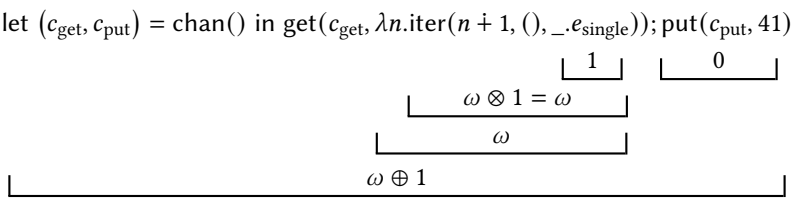


Fig. 8. Computation of the bound of e_{42}

an upper bound for e_2. The expression e_1 evaluates to a function which already accounts for the resource it will consume during execution.

For iteration $\text{iter}(e, e_0, x.e_S)$, we pick $\alpha \triangleq \alpha_e \oplus \alpha_0 \oplus (\omega \otimes \alpha_S)$ where α_e is the upper bound for e, α_0 is the upper bound for e_0, and α_S is the upper bound for e_S. Here, we write $\omega \otimes \alpha_S$ for the *natural multiplication* of ω and α_S (see below). By bounding the number of locations that are allocated during iteration by $\omega \otimes \alpha_S$, we do not even attempt to do any inference or prediction on the semantics of expressions. The result of e is guaranteed to be *some* natural number n so $\omega \otimes \alpha_S$ is certainly going to be sufficient as it is larger than $n \otimes \alpha_S$. Similarly, we over-approximate if e then e_1 else e_2 with $\alpha_e \oplus \alpha_1 \oplus \alpha_2$ where α_e is the bound for e, α_1 the bound for e_1, and α_2 the bound for e_2.

Following these rules, we compute $\alpha = 1$ for e_{single}, $\alpha = 0$ for e_{dupl}, and $\alpha = \omega$ for e_{dynamic}. For e_{42}, we compute $\alpha = \omega \oplus 1$, as depicted in Figure 8. As the example e_{42} shows, the bounds we compute are not and do not have to be tight at all due to the fact that we use ordinals.

In the computations above, we use *natural addition* $\alpha \oplus \beta$ to combine ordinals instead of standard ordinal addition (denoted by $\alpha + \beta$ below). To explain why, we consider $e_{\text{single}}; e_{\text{dynamic}}$. If we used standard ordinal addition to combine the ordinals, then $e_{\text{single}}; e_{\text{dynamic}}$ would be bounded by $1 + \omega$, but under standard ordinal addition, $1 + \omega = \omega$. The execution of e_{single} will already allocate a channel which means it will consume the resource $R_{\text{ord}(1)}$. To consume the resource, it would have to decrement ω to some natural number n. This natural number n would then have to bound the number of channels allocated by e_{dynamic}. As mentioned above, before the argument of e_{dynamic} is known, we cannot bound the number of channels that are allocated by a natural number n.

The issue with standard ordinal addition is that it is not commutative. We would like to have $1 + \omega = \omega + 1$ since $\omega + 1$ is strictly larger than ω and thus can be decreased to ω. We circumvent the problem of non-commutativity by using natural addition following da Rocha Pinto et al. [2016]. Natural addition on ordinals, defined in Section 4, is quite well-behaved and enjoys many of the same properties as addition on natural numbers, such as commutativity and cancellativity. Analogous to natural addition is *natural multiplication* (also defined in Section 4), a commutative notion of multiplication on ordinals which, as explained above, we use to bound iteration.

4 LOGICAL RELATION

In this section, we define a step-indexed logical relation for λ_{CHAN} and use it to prove safe termination of closed, well-typed λ_{CHAN} programs. Analogous to the logical relation of Section 3.1, the type interpretations (depicted in Figure 9) consist of a value relation $\mathcal{V}[\![A]\!]_i$, a heap typing relation $\mathcal{H}[\![\Phi]\!]_i$, and an expression relation $\mathcal{E}[\![A]\!]_i$ where the step-indices i, j, k are now transfinite and of the form (α, C) for some ordinal α and capability set C. In the following, we incrementally introduce and motivate the individual components.

Resources, Step-Indices, and Logical State

Capabilities	p, q	$::=$	$\mathsf{get}(\ell) \mid \mathsf{put}(\ell) \mid \mathsf{al}(\ell)$
Capability Sets	C, D	$::=$	$\emptyset \mid C \uplus \{p\}$
Invariant Maps	Φ	$::=$	$\emptyset \mid \Phi, \ell : A$
Resources	R	$::=$	$(C, \Phi, \alpha) \mid \lightning$
Resource Maps	ρ	$::=$	$\emptyset \mid \rho, \ell \mapsto R$
Step-Indices	i, j, k	$::=$	(α, C)
Logical State	s	$::=$	$\mathsf{Start} \mid \mathsf{Cont} \mid \mathsf{Val} \mid \mathsf{Done}$
Logical State Map	σ	$::=$	$\emptyset \mid \sigma, \ell \mapsto s$

Value Relation

$$\mathcal{V}[\![\mathbb{1}]\!]_i \triangleq \{((), \epsilon)\} \qquad \mathcal{V}[\![\mathbb{B}]\!]_i \triangleq \{(\mathsf{true}, \epsilon), (\mathsf{false}, \epsilon)\} \qquad \mathcal{V}[\![\mathbb{N}]\!]_i \triangleq \{(n, \epsilon) \mid n \in \mathbb{N}\}$$

$$\mathcal{V}[\![\mathsf{Get}\, A]\!]_i \triangleq \{(\ell, (\{\mathsf{get}(\ell)\}, \{\ell : A\}, 0))\} \qquad \mathcal{V}[\![\mathsf{Put}\, A]\!]_i \triangleq \{(\ell, (\{\mathsf{put}(\ell)\}, \{\ell : A\}, 0))\}$$

$$\mathcal{V}[\![A \otimes B]\!]_i \triangleq \left\{((v_1, v_2), R_1 \oplus R_2) \;\middle|\; (v_1, R_1) \in \mathcal{V}[\![A]\!]_i \text{ and } (v_2, R_2) \in \mathcal{V}[\![B]\!]_i\right\}$$

$$\mathcal{V}[\![A \multimap B]\!]_i \triangleq \left\{(\lambda x.e, R_e) \;\middle|\; \forall j \le i, (v, R_v) \in \mathcal{V}[\![A]\!]_j.\ (e[v/x], R_v \oplus R_e) \in \mathcal{E}[\![B]\!]_j\right\}$$

Heap Typing

$$\mathcal{H}[\![\Phi]\!]_i \triangleq \left\{(h, \sigma, \rho) \;\middle|\; \begin{array}{l}\mathrm{dom}\,\Phi = \mathrm{dom}\,h = \mathrm{dom}\,\sigma = \mathrm{dom}\,\rho \text{ and} \\ \forall \ell : A \in \Phi.\ (h\ell, \sigma\ell, \rho\ell) \in \mathcal{HV}[\![\ell : A]\!]_i\end{array}\right\}$$

$$\begin{aligned}\mathcal{HV}[\![\ell : A]\!]_{\alpha,C} \triangleq\ & \{(\mathsf{E}, \mathsf{Start}, \epsilon), (\mathsf{E}, \mathsf{Done}, \epsilon)\} \\ & \cup \left\{\ (\mathsf{V}(v), \mathsf{Val}, R)\ \middle|\ \exists C'.\ C = C' \uplus \{\mathsf{put}(\ell)\} \text{ and } (v, R) \in \mathcal{V}[\![A]\!]_{\alpha,C'}\right\} \\ & \cup \left\{(\mathsf{C}(v), \mathsf{Cont}, R) \;\middle|\; \exists C'.\ C = C' \uplus \{\mathsf{get}(\ell)\} \text{ and } (v, R) \in \mathcal{V}[\![A \multimap \mathbb{1}]\!]_{\alpha,C'}\right\}\end{aligned}$$

Expression Relation

$$\mathcal{E}[\![A]\!]_i \triangleq \left\{(e, R_e) \;\middle|\; \begin{array}{l}\forall j \le i, R_f, \Phi, h, \sigma, \rho.\ \mathsf{RI}_j(R_e, R_f, \Phi, h, \sigma, \rho) \Rightarrow \\ \exists k \le j, \Phi' \supseteq \Phi, h', \sigma', \rho', v, R_v.\ \mathsf{RI}_k(R_v, R_f, \Phi', h', \sigma', \rho') \\ \text{and } (e, h) \rightsquigarrow^* (v, h') \text{ and } \sigma \dashrightarrow^* \sigma' \text{ and } (v, R_v) \in \mathcal{V}[\![A]\!]_k\end{array}\right\}$$

where

$$\begin{aligned}\mathsf{RI}_{\alpha,C}(R_e, R_f, \Phi, h, \sigma, \rho) \triangleq\ & (h, \sigma, \rho) \in \mathcal{H}[\![\Phi]\!]_{\alpha,C} \text{ and } \mathsf{used}\,\sigma \,\#\, C \text{ and} \\ & \exists D.C = D \uplus \mathsf{idx}\,\sigma \text{ and } R_e \oplus R_f \oplus \textstyle\bigoplus_{\ell \in \mathrm{dom}\,\rho} \rho\ell \sqsubseteq (D, \Phi, \alpha)\end{aligned}$$

$$\mathsf{idx}(\ell, \mathsf{Val}) = \{\mathsf{put}(\ell)\} \qquad \mathsf{idx}(\ell, \mathsf{Cont}) = \{\mathsf{get}(\ell)\} \qquad \mathsf{idx}(\ell, s) = \emptyset \quad \text{othw.}$$

$$\mathsf{idx}\,\sigma = \bigcup_{\ell \in \mathrm{dom}\,\sigma} \mathsf{idx}(\ell, \sigma\ell)$$

$$\mathsf{used}(\ell, \mathsf{Done}) = \{\mathsf{al}(\ell), \mathsf{get}(\ell), \mathsf{put}(\ell)\} \qquad \mathsf{used}(\ell, s) = \{\mathsf{al}(\ell)\} \quad \text{othw.}$$

$$\mathsf{used}\,\sigma = \bigcup_{\ell \in \mathrm{dom}\,\sigma} \mathsf{used}(\ell, \sigma\ell)$$

Fig. 9. Logical Relation

Resources and ownership. In the logical relation, we relate each program component (*i.e.*, each expression, value, and value in the heap) with a resource, its local contribution to safe termination. Each resource R is either a triple (C, Φ, α) where C is a set of capabilities, Φ an invariant map, and α an ordinal, or it is invalid, meaning $R = \lightning$. We think of these resources as being owned by the respective program component, following existing work on substructural type systems [Krishnaswami et al. 2012]. Intuitively, we interpret owning a resource $R = (C, \Phi, \alpha)$ as: (1) the knowledge that the invariants $\ell : A \in \Phi$ are enforced in the heap typing relation[4], (2) the right to execute instructions corresponding to capabilities in C, and (3) the right to allocate α new channels. The capability $\mathsf{get}(\ell)$ corresponds to the right to receive a value on channel ℓ, the capability $\mathsf{put}(\ell)$ to the right to send a value over channel ℓ, and the capability $\mathsf{al}(\ell)$ to the right to physically allocate ℓ, meaning the right to add ℓ to the heap. To obtain ownership of fresh capabilities $\mathsf{get}(\ell)$, $\mathsf{put}(\ell)$, and $\mathsf{al}(\ell)$, we require giving up ownership of a fraction of α during the execution in the expression relation.

Given that each program component owns its own resource, to reason about composite expressions or interactions with the heap, we define an operation $R \oplus R'$ to combine them:

$$(C, \Phi, \alpha) \oplus (C', \Phi', \alpha') \triangleq (C \cup C', \Phi \cup \Phi', \alpha \oplus \alpha') \qquad \text{if } C \# C' \text{ and } \Phi \operatorname{ag} \Phi'$$
$$R \oplus R' \triangleq \lightning \qquad \text{otherwise}$$

For capability sets, we ensure linear use with a disjoint union where $C \# C'$ means C and C' are disjoint. The disjoint union ensures that we cannot combine the resources of expressions with overlapping capabilities. In other words, capabilities cannot be duplicated and thus convey the *exclusive* right to perform the respective operation.

For invariant maps, we take the union of both maps, provided they do not assign two different types to a single location, denoted by $\Phi_1 \operatorname{ag} \Phi_2$. Invariants can be duplicated (since $\{\ell : A\} \operatorname{ag} \{\ell : A\}$) and thus can be shared between program components. They do, however, have to agree on the type of the value exchanged over the channel.

For ordinals, we use natural addition. We only work with ordinals α, β, γ strictly smaller than ω^ω. Each such ordinal can be expressed in its Cantor normal form, intuitively a polynomial over powers of ω. More precisely, for each ordinal $\alpha < \omega^\omega$, there is some $k \in \mathbb{N}$ and coefficients $a_0, \ldots, a_k \in \mathbb{N}$ such that $\alpha = \sum_{i=0}^{k} a_i \omega^i \triangleq \omega^k a_k + \cdots + \omega^0 a_0$. In this representation, natural addition may be understood as polynomial addition. Formally, given two ordinals $\alpha = \sum_{i=0}^{k} \omega^i a_i$ and $\beta = \sum_{i=0}^{l} \omega^i b_i$, we define natural addition by $\alpha \oplus \beta \triangleq \sum_{i=0}^{\max(k,l)} \omega^i (a_i + b_i)$ where $a_i \triangleq 0$ for $i = k+1, \ldots, l$ and $b_i \triangleq 0$ for $i = l+1, \ldots, k$.

Resources with the operation $\oplus$ form a commutative monoid. This is, in part, due to the fact that natural addition is quite well-behaved, satisfying many of the laws already satisfied by addition on natural numbers.

Lemma 4.1. *Natural addition is associative and commutative, and* 0 *is an identity. Natural addition is compatible with* $<$ *on ordinals, meaning* $\alpha < \beta$ *implies* $\alpha \oplus \gamma < \beta \oplus \gamma$ *for all* α, β, γ. *Natural addition is cancellative, meaning* $\alpha \oplus \gamma = \beta \oplus \gamma$ *implies* $\alpha = \beta$.

Natural addition gives rise to *natural multiplication*, a commutative multiplication operation which distributes over natural addition. In the present work, we use only two special cases: Multiplication by natural numbers $n \otimes \alpha \triangleq \sum_{i=0}^{k} \omega^i (n \cdot a_i)$ and multiplication by ω defined as $\omega \otimes \alpha \triangleq \sum_{i=0}^{k} \omega^{i+1} a_i$ where α is given in Cantor normal form as $\alpha = \sum_{i=0}^{k} \omega^i a_i$. Importantly, we have $n \otimes \alpha \leq \omega \otimes \alpha$ which allows us to bound iteration (the inequality is strict for $\alpha \neq 0$).

[4]Compared to the logical relation in Section 3.1, this is a minor difference in that given the invariant $\ell : A$, the channel represented by ℓ is used to exchange a value of type A instead of store a value of type A.

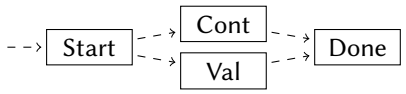
Lemma 4.2. *Natural multiplication has the following properties:*

$$0 \otimes \alpha = 0 \qquad 1 \otimes \alpha = \alpha \qquad (n_1 + n_2) \otimes \alpha = n_1 \otimes \alpha \oplus n_2 \otimes \alpha \qquad n \otimes \alpha \leq \omega \otimes \alpha$$

We write $R \sqsubseteq R'$, if $R' = R \oplus R''$ for some R'' and denote the empty resource by $\epsilon \triangleq (\emptyset, \emptyset, 0)$. In the following, it will sometimes be convenient to consider resources which only consist of capabilities $R_{\mathrm{cap}(C)} \triangleq (C, \emptyset, 0)$, invariants $R_{\mathrm{inv}(\Phi)} \triangleq (\emptyset, \Phi, 0)$, or ordinals $R_{\mathrm{ord}(\alpha)} \triangleq (\emptyset, \emptyset, \alpha)$.

Value relation. In the value relation $\mathcal{V}[\![A]\!]_i$, we relate values that semantically inhabit the type A with the resources they own. Unit, Booleans, and natural numbers only own the empty resource ϵ. Inhabitants of Get A are locations ℓ that own the resource $R_{\mathrm{get}(\ell,A)} \triangleq (\{\mathrm{get}(\ell)\}, \{\ell : A\}, 0)$. Ownership of $\mathrm{get}(\ell)$ entails the right to perform a get on location ℓ and ownership of $\ell : A$ guarantees that the invariant $\ell : A$ is satisfied by the heap in the heap typing relation $\mathcal{H}[\![\Phi]\!]_i$. Analogously, inhabitants of Put A are locations ℓ that own the resource $R_{\mathrm{put}(\ell,A)} \triangleq (\{\mathrm{put}(\ell)\}, \{\ell : A\}, 0)$. For linear pairs $A \otimes B$, we combine the resources owned by the individual components. In the interpretation of the linear function type $A \multimap B$, a function is related to those resources that are required to execute the body safely, provided the resources owned by the argument are added.

Logical state. There is no physical difference between a channel in its initial state and a channel after it has been used to exchange a value (as depicted in Figure 5): in both cases, the channel location stores the value E. However, there is a logical difference, which matters for keeping track of resource usage in our logical relation. Thus, to distinguish both states, we introduce the notion of the *logical state* of a channel. The logical state of every channel in the heap is tracked in a state map σ, which evolves according to the relation $\sigma \dashrightarrow \sigma'$ induced by the following transition system:



Heap typing. The heap typing relation $\mathcal{H}[\![\Phi]\!]_i$ ensures that the values in the heap satisfy the invariants $\ell : A \in \Phi$. More precisely, the heap typing relation $\mathcal{H}[\![\Phi]\!]_i$ relates the logical state σ with the physical heap h and the resources owned by values in the heap ρ such that the invariants in Φ are upheld.

Recall from Section 3 that the main purpose of step-indices is to stratify the definition of the logical relation. To achieve such a stratification, we need to decrease the step-index for the occurrence of the value relation inside of the heap typing relation. We decrease the step-index by removing a capability from the step-index in the cases Val and Cont in the definition of $\mathcal{HV}[\![\ell : A]\!]_{\alpha,C}$. By decreasing the step-index at this point, we inadvertently weaken the assurances about values in the heap — as usual for a step-indexed logical relation. In *this* logical relation, the weakening is not consequential due to linear use of capabilities. Linear use of capabilities ensures that the capabilities that are removed in $\mathcal{HV}[\![\ell : A]\!]_{\alpha,C}$ are not used by the values stored in the heap or any other expression. For instance, the capability $\mathrm{get}(\ell)$ is required to store the continuation $\lambda x.e$ at location ℓ. Linearity ensures that in an execution of the continuation, the capability $\mathrm{get}(\ell)$ cannot be used as it was used to store $\lambda x.e$ in the heap in the first place.

Note that in a conventional step-indexed logical relation the heap typing relation enjoys a downward closure property[5]: if an element is in the heap typing relation at step-index i, then it is also contained at all smaller step-indices $j \leq i$. The logical relation presented here does *not* enjoy the downward closure property in its general form. For example, in the base case $(0, \emptyset)$, the equivalent of step-index 0, the heap typing relation $\mathcal{HV}[\![\ell : A]\!]_{0,\emptyset}$ contains $(\mathsf{E}, \mathsf{Start}, \epsilon)$ and $(\mathsf{E}, \mathsf{Done}, \epsilon)$ instead of "all values". For the logical relation presented here, it suffices to have the

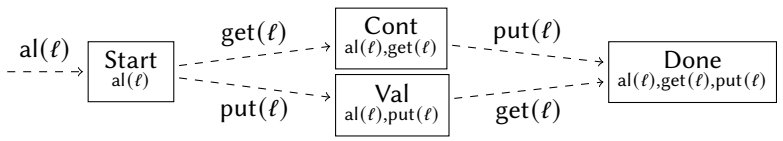
[5]We elaborate more on the downward closure property itself and its interpretation in Section 4.1.

weaker downward closure properties stated as parts (3) and (4) of Lemma 4.3. This weaker version suffices because in the cases where we use the downward closure property (the compatibility lemmas for iter, get, put, and chan) the additional precondition is satisfied which is ensured by the definition of the resource interpretation.

Expression relation. The expression relation $\mathcal{E}[\![A]\!]_i$ forms the heart of the logical relation. Intuitively, we relate an expression e with the resource R_e, if we can use the resource R_e to execute e to some value v with residual resource R_v. More precisely, to execute an expression e owning resource R_e in the expression relation, we assume some global invariant map Φ, a current logical state map σ, a current heap h, and a current resource map ρ. To remain compositional, following the approach of Morrisett et al. [2005], we additionally assume some *frame* resource R_f, representing the resource owned by a potential context in which e could be executed. We then show that the *resource interpretation* RI is preserved during the execution with a potential decrease in the step-index. That is, we assume the resource interpretation is initially satisfied and show that at the end of the execution the resource interpretation is satisfied for some resource R_v owned by the result v, some extended invariant map Φ', some updated logical state map σ', some updated heap h', and some updated resource map ρ'. After the execution, we ensure that the logical state map was advanced according to the transition relation $\dashrightarrow$ and that the result v is contained in the value relation.

The resource interpretation $\mathsf{RI}_{\alpha,C}(R_e, R_f, \Phi, h, \sigma, \rho)$ serves three purposes: First, it ensures that the heap is well-typed given the current invariant map Φ with $(h, \sigma, \rho) \in \mathcal{H}[\![\Phi]\!]_{\alpha,C}$. Second, it ensures *ownership* of resources has the meaning described above with $R_e \oplus R_f \oplus \bigoplus_{\ell \in \mathrm{dom}\,\rho} \rho\ell \sqsubseteq (D, \Phi, \alpha)$. For the resources owned by different program components, the invariant maps must all agree and be contained in the global invariant map Φ, the capability sets must be pairwise disjoint and contained in the step-index since $D \subseteq C$, and the sum of all ordinals must be at most α. The latter guarantees that we can compositionally decrease the global step-index by locally decreasing the ordinal in the resource of an expression. For example, if the expression resource is $R_e = R' \oplus R_{\mathrm{ord}(1)}$, we can decrease α and allocate new capabilities by giving up the resource $R_{\mathrm{ord}(1)}$.

Third, the resource interpretation enforces that capabilities are used according to the following state transition system with tokens [Turon et al. 2013], where "tokens" in our case are capabilities:



Below every state are the capabilities currently *owned* by that state. Formally, for a state map σ, the capabilities currently owned by σ are given by $\mathsf{idx}\,\sigma \cup \mathsf{used}\,\sigma$. The functions $\mathsf{idx}\,\sigma$ and $\mathsf{used}\,\sigma$ distinguish between two modes in which capabilities can be owned by σ, depending on whether they are still contained in the step-index or not. We transfer the capability $\mathsf{put}(\ell)$ (resp. $\mathsf{get}(\ell)$) to $\mathsf{idx}\,\sigma$ at the point where a value (resp. continuation) is stored in the heap. We move them to $\mathsf{used}\,\sigma$ at the point where the continuation or value is retrieved during the execution. If the state map σ owns capabilities, then no program component can own those capabilities since $R_e \oplus R_f \oplus \bigoplus_{\ell \in \mathrm{dom}\,\rho} \rho\ell \sqsubseteq (D, \Phi, \alpha)$ where $D \,\#\, \mathsf{idx}\,\sigma$ and $D \,\#\, \mathsf{used}\,\sigma$.

As an example of how capabilities are transferred, we consider $e \triangleq \mathsf{get}(\ell, \lambda x.x); \mathsf{put}(\ell, ())$ with resource $R_{\mathrm{cap}(\{\mathsf{get}(\ell),\mathsf{put}(\ell)\})}$ at step-index $(0, \{\mathsf{get}(\ell), \mathsf{put}(\ell)\})$. Initially, the expression owns the capabilities $\mathsf{get}(\ell)$ and $\mathsf{put}(\ell)$ and the logical state of ℓ must be Start (otherwise the expression could not own both capabilities). With the execution of $\mathsf{get}(\ell, \lambda x.x)$, the capability $\mathsf{get}(\ell)$ is transferred to the capabilities owned by the logical state, specifically $\mathsf{idx}(\ell, \mathsf{Cont})$, while the remaining expression

$(); \mathsf{put}(\ell, ())$ only retains the capability $\mathsf{put}(\ell)$. Note that $\mathsf{get}(\ell)$ remains in the step-index. While keeping capabilities of operations that have already been executed as part of the step-index may seem counterintuitive, it allows us to decrease the step-index in the heap typing relation $\mathcal{H}[\![\Phi]\!]_{\alpha,C}$ by removing them. When we eventually execute $\mathsf{put}(\ell, ())$, we move the capability $\mathsf{put}(\ell)$ owned by the expression, and the capability $\mathsf{get}(\ell)$ contained in $\mathsf{idx}(\ell, \mathsf{Cont})$, to $\mathsf{used}(\ell, \mathsf{Done})$. At this point, the capabilities are removed from the step-index.

4.1 Kripke Relation

Similar to traditional step-indexed logical relations, the logical relation for λ_{CHAN} is a *Kripke logical relation*. Characteristic for a Kripke logical relation is a notion of worlds w and a partial order $w \preceq w'$ relating worlds with "future" worlds and a monotonicity property with respect to future worlds: if a value is contained in world w, then it is contained in all future worlds $w' \succeq w$. This monotonicity property ensures that knowledge or assurances that we have about program components remain true during execution where the world is gradually advanced.

In this logical relation, step-indices may be considered worlds with the notion of a future world corresponding to a smaller step-index. That is $(\alpha, C) \preceq (\beta, D)$ if and only if $(\alpha, C) \geq (\beta, D)$. In our setting, a future world describes a future state of the program in the sense that a decrease in the ordinal and an increase in the capability set corresponds to fresh channels being allocated and a decrease in the capability sets to the respective channel operations being used.

The following lemma, Lemma 4.3, ensures closure under future worlds, meaning the type interpretations are closed under smaller step-indices. More precisely, the value relation and the expression relation are closed under future worlds while the heap typing relation is almost closed under future worlds. For heap typing, we have to ensure that whenever we move to a future world $(\beta, D) < (\alpha, C)$, we do not discard capabilities which are still used to decrease the step-index of values in the heap.

Lemma 4.3. *Let $i = (\alpha, C) \geq (\beta, D) = j$.*

(1) $\mathcal{E}[\![A]\!]_i \subseteq \mathcal{E}[\![A]\!]_j$
(2) $\mathcal{V}[\![A]\!]_i \subseteq \mathcal{V}[\![A]\!]_j$
(3) *If $(hv, s, R) \in \mathcal{HV}[\![\ell : A]\!]_i$ and $\mathsf{idx}(\ell, s) \subseteq D$, then $(hv, s, R) \in \mathcal{HV}[\![\ell : A]\!]_j$.*
(4) *If $(h, \sigma, \rho) \in \mathcal{H}[\![\Phi]\!]_i$ and $\mathsf{idx}\ \sigma \subseteq D$, then $(h, \sigma, \rho) \in \mathcal{H}[\![\Phi]\!]_j$.*

For the expression relation, the question arises why the relation is closed under smaller step-indices given that termination is not closed under taking fewer steps. The answer is contained in the definition of the expression relation. If the step-index is decreased below the capabilities and the ordinal contained in the resource of the expression, the implication in the definition of the relation becomes trivially true.

In contrast to traditional step-indexed logical relations such as the one from Section 3, this logical relation, specifically the value relation, is not closed under larger invariant maps. We have opted for not closing the value relation under larger invariant maps or larger resources in general as it is somewhat counterintuitive to the notion of ownership: in a setting with closure under extended resources, the unit value $()$ could own the invariant $\ell : \mathbb{N}$ and the capability $\mathsf{put}(\ell)$. It suffices that the expression relation is closed under larger invariants maps and resources in general:

Lemma 4.4. (1) *If $(e, R_e \oplus R_{ord(\alpha)}) \in \mathcal{E}[\![A]\!]_i$ and $\alpha \preceq \beta$, then $(e, R_e \oplus R_{ord(\beta)}) \in \mathcal{E}[\![A]\!]_i$.*
(2) *If $(e, R_e) \in \mathcal{E}[\![A]\!]_i$, then $(e, R_e \oplus R) \in \mathcal{E}[\![A]\!]_i$.*

4.2 Semantic Typing

To obtain a termination result for well-typed expressions from the logical relation, we connect the type system $\Gamma \vdash e : A$ to the type interpretations using a semantic typing judgement $\Gamma \vDash e : A$. To define $\Gamma \vDash e : A$, recall that the relations $\mathcal{V}[\![A]\!]_i$ and $\mathcal{E}[\![A]\!]_i$ are defined on closed values and expressions while the type system $\Gamma \vdash e : A$ is defined on open expressions, tracking free variables in the type context Γ. We close the expression with a substitution from the context interpretation $\mathcal{G}[\![\Gamma]\!]_i$:

$$\mathcal{G}[\![\cdot]\!]_i \triangleq \{(\theta, \epsilon)\}$$
$$\mathcal{G}[\![\Gamma, x : A]\!]_i \triangleq \left\{(\theta, R_\theta \oplus R) \;\middle|\; (\theta, R_\theta) \in \mathcal{G}[\![\Gamma]\!]_i \text{ and } (\theta x, R) \in \mathcal{V}[\![A]\!]_i\right\}$$
$$\Gamma \vDash e : A \triangleq \exists \alpha. \forall i. \forall (\theta, R_\theta) \in \mathcal{G}[\![\Gamma]\!]_i. (e[\theta], R_\theta \oplus R_{\mathrm{ord}(\alpha)}) \in \mathcal{E}[\![A]\!]_i$$

For the context interpretation, we assume the values inserted by the closing substitution come equipped with their own resources, following the compositional approach to resource consumption.

Recall that with the ordinal α in the definition of $\Gamma \vDash e : A$, we give an upper bound on the number of channels that will be allocated during the execution of e. We require the bound α to be chosen independently of the step-index i. Otherwise, for step-index (β, C) the ordinal $\alpha \triangleq \beta \oplus 1$ would make the judgement trivially true regardless of the expression e. As a consequence, the ordinal α also has to be chosen independently of the closing substitution $(\theta, R_\theta) \in \mathcal{G}[\![\Gamma]\!]_i$ similar to how we bound functions without knowing the argument.

4.2.1 Soundness. Before we can deduce termination using the semantic typing $\Gamma \vDash e : A$, we first have to prove the syntactic typing system $\Gamma \vdash e : A$ sound:

Theorem 4.5. *If* $\Gamma \vdash e : A$, *then* $\Gamma \vDash e : A$.

As for traditional logical relations, the proof proceeds by induction on the typing derivation. For each typing rule, we prove a corresponding *compatibility lemma* — a lemma that says we can replace all occurrences of the syntactic typing $\vdash$ in the rule with the semantic typing $\vDash$.

Given the rules for "computing" α from Section 3.4, the remaining proofs of the compatibility lemmas are quite similar to those of a traditional step-indexed logical relation. In particular, the logical relation satisfies many of the typical properties of step-indexed logical relations such as:

Lemma 4.6. (1) $\mathcal{V}[\![A]\!]_i \subseteq \mathcal{E}[\![A]\!]_i$.
(2) *If* $(e, R_e) \in \mathcal{E}[\![A]\!]_i$ *and* $e' \rightsquigarrow^* e$, *then* $(e', R_e) \in \mathcal{E}[\![A]\!]_i$.
(3) *If* $(e, R_e) \in \mathcal{E}[\![A]\!]_i$ *and* $\forall j \le i, (v, R_v) \in \mathcal{V}[\![A]\!]_j$. $(K[v], R_v \oplus R_K) \in \mathcal{E}[\![B]\!]_j$, *then* $(K[e], R_e \oplus R_K) \in \mathcal{E}[\![B]\!]_i$.

First, every value in the value relation is already contained in the expression relation. Second, we can take steps which do not manipulate the heap, written $e' \rightsquigarrow^* e$. Third, we can reason about composite expressions $K[e]$ by reasoning about e and the remaining expression $K[v]$ after e has been executed, if we abstract over the result v (sometimes referred to as the "bind" lemma).

To illustrate how these properties are used in the proof of a typical compatibility lemma, we showcase sequential composition:

Lemma 4.7.

$$\frac{\Gamma \vDash e_1 : \mathbb{1} \qquad \Delta \vDash e_2 : A}{\Gamma, \Delta \vDash e_1; e_2 : A}$$

Proof Sketch. By assumption, we have α_1 for e_1 such that $\forall i. \forall (\theta, R_\theta) \in \mathcal{G}[\![\Gamma]\!]_i. (e_1[\theta], R_\theta \oplus R_{\mathrm{ord}(\alpha_1)}) \in \mathcal{E}[\![\mathbb{1}]\!]_i$ and α_2 for e_2 such that $\forall i. \forall (\theta, R_\theta) \in \mathcal{G}[\![\Delta]\!]_i. (e_2[\theta], R_\theta \oplus R_{\mathrm{ord}(\alpha_2)}) \in \mathcal{E}[\![A]\!]_i$. We pick for $e_1; e_2$ the ordinal $\alpha_1 \oplus \alpha_2$.

Let $(\theta, R_\theta) \in \mathcal{G}[\![\Gamma, \Delta]\!]_i$. An induction on Δ shows that $R_\theta = R_1 \oplus R_2$ for some R_1, R_2 such that $(\theta, R_1) \in \mathcal{G}[\![\Gamma]\!]_i$ and $(\theta, R_2) \in \mathcal{G}[\![\Delta]\!]_i$. From the assumptions about e_1 and e_2, we obtain $(e_1[\theta], R_1 \oplus R_{\mathrm{ord}(\alpha_1)}) \in \mathcal{E}[\![\mathbb{1}]\!]_i$ and $(e_2[\theta], R_2 \oplus R_{\mathrm{ord}(\alpha_2)}) \in \mathcal{E}[\![A]\!]_i$. It remains to show:

$$(e_1[\theta]; e_2[\theta], R_1 \oplus R_{\mathrm{ord}(\alpha_1)} \oplus R_2 \oplus R_{\mathrm{ord}(\alpha_2)}) \in \mathcal{E}[\![A]\!]_i$$

With Lemma 4.6, it suffices to show $(v_1; e_2[\theta], R_{v_1} \oplus R_2 \oplus R_{\mathrm{ord}(\alpha_2)}) \in \mathcal{E}[\![A]\!]_j$ for all $j \leq i$ and $(v_1, R_{v_1}) \in \mathcal{V}[\![\mathbb{1}]\!]_j$. By definition of $\mathcal{V}[\![\mathbb{1}]\!]_j$, we know $v_1 = ()$ and $R_{v_1} = \epsilon$. The claim follows with Lemma 4.6 and Lemma 4.3 given the pure reduction $(); e_2[\theta] \rightsquigarrow e_2[\theta]$. □

Most of the remaining compatibility lemmas consist of similar routine context manipulations and applications of the properties of the logical relation outlined above. The only notable difference compared to standard compatibility lemmas of other logical relations is the instantiation of the existential quantifier which we have sketched in Section 3.4 for the interesting cases. In the few non-standard cases, those concerning channels and iteration, we sketch the proofs for closed expressions:

Lemma 4.8.

$$\frac{\forall \ell.(e[\ell/x, \ell/y], R_e \oplus R_{get(\ell,A)} \oplus R_{put(\ell,A)}) \in \mathcal{E}[\![B]\!]_i}{(\mathsf{let}\ (x, y) = \mathsf{chan}()\ \mathsf{in}\ e, R_e \oplus R_{ord(1)}) \in \mathcal{E}[\![B]\!]_i}$$

Proof Sketch. To execute $\mathsf{let}\ (x, y) = \mathsf{chan}()\ \mathsf{in}\ e$, we first assume the resource interpretation at some step-index $j \leq i$. Based on the locations allocated in the heap and the capabilities contained in the step-index, we pick a fresh location ℓ. We update the resource interpretation by adding $\ell \mapsto \mathsf{Start}$ to the logical state, $\ell \mapsto \mathsf{E}$ to the heap, and $\ell : A$ to the invariant map. Further, we replace the resource $R_{\mathrm{ord}(1)}$ with $R_{\mathrm{get}(\ell,A)}$ and $R_{\mathrm{put}(\ell,A)}$ and, in the process, we decrease the step-index to some $j' < j$ according to the lexicographic ordering. That is, we trade in a fraction of the ordinal in the step-index (which is at least 1 since the expression owns $R_{\mathrm{ord}(1)}$) for additional capabilities. We then use the assumption to execute $e[\ell/x, \ell/y]$ and obtain the resource interpretation at the end of the execution of $e[\ell/x, \ell/y]$ for some step-index $k \leq j' < j$. □

Lemma 4.9.

$$\frac{(v_{ch}, R_{ch}) \in \mathcal{V}[\![\mathsf{Get}\ A]\!]_i \qquad (v_\lambda, R_\lambda) \in \mathcal{V}[\![A \multimap \mathbb{1}]\!]_i}{(\mathsf{get}(v_{ch}, v_\lambda), R_{ch} \oplus R_\lambda) \in \mathcal{E}[\![\mathbb{1}]\!]_i}$$

Proof Sketch. By definition of $\mathcal{V}[\![\mathsf{Get}\ A]\!]_i$, the value v_{ch} and thus the entire expression owns $R_{\mathrm{get}(\ell,A)}$ for some ℓ, including the capability $\mathrm{get}(\ell)$. Thus, the logical state of ℓ is either Start or Val since the logical state cannot also own $\mathrm{get}(\ell)$.

If the logical state is Start, we store the continuation $\lambda x.e$ in the heap and advance the logical state to Cont. Fittingly, we transfer ownership of the capability $\mathrm{get}(\ell)$ to the logical state.

If the logical state is Val, there is some value v stored at location ℓ in the heap. We update the logical state of ℓ to Done. The ownership of $\mathrm{get}(\ell)$ is transferred from the expression to the logical state, specifically used, and the ownership of $\mathrm{put}(\ell)$ is transferred from idx to used. Fittingly, we remove both capabilities from the step-index. For the value v the heap typing relation provides guarantees for the decreased step-index. We use these guarantees to execute the continuation with the value v. □

Lemma 4.10.

$$\frac{(v_{ch}, R_{ch}) \in \mathcal{V}[\![\mathsf{Put}\ A]\!]_i \qquad (v, R_v) \in \mathcal{V}[\![A]\!]_i}{(\mathsf{put}(v_{ch}, v), R_{ch} \oplus R_v) \in \mathcal{E}[\![\mathbb{1}]\!]_i}$$

Proof Sketch. Analogous to Lemma 4.9. By definition of $\mathcal{V}[\![\mathsf{Put}\, A]\!]_i$, the value v_{ch} and thus the entire expression owns $R_{\mathrm{put}(\ell,A)}$ for some ℓ, including the capability $\mathrm{put}(\ell)$. Thus, the logical state of ℓ is either Start or Cont. If the logical state is Start, we store the value v in the heap, giving up $\mathrm{put}(\ell)$. If the logical state is Cont, we update the logical state of ℓ to Done, giving up $\mathrm{put}(\ell)$. Subsequently, we use the assumption about the value v to execute the continuation stored in the heap. □

Lemma 4.11.

$$\frac{(e,R) \in \mathcal{E}[\![\mathbb{N}]\!]_i \qquad (e_0,R_0) \in \mathcal{E}[\![A]\!]_i \qquad (\lambda x.e_S, R_{ord(\alpha_S)}) \in \mathcal{V}[\![A \multimap A]\!]_i}{(\mathsf{iter}(e,e_0,x.e_S), R \oplus R_0 \oplus R_{ord(\omega\otimes\alpha_S)}) \in \mathcal{E}[\![A]\!]_i}$$

Proof Sketch. Using Lemma 4.6, we first evaluate e to some $n \in \mathbb{N}$. Subsequently, we decrease $R_{\mathrm{ord}(\omega\otimes\alpha_S)}$ to $R_{\mathrm{ord}(n\otimes\alpha_S)}$ with Lemma 4.4 and similarly decrease the step-index. Thereafter, it remains to show that:

$$\frac{(e_0,R_0) \in \mathcal{E}[\![A]\!]_j \qquad (\lambda x.e_S, R_{\mathrm{ord}(\alpha_S)}) \in \mathcal{V}[\![A \multimap A]\!]_j}{(\mathsf{iter}(n,e_0,x.e_S), R_0 \oplus R_{\mathrm{ord}(n\otimes\alpha_S)}) \in \mathcal{E}[\![A]\!]_i}$$

for some $j \le i$. This claim follows by an induction on n and routine applications of Lemma 4.3 and Lemma 4.6 to evaluate the n iterations of $\lambda x.e_S$. □

4.2.2 Termination. Given the soundness of the type system, it is straightforward to prove termination of well-typed expressions. The only non-determinism in λ_{CHAN} is in the name of the locations that are chosen during execution, making it sufficient to prove weak normalization:

Corollary 4.12. *If* $\cdot \vdash e : A$, *then* $(e,\cdot) \rightsquigarrow^* (v,h)$ *for some value* v *and heap* h.

Proof Sketch. We have $\cdot \models e : A$ by Theorem 4.5 and hence $\forall i.(e[id], \epsilon \oplus R_{\mathrm{ord}(\alpha)}) \in \mathcal{E}[\![A]\!]_i$ for some ordinal α. We pick $i \triangleq (\alpha, \emptyset)$. Unfolding the definition of $\mathcal{E}[\![A]\!]_i$, the claim follows with $\mathsf{RI}_i(R_{\mathrm{ord}(\alpha)}, \epsilon, \emptyset, \cdot, \emptyset, \emptyset)$, which holds by definition. □

5 RELATED WORK

Termination in the presence of higher-order state, without step-indexing. There is some prior work on proving termination for languages with higher-order state, but in that work the expressiveness of the languages considered was restricted to such an extent that step-indexed logical relations were not required.

Morrisett et al. [2005] use a logical relation without step-indices to prove termination of L^3, a linear language with higher-order mutable references. Although references in L^3 can be aliased arbitrarily, the exclusive right to access and modify a reference, called a *capability*, must be passed around explicitly and may not be duplicated. As a result, L^3 is not expressive enough to implement the asynchronous channels of λ_{CHAN}, since they require *implicit sharing* (multiple references to the same location with the right for modification). For example, consider the following well-typed λ_{CHAN} expression:

$$e_{\mathrm{share}} \triangleq \mathsf{get}(d_{\mathrm{get}}, \lambda x.\mathsf{get}(c_{\mathrm{get}}, \lambda y.\mathsf{print}(x \dotplus y))); \mathsf{get}(d'_{\mathrm{get}}, \lambda x.\mathsf{put}(c_{\mathrm{put}}, x))$$

Here, $c_{\mathrm{get}} : \mathsf{Get}\,\mathbb{N}$ and $c_{\mathrm{put}} : \mathsf{Put}\,\mathbb{N}$ can refer to the same channel, yet individually convey the right to modify its state. The continuations $\lambda x.\mathsf{get}(c_{\mathrm{get}}, \lambda y.\mathsf{print}(x \dotplus y))$ and $\lambda x.\mathsf{put}(c_{\mathrm{put}}, x)$ thus both capture the right to modify the channel c. If the capability to modify a channel were exclusive, as in L^3, then only one of the continuations could capture it and there would be no way to transfer that capability to the other one, especially since in general we do not know which continuation gets invoked first. More generally, if one has to explicitly transfer the capability to access a channel between its sender and receiver ends, then that requires having some separate communication

mechanism M for transferring the capability, which in turn defeats the purpose of using the channel: the values sent on the channel might as well be transferred directly using M.

Boudol [2010] ensures termination in the presence of higher-order state by stratifying memory into different regions. He introduces an effect type system where a region context statically provides a well-founded ordering on the memory regions. Functions stored in the heap which manipulate references in region r may only be stored in a region $r' > r$. For asynchronous channels, static dependency restrictions rule out a number of dynamic characteristics such as *dynamic dependencies between channels* and *dynamic allocation of channels*. For example, consider the function:

$$\lambda b.\mathsf{let}\ (c_{\mathrm{get}}, c_{\mathrm{put}}) = \mathsf{chan}()\ \mathsf{in}\ \mathsf{let}\ (d_{\mathrm{get}}, d_{\mathrm{put}}) = \mathsf{chan}()\ \mathsf{in}$$
$$\mathsf{if}\ b\ \mathsf{then}\ \mathsf{forward}(c_{\mathrm{get}}, d_{\mathrm{put}}); (c_{\mathrm{put}}, d_{\mathrm{get}})\ \mathsf{else}\ \mathsf{forward}(d_{\mathrm{get}}, c_{\mathrm{put}}); (d_{\mathrm{put}}, c_{\mathrm{get}})$$

which allocates two channels c, d and forwards one to the other. Before the argument b is known, the order between the region which contains c and the region which contains d cannot be determined. Furthermore, with a static order on regions, dynamic allocation has to be restricted and it becomes impossible to type:

$$\lambda n.\mathsf{let}\ (c_{\mathrm{get}}, c_{\mathrm{put}}) = \mathsf{chan}()\ \mathsf{in}$$
$$\mathsf{let}\ y = \mathsf{iter}(n, c_{\mathrm{get}}, x.\mathsf{let}\ (d_{\mathrm{get}}, d_{\mathrm{put}}) = \mathsf{chan}()\ \mathsf{in}\ \mathsf{forward}(x, d_{\mathrm{put}}); d_{\mathrm{get}})\ \mathsf{in}\ (y, c_{\mathrm{put}})$$

which creates a chain of channels $c_0, \ldots, c_n$ forwarding a value according to $c_0 \to c_1 \to \cdots \to c_n$. The function cannot be typed if static dependencies between regions are fixed, since a dynamic number of regions would be needed.

Step-indexed logics for termination in the presence of higher-order state. There exist a few step-indexed logics for proving termination of programs in languages with higher-order state, but these logics require the user of the logic to supply termination measures explicitly.

Dockins and Hobor [2010] introduce a step-indexed program logic for proving termination of programs with function pointers. Unlike λ_{CHAN}, their language does not support dynamic allocation of references. In their logic, step-indexed by natural numbers, each function f must be equipped with a termination measure t which maps memory states to natural numbers. While they do give some higher-order examples where they construct the termination measure explicitly, we conjecture that it would be very hard at best to construct a logical relation where the termination measure is inferred compositionally. As explained above, if natural numbers are used, the termination measure is oftentimes closely connected to the semantic behavior of the function. In the present work, we rely on a *transfinite* step-indexing technique which allows us to use ordinals instead of explicit termination measures.

Mével et al. [2019] define *time credits* in the step-indexed program logic Iris [Jung et al. 2015, 2018b]. Time credits allow for proving complexity properties of programs in Iris, including programs with higher-order state, by explicitly counting their steps. But the step count has to be provided explicitly by the user. If one attempts to compute compositionally the number of steps to scale their technique to a logical relation, one encounters the problems discussed in Section 3 since the step-indices of the logic are natural numbers.

Implicit complexity. Girard [1995] introduces the field of implicit complexity where type systems for variants of light linear logic have been shown to only admit programs with polynomial running time. As part of this line of work, Madet and Amadio [2011]; Brunel and Madet [2012] use a clever counting technique for the term size to prove termination in a light linear language with higher-order references. Their proof relies on the property that reduction decreases the size of the expression and heap. In the present work, this property is not satisfied since, *e.g.*,

$\mathsf{iter}(42, (), \_.e_{\text{single}}) \rightsquigarrow \mathsf{iter}(41, e_{\text{single}}, \_.e_{\text{single}})$ does not decrease in size. Brunel and Madet even prove that every program in their language terminates in polynomial time. The language λ_{CHAN} enables computations of *exponential* runtime using the iter construct. That is, we can create an expression of exponential runtime using the function $\exp(m, n)$ from Section 2, making the language strictly more powerful.

The addition of a primitive iteration shows how fragile syntactic approaches based on term size are. Syntactic approaches break for seemingly insignificant changes. For example, if we add the typing rule $\vdash \lambda x.\, (x, x) : \mathbb{N} \multimap \mathbb{N} \otimes \mathbb{N}$ which constitutes a minor violation of linearity, an argument about the term size decreasing for reduction goes out the window. In contrast, our logical relation can cope with this addition. We have $\vDash \lambda x.(x, x) : \mathbb{N} \multimap \mathbb{N} \otimes \mathbb{N}$ since the expression $\lambda x.\, (x, x)$ is contained in $\mathcal{E}[\![\mathbb{N} \multimap \mathbb{N} \otimes \mathbb{N}]\!]_i$ with resource ϵ for all step-indices. In fact, we can add duplication rules for all ground types $G ::= \mathbb{1} \mid \mathbb{B} \mid \mathbb{N} \mid G_1 \otimes G_2$ since, semantically, values of those types own the resource ϵ. More generally, it is semantically sound for any value or expression which owns ϵ (or only invariants) to be duplicated.

Transfinite step-indexing. In the literature, transfinite step-indexing has already been used to define logical relations but not for termination in combination with higher-order state. For example, Schwinghammer et al. [2013] and Bizjak et al. [2014] use step-indexing up to the first uncountable ordinal ω_1 to define a logical relation for reasoning about may and must equivalence of programs with countable non-determinism. Svendsen et al. [2016] use step-indexing up to ω^2 to allow a finite number of decreases of the step-index with each step of computation.

6 CONCLUSIONS AND FUTURE WORK

In the present work, we have used transfinite step-indexing to prove termination of one specific language, the calculus λ_{CHAN}. Naturally, this begs the question: what can be done with transfinite step-indexing beyond proving termination of λ_{CHAN}?

Compilation of reactive programming languages. Over the past decade, a series of papers have studied how *functional reactive programming* (FRP) arises via the Curry-Howard correspondence for temporal logic [Krishnaswami and Benton 2011; Jeltsch 2012; Jeffrey 2012; Jeltsch 2013; Krishnaswami 2013; Cave et al. 2014; Bahr et al. 2019]. The core idea behind these calculi is that proof term assignments for temporal logic offer good programming languages for reactive programming. These languages model interactive programs like GUIs with coinductive resumptions [Cave et al. 2014], which take an input and eventually produce an output and a new resumption representing the new state of the system. The execution of such programs will then invoke these resumptions via an event loop, which of course may not terminate. However, the proof-theoretic properties of the calculus (like strong normalization) ensure that *each* input/output interaction prompted by invoking a resumption *will* terminate.

The normalization properties of the aforementioned reactive programming calculi are typically stated in terms of source-level reductions, but we would like to ensure that these properties continue to hold when we compile reactive programs to lower-level code, in which event-based programming is implemented by means of callbacks and channels. If we knew that termination at the source level ensured termination at the object code level, then the normalization property of the calculi would let us conclude that well-typed interactive programs have finite response: we would know that they would react in finite time to any user input. Transfinite step-indexing seems like a promising candidate for proving this kind of compilation scheme correct.

Transfinite step-indexing for termination. We conjecture that a termination result can still be obtained if λ_{CHAN} is extended with features such as parallelism, (countable) non-determinism,

general higher-order linear references, and inductive data types other than natural numbers, all of which would be natural generalizations for future work. In particular, we believe that, even in the presence of these extensions, ordinals can still be used to compute upper bounds compositionally.

In the present work, the compositional computation of upper bounds is enabled by linearity. What if the language of interest is not linear? We remark that, for entire languages, termination does not hold in general higher-order settings without imposing *some* restrictions, such as (but not necessarily) linearity. Those restrictions are what enable intrinsic termination arguments, as opposed to user-provided termination measures. In the more general case of a non-linear language, the compositional computation of bounds may still be applicable to linear fragments of the language. It would be interesting to develop a type system which combines such compositionally inferred bounds for linear fragments with user-provided bounds for non-linear fragments.

Transfinite step-indexing beyond termination. We believe that transfinitely step-indexed logical relations will have applications in proving other properties for languages with higher-order state that have fallen outside the reach of traditional step-indexed logical relations. These include termination-preserving refinement and progress (*e.g.*, always-eventually) properties.

To answer some of the above questions, we plan to use transfinite step-indexing to develop a transfinite version of the program logic Iris [Jung et al. 2018b]. Such a logic would offer a more abstract and language-independent way of interacting with transfinite step-indices. Furthermore, we plan to define a transfinite notion of time credits in our transfinite version of Iris, which could then be used to establish the termination result of this paper with a simpler, higher-level proof. Transfinite time credits would also allow us to consider termination at the level of individual functions or programs rather than entire languages.

ACKNOWLEDGMENTS

We thank Lars Birkedal, Lennard Gäher, Daniel Gratzer, Robbert Krebbers, and Joseph Tassarotti for helpful discussions about transfinite step-indexing. We additionally thank Alan Schmitt and the anonymous reviewers for their detailed and helpful feedback.

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).

REFERENCES

Amal Ahmed, Andrew W Appel, Christopher D Richards, Kedar N Swadi, Gang Tan, and Daniel C Wang. 2010. Semantic foundations for typed assembly languages. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 32, 3 (2010), 1–67. https://doi.org/10.1145/1709093.1709094

Amal Ahmed, Matthew Fluet, and Greg Morrisett. 2005. A step-indexed model of substructural state. In *Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming* (Tallinn, Estonia) *(ICFP '05)*. Association for Computing Machinery, New York, NY, USA, 78–91. https://doi.org/10.1145/1086365.1086376

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

Andrew W Appel and David McAllester. 2001. An indexed model of recursive types for foundational proof-carrying code. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 23, 5 (2001), 657–683. https://doi.org/10.1145/504709.504712

Patrick Bahr, Christian Uldal Graulund, and Rasmus Ejlers Møgelberg. 2019. Simply RaTT: a Fitch-style modal calculus for reactive programming without space leaks. *Proceedings of the ACM on Programming Languages* 3, ICFP (2019), 1–27. https://doi.org/10.1145/3341713

Aleš Bizjak, Lars Birkedal, and Marino Miculan. 2014. A model of countable nondeterminism in guarded type theory. In *Proceedings of TLCA*. https://doi.org/10.1007/978-3-319-08918-8_8

Gérard Boudol. 2010. Typing termination in a higher-order concurrent imperative language. *Information and Computation* 208, 6 (2010), 716–736. https://doi.org/10.1016/j.ic.2009.06.007

Aloïs Brunel and Antoine Madet. 2012. Indexed realizability for bounded-time programming with references and type fixpoints. In *Asian Symposium on Programming Languages and Systems*. Springer, 264–279. https://doi.org/10.1007/978-3-642-35182-2_19

Andrew Cave, Francisco Ferreira, Prakash Panangaden, and Brigitte Pientka. 2014. Fair reactive programming. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '14)*. ACM, San Diego, California, USA, 361–372. https://doi.org/10.1145/2535838.2535881

Koen Claessen. 1999. A poor man's concurrency monad. *Journal of Functional Programming* 9, 3 (1999), 313–323. https://doi.org/10.1017/S0956796899003342

Pedro da Rocha Pinto, Thomas Dinsdale-Young, Philippa Gardner, and Julian Sutherland. 2016. Modular termination verification for non-blocking concurrency. In *European Symposium on Programming*. Springer, 176–201. https://doi.org/10.1007/978-3-662-49498-1_8

Robert Dockins and Aquinas Hobor. 2010. A theory of termination via indirection. In *Dagstuhl Seminar Proceedings*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

Robert Dockins and Aquinas Hobor. 2012. Time bounds for general function pointers. *Electronic Notes in Theoretical Computer Science* 286 (2012), 139–155. https://doi.org/10.1016/j.entcs.2012.08.010

Derek Dreyer, Amal Ahmed, and Lars Birkedal. 2011. Logical step-indexed logical relations. *Logical Methods in Computer Science* 7, 2:16 (2011). https://doi.org/10.2168/LMCS-7(2:16)2011

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

Daniel Friedman and David Wise. 1976. The impact of applicative programming on multiprocessing. In *International Conference on Parallel Processing*. 263–272.

Jean-Yves Girard. 1995. Light linear logic. In *Logic and Computational Complexity*, Daniel Leivant (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 145–176. https://doi.org/10.1007/3-540-60178-3_83

Jean-Yves Girard, Paul Taylor, and Yves Lafont. 1989. *Proofs and types*. Cambridge University Press.

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

Alan Jeffrey. 2012. LTL types FRP: linear-time temporal logic propositions as types, proofs as functional reactive programs. In *Proceedings of the sixth workshop on Programming Languages meets Program Verification, PLPV 2012, Philadelphia, PA, USA, January 24, 2012*. Philadelphia, PA, USA, 49–60. https://doi.org/10.1145/2103776.2103783

Wolfgang Jeltsch. 2012. Towards a common categorical semantics for linear-time temporal logic and functional reactive programming. *Electronic Notes in Theoretical Computer Science* 286 (2012), 229–242. https://doi.org/10.1016/j.entcs.2012.08.015

Wolfgang Jeltsch. 2013. Temporal logic with "until", functional reactive programming with processes, and concrete process categories. In *Proceedings of the 7th Workshop on Programming Languages Meets Program Verification* (Rome, Italy) *(PLPV '13)*. ACM, New York, NY, USA, 69–78. https://doi.org/10.1145/2428116.2428128

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

Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. 2018b. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming* 28 (2018). https://doi.org/10.1017/S0956796818000151

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*. ACM New York, NY, USA, 637–650. https://doi.org/10.1145/2676726.2676980

Neelakantan R. Krishnaswami. 2013. Higher-order functional reactive programming without spacetime leaks. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming (ICFP '13)*. ACM, Boston, Massachusetts, USA, 221–232. https://doi.org/10.1145/2500365.2500588

Neelakantan R. Krishnaswami and Nick Benton. 2011. Ultrametric semantics of reactive programs. In *2011 IEEE 26th Annual Symposium on Logic in Computer Science*. IEEE Computer Society, Washington, DC, USA, 257–266. https://doi.org/10.1109/LICS.2011.38

Neelakantan R Krishnaswami, Aaron Turon, Derek Dreyer, and Deepak Garg. 2012. Superficially substructural types. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*. 41–54. https://doi.org/10.1145/2398856.2364536

Peter J Landin. 1964. The mechanical evaluation of expressions. *Comput. J.* 6, 4 (1964), 308–320. https://doi.org/10.1093/comjnl/6.4.308

Antoine Madet and Roberto M Amadio. 2011. An elementary affine λ-calculus with multithreading and side effects. In *International Conference on Typed Lambda Calculi and Applications*. Springer, 138–152. https://doi.org/10.1007/978-3-642-21691-6_13

Glen Mével, Jacques-Henri Jourdan, and François Pottier. 2019. Time credits and time receipts in Iris. In *European Symposium on Programming*. Springer, 3–29. https://doi.org/10.1007/978-3-030-17184-1_1

Greg Morrisett, Amal Ahmed, and Matthew Fluet. 2005. L^3: A linear language with locations. In *Typed Lambda Calculi and Applications*, Paweł Urzyczyn (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 293–307. https://doi.org/10.1007/11417170_22

John C Reynolds. 1983. Types, abstraction and parametric polymorphism. In *Information Processing 83, Proceedings of the IFIP 9th World Computer Congress*. 513–523.

Jan Schwinghammer, Aleš Bizjak, and Lars Birkedal. 2013. Step-indexed relational reasoning for countable nondeterminism. *Logical Methods in Computer Science* 9 (2013). https://doi.org/10.2168/LMCS-9(4:4)2013

Simon Spies, Neel Krishnaswami, and Derek Dreyer. 2021. *Transfinite step-indexing for termination*. Technical Report. https://plv.mpi-sws.org/transfinite-step-indexing/termination/

Kasper Svendsen and Lars Birkedal. 2014. Impredicative concurrent abstract predicates. In *Proceedings of ESOP*. https://doi.org/10.1007/978-3-642-54833-8_9

Kasper Svendsen, Filip Sieczkowski, and Lars Birkedal. 2016. Transfinite step-indexing: Decoupling concrete and logical steps. In *European Symposium on Programming*. Springer, 727–751. https://doi.org/10.1007/978-3-662-49498-1_28

William W Tait. 1967. Intensional interpretations of functionals of finite type I. *Journal of Symbolic Logic* 32, 2 (1967), 198–212. https://doi.org/10.2307/2271658

Aaron J Turon, Jacob Thamsborg, Amal Ahmed, Lars Birkedal, and Derek Dreyer. 2013. Logical relations for fine-grained concurrency. In *POPL*. ACM New York, NY, USA, 343–356. https://doi.org/10.1145/2480359.2429111

Nobuko Yoshida, Martin Berger, and Kohei Honda. 2004. Strong normalisation in the π-calculus. *Information and Computation* 191, 2 (2004), 145–202. https://doi.org/10.1016/j.ic.2003.08.004