Guarded Dependent Type Theory with Coinductive Types

Aleš Bizjak[1(✉)], Hans Bugge Grathwohl[1], Ranald Clouston[1], Rasmus E. Møgelberg[2], and Lars Birkedal[1]

[1] Aarhus University, Aarhus, Denmark
{abizjak,hbugge,ranald.clouston,birkedal}@cs.au.dk
[2] IT University of Copenhagen, Copenhagen, Denmark
mogel@itu.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. 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 with respect to a denotational model.

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 [17], Nuprl [11], Agda [21], and Idris [8], 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. Syntactic guarded recursion [12], as used for example in Coq [13], 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.

To address this challenge, a *type-based* approach to guarded recursion, more flexible than syntactic checks, was first suggested by Nakano [20]. A new modality, written $\triangleright$ and called 'later' [2], allows us to distinguish between data we have access to now, and data which we will get later. This modality 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.

B. Jacobs and C. Löding (Eds.): FOSSACS 2016, LNCS 9634, pp. 20–35, 2016.
DOI: 10.1007/978-3-662-49630-5_2

Types defined via guarded recursion with $\triangleright$ are not standard coinductive types, as their denotation is defined via models based on the *topos of trees* [5]. More pragmatically, the bare addition of $\triangleright$ disallows productive but *acausal* [16] functions such as the 'every other' function that returns every second element of a stream. Atkey and McBride proposed *clock quantifiers* [3] for such functions; these have been extended to dependent types [7,19], and Møgelberg [19, Theorem2] has shown that they allow the definition of types whose denotation is precisely that of standard coinductive types interpreted in set-based semantics. As such, they allow us to program with real coinductive types, while retaining productivity guarantees.

In this paper we introduce the extensional guarded dependent type theory gDTT, which provides a framework where guarded recursion can be used not just for programming with coinductive types but also for coinductive reasoning.

As types depend on terms, 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 $\triangleright A$. We do this by generalising the applicative functor structure of $\triangleright$ to the dependent setting. Recall the rules for applicative functors [18]:

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

The first rule allows us to make later use of data that we have now. The second allows, for example, functions to be applied recursively to the tails of streams.

Suppose now that f has type $\triangleright(\Pi x : A.B)$, and t has type $\triangleright A$. What should the type of $f \circledast t$ be? 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. This problem occurs in coinductive reasoning: if, e.g., A is $\mathsf{Str}^g_{\mathbb{N}}$, and B a property of streams, in our applications f will be a (guarded) coinduction assumption that we will want to apply to the tail of a stream, which has type $\triangleright\mathsf{Str}^g_{\mathbb{N}}$.

We hence must introduce a new notion, of *delayed substitution*, similar to let-binding, allowing us to give $f \circledast t$ the type

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

binding x in B. Definitional equality rules then allow us to simplify this type when t has form $\mathsf{next}\ u$, i.e., $\triangleright [x \leftarrow \mathsf{next}\ u] . B \equiv \triangleright(B[u/x])$. This construction generalises to bind a list of variables. Delayed substitution is essential to many examples, as shown in Sect. 3, and surprisingly the applicative functor termformer $\circledast$, so central to the standard presentation of applicative functors, turns out to be *definable* via delayed substitutions, as shown in Sect. 2.

Contributions. The contributions of this paper are:

- We introduce the extensional guarded dependent type theory gDTT, and show that it gives a framework for programming and proving with guarded recursive

and coinductive types. The key novel feature is the generalisation of the 'later' type-former and 'next' term-former via *delayed substitutions*;
- We prove the soundness of gDTT via a model similar to that used in earlier work on guarded recursive types and clock quantifiers [7,19].

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 Sects. 7 and 8).

In addition to the examples included in this paper, we are pleased to note 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. [22].

Note that for space reasons many details appear only in the technical report version of this paper [6].

2 Guarded Dependent Type Theory

gDTT is a type theory with base types unit $\mathbf{1}$, booleans $\mathbf{B}$, and natural numbers $\mathbf{N}$, along with Π-types, Σ-types, identity types, and universes. For space reasons we omit all definitions that are standard to such a type theory; see e.g. Jacobs [15]. 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 sending 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.

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

Fig. 1. Judgements in gDTT.

We fix a countable set of *clock variables* $\mathrm{CV} = \{\kappa_1, \kappa_2, \cdots\}$ and a single *clock constant* κ_0, which will be necessary to define, for example, the function hd in Sect. 5. A *clock* is either a clock variable or the clock constant; they are intuitively temporal dimensions on which types may depend. A *clock context* $\Delta, \Delta', \cdots$ is a finite *set* of *clock variables*. 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. All judgements, summarised in Fig. 1, are parametrised by clock contexts. Codes of types inhabit *universes* $\mathcal{U}_\Delta$ parametrised by clock contexts similarly. 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 these explicitly. Note that we do not have $\widehat{\mathcal{U}_{\Delta}} : \mathcal{U}_{\Delta'}$, 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 $\mathcal{U}^1_{\Delta} : \mathcal{U}^2_{\Delta} : \cdots$.

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, where clock substitution $[\kappa'/\kappa]$ is defined as obvious.

The rules for guarded recursion can be found in Figs. 2 and 3; rules for coinductive types are postponed until Sect. 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. As discussed in the introduction, we generalise the applicative functor structure of each $\overset{\kappa}{\triangleright}$ via *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 $f \circledast_{\kappa} t \circledast_{\kappa} u$ (where $\circledast_{\kappa}$ is the applicative functor operation $\circledast$ for clock κ) 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 Sect. 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

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

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 $\overset{\kappa}{\triangleright}\xi.A$. 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* $\circledast_{\kappa}$ as

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

Using the rules in Fig. 2 we can derive the following typing judgement for $\circledast_{\kappa}$

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

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 applies the substitution 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-Force we can derive the basic term equality

$$(\mathsf{next}^{\kappa}\xi.f) \circledast_{\kappa} (\mathsf{next}^{\kappa}\xi.t) \equiv \mathsf{next}^{\kappa}\xi.(ft).$$

typical of applicative functors [18].

It will often 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

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

In other words, delayed substitutions on the type are not necessary when we apply a non-dependent function.

Further, we have the applicative functor identity law

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

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.

Rule TmEq-Next-Comm is not used in the examples of this paper, but it implies the rule $\mathsf{next}^{\kappa}\xi\,[x \leftarrow t]\,.\mathsf{next}^{\kappa}x \equiv \mathsf{next}^{\kappa}t$, which is needed in Paviotti's PhD work.

2.1 Fixed Points and Guarded Recursive Types

In gDTT we have for each clock κ valid in the current clock context a fixed-point combinator fix^{κ}. This differs from a traditional fixed-point combinator in that the type of the recursion variable is not the same as the result type; instead its type is *guarded* with $\overset{\kappa}{\triangleright}$. 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, we say we are proving by *Löb induction* [2].

Guarded recursive types are defined as fixed-points of suitably guarded functions on universes. This is the approach of Birkedal and Møgelberg [4], but the generality of the rules of gDTT allows us to define more interesting dependent guarded recursive types, for example the predicates of Sect. 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, for clock κ, to satisfy the equation $\mathsf{Str}^{\kappa}_{A} \equiv A \times \overset{\kappa}{\triangleright}\mathsf{Str}^{\kappa}_{A}$.

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

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:

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

Typing rules:

$$\frac{\Gamma, \Gamma' \vdash_\Delta A\ \mathsf{type} \qquad \vdash_\Delta \xi : \Gamma \xrightarrow{\kappa} \Gamma'}{\Gamma \vdash_\Delta \overset{\kappa}{\triangleright}\xi.A\ \mathsf{type}}\ \textsc{Tf}\text{-}\triangleright \qquad\qquad \frac{\vdash_{\Delta'} \kappa \qquad \Gamma \vdash_\Delta A : \overset{\kappa}{\triangleright}\mathcal{U}_{\Delta'}}{\Gamma \vdash_\Delta \widehat{\triangleright}^\kappa A : \mathcal{U}_{\Delta'}}\ \textsc{Ty}\text{-}\widehat{\triangleright}$$

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

Fig. 2. Overview of the new typing rules involving $\triangleright$ and delayed substitutions.

universe $\mathcal{U}_{\Delta,\kappa}$ to be $S^\kappa_A \triangleq \mathsf{fix}^\kappa X.B \mathbin{\widehat{\times}} \widehat{\triangleright}^\kappa X$, where $\widehat{\times}$ is the code of the (simple) product type. Via the rules of gDTT we can show $\mathsf{Str}^\kappa_A \simeq A \times \overset{\kappa}{\triangleright}\mathsf{Str}^\kappa_A$ as desired.

The head and tail operations, $\mathsf{hd}^\kappa : \mathsf{Str}^\kappa_A \to A$ and $\mathsf{tl}^\kappa : \mathsf{Str}^\kappa_A \to \overset{\kappa}{\triangleright}\mathsf{Str}^\kappa_A$ are simply the first and the second projections. Conversely, we construct streams by pairing. We use the suggestive cons^κ notation which we define as

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

Defining guarded streams is also done via guarded recursion, for example the stream consisting only of ones is defined as $\mathsf{ones} \triangleq \mathsf{fix}^\kappa x.\mathsf{cons}^\kappa 1\, x$.

The rule $\textsc{TyEq-El}\text{-}\triangleright$ 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; see Sect. 3.

2.2 Identity Types

gDTT has standard extensional identity types $\mathsf{Id}_A(t,u)$ (see, e.g., Jacobs [15]) 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 $\textsc{TyEq}\text{-}\triangleright$. This rule, which is validated by the model of Sect. 6, may be thought of by analogy to type equivalences often considered in homotopy type theory [24], such as

$$\mathsf{Id}_{A\times B}(\langle s_1, s_2\rangle, \langle t_1, t_2\rangle) \equiv \mathsf{Id}_A(s_1,t_1) \times \mathsf{Id}_B(s_2,t_2). \tag{2}$$

There are two important differences. The first is that (2) is (using univalence) a propositional type equality, whereas $\textsc{TyEq}\text{-}\triangleright$ specifices a definitional type equality. This is natural in an extensional type theory. The second difference is that

Definitional type equalities:

$$\overset{\kappa}{\triangleright}\xi\,[x \leftarrow t]\,.A \equiv \overset{\kappa}{\triangleright}\xi.A \qquad (\textsc{TyEq-}\triangleright\textsc{-Weak})$$

$$\overset{\kappa}{\triangleright}\xi\,[x \leftarrow t, y \leftarrow u]\,\xi'.A \equiv \overset{\kappa}{\triangleright}\xi\,[y \leftarrow u, x \leftarrow t]\,\xi'.A \qquad (\textsc{TyEq-}\triangleright\textsc{-Exch})$$

$$\overset{\kappa}{\triangleright}\xi\,[x \leftarrow \mathsf{next}^{\kappa}\,\xi.t]\,.A \equiv \overset{\kappa}{\triangleright}\xi.A[t/x] \qquad (\textsc{TyEq-Force})$$

$$\mathsf{El}(\widehat{\triangleright}^{\kappa}\,(\mathsf{next}^{\kappa}\,\xi.t)) \equiv \overset{\kappa}{\triangleright}\xi.\,\mathsf{El}(t) \qquad (\textsc{TyEq-El-}\triangleright)$$

$$\mathsf{Id}_{\overset{\kappa}{\triangleright}\xi.A}(\mathsf{next}^{\kappa}\,\xi.t, \mathsf{next}^{\kappa}\,\xi.s) \equiv \overset{\kappa}{\triangleright}\xi.\mathsf{Id}_A(t,s) \qquad (\textsc{TyEq-}\triangleright)$$

Definitional term equalities:

$$\mathsf{next}^{\kappa}\,\xi\,[x \leftarrow t]\,.u \equiv \mathsf{next}^{\kappa}\,\xi.u \qquad (\textsc{TmEq-Next-Weak})$$

$$\mathsf{next}^{\kappa}\,\xi\,[x \leftarrow t]\,.x \equiv t \qquad (\textsc{TmEq-Next-Var})$$

$$\mathsf{next}^{\kappa}\,\xi\,[x \leftarrow t, y \leftarrow u]\,\xi'.v \equiv \mathsf{next}^{\kappa}\,\xi\,[y \leftarrow u, x \leftarrow t]\,\xi'.v \qquad (\textsc{TmEq-Next-Exch})$$

$$\mathsf{next}^{\kappa}\,\xi.\,\mathsf{next}^{\kappa}\,\xi'.u \equiv \mathsf{next}^{\kappa}\,\xi'.\,\mathsf{next}^{\kappa}\,\xi.u \qquad (\textsc{TmEq-Next-Comm})$$

$$\mathsf{next}^{\kappa}\,\xi\,[x \leftarrow \mathsf{next}^{\kappa}\,\xi.t]\,.u \equiv \mathsf{next}^{\kappa}\,\xi.u[t/x] \qquad (\textsc{TmEq-Force})$$

$$\mathsf{fix}^{\kappa}\,x.t \equiv t[\mathsf{next}^{\kappa}\,(\mathsf{fix}^{\kappa}\,x.t)\,/x] \qquad (\textsc{TmEq-Fix})$$

Fig. 3. New type and term equalities in gDTT. Rules TyEq-▷-Weak and TmEq-Next-Weak require that A and u are well-formed in a context without x. Rules TyEq-▷-Exch and TmEq-Next-Exch assume that exchanging x and y is allowed, i.e., that the type of x does not depend on y and vice versa. Likewise, rule TmEq-Next-Comm assumes that exchanging the codomains of ξ and ξ' is allowed and that none of the variables in the codomains of ξ and ξ' appear in the type of u.

there are terms going in both directions in (2), whereas we would have a term of type $\mathsf{Id}_{\overset{\kappa}{\triangleright}\xi.A}(\mathsf{next}^{\kappa}\xi.t, \mathsf{next}^{\kappa}\xi.u) \to \overset{\kappa}{\triangleright}\xi.\mathsf{Id}_A(t,u)$ without the rule TyEq-▷.

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

3 Examples

In this section we present some example terms typable in gDTT. Our examples will use a term, which we call $\mathsf{p}\eta$, of type $\Pi(s,t : A \times B).\mathsf{Id}_A(\pi_1 t, \pi_1 s) \to \mathsf{Id}_B(\pi_2 t, \pi_2 s) \to \mathsf{Id}_{A\times B}(t,s)$. This term is definable in any type theory with a strong (dependent) elimination rule for dependent sums. The second property we will use is that $\mathsf{Str}^{\kappa}_A \equiv A \times \overset{\kappa}{\triangleright}\mathsf{Str}^{\kappa}_A$. Because hd^{κ} and tl^{κ} are simply first and second projections, $\mathsf{p}\eta$ also has type $\Pi\,(xs, ys : \mathsf{Str}^{\kappa}_A)\,.\mathsf{Id}_A(\mathsf{hd}^{\kappa}xs, \mathsf{hd}^{\kappa}ys) \to \mathsf{Id}_{\overset{\kappa}{\triangleright}\mathsf{Str}^{\kappa}_A}(\mathsf{tl}^{\kappa}xs, \mathsf{tl}^{\kappa}ys) \to \mathsf{Id}_{\mathsf{Str}^{\kappa}_A}(xs, ys)$.

$\mathsf{zipWith}^{\kappa}$ *Preserves Commutativity.* In gDTT we define the $\mathsf{zipWith}^{\kappa}$ function which has the type $(A \to B \to C) \to \mathsf{Str}^{\kappa}_A \to \mathsf{Str}^{\kappa}_B \to \mathsf{Str}^{\kappa}_C$ by

$$\mathsf{zipWith}^{\kappa} f \triangleq \mathsf{fix}^{\kappa}\phi.\lambda xs, ys.\mathsf{cons}^{\kappa}\,(f\,(\mathsf{hd}^{\kappa}xs)\,(\mathsf{hd}^{\kappa}ys))\,(\phi \mathbin{\textcircled{\scriptsize κ}} \mathsf{tl}^{\kappa}xs \mathbin{\textcircled{\scriptsize κ}} \mathsf{tl}^{\kappa}ys).$$

We show that commutativity of f implies commutativity of $\mathsf{zipWith}^\kappa f$, i.e., that

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

is inhabited. The term that inhabits this type is

$$\lambda f.\lambda c.\mathsf{fix}^\kappa \phi.\lambda xs, ys.\mathsf{p}\eta\ (c\,(\mathsf{hd}^\kappa xs)\,(\mathsf{hd}^\kappa ys))\ (\phi \circledast_\kappa \mathsf{tl}^\kappa xs \circledast_\kappa \mathsf{tl}^\kappa ys).$$

Here, ϕ has type $\overset{\kappa}{\triangleright}(\Pi(xs, ys : \mathsf{Str}^\kappa_A).\mathsf{Id}_{\mathsf{Str}^\kappa_B}(\mathsf{zipWith}^\kappa f\,xs\,ys, \mathsf{zipWith}^\kappa f\,ys\,xs))$ so to type the term above, we crucially need delayed substitutions.

An Example with Covectors. The next example is more sophisticated, as it involves programming and proving with a data type that, unlike streams, is dependently typed. Indeed the generalised later, carrying a delayed substitution, is necessary to type even elementary programs. *Covectors* are the potentially infinite version of vectors (lists with length). To define guarded covectors we first need guarded co-natural numbers. The definition in gDTT is $\mathrm{Co}\mathbb{N}^\kappa \triangleq \mathsf{El}\left(\mathsf{fix}^\kappa X.(\widehat{\mathbf{1}} \mathbin{\widehat{+}} \widehat{\triangleright}^\kappa X)\right)$; this type satisfies $\mathrm{Co}\mathbb{N}^\kappa \equiv \mathbf{1} + \overset{\kappa}{\triangleright}\mathrm{Co}\mathbb{N}^\kappa$. Using $\mathrm{Co}\mathbb{N}^\kappa$ we can define the type family of covectors $\mathrm{CoVec}^\kappa_A\,n \triangleq \mathsf{El}(\widehat{\mathrm{CoVec}^\kappa_A}\,n)$, where

$$\begin{aligned}\widehat{\mathrm{CoVec}^\kappa_A} \triangleq\, &\mathsf{fix}^\kappa\left(\phi : \overset{\kappa}{\triangleright}(\mathrm{Co}\mathbb{N}^\kappa \to \mathcal{U}_{\Delta,\kappa})\right).\lambda(n : \mathrm{Co}\mathbb{N}^\kappa).\mathsf{case}\,n\,\mathsf{of}\\ &\mathsf{inl}\ u \Rightarrow\ \widehat{\mathbf{1}}\\ &\mathsf{inr}\ m \Rightarrow\ A \mathbin{\widehat{\times}} \widehat{\triangleright}^\kappa(\phi \circledast_\kappa m).\end{aligned}$$

We will not distinguish between CoVec^κ_A and $\widehat{\mathrm{CoVec}^\kappa_A}$. As an example of covectors, we define ones of type $\Pi(n : \mathrm{Co}\mathbb{N}^\kappa).\,\mathrm{CoVec}^\kappa_\mathbb{N}\,n$ which produces a covector of any length consisting only of ones:

$$\mathsf{ones} \triangleq \mathsf{fix}^\kappa\phi.\lambda(n : \mathrm{Co}\mathbb{N}^\kappa).\mathsf{case}\,n\,\mathsf{of}\,\{\mathsf{inl}\ u \Rightarrow\ \mathsf{inl}\ \langle\rangle; \mathsf{inr}\ m \Rightarrow\ \langle 1, \phi \circledast_\kappa m\rangle\}\,.$$

Although this is one of the simplest covector programs one can imagine, it does not type-check without the generalised later with delayed substitutions.

The map function on covectors is defined as

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

It preserves composition: the following type is inhabited

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

by the term

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

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 $\overset{\kappa}{\triangleright}$. If we had an unrestricted elimination term $\mathsf{elim} : \overset{\kappa}{\triangleright} A \to A$ every type would be inhabited via fix^κ, making the type theory inconsistent.

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. Figure 4 extends the system of Fig. 2 to allow the removal of clocks in such a setting, by introducing *clock quantifiers* $\forall\kappa$ [3,7,19]. 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); see Fig. 5 for the definition. The judgement $\rho :_\Delta \Gamma \to \Gamma'$ expresses that ρ is a context morphism from context $\Gamma \vdash_\Delta$ to the context $\Gamma' \vdash_\Delta$. We use the notation $\rho[t/x]$ 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:

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

The type is correct because advancing the empty delayed substitution in $\overset{\kappa}{\triangleright}$ turns it into the identity substitution ι, and $A\iota \equiv A$. The β and η rules (Fig. 6) 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, we may see an example with a non-empty delayed substitution in 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

$$\mathsf{prev}\ \kappa.\mathsf{next}^\kappa \begin{bmatrix} f \leftarrow \mathsf{next}^\kappa \lambda n.\mathsf{succ}\ n \\ x \leftarrow \mathsf{next}^\kappa 0 \end{bmatrix}.f\,x. \tag{3}$$

$$\frac{\Gamma \vdash_{\Delta} \qquad \Gamma \vdash_{\Delta,\kappa} A \ \mathsf{type}}{\Gamma \vdash_{\Delta} \forall\kappa.A \ \mathsf{type}}\ \textsc{Tf-}\forall \qquad \frac{\Delta' \subseteq \Delta \qquad \Gamma \vdash_{\Delta} t : \forall\kappa.\mathcal{U}_{\Delta',\kappa}}{\Gamma \vdash_{\Delta} \widehat{\forall} t : \mathcal{U}_{\Delta'}}\ \textsc{Ty-}\forall\textsc{-code}$$

$$\frac{\Gamma \vdash_{\Delta} \qquad \Gamma \vdash_{\Delta,\kappa} t : A}{\Gamma \vdash_{\Delta} \Lambda\kappa.t : \forall\kappa.A}\ \textsc{Ty-}\Lambda \qquad \frac{\vdash_{\Delta} \kappa' \qquad \Gamma \vdash_{\Delta} t : \forall\kappa.A}{\Gamma \vdash_{\Delta} t[\kappa'] : A[\kappa'/\kappa]}\ \textsc{Ty-app}$$

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

Fig. 4. Overview of the new typing rules for coinductive types.

$$\frac{\vdash_{\Delta,\kappa} \cdot : \Gamma \overset{\kappa}{\rightarrow\!\!\triangleright} \cdot \qquad \Gamma \vdash_{\Delta}}{\mathsf{adv}^{\kappa}_{\Delta}(\cdot) \triangleq \iota :_{\Delta,\kappa} \Gamma \to \Gamma}$$

$$\frac{\vdash_{\Delta,\kappa} \xi[x \leftarrow t] : \Gamma \overset{\kappa}{\rightarrow\!\!\triangleright} \Gamma', x : A \qquad \Gamma \vdash_{\Delta}}{\mathsf{adv}^{\kappa}_{\Delta}(\xi[x \leftarrow t]) \triangleq \mathsf{adv}^{\kappa}_{\Delta}(\xi)[(\mathsf{prev}\,\kappa.t)[\kappa]\,/x] :_{\Delta,\kappa} \Gamma \to \Gamma, \Gamma', x : A}$$

Fig. 5. Advancing a delayed substitution.

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 (3) 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 the explicit substitutions used, for example, by Clouston et al. [9] where the unary type-former $\Box$ was used in place of clocks. This binding structure ensures, for instance, that the introduction rule Ty-Λ closed under substitution in Γ.

The rule TmEq-$\forall$-fresh 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. In the polymorphic lambda calculus, the corresponding rule for universal quantification over types would be a consequence of relational parametricity.

We further have the construct $\widehat{\forall}$ and the rule Ty-$\forall$-code which witness that the universes are closed under $\forall\kappa$.

To summarise, the new raw types and terms, extending those of Sect. 2, are

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

Finally, we have the equality rule TyEq-$\forall$-Id analogous to the rule TyEq-$\triangleright$. Note that, as in Sect. 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, without this rule, no term in the reverse direction.

4.1 Derivable Type Isomorphisms

The encoding of coinductive types using guarded recursive types crucially uses a family of type isomorphisms commuting $\forall\kappa$ over other type formers [3,19]. 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 $g(f\,x) \equiv x$. The first type isomorphism is $\forall\kappa.A \cong A$ whenever κ is not free in A. The terms $g = \lambda x.\Lambda\kappa.x$ of type $A \to \forall\kappa.A$ and $f = \lambda x.x[\kappa_0]$ of type $A \to \forall\kappa.A$ witness the isomorphism. Note that we used the clock constant κ_0 in an essential way. The equality $f(g\,x) \equiv x$ follows using only the β rule for clock application. The equality $g(f\,x) \equiv x$ follows using by the rule TmEq-$\forall$-fresh.

The following type isomorphisms follow by using β and η laws for the constructs involved.

– 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 important additional type isomorphism witnessing that $\forall\kappa$ commutes with binary sums; however unlike the isomorphisms above we require equality reflection to show that the two functions are inverse to each other up to definitional equality. 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 define a term com^+ of type $\forall\kappa.(A + B) \to \forall\kappa.A + \forall\kappa.B$ which is a inverse to the canonical term. In particular com^+ satisfies the following two equalities which will be used below.

$$\mathsf{com}^+\,(\Lambda\kappa.\mathsf{inl}\;\,t) \equiv \mathsf{inl}\;\,\Lambda\kappa.t \qquad\qquad \mathsf{com}^+\,(\Lambda\kappa.\mathsf{inr}\;\,t) \equiv \mathsf{inr}\;\,\Lambda\kappa.t. \tag{4}$$

5 Example Programs with Coinductive Types

Let A be a type with code $\widehat{A}$ 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

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

With these we can define the *acausal* ‘every other’ function eo^κ that removes every second element of the input stream. It 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, so clock quantification must be used. The function eo^κ of type $\mathsf{Str}_A \to \mathsf{Str}^\kappa_A$ is defined as

$$\mathsf{eo}^\kappa \triangleq \mathsf{fix}^\kappa\phi.\lambda\,(xs : \mathsf{Str}_A)\,.\mathsf{cons}^\kappa(\mathsf{hd}\;xs)\,(\phi \circledast^{\kappa} \mathsf{next}^\kappa\,((\mathsf{tl}\;(\mathsf{tl}\;xs)))).$$

Definitional type equalities:

$$\frac{\Gamma \vdash_{\Delta} \qquad \Delta' \subseteq \Delta \qquad \Gamma \vdash_{\Delta,\kappa} t : \mathcal{U}_{\Delta',\kappa}}{\Gamma \vdash_{\Delta} \mathsf{El}(\widehat{\forall}\, \Lambda\kappa.t) \equiv \forall\kappa.\, \mathsf{El}(t)} \text{ TyEq-}\forall\text{-el}$$

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

Definitional term equalities:

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

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

$$\frac{\Gamma \vdash_{\Delta} \qquad \vdash_{\Delta,\kappa} \xi : \Gamma \overset{\kappa}{\rightarrow} \Gamma' \qquad \Gamma, \Gamma' \vdash_{\Delta,\kappa} t : A}{\Gamma \vdash_{\Delta} \mathsf{prev}\,\kappa.\, \mathsf{next}^{\kappa}\, \xi.t \equiv \Lambda\kappa.t(\mathsf{adv}^{\kappa}_{\Delta}(\xi)) : \forall\kappa.(A(\mathsf{adv}^{\kappa}_{\Delta}(\xi)))} \text{ TmEq-prev-}\beta$$

$$\frac{\Gamma \vdash_{\Delta} \qquad \Gamma \vdash_{\Delta,\kappa} t : \overset{\kappa}{\triangleright} A}{\Gamma \vdash_{\Delta,\kappa} \mathsf{next}^{\kappa}\, ((\mathsf{prev}\,\kappa.t)[\kappa]) \equiv t : \overset{\kappa}{\triangleright} A} \text{ TmEq-prev-}\eta$$

Fig. 6. Type and term equalities involving clock quantification.

The result is a *guarded* stream, but we can easily strengthen it and define eo of type $\mathsf{Str}_A \rightarrow \mathsf{Str}_A$ as $\mathsf{eo} \triangleq \lambda xs.\Lambda\kappa.\mathsf{eo}^{\kappa} xs$.

We can also work with covectors (not just guarded covectors as in Sect. 3). This is a dependent coinductive type indexed by conatural numbers which is the type $\mathsf{Co}\mathbb{N} = \forall\kappa.\,\mathrm{Co}\mathbb{N}^{\kappa}$. It is easy to define $\overline{0}$ and $\overline{\mathsf{succ}}$ as $\overline{0} \triangleq \Lambda\kappa.\mathsf{inl}\ \langle\rangle$ and $\overline{\mathsf{succ}} \triangleq \lambda n.\Lambda\kappa.\mathsf{inr}\ (\mathsf{next}^{\kappa}\ (n[\kappa]))$. Next, we can define a transport function $\mathsf{com}^{\mathsf{Co}\mathbb{N}}$ of type $\mathsf{com}^{\mathsf{Co}\mathbb{N}} : \mathsf{Co}\mathbb{N} \rightarrow 1 + \mathsf{Co}\mathbb{N}$ satisfying

$$\mathsf{com}^{\mathsf{Co}\mathbb{N}}\overline{0} \equiv \mathsf{inl}\ \langle\rangle \qquad \mathsf{com}^{\mathsf{Co}\mathbb{N}}(\overline{\mathsf{succ}} n) \equiv \mathsf{inr}\ n. \tag{5}$$

This function is used to define the type family of covectors as $\mathrm{CoVec}_A\ n \triangleq \forall\kappa.\,\mathrm{CoVec}^{\kappa}_A\ n$ where $\mathrm{CoVec}^{\kappa}_A : \mathsf{Co}\mathbb{N} \rightarrow \mathcal{U}_{\Delta,\kappa}$ is the term

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

Using term equalities (4) and (5) we can derive the type isomorphisms

$$\begin{aligned} \mathrm{CoVec}_A\ \overline{0} &\equiv \forall\kappa.1 \cong 1 \\ \mathrm{CoVec}_A\,(\overline{\mathsf{succ}} n) &\equiv \forall\kappa\left(A \times \overset{\kappa}{\triangleright}(\mathrm{CoVec}^{\kappa}_A\ n)\right) \cong A \times \mathrm{CoVec}_A\ n \end{aligned} \tag{6}$$

which are the expected properties of the type of covectors.

A simple function we can define is the tail function

$$\mathsf{tl} : \mathrm{CoVec}_A(\overline{\mathsf{succ}} n) \rightarrow \mathrm{CoVec}_A \qquad \mathsf{tl} \triangleq \lambda v.\mathsf{prev}\ \kappa.\pi_2\,(v[\kappa])\,.$$

Note that (6) is needed to type tl. The map function of type

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

is defined as $\mathsf{map} f \triangleq \lambda n.\lambda xs.\Lambda\kappa.\mathsf{map}^{\kappa} f\, n\ (xs[\kappa])$ where map^{κ} is

$$\begin{aligned}
\mathsf{map}^{\kappa} &: (A \to B) \to \Pi(n : \mathsf{Co}\mathbb{N}).\, \mathrm{CoVec}^{\kappa}_A\, n \to \mathrm{CoVec}^{\kappa}_B\, n \\
\mathsf{map}^{\kappa} &= \lambda f.\mathsf{fix}^{\kappa}\phi.\lambda n.\mathsf{case}\ \mathsf{com}^{\mathsf{Co}\mathbb{N}} n\ \mathsf{of} \\
&\qquad\qquad\qquad \mathsf{inl}\ \_ \Rightarrow\ \lambda v.v \\
&\qquad\qquad\qquad \mathsf{inr}\ n \Rightarrow\ \lambda v.\, \langle f(\pi_1 v), \phi \circledast (\mathsf{next}^{\kappa} n) \circledast \pi_2(v)\rangle.
\end{aligned}$$

5.1 Lifting Guarded Functions

In this section we show how in general we may lift a function on guarded recursive types, such as addition of guarded streams, to a function on coinductive streams. 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 $\Gamma \vdash_{\Delta,\kappa} A\,\mathsf{type}$ and $\Gamma, x : A \vdash_{\Delta,\kappa} B\,\mathsf{type}$. 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 $\mathfrak{L}(f) \triangleq \lambda y.\Lambda\kappa.f\,(y[\kappa])$.

Now assume that f' is another term of type $\Pi(x : A).B$ (in the same context) and that we have proved $\Gamma \vdash_{\Delta,\kappa} p : \Pi(x : A).\mathsf{Id}_B(f\,x, f'\,x)$. As above we can give the term $\mathfrak{L}(p)$ the type $\Pi(y : \forall\kappa.A).\forall\kappa.\mathsf{Id}_{B[y[\kappa]/x]}(f(y[\kappa]), f'(y[\kappa]))$. which by using the type equality TyEq-$\forall$-Id and the η rule for $\forall$ is equal to the type $\Pi(y : \forall\kappa.A).\mathsf{Id}_{\forall\kappa.B[y[\kappa]/x]}(\mathfrak{L}(f)\,y, \mathfrak{L}(f')\,y)$. So we have derived a property of lifted functions $\mathfrak{L}(f)$ and $\mathfrak{L}(f')$ from the properties of the guarded 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.

For example we can lift the $\mathsf{zipWith}$ function from guarded streams to coinductive streams and prove that it preserves commutativity, using the result on guarded streams of Sect. 3.

6 Soundness

gDTT can be shown to be sound with respect to a denotational model interpreting the type theory. The model is a refinement of Bizjak and Møgelberg’s [7] but for reasons of space we leave the description of a full model of gDTT for future work. Instead, to provide some intuition for the semantics of delayed substitutions, we just describe how to interpret the rule

$$\frac{x : A \vdash B\,\mathsf{type} \qquad \vdash t : \triangleright A}{\vdash \triangleright[x \leftarrow t].B\,\mathsf{type}} \tag{7}$$

in the case where we only have one clock available.

The subsystem of gDTT with only one clock can be modelled in the category $\mathcal{S}$, known as the topos of trees [5], the presheaf category over the first infinite ordinal ω. The objects X of $\mathcal{S}$ are families of sets $X_1, X_2, \ldots$ indexed by the positive integers, together with families of *restriction functions* $r_i^X : X_{i+1} \to X_i$ indexed similarly. There is a functor $\blacktriangleright : \mathcal{S} \to \mathcal{S}$ which maps an object X to the object

$$1 \xleftarrow{!} X_1 \xleftarrow{r_1^X} X_2 \xleftarrow{r_2^X} X_3 \longleftarrow \cdots$$

where ! is the unique map into the terminal object.

In this model, a closed type A is interpreted as an object of $\mathcal{S}$ and the type $x : A \vdash B\,\mathsf{type}$ is interpreted as an indexed family of sets $B_i(a)$, for a in A_i together with maps $r_i^B(a)\colon B_{i+1}(a) \to B_i(r_i^A(a))$. The term t in (7) is interpreted as a morphism $t : 1 \to \triangleright A$ so $t_i(*)$ is an element of A_i (here we write $*$ for the element of 1).

The type $\vdash \triangleright[x \leftarrow t].B\,\mathsf{type}$ is then interpreted as the object X, defined by

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

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 $\vdash \triangleright [x \leftarrow t] .B\,\mathsf{type}$ reduces to the interpretation $\triangleright B$, which is defined to be $\blacktriangleright$ applied to the interpretation of B.

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 model.

7 Related Work

Birkedal et al. [5] 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 variable lies beneath a $\triangleright$. This requirement was subsequently refined by Birkedal and Møgelberg [4], who showed that guarded recursive types could be constructed via fixed-points of functions on universes. 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 Sect. 3) and proving, noting the extensive use of delayed substitutions in our example proofs. They further do not consider coinductive types, and so are restricted to causal functions.

The extension to coinductive types, and hence acausal functions, is due to Atkey and McBride [3], who introduced *clock quantifiers* into a simply typed setting with guarded recursion. Møgelberg [19] extended this work to dependent types and Bizjak and Møgelberg [7] refined the model further to allow clock synchronisation.

Clouston et al. [9] introduced the logic $L\mathsf{g}\lambda$ to prove properties of terms of the (simply typed) guarded λ-calculus, $\mathsf{g}\lambda$. This allowed proofs about coinductive types, but not in the integrated fashion supported by dependent type theories. Moreover it relied on types being "total", a property that in a dependently typed setting would entail a strong elimination rule for $\triangleright$, which would lead to inconsistency.

Sized types [14] have been combined with copatterns [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. One advantage of gDTT is that 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 [23].

8 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.

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. Preliminary work has suggested that the path type of cubical type theory [10] interacts better with the new constructs of gDTT than the ordinary Martin-Löf identity type.

Finally, we are investigating whether the generalisation of applicative functors [18] to apply over *dependent* function spaces, via delayed substitutions, might also apply to examples quite unconnected to the later modality.

Acknowledgements. This research was supported in part by the ModuRes Sapere Aude Advanced Grant and DFF-Research Project 1 Grant no. 4002-00442, both from The Danish Council for Independent Research for the Natural Sciences (FNU). Aleš Bizjak was supported in part by a Microsoft Research PhD grant.

References

1. Abel, A., Pientka, B.: Wellfounded recursion with copatterns: A unified approach to termination and productivity. In: ICFP, pp. 185–196 (2013)
2. Appel, A.W., Melliès, P.A., Richards, C.D., Vouillon, J.: A very modal model of a modern, major, general type system. In: POPL, pp. 109–122 (2007)

3. Atkey, R., McBride, C.: Productive coprogramming with guarded recursion. In: ICFP, pp. 197–208 (2013)
4. Birkedal, L., Møgelberg, R.E.: Intensional type theory with guarded recursive types qua fixed points on universes. In: LICS, pp. 213–222 (2013)
5. Birkedal, L., Møgelberg, R.E., Schwinghammer, J., Støvring, K.: First steps in synthetic guarded domain theory: step-indexing in the topos of trees. LMCS 8(4) (2012)
6. Bizjak, A., Grathwohl, H.B., Clouston, R., Møgelberg, R.E., Birkedal, L.: Guarded dependent type theory with coinductive types (2016). http://arxiv.org/abs/1601.01586
7. Bizjak, A., Møgelberg, R.E.: A model of guarded recursion with clock synchronisation. In: MFPS (2015)
8. Brady, E.: Idris, a general-purpose dependently typed programming language: design and implementation. J. Funct. Program. **23**(5), 552–593 (2013)
9. Clouston, R., Bizjak, A., Grathwohl, H.B., Birkedal, L.: Programming and reasoning with guarded recursion for coinductive types. In: FoSSaCS (2015)
10. Cohen, C., Coquand, T., Huber, S., Mörtberg, A.: Cubical type theory: a constructive interpretation of the univalence axiom, unpublished (2015)
11. Constable, R.L., Allen, S.F., Bromley, H.M., Cleaveland, W.R., Cremer, J.F., Harper, R.W., Howe, D.J., Knoblock, T.B., Mendler, N.P., Panangaden, P., Sasaki, J.T., Smith, S.F.: Implementing Mathematics with the Nuprl Proof Development System. Prentice-Hall Inc, Upper Saddle River, NJ, USA (1986)
12. Coquand, T.: Infinite objects in type theory. In: TYPES, pp. 62–78 (1993)
13. Giménez, E.: Codifying guarded definitions with recursive schemes. In: TYPES, pp. 39–59 (1995)
14. Hughes, J., Pareto, L., Sabry, A.: Proving the correctness of reactive systems using sized types. In: POPL, pp. 410–423 (1996)
15. Jacobs, B.: Categorical Logic and Type Theory. Studies in Logic and the Foundations of Mathematics, vol. 141. North Holland, Amsterdam (1999)
16. Krishnaswami, N.R., Benton, N.: Ultrametric semantics of reactive programs. In: LICS, pp. 257–266 (2011)
17. The Coq development team: the coq proof assistant reference manual. LogiCal Project (2004). version 8.0. http://coq.inria.fr
18. McBride, C., Paterson, R.: Applicative programming with effects. J. Funct. Program. **18**(1), 1–13 (2008)
19. Møgelberg, R.E.: A type theory for productive coprogramming via guarded recursion. In: CSL-LICS (2014)
20. Nakano, H.: A modality for recursion. In: LICS, pp. 255–266 (2000)
21. Norell, U.: Towards a practical programming language based on dependent type theory. Ph.D. thesis, Chalmers University of Technology (2007)
22. Paviotti, M., Møgelberg, R.E., Birkedal, L.: A model of PCF in guarded type theory. In: MFPS (2015)
23. Svendsen, K., Birkedal, L.: Impredicative concurrent abstract predicates. In: Shao, Z. (ed.) ESOP 2014 (ETAPS). LNCS, vol. 8410, pp. 149–168. Springer, Heidelberg (2014)
24. The Univalent Foundations Program: Homotopy Type Theory: Univalent Foundations of Mathematics, Institute for Advanced Study (2013). http://homotopytypetheory.org/book