A Generalized Modality for Recursion

Adrien Guatto
University of Bamberg
adrien@guatto.org

Abstract

Nakano's *later* modality allows types to express that the output of a function does not immediately depend on its input, and thus that computing its fixpoint is safe. This idea, guarded recursion, has proved useful in various contexts, from functional programming with infinite data structures to formulations of step-indexing internal to type theory. Categorical models have revealed that the later modality corresponds in essence to a simple reindexing of the discrete time scale.

Unfortunately, existing guarded type theories suffer from significant limitations for programming purposes. These limitations stem from the fact that the later modality is not expressive enough to capture precise input-output dependencies of functions. As a consequence, guarded type theories reject many productive definitions.

Combining insights from guarded type theories and synchronous programming languages, we propose a new modality for guarded recursion. This modality can apply any well-behaved reindexing of the time scale to a type. We call such reindexings *time warps*. Several modalities from the literature, including later, correspond to fixed time warps, and thus arise as special cases of ours.

Keywords Guarded Recursion; Functional Programming; Streams; Type Systems; Category Theory; Synchronous Programming.

1 Introduction

Consider the following piece of pseudocode.

$$\mathrm{nat} = \textbf{fix}\ \mathrm{natrec}\ \textbf{where}\ \mathrm{natrec\ xs} = 0 :: (\mathrm{map}\ (\lambda \mathrm{x}.\mathrm{x} + 1)\ \mathrm{xs})$$

This defines nat, the stream of natural numbers, as the fixpoint of a function natrec. How does one make sure that this definition is *productive*, in the sense that the next element of nat can always be computed in finite time?

Guarded recursion, due to Nakano [27], provides a type-based answer to this question. In type systems such as Nakano's, types capture precedence relationships between pieces of data, expressed with respect to an implicit discrete time scale. For example, natrec would receive the type natrec : ► **Stream Int** → **Stream Int**. The type **Stream Int** describes streams which unfold in time at the rate of one new element per step. The *later* (►) modality shifts the type it is applied to one step into the future; thus, ► **Stream Int** also unfolds at the rate of one element per step, but only starts unfolding after the first step. Hence, the type of natrec expresses that the *n*th element of its output stream, which is produced at the *n*th step, does not depend on the *n*th element of its input stream, since the latter arrives at the $(n + 1)$th step. This absence of *instantaneous* input-output dependence guarantees the productivity of **fix** natrec. Guarded recursion enforces this uniformly: fixpoints are restricted to functions with a type of the form $\blacktriangleright \tau \to \tau$.

Nakano's original insight has led to a flurry of proposals [3, 4, 6–8, 13, 20, 21, 26, 31]. Recent developments have integrated several advances—such as *clock variables* [3] or the *constant* (■) modality [13]—into expressive languages capturing many recursive definitions out of reach of more syntactic productivity checks. The *topos of trees* [7] provides an elegant categorical setting where such languages find their natural home.

Unfortunately, guarded recursion is currently limited by the inability of existing languages to capture fine-grained dependencies. Consider the following function, which returns a pair of streams.

$$\mathrm{natposrec} = \textbf{fun}\ (\mathrm{xs}, \mathrm{ys}).(0 :: \mathrm{ys}, \mathrm{map}\ (\lambda \mathrm{x}.\mathrm{x} + 1)\ \mathrm{xs})$$

Its fixpoint is productive. This can be seen in the table below, which gives the first iterations of $(\mathrm{nat}, \mathrm{pos}) = \mathrm{natposrec}(\mathrm{nat}, \mathrm{pos})$.

$$\begin{array}{c|cccccc} \mathrm{nat} & \bot & 0 :: \bot & 0 :: \bot & 0 :: 1 :: \bot & 0 :: 1 :: \bot & \dots \\ \mathrm{pos} & \bot & \bot & 1 :: \bot & 1 :: \bot & 1 :: 2 :: \bot & \dots \end{array}$$

Each stream grows infinitely often but only by one element every two steps. The later modality, by itself, cannot capture this growth pattern, and thus this program cannot be expressed as is in the guarded languages we know of. Since natposrec is simply natrec modified to expose the result of a subterm, this shows that existing systems can be overly rigid. Clouston et al. [13, p. 12] give other examples hampered by similar problems.

In our view, the example above does not indicate a problem with guarded recursion *per se*, but rather illustrates the need for other temporal modalities beyond later (and constant). Like later and constant, these new modalities would apply temporal transformations onto types, *reindexing* them to change how much data is available at each step. In the example above, one would use a modality expressing growth at even time steps, and another for growth at odd time steps. Moreover, these new modalities should be interrelated, generalizing the known interactions between later and constant.

Contribution In this paper, we propose a theory of temporal modalities subsuming later and constant. Rather than studying a fixed number of modalities, we merge all of them into a single modality $*$ parameterized by well-behaved reindexings of the discrete time scale. We call such reindexings *time warps* and speak of the *warping modality*. The later and constant modalities correspond to specific time warps, and thus arise as special cases of $*$.

We build a simply-typed λ-calculus, Core λ^*, around the warping modality. Core λ^* integrates a notion of subtyping which internalizes the mathematical structure of time warps. We describe its operational semantics, as well as a denotational semantics in the topos of trees. We show that the type-checking problem for Core λ^* terms, while delicate because of subtyping, is actually decidable.

LICS '18, July 9–12, 2018, Oxford, United Kingdom

2 The Calculus

2.1 Time Warps

Let ω denote the first infinite ordinal and $\omega + 1$ denote its successor, which extends ω with a maximal element ω. It is technically convenient to see 0 as a special vacuous time step, and thus we assume that ω begins at 1. (Clouston et al. [13] follow the same convention.) However, $\omega + 1$ still begins with 0. If P is a preorder, we denote by $\widehat{P}$ the set of its downward-closed subsets ordered by inclusion, and write $\mathbf{y} : P \longrightarrow \widehat{P}$ for the order embedding sending x to $\{x' \mid x' \leq x\}$. Observe that $\widehat{\omega}$ is isomorphic to $\omega + 1$, with the isomorphism sending the empty subset to 0 and the maximal subset ω to itself in $\omega + 1$. Thus, abusing notation, we write $\mathbf{y} : \omega \longrightarrow \omega + 1$ for the map sending positive natural numbers to their image in $\omega + 1$.

Definition 1 (Time Warps)**.** *A time warp is a cocontinuous (sup-preserving) function from $\omega + 1$ to itself. Equivalently, it is a monotonic function $p : \omega + 1 \longrightarrow \omega + 1$ such that $p(0) = 0$ and $p(\omega) = \bigsqcup_{n<\omega} p(n)$.*

We write $p \leq q$ when p is pointwise smaller than q, that is, when $p(n) \leq q(n)$ holds for all n. Given time warps p and q, we write $p * q$ for $q \circ p$, which is cocontinuous. So is the identity function. Moreover, function composition is left- and right-monotonic for the pointwise order. As a consequence,

Property 1. *Time warps, ordered pointwise and equipped with composition, form a partially-ordered monoid, denoted $\mathcal{W}$.*

The following time warps play a special role in our development.

$$\underline{id}(n) = n \qquad \underline{0}(n) = 0 \qquad \underline{-1}(n) = n - 1 \qquad \underline{\omega}(n) = \omega$$

The definitions above are given for $0 < n < \omega$ since the values at 0 and ω follow from cocontinuity. The time warps $\underline{0}$ and $\underline{\omega}$ are respectively the least and greatest elements of $\mathcal{W}$.

2.2 Syntax and Declarative Type System

Core λ^* is a two-level calculus distinguishing between *implicit terms* and *explicit terms*. Implicit terms correspond to source-level programs. Explicit terms decorate implicit terms with type coercions. Coercions act as proof terms for the subtyping judgment [9, 16]. They offer a convenient alternative to the manipulation of typing derivations in non-syntax-directed type systems such as ours.

Ground types and scalars We assume given a finite set of ground types G and a family of pairwise disjoint sets $(S_\nu)_{\nu \in G}$. The elements of S_ν are the scalars (ground values) of type $\nu \in G$. We denote by s the elements of $S \triangleq \bigcup_{\nu \in G} S_\nu$.

Types The types of Core λ^* are those of simply-typed λ-calculus, including products and sums, together with ground types, streams, and our warping modality $*_p$:

$$\tau ::= \nu \mid \mathbf{Stream}\ \tau \mid \tau \to \tau \mid \tau \times \tau \mid \tau + \tau \mid *_p\ \tau.$$

Informally, $*_p\ \tau$ should be seen as a "p-times" faster version of τ, in the sense of providing p-times more data than τ per step, with the caveat that if p is less than $\underline{id}$, $*_p\ \tau$ is actually "slower" than τ.

Typing Contexts Typing contexts are lists of bindings $x_i : \tau_i$ with the x_i pairwise distinct. We use "$\cdot$" for the empty context and $\mathrm{dom}(\Gamma)$ for the finite set of variables present in Γ. We write $\Gamma(x)$ for the unique τ such that $(x : \tau)$ occurs in Γ, if it exists.

$$\boxed{\Gamma \vdash e : \tau} \qquad \textsc{Var}\ \frac{}{\Gamma, x : \tau \vdash x : \tau} \qquad \textsc{Fun}\ \frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \mathbf{fun}\,(x : \tau_1).e : \tau_1 \to \tau_2}$$

$$\textsc{App}\ \frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2} \qquad \textsc{Pair}\ \frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2} \qquad \textsc{Proj}_{i \in \{1,2\}}\ \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \mathbf{proj}_i\ e : \tau_i}$$

$$\textsc{Inj}_{i \in \{0,1\}}\ \frac{\Gamma \vdash e : \tau_{1+i}}{\Gamma \vdash \mathbf{inj}_{1+i}^{\tau_{2-i}}\ e : \tau_1 + \tau_2} \qquad \textsc{Case}\ \frac{\Gamma \vdash e : \tau_1 + \tau_2 \qquad \Gamma, x_i : \tau_i \vdash e_i : \tau \text{ for } i \in \{1,2\}}{\Gamma \vdash \mathbf{case}\ e\ \mathbf{of}\,\{\mathbf{inj}_1\ x_1.e_1 \mid \mathbf{inj}_2\ x_2.e_2\} : \tau}$$

$$\textsc{Head}\ \frac{\Gamma \vdash e : \mathbf{Stream}\ \tau}{\Gamma \vdash \mathbf{head}\ e : \tau} \qquad \textsc{Tail}\ \frac{\Gamma \vdash e : \mathbf{Stream}\ \tau}{\Gamma \vdash \mathbf{tail}\ e : *_{\underline{-1}}\ \mathbf{Stream}\ \tau} \qquad \textsc{Cons}\ \frac{\Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : *_{\underline{-1}}\ \mathbf{Stream}\ \tau}{\Gamma \vdash e_1 :: e_2 : \mathbf{Stream}\ \tau}$$

$$\textsc{Const}\ \frac{s \in S_\nu}{\Gamma \vdash s : \nu} \qquad \textsc{Rec}\ \frac{\Gamma, x : *_{\underline{-1}}\ \tau \vdash e : \tau}{\Gamma \vdash \mathbf{rec}\ (x : \tau).e : \tau} \qquad \textsc{Warp}\ \frac{\Gamma \vdash e : \tau}{*_p\ \Gamma \vdash e\ \mathbf{by}\ p : *_p\ \tau}$$

$$\textsc{SubR}\ \frac{\Gamma \vdash e : \tau \qquad \alpha : \tau <: \tau'}{\Gamma \vdash e; \alpha : \tau'} \qquad \textsc{SubL}\ \frac{\beta : \Gamma <: \Gamma' \qquad \Gamma' \vdash e : \tau}{\Gamma \vdash \beta; e : \tau} \qquad \textsc{Struct}\ \frac{\Gamma \vdash e : \tau \qquad \sigma \in \Sigma(\Gamma; \Gamma')}{\Gamma' \vdash \sigma[e] : \tau}$$

Figure 1. Typing Judgment

Explicit Terms The typing judgment for explicit terms e of Core λ^* is given in Figure 1. Every typing rule from Var to Case is a standard one from simply-typed λ-calculus with products and sums. We describe every other rule in turn, introducing the corresponding term formers as we go.

The typing rules for stream destructors (**head** e, **tail** e) and the stream constructor ($e_1 :: e_2$) capture the fact that streams unfold at the rate of one element per step. As a consequence, the tail of a stream exists not now but *later.* Since the later modality corresponds in our setting to the time warp $\underline{-1}$, the result of **tail** and the second argument of (::) must be of type $*_{\underline{-1}}\ \mathbf{Stream}\ \tau$.

Core λ^* terms include scalars from S. A scalar s is assigned the unique ground type ν such that $s \in S_\nu$, as specified in rule Const.

Recursive definitions $\mathbf{rec}\ (x : \tau).e$ follow the insight of Nakano: the self-reference to x is only available later in the body e, and thus here receives type $*_{\underline{-1}}\ \tau$ in rule Rec.

The term e **by** p marks an introduction point for the warping modality. Intuitively, it runs e in a local time scale whose relationship to the surrounding time scale is goverened by the time warp p: the nth tick of the external time scale corresponds to the $p(n)$th tick of the internal one. Thus, assuming e has type τ, e **by** p has type $*_p\ \tau$. This change in the amount of data produced comes at the price of a change in the amount of data consumed: the free variables of e should themselves be under the $*_p$ modality. The context $*_p\ \Gamma$ denotes Γ with $*_p$ applied to each of its types.

Explicit terms may include type coercions, applied either covariantly or contravariantly. Covariant coercion application $e; \alpha$ applies the type coercion α to the result of e. Contravariant coercion application $\beta; e$ coerces the free variables of e using the context coercion β. We will describe both kinds of coercions in a few paragraphs.

Structure maps Rule Struct is the only non-syntax-directed rule in our system. It performs weakening, contraction, and exchange in a single step, depending on the chosen *structure map* between contexts [2, 15]. Structure maps $\sigma \in \Sigma(\Gamma; \Gamma')$ are functions from $\mathrm{dom}(\Gamma)$ to $\mathrm{dom}(\Gamma')$ such that $\Gamma'(\sigma(x)) = \Gamma(x)$ for all $x \in \Gamma$.

$$\boxed{\alpha : \tau <: \tau'} \qquad \frac{}{\mathbf{id} : \tau <: \tau} \qquad \frac{\alpha_1 : \tau_1 <: \tau_2 \qquad \alpha_2 : \tau_2 <: \tau_3}{\alpha_1 ; \alpha_2 : \tau_1 <: \tau_3}$$

$$\frac{\alpha : \tau <: \tau'}{\mathbf{Stream}\ \alpha : \mathbf{Stream}\ \tau <: \mathbf{Stream}\ \tau'} \qquad \frac{\alpha_1 : \tau_1' <: \tau_1 \qquad \alpha_2 : \tau_2 <: \tau_2'}{\alpha_1 \to \alpha_2 : \tau_1 \to \tau_2 <: \tau_1' \to \tau_2'}$$

$$\frac{\alpha_1 : \tau_1 <: \tau_1' \qquad \alpha_2 : \tau_2 <: \tau_2'}{\alpha_1 \times \alpha_2 : \tau_1 \times \tau_2 <: \tau_1' \times \tau_2'} \qquad \frac{\alpha_1 : \tau_1 <: \tau_1' \qquad \alpha_2 : \tau_2 <: \tau_2'}{\alpha_1 + \alpha_2 : \tau_1 + \tau_2 <: \tau_1' + \tau_2'}$$

$$\frac{\alpha : \tau <: \tau'}{\circledast_p\, \alpha : \circledast_p\, \tau <: \circledast_p\, \tau'} \qquad \frac{}{(\mathbf{wrap}, \mathbf{unwrap}) : \tau \equiv \circledast_{\underline{id}}\, \tau}$$

$$\frac{}{(\mathbf{concat}^{p,q}, \mathbf{decat}^{p,q}) : \circledast_p \circledast_q\, \tau \equiv \circledast_{p * q}\, \tau} \qquad \frac{}{\mathbf{inflate} : \nu <: \circledast_{\underline{\omega}}\, \nu}$$

$$\frac{}{(\mathbf{dist}_\times, \mathbf{fact}_\times) : \circledast_p\, (\tau_1 \times \tau_2) \equiv \circledast_p\, \tau_1 \times \circledast_p\, \tau_2} \qquad \frac{q \leq p}{\mathbf{delay}^{p,q} : \circledast_p\, \tau <: \circledast_q\, \tau}$$

Figure 2. Subtyping Judgment

The application of a structure map σ, seen as a variable substitution, to an explicit term e is written $\sigma[e]$.

Type annotations Both λ-abstractions and injections must contain type annotations. This technical choice ensures that explicit terms are in Church style, and makes their typing judgment essentially syntax directed (up to rule STRUCT).

Type Coercions A coercion $\alpha : \tau <: \tau'$ performs a type conversion, transforming input values of type τ into output values of type τ'. The rules for this syntax-directed subtyping judgment are given in Figure 2, where $(\alpha, \alpha') : \tau \equiv \tau'$ is a shorthand for $\alpha : \tau <: \tau'$ and $\alpha' : \tau' <: \tau$. They fit into three groups.

The first group contains the identity coercion and sequential coercion composition. The identity coercion **id** does nothing. Two coercions α_1 and α_2 can be composed to obtain $\alpha_1; \alpha_2$, assuming the output type of α_1 matches the input type of α_2.

The second group contains one coercion former for each type former. Such coercions allow us to coerce values in depth. Their typing rules express that subtyping is a congruence for all type formers in the language.

The third group is where the interest of our subtyping relationship lies. It contains coercions reflecting the mathematical structure of the warping modality as subtyping axioms, including its interaction with other type formers. This group can be divided again, now between several invertible coercions and a non-invertible one.

- Coercions **wrap**, **unwrap**, $\mathbf{concat}^{p,q}$, and $\mathbf{decat}^{p,q}$ reflect the monoidal structure of time warps at the type level. The coercions $\mathbf{dist}_\times$ and $\mathbf{fact}_\times$ ensure that the warping modality commutes with products. The coercion **inflate** expresses that ground types stay constant through time, i.e., $\nu <: \circledast_{\underline{\omega}}\, \nu$.
- The remaining coercion, $\mathbf{delay}^{p,q} : \circledast_p\, \tau <: \circledast_q\, \tau$, reflects the ordering of time warps. Intuitively, it pushes data further into the future, and must thus ensure that $p(n) \geq q(n)$ at any step n. Its action cannot be undone when $p \neq q$; for example, the coercion $\mathbf{next} \triangleq \mathbf{wrap}; \mathbf{delay}^{\underline{id}, \underline{-1}} : \tau <: \circledast_{\underline{-1}}\, \tau$ has no inverse. (This coercion appears an an operator in some guarded type theories [7, 13, . . .].)

Note that we did not need to introduce an explicit inverse for **inflate** since one is already derivable as $\mathbf{delay}^{\underline{\omega}, \underline{id}}; \mathbf{unwrap} : \circledast_{\underline{\omega}}\, \nu <: \nu$.

Context Coercions A context coercion β is a finite map from variables to coercions. We have $\beta : \Gamma <: \Gamma'$ iff $\mathrm{dom}(\Gamma) = \mathrm{dom}(\Gamma') \subseteq \mathrm{dom}(\beta)$, and for every variable $x \in \mathrm{dom}(\Gamma)$, $\beta(x)$ coerces $\Gamma(x)$ into $\Gamma'(x)$. Context subtyping preserves the order of bindings. This definition implies that rule SUBL in Figure 1 can only be applied when $\beta(x)$ is defined for every free variable x of e.

Implicit Terms and Erasure We define implicit terms, denoted t, as explicit terms that do not contain any coercions. Each explicit term e thus corresponds to a unique implicit term obtained by removing every coercion present in e. We adopt the notations of Melliès and Zeilberger [24], and write $\mathbf{U}(e)$ for this implicit term. We also write that e *refines* t, noted $e \sqsubset t$, when $\mathbf{U}(e) = t$.

An implicit term is well-typed simply if it has a well-typed refiner, in the sense expressed by the definition below.

$$\boxed{\Gamma \vdash t : \tau} \quad \Leftrightarrow \quad \exists e \sqsubset t, \Gamma \vdash e : \tau \tag{1}$$

2.3 Type-Checking Explicit Terms

The language of coercions and explicit terms enjoys uniqueness of typing. The following result reflects this fact for coercions.

Property 2 (Uniqueness of Types for Coercions)**.** *For any coercion α, for any type τ (resp. τ') there is at most one type τ' (resp. τ) such that $\alpha : \tau <: \tau'$ holds.*

The analogous result for explicit terms is more delicate since rule STRUCT is not syntax-directed. Furthermore, the rule is not admissible: its use is sometimes required in order to be able to use another rule. One can prove that there are only three cases where rule STRUCT is needed, establishing the following result.

Property 3. *Any well-typed explicit term $\Gamma \vdash e : \tau$ has a canonical derivation where rule STRUCT is only used exactly once before every instance of rules VAR, WARP, and SUBL.*

This characterization provides almost immediately an abstract deterministic algorithm to type-check explicit terms; see the appendix for details. Its correctness implies the expected result.

Property 4 (Uniqueness of Types for Explicit Terms)**.** *For any fixed Γ and e, there is at most one type τ such that $\Gamma \vdash e : \tau$ holds.*

Type-checking an *implicit* term t is a much harder problem, since it involves finding a well-typed refiner $e \sqsubset t$. Moreover, this refiner should be canonical in a certain sense. We study it in Section 5.

2.4 Examples

We finish this section with a few examples illustrating the type system, given mostly as implicit terms. We also assume that ground values and types include the integers.

Example 2.1 (Constant Stream)**.** This prototypical example defines a constant stream of zeroes.

$$\mathbf{rec}\ (\mathrm{zeroes} : \mathbf{Stream\ Int}).(0 :: \mathrm{zeroes}) : \mathbf{Stream\ Int}$$

This works as in other guarded-recursive languages: the stream constructor (::) expects its second argument to have a type of the form $\circledast_{\underline{-1}}\, \mathbf{Stream}\ \tau$ ($\blacktriangleright \mathbf{Stream}\ \tau$ in other guarded type theories), which is exactly the one provided by guarded recursion.

Example 2.2 (Non-productive Stream)**.** The non-productive definition below does not define a stream, which by definition should hold an infinite number of values.

$$\mathbf{rec}\ (\mathrm{nothing} : \mathbf{Stream\ Int}).\mathrm{nothing} \ – \text{ill-typed!}$$

This definition is ill-typed in Core λ^* since, in the absence of a coercion $\ast_{\underline{-1}}$ **Stream Int** <: **Stream Int**, we cannot apply rule Rec.

Example 2.3 (Silent Stream). While Example 2.2 does not define a real stream holding an infinite number of values, it could be said to define a "silent" stream holding no value at all. Such streams are captured in Core λ^* as inhabitants of $\ast_{\underline{0}}$ **Stream** τ, e.g.,

$$\textbf{rec}\ (\mathrm{nothing} : \ast_{\underline{0}}\ \textbf{Stream Int}).\mathrm{nothing} : \ast_{\underline{0}}\ \textbf{Stream Int}.$$

This definition is well-typed since the explicit term **rec** (nothing : $\ast_{\underline{0}}$ **Stream Int**).(nothing; $\textbf{concat}^{\underline{-1},\underline{0}}$) refines it and has the expected type. Here, $\textbf{concat}^{\underline{-1},\underline{0}}$ coerces values of type $\ast_{\underline{-1}} \ast_{\underline{0}}$ **Stream Int** to values of type $\ast_{\underline{-1} * \underline{0}}$ **Stream Int** = $\ast_{\underline{0}}$ **Stream Int**.

Example 2.3 illustrates how Core λ^* shifts the focus away from productivity, seen as a yes-or-no question, to a more quantitative aspect of program execution: the amount of data produced. Other warps make it possible to capture other forms of partial definitions, beyond completely silent streams. For example, writing $\underline{5}$ for the warp sending any finite n to 5, the type $\ast_{\underline{5}}$ **Stream Int** describes streams containing only 5 elements, all of them available starting at the first time step. The type system of Core λ^* ensures that the non-existent elements in such partial streams can never be accessed; in particular, in a well-typed program deconstructing a silent stream xs (via **head** or **tail**) can only happen under a context of the form $C_1[C_2[-]$ **by** $\underline{0}]$. We will see in Section 3 that the expression e in e **by** $\underline{0}$ is never actually executed.

Example 2.4. The example below implements the classic higher-order function map on streams, specialized for streams of integers since Core λ^* is monomorphic. As usual, **let** $x : \tau = t_1$ **in** t_2 is shorthand for (**fun** $(x : \tau).t_2)\ t_1$. We assume that function application and **by** bind tighter than stream construction (::).

rec (map : (**Int** → **Int**) → **Stream Int** → **Stream Int**).
fun (f : **Int** → **Int**) (xs : **Stream Int**).
let ys : $\ast_{\underline{-1}}$ **Stream Int** = **tail** xs **in** f (**head** xs) :: (map f ys) **by** $\underline{-1}$

Here, using **by** allows us to temporarily remove the $\ast_{\underline{-1}}$ modality from the type of map in order to perform the recursive call. Rule Warp requires map, ys, and f to have types of the form $\ast_{\underline{-1}}\,\tau$. This is already the case for map and ys, and can be achieved for f using rule SubL with a context coercion sending f to **next** : **Int** → **Int** <: $\ast_{\underline{-1}}$ (**Int** → **Int**) and the other variables to **id**.

Example 2.5. The definition given in Example 2.4, since it is closed, can be put inside a local time scale driven by $\underline{\omega}$. It thus receives the type $\ast_{\underline{\omega}}$ ((**Int** → **Int**) → **Stream Int** → **Stream Int**). Such a type is in effect not subject to the context restriction in rule Warp, since for any p we have $\ast_{\underline{\omega}}\,\tau$ <: $\ast_p \ast_{\underline{\omega}}\,\tau$. Thus, $\ast_{\underline{\omega}}$ corresponds to the *constant* (■) modality used in some guarded type theories [13].

In the remaining examples, we represent certain time warps as running sums of ultimately periodic sequences of numbers, following ideas from n-synchrony [14, 29]. For example, the sequence $(1\,0)^\infty$ represents the time warp sending $2n$ to n and $2n+1$ to $n+1$ for any finite positive n, while the sequence $(0\,1)^\infty$ represents the time warp sending both $2n$ and $2n+1$ to n. All the time warps we have used up to now can be represented in this way: $\underline{id}$, $\underline{0}$, $\underline{-1}$, and $\underline{\omega}$ are represented by $(1)^\infty$, $(0)^\infty$, $0\,(1)^\infty$, and $\omega\,(0)^\infty$ respectively.

Example 2.6 (Mutual Recursion). As announced in Section 1, the streams of natural and positive numbers can be defined in a guarded yet mutually-recursive way in Core λ^*. This is achieved by reflecting the rate at which each stream grows during a fixpoint computation within its type. (In the definition below, we represent the time warp $\underline{-1}$ by the sequence $0\,(1)^\infty$ for consistency; in particular, the types of (::) becomes $\tau \to \ast_{0\,(1)^\infty}$ **Stream** $\tau \to$ **Stream** τ.)

rec natpos : $\ast_{(1\,0)^\infty}$ **Stream Int** × $\ast_{(0\,1)^\infty}$ **Stream Int**.
let nat : $\ast_{0\,(1)^\infty} \ast_{(1\,0)^\infty}$ **Stream Int** = $\textbf{proj}_1$ natpos **in**
let pos : $\ast_{0\,(1)^\infty} \ast_{(0\,1)^\infty}$ **Stream Int** = $\textbf{proj}_2$ natpos **in**
((0 :: pos) **by** $(1\,0)^\infty$, (map (**fun** (x : **Int**).x + 1) nat) **by** $(0\,1)^\infty$

The uses of projections are well-typed since the warping modality distributes over products via the $\textbf{dist}_\times$ coercion. We assume that map has received the type given in Example 2.5, and thus its use below **by** $(0\,1)^\infty$ is well-typed. Since $0\,(1)^\infty * (1\,0)^\infty = (0\,1)^\infty$, coercing nat by $\textbf{concat}^{0\,(1)^\infty,(1\,0)^\infty}$ gives the type $\ast_{(0\,1)^\infty}$ **Stream Int**, which lets us use nat with type **Stream Int** under **by** $(0\,1)^\infty$. For pos, since $0\,(1)^\infty * (0\,1)^\infty = 0\,(0\,1)^\infty = (1\,0)^\infty * 0\,(1)^\infty$, applying the coercion $\textbf{concat}^{0\,(1)^\infty,(0\,1)^\infty}; \textbf{decat}^{(1\,0)^\infty,0\,(1)^\infty}$ lets us use pos with type $\ast_{0\,(1)^\infty}$ **Stream Int** below **by** $(1\,0)^\infty$.

Example 2.7. Clouston et al. [13, Example 1.10] present the *Thue-Morse sequence* as a recursive stream definition which is difficult to capture in guarded calculi. The productivity of this definition follows from the fact that a certain auxiliary stream function h produces two new elements of its output stream for each new element of its input stream. In Core λ^*, h can be given type **Stream Bool** → $\ast_{(2)^\infty}$ **Stream Bool**, allowing us to implement the Thue-Morse sequence with guarded recursion. (See the appendix.)

3 Operational Semantics

In this section, we present an operational semantics for explicit terms in the form of a big-step, call-by-value evaluation judgment. Intuitively, the evaluation judgment $e; \gamma \Downarrow_n v$ expresses that the value v is a finite prefix of length n of the possibly infinite result computed by e in the environment γ. We will say that the evaluation of e occurred "at step n", or simply "at n" following the intuition that n is a Kripke world. Another intuition is that this judgment describes an interpreter receiving a certain amount n of "fuel" which controls how many times recursive definitions have to be unrolled [1].

In most fuel-based definitional interpreters, the fuel parameter only decreases along evaluation, typically by one unit at each recursive unfolding. In our case, its evolution is much less constrained: the amount of fuel may actually increase or decrease by an arbitrary amount many times during the execution of a single term. This behavior follows from the presence of time warps: to evaluate e **by** p at n, one evaluates e at $p(n)$. Nevertheless, we show that the evaluation of a well-typed term always terminates regardless of the quantity of fuel initially provided.

Since the evaluation of a term at n might involve the evaluation of one of its subterms at $p(n)$ with p an arbitrary warp, we may need to evaluate a term at 0 or ω. The former case is dealt with using a dummy value **stop** which inhabits all types at 0. The latter case might seem problematic, as evaluating a term at ω should intuitively result in an infinite object rather than a finite one. We represent such results by suspended computations (*thunks*) to be forced only when used at a finite n. This is a standard operational interpretation of the constant modality [5, 13].

$$\boxed{v : \tau \mathrel{@} n}$$

$$\textsc{VScal}\ \frac{s \in S_\nu}{s : \nu \mathrel{@} n+1} \qquad \textsc{VCons}\ \frac{v_1 : \tau \mathrel{@} n+1 \qquad v_2 : \mathbf{Stream}\ \tau \mathrel{@} n}{v_1 :: v_2 : \mathbf{Stream}\ \tau \mathrel{@} n+1}$$

$$\textsc{VClosure}\ \frac{\Gamma, x : \tau_1 \vdash e : \tau_2 \qquad \gamma : \Gamma \mathrel{@} n}{(x.e)\{\gamma\} : \tau_1 \to \tau_2 \mathrel{@} n} \qquad \textsc{VPair}\ \frac{v_1 : \tau_1 \mathrel{@} n \qquad v_2 : \tau_2 \mathrel{@} n}{(v_1, v_2) : \tau_1 \times \tau_2 \mathrel{@} n}$$

$$\textsc{VInj}_{i \in \{1,2\}}\ \frac{v : \tau_i \mathrel{@} n}{\mathbf{inj}_i v : \tau_1 + \tau_2 \mathrel{@} n} \qquad \textsc{VStop}\ \frac{}{\mathbf{stop} : \tau \mathrel{@} 0} \qquad \textsc{VThunk}\ \frac{\Gamma \vdash e : \tau \qquad \gamma : \Gamma \mathrel{@} \omega}{\mathbf{box}(e)\{\gamma\} : \tau \mathrel{@} \omega}$$

$$\textsc{VWarp}\ \frac{v : \tau \mathrel{@} p(n)}{\mathbf{w}(p, v) : *_p\, \tau \mathrel{@} n} \qquad \boxed{\gamma : \Gamma \mathrel{@} n} \qquad \frac{\begin{array}{c}\mathrm{dom}(\gamma) = \mathrm{dom}(\Gamma) \\ \forall x \in \mathrm{dom}(\gamma), \gamma(x) : \Gamma(x) \mathrel{@} n\end{array}}{\gamma : \Gamma \mathrel{@} n}$$

Figure 3. Typing Judgment for Values and Environments

$$\boxed{\lfloor v \rfloor_n \Downarrow v'} \qquad \frac{}{\lfloor s \rfloor_{n+1} \Downarrow s} \qquad \frac{\lfloor v_1 \rfloor_{n+1} \Downarrow v_1' \qquad \lfloor v_2 \rfloor_n \Downarrow v_2'}{\lfloor v_1 :: v_2 \rfloor_{n+1} \Downarrow v_1' :: v_2'}$$

$$\frac{\lfloor \gamma \rfloor_{n+1} \Downarrow \gamma'}{\lfloor (x.e)\{\gamma\} \rfloor_{n+1} \Downarrow (x.e)\{\gamma'\}} \qquad \frac{\lfloor v_1 \rfloor_{n+1} \Downarrow v_1' \qquad \lfloor v_2 \rfloor_{n+1} \Downarrow v_2'}{\lfloor (v_1, v_2) \rfloor_{n+1} \Downarrow (v_1', v_2')}$$

$$\frac{\lfloor v \rfloor_{n+1} \Downarrow v'}{\lfloor \mathbf{inj}_i v \rfloor_{n+1} \Downarrow \mathbf{inj}_i v'} \qquad \frac{\lfloor \gamma \rfloor_{n+1} \Downarrow \gamma' \qquad e; \gamma' \Downarrow_{n+1} v}{\lfloor \mathbf{box}(e)\{\gamma\} \rfloor_{n+1} \Downarrow v} \qquad \frac{\lfloor v \rfloor_{p(n+1)} \Downarrow v'}{\lfloor \mathbf{w}(p, v) \rfloor_{n+1} \Downarrow \mathbf{w}(p, v')}$$

$$\frac{}{\lfloor v \rfloor_0 \Downarrow \mathbf{stop}} \qquad \frac{}{\lfloor v \rfloor_\omega \Downarrow v}$$

Figure 4. Truncation of Values

3.1 Values and Environments

The judgment $v : \tau \mathrel{@} n$ expresses that a value v is a prefix of some infinite object of type τ at $n \in \omega + 1$. Its rules are given in Figure 3. For instance, if τ is of the form **Stream** τ', the number of elements of type τ' contained in v is exactly n.

Closures, pairs, and injections are unremarkable. Stream prefixes $v_1 :: v_2$ can only be well-typed at some $n > 0$, in which case v_2 is well-typed at $n - 1$. The dummy value **stop** inhabits all types but only at 0. Thunks $\mathbf{box}(e)\{\gamma\}$ inhabit types only at ω. Finally, warped values $\mathbf{w}(p, v)$ inhabit the warping modality, marking that v has been computed at $p(n)$.

An environment γ has type Γ at n if all its constituent values have types at n matching those prescribed by Γ.

3.2 Evaluation Judgment

The evaluation relation depends on several auxiliary judgments, which depend on evaluation themselves. They are all parameterized by a step $n \in \omega + 1$. Several of these judgments have to be extended from values to environments pointwise; since this extension is always completely unremarkable, we omit the corresponding rules.

Truncation The value typing judgment is not monotonic, in the sense that $v : \tau \mathrel{@} n + 1$ does not entail $v : \tau \mathrel{@} n$ in general. This choice makes value typing more precise, making sure that the result of a program of type **Stream Int** at n is exactly a list containing n elements. However, evaluation sometimes needs to turn a value at m into a value at $n < m$ in order to mediate between different steps. Thus, we introduce a *truncation* judgment $\lfloor v \rfloor_n \Downarrow v'$ expressing that removing all information pertaining to steps greater than n from the value v gives a value v'. Its rules are given in Figure 4.

Most rules apply when v is to be truncated to a step of the form $n + 1$. Scalars contain the same amount of information at all finite steps, and thus remain themselves. The tail v_2 of a stream constructor $v_1 :: v_2$ is truncated to n, ensuring that the final stream contains $n+1$ elements. Closures, pairs, and injections are truncated structurally; for closures, we truncate the environment. To truncate a thunk to a positive finite step is to evaluate it, obtaining a finite result; this is why truncation depends on evaluation, defined below. To truncate a value warped by p at n, truncate it at $p(n)$.

Finally, truncation to 0 and truncation to ω are symmetric. Truncation to 0 erases the value completely, leaving only **stop**. Truncation to ω keeps the value completely intact.

Coercion Application The judgment $\alpha[v] \Downarrow_n v'$ expresses that v' is the result of coercing v by α. Its rules are given in Figure 5.

As for truncation, most rules here deal with finite positive n. The identity coercion does nothing, $\alpha_1; \alpha_2$ first applies α_1 then α_2. The remaining composite coercions apply coercions in depth, as expected; note that $*_p\, \alpha$ applies α at $p(n+1)$. The wrapping (resp. unwrapping) coercion adds (resp. removes) a constructor $\mathbf{w}(\underline{id}, -)$. The coercions $\mathbf{concat}^{p,q}$ and $\mathbf{dist}_\times$ implement the transformations and commutations corresponding to their types, but have to deal with the cases where $p(n + 1) = 0$ or $p(n + 1) = \omega$ explicitly. The coercions $\mathbf{decat}^{p,q}$ and $\mathbf{fact}_\times$ are similar but simpler. Inflation creates a dummy thunk around a scalar; this is type safe since scalars are well-typed at any finite n. A delay coercion $\mathbf{delay}^{p,q}$ receives an input at $p(n + 1)$ and truncates it to $q(n + 1)$, which is smaller or equal to $p(n + 1)$ if $\mathbf{delay}^{p,q}$ is well-typed.

Evaluating a coercion at 0 immediately returns **stop**, as for truncation. On the other hand, a coercion applied at ω is necessarily applied to a thunk, and must be delayed itself. We accomplish this by pushing the coercion inside the thunk.

We have elided the context-coercion application judgment, which simply lifts coercion application componentwise to environments.

Evaluation The evaluation judgment is given in Figure 6.

Again, most of the work is done at $0 < n < \omega$, so we begin by decribing the corresponding rules. The rules for variables, functions, application, pairs, projections, injections, pattern-matching, and scalars are the standard ones of call-by-value λ-calculus. We will explain recursion shortly. To evaluate e **by** p at $n + 1$, evaluate e at $p(n + 1)$. Its result should be wrapped in $\mathbf{w}(p, -)$ to mark its provenance, and symmetrically the environment γ should be purged of a layer of $\mathbf{w}(p, -)$ value formers. The latter operation is denoted by purge(γ); it is undefined if γ contains values that are not of the form $\mathbf{w}(p, v)$. Coercions rely on the coercion application judgment and its lifting to context coercions.

All terms evaluate to **stop** at 0. The evaluation of a term e at ω suspends its execution, building a thunk $\mathbf{box}(e)\{\gamma\}$ pairing it with the current environment γ.

Recursion and Iteration Rule ERec depends on the *iteration* judgment $x; e; \gamma; v \Uparrow^n_m v'$. To explain this judgment informally, let us write f for $\mathbf{fun}\,(x : \_).e$ and assume that $m \leq n$. Then, this judgment computes $v' = f^{n-m}(v)$. Its use in rule ERec with $m = 0$ and $v = \mathbf{stop}$ ensures that $v = f^n(\mathbf{stop})$. Thus iteration can be viewed as an operational approximation of Kleene's fixpoint theorem if one identifies **stop** with $\bot$ from domain theory.

$$\boxed{\alpha[v] \Downarrow_n v'}$$

$$\textsc{ECId}\ \frac{}{\mathbf{id}[v] \Downarrow_{n+1} v} \qquad \textsc{ECSeq}\ \frac{\alpha_1[v_1] \Downarrow_{n+1} v_2 \qquad \alpha_2[v_2] \Downarrow_{n+1} v_3}{(\alpha_1;\alpha_2)[v_1] \Downarrow_{n+1} v_3}$$

$$\textsc{ECStream}\ \frac{\alpha[v_1] \Downarrow_{n+1} v_1' \qquad \mathbf{Stream}\ \alpha[v_2] \Downarrow_n v_2'}{(\mathbf{Stream}\ \alpha)[(v_1 :: v_2)] \Downarrow_{n+1} (v_1' :: v_2')} \qquad \textsc{ECFun}\ \frac{}{(\alpha_1 \to \alpha_2)[(x.e)\{\gamma\}] \Downarrow_{n+1} (x.(e\ (x;\alpha_1));\alpha_2)\{\gamma\}}$$

$$\textsc{ECProd}\ \frac{\alpha_1[v_1] \Downarrow_{n+1} v_1' \qquad \alpha_2[v_2] \Downarrow_{n+1} v_2'}{(\alpha_1 \times \alpha_2)[(v_1, v_2)] \Downarrow_{n+1} (v_1', v_2')} \qquad \textsc{ECSum}\ \frac{\alpha_i[v] \Downarrow_{n+1} v'}{(\alpha_1 + \alpha_2)[\mathbf{inj}_i v] \Downarrow_{n+1} \mathbf{inj}_i v'}$$

$$\textsc{ECWarp}\ \frac{\alpha[v] \Downarrow_{p(n+1)} v'}{(*_p\, \alpha)[\mathbf{w}(p, v)] \Downarrow_{n+1} \mathbf{w}(p, v')} \qquad \textsc{ECWrap}\ \frac{}{\mathbf{wrap}[v] \Downarrow_{n+1} \mathbf{w}(\underline{id}, v)} \qquad \textsc{ECUnwrap}\ \frac{}{\mathbf{unwrap}[\mathbf{w}(\underline{id}, v)] \Downarrow_{n+1} v}$$

$$\textsc{ECConcat}\ \frac{}{\mathbf{concat}^{p,q}[\mathbf{w}(p, \mathbf{w}(q, v))] \Downarrow_{n+1} \mathbf{w}(p * q, v)} \qquad \textsc{ECConcatStop}\ \frac{}{\mathbf{concat}^{p,q}[\mathbf{w}(p, \mathbf{stop})] \Downarrow_{n+1} \mathbf{w}(p * q, \mathbf{stop})}$$

$$\textsc{ECConcatBox}\ \frac{}{\mathbf{concat}^{p,q}[\mathbf{w}(p, \mathbf{box}(e)\{\gamma\})] \Downarrow_{n+1} \mathbf{w}(p * q, \mathbf{box}(e)\{\gamma\})}$$

$$\textsc{ECDecat}\ \frac{}{\mathbf{decat}^{p,q}[\mathbf{w}(p * q, v)] \Downarrow_{n+1} \mathbf{w}(p, \mathbf{w}(q, v))} \qquad \textsc{ECDist}\ \frac{}{\mathbf{dist}_\times[\mathbf{w}(p, (v_1, v_2))] \Downarrow_{n+1} (\mathbf{w}(p, v_1), \mathbf{w}(p, v_2))}$$

$$\textsc{ECDistStop}\ \frac{}{\mathbf{dist}_\times[\mathbf{w}(p, \mathbf{stop})] \Downarrow_{n+1} (\mathbf{w}(p, \mathbf{stop}), \mathbf{w}(p, \mathbf{stop}))}$$

$$\textsc{ECDistBox}\ \frac{}{\mathbf{dist}_\times[\mathbf{w}(p, \mathbf{box}(e)\{\gamma\})] \Downarrow_{n+1} (\mathbf{w}(p, \mathbf{box}(e)\{\gamma\}), \mathbf{w}(p, \mathbf{box}(e)\{\gamma\}))} \qquad \textsc{ECFact}\ \frac{}{\mathbf{fact}_\times[(\mathbf{w}(p, v_1), \mathbf{w}(p, v_2))] \Downarrow_{n+1} \mathbf{w}(p, (v_1, v_2))}$$

$$\textsc{ECInfl}\ \frac{}{\mathbf{inflate}[c] \Downarrow_{n+1} \mathbf{w}(\underline{\omega}, \mathbf{box}(c)\{\emptyset\})} \qquad \textsc{ECDelay}\ \frac{\lfloor v \rfloor_{q(n+1)} \Downarrow v'}{\mathbf{delay}^{p,q}[v] \Downarrow_{n+1} v'}$$

$$\textsc{ECZero}\ \frac{}{\alpha[v] \Downarrow_0 \mathbf{stop}} \qquad \textsc{ECOmega}\ \frac{}{\alpha[\mathbf{box}(e)\{\gamma\}] \Downarrow_\omega \mathbf{box}(e;\alpha)\{\gamma\}}$$

Figure 5. Coercion Application Judgment

$$\boxed{e;\gamma \Downarrow_n v}$$

$$\textsc{EVar}\ \frac{}{x;\gamma \Downarrow_{n+1} \gamma(x)} \qquad \textsc{EFun}\ \frac{}{\mathbf{fun}\,(x : \tau).e;\gamma \Downarrow_{n+1} (x.e)\{\gamma\}}$$

$$\textsc{EApp}\ \frac{e_1;\gamma \Downarrow_{n+1} (x.e)\{\gamma'\} \qquad e_2;\gamma \Downarrow_{n+1} v \qquad e;\gamma'[x \mapsto v] \Downarrow_{n+1} v'}{e_1\ e_2;\gamma \Downarrow_{n+1} v'}$$

$$\textsc{EPair}\ \frac{e_1;\gamma \Downarrow_{n+1} v_1 \qquad e_2;\gamma \Downarrow_{n+1} v_2}{(e_1, e_2);\gamma \Downarrow_{n+1} (v_1, v_2)} \qquad \textsc{EProj}_{i\in\{1,2\}}\ \frac{e;\gamma \Downarrow_{n+1} (v_1, v_2)}{\mathbf{proj}_i\ e;\gamma \Downarrow_{n+1} v_i} \qquad \textsc{EInj}_{i\in\{1,2\}}\ \frac{e;\gamma \Downarrow_{n+1} v}{\mathbf{inj}_i^\tau\ e;\gamma \Downarrow_{n+1} \mathbf{inj}_i v}$$

$$\textsc{ECase}_{i\in\{1,2\}}\ \frac{e;\gamma \Downarrow_{n+1} \mathbf{inj}_i v \qquad e_i;\gamma[x_i \mapsto v] \Downarrow_{n+1} v'}{\mathbf{case}\ e\ \mathbf{of}\{\mathbf{inj}_1\ x_1.e_1 \mid \mathbf{inj}_2\ x_2.e_2\};\gamma \Downarrow_{n+1} v'} \qquad \textsc{EConst}\ \frac{}{c;\gamma \Downarrow_{n+1} c} \qquad \textsc{ERec}\ \frac{x;e;\gamma;\mathbf{stop} \Uparrow_0^{n+1} v}{\mathbf{rec}\ (x : \tau).e;\gamma \Downarrow_{n+1} v}$$

$$\textsc{EBy}\ \frac{e;\mathrm{purge}(\gamma) \Downarrow_{p(n+1)} v}{e\ \mathbf{by}\ p;\gamma \Downarrow_{n+1} \mathbf{w}(p, v)} \qquad \textsc{EHead}\ \frac{e;\gamma \Downarrow_{n+1} v_1 :: v_2}{\mathbf{head}\ e;\gamma \Downarrow_{n+1} v_1} \qquad \textsc{ETail}\ \frac{e;\gamma \Downarrow_{n+1} v_1 :: v_2}{\mathbf{tail}\ e;\gamma \Downarrow_{n+1} \mathbf{w}(\underline{-1}, v_2)}$$

$$\textsc{ECons}\ \frac{e_1;\gamma \Downarrow_{n+1} v_1 \qquad e_2;\gamma \Downarrow_{n+1} \mathbf{w}(\underline{-1}, v_2)}{e_1 :: e_2;\gamma \Downarrow_{n+1} v_1 :: v_2} \qquad \textsc{ECoeR}\ \frac{e;\gamma \Downarrow_{n+1} v \qquad \alpha[v] \Downarrow_{n+1} v'}{e;\alpha;\gamma \Downarrow_{n+1} v'}$$

$$\textsc{ECoeL}\ \frac{\beta[\gamma] \Downarrow_{n+1} \gamma' \qquad e;\gamma' \Downarrow_{n+1} v}{(\beta;e);\gamma \Downarrow_{n+1} v} \qquad \textsc{EZero}\ \frac{}{e;\gamma \Downarrow_0 \mathbf{stop}} \qquad \textsc{EOmega}\ \frac{}{e;\gamma \Downarrow_\omega \mathbf{box}(e)\{\gamma\}}$$

Figure 6. Evaluation Judgment

$$\boxed{x;e;\gamma;v \Uparrow_m^n v'} \qquad \textsc{IFinish}\ \frac{}{x;e;\gamma;v \Uparrow_n^n v} \qquad \textsc{IStep}\ \frac{\begin{array}{c} m < n \qquad \lfloor \gamma \rfloor_{m+1} \Downarrow \gamma' \\ e;\gamma'[x \mapsto v] \Downarrow_m v' \\ x;e;\gamma;v' \Uparrow_{m+1}^n v'' \end{array}}{x;e;\gamma;v \Uparrow_m^n v''}$$

Figure 7. Iteration Judgment

Rule IFinish terminates the iteration sequence when $m = n$. Rule IStep computes $f(f^m(\mathbf{stop})) = f^{m+1}(\mathbf{stop})$ if $m < n$. The environment γ is well-typed at n and so must be truncated to $m + 1$.

3.3 Metatheoretical Results

Our evaluation judgments represent runtime errors by the absence of a result, as is common in big-step semantics. Thus, our judgments define partial functions: there is at most one value v such that $e;\gamma \Downarrow_n v$, and similarly for all the other judgments.

Type Safety The following basic type safety result ensures that if a closed program of type τ evaluates to a value v at n, then v is of type τ at n. Given the typing rules for values, this ensure in particular that streams have the length described by their types.

Theorem 3.1 (Type Safety). *If $\Gamma \vdash e : \tau$, $\gamma : \Gamma$ @ n, and $e;\gamma \Downarrow_n v$, then $v : \tau$ @ n.*

Since the evaluation judgment depends on the truncation, coercion application, and iteration judgments and vice-versa, the proof must proceed by mutual induction, using the relevant type safety lemmas for auxiliary judgments.

Lemma 3.2 (Type Safety, Truncation). *If $v : \tau$ @ m and $\lfloor v \rfloor_n \Downarrow v'$ with $n \leq m$, then $v' : \tau$ @ n.*

Lemma 3.3 (Type Safety, Coercion Application). *If $\alpha : \tau <: \tau'$, $v : \tau$ @ n, and $\alpha[v] \Downarrow_n v'$, then $v' : \tau'$ @ n.*

Lemma 3.4 (Type Safety, Iteration). *If $\Gamma, x : *_{\underline{-1}}\, \tau \vdash e : \tau$, $\gamma : \Gamma$ @ n, $v : \tau$ @ m and $x;e;\gamma;v \Uparrow_m^n v'$, then $v : \tau$ @ n.*

Totality In addition to the usual type errors, in our setting partiality may also arise from time-related operations. For instance, a term might try to truncate a value at n to $m > n$, or to evaluate e **by** p in an environment which contains values that are not of the form $\mathbf{w}(p, -)$. The following theorem asserts that this cannot occur with well-typed terms.

Theorem 3.5 (Totality). *If $\Gamma \vdash e : \tau$ and $\gamma : \Gamma$ @ n, then there exists v such that $e;\gamma \Downarrow_n v$.*

The proof uses a realizability predicate, as explained in the appendix. It also requires the following result, also used in Section 4.

Property 5 (Functoriality of Truncation). *If $\lfloor v \rfloor_n \Downarrow v'$ and $\lfloor v \rfloor_m \Downarrow v''$ with $m \leq n$, then $\lfloor v' \rfloor_m \Downarrow v''$.*

Monotonicity Finally, we prove that evaluation indeed computes longer and longer prefixes of the same object.

Property 6 (Monotonicity). *If $e; \gamma \Downarrow_n v$ and $e; \gamma' \Downarrow_m v'$ with $m \leq n$ and $\lfloor \gamma \rfloor_m \Downarrow \gamma'$, then $\lfloor v \rfloor_m \Downarrow v'$.*

Coherence We have defined evaluation only on explicit terms, and indeed coercions play a crucial role in determining the result of a computation. Thus the question of *coherence* arises: do all refiners of the same implicit term having the same type compute the same results? We give a positive answer to this question in Section 5 using the denotational semantics developed in the next section.

4 Denotational Semantics

4.1 Preliminaries

Let **Bool** denote the category with two objects $\bot$, $\top$ and a single non-identity morphism $t : \bot \to \top$, equipped with the strict monoidal structure given by conjunction. Then, preorders are **Bool**-enriched categories, and $\widehat{P}$ is isomorphic to $P^{op} \to \mathbf{Bool}$. Let **P** denote the strict monoidal functor $\mathbf{P} : \mathbf{Bool} \to \mathbf{Set}$ sending $\bot$ to $\emptyset$ and $\top$ to $\{*\}$. We write $\mathbf{P}[P]$ for the degenerate category associated with a preorder P; its hom-sets contain at most one morphism.

Given a category $\mathscr{C}$, we denote by $\widehat{\mathscr{C}} = \mathscr{C}^{op} \to \mathbf{Set}$ the category of contravariant presheaves over $\mathscr{C}$. Note that $\widehat{P}$ differs from $\widehat{\mathbf{P}[P]}$, hence our unsual choice of to formally distinguish preorders from ordinary categories.

4.2 The Topos of Trees

In this section we sketch a model of Core λ^* in the topos of trees. Birkedal et al. [7] show that this category is a convenient setting for modeling guarded recursion and synthetic step-indexing. We follow their terminology and notations.

Definition 2. *The* topos of trees*, denoted $\mathcal{S}$, is $\widehat{\mathbf{P}[\omega]}$.*

Briefly, an object X in the topos of trees can be described as a family of sets $(X(n))_{n \in \omega}$, together with a family of *restriction* functions $(r^X_n : X(n+1) \to X(n))_{n \in \omega}$. The set $X(n)$ describes what can be observed of X at step n, and the restriction functions define how future observations extend current ones. Morphisms $f : X \to Y$ are collections of functions $(f_n : X(n) \to Y(n))_{n \in \omega}$ commuting with restriction functions.

As a topos, this category naturally has all the structure required for interpreting simply-typed λ-calculus with products and sums.

$$\_\times\_ : \mathcal{S} \times \mathcal{S} \to \mathcal{S} \qquad \_+\_ : \mathcal{S} \times \mathcal{S} \to \mathcal{S} \qquad (\_)^{(\_)} : \mathcal{S}^{op} \times \mathcal{S} \to \mathcal{S}$$

This structure follows from general constructions in presheaf categories. Products and sums, as limits and colimits, are given pointwise. Exponentiation can be deduced from the Yoneda lemma.

The later modality is interpreted in $\mathcal{S}$ by the functor $\blacktriangleright$ such that $(\blacktriangleright X)(0) = \{*\}$ and $(\blacktriangleright X)(n+1) = X(n)$. A certain family of morphisms $\mathit{fix}_X : X^{\blacktriangleright X} \to X$ of $\mathcal{S}$ provide fixpoint combinators, and are used to interpret guarded recursion. We refer to Birkedal et al. [7] for additional information.

4.3 Interpreting the Warping Modality

In order to interpret the warping modality, we need to equip the topos of trees with a functor $*_p : \mathcal{S} \to \mathcal{S}$ for every time warp p. Intuitively, $(*_p X)(n)$ should contain "p-times" more information than $X(n)$. Moreover, the family of functors $*_{(-)}$ should come equipped with enough structure to interpret atomic coercions.

Pulling Presheaves along Functions To understand what this operation should look like, let us first consider a restricted class of time warps. By definition, time warps p such that $0 < p(n) < \omega$ for all $0 < n < \omega$ are in a one-to-one correspondence with monotonic functions $f : \omega \to \omega$. In this case, one can simply define $(*_p X)(n) = X(f(n))$. Thus, if p happens to be equivalent to a function $\omega \to \omega$, the functor $*_p : \mathcal{S} \to \mathcal{S}$ is simply given by precomposition with p. From a categorical logic perspective, computing $*_p X$ corresponds to *pulling X along p.*

This special case already captures some examples from the literature. For instance, Birkedal et al. [7] study the left adjoint $\blacktriangleleft$ of $\blacktriangleright$ given by $(\blacktriangleleft X)(n) \triangleq X(n+1)$, which would thus correspond to $*_{n \mapsto n+1}$. However, most interesting time warps are not ω-valued, including those corresponding to the later and constant modalities, and thus cannot be naively precomposed with presheaves from $\mathcal{S}$.

Pulling Presheaves along Distributors A solution to the above problem is provided by the theory of *distributors*, which are to functors what relations are to functions. A distributor $P : \mathscr{C} \nrightarrow \mathscr{D}$ from a category $\mathscr{C}$ to a category $\mathscr{D}$ is a functor $P : \mathscr{D}^{op} \times \mathscr{C} \to \mathbf{Set}$. Distributors form a (bi)category, and enjoy properties that plain functors lack. We refer to Bénabou [10] for an introduction.

Any presheaf $X : \mathscr{C}^{op} \to \mathbf{Set}$ is by definition equivalent to a distributor $\mathbf{1} \nrightarrow \mathscr{C}$, with $\mathbf{1}$ the category with a single object and morphism. Postcomposing a distributor $M : \mathscr{C} \nrightarrow \mathscr{D}$ with X gives a presheaf $MX : \mathbf{1} \nrightarrow \mathscr{D}$ which, intuitively, is X *pushed along* M. It is a crucial characteristic of distributors that post-composition with M always has a right adjoint, which we will write $(-)/M$. This right adjoint can be described by the end formula

$$Y/M \triangleq \int_{d \in \mathscr{D}} Y(d)^{M(d,-)}. \tag{2}$$

The presheaf Y/M is the result of *pulling Y along M*, as recently expounded by Melliès and Zeilberger [25].

Pulling Presheaves along Time Warps We can extend the construction given above to time warps by realizing that the latter are miniature distributors.

It is a consequence of the Yoneda lemma that every distributor $\mathscr{C} \nrightarrow \mathscr{D}$ can be seen as a cocontinuous functor $\widehat{\mathscr{C}} \to \widehat{\mathscr{D}}$ and vice-versa. A similar result holds for preorders: every cocontinuous function $\widehat{P} \to \widehat{Q}$ corresponds to a monotonic function $Q^{op} \times P \to \mathbf{Bool}$, and vice-versa. We call such functions *linear systems*, adopting the terminology attributed to Winskel by Hyland [19, §4.1]. Given a time warp $p : \widehat{\omega} \to \widehat{\omega}$, we refer to the corresponding linear system as $\overline{p} : \omega \nrightarrow \omega$. We have $(m, n) \in \overline{p}$ if and only if $\mathbf{y}(m) \leq p(\mathbf{y}(n))$.

Pulling a presheaf along a time warp is now possible since linear systems are nothing but **Bool**-enriched distributors. Precomposing the linear system $\overline{p}$ with $\mathbf{P} : \mathbf{Bool} \to \mathbf{Set}$, we obtain a standard distributor $\mathbf{P}\overline{p} : \omega \nrightarrow \omega$, which we then combine with Equation (2).

Definition 3 (Warping Functor). *Given a time warp p, we define the warping functor $*_p : \mathcal{S} \to \mathcal{S}$ as*

$$*_p \triangleq (-)/(\mathbf{P}\overline{p}). \tag{3}$$

Unfolding and simplifying the above definition, we obtain an explicit formula for the observations of $*_p X$ at n.

$$(*_p X)(n) = \left\{ (x_m) \in \prod_{m=1}^{p(n)} X(m) \;\middle|\; x_m = r^X_m(x_{m+1}) \right\} \tag{4}$$

The reader may check using the above formula that $(*_{\underline{-1}} X)(n)$ coincides with $\blacktriangleright X(n)$. The same is true for $*_{\underline{\omega}}$ and $\blacksquare$.

Once the relatively intuitive Equation (4) has been found, it might seem that the abstract Definition 3 becomes unnecessary. However, the abstract approach gives insight into the structure of $*_{(-)}$. In particular, routine categorical considerations imply the following.

Property 7. *Warping defines a strong monoidal functor*

$$*_{(-)} : \mathcal{W}^{op(0,1)} \to End(\mathcal{S}).$$

Here $\mathcal{W}$ and $End(\mathcal{S})$ are considered as monoidal categories whose objects are respectively time warps and endofunctors of $\mathcal{S}$, and where the monoidal structure is given by composition in both cases. The category $\mathcal{W}$ is preordered. Property 7 entails the existence of the following structure.

$$*_{p \geq q} : *_p \to *_q \qquad \epsilon : *_{\underline{1}} \cong Id \qquad \mu^{p,q} : *_p *_q \cong *_{p*q}$$

Moreover, every functor $*_p$ is a right adjoint, hence limit-preserving.

4.4 The Interpretation

Ground types are interpreted using the functor $\Delta : \mathbf{Set} \to \mathcal{S}$ mapping every set to a constant presheaf. The interpretation of **Stream** τ is characterized by $[\![\mathbf{Stream}\ \tau]\!](n) = \prod_{m=1}^{n} [\![\tau]\!](m)$. We have already given the interpretation of all other types. Typing contexts are interpreted as cartesian products.

Coercions $\alpha : \tau <: \tau'$ give rise to morphisms $[\![\alpha]\!] : [\![\tau]\!] \to [\![\tau']\!]$. Composite coercions are interpreted by the functorial actions of type constructors, plus plain composition. Atomic coercions take advantage of the structure arising from Property 7. For example, $[\![\mathbf{concat}^{p,q} : *_p *_q \tau <: *_{p*q} \tau]\!] = \mu^{p,q}_{[\![\tau]\!]}$ and $[\![\mathbf{delay}^{p,q} : *_p \tau <: *_q \tau]\!] = (*_{p \geq q})_{[\![\tau]\!]}$. The **inflate** coercion is interpreted by the general isomorphism between $\Delta(S)$ and $*_{\underline{\omega}} \Delta(S)$. The $\mathbf{dist}_\times$ and $\mathbf{fact}_\times$ coercions are interpreted by the natural isomorphisms arising from the limit-preservation property of $*_p$.

Since the type system of Figure 1 is not exactly syntax-directed, we will interpret typing derivations rather than terms. Guarded recursion is interpreted using the $\mathit{fix}_{[\![\tau]\!]}$ morphisms. We interpret structure maps $\sigma \in \Sigma(\Gamma; \Gamma')$ as morphisms $[\![\sigma]\!] : [\![\Gamma']\!] \to [\![\Gamma]\!]$ and rule Struct by precomposition. Other cases are standard [22].

Property 8 (Coherence for Explicit Terms)**.** *Any two derivations of* $\Gamma \vdash e : \tau$ *are interpreted by the same morphism in* $\mathcal{S}$.

The proof shows that the interpretation of any derivation of e is equal to the interpretation of the canonical derivation for e built in Property 3. Since this canonical derivation is unique, this entails the coherence of the interpretation for explicit terms.

4.5 Adequacy

The interpretation reflects operational equivalence, which in Core λ^* consists in observing scalars at the first step.

Theorem 4.1. *If* $[\![\Gamma \vdash e : \tau]\!] = [\![\Gamma \vdash e' : \tau]\!]$ *then* $\Gamma \vdash e \cong_{ctx} e' : \tau$.

To prove the result, we remark that the values described in Section 3 can be organized as an object of $\mathcal{S}$, using results such as Property 6. The details can be found in the appendix.

5 Algorithmic Type Checking

The abstract type-checking algorithm we present in this section builds an explicit term from an implicit one in a canonical way. This involves two main challenges: deciding the subtyping judgment, and dealing with the context restriction arising in rule Warp.

5.1 Deciding Subtyping

To decide subtyping, we start with the observation that most atomic coercions $\alpha : \tau <: \tau'$ from Figure 2 come in pairs, in the sense that there exists α^{-1} such that $\alpha^{-1} : \tau' <: \tau$. This is even true for **inflate**, since we can take $\mathbf{inflate}^{-1}$ to be $\mathbf{delay}^{\underline{\omega},\underline{id}}$; **unwrap**. The only atomic coercion for which this is not the case is $\mathbf{delay}^{p,q}$ when $q < p$. This suggests dealing with delays separately.

Normalizing Types To deal with invertible coercions, we define a function τ mapping each type to an equivalent but simpler form. Such *normal* types τ^n obey the following grammar.

$$\tau^n ::= *_p \tau^r \mid \tau^n \times \tau^n \qquad \tau^r ::= \nu \mid \mathbf{Stream}\ \tau^n \mid \tau^n \to \tau^n \mid \tau^n + \tau^n$$

In other words, normal types feature exactly one warping modality immediately above every non-product type former.

The total function $\wr \tau \int$ returns the normalized form of τ. It is defined by recursion on τ in Figure 8-A. For every τ, there are coercions $(\wr \tau \int_{\text{in}}, \wr \tau \int_{\text{out}}) : \tau \equiv \wr \tau \int$, defined in the appendix.

Deciding Precedence We now decide subtyping in the special case where the only atomic coercions allowed are delays, a case we call *precedence*. The corresponding partial computable function $\mathit{Prec}(-;-)$, when defined, builds a coercion $\mathit{Prec}(\tau;\tau') : \tau <: \tau'$. It is given in Figure 8-B. In the absence of $\mathbf{concat}^{p,q}$ and $\mathbf{decat}^{p,q}$ coercions, it is enough to traverse τ and τ' in lockstep, checking whether $p \geq q$ holds when comparing $*_p \tau$ and $*_q \tau'$.

Putting it all together We decide subtyping in the general case by combining precedence with normalization:

$$\mathit{Coe}(\tau;\tau') \triangleq \wr \tau \int_{\text{in}}; \mathit{Prec}(\wr \tau \int; \wr \tau' \int); \wr \tau' \int_{\text{out}}.$$

We write $\mathit{Coe}(\Gamma;\Gamma')$ for the pointwise extension to contexts.

5.2 Adjoint Typing Contexts

Consider the type-checking problem for t **by** p in a given context Γ. If every type $\tau = \Gamma(x)$, with x a free variable of t, is of the form $*_p \tau'$, we may apply rule Warp. Otherwise, we have to find a type τ' such that $\tau <: *_p \tau'$. There are several choices for τ', and they are far from equivalent. For instance, taking $\tau' \triangleq *_{\underline{0}} \tau$ would work since $\tau <: *_{\underline{0}} \tau <: *_p *_{\underline{0}} \tau$ always holds, but will in general impose artificial constraints on the type of t. For τ' to be a canonical choice,

$$\tau <: *_p \tau'' \Leftrightarrow \tau' <: \tau'' \tag{5}$$

needs to hold for any type τ''. Now, assume that τ and τ'' are normalized types which are not products, and thus necessarily start with a warping modality. Equivalence (5) becomes

$$*_q \tau <: *_p *_r \tau'' \Leftrightarrow \tau' <: *_r \tau''. \tag{6}$$

Then, a solution satisfying (5) is given by $\tau' = *_{q \setminus p} \tau$, with $q \setminus p$ a hypothetical time warp such that

$$r \circ p \leq q \Leftrightarrow r \leq q \setminus p. \tag{7}$$

We are thus looking for an operation $(-) \setminus p$ right adjoint to precomposition $(-) \circ p$. Right adjoints to precomposition (and post-composition, cf. Section 4) always exist for distributors [10, §4], and thus linear systems. The general formula, specialized to linear systems and time warps, gives

$$(q \setminus p)(n) = p(\min\{m \in \omega + 1 \mid n \leq q(m)\}). \tag{8}$$

$$\begin{aligned} \lbag \nu \rbag &= \ast_{\underline{\omega}}\, \nu \\ \lbag \mathbf{Stream}\ \tau \rbag &= \ast_{\underline{id}}\, \mathbf{Stream}\ \lbag \tau \rbag \\ \lbag \tau_1 \to \tau_2 \rbag &= \ast_{\underline{id}}\, (\lbag \tau_1 \rbag \to \lbag \tau_2 \rbag) \\ \lbag \tau_1 \times \tau_2 \rbag &= \lbag \tau_1 \rbag \times \lbag \tau_2 \rbag \end{aligned} \qquad \begin{aligned} \lbag \tau_1 + \tau_2 \rbag &= \ast_{\underline{id}}\, (\lbag \tau_1 \rbag + \lbag \tau_2 \rbag) \\ \lbag \ast_p\, (\tau_1 \times \tau_2) \rbag &= \lbag \ast_p\, \tau_1 \rbag \times \lbag \ast_p\, \tau_2 \rbag \\ \lbag \ast_p\, \tau \rbag &= \ast_{p \ast q}\, \tau' \text{ where } \tau \neq (\_ \times \_) \\ &\qquad \text{and } \ast_q\, \tau' = \lbag \tau \rbag \end{aligned}$$

(A) Type Normalization

$$\begin{aligned} \mathit{Prec}(\nu; \nu) &= \mathbf{id} \\ \mathit{Prec}(\mathbf{Stream}\ \tau_1; \mathbf{Stream}\ \tau_2) &= \mathbf{Stream}\ \mathit{Prec}(\tau_1; \tau_2) \\ \mathit{Prec}(\tau_1' \to \tau_1''; \tau_2' \to \tau_2'') &= \mathit{Prec}(\tau_2'; \tau_1') \to \mathit{Prec}(\tau_1''; \tau_2'') \\ \mathit{Prec}(\tau_1' \times \tau_1''; \tau_2' \times \tau_2'') &= \mathit{Prec}(\tau_1'; \tau_2') \times \mathit{Prec}(\tau_1''; \tau_2'') \\ \mathit{Prec}(\tau_1' + \tau_1''; \tau_2' + \tau_2'') &= \mathit{Prec}(\tau_1'; \tau_2') + \mathit{Prec}(\tau_1''; \tau_2'') \\ \mathit{Prec}(\ast_p\, \tau_1; \ast_q\, \tau_2) &= \mathbf{delay}^{p,q}; \ast_q\, \mathit{Prec}(\tau_1; \tau_2) \text{ if } p \geq q \end{aligned}$$

(B) Type Precedence

Figure 8. Type Normalization and Precedence

Thus, we define normal-type division as

$$(\tau_1 \times \tau_2) \setminus_n p = (\tau_1 \setminus_n p) \times (\tau_2 \setminus_n p) \text{ and } (\ast_q\, \tau) \setminus_n p = \ast_{q \setminus p}\, \tau$$

and general type division as $\tau \setminus p \triangleq \lbag \tau \rbag \setminus_n p$.

5.3 The Algorithm

The partial computable function $\mathit{Elab}(\Gamma; t)$ returns a pair (τ, e) with $e \sqsubset t$ such that $\Gamma \vdash e : \tau$ holds. Its definition is given in Figure 9. It uses the algorithmic subtyping judgment when type-checking destructors, and the context division judgment when applying rule Warp. The case of pattern-matching relies on the existence of type suprema, which are easy to compute structurally for normal types; see the appendix.

5.4 Metatheoretical Results

Lemma 5.1. *If $\alpha : \tau <: \tau'$ then $\mathit{Coe}(\tau; \tau')$ is defined and*

$$[\![\alpha : \tau <: \tau']\!] = [\![\mathit{Coe}(\tau; \tau') : \tau <: \tau']\!].$$

Theorem 5.2 (Completeness of Algorithmic Typing)**.** *If $\Gamma \vdash e : \tau$, there is e^m, τ^m, α^m with $(\tau^m, e^m) = \mathit{Elab}(\Gamma; t)$, $\alpha^m : \tau^m <: \tau$, and*

$$[\![\Gamma \vdash e : \tau]\!] = [\![\Gamma \vdash e : \tau^m]\!]; [\![\alpha : \tau^m <: \tau]\!].$$

The fact that algorithmic subtyping is deterministic together with Lemma 5.1 and Theorem 5.2 immediately entails coherence.

Corollary 1 (Denotational Coherence)**.** *For any $e_1, e_2 \sqsubset t$ such that $\Gamma \vdash e_1 : \tau$ and $\Gamma \vdash e_2 : \tau$, we have $[\![\Gamma \vdash e_1 : \tau]\!] = [\![\Gamma \vdash e_2 : \tau]\!]$.*

Corollary 2 (Operational Coherence)**.** *For any $e_1, e_2 \sqsubset t$ such that $\Gamma \vdash e_1 : \tau$ and $\Gamma \vdash e_2 : \tau$, we have $\Gamma \vdash e_1 \cong_{ctx} e_2 : \tau$.*

6 Discussion and Related Work

6.1 Guarded Type Theories

Expressiveness On the one hand, Core λ^* captures finer-grained temporal information than existing guarded type theories, and also recasts their modalities in a uniform setting. We illustrate this point by comparing Core λ^* to the gλ-calculus [13], since they are relatively close. The later and constant modality correspond respectively to $\ast_{\underline{-1}}$ and $\ast_{\underline{\omega}}$. The g$\lambda$-calculus operations **next** $: \tau \longrightarrow \blacktriangleright \tau$ and **unbox** $: \blacksquare\, \tau \longrightarrow \tau$ correspond to the coercions **wrap**; $\mathbf{delay}^{\underline{id}, \underline{-1}}$ and $\mathbf{delay}^{\underline{\omega}, \underline{id}}$; **unwrap**. Erasing later modalities in the gλ-calculus happens via the term former **prev**, which restricts the context to be constant (essentially, under $\blacksquare$); in Core λ^*, this would arise from the implicit type equivalence $\ast_{\underline{\omega}} \ast_{\underline{-1}} \tau \equiv \ast_{\underline{\omega} \ast \underline{-1}}\, \tau = \ast_{\underline{\omega}}\, \tau$. Additionally, the introduction rule for $\blacksquare$ in the gλ-calculus is more restrictive than rule Warp for t **by** $\underline{\omega}$, since the latter allows the free variables of t to have types $\ast_p\, \tau$ where p is constant but not necessarily $\underline{\omega}$. The gλ-calculus makes $\blacktriangleright$ into an “applicative functor” [23], implementing only the left-to-right direction of the type isomorphism $\ast_{\underline{-1}}\, (\tau_1 \longrightarrow \tau_2) \cong \ast_{\underline{-1}}\, \tau_1 \longrightarrow \ast_{\underline{-1}}\, \tau_2$. In Core λ^*, both directions are definable, the right-to-left one as

$$\mathbf{fun}\, (\mathrm{f} : \ast_{\underline{-1}}\, \tau_1 \longrightarrow \ast_{\underline{-1}}\, \tau_2).((\mathbf{fun}\, (x : \tau_1).(\mathrm{f}\ \mathrm{x})\ \mathbf{by}\ \underline{+1})\ \mathbf{by}\ \underline{-1})$$

where $\underline{+1}$ is is the time warp which is left adjoint to $\underline{-1}$ ($\blacktriangleleft$ in [7]).

On the other hand Core λ^* lacks many features present in other guarded type theories (including the gλ-calculus). It would be useful, for instance, to replace the fixed stream type with general guarded recursive types [7, 13]; this requires designing a guardedness criterion in the presence of the warping modality. Clock variables [3] would allow types to express that unrelated program pieces may operate within disjoint time scales. Core λ^* enjoys decidable type-checking, but not type inference; in contrast, type inference for the later modality has been studied by Severi [31]. Finally, Core λ^* might be difficult to extend to dependent types, since it is inherently call-by-value, whereas several dependent type theories with later have been proposed [6, 8].

Metatheory Core λ^* also stands out among guarded type theories by the design of its metatheory. First, as mentioned above, its semantics fixes a call-by-value evaluation strategy, in contrast with actual calculi enjoying unrestricted β-reduction. We believe that this is natural since t **by** p is in essence an effectful term which modifies the current time step.

Second, the context restriction in rule Warp is perhaps controversial from a technical perspective. This kind of rule, acting on the left of the turnstile, is normally avoided in natural-deduction presentations as it is known to cause “anomalies” [28], e.g., breaking substitution lemmas. Since Core λ^* is call-by-value, we do not need subtitution to hold for arbitrary terms. We do not expect difficulties in proving a substitution lemma for values in a variant of Core λ^* where they have been made a subclass of expressions, defining $(t\ \mathbf{by}\ p)[x \backslash v]$ to be $t[x \backslash \mathrm{purge}(v)]$ **by** p, with purge(v) defined as in Section 3.

Third, Core λ^* uses subtyping, which has been eschewed by guarded type theories after Nakano’s original proposal. Yet, the context restriction of rule Warp makes subtyping extremely useful in practice. In its absence, terms would have to massage the typing context before introducing the warping modality. Guarded recursion would also be more difficult to use without the ability to reason up to time warp composition.

6.2 Synchronous Programming Languages

Core λ^* is a relative of synchronous programming languages in the vein of Lustre [11, 12, 14, 17, 18, . . .]. Such languages use “clocks” (not to be confused with clock variables) to describe stream growth;

$$
\begin{aligned}
\mathit{Elab}(\Gamma; x) &= (\Gamma(x), x) \\
\mathit{Elab}(\Gamma; \mathbf{fun}\,(x : \tau).t) &= (\tau \to \tau', \mathbf{fun}\,(x : \tau).e) \text{ where } (\tau', e) = \mathit{Elab}(\Gamma, x : \tau; t) \\
\mathit{Elab}(\Gamma; t_1\ t_2) &= (\tau_1'', (e_1; \mathit{Coe}(\tau_1; \tau_1' \to \tau_1''))\ (e_2; \mathit{Coe}(\tau_2; \tau_1'))) \\
&\quad \text{where } (\tau_i, e_i) = \mathit{Elab}(\Gamma; t_i) \text{ and } *_{-}\,(\tau_1' \to \tau_1'') = \wr \tau_1 \int \\
\mathit{Elab}(\Gamma; (t_1, t_2)) &= (\tau_1 \times \tau_2, (e_1, e_2)) \text{ where } (\tau_i, e_i) = \mathit{Elab}(\Gamma; t_i) \\
\mathit{Elab}(\Gamma; \mathbf{proj}_i\ t) &= (\tau_i, \mathbf{proj}_i\,(e; \mathit{Coe}(\tau; \tau_1 \times \tau_2))) \text{ where } (\tau, e) = \mathit{Elab}(\Gamma; t) \text{ and } \tau_1 \times \tau_2 = \wr \tau \int \\
\mathit{Elab}(\Gamma; \mathbf{inj}_{1+i}^{\tau_{2-i}}\ t) &= (\tau_1 + \tau_2, \mathbf{inj}_{1+i}^{\tau_{2-i}}\ e) \text{ where } (\tau_{1+i}, e) = \mathit{Elab}(\Gamma; t) \\
\mathit{Elab}(\Gamma; \mathbf{case}\ t\ \mathbf{of}\,\{\mathbf{inj}_1\ x_1.t_1 \mid \mathbf{inj}_2\ x_2.t_2\}) &= (\tau_1' \sqcup \tau_2', \mathbf{case}\ e; \mathit{Coe}(\tau; \tau_1 + \tau_2)\ \mathbf{of}\,\{\mathbf{inj}_1\ x_1.e_1; \mathit{Coe}(\tau_1'; \tau_1' \sqcup \tau_2') \mid \mathbf{inj}_2\ x_2.e_2; \mathit{Coe}(\tau_2'; \tau_1' \sqcup \tau_2')\}) \\
&\quad \text{where } (\tau, e) = \mathit{Elab}(\Gamma; t) \text{ and } *_{-}\,(\tau_1 + \tau_2) = \wr \tau \int \text{ and } (\tau_i', e_i) = \mathit{Elab}(\Gamma, x : \tau_i; t_i) \\
\mathit{Elab}(\Gamma; s) &= (\nu, s) \text{ where } s \in S_\nu \\
\mathit{Elab}(\Gamma; \mathbf{rec}\ (x : \tau).t) &= (\tau, \mathbf{rec}\ (x : \tau).(e; \mathit{Coe}(\tau'; \tau))) \text{ where } (\tau', e) = \mathit{Elab}(\Gamma, x : *_{\underline{-1}}\,\tau; t) \\
\mathit{Elab}(\Gamma; t\ \mathbf{by}\ p) &= (*_p\,\tau, \mathit{Coe}(\Gamma; *_p\,(\Gamma \setminus p)); e\ \mathbf{by}\ p) \text{ where } (\tau, e) = \mathit{Elab}(\Gamma \setminus p; t) \\
\mathit{Elab}(\Gamma; \mathbf{head}\ t) &= (\tau', \mathbf{head}\,(e; \mathit{Coe}(\tau; \mathbf{Stream}\ \tau'))) \text{ where } (\tau, e) = \mathit{Elab}(\Gamma; t) \text{ and } *_{-}\,\mathbf{Stream}\ \tau' = \wr \tau \int \\
\mathit{Elab}(\Gamma; \mathbf{tail}\ t) &= (*_{\underline{-1}}\,\mathbf{Stream}\ \tau', \mathbf{tail}\,(e; \mathit{Coe}(\tau; \mathbf{Stream}\ \tau'))) \text{ where } (\tau, e) = \mathit{Elab}(\Gamma; t) \text{ and } *_{-}\,\mathbf{Stream}\ \tau' = \wr \tau \int \\
\mathit{Elab}(\Gamma; t_1 :: t_2) &= (\mathbf{Stream}\ (\tau_1 \sqcup \tau_2'), (e_1; \mathit{Coe}(\tau_1; \tau_1 \sqcup \tau_2')) :: (e_2; \mathit{Coe}(\tau_2; *_{\underline{-1}}\,\mathbf{Stream}\ (\tau_1 \sqcup \tau_2')))) \\
&\quad \text{where } (\tau_i, e_i) = \mathit{Elab}(\Gamma; t_i) \text{ and } *_{-}\,\mathbf{Stream}\ \tau_2' = \wr \tau_2 \int
\end{aligned}
$$

Figure 9. Elaboration

such a clock is a time warp whose image forms a downward-closed subset of ω (except in [18]). Synchronous languages are generally first-order (with exceptions [18, 30]) and separate clock analysis from productivity checking. As a result, Core λ^* is both more flexible and simpler from a metatheoretical standpoint. However, it does not enforce bounds on space usage, in contrast with synchronous languages or the work of Krishnaswami [20, 21].

Acknowledgements This work has benefited from conversations with many researchers, including Albert Cohen, Louis Mandel, Paul-André Melliès, and Marc Pouzet. It owes an especially great deal to Paul-André Melliès, who introduced the author to the topos of trees and distributors. This work was partially supported by the German Research Council (DFG) under Grant No. ME14271/6-2.

References

[1] Nada Amin and Tiark Rompf. 2017. Type Soundness Proofs with Definitional Interpreters. *Principles of Programming Languages (POPL'17).*

[2] Robert Atkey. 2006. *Substructural Simple Type Theories for Separation and In-place Update.* Ph.D. Dissertation. University of Edinburgh.

[3] Robert Atkey and Conor McBride. 2013. Productive Coprogramming with Guarded Recursion. In *International Conference on Functional Programming (ICFP 2013).* ACM.

[4] Patrick Bahr, Hans Bugge Grathwohl, and Rasmus Ejlers Møgelberg. 2017. The Clocks Are Ticking: No More Delays! Reduction Semantics for Type Theory with Guarded Recursion. In *Logic in Computer Science (LICS'17).* Springer.

[5] Gavin Bierman and Valeria de Paiva. 2000. On an Intuitionistic Modal Logic. *Studia Logica* 65, 3 (2000), 383–416.

[6] Lars Birkedal, Ales Bizjak, Ranald Clouston, Hans Bugge Grathwohl, Bas Spitters, and Andrea Vezzosi. 2016. Guarded Cubical Type Theory: Path Equality for Guarded Recursion. In *Computer Science Logic (CSL'16).*

[7] Lars Birkedal, Rasmus Ejlers Møgelberg, Jan Schwinghammer, and Kristian Støvring. 2012. First steps in synthetic guarded domain theory: step-indexing in the topos of trees. *Logical Methods in Computer Science* 8, 4 (2012).

[8] Ale Bizjak, Hans Bugge Grathwohl, Ranald Clouston, Rasmus E. Møgelberg, and Lars Birkedal. 2016. Guarded Dependent Type Theory with Coinductive Types. In *Foundations of Software Science and Computation Structures (FoSSaCS'16).* Springer.

[9] Val Breazu-Tannen, Thierry Coquand, Carl A. Gunther, and Andre Scedrov. 1991. Inheritance as Implicit Coercion. *Information and Computation* (1991).

[10] Jean Bénabou. 2000. Distributors at Work. (2000).

[11] Paul Caspi, Daniel Pilaud, Nicolas Halbwachs, and John Plaice. 1987. LUSTRE: A declarative language for programming synchronous systems. In *Principles of Programming Languages (POPL'87).*

[12] Paul Caspi and Marc Pouzet. 1996. Synchronous Kahn Networks. In *International Conference on Functional Programming (ICFP'96).* ACM.

[13] Ranald Clouston, Ales Bizjak, Hans Bugge Grathwohl, and Lars Birkedal. 2016. The Guarded Lambda-Calculus: Programming and Reasoning with Guarded Recursion for Coinductive Types. *Logical Methods in Computer Science* (2016).

[14] Albert Cohen, Marc Duranton, Christine Eisenbeis, Claire Pagetti, Florence Plateau, and Marc Pouzet. 2006. *N*-synchronous Kahn networks: a relaxed model of synchrony for real-time systems. In *Principles of Programming Languages (POPL'06).*

[15] Pierre-Louis Curien, Marcelo Fiore, and Guillaume Munch-Maccagnoni. 2016. A Theory of Effects and Resources: Adjunction Models and Polarised Calculi. In *Principles of Programming Languages (POPL'16).* ACM.

[16] Pierre-Louis Curien and Giorgio Ghelli. 1990. Coherence of subsumption. In *Colloquium on Trees in Algebra and Programming (CAAP'90).* Springer.

[17] Julien Forget, Fréderic Boniol, Daniel Lesens, and Claire Pagetti. 2008. A Multi-Periodic Synchronous Data-Flow Language. In *High-Assurance Systems Engineering (HASE'08).* IEEE.

[18] Adrien Guatto. 2016. *A Synchronous Functional Language with Integer Clocks.* Ph.D. Dissertation. École normale supérieure.

[19] Martin Hyland. 2010. Some Reasons for Generalising Domain Theory. *Mathematical Structures in Computer Science* 20, 02 (Mar 2010), 239.

[20] Neelakantan R Krishnaswami. 2013. Higher-Order Functional Reactive Programming without Spacetime Leaks. In *International Conference on Functional Programming (ICFP'13).* ACM.

[21] Neelakantan R. Krishnaswami and Nick Benton. 2011. Ultrametric Semantics of Reactive Programs. In *Logic in Computer Science (LICS'11).* IEEE.

[22] Joachim Lambek and Philip J. Scott. 1986. *Introduction to Higher-Order Categorical Logic.* Cambridge University Press.

[23] Conor McBride and Ross Paterson. 2008. Applicative Programming with Effects. *Journal of Functional Programming* 18, 1 (2008), 1–13.

[24] Paul-André Melliès and Noam Zeilberger. 2015. Functors Are Type Refinement Systems. In *Principles of Programming Languages (POPL'15).* ACM.

[25] Paul-André Melliès and Noam Zeilberger. 2016. A bifibrational reconstruction of Lawveres presheaf hyperdoctrine. In *Logic in Computer Science (LICS'16).* IEEE.

[26] Rasmus Ejlers Møgelberg. 2014. A type theory for productive coprogramming with guarded recursion. In *Logic in Computer Science (LICS'14).* IEEE.

[27] Hiroshi Nakano. 2000. A Modality for Recursion. In *Logic in Computer Science (LICS'00).* IEEE.

[28] Frank Pfenning and Rowan Davies. 2001. A Judgmental Reconstruction of Modal Logic. *Mathematical Structures in Computer Science* 11, 04 (Jul 2001).

[29] Florence Plateau. 2010. *Modèle n-synchrone pour la programmation de réseaux de Kahn à mémoire bornée.* Ph.D. Dissertation. Université Paris-Sud.

[30] Marc Pouzet. 2006. *Lucid Synchrone, version 3. Tutorial and reference manual.* Université Paris-Sud, LRI.

[31] Paula Severi. 2017. A Light Modality for Recursion. In *Foundations of Software Science and Computation Structures (FoSSaCS'17).* Springer.