Guarded Dependent Type Theory with Coinductive Types

Aleš Bizjak
Aarhus University, Denmark
abizjak@cs.au.dk

Hans Bugge Grathwohl
Aarhus University, Denmark
hbugge@cs.au.dk

Ranald Clouston
Aarhus University, Denmark
ranald.clouston@cs.au.dk

Rasmus E. Møgelberg
IT University of Copenhagen, Denmark
mogel@itu.dk

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

Abstract

We present guarded dependent type theory, gDTT, an extensional dependent type theory with a 'later' modality and clock quantifiers for programming and proving with guarded recursive and coinductive types. The later modality is used to ensure the productivity of recursive definitions in a modular, type based, way. Clock quantifiers are used for controlled elimination of the later modality and for encoding coinductive types using guarded recursive types. We demonstrate the expressiveness of gDTT via a range of examples. Key to the development of gDTT are novel type and term formers involving what we call 'delayed substitutions'. These generalise the applicative functor rules for the later modality considered in earlier work, and are crucial for programming and proving with dependent types. We show soundness of the type theory using a denotational model.

Categories and Subject Descriptors F.3.3 [*Logics and Meanings of Programs*]: Studies of Program Constructs

Keywords Dependent Type Theory, Guarded Recursion, Coinductive Types, Denotational Semantics

1. Introduction

Dependent type theory is useful both for programming, and for proving properties of elements of types. Modern implementations of dependent type theories such as Coq [20], Agda [24], Idris [9], and Nuprl [11], have been used successfully in many projects. However they offer limited support for programming and proving with *coinductive* types.

One of the key challenges is to ensure that functions on coinductive types are well-defined; that is, productive with unique solutions. Guarded recursion [12], as used for example in Coq [14], ensures productivity by requiring that recursive calls be nested directly under a constructor, but it is well known that such syntactic checks exclude many valid definitions, particularly in the presence of higher-order functions.

[Copyright notice will appear here once 'preprint' option is removed.]

To address this challenge, a *type-based* approach, more flexible than syntactic checks, was first suggested by Nakano [23]. A new modality, for which we follow Appel et. al. [3] by writing $\triangleright$ and using the name 'later', allows us to distinguish between data we have access to now, and data which we have only later. This $\triangleright$ must be used to guard self-reference in type definitions, so for example *guarded streams* of natural numbers are described by the guarded recursive equation

$$\mathsf{Str}^g_{\mathbb{N}} \simeq \mathbb{N} \times \triangleright \mathsf{Str}^g_{\mathbb{N}}$$

asserting that stream heads are available now, but tails only later.

Using $\triangleright$ alone, however, enforces a discipline more rigid than productivity. For example, all stream functions must be *causal* [17], so elements of the result must not depend on deeper elements of the argument, ruling out the 'every other' function that returns every second element of a given stream. This limitation motivated the introduction of *clock quantifiers* by Atkey and McBride [4], which permit $\triangleright$ to be eliminated in a controlled way; these have been extended to dependent types [8, 22]. This approach entails multiple later modalities $\overset{\kappa}{\triangleright}$, indexed by clocks κ, which represent the temporal dimensions on which types may depend. Access to multiple clocks may assist programming with data structures that are infinite in multiple dimensions, such as infinite-breadth non-well-founded trees, although the examples of this paper focus on simpler single-clocked structures. For example we have guarded streams for each clock κ, satisfying

$$\mathsf{Str}^\kappa_{\mathbb{N}} \simeq \mathbb{N} \times \overset{\kappa}{\triangleright} \mathsf{Str}^\kappa_{\mathbb{N}} .$$

Clock quantifiers can then give rise to types whose denotation is exactly that of standard coinductive types [22, Theorem 2]. Hence they allow us to program and prove with 'real' coinductive types, with guarded recursion providing the 'rule format' for their manipulation. For example, the coinductive type of streams of natural numbers can be defined simply by

$$\mathsf{Str}_{\mathbb{N}} \triangleq \forall\kappa.\, \mathsf{Str}^\kappa_{\mathbb{N}} .$$

We wish to construct proofs of properties of elements of guarded recursive and coinductive types. Clouston et. al. [10] introduced the logic $Lg\lambda$ to prove properties of terms of their (simply typed) guarded λ-calculus, gλ. This allowed proofs about coinductive types, but not in the integrated fashion supported by dependent type theories and, moreover, it relied on types being "total". Totality implied that every term of type $\triangleright A$ could be assumed to be of the form $\mathsf{next}\, t$, for some term t of type A (where next is the constructor for $\triangleright A$). This property cannot be used in dependent type theory because it would amount to a strong elimination rule for $\triangleright$, which would lead to inconsistency; see Section 2.2.

In this paper we introduce the extensional guarded dependent type theory gDTT, which supports programming and proving with guarded and coinductive dependent types.

Since one cannot rely on totality, one of the key challenges in designing gDTT is coping with elements that are only available later, i.e., elements of types of the form $\overset{\kappa}{\triangleright} A$. In Section 3 we present detailed concrete examples that illustrate this point, but the essence of this challenge can be understood more abstractly and intuitively. In gλ [10], there were term formation rules

$$\frac{\Gamma \vdash t : A}{\Gamma \vdash \mathsf{next}\, t : \triangleright A} \qquad \frac{\Gamma \vdash f : \triangleright(A \to B) \qquad \Gamma \vdash t : \triangleright A}{\Gamma \vdash f \circledast t : \triangleright B} \tag{1}$$

The first rule allows us to make later use of data that we have now. The second is part of the applicative functor [21] structure of $\triangleright$. Intuitively, if f will later be a function mapping A to B, and we will later have an element t of A, then, later, we can apply f to t to get an element of B.

Let us consider the $\circledast$ rule for dependent types, where function spaces are generalised to Π-types. Supposing that f has type $\triangleright(\Pi x : A.B)$, what type should $f \circledast t$ have? If B depends on x, it does not make sense to follow the simply-typed approach and use $\triangleright B$. But since t has type $\triangleright A$, and not A, we cannot substitute t for x. Intuitively, t will eventually reduce to some value $\mathsf{next}\, u$, and so the resulting type should be $\triangleright(B[u/x])$. But if t is an open term we may not be able to perform this reduction, so cannot type this term. To address this issue we introduce a new notion, of *delayed substitution*, allowing us to express the resulting type as:

$$\triangleright \left[x \leftarrow t\right] .B$$

binding x in B. Definitional equality rules will allow us to simplify this type when t has form $\mathsf{next}\, u$, i.e.,

$$\triangleright \left[x \leftarrow \mathsf{next}\, u\right] .B \simeq \triangleright(B[u/x])$$

as expected. Formally in gDTT we have for each clock κ a generalised $\overset{\kappa}{\triangleright}\xi.B$, where ξ is a 'delayed substitution' such as $[x \leftarrow t]$, defined formally in Section 2. In fact, given this structure $\circledast$ turns out to be definable, as shown in Section 2. We write $\textcircled{\scriptsize$\kappa$}$ for this defined operator $\circledast$ on clock κ.

The novel features of gDTT will allow us to address the programming and proving challenges posed by coinductive types. We will give detailed examples to substantiate this claim in Section 3, but we will here illustrate our point using a simple example on guarded, then coinductive, streams.

Let κ be some clock variable. In gDTT we can define addition of two guarded streams of natural numbers as follows:

$$\begin{aligned}
&\mathsf{plus}^\kappa : \mathsf{Str}^\kappa_{\mathbb{N}} \to \mathsf{Str}^\kappa_{\mathbb{N}} \to \mathsf{Str}^\kappa_{\mathbb{N}} \\
&\mathsf{plus}^\kappa \triangleq \mathsf{fix}^\kappa\, \phi.\lambda(xs : \mathsf{Str}^\kappa_{\mathbb{N}})(ys : \mathsf{Str}^\kappa_{\mathbb{N}}). \\
&\quad \mathsf{cons}^\kappa\, ((\mathsf{hd}^\kappa\, xs) + (\mathsf{hd}^\kappa\, ys))\ (\phi \textcircled{\scriptsize$\kappa$} \mathsf{tl}^\kappa\, xs \textcircled{\scriptsize$\kappa$} \mathsf{tl}^\kappa\, ys)
\end{aligned}$$

Here hd^κ takes the head of a stream, and tl^κ its tail (recalling that tails of guarded streams have type $\overset{\kappa}{\triangleright}\mathsf{Str}^\kappa_{\mathbb{N}}$). This function is defined by guarded recursion, so ϕ has type $\overset{\kappa}{\triangleright}(\mathsf{Str}^\kappa_{\mathbb{N}} \to \mathsf{Str}^\kappa_{\mathbb{N}} \to \mathsf{Str}^\kappa_{\mathbb{N}})$. The $\textcircled{\scriptsize$\kappa$}$-application $\phi \textcircled{\scriptsize$\kappa$} \mathsf{tl}^\kappa\, xs \textcircled{\scriptsize$\kappa$} \mathsf{tl}^\kappa\, ys$ has type $\overset{\kappa}{\triangleright}\mathsf{Str}^\kappa_{\mathbb{N}}$, so the $\mathsf{cons}^\kappa(\cdots)$ sub-expression has type $\mathsf{Str}^\kappa_{\mathbb{N}}$.

Now suppose that we want to prove the commutativity of addition on streams (assuming commutativity of $+$ on natural numbers). Then we want to construct a proof term of the following type:

$$\Pi(xs, ys : \mathsf{Str}^\kappa_{\mathbb{N}}).\mathsf{Id}_{\mathsf{Str}^\kappa_{\mathbb{N}}}(\mathsf{plus}^\kappa\, xs\, ys, \mathsf{plus}^\kappa\, ys\, xs) \tag{2}$$

Assuming a proof term c witnessing commutativity of addition on natural numbers, and a proof term $\mathsf{p}\eta$ yielding equality of two pairs when given proofs of equality of the first and second components of the pairs. Then we can construct a proof term p inhabiting the above type as follows:

$$\begin{aligned}
&p \triangleq \mathsf{fix}^\kappa\, \phi.\lambda\, (xs, ys : \mathsf{Str}^\kappa_{\mathbb{N}})\ . \\
&\quad \mathsf{p}\eta\ (c\, (\mathsf{hd}^\kappa\, xs)\, (\mathsf{hd}^\kappa\, ys))\ (\phi \textcircled{\scriptsize$\kappa$} \mathsf{tl}^\kappa\, xs \textcircled{\scriptsize$\kappa$} \mathsf{tl}^\kappa\, ys)
\end{aligned}$$

Note that p is again defined by guarded recursion, and that it is quite simple: it says that to show commutativity of plus^κ we proceed by using commutativity of $+$ on the heads, then continuing recursively on the tails. While p itself is simple, showing that p indeed has type (2) makes use of the new features in gDTT; we give details in Section 3.1. We remark that p, via the Curry-Howard isomorphism, corresponds to a proof by Löb induction (see e.g. Appel et al. [3]).

Following Clouston et al. [10], we can define a function $\mathfrak{L}$ which lifts plus^κ to a function on *coinductive* streams:

$$\begin{aligned}
&\mathsf{plus} : \mathsf{Str}_{\mathbb{N}} \to \mathsf{Str}_{\mathbb{N}} \to \mathsf{Str}_{\mathbb{N}} \\
&\mathsf{plus} \triangleq \mathfrak{L}(\mathsf{plus}^\kappa)
\end{aligned}$$

Likewise, using the new features of gDTT, we can take the proof p of commutativity of plus^κ and turn it into a proof of commutativity of plus, i.e., construct an element of type

$$\Pi(xs, ys : \mathsf{Str}_{\mathbb{N}}).\mathsf{Id}_{\mathsf{Str}_{\mathbb{N}}}(\mathsf{plus}\, xs\, ys, \mathsf{plus}\, ys\, xs) \tag{3}$$

We detail how this is done in Section 6.

Thus guarded recursion underlies a type theory for programming and proving properties on guarded streams, and then converting these into programs and proofs for coinductive streams.

Contributions The contributions of this paper are:

- We introduce a new guarded dependent type theory gDTT, and show that it supports both programming and proving for guarded recursive and coinductive types. Key features are delayed substitutions and clock quantifiers.
- We prove the soundness of gDTT by means of a model involving families of sets indexed over natural numbers, one for each clock variable. The core part of the model has been used in earlier work on guarded recursive types and clock quantifiers [8, 22]; here we show that it also models delayed substitutions.

We focus on the design and soundness of the type theory and restrict attention to an extensional type theory. We postpone a treatment of an intensional version of the theory to future work (see the discussion of related and future work in Sections 8 and 9).

In addition to the examples included in this paper, we are pleased to report that a preliminary version of gDTT has already proved crucial for formalizing a logical relations adequacy proof of a semantics for PCF using guarded recursive types by Paviotti et. al. [25].

Outline The paper is organized as follows. In Section 2, we give an introduction to gDTT. We recall how to define guarded recursive functions and guarded recursive types, and introduce the new constructs of gDTT. We only consider guarded recursive types and postpone a treatment of coinductive types, using clock quantifiers, to Section 4. After the introduction to gDTT, we present, in Section 3, several worked examples, which illustrate how the rules of gDTT can be used to program and prove with guarded recursive types. In Section 6 we present further examples involving coinductive types. In Section 7 we outline how to show soundness of gDTT, and in Section 8 we discuss related work. Finally, we conclude and discuss future work in Section 9.

2. Guarded Dependent Type Theory

We fix a countable set of *clock variables* $\mathrm{CV} = \{\kappa_1, \kappa_2, \cdots\}$ and a single *clock constant* κ_0. A *clock* is either a clock variable or the

$\vdash_\Delta \kappa$	valid clock
$\Gamma \vdash_\Delta$	well-formed context
$\Gamma \vdash_\Delta A \ \mathsf{type}$	well-formed type
$\Gamma \vdash_\Delta t : A$	typing judgment
$\Gamma \vdash_\Delta A \simeq B$	type equality
$\Gamma \vdash_\Delta t \equiv u : A$	term equality
$\vdash_\Delta \xi : \Gamma \overset{\kappa}{\to} \Gamma'$	delayed substitution

Figure 1. Judgements in gDTT.

clock constant. A *clock context* is a finite set of *clock variables*. We write $\Delta, \Delta', \cdots$ for clock contexts. We use the judgement $\vdash_\Delta \kappa$ to express that either κ is a clock variable in the set Δ or κ is the clock constant κ_0.

gDTT is a type theory with unit $\mathbf{1}$, booleans $\mathbf{B}$, and natural numbers $\mathbf{N}$ as base types, along with Π-types, Σ-types, identity types, and universes. Our universes are à la Tarski, so we distinguish between types and terms, and have terms that represent types; they are called *codes* of types and they can be recognised by their circumflex, e.g., $\widehat{\mathbf{N}}$ is the code of the type $\mathbf{N}$. We have a map El from terms to types which sends codes of types to their corresponding type. We follow standard practice and often omit El in examples, except where it is important to avoid confusion. An overview of the different judgments in the theory is given in Figure 1.

All judgements are parametrised by a *clock context* Δ. Clocks should be thought of as temporal dimensions on which types may depend. Codes of types inhabit *universes*, denoted $\mathcal{U}_\Delta$, where the Δ is a finite set of clock variables. The universe $\mathcal{U}_\Delta$ is only well-formed in clock contexts Δ' where $\Delta \subseteq \Delta'$. Intuitively, $\mathcal{U}_\Delta$ contains codes of types that can vary only along dimensions in Δ. We have *universe inclusions* from $\mathcal{U}_\Delta$ to $\mathcal{U}_{\Delta'}$ whenever $\Delta \subseteq \Delta'$. In the examples we will not write the inclusion explicitly, since type equalities ensure that it does not matter at which universe we use El to map a code of a type to a type. Note that *we do not have* $\mathcal{U}_\Delta : \mathcal{U}_{\Delta'}$, i.e., these universes do not form a hierarchy. We could additionally have an orthogonal hierarchy of universes, i.e. for each clock context Δ a hierarchy of universes $\mathcal{U}^1_\Delta : \mathcal{U}^2_\Delta : \cdots$, but omit this for space reasons.

All judgements are closed under clock weakening and clock substitution. The former means that if, e.g., $\Gamma \vdash_\Delta t : A$ is derivable then, for any clock variable $\kappa \notin \Delta$, the judgement $\Gamma \vdash_{\Delta,\kappa} t : A$ is also derivable. The latter means that if, e.g., $\Gamma \vdash_{\Delta,\kappa} t : A$ is derivable and $\vdash_\Delta \kappa'$ then the judgement $\Gamma[\kappa'/\kappa] \vdash_\Delta t[\kappa'/\kappa] : A[\kappa'/\kappa]$ is also derivable. Note that we use the same notation $[\kappa'/\kappa]$ for clock substitution as for ordinary term substitution. Clock substitution behaves as expected, but we point out some noteworthy cases.

$$
\begin{aligned}
(\overset{\kappa}{\triangleright} A)[\kappa'/\kappa] &= \overset{\kappa'}{\triangleright} A[\kappa'/\kappa] \\
(\mathsf{next}^\kappa\, t)[\kappa'/\kappa] &= \mathsf{next}^{\kappa'}\, t[\kappa'/\kappa] \\
(\mathsf{fix}^\kappa\, x.t)[\kappa'/\kappa] &= \mathsf{fix}^{\kappa'}\, x.t[\kappa'/\kappa] \\
(\mathcal{U}_{\Delta',\kappa})[\kappa'/\kappa] &= \mathcal{U}_{\Delta'} && \text{if } \kappa' \in \Delta' \\
(\mathcal{U}_{\Delta',\kappa})[\kappa'/\kappa] &= \mathcal{U}_{\Delta',\kappa'} && \text{if } \kappa' \notin \Delta' \\
(\mathcal{U}_{\Delta'})[\kappa'/\kappa] &= \mathcal{U}_{\Delta'} && \text{if } \kappa \notin \Delta'.
\end{aligned}
$$

The rules for guarded recursion can be found in Figure 2; rules for coinductive types will be postponed until Section 4.

Recall the 'later' type former $\triangleright$, which expresses that something will be available at a later time. In gDTT we have $\overset{\kappa}{\triangleright}$, for each clock κ, so we can delay a type along different dimensions. Next we generalise earlier work so that the applicative functor structure of each $\overset{\kappa}{\triangleright}$ can work on dependent function types. To do this, we introduce *delayed substitutions*, which allow a substitution to be delayed until its substituent is available. We showed in the introduction how a type with a single delayed substitution $\overset{\kappa}{\triangleright} [x \leftarrow t] .A$ should work. However if we have a term f with more than one argument, for example of type $\overset{\kappa}{\triangleright} (\Pi(x : A).\Pi(y : B).C)$, and wish to type an application $f \textcircled{\kappa} t \textcircled{\kappa} u$, we may have neither t nor u available now, and so we need sequences of delayed substitutions to define the type $\overset{\kappa}{\triangleright} [x \leftarrow t, y \leftarrow u] .C$. Our concrete examples of Section 3 will show that this issue arises in practice. We therefore define sequences of delayed substitutions ξ. The new raw types, terms, and delayed substitutions of gDTT are given by the grammar

$$
\begin{aligned}
A, B &::= \cdots \mid \overset{\kappa}{\triangleright}\xi.A \\
t, u &::= \cdots \mid \mathsf{next}^\kappa\, \xi.t \mid \widehat{\triangleright}^\kappa t \\
\xi &::= \cdot \mid \xi\, [x \leftarrow t]\,.
\end{aligned}
$$

Note that we just write $\overset{\kappa}{\triangleright} A$ where its delayed substitution is the empty $\cdot$, and that $\overset{\kappa}{\triangleright}\xi.A$ binds the variables substituted for by ξ in A, and similarly for next.

The three rules DS-Emp, DS-Cons, and Tf-$\triangleright$ are used to construct the type $\overset{\kappa}{\triangleright}\xi.A$. The special cases where $\xi = \cdot$ and $\xi = [x \leftarrow t]$ are the situations that are described in the introduction. These rules formulate how to generalise these types to arbitrarily long delayed substitutions. Once the type formation rule is established, the introduction rule Ty-Next is the natural one.

With delayed substitutions we can *define* $\textcircled{\kappa}$ as

$$
f \textcircled{\kappa} t \triangleq \mathsf{next}^\kappa \begin{bmatrix} g \leftarrow f \\ x \leftarrow t \end{bmatrix} .g\, x
$$

Using the rules in Figure 2 we can derive the following typing judgement for $\textcircled{\kappa}$

$$
\frac{\Gamma \vdash_\Delta f : \overset{\kappa}{\triangleright}\xi.\Pi(x : A).B \qquad \Gamma \vdash_\Delta t : \overset{\kappa}{\triangleright}\xi.A}{\Gamma \vdash_\Delta f \textcircled{\kappa} t : \overset{\kappa}{\triangleright}\xi[x \leftarrow t].B}\ \text{Ty-}\circledast
$$

which was the motivation for introducing delayed substitutions.

When a term has the form $\mathsf{next}^\kappa\, \xi\, [x \leftarrow \mathsf{next}^\kappa\, \xi.u]\, .t$, then we have enough information to perform the substitution in both the term and its type. The rule TmEq-Force forces the substitution to go through by equating the term with the result of an actual substitution, $\mathsf{next}^\kappa\, \xi.t[u/x]$. The rule TyEq-Force does the same for its type. Using TmEq-$\circledast$ we can derive the basic term equality

$$
(\mathsf{next}^\kappa\, \xi.f) \textcircled{\kappa} (\mathsf{next}^\kappa\, \xi.t) \equiv \mathsf{next}^\kappa\, \xi.(f t).
$$

typical of applicative functors [21].

Often it will be the case that a delayed substitution is unnecessary, because the variable to be substituted for does not occur free in the type/term. This is what TyEq-$\triangleright$-Weak and TmEq-Next-Weak express, and with these we can justify the simpler typing rule

$$
\frac{\Gamma \vdash_\Delta f : \overset{\kappa}{\triangleright}\xi.(A \to B) \qquad \Gamma \vdash_\Delta t : \overset{\kappa}{\triangleright}\xi.A}{\Gamma \vdash_\Delta f \textcircled{\kappa} t : \overset{\kappa}{\triangleright}\xi.B}
$$

the special case of which is the typing rule (1). In other words, we need not bother with delayed substitutions on the type when we apply a non-dependent function.

Further, we have the applicative functor identity law

$$
(\mathsf{next}^\kappa\, \xi.\lambda x.x) \textcircled{\kappa} t \equiv t.
$$

This follows from the rule TmEq-Next-Var, which allows us to simplify a term $\mathsf{next}^\kappa\, \xi\, [y \leftarrow t]\, .y$ to t.

Sometimes it is necessary to switch the order in the delayed substitution. Two substitutions can switch places, as long as they do not depend on each other; this is what TyEq-$\triangleright$-Exch and TmEq-Next-Exch express.

2.1 Fixed points and guarded recursive types

A crucial concept in the theory of guarded recursion is the guarded recursive fixed-point combinator fix^κ. In gDTT we have for each clock κ valid in the current clock context a fixed-point combinator fix^κ. It differs from a traditional fixed-point combinator in that the type of the recursion variable is not the same as the result type. Instead it is *guarded* with $\overset{\kappa}{\triangleright}$. More precisely, the type of recursion variable for the combinator fix^κ is guarded by $\overset{\kappa}{\triangleright}$ (rule Ty-Fix).

When we define a term using the fixed-point, we say that it is defined by *guarded recursion*. When the term is intuitively a proof, e.g. it is of the identity type, then we say we are proving by *Löb induction* [3].

Guarded recursive types are defined as fixed-points of suitably guarded functions on universes. This is the approach of Birkedal and Møgelberg [6], but the generality of the rules of gDTT allows us to define more interesting dependent guarded recursive types, for example the predicates of Section 3.3.

We first illustrate the technique by defining the (non-dependent) type of guarded streams. Recall from the introduction that we want the type of guarded streams of elements of type A, for clock κ, to satisfy the equation

$$\mathsf{Str}^\kappa_A \simeq A \times \overset{\kappa}{\triangleright} \mathsf{Str}^\kappa_A\,.$$

The type A will be equal to $\mathsf{El}(B)$ for some code B in some universe $\mathcal{U}_\Delta$ and the clock variable κ is fresh for Δ, i.e., *not* in Δ. We then define the *code* S^κ_A of Str^κ_A in the universe $\mathcal{U}_{\Delta,\kappa}$ to be

$$\begin{aligned} &S^\kappa_A : \mathcal{U}_{\Delta,\kappa} \\ &S^\kappa_A \triangleq \mathsf{fix}^\kappa\, X.B \mathbin{\widehat{\times}} \widehat{\triangleright}^\kappa X, \end{aligned}$$

where $\widehat{\times}$ is the code of the (simple) product type. Let us check that S^κ_A is well typed: X has the type $\overset{\kappa}{\triangleright}\mathcal{U}_{\Delta,\kappa}$, so by Ty-$\widehat{\triangleright}$ we can see that $\widehat{\triangleright}^\kappa X$ has the type $\mathcal{U}_{\Delta,\kappa}$. Therefore $B \mathbin{\widehat{\times}} \widehat{\triangleright}^\kappa X$ has the type $\mathcal{U}_{\Delta,\kappa}$ too, and by Ty-Fix, S^κ_A has type $\mathcal{U}_{\Delta,\kappa}$. Using TmEq-Fix we get that $S^\kappa_A \equiv B \mathbin{\widehat{\times}} \widehat{\triangleright}^\kappa\, \mathsf{next}^\kappa\, S_A$. So defining $\mathsf{Str}^\kappa_A \triangleq \mathsf{El}(S^\kappa_A)$ we get

$$\begin{aligned} \mathsf{Str}^\kappa_A &\simeq \mathsf{El}(S^\kappa_A) \\ &\simeq \mathsf{El}(B \mathbin{\widehat{\times}} \widehat{\triangleright}^\kappa\, \mathsf{next}^\kappa\, S^\kappa_A) && \text{(by TmEq-Fix)} \\ &\simeq \mathsf{El}(B) \times \mathsf{El}(\widehat{\triangleright}^\kappa\, \mathsf{next}^\kappa\, S^\kappa_A) && \text{(by eq. rules for } \widehat{\times}) \\ &\simeq \mathsf{El}(B) \times \overset{\kappa}{\triangleright} \mathsf{El}\,(S^\kappa_A) && \text{(by TyEq-El-}\triangleright) \\ &\simeq A \times \overset{\kappa}{\triangleright} \mathsf{Str}^\kappa_A, \end{aligned}$$

which is what we wanted. We can define the head and tail operations, $\mathsf{hd}^\kappa : \mathsf{Str}^\kappa_A \to A$ and $\mathsf{tl}^\kappa : \mathsf{Str}^\kappa_A \to \overset{\kappa}{\triangleright} \mathsf{Str}^g_A$ to simply be the first and the second projections. Conversely, we construct streams by pairing. To make the examples more readable we will use the suggestive cons^κ notation which we define as

$$\begin{aligned} &\mathsf{cons}^\kappa : A \to \overset{\kappa}{\triangleright} \mathsf{Str}^\kappa_A \to \mathsf{Str}^\kappa_A \\ &\mathsf{cons}^\kappa \triangleq \lambda\,(a : A)\left(as : \overset{\kappa}{\triangleright} \mathsf{Str}^\kappa_A\right).\,\langle a, as\rangle \end{aligned}$$

Defining guarded streams is also done via guarded recursion, for example the stream consisting only of ones:

$$\begin{aligned} &\mathsf{ones} : \mathsf{Str}^\kappa_{\mathbb{N}} \\ &\mathsf{ones} \triangleq \mathsf{fix}^\kappa\, x.\, \mathsf{cons}^\kappa\, 1\, x. \end{aligned}$$

The rule TyEq-El-$\triangleright$ is essential for defining guarded recursive types as fixed-points on universes, and it can also be used for defining more advanced guarded recursive dependent types such as covectors. We will see examples of this in Sections 3.2 and 3.3.

2.2 Identity types

gDTT has standard extensional identity types $\mathsf{Id}_A(t, u)$ (see e.g., Jacobs [16]) but with two additional type equivalences necessary for working with guarded dependent types. We write $\mathsf{r}_A\, t$ for the reflexivity proof $\mathsf{Id}_A(t, t)$. The first type equivalence is the rule TyEq-$\triangleright$ which states the type equality

$$\mathsf{Id}_{\overset{\kappa}{\triangleright}\xi.A}(\mathsf{next}^\kappa\, \xi.t, \mathsf{next}^\kappa\, \xi.u) \simeq \overset{\kappa}{\triangleright}\xi.\mathsf{Id}_A(t, u).$$

This rule, which will be validated by the model of Section 7, may be thought of by analogy to type equivalences often considered in homotopy type theory [27], such as

$$\mathsf{Id}_{A\times B}(\langle s_1, s_2\rangle\,, \langle t_1, t_2\rangle) \simeq \mathsf{Id}_A(s_1, t_1) \times \mathsf{Id}_B(s_2, t_2). \qquad (4)$$

However there are two important differences. The first one is that (4) is (using univalence) a propositional type equality whereas TyEq-$\triangleright$ specifices a definitional type equality. This is natural in an extensional type theory. The second difference is that there are terms going in both directions in (4) and additional axioms are only needed to show equality, whereas in the calculus without TyEq-$\triangleright$ we only have the term

$$\lambda p.\, \mathsf{next}^\kappa\, \xi.\mathsf{r}_A\, t$$

of type

$$\mathsf{Id}_{\overset{\kappa}{\triangleright}\xi.A}(\mathsf{next}^\kappa\, \xi.t, \mathsf{next}^\kappa\, \xi.u) \to \overset{\kappa}{\triangleright}\xi.\mathsf{Id}_A(t, u)$$

using equality reflection and no term in the other direction, which is the direction crucial in examples. We elect to make the types equal instead of introducing a new term construct and adding type equalities witnessing that it is in fact the inverse. The reason there can be no such term is that $\overset{\kappa}{\triangleright}$ does not have a strong elimination or induction principle. If such a principle for $\overset{\kappa}{\triangleright}$ was allowed, then we would be able to construct, using the "theorem of choice", a term t of type

$$\Sigma(f : \overset{\kappa}{\triangleright}A \to A).\Pi(x : \overset{\kappa}{\triangleright}A).\mathsf{Id}_{\overset{\kappa}{\triangleright}A}(\mathsf{next}^\kappa\, f(x), x),$$

which would allow us to construct an inhabitant $\mathsf{fix}^\kappa\, x.\pi_1(t)\, x$ of any type A, rendering the type theory inconsistent.

The second novel type equality rule, which involves clock quantification, will be presented in Section 4.

Using TyEq-$\triangleright$ there is a derivable "commuting conversion" term equality rule

$$\frac{\Gamma \vdash_\Delta \mathsf{next}^\kappa\, \xi.\mathsf{r}_A\, t : \overset{\kappa}{\triangleright}\xi.\mathsf{Id}_A(t, t)}{\Gamma \vdash_\Delta \mathsf{next}^\kappa\, \xi.\mathsf{r}_A\, t \equiv \mathsf{r}_{\overset{\kappa}{\triangleright}\xi.A}\, \mathsf{next}^\kappa\, \xi.t : \mathsf{Id}_{\overset{\kappa}{\triangleright}\xi.A}\begin{pmatrix} \mathsf{next}^\kappa\, \xi.t, \\ \mathsf{next}^\kappa\, \xi.t \end{pmatrix}}$$

which is derivable because in an extensional type theory all proofs of equality are definitionally equal [16, Proposition 10.1.3].

3. Examples

The examples below are designed to highlight the capabilities of gDTT and to show how the rules are used. More precisely:

- The example in Section 3.1 is the simplest and shows a basic property of a function on streams. It shows in detail how delayed substitutions are used and why TyEq-$\triangleright$ is necessary.
- The example in Section 3.2 shows an example with covectors where even defining basic functions requires delayed substitutions. The example also shows an example proof of a property of a function defined on covectors. This proof shows why TmEq-Force is necessary.

Universes

$$\frac{\Delta' \subseteq \Delta \qquad \Gamma \vdash_\Delta}{\Gamma \vdash_\Delta \mathcal{U}_{\Delta'}\ \mathsf{type}}\ \textsc{Univ} \qquad\qquad \frac{\Gamma \vdash_\Delta A : \mathcal{U}_{\Delta'}}{\Gamma \vdash_\Delta \mathsf{El}(A)\ \mathsf{type}}\ \textsc{El}$$

Delayed substitutions:

$$\frac{\Gamma \vdash_\Delta \qquad \vdash_\Delta \kappa}{\vdash_\Delta \cdot : \Gamma \xrightarrow{\kappa} \cdot}\ \text{DS-}\textsc{Emp} \qquad\qquad \frac{\vdash_\Delta \xi : \Gamma \xrightarrow{\kappa} \Gamma' \qquad \Gamma \vdash_\Delta t : \overset{\kappa}{\triangleright}\xi.A}{\vdash_\Delta \xi\,[x \leftarrow t] : \Gamma \xrightarrow{\kappa} \Gamma', x : A}\ \text{DS-}\textsc{Cons}$$

Type formation:

$$\frac{\Gamma, \Gamma' \vdash_\Delta A\ \mathsf{type} \qquad \vdash_\Delta \xi : \Gamma \xrightarrow{\kappa} \Gamma'}{\Gamma \vdash_\Delta \overset{\kappa}{\triangleright}\xi.A\ \mathsf{type}}\ \textsc{Tf-}\triangleright$$

Typing rules:

$$\frac{\Gamma, \Gamma' \vdash_\Delta t : A \qquad \vdash_\Delta \xi : \Gamma \xrightarrow{\kappa} \Gamma'}{\Gamma \vdash_\Delta \mathsf{next}^\kappa\,\xi.t : \overset{\kappa}{\triangleright}\xi.A}\ \textsc{Ty-Next} \qquad \frac{\vdash_\Delta \kappa \qquad \Gamma, x : \overset{\kappa}{\triangleright}A \vdash_\Delta t : A}{\Gamma \vdash_\Delta \mathsf{fix}^\kappa\,x.t : A}\ \textsc{Ty-Fix} \qquad \frac{\vdash_{\Delta'} \kappa \qquad \Gamma \vdash_\Delta A : \overset{\kappa}{\triangleright}\mathcal{U}_{\Delta'}}{\Gamma \vdash_\Delta \widehat{\triangleright}^\kappa A : \mathcal{U}_{\Delta'}}\ \textsc{Ty-}\widehat{\triangleright}$$

Definitional type equalities:

$$\frac{\Gamma, \Gamma' \vdash_\Delta A\ \mathsf{type} \qquad \vdash_\Delta \xi[x \leftarrow t] : \Gamma \xrightarrow{\kappa} \Gamma', x : B}{\Gamma \vdash_\Delta \overset{\kappa}{\triangleright}\xi\,[x \leftarrow t]\,.A \simeq \overset{\kappa}{\triangleright}\xi.A}\ \textsc{TyEq-}\triangleright\textsc{-Weak}$$

$$\frac{\Gamma, \Gamma', x : B, y : C, \Gamma'' \vdash_\Delta A\ \mathsf{type} \qquad \vdash_\Delta \xi\,[x \leftarrow t, y \leftarrow u]\,\xi' : \Gamma \xrightarrow{\kappa} \Gamma', x : B, y : C, \Gamma'' \qquad x \text{ not free in } C}{\Gamma \vdash_\Delta \overset{\kappa}{\triangleright}\xi\,[x \leftarrow t, y \leftarrow u]\,\xi'.A \simeq \overset{\kappa}{\triangleright}\xi\,[y \leftarrow u, x \leftarrow t]\,\xi'.A}\ \textsc{TyEq-}\triangleright\textsc{-Exch}$$

$$\frac{\Gamma \vdash_\Delta \overset{\kappa}{\triangleright}\xi\,[x \leftarrow \mathsf{next}^\kappa\,\xi.t]\,.A\ \mathsf{type}}{\Gamma \vdash_\Delta \overset{\kappa}{\triangleright}\xi\,[x \leftarrow \mathsf{next}^\kappa\,\xi.t]\,.A \simeq \overset{\kappa}{\triangleright}\xi.A[t/x]}\ \textsc{TyEq-Force} \qquad \frac{\Delta' \subseteq \Delta \qquad \vdash_{\Delta'} \kappa \qquad \Gamma, \Gamma' \vdash_\Delta A : \mathcal{U}_{\Delta'} \qquad \vdash_\Delta \xi : \Gamma \xrightarrow{\kappa} \Gamma'}{\Gamma \vdash_\Delta \mathsf{El}(\widehat{\triangleright}^\kappa\,(\mathsf{next}^\kappa\,\xi.A)) \simeq \overset{\kappa}{\triangleright}\xi.\,\mathsf{El}(t)}\ \textsc{TyEq-El-}\triangleright$$

$$\frac{\vdash_\Delta \xi : \Gamma \xrightarrow{\kappa} \Gamma' \qquad \Gamma, \Gamma' \vdash_\Delta t : A \qquad \Gamma, \Gamma' \vdash_\Delta s : A}{\Gamma \vdash_\Delta \mathsf{Id}_{\overset{\kappa}{\triangleright}\xi.A}(\mathsf{next}^\kappa\,\xi.t, \mathsf{next}^\kappa\,\xi.s) \simeq \overset{\kappa}{\triangleright}\xi.\mathsf{Id}_A(t, s)}\ \textsc{TyEq-}\triangleright$$

Definitional term equalities:

$$\frac{\Gamma, \Gamma' \vdash_\Delta u : A \qquad \vdash_\Delta \xi\,[x \leftarrow t] : \Gamma \xrightarrow{\kappa} \Gamma', x : B}{\Gamma \vdash_\Delta \mathsf{next}^\kappa\,\xi\,[x \leftarrow t]\,.u \equiv \mathsf{next}^\kappa\,\xi.u : \overset{\kappa}{\triangleright}\xi.A}\ \textsc{TmEq-Next-Weak} \qquad \frac{\Gamma \vdash_\Delta t : \overset{\kappa}{\triangleright}\xi.A}{\Gamma \vdash_\Delta \mathsf{next}^\kappa\,\xi\,[x \leftarrow t]\,.x \equiv t : \overset{\kappa}{\triangleright}\xi.A}\ \textsc{TmEq-Next-Var}$$

$$\frac{\Gamma, \Gamma', x : B, y : C, \Gamma'' \vdash_\Delta t : A \qquad \vdash_\Delta \xi\,[x \leftarrow t, y \leftarrow u]\,\xi' : \Gamma \xrightarrow{\kappa} \Gamma', x : B, y : C, \Gamma'' \qquad x \text{ not free in } C}{\Gamma \vdash_\Delta \mathsf{next}^\kappa\,\xi\,[x \leftarrow t, y \leftarrow u]\,\xi'.v \equiv \mathsf{next}^\kappa\,\xi\,[y \leftarrow u, x \leftarrow t]\,\xi'.v : \overset{\kappa}{\triangleright}\xi\,[y \leftarrow u, x \leftarrow t]\,\xi'.A}\ \textsc{TmEq-Next-Exch}$$

$$\frac{\Gamma \vdash_\Delta \mathsf{next}^\kappa\,\xi\,[x \leftarrow \mathsf{next}^\kappa\,\xi.t]\,.u : \overset{\kappa}{\triangleright}\xi\,[x \leftarrow \mathsf{next}^\kappa\,\xi.t]\,.A}{\Gamma \vdash_\Delta \mathsf{next}^\kappa\,\xi\,[x \leftarrow \mathsf{next}^\kappa\,\xi.t]\,.u \equiv \mathsf{next}^\kappa\,\xi.u[t/x] : \overset{\kappa}{\triangleright}\xi.A[t/x]}\ \textsc{TmEq-Force} \qquad \frac{\Gamma \vdash_\Delta \mathsf{fix}^\kappa\,x.t : A}{\Gamma \vdash_\Delta \mathsf{fix}^\kappa\,x.t \equiv t[\mathsf{next}^\kappa\,(\mathsf{fix}^\kappa\,x.t)\,/x] : A}\ \textsc{TmEq-Fix}$$

Figure 2. Overview of new rules in gDTT.

- The example in Section 3.3 shows how to first lift a predicate on some type A to a predicate on Str^κ_A and then derive a property of a lifted predicate from the property of the original predicate.
- The example in Section 5 shows programming with (dependent) coinductive types involving binary sums. It uses (derivable) type isomorphisms discussed in Section 4.1.
- The examples in Section 6.1 show how to (1) derive functions on coinductive types from functions on guarded types and (2) how to also derive properties of derived functions from properties of the original functions.
- The example in Section 6.2 shows how to prove properties of acausal functions.

In the examples we will frequently use two facts. First, there is a term, which we call $\mathsf{p}\eta$ (leaving the types A and B implicit to avoid clutter), of type

$$\Pi(s, t : A \times B).\mathsf{Id}_A(\pi_1 t, \pi_1 s) \to \mathsf{Id}_B(\pi_2 t, \pi_2 s) \to \mathsf{Id}_{A \times B}(t, s)$$

stating that two elements of the product type are equal if their first and second projections are equal. This is provable in any type theory with a strong (dependent) elimination rule for dependent sums.

The second property we will use is that $\mathsf{Str}^\kappa_A \simeq A \times \overset{\kappa}{\triangleright}\mathsf{Str}^\kappa_A$, i.e., that the type of streams is convertible to a product type. Because hd^κ and and tl^κ are simply first and second projections, if we show that heads and tails of two streams are equal then, using $\mathsf{p}\eta$, we have that the two streams are equal. That is, $\mathsf{p}\eta$ also has type

$$\begin{aligned}&\Pi\,(xs, ys : \mathsf{Str}^{\kappa}_{A})\,.\\ &\mathsf{Id}_{A}(\mathsf{hd}^{\kappa}\,xs, \mathsf{hd}^{\kappa}\,ys) \to \mathsf{Id}_{\overset{\kappa}{\triangleright}\mathsf{Str}^{\kappa}_{A}}(\mathsf{tl}^{\kappa}\,xs, \mathsf{tl}^{\kappa}\,ys)\\ &\to \mathsf{Id}_{\mathsf{Str}^{\kappa}_{A}}(xs, ys).\end{aligned}$$

3.1 zw^{κ} preserves commutativity

The first proof is the simplest. We will define the standard zw^{κ} (zipWith) function on streams and show that if a binary function f is commutative, then so is $\mathsf{zw}^{\kappa}\,f$. Note that this generalizes the example outlined in the introduction, since $\mathsf{plus}^{\kappa} \triangleq \mathsf{zw}^{\kappa}\,+$, where $+$ is the addition function on natural numbers.

The $\mathsf{zw}^{\kappa} : (A \to B \to C) \to \mathsf{Str}^{\kappa}_{A} \to \mathsf{Str}^{\kappa}_{B} \to \mathsf{Str}^{\kappa}_{C}$ is defined by guarded recursion as

$$\begin{aligned}\mathsf{zw}^{\kappa}\,f \triangleq \mathsf{fix}^{\kappa}\,\phi.\lambda(xs : \mathsf{Str}^{\kappa}_{A})(ys : \mathsf{Str}^{\kappa}_{B}).&\\ \mathsf{cons}^{\kappa}\,(f\,(\mathsf{hd}^{\kappa}\,xs)\,(\mathsf{hd}^{\kappa}\,ys))&\\ (\phi \textcircled{\kappa}\, \mathsf{tl}^{\kappa}\,xs \textcircled{\kappa}\, \mathsf{tl}^{\kappa}\,ys)&\end{aligned}$$

Note that none of the new $\triangleright$ and $\circledast$ rules of gDTT are needed to type this function; this is a function on simple types.

Where we need dependent types is, of course, to state and prove properties. To prove our example, that commutativity of f implies commutativity of $\mathsf{zw}^{\kappa}\,f$, means we must show that the type

$$\begin{aligned}&\Pi(f : A \to A \to B).\,(\Pi(x, y : A).\mathsf{Id}_{B}(f\,x\,y, f\,y\,x)) \to\\ &\Pi(xs, ys : \mathsf{Str}^{\kappa}_{A}).\mathsf{Id}_{\mathsf{Str}^{\kappa}_{B}}(\mathsf{zw}^{\kappa}\,f\,xs\,ys, \mathsf{zw}^{\kappa}\,f\,ys\,xs).\end{aligned}$$

is inhabited. We will explain how to construct such a term, and why it is typeable in gDTT. Although this construction might appear complicated at first, the actual proof term that we construct will be as simple as possible.

Let $f : A \to A \to B$ be a function and say we have a term

$$c : \Pi(x, y : A).\mathsf{Id}_{B}(f\,x\,y, f\,y\,x)$$

witnessing commutativity of f. We now wish to construct a term of type

$$\Pi(xs, ys : \mathsf{Str}^{\kappa}_{A}).\mathsf{Id}_{\mathsf{Str}^{\kappa}_{C}}(\mathsf{zw}^{\kappa}\,f\,xs\,ys, \mathsf{zw}^{\kappa}\,f\,ys\,xs)$$

We do this by guarded recursion. To this end we assume

$$\phi : \overset{\kappa}{\triangleright}\left(\Pi(xs, ys : \mathsf{Str}^{\kappa}_{A}).\mathsf{Id}_{\mathsf{Str}^{\kappa}_{B}}(\mathsf{zw}^{\kappa}\,f\,xs\,ys, \mathsf{zw}^{\kappa}\,f\,ys\,xs)\right)$$

and take $xs, ys : \mathsf{Str}^{\kappa}_{A}$. Using c (the proof that f is commutative) we first have $c\,(\mathsf{hd}^{\kappa}\,xs)\,(\mathsf{hd}^{\kappa}\,ys)$ of type

$$\mathsf{Id}_{B}(f\,(\mathsf{hd}^{\kappa}\,xs)\,(\mathsf{hd}^{\kappa}\,ys), f\,(\mathsf{hd}^{\kappa}\,ys)\,(\mathsf{hd}^{\kappa}\,xs))$$

and because we have by definition of zw^{κ}

$$\begin{aligned}\mathsf{hd}^{\kappa}\,(\mathsf{zw}^{\kappa}\,f\,xs\,ys) &\equiv f\,(\mathsf{hd}^{\kappa}\,xs)\,(\mathsf{hd}^{\kappa}\,ys)\\ \mathsf{hd}^{\kappa}\,(\mathsf{zw}^{\kappa}\,f\,ys\,xs) &\equiv f\,(\mathsf{hd}^{\kappa}\,ys)\,(\mathsf{hd}^{\kappa}\,xs)\end{aligned}$$

we see that $c\,(\mathsf{hd}^{\kappa}\,xs)\,(\mathsf{hd}^{\kappa}\,ys)$ has type

$$\mathsf{Id}_{B}(\mathsf{hd}^{\kappa}\,(\mathsf{zw}^{\kappa}\,f\,xs\,ys), \mathsf{hd}^{\kappa}\,(\mathsf{zw}^{\kappa}\,f\,ys\,xs)).$$

To show that the tails are equal we use the induction hypothesis ϕ. The terms $\mathsf{tl}^{\kappa}\,xs$ and $\mathsf{tl}^{\kappa}\,ys$ are of type $\overset{\kappa}{\triangleright}\mathsf{Str}^{\kappa}_{A}$, so we first have $\phi \textcircled{\kappa}\, \mathsf{tl}^{\kappa}\,xs$ of type

$$\overset{\kappa}{\triangleright}[xs \leftarrow \mathsf{tl}^{\kappa}\,xs]\,.\left(\Pi\,(ys : \mathsf{Str}^{\kappa}_{A})\,.\mathsf{Id}_{\mathsf{Str}^{\kappa}_{C}}\begin{pmatrix}\mathsf{zw}^{\kappa}\,f\,xs\,ys,\\ \mathsf{zw}^{\kappa}\,f\,ys\,xs\end{pmatrix}\right)$$

Note the appearance of the generalised $\triangleright$, carrying a delayed substitution. Because the variable xs does not appear in $\overset{\kappa}{\triangleright}\mathsf{Str}^{\kappa}_{A}$ we may apply the weakening rule TmEq-Next-Weak to derive

$$\mathsf{tl}^{\kappa}\,ys : \overset{\kappa}{\triangleright}[xs \leftarrow \mathsf{tl}^{\kappa}\,xs]\,.\,\mathsf{Str}^{\kappa}_{A}$$

Hence we may use the derived applicative rule to have $\phi\textcircled{\kappa}\mathsf{tl}^{\kappa}\,xs\textcircled{\kappa}\,\mathsf{tl}^{\kappa}\,ys$ of type

$$\overset{\kappa}{\triangleright}\begin{bmatrix}xs \leftarrow \mathsf{tl}^{\kappa}\,xs\\ ys \leftarrow \mathsf{tl}^{\kappa}\,ys\end{bmatrix}.\mathsf{Id}_{\mathsf{Str}^{\kappa}_{C}}(\mathsf{zw}^{\kappa}\,f\,xs\,ys, \mathsf{zw}^{\kappa}\,f\,ys\,xs)$$

and which is definitionally equal to the type

$$\mathsf{Id}_{\overset{\kappa}{\triangleright}\mathsf{Str}^{\kappa}_{C}}\left(\begin{array}{l}\mathsf{next}^{\kappa}\begin{bmatrix}xs \leftarrow \mathsf{tl}^{\kappa}\,xs\\ ys \leftarrow \mathsf{tl}^{\kappa}\,ys\end{bmatrix}.\,\mathsf{zw}^{\kappa}\,f\,xs\,ys,\\ \mathsf{next}^{\kappa}\begin{bmatrix}xs \leftarrow \mathsf{tl}^{\kappa}\,xs\\ ys \leftarrow \mathsf{tl}^{\kappa}\,ys\end{bmatrix}.\,\mathsf{zw}^{\kappa}\,f\,ys\,xs\end{array}\right).$$

We also compute

$$\begin{aligned}\mathsf{tl}^{\kappa}\,(\mathsf{zw}^{\kappa}\,f\,xs\,ys) &\equiv \mathsf{next}^{\kappa}(\mathsf{zw}^{\kappa}\,f)\,\textcircled{\kappa}\,\mathsf{tl}^{\kappa}\,xs\,\textcircled{\kappa}\,\mathsf{tl}^{\kappa}\,ys\\ &\equiv \mathsf{next}^{\kappa}\begin{bmatrix}xs \leftarrow \mathsf{tl}^{\kappa}\,xs\\ ys \leftarrow \mathsf{tl}^{\kappa}\,ys\end{bmatrix}.(\mathsf{zw}^{\kappa}\,f\,xs\,ys)\end{aligned}$$

and

$$\mathsf{tl}^{\kappa}\,(\mathsf{zw}^{\kappa}\,f\,ys\,xs) \equiv \mathsf{next}^{\kappa}\begin{bmatrix}ys \leftarrow \mathsf{tl}^{\kappa}\,ys\\ zs \leftarrow \mathsf{tl}^{\kappa}\,xs\end{bmatrix}.(\mathsf{zw}^{\kappa}\,f\,ys\,xs).$$

Using the exchange rule TmEq-Next-Exch we have the equality

$$\begin{array}{c}\mathsf{next}^{\kappa}\begin{bmatrix}ys \leftarrow \mathsf{tl}^{\kappa}\,ys\\ xs \leftarrow \mathsf{tl}^{\kappa}\,xs\end{bmatrix}.(\mathsf{zw}^{\kappa}\,f\,xs\,ys)\\ \equiv\\ \mathsf{next}^{\kappa}\begin{bmatrix}xs \leftarrow \mathsf{tl}^{\kappa}\,xs\\ ys \leftarrow \mathsf{tl}^{\kappa}\,ys\end{bmatrix}.(\mathsf{zw}^{\kappa}\,f\,xs\,ys)\end{array}.$$

Putting it all together we have shown that the term $\phi\,\textcircled{\kappa}\,\mathsf{tl}^{\kappa}\,xs\,\textcircled{\kappa}\,\mathsf{tl}^{\kappa}\,ys$ has type

$$\mathsf{Id}_{\overset{\kappa}{\triangleright}\mathsf{Str}^{\kappa}_{B}}(\mathsf{tl}^{\kappa}\,(\mathsf{zw}^{\kappa}\,f\,xs\,ys), \mathsf{tl}^{\kappa}\,(\mathsf{zw}^{\kappa}\,f\,ys\,xs))$$

which means that the term

$$\begin{aligned}&\mathsf{fix}^{\kappa}\,\phi.\lambda\,(xs, ys : \mathsf{Str}^{\kappa}_{A})\,.\\ &\mathsf{p}\eta\;\,(c\,(\mathsf{hd}^{\kappa}\,xs)\,(\mathsf{hd}^{\kappa}\,ys))\;\,(\phi\,\textcircled{\kappa}\,\mathsf{tl}^{\kappa}\,xs\,\textcircled{\kappa}\,\mathsf{tl}^{\kappa}\,ys)\end{aligned}$$

has type

$$\Pi(xs, ys : \mathsf{Str}^{\kappa}_{A}).\mathsf{Id}_{\mathsf{Str}^{\kappa}_{B}}(\mathsf{zw}^{\kappa}\,f\,xs\,ys, \mathsf{zw}^{\kappa}\,f\,ys\,xs).$$

Notice that the resulting proof term could not be simpler than it is. In particular, we do not have to write delayed substitutions in terms, but only in the intermediate types.

3.2 An example with covectors

The next example is more sophisticated, as it will involve programming and proving with a data type that, unlike streams, is dependently typed. In particular, we will see that the generalised later, carrying a delayed substitution, is necessary to type even the most elementary programs.

Covectors are to colists (potentially infinite lists) as vectors are to lists. To define guarded covectors we first need guarded conatural numbers. This is the type satisfying

$$\mathsf{Co}\mathbb{N}^{\kappa} \simeq \mathbf{1} + \overset{\kappa}{\triangleright}\mathsf{Co}\mathbb{N}^{\kappa}\,.$$

where binary sums are encoded in the type theory in a standard way. The definition in gDTT is $\mathsf{Co}\mathbb{N}^{\kappa} \triangleq \mathsf{El}\left(\mathsf{fix}^{\kappa}\,\phi.(\widehat{\mathbf{1}} \mathbin{\widehat{+}} \widehat{\triangleright}^{\kappa}\phi)\right)$.

Using $\mathsf{Co}\mathbb{N}^{\kappa}$ we define the type of covectors of type A, written $\mathsf{CoVec}^{g_{\kappa}}_{A}$, as a $\mathsf{Co}\mathbb{N}^{\kappa}$-indexed type satisfying

$$\begin{aligned}\mathsf{CoVec}^{g_{\kappa}}_{A}\,(\mathsf{inl}\,\langle\rangle) &\simeq \mathbf{1}\\ \mathsf{CoVec}^{g_{\kappa}}_{A}\,(\mathsf{inr}(\mathsf{next}^{\kappa}\,m)) &\simeq A \times \overset{\kappa}{\triangleright}(\mathsf{CoVec}^{g_{\kappa}}_{A}\,m)\end{aligned}$$

In gDTT we first define $\widehat{\mathsf{CoVec}^{g_\kappa}_A}$

$$\begin{aligned}\widehat{\mathsf{CoVec}^{g_\kappa}_A} \triangleq \mathsf{fix}^\kappa\,\phi.\lambda(n : \mathsf{Co}\mathbb{N}^\kappa).\,&\mathsf{case}\,n\,\mathsf{of}\\ &\mathsf{inl}\,u \Rightarrow \widehat{\mathbf{1}}\\ &\mathsf{inr}\,m \Rightarrow A \mathbin{\widehat{\times}} \widehat{\triangleright}^\kappa(\phi \textcircled{\kappa}\, m).\end{aligned}$$

and then $\mathsf{CoVec}^{g_\kappa}_A\, n \triangleq \mathsf{El}(\widehat{\mathsf{CoVec}^{g_\kappa}_A}\, n)$. In the examples we will not distinguish between $\mathsf{CoVec}^{g_\kappa}_A$ and $\widehat{\mathsf{CoVec}^{g_\kappa}_A}$. In the above ϕ has type $\overset{\kappa}{\triangleright}(\mathsf{Co}\mathbb{N}^\kappa \to \mathcal{U}_{\Delta,\kappa})$ and inside the branches, u has type $\mathbf{1}$ and m has type $\overset{\kappa}{\triangleright}\,\mathsf{Co}\mathbb{N}^\kappa$, which is evident from the definition of $\mathsf{Co}\mathbb{N}^\kappa$. As an example of covectors, we define ones of type $\Pi(n : \mathsf{Co}\mathbb{N}^\kappa).\,\mathsf{CoVec}^{g_\kappa}_{\mathbb{N}}\, n$ which produces a covector of any length consisting only of ones:

$$\begin{aligned}\mathsf{ones} \triangleq \mathsf{fix}^\kappa\,\phi.\lambda(n : \mathsf{Co}\mathbb{N}^\kappa).\,&\mathsf{case}\,n\,\mathsf{of}\\ &\mathsf{inl}\,u \Rightarrow \mathsf{inl}\,\langle\rangle\\ &\mathsf{inr}\,m \Rightarrow \langle 1, \phi \textcircled{\kappa}\, m\rangle\,.\end{aligned}$$

When checking the type of this program, we need the generalised later. The type of the recursive call is $\overset{\kappa}{\triangleright}(\Pi(n : \mathsf{Co}\mathbb{N}^\kappa).\,\mathsf{CoVec}^{g_\kappa}_{\mathbb{N}}\, n)$, the type of m is $\overset{\kappa}{\triangleright}\,\mathsf{Co}\mathbb{N}^\kappa$, and therefore the type of the subterm $\phi \textcircled{\kappa}\, m$ must be

$$\overset{\kappa}{\triangleright}[n \leftarrow m]\,.\Pi(n : \mathsf{Co}\mathbb{N}^\kappa).\,\mathsf{CoVec}^{g_\kappa}_{\mathbb{N}}\, n.x$$

We now aim to define the function map on covectors and show that it preserves composition. Given two types A and B the map function has type

$$\mathsf{map} : (A \to B) \to \Pi(n : \mathsf{Co}\mathbb{N}^\kappa).\,\mathsf{CoVec}^{g_\kappa}_A\, n \to \mathsf{CoVec}^{g_\kappa}_B\, n.$$

and is defined by guarded recursion as

$$\begin{aligned}&\mathsf{map}\,f \triangleq \mathsf{fix}^\kappa\,\phi.\lambda(n : \mathsf{Co}\mathbb{N}^\kappa).\\ &\quad\mathsf{case}\,n\,\mathsf{of}\\ &\quad\mathsf{inl}\,u \Rightarrow \lambda(x : 1).x\\ &\quad\mathsf{inr}\,m \Rightarrow \begin{array}{l}\lambda\left(p : A \times \overset{\kappa}{\triangleright}[n \leftarrow m]\,.(\mathsf{CoVec}^{g_\kappa}_A\, n)\right).\\ \langle f\,(\pi_1 p)\,, \phi \textcircled{\kappa}\, m \textcircled{\kappa}\,(\pi_2 p)\rangle\end{array}\end{aligned}$$

Let us see why the definition has the correct type. First, the types of subterms are

$$\begin{aligned}&\phi : \overset{\kappa}{\triangleright}(\Pi(n : \mathsf{Co}\mathbb{N}^\kappa).\,\mathsf{CoVec}^{g_\kappa}_A\, n \to \mathsf{CoVec}^{g_\kappa}_B\, n)\\ &u : \mathbf{1}\\ &m : \overset{\kappa}{\triangleright}\,\mathsf{Co}\mathbb{N}^\kappa\end{aligned}$$

Let $C = \mathsf{CoVec}^{g_\kappa}_A\, n \to \mathsf{CoVec}^{g_\kappa}_B\, n$, and write $C(t)$ for $C[t/n]$. By the definition of $\mathsf{CoVec}^{g_\kappa}_A$ and $\mathsf{CoVec}^{g_\kappa}_B$ we have $C(\mathsf{inl}\,u) \simeq \mathbf{1} \to \mathbf{1}$, and so $\lambda(x : \mathbf{1}).x$ has type $C(\mathsf{inl}\,u)$.

By the definition of $\mathsf{CoVec}^{g_\kappa}_A$ we have

$$\begin{aligned}\mathsf{CoVec}^{g_\kappa}_A(\mathsf{inr}\,m) &\simeq A \times \mathsf{El}\,(\widehat{\triangleright}^\kappa(\mathsf{next}^\kappa(\mathsf{CoVec}^{g_\kappa}_A) \textcircled{\kappa}\, m))\\ &\simeq A \times \overset{\kappa}{\triangleright}[n \leftarrow m]\,.\,(\mathsf{CoVec}^{g_\kappa}_A\, n)\end{aligned}$$

and analogously for $\mathsf{CoVec}^{g_\kappa}_B(\mathsf{inr}\,m)$. Hence the type $C(\mathsf{inr}\,m)$ is convertible to

$$\left(A \times \overset{\kappa}{\triangleright}[n \leftarrow m]\,.\,(\mathsf{CoVec}^{g_\kappa}_A\, n)\right) \to \left(B \times \overset{\kappa}{\triangleright}[n \leftarrow m]\,.\,(\mathsf{CoVec}^{g_\kappa}_B\, n)\right).$$

Further, using the derived applicative rule we have

$$\phi \textcircled{\kappa}\, m : \overset{\kappa}{\triangleright}[n \leftarrow m]\,.C(n)$$

and because $\pi_2 p$ in the second branch has type

$$\overset{\kappa}{\triangleright}[n \leftarrow m]\,.(\mathsf{CoVec}^{g_\kappa}_A\, n)$$

we may use the (simple) applicative rule again to get

$$\phi \textcircled{\kappa}\, m \textcircled{\kappa}\,(\pi_2 p) : \overset{\kappa}{\triangleright}[n \leftarrow m]\,.(\mathsf{CoVec}^{g_\kappa}_B\, n)$$

which allows us to type

$$\lambda\left(p : A \times \overset{\kappa}{\triangleright}[n \leftarrow m]\,.(\mathsf{CoVec}^{g_\kappa}_A\, n)\right).\,\langle f\,(\pi_1 p)\,, \phi \textcircled{\kappa}\, m \textcircled{\kappa}\,\pi_2(p)\rangle$$

with type $C(\mathsf{inr}\,m)$. Notice that we have made essential use of the more general applicative rule to apply $\phi \textcircled{\kappa}\, m$ to $\pi_2 p$. Using the strong (dependent) elimination rule for binary sums we can type the whole case construct with type $C(n)$, which is what we need to give map the desired type.

Now we will show that map so defined satisfies a basic property, namely that it preserves composition in the sense that the type (in the context where we have types A, B and C)

$$\begin{aligned}&\Pi(f : A \to B)(g : B \to C)(n : \mathsf{Co}\mathbb{N}^\kappa)(xs : \mathsf{CoVec}^{g_\kappa}_A\, n).\\ &\quad\mathsf{Id}_{\mathsf{CoVec}^{g_\kappa}_C\, n}(\mathsf{map}\,g\,n(\mathsf{map}\,f\,n\,xs), \mathsf{map}(g \circ f)\,n\,xs)\end{aligned} \tag{5}$$

is inhabited. The proof is, of course, by Löb induction.

First we record some definitional equalities which follow directly by unfolding the definitions

$$\begin{aligned}\mathsf{map}\,f\,(\mathsf{inl}\,u)\,x &\equiv x\\ \mathsf{map}\,f\,(\mathsf{inr}\,m)\,xs &\equiv \langle f\,(\pi_1 xs)\,, \mathsf{next}^\kappa(\mathsf{map}\,f) \textcircled{\kappa}\, m \textcircled{\kappa}\,\pi_2(xs)\rangle\\ &\equiv \langle f(\pi_1 xs), \mathsf{next}^\kappa\left[\begin{array}{l}n \leftarrow m\\ ys \leftarrow \pi_2 xs\end{array}\right].(\mathsf{map}\,f\,n\,ys)\rangle\end{aligned}$$

and so iterating these two equalities we get

$$\begin{aligned}\mathsf{map}\,g\,(\mathsf{inl}\,u)\,(\mathsf{map}\,f(\mathsf{inl}\,u)\,x) &\equiv x\\ \mathsf{map}\,g\,(\mathsf{inr}\,m)\,(\mathsf{map}\,f(\mathsf{inr}\,m)\,xs) &\equiv \langle g(f(\pi_1 xs)), s\rangle\end{aligned}$$

where s is the term

$$\mathsf{next}^\kappa\left[\begin{array}{l}n \leftarrow m\\ zs \leftarrow \mathsf{next}^\kappa\left[\begin{array}{l}n \leftarrow m\\ ys \leftarrow \pi_2 xs\end{array}\right].(\mathsf{map}\,f\,n\,ys)\end{array}\right].(\mathsf{map}\,g\,n\,zs)$$

which is convertible, by the rule TmEq-Force, to the term

$$\mathsf{next}^\kappa\left[\begin{array}{l}n \leftarrow m\\ ys \leftarrow \pi_2 xs\end{array}\right].(\mathsf{map}\,g\,n\,(\mathsf{map}\,f\,n\,ys)).$$

Similarly we have

$$\mathsf{map}(g \circ f)\,(\mathsf{inl}\,u)\,x \equiv x$$

and $\mathsf{map}(g \circ f)\,(\mathsf{inr}\,m)\,xs$ convertible to

$$\left\langle g(f(\pi_1 xs)), \mathsf{next}^\kappa\left[\begin{array}{l}n \leftarrow m\\ ys \leftarrow \pi_2 xs\end{array}\right].(\mathsf{map}(g \circ f)\,n\,ys)\right\rangle.$$

Now let us get back to proving property (5). Take $f : A \to B$, $g : B \to C$ and assume

$$\begin{aligned}&\phi : \overset{\kappa}{\triangleright}\Pi(n : \mathsf{Co}\mathbb{N}^\kappa)(xs : \mathsf{CoVec}^{g_\kappa}_A\, n).\\ &\quad\mathsf{Id}_{\mathsf{CoVec}^{g_\kappa}_C\, n}(\mathsf{map}\,g\,n(\mathsf{map}\,f\,n\,xs), \mathsf{map}(g \circ f)\,n\,xs)\end{aligned}$$

We take $n : \mathsf{Co}\mathbb{N}^\kappa$ and write

$$\begin{aligned}&P(n) = \Pi(xs : \mathsf{CoVec}^{g_\kappa}_A\, n).\\ &\quad\mathsf{Id}_{\mathsf{CoVec}^{g_\kappa}_C\, n}(\mathsf{map}\,g\,n(\mathsf{map}\,f\,n\,xs), \mathsf{map}(g \circ f)\,n\,xs).\end{aligned}$$

Then similarly as in the definition of map and the definitional equalities for map above we compute

$$P(\mathsf{inl}\,u) \simeq \Pi(xs : 1).\mathsf{Id}_1(xs, xs)$$

and so we have $\lambda(xs : 1).\mathsf{r}_1\, xs$ of type $P(\mathsf{inl}\,u)$.

The other branch (when $n = \mathsf{inr}\,m$) is of course a bit more complicated. As before we have

$$\mathsf{CoVec}^{g_\kappa}_A(\mathsf{inr}\,m) \simeq A \times \overset{\kappa}{\triangleright}[n \leftarrow m]\,.\,\mathsf{CoVec}^{g_\kappa}_A\, n \tag{6}$$

So take xs of type $\mathsf{CoVec}^{g_\kappa}_{A}(\mathsf{inr}\, m)$. We need to construct a term of type $\mathsf{Id}_{\mathsf{CoVec}^{g_\kappa}_{C}\, n}(\mathsf{map}\, g\, n(\mathsf{map}\, f\, n\, xs), \mathsf{map}(g \circ f)\, n\, xs)$.

First we type

$$\mathsf{r}_C\, g(f(\pi_1 xs)) : \mathsf{Id}_C(g(f(\pi_1 xs)), g(f(\pi_1 xs))).$$

Then because m is of type $\overset{\kappa}{\triangleright} \mathsf{Co}\mathbb{N}^\kappa$ we can use the induction hypothesis ϕ to get $\phi \textcircled{\kappa}\, m$ of type

$$\overset{\kappa}{\triangleright} [n \leftarrow m] . \Pi(xs : \mathsf{CoVec}^{g_\kappa}_{A}\, n). \\ \mathsf{Id}_{\mathsf{CoVec}^{g_\kappa}_{C}\, n}(\mathsf{map}\, g\, n(\mathsf{map}\, f\, n\, xs), \mathsf{map}(g \circ f)\, n\, xs).$$

Using (6) we have $\pi_2 xs$ of type $\overset{\kappa}{\triangleright} [n \leftarrow m] . \mathsf{CoVec}^{g_\kappa}_{A}\, n$ and so we can use the applicative rule again to give $\phi \textcircled{\kappa}\, m \textcircled{\kappa}\, \pi_2 xs$ the type

$$\overset{\kappa}{\triangleright} \begin{bmatrix} n \leftarrow m \\ xs \leftarrow \pi_2 xs \end{bmatrix} . \mathsf{Id}_{\mathsf{CoVec}^{g_\kappa}_{C}\, n} \begin{pmatrix} \mathsf{map}\, g\, n(\mathsf{map}\, f\, n\, xs), \\ \mathsf{map}(g \circ f)\, n\, xs \end{pmatrix}$$

which by the rule $\mathsf{TyEq}\text{-}\triangleright$ is the same as

$$\mathsf{Id}_D \begin{pmatrix} \mathsf{next}^\kappa \begin{bmatrix} n \leftarrow m \\ xs \leftarrow \pi_2 xs \end{bmatrix} . (\mathsf{map}\, g\, n(\mathsf{map}\, f\, n\, xs)), \\ \mathsf{next}^\kappa \begin{bmatrix} n \leftarrow m \\ xs \leftarrow \pi_2 xs \end{bmatrix} . (\mathsf{map}(g \circ f)\, n\, xs) \end{pmatrix}$$

where D is the type $\overset{\kappa}{\triangleright} [n \leftarrow m] . \mathsf{CoVec}^{g_\kappa}_{C}\, n$. Thus we can give to the term

$$\lambda(xs : \mathsf{CoVec}^{g_\kappa}_{A}(\mathsf{inr}\, m)). \\ \mathsf{p}\eta\, (\mathsf{r}_C\, g(f(\pi_1 xs)))\, (\phi \textcircled{\kappa}\, m \textcircled{\kappa}\, \pi_2 xs)$$

the type $P(\mathsf{inr}\, m)$. Using the dependent elimination rule for binary sums we get the final proof of property (5) as the term

$$\begin{aligned} &\lambda(f : A \to B)(g : B \to C). \mathsf{fix}^\kappa\, \phi. \lambda(n : \mathsf{Co}\mathbb{N}^\kappa). \\ &\mathsf{case}\, n\, \mathsf{of} \\ &\mathsf{inl}\, u \Rightarrow \lambda(xs : 1). \mathsf{r}_1\, xs \\ &\mathsf{inr}\, m \Rightarrow \lambda(xs : \mathsf{CoVec}^{g_\kappa}_{A}(\mathsf{inr}\, m)). \\ &\qquad \mathsf{p}\eta\, (\mathsf{r}_C\, g(f(\pi_1 xs)))\, (\phi \textcircled{\kappa}\, m \textcircled{\kappa}\, \pi_2 xs) \end{aligned}$$

which is as simple as could be expected.

3.3 Lifting predicates to streams

Let $P : A \to \mathcal{U}_\Delta$ be a predicate on type A and κ a clock variable not in Δ. We can define a lifting of this predicate to a predicate P^κ on streams of elements of type A. The idea is that $P^\kappa xs$ will hold precisely when P holds for all elements of the stream. However we do not have access to all the element of the stream at the same time. As such we will have $P^\kappa xs$ if P holds for the first element of the stream xs now, and P holds for the second element of the stream xs one time step later, and so on. The precise definition uses guarded recursion:

$$\begin{aligned} &P^\kappa : \mathsf{Str}^\kappa_A \to \mathcal{U}_{\Delta,\kappa} \\ &P^\kappa \triangleq \mathsf{fix}^\kappa\, \phi. \lambda\, (xs : \mathsf{Str}^\kappa_A) . P\, (\mathsf{hd}^\kappa\, xs)\, \widehat{\times}\, \widehat{\triangleright}^\kappa\, (\phi \textcircled{\kappa}\, \mathsf{tl}^\kappa\, xs) . \end{aligned}$$

In the above term the subterm ϕ has type $\overset{\kappa}{\triangleright} (\mathsf{Str}^\kappa_A \to \mathcal{U}_{\Delta,\kappa})$ and so because $\mathsf{tl}^\kappa\, xs$ has type $\overset{\kappa}{\triangleright} \mathsf{Str}^\kappa_A$ we may form $\phi \textcircled{\kappa}\, \mathsf{tl}^\kappa\, xs$ of type $\overset{\kappa}{\triangleright} \mathcal{U}_{\Delta,\kappa}$ and so finally $\widehat{\triangleright}^\kappa (\phi \textcircled{\kappa}\, \mathsf{tl}^\kappa\, xs)$ has type $\mathcal{U}_{\Delta,\kappa}$ as needed.

To see that this makes sense, we have for a stream $xs : \mathsf{Str}^\kappa_A$

$$\mathsf{El}\, (P^\kappa\, xs) \simeq \mathsf{El}\, (P\, (\mathsf{hd}^\kappa\, xs)) \times \mathsf{El}\, (\widehat{\triangleright}^\kappa\, (\mathsf{next}^\kappa\, P^\kappa \textcircled{\kappa}\, \mathsf{tl}^\kappa\, xs)) .$$

Using delayed substitution rules we have

$$\mathsf{next}^\kappa\, P^\kappa \textcircled{\kappa}\, \mathsf{tl}^\kappa\, xs \equiv \mathsf{next}^\kappa\, [xs \leftarrow \mathsf{tl}^\kappa\, xs] . (P^\kappa\, xs)$$

which gives rise to the type equality

$$\begin{aligned} &\mathsf{El}(\widehat{\triangleright}^\kappa\, \mathsf{next}^\kappa\, P^\kappa \textcircled{\kappa}\, \mathsf{tl}^\kappa\, xs) \\ &\simeq \mathsf{El}\, (\widehat{\triangleright}^\kappa\, \mathsf{next}^\kappa\, [xs \leftarrow \mathsf{tl}^\kappa\, xs] . (P^\kappa\, xs)) . \end{aligned}$$

Finally, the type equality rule $\textsc{TyEq-El-}\triangleright$ gives us

$$\begin{aligned} &\mathsf{El}\, (\widehat{\triangleright}^\kappa\, \mathsf{next}^\kappa\, [xs \leftarrow \mathsf{tl}^\kappa\, xs] . (P^\kappa\, xs)) \\ &\simeq \overset{\kappa}{\triangleright} [xs \leftarrow \mathsf{tl}^\kappa\, xs] . \mathsf{El}(P^\kappa\, xs). \end{aligned}$$

All of these together then give us the type equality

$$\mathsf{El}\, (P^\kappa\, xs) \simeq \mathsf{El}(P\, (\mathsf{hd}^\kappa\, xs)) \times \overset{\kappa}{\triangleright} [xs \leftarrow \mathsf{tl}^\kappa\, xs] . \mathsf{El}(P^\kappa\, xs).$$

And so if $xs = \mathsf{cons}^\kappa\, x\, (\mathsf{next}^\kappa\, ys)$ we can further simplify, using rule $\textsc{TyEq-Force}$, to get

$$\begin{aligned} \overset{\kappa}{\triangleright} [xs \leftarrow \mathsf{next}^\kappa\, ys] . \mathsf{El}(P^\kappa\, xs) &\simeq \overset{\kappa}{\triangleright} (\mathsf{El}(P^\kappa\, xs)[ys/xs]) \\ &\simeq \overset{\kappa}{\triangleright} \mathsf{El}(P^\kappa\, ys) \end{aligned}$$

which then gives $\mathsf{El}(P^\kappa xs) \simeq \mathsf{El}\, (P\, x) \times \overset{\kappa}{\triangleright} \mathsf{El}\, (P^\kappa\, ys)$ which is in accordance with the motivation given above.

Because P^κ is defined by guarded recursion, we prove its properties by Löb induction. In particular, we may prove that if P holds on A then P^κ holds on Str^κ_A, i.e., that the type

$$(\Pi(x : A). \mathsf{El}\, (P\, x)) \to (\Pi(xs : \mathsf{Str}^\kappa_A). \mathsf{El}\, (P^\kappa\, xs))$$

is inhabited (in a context where we have a type A and a predicate P). Take $p : \Pi(x : A). \mathsf{El}\, (P\, x)$, and since we are proving by Löb induction we assume the induction hypothesis later

$$\phi : \overset{\kappa}{\triangleright} (\Pi(xs : \mathsf{Str}^\kappa_A). \mathsf{El}\, (P^\kappa\, xs)) .$$

Let $xs : \mathsf{Str}^\kappa_A$ be a stream. By definition of P^κ we have the type equality

$$\mathsf{El}(P^\kappa xs) \simeq \mathsf{El}\, (P\, \mathsf{hd}^\kappa\, xs) \times \overset{\kappa}{\triangleright} [xs \leftarrow \mathsf{tl}^\kappa\, xs] . \mathsf{El}\, (P^\kappa\, xs)$$

Applying p to $\mathsf{hd}^\kappa\, xs$ gives us the first component

$$p(\mathsf{hd}^\kappa\, xs) : \mathsf{El}\, (P\, (\mathsf{hd}^\kappa\, xs))$$

and applying the induction hypothesis ϕ we have

$$\phi \textcircled{\kappa}\, \mathsf{tl}^\kappa\, xs : \overset{\kappa}{\triangleright} [xs \leftarrow \mathsf{tl}^\kappa\, xs] . \mathsf{El}(P^\kappa\, xs)$$

Thus combining this with the previous term we have the proof of the lifting property as the term

$$\begin{aligned} &\lambda\, (p : \Pi(x : A). \mathsf{El}\, (P\, x)) . \\ &\quad \mathsf{fix}^\kappa\, \phi. \lambda\, (xs : \mathsf{Str}^\kappa_A)\, \langle p\, (\mathsf{hd}^\kappa\, xs)\, , \phi \textcircled{\kappa}\, \mathsf{tl}^\kappa\, xs \rangle . \end{aligned}$$

4. Coinductive types

As discussed in the introduction, guarded recursive types on their own disallow productive but acausal function definitions. To capture such functions we need to be able to remove $\overset{\kappa}{\triangleright}$. However such eliminations must be controlled to avoid trivialising $\overset{\kappa}{\triangleright}$. Consider an unrestricted elimination term $\mathsf{elim} : \overset{\kappa}{\triangleright} A \to A$ for every type A, for instance when $A = \mathsf{Str}^\kappa_{\mathbb{N}}$. We could then type the term

$$\mathsf{bad} = \mathsf{fix}^\kappa\, xs.\, \mathsf{elim}\, xs$$

of type $\mathsf{Str}^g_\kappa\, \mathbb{N}$. However this would have to be a solution to the stream equation $xs = xs$, which is obviously unproductive.

However we may eliminate $\overset{\kappa}{\triangleright}$ provided that the term does not depend on the clock κ, i.e., the term is typeable in a context where κ does not appear. Intuitively, such contexts have no temporal properties along the κ dimension, so we may progress the computation without violating guardedness requirements. Figure 3 extends the system of Figure 2 to allow the removal of clocks in such a setting, by introducing *clock quantifiers* $\forall\kappa$ [4, 8, 22]. This is a binding construct with associated term constructor $\Lambda\kappa$, which also binds κ. The elimination term is *clock application*. Application of the term t of type $\forall\kappa.A$ to a clock κ is written as $t[\kappa]$. One may think of $\forall\kappa.A$ as analogous to the type $\forall\alpha.A$ in polymorphic lambda calculus; indeed the basic rules are precisely the same, but we have an additional construct $\mathsf{prev}\,\kappa.t$, called 'previous', to allow removal of the later modality $\overset{\kappa}{\triangleright}$.

Typing this new construct $\mathsf{prev}\,\kappa.t$ is somewhat complicated, as it requires 'advancing' a delayed substitution, which turns it into a context morphism (an actual substitution). The judgement $\rho :_\Delta \Gamma \to \Gamma'$ expresses that ρ is a context morphisms from context $\Gamma \vdash_\Delta$ to the context $\Gamma' \vdash_\Delta$. We use the notation $\rho[x := t]$ for extending the context morphism by mapping the variable x to the term t. We illustrate this with two concrete examples.

First, we can indeed remove later under a clock quantier:

$$\begin{aligned}&\mathsf{force} : \forall\kappa.\overset{\kappa}{\triangleright}A \to \forall\kappa.A\\ &\mathsf{force} \triangleq \lambda x.\,\mathsf{prev}\,\kappa.x[\kappa]\,.\end{aligned}$$

This is type correct because advancing the empty delayed substitution in $\overset{\kappa}{\triangleright}$ turns it into the identity substitution ι, and $A\iota \simeq A$. The β and η rules ensure that force is the inverse to the canonical term $\lambda x.\Lambda\kappa.\,\mathsf{next}^\kappa\, x[\kappa]$ of type $\forall\kappa.A \to \forall\kappa.\overset{\kappa}{\triangleright}A$.

Second, an example with a non-empty delayed substitution is the term $\mathsf{prev}\,\kappa.\,\mathsf{next}^\kappa\,\lambda n.\,\mathsf{succ}\,n \circledast_\kappa \mathsf{next}^\kappa\,0$ of type $\forall\kappa.\mathbb{N}$. Recall that $\circledast_\kappa$ is syntactic sugar and so more precisely the term is

$$\mathsf{prev}\,\kappa.\,\mathsf{next}^\kappa \left[\begin{array}{l} f \leftarrow \mathsf{next}^\kappa\,\lambda n.\,\mathsf{succ}\,n \\ x \leftarrow \mathsf{next}^\kappa\,0 \end{array}\right].f\,x. \tag{7}$$

Advancing the delayed substitution turns it into the substitution mapping the variable f to the term $(\mathsf{prev}\,\kappa.\,\mathsf{next}^\kappa\,\lambda n.\,\mathsf{succ}\,n)[\kappa]$ and the variable x to the term $(\mathsf{prev}\,\kappa.\,\mathsf{next}^\kappa\,0)[\kappa]$. Using the β rule for prev, then the β rule for $\forall\kappa$, this simplifies to the substitution mapping f to $\lambda n.\,\mathsf{succ}\,n$ and x to 0. With this we have that the term (7) is equal to $\Lambda\kappa.\,((\lambda n.\,\mathsf{succ}\,n)\,0)$ which is in turn equal to $\Lambda\kappa.1$.

An important property of the term $\mathsf{prev}\,\kappa.t$ is that κ is *bound* in t; hence $\mathsf{prev}\,\kappa.t$ has type $\forall\kappa.A$ instead of just A. This ensures that substitution of terms in types and terms is well-behaved and we do not need explicit substitutions as needed, for example, in Clouston et al. [10] where the unary type-former $\Box$ was used in place of clocks. This binding structure makes, for instance, the introduction rule TY-Λ closed under substitution in Γ without using explicit substitions on terms or types.

We have one final term equality rule TMEQ-$\forall$-FRESH. This rule states that if t has type $\forall\kappa.A$ and the clock κ does not appear in the *type* A, then it does not matter to which clock t is applied, as the resulting term will be the same.

Finally we have the construct $\widehat{\forall}$ and the rule TY-$\forall$-CODE which witness that the universes are closed under $\forall\kappa$.

To summarise, the novel raw types and terms of gDTT, extending those of Section 2, are

$$\begin{aligned}A, B &::= \cdots \mid \forall\kappa.A\\ t, u &::= \cdots \mid \Lambda\kappa.t \mid t[\kappa] \mid \widehat{\forall}\,t \mid \mathsf{prev}\,\kappa.t\end{aligned}$$

Finally, we have the equality rule TYEQ-$\forall$-ID analogous to the rule TYEQ-$\triangleright$. Using it we can also derive the "commuting conversion" term equality

$$\frac{\Gamma \vdash_\Delta \qquad \Gamma \vdash_{\Delta,\kappa} A \text{ type} \qquad \Gamma \vdash_\Delta t : \forall\kappa.A}{\Gamma \vdash_\Delta \Lambda\kappa.\mathsf{r}_A\, t[\kappa] \equiv \mathsf{r}_{\forall\kappa.A}\, t : \mathsf{Id}_{\forall\kappa.A}(t,t)}$$

using the fact that all proofs of equality are definitionally equal in an extensional type theory. Note that as in Section 2.2 there is a canonical term of type $\mathsf{Id}_{\forall\kappa.A}(t,s) \to \forall\kappa.\mathsf{Id}_A(t[\kappa]\,, s[\kappa])$ but no term in the reverse direction. Since TYEQ-$\forall$-ID is validated by the model we choose to add it instead of having explicit terms.

4.1 Derivable type isomorphisms

To work with coinductive types we shall need type isomorphisms commuting $\forall\kappa$ over other type formers. By a type isomorphism $A \cong B$ we mean two well-typed terms f and g of types $f : A \to B$ and $g : B \to A$ such that $f(g\,x) \equiv x$ and $g(f\,x) \equiv x$. We record these now.

The following type isomorphisms follow by using β and η laws for the constructs involved and the rule TMEQ-$\forall$-FRESH.

- If $\kappa \notin A$ then $\forall\kappa.A \cong A$.
- If $\kappa \notin A$ then $\forall\kappa.\Pi(x : A).B \cong \Pi(x : A).\forall\kappa.B$.
- $\forall\kappa.\Sigma\,(x : A)\,B \cong \Sigma\,(y : \forall\kappa.A)\,(\forall\kappa.B\,[y[\kappa]/\,x])$.
- $\forall\kappa.A \cong \forall\kappa.\overset{\kappa}{\triangleright}A$.

There is an additional weak type isomorphism witnessing that $\forall\kappa$ commutes with binary sums. *Weak* in the sense that it requires equality reflection to show that the two functions are inverses to each other up to definitional equality. The construction is in the appendix provided as supplementary material, here we just record its properties.

There is a canonical term of type $\forall\kappa.A + \forall\kappa.B \to \forall\kappa.(A+B)$ using just ordinary elimination of coproducts. Using the fact that we encode binary coproducts using Σ-types and universes we can additionally define a term com^+ of type

$$\forall\kappa.(A+B) \to \forall\kappa.A + \forall\kappa.B$$

which is a weak inverse to the canonical term. The term com^+ satisfies two definitional equalities

$$\begin{aligned}\mathsf{com}^+\,(\Lambda\kappa.\,\mathsf{inl}\,t) &\equiv \mathsf{inl}\,\Lambda\kappa.t\\ \mathsf{com}^+\,(\Lambda\kappa.\,\mathsf{inr}\,t) &\equiv \mathsf{inr}\,\Lambda\kappa.t\end{aligned} \tag{8}$$

which are used in the examples below.

5. Example programs with coinductive types

Let A be some small type in clock context Δ and κ, a fresh clock variable. Let $\mathsf{Str}_A = \forall\kappa.\,\mathsf{Str}^\kappa_A$. We can define head, tail and cons functions

$$\begin{aligned}&\mathsf{hd} : \mathsf{Str}_A \to A && \mathsf{tl} : \mathsf{Str}_A \to \mathsf{Str}_A\\ &\mathsf{hd} \triangleq \lambda xs.\,\mathsf{hd}^{\kappa_0}\,(xs[\kappa_0]) && \mathsf{tl} \triangleq \lambda xs.\,\mathsf{prev}\,\kappa.\,\mathsf{tl}^\kappa\,(xs[\kappa])\end{aligned}$$

$$\begin{aligned}&\mathsf{cons} : A \to \mathsf{Str}_A \to \mathsf{Str}_A\\ &\mathsf{cons} \triangleq \lambda x.\lambda xs.\Lambda\kappa.\,\mathsf{cons}^\kappa\,x\,(\mathsf{next}^\kappa\,(xs[\kappa]))\,.\end{aligned}$$

With these we can define the *acausal* 'every other' function eo^κ that removes every second element of the input stream. This is acausal because the second element of the output stream is the third element of the input. Therefore to type the function we need to have the input stream always available, necessitating the use clock quantification. The function eo^κ is

$$\begin{aligned}&\mathsf{eo}^\kappa : \mathsf{Str}_A \to \mathsf{Str}^\kappa_A\\ &\mathsf{eo}^\kappa \triangleq \mathsf{fix}^\kappa\,\phi.\lambda\,(xs : \mathsf{Str}_A)\,.\\ &\qquad\qquad \mathsf{cons}^\kappa(\mathsf{hd}\,xs)\,(\phi \circledast_\kappa \mathsf{next}^\kappa\,((\mathsf{tl}\,(\mathsf{tl}\,xs))))\,.\end{aligned}$$

i.e., we return the head immediately and then recursively call the function on the stream with the first two elements removed. Note that the result is a *guarded* stream, but we can easily strengthen it and define eo of type $\mathsf{Str}_A \to \mathsf{Str}_A$ as $\mathsf{eo} \triangleq \lambda xs.\Lambda\kappa.\,\mathsf{eo}^\kappa\,xs$.

A more interesting type is the type of covectors, which is a refinement of the guarded type of covectors defined in Section 3.2.

Type formation:

$$\frac{\Gamma \vdash_\Delta \qquad \Gamma \vdash_{\Delta,\kappa} A \ \mathsf{type}}{\Gamma \vdash_\Delta \forall\kappa.A \ \mathsf{type}} \ \textsc{Tf-}\forall$$

Typing rules:

$$\frac{\Gamma \vdash_\Delta \qquad \Gamma \vdash_{\Delta,\kappa} t : A}{\Gamma \vdash_\Delta \Lambda\kappa.t : \forall\kappa.A} \ \textsc{Ty-}\Lambda \qquad \frac{\vdash_\Delta \kappa' \qquad \Gamma \vdash_\Delta t : \forall\kappa.A}{\Gamma \vdash_\Delta t\left[\kappa'\right] : A[\kappa'/\kappa]} \ \textsc{Ty-app} \qquad \frac{\Delta' \subseteq \Delta \qquad \Gamma \vdash_\Delta t : \forall\kappa.\mathcal{U}_{\Delta',\kappa}}{\Gamma \vdash_\Delta \widehat{\forall}\, t : \mathcal{U}_{\Delta'}} \ \textsc{Ty-}\forall\textsc{-code}$$

$$\frac{\Gamma \vdash_\Delta \qquad \Gamma \vdash_{\Delta,\kappa} t : \overset{\kappa}{\triangleright}\xi.A}{\Gamma \vdash_\Delta \mathsf{prev}\,\kappa.t : \forall\kappa.(A(\mathsf{adv}^\kappa_\Delta(\xi)))} \ \textsc{Ty-prev}$$

Advancing a delayed substitution:

$$\frac{\vdash_{\Delta,\kappa} \cdot : \Gamma \xrightarrow{\kappa} \cdot \qquad \Gamma \vdash_\Delta}{\mathsf{adv}^\kappa_\Delta(\cdot) \triangleq \iota :_{\Delta,\kappa} \Gamma \to \Gamma} \qquad \frac{\vdash_{\Delta,\kappa} \xi[x \leftarrow t] : \Gamma \xrightarrow{\kappa} \Gamma', x : A \qquad \Gamma \vdash_\Delta}{\mathsf{adv}^\kappa_\Delta(\xi[x \leftarrow t]) \triangleq \mathsf{adv}^\kappa_\Delta(\xi)[x := (\mathsf{prev}\,\kappa.t)[\kappa]] :_{\Delta,\kappa} \Gamma \to \Gamma, \Gamma', x : A}$$

Definitional type equalities:

$$\frac{\Gamma \vdash_\Delta \qquad \Delta' \subseteq \Delta \qquad \Gamma \vdash_{\Delta,\kappa} t : \mathcal{U}_{\Delta',\kappa}}{\Gamma \vdash_\Delta \mathsf{El}(\widehat{\forall}\,\Lambda\kappa.t) \simeq \forall\kappa.\,\mathsf{El}(t)} \ \textsc{TyEq-}\forall\textsc{-el} \qquad \frac{\Gamma \vdash_\Delta \qquad \Gamma \vdash_{\Delta,\kappa} A \ \mathsf{type} \qquad \Gamma \vdash_\Delta t : \forall\kappa.A \qquad \Gamma \vdash_\Delta s : \forall\kappa.A}{\Gamma \vdash_\Delta \forall\kappa.\mathsf{Id}_A(t[\kappa]\,, s[\kappa]) \simeq \mathsf{Id}_{\forall\kappa.A}(t, s)} \ \textsc{TyEq-}\forall\textsc{-Id}$$

Definitional term equalities:

$$\frac{\Gamma \vdash_\Delta \qquad \vdash_\Delta \kappa' \qquad \Gamma \vdash_{\Delta,\kappa} t : A}{\Gamma \vdash_\Delta (\Lambda\kappa.t)\left[\kappa'\right] \equiv t[\kappa'/\kappa] : A[\kappa'/\kappa]} \ \textsc{TmEq-}\forall\text{-}\beta \qquad \frac{\kappa \notin \Delta \qquad \Gamma \vdash_\Delta t : \forall\kappa.A}{\Gamma \vdash_\Delta \Lambda\kappa.t[\kappa] \equiv t : \forall\kappa.A} \ \textsc{TmEq-}\forall\text{-}\eta$$

$$\frac{\kappa \notin \Delta \qquad \Gamma \vdash_\Delta A \ \mathsf{type} \qquad \Gamma \vdash_\Delta t : \forall\kappa.A \qquad \vdash_\Delta \kappa' \qquad \vdash_\Delta \kappa''}{\Gamma \vdash_\Delta t\left[\kappa'\right] \equiv t\left[\kappa''\right] : A} \ \textsc{TmEq-}\forall\textsc{-fresh}$$

$$\frac{\Gamma \vdash_\Delta \qquad \vdash_{\Delta,\kappa} \xi : \Gamma \xrightarrow{\kappa} \Gamma' \qquad \Gamma, \Gamma' \vdash_{\Delta,\kappa} t : A}{\Gamma \vdash_\Delta \mathsf{prev}\,\kappa.\,\mathsf{next}^\kappa\,\xi.t \equiv \Lambda\kappa.t(\mathsf{adv}^\kappa_\Delta(\xi)) : \forall\kappa.(A(\mathsf{adv}^\kappa_\Delta(\xi)))} \ \textsc{TmEq-prev-}\beta \qquad \frac{\Gamma \vdash_\Delta \qquad \Gamma \vdash_{\Delta,\kappa} t : \overset{\kappa}{\triangleright}A}{\Gamma \vdash_{\Delta,\kappa} \mathsf{next}^\kappa\,((\mathsf{prev}\,\kappa.t)[\kappa]) \equiv t : \overset{\kappa}{\triangleright}A} \ \textsc{TmEq-prev-}\eta$$

Figure 3. Overview of new rules for coinductive types.

First we define the type of co-natural numbers $\mathsf{Co}\mathbb{N}$ as

$$\mathsf{Co}\mathbb{N} = \forall\kappa.\,\mathsf{Co}\mathbb{N}^\kappa\,.$$

It is easy to define $\overline{0}$ and $\overline{\mathsf{succ}}$ as

$$\begin{array}{ll} \overline{0} : \mathsf{Co}\mathbb{N} & \overline{\mathsf{succ}} : \mathsf{Co}\mathbb{N} \to \mathsf{Co}\mathbb{N} \\ \overline{0} \triangleq \Lambda\kappa.\,\mathsf{inl}\,\langle\rangle & \overline{\mathsf{succ}} \triangleq \lambda n.\Lambda\kappa.\,\mathsf{inr}\,(\mathsf{next}^\kappa\,(n[\kappa])) \end{array}.$$

Next, we will use type isomorphisms to define a transport function $\mathsf{com}^{\mathsf{Co}\mathbb{N}}$ of type $\mathsf{com}^{\mathsf{Co}\mathbb{N}} : \mathsf{Co}\mathbb{N} \to 1 + \mathsf{Co}\mathbb{N}$ as

$$\begin{aligned} \mathsf{com}^{\mathsf{Co}\mathbb{N}} \triangleq \lambda n.\,&\mathsf{case}\,\mathsf{com}^+\,n\,\mathsf{of} \\ &\mathsf{inl}\,u \Rightarrow \mathsf{inl}\,u[\kappa_0] \\ &\mathsf{inr}\,n \Rightarrow \mathsf{inr}\,\mathsf{prev}\,\kappa.n[\kappa] \end{aligned}$$

This function satisfies term equalities

$$\mathsf{com}^{\mathsf{Co}\mathbb{N}}\,\overline{0} \equiv \mathsf{inl}\,\langle\rangle \qquad \mathsf{com}^{\mathsf{Co}\mathbb{N}}(\overline{\mathsf{succ}}\,n) \equiv \mathsf{inr}\,n. \tag{9}$$

Using this we can define type of covectors CoVec_A as

$$\mathsf{CoVec}_A\ n \triangleq \forall\kappa.\,\mathsf{CoVec}^\kappa_A\ n$$

where $\mathsf{CoVec}^\kappa_A : \mathsf{Co}\mathbb{N} \to \mathcal{U}_{\Delta,\kappa}$ is the term

$$\begin{aligned} \mathsf{fix}^\kappa\,\phi.\lambda\,(n : \mathsf{Co}\mathbb{N})\,.\,&\mathsf{case}\,\mathsf{com}^{\mathsf{Co}\mathbb{N}}\,n\,\mathsf{of} \\ &\mathsf{inl}\,\_ \Rightarrow \widehat{1} \\ &\mathsf{inr}\,n \Rightarrow A\widehat{\times}\widehat{\triangleright}^\kappa\,(\phi \circledast_\kappa (\mathsf{next}^\kappa\,n))\,. \end{aligned}$$

Notice the use of $\mathsf{com}^{\mathsf{Co}\mathbb{N}}$ to transport n of type $\mathsf{Co}\mathbb{N}$ to a term of type $1 + \mathsf{Co}\mathbb{N}$ which we can case analyse. To see that this type satisfies the correct type equalities we need some auxiliary term equalities which follow from the way we have defined the terms.

Using term equalities (8) and (9) we can derive the (almost) expected type equalities

$$\begin{aligned} \mathsf{CoVec}_A\ \overline{0} &\simeq \forall\kappa.1 \\ \mathsf{CoVec}_A\,(\overline{\mathsf{succ}}\,n) &\simeq \forall\kappa.\,\left(A \times \overset{\kappa}{\triangleright}(\mathsf{CoVec}^\kappa\,n)\right) \end{aligned} \tag{10}$$

and using the type isomorphisms we can extend these type equalities to type isomorphisms

$$\begin{aligned} \mathsf{CoVec}_A\ \overline{0} &\cong 1 \\ \mathsf{CoVec}_A\,(\overline{\mathsf{succ}}\,n) &\cong A \times \mathsf{CoVec}_A\ n \end{aligned}$$

which are the expected type properties of the covector type.

A simple function we can define is the tail function

$$\begin{aligned} &\mathsf{tl} : \mathsf{CoVec}_A(\overline{\mathsf{succ}}\,n) \to \mathsf{CoVec}_A \\ &\mathsf{tl} \triangleq \lambda v.\,\mathsf{prev}\,\kappa.\pi_2\,(v[\kappa])\,. \end{aligned}$$

Note that we have used (10) to ensure that tl is type correct.

Next, we define the map function on covectors.

$$\begin{aligned} \mathsf{map} &: (A \to B) \to \Pi(n : \mathsf{Co}\mathbb{N}).\,\mathsf{CoVec}_A\,n \to \mathsf{CoVec}_B\,n \\ \mathsf{map}\,f &= \lambda n.\lambda xs.\Lambda\kappa.\,\mathsf{map}^\kappa\,f\,n\ (xs[\kappa]) \end{aligned}$$

where map^κ is the function of type

$$\mathsf{map}^\kappa : (A \to B) \to \Pi(n : \mathsf{Co}\mathbb{N}).\,\mathsf{CoVec}^\kappa_A\,n \to \mathsf{CoVec}^\kappa_B\,n$$

defined as

$$\begin{aligned}\lambda f.\, \mathsf{fix}^{\kappa}\, \phi.\lambda n.\, & \mathsf{case}\ \mathsf{com}^{\mathsf{Co}\mathbb{N}}\, n\ \mathsf{of} \\ & \mathsf{inl}\ \_ \Rightarrow \lambda v.v \\ & \mathsf{inr}\, n \Rightarrow \lambda v.\, \langle f(\pi_1 v), \phi \text{ \textcircled{κ} } (\mathsf{next}^{\kappa}\, n) \text{ \textcircled{κ} } \pi_2(v)\rangle\, .\end{aligned}$$

Let us see that this has the correct type. Let $D_A(x)$ (and analogously $D_B(x)$) be the type

$$\begin{aligned} & \quad \mathsf{case}\ x\ \mathsf{of} \\ D_A(x) \quad \triangleq \quad & \quad \mathsf{inl}\ \_ \Rightarrow \widehat{1} \\ & \quad \mathsf{inr}\, n \Rightarrow A \widehat{\times} \widehat{\triangleright}^{\kappa} \left((\mathsf{next}^{\kappa}\, \mathsf{CoVec}^{\kappa}_{A}) \text{ \textcircled{κ} } (\mathsf{next}^{\kappa}\, n)\right).\end{aligned}$$

where x is of type $1 + \mathsf{Co}\mathbb{N}$. Using this abbreviation we can write the type of map^{κ} as

$$(A \to B) \to \Pi(n : \mathsf{Co}\mathbb{N}).D_A(\mathsf{com}^{\mathsf{Co}\mathbb{N}}\, n) \to D_B(\mathsf{com}^{\mathsf{Co}\mathbb{N}}\, n).$$

Using this it is straightforward to show, using the dependent elimination rule for sums, as we did in Section 3.2, that map^{κ} has the correct type. Indeed we have $D_A(\mathsf{inl}\, z) \simeq 1$ and $D_A(\mathsf{inr}\, n) \simeq A \times \overset{\kappa}{\triangleright} (\mathsf{CoVec}_A\, n)$.

6. Example proofs with coinductive types

6.1 Lifting guarded functions

In this section we show how to lift a function on guarded recursive types, such as addition of guarded streams, to a function on coinductive streams, as mentioned in the introduction. Moreover, we show how to lift proofs of properties, such as the commutativity of addition, from guarded recursive types to coinductive types.

Let Γ be a context in clock context Δ and κ a fresh clock. Suppose A and B are types such that $\Gamma \vdash_{\Delta,\kappa} A$ type and $\Gamma, x : A \vdash_{\Delta,\kappa} B$ type. Finally let f be a function of type $\Gamma \vdash_{\Delta,\kappa} f : \Pi(x : A).B$. We define $\mathfrak{L}(f)$ satisfying the typing judgement

$$\Gamma \vdash_{\Delta} \mathfrak{L}(f) : \Pi(y : \forall\kappa.A).\forall\kappa.\, (B\, [y[\kappa]\, /x])$$

as $\mathfrak{L}(f) \triangleq \lambda y.\Lambda\kappa.f\, (y[\kappa])$ which is easily seen to be type correct.

Now assume that f' is another term of type $\Pi(x : A).B$ (in the same context) and that we have proved

$$\Gamma \vdash_{\Delta,\kappa} p : \Pi(x : A).\mathsf{Id}_B\big(f\, x, f'\, x\big).$$

As above we can give the term $\mathfrak{L}(p)$ the type

$$\Pi(y : \forall\kappa.A).\forall\kappa.\mathsf{Id}_{B[y[\kappa]/x]}\big(f(y[\kappa]), f'(y[\kappa])\big)$$

which by using the type equality TYEQ-∀-ID and the η rule for $\forall$ is equal to the type

$$\Pi(y : \forall\kappa.A).\mathsf{Id}_{\forall\kappa.B[y[\kappa]/x]}\big(\mathfrak{L}(f)\, y, \mathfrak{L}(f')\, y\big).$$

So we have derived a property of lifted functions $\mathfrak{L}(f)$ and $\mathfrak{L}(f')$ from the properties of the guared versions f and f'. This is a standard pattern. Using Löb induction we prove a property of a function whose result is a "guarded" type and derive the property for the lifted function.

Using this pattern we can lift the addition function from guarded streams to coinductive streams and prove that this addition function on streams is commutative, because the addition function on guarded streams is, as we proved in Section 3.1. This justifies (3) from the introduction.

6.2 A property of an acausal function

We will now prove that if we interleave two streams and then apply the 'every other' function eo^{κ} to its output, then we get back the first stream. Precisely, this means that the type

$$\begin{aligned}&\Pi\, (xs, ys : \mathsf{Str}_A)\, . \\ &\mathsf{Id}_{\mathsf{Str}^{\kappa}_A}\begin{pmatrix} \mathsf{eo}^{\kappa}(\Lambda\kappa'.\, \mathsf{merge}^{\kappa'}\, (xs[\kappa'])\, (ys[\kappa'])), \\ xs[\kappa] \end{pmatrix}\end{aligned} \tag{11}$$

is inhabited in clock context Δ, κ.

The function $\mathsf{merge}^{\kappa'}$ of type $\mathsf{merge}^{\kappa'} : \mathsf{Str}^{\kappa'}_A \to \mathsf{Str}^{\kappa'}_A \to \mathsf{Str}^{\kappa'}_A$ is defined using guarded recursion as

$$\begin{aligned}\mathsf{merge}^{\kappa'} \triangleq \mathsf{fix}^{\kappa'}\, \phi.\lambda\, & \left(xs, ys : \mathsf{Str}^{\kappa'}_A\right). \\ & \mathsf{cons}^{\kappa'}(\mathsf{hd}^{\kappa'}\, xs)\, \left(\phi \text{ \textcircled{κ'} } (\mathsf{next}^{\kappa'}\, ys) \text{ \textcircled{κ'} } \mathsf{tl}^{\kappa'}\, xs\right)\end{aligned}$$

Let us prove (11) by Löb induction on the clock κ. First we compute some definitional equalities where we use is as the abbreviation for the term $\Lambda\kappa'.\, \mathsf{merge}^{\kappa'}\, (xs[\kappa'])\, (ys[\kappa'])$. We have

$$\begin{aligned}\mathsf{hd}\, is &\equiv \mathsf{hd}^{\kappa_0}\, (xs[\kappa_0]) \\ \mathsf{tl}\, (\mathsf{tl}\, is) &\equiv \Lambda\kappa'.\, \mathsf{merge}^{\kappa'}\, \big(\mathsf{tl}\, (xs)\big[\kappa'\big]\big)\, \big(\mathsf{tl}\, (ys)\big[\kappa'\big]\big)\, .\end{aligned} \tag{12}$$

Hence from (12) we get $\mathsf{hd}^{\kappa}\, \mathsf{eo}^{\kappa}(is) \equiv \mathsf{hd}^{\kappa_0}\, (xs[\kappa_0])$ and then the rule TMEQ-∀-FRESH gives us $\mathsf{hd}^{\kappa}\, xs[\kappa] \equiv \mathsf{hd}^{\kappa}\, \mathsf{eo}^{\kappa}(is)$. From the definition of eo^{κ} we get $\mathsf{tl}^{\kappa}(\mathsf{eo}^{\kappa}(is)) \equiv \mathsf{next}^{\kappa}\, (\mathsf{eo}^{\kappa}(\mathsf{tl}\, (\mathsf{tl}\, (is))))$ and notice that $\mathsf{tl}^{\kappa}\, (xs[\kappa]) \equiv \mathsf{next}^{\kappa}\, \mathsf{tl}\, (xs)[\kappa]$ by the TMEQ-prev-η and the definition of tl. Thus by Löb induction we have that the tails are equal and we have shown above that the heads of streams $\mathsf{eo}^{\kappa}\, is$ and $xs[\kappa]$ are also equal so we can use $\mathsf{p}\eta$ to construct the proof. The full term is

$$\begin{aligned}&\lambda\, (xs\; ys : \mathsf{Str}_A)\, . \\ &\quad \mathsf{fix}^{\kappa}\, \phi.\, \mathsf{p}\eta\, (\mathsf{r}\, (\mathsf{hd}^{\kappa}\, xs[\kappa])) \\ &\qquad (\phi \text{ \textcircled{κ} } \mathsf{next}^{\kappa}\, (\mathsf{tl}\, (xs)) \text{ \textcircled{κ} } \mathsf{next}^{\kappa}\, (\mathsf{tl}\, (ys)))\, .\end{aligned}$$

7. Soundness

We show soundness of gDTT by giving a model interpreting the type theory. In this section we give an overview of how that is done. The model is quite intricate, but most of the complexity has to do with clock quantification and clock substitution. We therefore first explain how a calculus without clock quantification and where we only have one clock available would be modelled.

7.1 Model of the calculus with only one clock

The subsystem of gDTT with only one clock can be modelled in the category $\mathcal{S}$, known as the topos of trees, the presheaf category over the first infinite ordinal ω.

Concretely, the objects X of $\mathcal{S}$ are families of sets $X_1, X_2, \ldots$ indexed by the positive integers, equipped with families of *restriction functions* $r^X_i : X_{i+1} \to X_i$ indexed similarly. Morphisms $f : X \to Y$ are families of functions $f_i : X_i \to Y_i$ indexed similarly obeying the naturality condition $f_i \circ r^X_i = r^Y_i \circ f_{i+1}$.

There is a functor $\blacktriangleright : \mathcal{S} \to \mathcal{S}$ which maps an object X to the object

$$\begin{tikzcd} 1 & X(1) \arrow[l, "!"'] & X(2) \arrow[l, "r^X_1"'] & X(3) \arrow[l, "r^X_2"'] & \cdots \arrow[l] \end{tikzcd}$$

where ! is the unique map into the terminal object. Intuitively, the functor $\blacktriangleright$ freezes the value so that it is not available now, but only *later*.

The category $\mathcal{S}$ is known to satisfy enough categorical properties to interpret dependent type theory, and Birkedal et al. [7] showed that it also models guarded recursive types and functions. For example, the type of guarded streams of natural numbers is modelled as the family

$$N^1 \longleftarrow N^2 \longleftarrow N^3 \longleftarrow \cdots \tag{13}$$

of sets N^i, for $i \in 1, 2, \ldots$, with restriction maps $r_i : N^{i+1} \to N^i$ mapping $(n_1, \ldots, n_i, n_{i+1})$ to $(n_1, \ldots, n_i)$. Thus the head is available immediately, but the tail is only available after one time step, and if we have i time steps, we know the first i elements of a stream.

We now explain how $\mathcal{S}$ also models delayed substitutions.

Let us look at a simple example, where the context Γ is empty. Suppose that we have a type $x : A \vdash B$ type and term $\vdash t : \triangleright A$. In this model, the type A is interpreted as a family of sets A_i and the type $x : A \vdash B$ type is interpreted as an indexed family of sets $B_i(a)$, for a in A_i. The term t gives us a morphism $t : 1 \to \triangleright A$ so $t_i(*)$ is an element of A_i (here we write $*$ for the element of 1).

We can then form the type $\vdash \triangleright [x \leftarrow t] .B$ type by the following instance of the delayed substitution in types judgment

$$\frac{x : A \vdash B \ \mathsf{type} \qquad \vdash t : \triangleright A}{\vdash \triangleright [x \leftarrow t] .B \ \mathsf{type}}$$

This type $\vdash \triangleright [x \leftarrow t] .B$ type is interpreted as a family of sets X_i, with

$$X_1 = 1 \qquad\qquad X_{i+1} = B_i(t_{i+1}(*)).$$

Notice that the delayed substitution is interpreted by substitution (reindexing) in the model; the change of the index in the model (B_i is reindexed along $t_{i+1}(*)$) corresponds to the delayed substitution in the type theory. Further notice that if B does not depend on x, then the interpretation of $\vdash \triangleright [x \leftarrow t] .B$ type reduces to the interpretation $\triangleright B$, as it should.

The above can be generalised to work for general contexts and sequences of delayed substitutions, and one can then validate that the definitional equality rules do indeed hold in this the model.

7.2 The model in general

To model multiple clocks and clock substitution we use a refinement of Bizjak and Møgelberg's [8] model. The refinement addresses the coherence problem which was left open in *loc. cit.*. Moreover, we show that the model also supports the new constructs of gDTT, in particular delayed substitutions. We now briefly explain how the calculus is modelled and how we address coherence. More details are in the appendix provided as supplementary material.

The model $\mathfrak{M}$ we construct is an indexed category. This means that for each clock context Δ there is a category $\mathfrak{M}(\Delta)$, which is a model of extensional dependent type theory. For instance if there is only one clock, then $\mathfrak{M}(\Delta)$ is the category $\mathcal{S}$ from above. Judgements in clock context Δ are interpreted in $\mathfrak{M}(\Delta)$. Further for each function σ from $\Delta \to \Delta' \cup \{\kappa_0\}$ there is a functor $\mathfrak{M}(\sigma)$ from $\mathfrak{M}(\Delta)$ to $\mathfrak{M}(\Delta')$. In particular, weakening gives rise to a functor $\mathfrak{M}(\Delta) \to \mathfrak{M}(\Delta, \kappa)$ which has a right adjoint, which is used to interpret $\forall\kappa$.

There are two kinds of substitutions in gDTT. The first is clock substitution, which is semantically interpreted using the functors $\mathfrak{M}(\sigma)$. However this interpretation needs to be coherent in the following sense. Given a clock substitution $\sigma : \Delta \to \Delta' \cup \{\kappa_0\}$ we must have e.g., $[\![\Gamma\sigma \vdash_{\Delta'}]\!] = \mathfrak{M}(\sigma) [\![\Gamma \vdash_\Delta]\!]$, where we write $\Gamma\sigma$ for syntactic clock substitution, and analogously for other judgements. The second kind of substitution is ordinary substitution of terms in types and terms in dependent type theory. This does not change clock context, but changes the term variable context so is interpreted separately in each $\mathfrak{M}(\Delta)$. Each context $\Gamma \vdash_\Delta$ is interpreted as an object of $\mathfrak{M}(\Delta)$. A context morphism $\rho : \Gamma \to \Gamma'$ is interpreted as a morphism of type $[\![\Gamma \vdash_\Delta]\!] \to [\![\Gamma' \vdash_\Delta]\!]$ which in turn gives rise to an operation ρ^* used to model substitution along ρ. This operation should also be coherent in the sense that, for instance, $\rho^* ([\![\Gamma \vdash_\Delta]\!]) = [\![\Gamma\rho \vdash_\Delta]\!]$.

It is well-known that constructing models of dependent type theories that are coherent in the latter sense is already challenging. It involves *choosing* categorical structure to intrepret, e.g., Π types, and the choices must be preserved by ρ^* *up to equality*. However to model gDTT we not only need each $\mathfrak{M}(\Delta)$ to be coherent in the latter sense; we also need each functor $\mathfrak{M}(\sigma)$ to satisfy the former coherence condition, which means that it must preserve the different choices of structure made in different categories $\mathfrak{M}(\Delta)$.

Bizjak and Møgelberg's model [8] only satisfies these coherence conditions up to isomorphism. Our refinement of the model replaces the categories $\mathfrak{GR}(\Delta)$ used in *loc. cit.* by equivalent categories $\mathfrak{M}(\Delta)$, which can be equipped with a choice of structure, which is preserved on the nose by the functors $\mathfrak{M}(\Delta)$. To soundly interpret equalities involving universe inclusions, e.g., that if A is a code in $\mathcal{U}_\Delta$ and A' is the inclusion of A into the universe $\mathcal{U}_{\Delta,\kappa}$ then $\mathsf{El}(A) \simeq \mathsf{El}(A')$, we construct universes with *unique codes* [22]. See the appendix provided as supplementary material for more detail.

Altogether we get the following soundness theorem:

Theorem. *All the rules of* gDTT *are validated by the model* $\mathfrak{M}$.

8. Related Work

Birkedal et al. [7] introduced dependent type theory with the $\triangleright$ modality, with semantics in the topos of trees. The guardedness requirement was expressed using the syntactic check that every occurrence of a type variables lies beneath a $\triangleright$. This requirement was subsequently refined by Birkedal and Møgelberg [6], who showed that guarded recursive types could be constructed via fixed-points of certain functions on universes. The authors also showed that such a dependent type theory is consistent with models of intensional type theory, and in particular with univalence [27].

However the rules considered in these papers do not allow one to apply terms of type $\triangleright(\Pi(x : A).B)$, as the applicative functor construction $\circledast$ was defined only for simple function spaces. They are therefore less expressive for both programming (consider the covector ones, and function map, of Section 3.2) and proving, noting the extensive use of generalised $\triangleright$ in our example proofs. They further do not consider first-class coinductive types, and so are restricted to causal functions, as discussed in the introduction.

The extension to coinductive types, and hence acausal functions, is due to Atkey and McBride [4] who introduced *clock quantifiers* into a simply typed setting with guarded recursion. Møgelberg [22] extended Atkey and McBride's clocks to dependent types and Bizjak and Møgelberg [8] refined the model further to allow clock synchronisation. This model is the basis for the rules of gDTT that we have described.

Sized types [15] have been combined with copatterns by Abel and Pientka [1] as an alternative type-based approach for modular programming with coinductive types. This work is more mature than ours with respect to implementation and the demonstration of syntactic properties such as normalisation, and so further development of gDTT is essential to enable proper comparison. We can see that gDTT has a particularly well-developed denotational semantics, which in particular allow equality proofs on coinductive types to be expressed as elements of an extensional identity type, rather than less directly via a bisimulation argument. Further, the later modality is useful for examples beyond coinduction, and beyond the utility of sized types, such as the guarded recursive domain equations used to model program logics [26].

9. Conclusion and Future Work

We have described the dependent type theory gDTT. The examples we have detailed show that gDTT provides a setting for programming and proving with guarded recursive and coinductive types.

Inevitably, dependent type theory is notationally heavy, but we believe that our example programs, including our proof terms, are fairly straightforward. The majority of the complexity comes from confirming that the terms we have written have the type we claim; even this is achieved by a fairly mechanical application of rules.

In future work we plan to investigate an intensional version of the type theory and construct a prototype implementation to allow us to experiment with larger examples. Our use of type equivalences (rules TYEQ-$\forall$-ID and TYEQ-$\triangleright$) suggests that an approach based on observational type theory [2] is the most natural way towards an intensional version with decidable typechecking and good syntactic properties, in particular normalisation and canonicity. Our type equivalences are reminiscent to type equivalences arising in homotopy type theory [27] and thus recent work on the computational interpretation of homotopy type theory [5, 13, 18, 19] may also prove useful.

References

[1] A. Abel and B. Pientka. Wellfounded recursion with copatterns: A unified approach to termination and productivity. In *ICFP*, pages 185–196, 2013.

[2] T. Altenkirch, C. McBride, and W. Swierstra. Observational equality, now! In *PLPV*, pages 57–68. ACM, 2007.

[3] A. W. Appel, P.-A. Melliès, C. D. Richards, and J. Vouillon. A very modal model of a modern, major, general type system. In *POPL*, pages 109–122, 2007.

[4] R. Atkey and C. McBride. Productive coprogramming with guarded recursion. In *ICFP*, pages 197–208, 2013.

[5] M. Bezem, T. Coquand, and S. Huber. A Model of Type Theory in Cubical Sets. In *TYPES 2013*, volume 26 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 107–128. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2014.

[6] L. Birkedal and R. E. Møgelberg. Intensional type theory with guarded recursive types qua fixed points on universes. In *LICS*, pages 213–222, 2013.

[7] L. Birkedal, R. E. Møgelberg, J. Schwinghammer, and K. Støvring. First steps in synthetic guarded domain theory: step-indexing in the topos of trees. *LMCS*, 8(4), 2012.

[8] A. Bizjak and R. E. Møgelberg. A model of guarded recursion with clock synchronisation. In *MFPS*, 2015.

[9] E. Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *J. Funct. Programming*, 23(5): 552–593, 2013.

[10] R. Clouston, A. Bizjak, H. B. Grathwohl, and L. Birkedal. Programming and reasoning with guarded recursion for coinductive types. In *FoSSaCS*, 2015.

[11] R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1986. ISBN 0-13-451832-2.

[12] T. Coquand. Infinite objects in type theory. In *TYPES*, pages 62–78, 1993.

[13] T. Coquand. Lecture notes on cubical sets. Lecture notes (http://www.cse.chalmers.se/~coquand/course3.pdf), 2015.

[14] E. Giménez. Codifying guarded definitions with recursive schemes. In *TYPES*, pages 39–59, 1995.

[15] J. Hughes, L. Pareto, and A. Sabry. Proving the correctness of reactive systems using sized types. In *POPL*, pages 410–423, 1996.

[16] B. Jacobs. *Categorical Logic and Type Theory*. Number 141 in Studies in Logic and the Foundations of Mathematics. North Holland, Amsterdam, 1999.

[17] N. R. Krishnaswami and N. Benton. Ultrametric semantics of reactive programs. In *LICS*, pages 257–266, 2011.

[18] D. R. Licata and G. Brunerie. A cubical approach to synthetic homotopy theory. In *LICS*, 2015.

[19] D. R. Licata and R. Harper. Canonicity for 2-dimensional type theory. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, pages 337–348. ACM, 2012.

[20] The Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2004. URL http://coq.inria.fr. Version 8.0.

[21] C. McBride and R. Paterson. Applicative programming with effects. *J. Funct. Programming*, 18(1):1–13, 2008.

[22] R. E. Møgelberg. A type theory for productive coprogramming via guarded recursion. In *CSL-LICS*, 2014.

[23] H. Nakano. A modality for recursion. In *LICS*, pages 255–266, 2000.

[24] U. Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers University of Technology, 2007.

[25] M. Paviotti, R. E. Møgelberg, and L. Birkedal. A model of PCF in guarded type theory. In *MFPS*, 2015.

[26] K. Svendsen and L. Birkedal. Impredicative concurrent abstract predicates. In *ESOP*, pages 149–168, 2014.

[27] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. http://homotopytypetheory.org/book, Institute for Advanced Study, 2013.