The Clocks Are Ticking: No More Delays!

Reduction Semantics for Type Theory with Guarded Recursion

Patrick Bahr
IT University of Copenhagen

Hans Bugge Grathwohl
Aarhus University

Rasmus Ejlers Møgelberg
IT University of Copenhagen

***Abstract*—Guarded recursion in the sense of Nakano allows general recursive types and terms to be added to type theory without breaking consistency. Recent work has demonstrated applications of guarded recursion such as programming with codata, reasoning about coinductive types, as well as constructing and reasoning about denotational models of general recursive types. Guarded recursion can also be used as an abstract form of step-indexing for reasoning about programming languages with advanced features.**

Ultimately, we would like to have an implementation of a type theory with guarded recursion in which all these applications can be carried out and machine-checked. In this paper, we take a step towards this goal by devising a suitable reduction semantics.

We present Clocked Type Theory, a new type theory for guarded recursion that is more suitable for reduction semantics than the existing ones. We prove confluence, strong normalisation and canonicity for its reduction semantics, constructing the theoretical basis for a future implementation. We show how coinductive types as exemplified by streams can be encoded, and derive that the type system ensures productivity of stream definitions.

I. Introduction

Type theory in the sense of Martin-Löf [1] plays a double role as a programming language and a logic for verifying programs and formalising mathematics. The logical interpretation forces a requirement of totality on the programming language interpretation, and in particular rules out general recursion. While total programming can be very expressive, it does have limitations also in practice. For example, when programming with coinductive types, the totality requirement enforces the restriction that all coinductive definitions must be productive. In the case of streams, productivity means that any particular element of the stream can be computed in finite time. Modern implementations of type theory such as Coq and Agda, have syntactic productivity checks, but these are not modular and can therefore be difficult to work with [2].

Another place where the limitation of total programming shows is modelling or reasoning about programming languages inside type theory: A common problem is that type theory (like set theory) lacks the recursive types or terms needed, e.g. for modelling the untyped lambda calculus or programming languages with higher-order store. One solution is to formalise theories for recursion, e.g. domain theory, inside type theory, but this can be non-trivial. Having fixed points directly in type theory would allow a more direct (synthetic) approach.

Guarded recursion [3] allows recursion to be added to type theory without breaking consistency by introducing time steps in the form of a delay type modality $\triangleright$ (pronounced 'later'). Elements of type $\triangleright A$ are to be thought of as elements of type A only available one time step from now. Recursion arises from a fixed point operator mapping each productive endofunction (i.e. a function of type $\triangleright A \to A$) to its unique fixed point. Recursive types are fixed points of productive endofunctions on a universe of types. We refer to these as *guarded recursive types* and emphasize that these include recursive types with negative occurrences of type variables.

The most advanced type theory with guarded recursion to date is Guarded Dependent Type Theory (GDTT) [4], an extensional type theory with a notion of clocks, which each have a delay modality. Following an idea of Atkey and McBride [5], coinductive types can be encoded using guarded recursive types and universal quantification over clocks, which allows productivity to be expressed in types. In addition, GDTT has a notion of *delayed substitutions* allowing for coinductive, type theoretic reasoning about coinductive data and functions manipulating coinductive data.

GDTT can also be used as a language for denotational semantics. Møgelberg and Paviotti [6] have shown how to construct a denotational model of FPC (simply typed lambda calculus extended with general recursive types) inside GDTT modeling the recursive types of FPC as guarded recursive types in GDTT. The constructed model is computationally adequate, and this can be proved *entirely within the type theory* GDTT using guarded recursive terms and types. This application of GDTT can be viewed as a form of synthetic domain theory, and it allows for often simpler proofs than those of the classical domain theoretic counterparts, in the same way that the abstract setting of synthetic homotopy theory [7] allows for homotopy theoretic proofs to be simplified.

Guarded recursion has also been used for operational reasoning about languages beyond the reach of known domain theoretic techniques. These include languages with combinations of advanced features such as general references, recursive types, countable non-determinism, and concurrency [8], [9], [10]. The logics used for reasoning about these features, and the syntactic models on which they are based can be understood as a synthetic approach to step-indexing [11]. Thus guarded recursion provides an abstract setting for this powerful technique that hides the intricacies of step-indexing and reveals a formal type system for expressing the model constructions. Note that although these applications use guarded recursion, they have not been formalised in GDTT, but we expect that this is possible.

So far, most work on guarded recursion has used denota-

tional arguments. Indeed, the soundness of GDTT is argued for by using a denotational model. In this work we take the first step towards a syntactic metatheory for GDTT, focusing on the fragment obtained by removing identity types. The development of Guarded Cubical Type Theory [12] has shown that in type theories with guarded recursion, propositional equality should be treated using path types in the sense of Cubical Type Theory [13]. However, the metatheory of the latter is still an open problem.

A. Clocked Type Theory

Delayed substitutions make it difficult to define a reduction semantics directly on GDTT. To solve this problem, we introduce a new type theory called Clocked Type Theory (CloTT), which can be seen as a refinement of GDTT. Like GDTT, CloTT has a notion of clocks, but it also has a notion of ticks on a clock, which can be used to translate the delayed substitutions of GDTT to actual substitutions in CloTT (justifying the title of the paper). Ticks are resources that can be used to unfold fixed point definitions, an operation that must be restricted to avoid divergence. The delay modality $\triangleright$ is replaced by a form of dependent function type over clocks with introduction and elimination rules given by abstraction over ticks, and application to ticks, respectively.

We argue that CloTT is at least as expressive as the fragment of GDTT without identity types, by giving a translation of the latter into the former. The translation maps most of the equational rules of GDTT to equalities that follow from the reduction semantics of CloTT. In particular, most of the rules for delayed substitutions follow in fact from the β and η rules for tick abstraction and standard rules for substitutions. Some of the equational rules of GDTT do not follow from the reduction semantics, but we argue that these are most naturally expressed in CloTT as propositional equalities stating that 'all ticks are equal' and 'all clocks are equal'. We argue informally why these can be added to a future extension of CloTT with path types without breaking canonicity.

Our main results concern the reduction semantics of CloTT, which we show is confluent and strongly normalising. As a consequence of this, equality of terms and types can be decided by reducing these to their unique normal forms. This decision procedure is a major step towards a type checking algorithm for CloTT. We also prove a canonicity theorem stating that every closed term of type Nat reduces to a natural number. As a consequence of this we derive the statement of productivity: Given a well typed closed term of stream type, its n'th element can be computed in finite time. This is a formal statement of the fact that guarded recursion captures productivity of coinductive definitions in types.

B. Related work

This paper is part of a line of work on guarded recursion including the already mentioned Guarded Dependent Type Theory [4] and Guarded Cubical Type Theory [12]. The latter (GCTT) is an extension of Cubical Type Theory [13] with guarded recursion for a single clock (i.e. with no clock variables, disallowing encoding of coinductive types). It has a denotational semantics given by a presheaf model and a prototype implementation. Fixed points are not unfolded automatically, instead there is a path from a fixed point to its unfolding.

In GCTT (like CloTT) guarded dependent types are constructed as fixed points of productive endomaps on a universe. The fold and unfold terms of guarded recursive types arise as transports along the path from the fixed point to its unfolding. Since CloTT has no path types, we have added the fold and unfold terms as primitives to CloTT, but the reduction semantics of these mimic closely those of GCTT. For example, there are no β and η rules for fold and unfold, since these are equivalences of types in GCTT – but not isomorphisms.

Clouston et at. [14] study operational semantics for a simply typed lambda calculus with guarded recursive types and a modal operator related to universal quantification over clocks, which allows coinductive types to be encoded. They show a canonicity result for closed terms of natural number type and construct a logic for the language. Our work extends this in many ways, for example by considering dependent types, reduction of open terms and not fixing the evaluation strategy. These extensions are necessary (the last one perhaps just convenient) for type checking type theories.

The typing rules for ticks, tick abstraction and tick application in CloTT are remarkably similar to those for names in Cheney's Dependent Nominal Type Theory [15]. In CloTT the position of a tick variable in a context can be read as dividing the context into a collection of variables available before that tick (the left half) and those available after. In Dependent Nominal Type Theory, variables declared before a name variable in a context are considered fresh for that name, whereas the ones following are not assumed fresh.

Copatterns [16] and sized types [17], [18] are an alternative approach to ensuring productivity of coinductive definitions through types, whose syntactic metatheory is much further developed than that of guarded recursion. For example, Abel and Pientka [19] construct a type checking algorithm for an extension of System F_ω with these features. The two approaches appear related, since guarded recursion essentially is a synthetic form of step-indexing, and sized types index types with steps, but no formal relation has yet been established. We emphasize that the goal of guarded recursion is more general, since general recursive types are treated, also with negative occurrences of type variables.

Abel and Vezzosi [20] study an increasing, natural number indexed family of reduction relations for a simply typed lambda calculus with guarded recursive types. They prove strong normalisation for each of the reduction relations, but none of them are confluent. Unlike them, we do not unfold fixed points automatically, but instead we gain confluence.

Severi and de Vries [21] study Pure Type Systems, which include dependently typed calculi such as the Calculus of Constructions, extended with essentially a guarded recursive type of streams. Instead of restricting the unfolding of fixed points to obtain strong normalisation as we do for CloTT, their

calculi allow arbitrary unfolding of fixed points. While this choice prevents these calculi from being strongly normalising, Severi and de Vries are able to establish an infinitary strong normalisation property, which implies productivity.

A number of authors [22], [23], [24] have suggested using essentially guarded recursive methods in functional reactive programming. The benefit in that setting is to allow static checking for the lack of time-leaks, a property similar to productivity.

C. Overview

Sections II and III present Clocked Type Theory, its reduction semantics, and the main metatheoretic results. Section IV shows how the guarded recursive types can be used to construct a type of streams and gives examples of basic programs for constructing, manipulating, and reasoning about streams.

Section V recalls the delayed substitutions of GDTT and constructs the translation from GDTT to CloTT. Section VI outlines the proofs of the main results with emphasis on strong normalisation. The full proofs are quite long and technical and we therefore omit them from this extended abstract. The referees can find full proofs in the online appendix [25]. Finally, we conclude in Section VII.

II. Clocked Type Theory

Clocked Type Theory is an extension of dependent type theory with a special collection of sorts called *clocks*. An inhabitant of a clock is referred to as a *tick*, and these are resources that can be used to unfold fixed point definitions or fold or unfold elements of recursive types. We use κ to range over *clock variables* and α to range over ticks, so that an assumption of the form $\alpha : \kappa$ states that α is a tick on clock κ. Clock contexts Δ are finite sets of clock variables and in our syntax these are given a special role by being subscript to the turnstile as in e.g. typing judgements $\Gamma \vdash_\Delta t : A$.

Context and type formation rules of the calculus are shown in Figure 1. Apart from the three base types $\mathsf{1}$, Bool, and Nat, the calculus has standard Π- and Σ-types as well as a Tarski-style universe $\mathcal{U}$ with a corresponding interpretation operator $\mathsf{El}\,(\cdot)$. There are no constructors for clocks, they are just names, and clocks are not types, so declarations like $\alpha : \kappa \to \kappa'$ are not allowed. By not being types, the status of clocks is similar to that of the interval in Cubical Type Theory [13].

In a context $\Gamma, \alpha : \kappa, \Gamma' \vdash_\Delta$, tick variable α represents the assumption that a tick of time occurs on clock κ between the time when the values represented by the variables in Γ and those in Γ' are received. Since values can be implicitly stored over time steps, the context $\Gamma, x : B, \alpha : \kappa, \Gamma' \vdash_\Delta$ represents a stronger assumption than $\Gamma, \alpha : \kappa, x : B, \Gamma' \vdash_\Delta$. The former requires availability of x for one more time step. This intuition is reflected in the structural rules. For example, the rule

$$\frac{\Gamma, \alpha : \kappa, y : B, \Gamma' \vdash_\Delta C \text{ type} \qquad \Gamma, y : B, \alpha : \kappa, \Gamma' \vdash_\Delta}{\Gamma, y : B, \alpha : \kappa, \Gamma' \vdash_\Delta C \text{ type}}$$

is admissible, but the opposite direction is not.

Contexts:

$$\frac{}{\cdot \vdash_\Delta} \qquad \frac{\Gamma \vdash_\Delta \qquad \Gamma \vdash_\Delta A \text{ type}}{\Gamma, x : A \vdash_\Delta} \qquad \frac{\Gamma \vdash_\Delta \qquad \kappa \in \Delta}{\Gamma, \alpha : \kappa \vdash_\Delta}$$

Types:

$$\frac{\Gamma, \alpha : \kappa \vdash_\Delta A \text{ type} \qquad \kappa \in \Delta}{\Gamma \vdash_\Delta \triangleright (\alpha : \kappa).A \text{ type}}$$

$$\frac{\Gamma \vdash_{\Delta,\kappa} A \text{ type} \qquad \Gamma \vdash_\Delta}{\Gamma \vdash_\Delta \forall \kappa.A \text{ type}} \qquad \frac{\Gamma \vdash_\Delta}{\Gamma \vdash_\Delta \mathsf{1} \text{ type}}$$

$$\frac{\Gamma \vdash_\Delta}{\Gamma \vdash_\Delta \mathsf{Bool} \text{ type}} \qquad \frac{\Gamma, x : A \vdash_\Delta B \text{ type}}{\Gamma \vdash_\Delta \Pi x : A.\, B \text{ type}}$$

$$\frac{\Gamma, x : A \vdash_\Delta B \text{ type}}{\Gamma \vdash_\Delta \Sigma x : A.\, B \text{ type}} \qquad \frac{\Gamma \vdash_\Delta}{\Gamma \vdash_\Delta \mathsf{Nat} \text{ type}}$$

$$\frac{\Gamma \vdash_\Delta}{\Gamma \vdash_\Delta \mathcal{U} \text{ type}} \qquad \frac{\Gamma \vdash_\Delta A : \mathcal{U}}{\Gamma \vdash_\Delta \mathsf{El}\,(A) \text{ type}}$$

Fig. 1. Context and type formation rules for Clocked Type Theory.

Selected typing rules can be found in Figure 2. We omit rules for sum types, unit types and natural numbers which are just standard rules ignoring the new context Δ, as in the rules for dependent products included in the figure. Note that, although we think of ticks as resources, the type discipline is not linear as in linear logic [26]. For example, in a function application $t\,u$ a tick variable α can appear both in t and u. Rather, the typing discipline ensures that a tick is not used to unfold a fixed point twice.

The type $\triangleright (\alpha : \kappa).A$ is a type of suspended computations requiring a tick on the clock to compute elements of type A. This can be understood as a dependent function type, but we stick to the notation of the guarded recursion literature, where the $\triangleright$ (pronounced 'later') is a type modality. We write simply $\triangleright^\kappa A$ for $\triangleright (\alpha : \kappa).A$ if α is not free in A. In the elimination rule for $\triangleright (\alpha : \kappa).A$, a term t can be applied to a tick α', only if t computes to a value of type $\triangleright (\alpha : \kappa).A$ before α', i.e., if all of the variables that t depend on are available before α'.

A suspended computation represented by a closed term of type $\triangleright (\alpha : \kappa).A$ can be forced by applying it to the tick constant $\diamond$. In general, this is unsafe for open terms, since it breaks the productivity guarantees that the typing system should provide. For example, the term $\lambda(x : \triangleright^\kappa A).x\,[\diamond]$ should not be typeable, because it is not productive. It is safe, however, to force an open term if the clock κ does not appear free in the context Γ as in the rule

$$\frac{\Gamma \vdash_{\Delta,\kappa} t : \triangleright (\alpha : \kappa).A \qquad \Gamma \vdash_\Delta}{\Gamma \vdash_{\Delta,\kappa} t\,[\diamond] : A\,[\diamond/\alpha]} \tag{1}$$

In that case the context can be thought of as being κ-stable, i.e., not affected by ticks on clock κ. Rule (1) is not closed

$$\frac{\Gamma \vdash_\Delta t : A \qquad A \leftrightarrow^* B \qquad \Gamma \vdash_\Delta B \text{ type}}{\Gamma \vdash_\Delta t : B} \qquad \frac{\Gamma, x : A \vdash_\Delta t : B}{\Gamma \vdash_\Delta \lambda(x : A).t : \Pi(x : A).\, B} \qquad \frac{\Gamma \vdash_\Delta t : \Pi(x : A).\, B \qquad \Gamma \vdash_\Delta u : A}{\Gamma \vdash_\Delta t\, u : B\,[u/x]}$$

$$\frac{\Gamma \vdash_{\Delta,\kappa} t : A \qquad \Gamma \vdash_\Delta}{\Gamma \vdash_\Delta \Lambda\kappa.t : \forall\kappa.A} \qquad \frac{\Gamma \vdash_\Delta t : \forall\kappa.A \qquad \kappa' \in \Delta}{\Gamma \vdash_\Delta t[\kappa'] : A\,[\kappa'/\kappa]} \qquad \frac{\Gamma, \alpha : \kappa \vdash_\Delta t : A \qquad \kappa \in \Delta}{\Gamma \vdash_\Delta \lambda(\alpha : \kappa).t : \triangleright\,(\alpha : \kappa).A}$$

$$\frac{\Gamma \vdash_\Delta t : \triangleright\,(\alpha : \kappa).A \qquad \Gamma, \alpha' : \kappa, \Gamma' \vdash_\Delta}{\Gamma, \alpha' : \kappa, \Gamma' \vdash_\Delta t\,[\alpha'] : A\,[\alpha'/\alpha]} \qquad \frac{\Gamma \vdash_{\Delta,\kappa} t : \triangleright\,(\alpha : \kappa).A \qquad \Gamma \vdash_\Delta \qquad \kappa' \in \Delta}{\Gamma \vdash_\Delta (t\ [\kappa'/\kappa])\,[\diamond] : A\,[\kappa'/\kappa]\,[\diamond/\alpha]} \qquad \frac{\Gamma \vdash_\Delta t : \triangleright^\kappa A \to A}{\Gamma \vdash_\Delta \mathsf{dfix}^\kappa\, t : \triangleright^\kappa A}$$

$$\frac{\Gamma \vdash_\Delta F : \triangleright^\kappa(A \to \mathcal{U}) \to A \to \mathcal{U} \qquad \Gamma \vdash_\Delta u : A \qquad \Gamma \vdash_\Delta t : \mathsf{El}\,((\mathsf{dfix}^\kappa\, F)\,[\alpha]\,u) \qquad \Gamma \vdash_\Delta \alpha : \kappa}{\Gamma \vdash_\Delta \mathsf{unfold}_\alpha\, t : \mathsf{El}\,(F\,(\mathsf{dfix}^\kappa\, F)\,u)}$$

$$\frac{\Gamma \vdash_\Delta F : \triangleright^\kappa(A \to \mathcal{U}) \to A \to \mathcal{U} \qquad \Gamma \vdash_\Delta u : A \qquad \Gamma \vdash_\Delta t : \mathsf{El}\,(F\,(\mathsf{dfix}^\kappa\, F)\,u) \qquad \Gamma \vdash_\Delta \alpha : \kappa}{\Gamma \vdash_\Delta \mathsf{fold}_\alpha\, t : \mathsf{El}\,((\mathsf{dfix}^\kappa\, F)\,[\alpha]\,u)}$$

$$\frac{\Gamma, x : \mathsf{El}\,(A) \vdash_\Delta B : \mathcal{U}}{\Gamma \vdash_\Delta \hat{\Pi} x : A.\, B : \mathcal{U}} \qquad \frac{\Gamma, \alpha : \kappa \vdash_\Delta A : \mathcal{U} \qquad \kappa \in \Delta}{\Gamma \vdash_\Delta \hat{\triangleright}\,\alpha : \kappa.A : \mathcal{U}} \qquad \frac{\Gamma \vdash_{\Delta,\kappa} A : \mathcal{U} \qquad \Gamma \vdash_\Delta}{\Gamma \vdash_\Delta \hat{\forall}\kappa.A : \mathcal{U}} \qquad \frac{\Gamma, x : \mathsf{El}\,(A) \vdash_\Delta B : \mathcal{U}}{\Gamma \vdash_\Delta \hat{\Sigma} x : A.\, B : \mathcal{U}}$$

$$\frac{\Gamma \vdash_\Delta}{\Gamma \vdash_\Delta \hat{1} : \mathcal{U}} \qquad \frac{\Gamma \vdash_\Delta}{\Gamma \vdash_\Delta \hat{\mathsf{Bool}} : \mathcal{U}} \qquad \frac{\Gamma \vdash_\Delta}{\Gamma \vdash_\Delta \hat{\mathsf{Nat}} : \mathcal{U}}$$

Fig. 2. Selected typing rules of Clocked Type Theory. In the first rule $\leftrightarrow^*$ is the least equivalence relation containing the reduction relation.

under substitution (or even weakening) since introducing a variable with a type containing κ breaks the assumption of the rule. The rule for application to $\diamond$ in Figure 2 (third line, second rule) solves this problem by substituting away the clock κ in the conclusion. An easy induction argument shows that typing judgements are closed under weakening in clock variables, and renaming of clocks. Using this, one can show that rule (1) is admissible.

The type $\forall\kappa.A$ is a dependent product over clocks. An element of this type is a term of type A that can compute for an arbitrary number of ticks on clock κ. For example, the operator $\mathsf{force} : (\forall\kappa.\triangleright^\kappa A) \to \forall\kappa.A$ of Atkey and McBride [5] can be encoded as $\lambda x : (\forall\kappa.\triangleright^\kappa A).\Lambda\kappa.x[\kappa]\,[\diamond]$. If $t : \triangleright\,(\alpha : \kappa).A$ with α not free in A, then $t\,[\diamond]$ can be encoded using force, and application to $\diamond$ is also equivalent to prev as used in GDTT. We prefer the present formulation because it can be expressed without reference to quantification over clocks, and can thus be seen as an eliminator for $\triangleright^\kappa$.

A term $f : \triangleright^\kappa A \to A$ is a productive function taking suspended computations of type A and returning values of type A. The *delayed fixed point* $\mathsf{dfix}^\kappa f$ of f is an element of type $\triangleright^\kappa A$ which, when given a tick, applies f to itself.

Recursive types arise as fixed points of productive endofunctions on the universe. To treat dependent recursive types, the endofunctions are allowed to have a parameter type A, so the general type considered is $F : \triangleright^\kappa(A \to \mathcal{U}) \to A \to \mathcal{U}$. To ensure termination, fixed points do not unfold until applied to the tick constant $\diamond$, so although $\mathsf{El}\,(F(\mathsf{dfix}^\kappa\, F)\,u)$ and $\mathsf{El}\,(\mathsf{dfix}^\kappa F\,[\diamond]\,u)$ are equal, the types $\mathsf{El}\,(F(\mathsf{dfix}^\kappa\, F)\,u)$ and $\mathsf{El}\,(\mathsf{dfix}^\kappa F\,[\alpha]\,u)$ are not, if α is a tick variable. The terms fold_α and unfold_α allow us to program with this suspended type equality. This corresponds to the situation in Guarded Cubical Type Theory [12], where (under the translation given in Section V) the terms $\mathsf{dfix}^\kappa\, F$ and $\lambda(\alpha : \kappa).F(\mathsf{dfix}^\kappa\, F)$ are propositionally equal and the terms unfold_α and fold_α are the equivalence of types arising from this equality. Section IV shows how to use fold_α and unfold_α to program and reason about streams.

Unlike GDTT which has a family of universes indexed by clock contexts, for simplicity of presentation, here CloTT has a single universe. The indexing of universes is needed in GDTT for soundness of the *clock irrelevance rule* stating essentially that types $\forall\kappa.A$ and A are isomorphic if κ is not free in A. The clock irrelevance rule does not follow from the reduction semantics given here (see the discussion in Section V-C).

Remark II.1. We will sometimes assume a single clock constant κ_0. Since there are no special rules for κ_0 that do not apply to all clock variables, this corresponds to working in a clock context Δ containing κ_0.

III. Reduction semantics

Figure 3 describes the reduction semantics. The reduction semantics is given in full and also includes rules for the introduction and elimination forms of Bool (true, false, if) and Nat (0, suc , rec), which were omitted in the typing rules in Figure 4. As stated above, ticks are resources that can be spent to unfold fixed points. This can be seen from the reduction rule for dfix^κ. Tick variables should be understood as

$$(\lambda x : A.t)s \to t\,[s/x] \qquad (\Lambda\kappa.t)[\kappa'] \to t\,[\kappa'/\kappa] \qquad \mathsf{El}\left(\hat{\Pi} x : s.\, t\right) \to \Pi x : \mathsf{El}\,(s)\,.\,\mathsf{El}\,(t)$$

$$(\lambda(\alpha' : \kappa).t)\,[\alpha] \to t\,[\alpha/\alpha'] \qquad \lambda(\alpha : \kappa).(t\,[\alpha]) \to t \qquad \mathsf{El}\left(\hat{\Sigma} x : s.\, t\right) \to \Sigma x : \mathsf{El}\,(s)\,.\,\mathsf{El}\,(t)$$

$$(\Lambda\kappa.t[\kappa]) \to t \qquad \pi_i\,\langle t_1, t_2\rangle \to t_i \qquad \mathsf{El}\left(\hat{\mathsf{Nat}}\right) \to \mathsf{Nat}$$

$$\mathsf{fold}_\diamond t \to t \qquad \mathsf{unfold}_\diamond t \to t \qquad \mathsf{El}\left(\hat{1}\right) \to 1$$

$$\mathsf{if}\,\mathsf{true}\,t_1\,t_2 \to t_1 \qquad \mathsf{if}\,\mathsf{false}\,t_1\,t_2 \to t_2 \qquad \mathsf{El}\left(\hat{\mathsf{Bool}}\right) \to \mathsf{Bool}$$

$$\mathsf{rec}\,(\mathsf{suc}\ t_1)\,t_2\,t_3 \to t_3\,t_1\,(\mathsf{rec}\,t_1\,t_2\,t_3) \qquad \mathsf{rec}\,0\,t\,s \to t \qquad \mathsf{El}\left(\hat{\forall}\kappa.t\right) \to \forall\kappa.\mathsf{El}\,(t)$$

$$(\mathsf{dfix}^\kappa\,t)\,[\diamond] \to t\,(\mathsf{dfix}^\kappa\,t) \qquad \mathsf{El}\,(\hat{\triangleright}\,\alpha : \kappa.t) \to \triangleright\,(\alpha : \kappa).\mathsf{El}\,(t)$$

$$\frac{t \to u}{C[t] \to C[u]}$$

Fig. 3. Reduction relation $\to$ on terms. The η reductions for tick and clock abstraction are subject to the usual side condition that $\alpha \notin \mathsf{ft}(t)$ and $\kappa \notin \mathsf{fc}(t)$, respectively. The notation $C[-]$ ranges over all contexts.

placeholders for the actual tick $\diamond$, and thus do not themselves force unfoldings of fixed points. Similarly, the β and η reductions for recursive types requires an actual tick, in the sense that e.g. $\mathsf{unfold}_\alpha(\mathsf{fold}_\alpha\,t)$ does not reduce. Only when $\diamond$ is substituted for α do the reductions apply. More precisely, at this point the folded type $(\mathsf{El}\,((\mathsf{dfix}^\kappa\,F)\,[\diamond]\,u))$ and unfolded type $(\mathsf{El}\,(F(\mathsf{dfix}^\kappa\,F)\,u))$ are equal, and so the folds and unfolds simply disappear.

We include η reductions for tick and clock abstractions. The former is needed for the correctness of the translation of GDTT into CloTT. These η reductions are subject to the usual side condition that α is not in the set $\mathsf{ft}(t)$ of free tick variables of t, and κ is not in the set $\mathsf{fc}(t)$ of free clock variables of t. However, for well-typed terms, the side condition for η reduction for tick abstractions is always satisfied.

A. Metatheoretic results

We now state our main metatheoretic results. Section VI outlines the proofs, but full proofs are beyond the scope of this extended abstract. Referees can find the full proofs in the online appendix [25].

Proposition III.1 (Subject reduction)**.** *If* $\Gamma \vdash_\Delta s : A$ *and* $s \to t$*, then* $\Gamma \vdash_\Delta t : A$*.*

Theorem III.2 (Confluence)**.** If $s \to^* t_1, s \to^* t_2$, then $t_1 \to^* t$ and $t_2 \to^* t$ for some t.

Theorem III.3 (Strong normalisation)**.** If $\Gamma \vdash_\Delta t : A$, then t is strongly normalising. If $\Gamma \vdash_\Delta A$ type then A is strongly normalising.

As is standard, strong normalisation and confluence together yield a decision procedure for the equational theory $\leftrightarrow^*$ defined as the least equivalence relation containing $\to$: Given two terms, reduce them to their unique normal forms and compare them.

Corollary III.4. *The equational theory* $\leftrightarrow^*$ *is decidable.*

Finally, we state the canonicity theorem. Note that the closed terms we consider may have free clock variables. Consequently, canonicity is preserved when clock constants are added, which is needed for productivity of coinductive definitions (see Corollary IV.1).

Theorem III.5 (Canonicity)**.** If $\vdash_\Delta t : \mathsf{Nat}$, then $t \to^* \mathsf{suc}^{\,n}\,0$ for some $n \in \mathbb{N}$.

IV. Example: Programming with streams

We now show how to encode a type of streams in CloTT. The purpose of this section is to illustrate how to program in the type theory, rather than to argue for expressiveness. The latter is addressed in Section V.

Following Remark II.1, we will assume a clock constant κ_0. We use the standard type theoretic notation $A \times B$ for $\Sigma x : A.B$ and $A \to B$ for $\Pi x : A.B$ assuming x is not free in B.

Let $F : \triangleright^\kappa\,\mathcal{U} \to \mathcal{U}$ be $\lambda x : \triangleright^\kappa\,\mathcal{U}.\hat{\mathsf{Nat}} \mathbin{\hat{\times}} \hat{\triangleright}\,\alpha : \kappa.x\,[\alpha]$ and define

$$\mathsf{Str}^\kappa := \mathsf{El}\,(F(\mathsf{dfix}^\kappa F))$$
$$\mathsf{Str}^\kappa_\alpha := \mathsf{El}\,(\mathsf{dfix}^\kappa F\,[\alpha])$$

In a context containing $\alpha : \kappa$, this definition gives us the terms[1]

$$\mathsf{fold}_\alpha : \mathsf{Str}^\kappa \to \mathsf{Str}^\kappa_\alpha$$
$$\mathsf{unfold}_\alpha : \mathsf{Str}^\kappa_\alpha \to \mathsf{Str}^\kappa$$

Since $\mathsf{Str}^\kappa \to^* \mathsf{Nat} \times \triangleright\,(\alpha : \kappa).\mathsf{Str}^\kappa_\alpha$ we can define

$$\mathsf{cons}^\kappa : \mathsf{Nat} \to \triangleright^\kappa\,\mathsf{Str}^\kappa \to \mathsf{Str}^\kappa$$
$$\mathsf{cons}^\kappa := \lambda x : \mathsf{Nat}.\lambda y : \triangleright^\kappa\,\mathsf{Str}^\kappa.\,\langle x, \lambda(\alpha : \kappa).\mathsf{fold}_\alpha(y\,[\alpha])\rangle$$
$$\mathsf{hd}^\kappa : \mathsf{Str}^\kappa \to \mathsf{Nat}$$
$$\mathsf{hd}^\kappa := \lambda x : \mathsf{Str}^\kappa.\pi_1\,x$$
$$\mathsf{tl}^\kappa : \mathsf{Str}^\kappa \to \triangleright^\kappa\,\mathsf{Str}^\kappa$$
$$\mathsf{tl}^\kappa := \lambda x : \mathsf{Str}^\kappa.\lambda(\alpha : \kappa).\mathsf{unfold}_\alpha((\pi_2\,x)\,[\alpha])$$

We write $x ::^\kappa_\alpha xs$ for $\mathsf{cons}^\kappa\,x\,(\lambda(\alpha : \kappa).xs)$. Note that $x ::^\kappa_\alpha xs$ binds α in xs.

The type Str^κ is a guarded recursive type of streams, i.e., it is clock sensitive as can be seen e.g. by the type of tl^κ.

[1] Strictly speaking, F would need to have the type $\triangleright^\kappa(1 \to \mathcal{U}) \to 1 \to \mathcal{U}$, but for the sake of clarity we define it with type $\triangleright^\kappa\,\mathcal{U} \to \mathcal{U}$ instead.

One consequence of this is that all functions $\mathsf{Str}^\kappa \to \mathsf{Str}^\kappa$ are causal in the sense that the n'th output element depends only on the n first inputs [27]. Universal quantification over clocks is used to convert guarded recursive types to coinductive types, as explained by Atkey and McBride [5]. For example, we can define a type Str of coinductive streams as

$$\mathsf{Str} := \forall\kappa.\mathsf{Str}^\kappa$$

The functions cons^κ, tl^κ, and hd^κ can be lifted to coinductive streams using the fixed clock constant κ_0:

$$\begin{aligned}
&\mathsf{cons} : \mathsf{Nat} \to \mathsf{Str} \to \mathsf{Str}\\
&\mathsf{cons} := \lambda x : \mathsf{Nat}.\lambda y : \mathsf{Str}.\Lambda\kappa.\, x ::^\kappa_\alpha\ (y\,[\kappa])\\
&\quad \mathsf{hd} : \mathsf{Str} \to \mathsf{Nat}\\
&\quad \mathsf{hd} := \lambda x : \mathsf{Str}.\mathsf{hd}^{\kappa_0}\,(x\,[\kappa_0])\\
&\quad\ \mathsf{tl} : \mathsf{Str} \to \mathsf{Str}\\
&\quad\ \mathsf{tl} := \lambda x : \mathsf{Str}.\Lambda\kappa.(\mathsf{tl}^\kappa\,(x\,[\kappa]))\,[\diamond]
\end{aligned}$$

The function $\mathsf{nth} : \mathsf{Nat} \to \mathsf{Str} \to \mathsf{Nat}$ returning the n'th element of a stream can be defined by ordinary natural number recursion. As a corollary to canonicity we then get a statement of productivity for streams.

Corollary IV.1 (Productivity)**.** *If t is a closed term of type* Str *and n is a closed term of type* Nat *then* $\mathsf{nth}\,n\,t$ *reduces to a natural number.*

Note that the clock constant κ_0 is needed to define nth. Since κ_0 is added by working in a context where it is a free variable, it is important that the closed terms considered in Theorem III.5 can have free clock variables. On the other hand, for this result, the canonicity result does not need to allow other clock variables than κ_0 in closed terms.

A. Example programs

We now give a few examples of streams. We use the notation $\mathsf{fix}^\kappa x.t$ defined as $(\lambda x : \triangleright^\kappa A.t)(\mathsf{dfix}^\kappa(\lambda x : \triangleright^\kappa A.t))$. First the constant stream: if n is of type Nat define

$$\begin{aligned}
\mathsf{const}^\kappa\, n &:= \mathsf{fix}^\kappa x.n ::^\kappa_\alpha x\,[\alpha] &&: \mathsf{Str}^\kappa\\
\mathsf{const}\, n &:= \Lambda\kappa.\mathsf{const}^\kappa\, n &&: \mathsf{Str}
\end{aligned}$$

Map over streams can be defined as

$$\begin{aligned}
&\mathsf{map}^\kappa : (\mathsf{Nat} \to \mathsf{Nat}) \to \mathsf{Str}^\kappa \to \mathsf{Str}^\kappa\\
&\mathsf{map}^\kappa\, f := \mathsf{fix}^\kappa g.\lambda x : \mathsf{Str}^\kappa.f(\mathsf{hd}^\kappa x) ::^\kappa_\alpha\ g\,[\alpha](\mathsf{tl}^\kappa\, x\,[\alpha])\\
&\mathsf{map} : (\mathsf{Nat} \to \mathsf{Nat}) \to \mathsf{Str} \to \mathsf{Str}\\
&\mathsf{map}\, f\, x := \Lambda\kappa.\mathsf{map}^\kappa\, f(x\,[\kappa])
\end{aligned}$$

Using these we can define a stream of natural numbers

$$\begin{aligned}
\mathsf{nats}^\kappa &:= \mathsf{fix}^\kappa x.0 ::^\kappa_\alpha \mathsf{map}^\kappa\mathsf{suc}\,(x\,[\alpha]) &&: \mathsf{Str}^\kappa\\
\mathsf{nats} &:= \Lambda\kappa.\mathsf{nats}^\kappa &&: \mathsf{Str}\,.
\end{aligned}$$

It is easy to check that $\mathsf{hd}(\mathsf{tl}(\mathsf{nats}))$ reduces to $\mathsf{suc}\,0$.

As a simple example of a non-causal function on streams, we define the function eo that removes every other element of the input stream.

$$\begin{aligned}
&\mathsf{eo}^\kappa : \mathsf{Str} \to \mathsf{Str}^\kappa\\
&\mathsf{eo}^\kappa := \mathsf{fix}^\kappa f.\lambda x : \mathsf{Str}.(\mathsf{hd}\, x) ::^\kappa_\alpha\ (f\,[\alpha]\,(\mathsf{tl}\,(\mathsf{tl}\, x)))\\
&\ \mathsf{eo} : \mathsf{Str} \to \mathsf{Str}\\
&\ \mathsf{eo} := \lambda x : \mathsf{Str}.\Lambda\kappa.\mathsf{eo}^\kappa\, x
\end{aligned}$$

B. Reasoning about streams

We now consider an example of reasoning about streams. Since CloTT does not have identity types the example is very basic. We show how to lift a predicate $P : \mathbb{N} \to \mathcal{U}$ to a pointwise predicate $\mathsf{L}P$ on coinductive streams, and how to lift a proof of $\Pi(x : \mathbb{N}).P(x)$ to a proof of $\Pi(x : \mathsf{Str}).\mathsf{L}P(x)$.

Define $G : \triangleright^\kappa(\mathsf{Str}^\kappa \to \mathcal{U}) \to (\mathsf{Str}^\kappa \to \mathcal{U})$ as

$$G\,X\,x := P(\mathsf{hd}^\kappa x) \mathbin{\hat{\times}} \hat{\triangleright}\,\alpha : \kappa.X\,[\alpha](\mathsf{tl}^\kappa x\,[\alpha])$$

and define, for $x : \mathsf{Str}^\kappa$,

$$\begin{aligned}
\mathsf{L}^\kappa P\,(x) &:= \mathsf{El}\,(G(\mathsf{dfix}^\kappa G)\,x)\\
\mathsf{L}^\kappa_\alpha P\,(x) &:= \mathsf{El}\,(\mathsf{dfix}^\kappa G\,[\alpha]\,x)\ .
\end{aligned}$$

Note that $\mathsf{L}^\kappa P\,x \to^* \mathsf{El}\,(P(\mathsf{hd}^\kappa x)) \times \triangleright\,(\alpha : \kappa).\mathsf{L}^\kappa_\alpha P(\mathsf{tl}^\kappa x\,[\alpha])$. An element of $\mathsf{L}^\kappa P\,x$ is essentially a pair of a proof of $P(\mathsf{hd}^\kappa x)$ and a delayed proof of $\mathsf{L}^\kappa P$ applied to the tail of x. More precisely, if $p : \mathsf{El}\,(P(\mathsf{hd}^\kappa x))$, $q : (\triangleright\,(\alpha : \kappa).\mathsf{L}^\kappa P(\mathsf{tl}^\kappa x\,[\alpha]))$ and $r : \mathsf{L}^\kappa P(\mathsf{tl}^\kappa x\,[\alpha])$ define

$$\begin{aligned}
\mathsf{cons}^\kappa\, p\, q &:= \langle p, \lambda(\alpha : \kappa).\mathsf{fold}_\alpha(q\,[\alpha])\rangle &&: \mathsf{L}^\kappa P(x)\\
p ::^\kappa_\alpha r &:= \mathsf{cons}^\kappa\, p\,(\lambda(\alpha : \kappa).r) &&: \mathsf{L}^\kappa P(x)
\end{aligned}$$

Suppose now that we are given p of type $\Pi x : \mathsf{Nat}.\mathsf{El}\,(P(x))$. Then p lifts to a proof q of $\Pi x : \mathsf{Str}^\kappa.\mathsf{L}^\kappa P(x)$ defined as

$$q := \mathsf{fix}^\kappa r.\lambda x : \mathsf{Str}^\kappa.p(\mathsf{hd}^\kappa x) ::^\kappa_\alpha\ (r\,[\alpha](\mathsf{tl}^\kappa x\,[\alpha])) \qquad (2)$$

All this lifts to coinductive streams. Define, for $x : \mathsf{Str}$ the type $\mathsf{L}P(x)$ as $\forall\kappa.\mathsf{L}^\kappa P(x[\kappa])$, and define the lifting of a proof p of $\Pi x : \mathsf{Nat}.\mathsf{El}\,(P(x))$ to be

$$\lambda x : \mathsf{Str}.\Lambda\kappa.q(x[\kappa])$$

where q is as in (2).

V. Translation from GDTT

In this section we show how the fragment of GDTT [4] obtained by removing identity types can be translated to CloTT, thus showing that CloTT is at least as expressive as this fragment of GDTT. Recalling the applications of GDTT listed in the introduction, this shows a range of application of CloTT. The translation preserves most of the equalities of GDTT, but some equalities are not preserved, and we believe these should be considered propositional equalities (see the discussion in Section V-C).

Delayed substitutions:

$$\frac{\Delta \vdash_\Delta \Gamma \qquad \vdash_\Delta \kappa}{\vdash_\Delta \cdot : \Gamma \overset{\kappa}{\to} \cdot} \qquad \frac{\vdash_\Delta \xi : \Gamma \overset{\kappa}{\to} \Gamma' \qquad \Gamma \vdash_\Delta t : \triangleright^\kappa \xi.A}{\vdash_\Delta \xi\,[x \leftarrow t] : \Gamma \overset{\kappa}{\to} \Gamma', x : A}$$

Typing rules:

$$\frac{\Gamma, \Gamma' \vdash_\Delta A \text{ type} \qquad \vdash_\Delta \xi : \Gamma \overset{\kappa}{\to} \Gamma'}{\Gamma \vdash_\Delta \triangleright^\kappa \xi.A \text{ type}} \qquad \frac{\Delta' \subseteq \Delta}{\Gamma \vdash_\Delta \mathcal{U}_{\Delta'} \text{ type}}$$

$$\frac{\vdash_{\Delta'} \kappa \qquad \Gamma \vdash_\Delta A : \triangleright^\kappa \mathcal{U}_{\Delta'}}{\Gamma \vdash_\Delta \widehat{\triangleright}^\kappa A : \mathcal{U}_{\Delta'}}$$

$$\frac{\Gamma, \Gamma' \vdash_\Delta t : A \qquad \vdash_\Delta \xi : \Gamma \overset{\kappa}{\to} \Gamma'}{\Gamma \vdash_\Delta \mathsf{next}^\kappa \xi.t : \triangleright^\kappa \xi.A}$$

$$\frac{\vdash_\Delta \kappa \qquad \Gamma, x : \triangleright^\kappa A \vdash_\Delta t : A}{\Gamma \vdash_\Delta \mathsf{fix}^\kappa x.t : A}$$

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

Advancing delayed substitutions:

$$\frac{\vdash_{\Delta,\kappa} \cdot : \Gamma \overset{\kappa}{\to} \cdot \qquad \Delta \vdash_\Delta \Gamma}{\mathrm{adv}^\kappa_\Delta(\cdot) := \mathsf{id} : \Gamma \to \Gamma}$$

$$\frac{\vdash_{\Delta,\kappa} \xi[x \leftarrow t] : \Gamma \overset{\kappa}{\to} \Gamma', x : A \qquad \Delta \vdash_\Delta \Gamma}{\mathrm{adv}^\kappa_\Delta(\xi[x \leftarrow t]) := \mathrm{adv}^\kappa_\Delta(\xi)[(\mathsf{prev}\kappa.t)[\kappa]\,/x]}$$

Fig. 4. GDTT typing rules and rules for delayed substitutions. The judgement $\vdash_\Delta \kappa$ means $\kappa \in \Delta$ or $\kappa = \kappa_0$ the clock constant.

A. Guarded dependent type theory

Like CloTT, GDTT is a dependent type theory with a special context of clocks subscript to the turnstile in judgements. It has guarded recursive fixed points, and a type operator $\triangleright^\kappa$ with introduction form $\mathsf{next}^\kappa t : \triangleright^\kappa A$ if $t : A$, but there are no ticks on clocks. Reasoning about delayed data is done using *delayed substitutions*, a restricted kind of elimination form for $\triangleright^\kappa$ that we now explain. If $t : A$ in context $x : B$ and $u : \triangleright^\kappa B$ then $\mathsf{next}^\kappa\,[x \leftarrow u]\,.t$ has type $\triangleright^\kappa\,[x \leftarrow u]\,.A$. Intuitively, the delayed substitution $[x \leftarrow u]$ means 'reduce u to $\mathsf{next}^\kappa s$, then substitute s for x in t and A', but since u can be open, it may not be possible to reduce it to the form $\mathsf{next}^\kappa s$, and thus delayed substitutions are necessary for reasoning about data of type $\triangleright^\kappa B$. General delayed substitutions substitute more than one variable, and there can be dependencies in the types of these.

Figure 4 recalls the typing rules for delayed substitutions and the typing rules for the fragment of GDTT relating to $\triangleright^\kappa$. Figure 5 recalls the equality theory. Constructions, such as dependent products and sums, and quantification over clocks that have direct counterparts in CloTT are therefore omitted.

In GDTT universes are indexed by sets of clocks. This is

Definitional type equalities:

$$\triangleright^\kappa \xi\,[x \leftarrow t]\,.A \equiv \triangleright^\kappa \xi.A \tag{3}$$
$$\triangleright^\kappa \xi\,[x \leftarrow t, y \leftarrow u]\,\xi'.A \equiv \triangleright^\kappa \xi\,[y \leftarrow u, x \leftarrow t]\,\xi'.A \tag{4}$$
$$\triangleright^\kappa \xi\,[x \leftarrow \mathsf{next}^\kappa \xi.t]\,.A \equiv \triangleright^\kappa \xi.A\,[t/x] \tag{5}$$
$$\mathsf{El}\left(\widehat{\triangleright}^\kappa\,(\mathsf{next}^\kappa \xi.t)\right) \equiv \triangleright^\kappa \xi.\mathsf{El}\,(t) \tag{6}$$

Definitional term equalities:

$$\mathsf{next}^\kappa \xi\,[x \leftarrow t]\,.u \equiv \mathsf{next}^\kappa \xi.u \tag{7}$$
$$\mathsf{next}^\kappa \xi\,[x \leftarrow t, y \leftarrow u]\,\xi'.v \equiv \mathsf{next}^\kappa \xi\,[y \leftarrow u, x \leftarrow t]\,\xi'.v \tag{8}$$
$$\mathsf{next}^\kappa \xi\,[x \leftarrow \mathsf{next}^\kappa \xi.t]\,.u \equiv \mathsf{next}^\kappa \xi.u\,[t/x] \tag{9}$$
$$\mathsf{next}^\kappa \xi\,[x \leftarrow t]\,.x \equiv t \tag{10}$$
$$\mathsf{prev}\kappa.\mathsf{next}^\kappa \xi.t \equiv \Lambda\kappa.t(\mathrm{adv}^\kappa_\Delta(\xi)) \tag{11}$$
$$\mathsf{next}^\kappa((\mathsf{prev}\kappa.t)[\kappa]) \equiv t \tag{12}$$
$$\mathsf{next}^\kappa \xi.\mathsf{next}^\kappa \xi'.u \equiv \mathsf{next}^\kappa \xi'.\mathsf{next}^\kappa \xi.u \tag{13}$$
$$\mathsf{fix}^\kappa x.t \equiv t\,[\mathsf{next}^\kappa(\mathsf{fix}^\kappa x.t)/x] \tag{14}$$
$$\frac{t : \forall\kappa.A \qquad \kappa \notin \mathsf{fc}(A)}{t[\kappa'] \equiv t[\kappa'']} \tag{15}$$

Fig. 5. Type and term equalities of GDTT. All rules should be read as equalities in a context, and have the implicit assumption that both sides are wellformed and welltyped in that context. For example, rules (3) and (7) require that A and u are well-formed in a context without x. Rule (13) moreover assumes that none of the variables in the codomains of ξ and ξ' appear in the type of u.

necessary for soundness of rule (15) of Figure 5. A delayed substitution $\vdash_\Delta \xi : \Gamma \overset{\kappa}{\to} \Gamma'$ can be *advanced* to yield an ordinary substitution $\mathrm{adv}^\kappa_\Delta(\xi)$ from Γ to Γ, Γ'. In the rule for $\mathsf{prev}\kappa.t$ as well as in equality (11) this substitution is performed on type A and term t respectively.

B. Translation

The translation from GDTT to CloTT is presented in Figure 6. Again we focus on the non-trivial fragment, but the translation can be extended to the entire fragment of GDTT not mentioning identity types in a trivial way. The figure defines a translation of types and terms simultaneously with a translation of delayed substitutions to ordinary substitutions of dependent contexts.

Proposition V.1. *The translation preserves wellformed judgements in the following sense.*

1) *If* $\Gamma \vdash_\Delta A$ type *is a wellformed type judgement in* GDTT*, then* $\Gamma^* \vdash_{\Delta,\kappa_0} A^*$ type *is wellformed in* CloTT.
2) *If* $\Gamma \vdash_\Delta t : A$ *is a wellformed* GDTT *typing judgement, then* $\Gamma^* \vdash_{\Delta,\kappa_0} t^* : A^*$ *is wellformed in* CloTT.
3) *If* $\vdash_\Delta \xi : \Gamma \overset{\kappa}{\to} \Gamma'$ *is a delayed substitution in* GDTT*, then* ξ^*_α *is a substitution from* $\Gamma^*, \alpha : \kappa \vdash_{\Delta,\kappa_0}$ *to* $\Gamma^*, \alpha : \kappa, (\Gamma')^* \vdash_{\Delta,\kappa_0}$ *in* CloTT.

Because of the special status of tick variables, the notion of substitutions between CloTT contexts is non-standard. For reasons of space, we omit the details.

Translation of contexts:

$$(\cdot)^* := \cdot \qquad (\Gamma, x : A)^* := \Gamma^*, x : A^*$$

Translation of delayed substitutions

$$(\cdot)^*_\alpha := \mathsf{id}_{(\Gamma,\alpha:\kappa\vdash_\Delta)} \qquad (\xi\,[x \leftarrow t])^*_\alpha := \xi^*_\alpha[x \mapsto t^*\,[\alpha]]$$

Translation of types:

$$\mathcal{U}^*_{\Delta'} := \mathcal{U} \qquad \mathsf{El}\,(t)^* := \mathsf{El}\,(t^*)$$
$$(\triangleright^\kappa \xi.A)^* := \triangleright\,(\alpha : \kappa).A^*\,\xi^*_\alpha$$

Translation of terms:

$$(\widehat{\triangleright}^\kappa A)^* := \hat{\triangleright}\,\alpha : \kappa.A^*\,[\alpha]$$
$$(\mathsf{next}^\kappa \xi.t)^* := \lambda(\alpha : \kappa).t^*\xi^*_\alpha$$
$$(\mathsf{prev}\kappa.t)^* := \Lambda\kappa.t^*\,[\diamond]$$
$$(\mathsf{fix}^\kappa x.t)^* := t^*\,[\mathsf{dfix}^\kappa(\lambda x.t^*)/x]$$

Fig. 6. Translation from GDTT to CloTT. The notation $A^*\,\xi^*_\alpha$ means perform the substitution ξ^*_α in type A^*. Likewise for $t^*\xi^*_\alpha$.

We say that the translation preserves a set of rules X, if whenever $t \equiv u$ can be proved using the rules in X, also $t^* \leftrightarrow^* u^*$ and, similarly, $A \equiv B$ implies $A^* \leftrightarrow^* B^*$.

Theorem V.2. The translation from GDTT to CloTT preserves all rules of Figure 5 except (12), (13), (14) and (15).

C. Discussion: The remaining equational rules

As noted above, our translation fails to preserve four GDTT equality rules. In order to maintain the expressivity of GDTT, these must be added to CloTT as propositional equalities. We now discuss how this can be achieved without breaking canonicity in a future version of CloTT with path types in the sense of Cubical Type Theory [13], but leave the details for future work.

First consider the fixed point equality (14). This equality is not preserved by the translation by design, since it requires unfolding of fixed points, which leads to divergence. Rather than being unfolded automatically, fixed point unfolding should be a path that the user should explicitly apply. This is the approach used in Guarded Cubical Type Theory (GCTT) [12], which shows how these paths can be used for programming with relatively little overhead.

The fixed point unfolding path of GCTT can be translated into CloTT if a constant

$$\mathsf{pfix}^\kappa f : \triangleright\,(\alpha : \kappa).\mathsf{Path}_A((\mathsf{dfix}^\kappa f)\,[\alpha], f(\mathsf{dfix}^\kappa f))$$

(for $f : \triangleright^\kappa A \to A$) is added to an extension of CloTT with path types. For canonicity it is not necessary that $\mathsf{pfix}^\kappa\, f\,[\alpha]$ reduces (since it is an open term), but $(\mathsf{pfix}^\kappa f)\,[\diamond]$ must. This should reduce to the reflexivity path, which is welltyped, since $(\mathsf{dfix}^\kappa f)\,[\diamond] \to f(\mathsf{dfix}^\kappa f)$.

Since $\mathsf{next}^\kappa((\mathsf{prev}\kappa.t)[\kappa])^*$ reduces to $\lambda(\alpha : \kappa).t^*\,[\diamond]$ and $\lambda(\alpha : \kappa).t^*\,[\alpha] \to t^*$, rule (12) boils down to an equality between terms of the form $u\,[\alpha]$ and $u\,[\diamond]$. Likewise (13) compiles to the equality

$$\lambda(\alpha : \kappa).\lambda(\alpha' : \kappa).t^*\,\xi^*_\alpha\,\xi'^*_{\alpha'} \equiv \lambda(\alpha : \kappa).\lambda(\alpha' : \kappa).t^*\,\xi^*_{\alpha'}\,\xi'^*_\alpha$$

which can be proved if $\xi^*_{\alpha'}$ equal to ξ^*_α and $\xi'^*_{\alpha'}$ equal to ξ'^*_α can be proved, i.e., if we can prove equalities between terms of the form $u\,[\alpha]$ and $u\,[\alpha']$. Since clocks are not types, we cannot postulate a path from α to α', or from α to $\diamond$, but it does make sense to postulate a tick irrelevance axiom:

$$\mathsf{tirr}^\kappa t : \triangleright\,(\alpha : \kappa).\triangleright\,(\alpha' : \kappa).\mathsf{Path}_A(t\,[\alpha], t\,[\alpha'])$$

whenever $t : \triangleright^\kappa A$. To maintain canonicity $(\mathsf{tirr}^\kappa t)\,[\diamond]\,[\diamond]$ must reduce to reflexivity.

Similarly, the *clock irrelevance rule* (5) can be added as an axiom

$$\mathsf{cirr}^\kappa t : \forall\kappa'.\forall\kappa''.\mathsf{Path}_A(t[\kappa'], t[\kappa''])$$

for $t : \forall\kappa.A$ and $\kappa \notin \mathsf{fc}(A)$, with the reduction rule that $\mathsf{cirr}^\kappa t\,[\kappa_0]\,[\kappa_0]$ reduces to reflexivity. This should suffice to prove canonicity in the special case that the clock context Δ consists only of the clock constant κ_0, which in turn suffices for productivity results such as Corollary IV.1.

VI. Proofs

In this section we briefly outline the proofs of the main metatheoretic results that we list in Section III-A.

A. Confluence, Subject Reduction, and Canonicity

The proof of confluence (Theorem III.2) is straightforward and follows the proof technique of Takahashi [28]: We define a parallel reduction relation $\twoheadrightarrow$ satisfying $\to \subseteq \twoheadrightarrow \subseteq \to^*$, and for each term t we define the term t^* that is obtained from t by simultaneously contracting all redexes in t. Confluence follows from the property that $s \twoheadrightarrow t$ implies $t \twoheadrightarrow s^*$, which can be proved by induction on s.

Also our subject reduction and canonicity results (Proposition V.1 and Theorem III.5) are proved by conventional methods: Subject reduction follows by an induction on the typing derivation using the fact that typing is preserved by clock, term, and tick substitution. One non-standard element of the subject reduction proof is the case for application to $\diamond$, because a clock variable is substituted in the term in the conclusion of the corresponding typing rule. To conclude subject reduction from the induction hypothesis for this typing rule, we need to use the following lemma:

Lemma VI.1. *If $s\,\sigma \to t$ for some clock substitution σ, then there is a term t' with $s \to t'$ and $t'\,\sigma = t$.*

For canonicity, we show by induction that any term $\vdash_\Delta t : A$ that is in normal form must be an introduction form, i.e. of the form $\lambda x : A.s$, $\mathsf{suc}\ s$, $\langle u, v\rangle$, etc. Then canonicity follows from strong normalisation and subject reduction.

B. Strong Normalisation

Our proof of strong normalisation follows Tait's proof technique [29]: We interpret each type A as a set $[\![A]\!]$ of strongly normalising terms and then show that each term of type A is in $[\![A]\!]$. More precisely, the interpretation of types is given in a Kripke-style [30], that is, $[\![A]\!]$ is indexed by objects in a category $\mathcal{K}$. The objects in $\mathcal{K}$ are pairs (Δ, δ) consisting of a clock context Δ and a mapping $\delta\colon \Delta \to \mathbb{N}$ and we write $[\![\vdash_\Delta A]\!]_\delta$ instead of $[\![A]\!]_{(\Delta,\delta)}$. A morphism $\sigma\colon (\Delta,\delta) \to (\Delta',\delta')$ is a mapping $\sigma\colon \Delta \to \Delta'$ (thought of as a clock substitution) such that $\delta'(\sigma(\kappa)) \leq \delta(\kappa)$ for all $\kappa \in \Delta$. Intuitively, $\delta(\kappa)$ is the time that is left on clock κ. Time passes when a term is applied to $\diamond$. Note that the clock substitutions can synchronise clocks by mapping them to the same clock, and setting their time to the minimum of the times of the synchronised clocks.

Since, in a dependent type theory, types depend on terms, $[\![\vdash_\Delta A]\!]_\delta$ cannot simply be defined inductively on the structure of A. The set of terms A for which $[\![\vdash_\Delta A]\!]_\delta$ is defined must be defined simultaneously with their interpretations $[\![\vdash_\Delta A]\!]_\delta$ in an essentially inductive-recursive definition. To this end, we follow the approach of Harper [31] and define the interpretation function $[\![\vdash_\Delta \cdot]\!]_\delta$ as the least fixed point of a monotone function.

The fixed point construction requires a complete pointed partial order structure. Recall that a complete pointed partial order (CPPO) is a partially ordered set $(S, \leq)$ with a least element such that every directed subset D of S (i.e. every pair $x, y \in D$ has an upper bound in D) has a least upper bound. The following fixed point theorem, which appears in Harper [31], will allow us to construct $[\![\vdash_\Delta A]\!]_\delta$:

Theorem VI.2. Any monotone function on a CPPO has a least fixed point.

Specifically, the least fixed point can be constructed as follows: Given a monotone function $f : X \to X$ on a CPPO $(X, \leq)$, we construct the following transfinite sequence (x_α) starting with the least element $\bot$:

$$x_0 = \bot \qquad x_{\alpha+1} = f(x_\alpha)$$
$$x_\gamma = \bigsqcup_{\alpha<\gamma} x_\alpha \qquad \text{if } \gamma \text{ is a limit ordinal}$$

Then there is an ordinal α such that x_α is the least fixed point.

We shall use the above fixed point construction to simultaneously define (1) partial mappings $[\![\vdash_\Delta \cdot]\!]_\delta$ from terms to sets of terms, and (2) the domain of $[\![\vdash_\Delta \cdot]\!]_\delta$. To this end, we use the notation $\mathsf{Terms}(\Delta)$ to denote the set of terms t with $\mathsf{fc}(t) \subseteq \Delta$. Since the interpretation of types is indexed by clock contexts Δ and mappings $\delta\colon \Delta \to \mathbb{N}$, we consider families of partial mappings $[\![\vdash_\Delta \cdot]\!]_\delta$ and their corresponding domains. That is, our CPPO consists of pairs $(\mathcal{D}, \phi)$, where $\mathcal{D} = (\mathcal{D}_{\Delta,\delta})$ is a family of sets $\mathcal{D}_{\Delta,\delta} \subseteq \mathsf{Terms}(\Delta)$ and $\phi = (\phi_{\Delta,\delta})$ is a family of partial maps $\phi_{\Delta,\delta}\colon \mathsf{Terms}(\Delta) \rightharpoonup \mathcal{P}(\mathsf{Terms}(\Delta))$. These pairs $(\mathcal{D}, \phi)$ are subject to a number of saturation properties, which are listed in Figure 7. We call a pair $(\mathcal{D}, \phi)$ that satisfies these properties a *saturated family*, and write Sat to denote the set of all saturated families. Before we review the saturation properties in detail, we need to introduce some notation.

(S1) $\mathcal{D}_{\Delta,\delta} = \mathsf{dom}(\phi_{\Delta,\delta})$.
(S2) If $A \to B$, then $A \in \mathcal{D}_{\Delta,\delta}$ iff $B \in \mathcal{D}_{\Delta,\delta}$ and $A \in \mathsf{SN}(\Delta)$.
(S3) If $A \to B$ and $A, B \in \mathcal{D}_{\Delta,\delta}$, then $\phi_{\Delta,\delta}(A) = \phi_{\Delta,\delta}(B)$.
(S4) If $t \in \phi_{\Delta,\delta}(A)$, then $t \in \mathsf{SN}$.
(S5) If $t \in \phi_{\Delta,\delta}(A)$, $\sigma\colon (\Delta,\delta) \to (\Delta',\delta')$, then $t\,\sigma \in \phi_{\Delta',\delta'}(A\,\sigma)$.
(S6) If $t \in \phi_{\Delta,\delta}(A)$, $s \in \mathsf{Terms}(\Delta)$ and $s \to_{\mathsf{WH}} t$, then $s \in \phi_{\Delta,\delta}(A)$.
(S7) If $t \in \mathsf{Neu}(\Delta)$ and $A \in \mathcal{D}_{\Delta,\delta}$, then $t \in \phi_{\Delta,\delta}(A)$.

Fig. 7. Saturation Properties.

$$\begin{aligned} E ::= {} & [\,] \mid E\,t \mid E\,[\alpha] \mid E\,[\kappa] \mid \pi_i\,E \\ & \mid \mathsf{if}\,E\,t_1\,t_2 \mid \mathsf{rec}\,E\,t_1\,t_2 \mid \mathsf{El}\,(E) \end{aligned}$$

where α ranges over $\mathsf{TV} \cup \{\diamond\}$.

$$\begin{aligned} (\lambda x.s)t &\mapsto s\,[t/x] && \text{if } t \in \mathsf{SN} \\ (\lambda(\alpha:\kappa).t)\,[\alpha'] &\mapsto t\,[\alpha'/\alpha] && \text{if } \alpha' \in \mathsf{TV} \cup \{\diamond\} \\ (\mathsf{dfix}^\kappa\,t)\,[\diamond] &\mapsto t\,(\mathsf{dfix}^\kappa\,t) \\ (\Lambda\kappa.t)[\kappa'] &\mapsto t\ [\kappa'/\kappa] \\ \mathsf{fold}_\diamond t &\mapsto t \\ \mathsf{unfold}_\diamond t &\mapsto t \\ \mathsf{if}\,\mathsf{true}\,t_1\,t_2 &\mapsto t_1 && \text{if } t_2 \in \mathsf{SN} \\ \mathsf{if}\,\mathsf{false}\,t_1\,t_2 &\mapsto t_2 && \text{if } t_1 \in \mathsf{SN} \\ \pi_i\,\langle t_1, t_2\rangle &\mapsto t_i && \text{if } t_{3-i} \in \mathsf{SN} \\ \mathsf{rec}\,0\,s\,t &\mapsto s && \text{if } t \in \mathsf{SN} \\ \mathsf{rec}\,(\mathsf{suc}\ t)\,v\,u &\mapsto u\,t\,(\mathsf{rec}\,t\,v\,u) \end{aligned}$$

Fig. 8. Evaluation contexts and weak head reduction.

We write SN for the set of all terms that are strongly normalising and $\mathsf{SN}(\Delta)$ for the set $\mathsf{SN} \cap \mathsf{Terms}(\Delta)$ of strongly normalising terms with free clock variables in Δ. Furthermore, we write TV for the set of tick variables. In addition we also need to define weak head reduction and neutral terms.

Definition VI.3 (weak head reduction)**.** The *weak head reduction* relation $\to_{\mathsf{WH}}$ is defined as follows: $s \to_{\mathsf{WH}} t$ iff $s = E[s']$, $t = E[t']$, and $s' \mapsto t'$, where the evaluation contexts E and the relation $\mapsto$ are defined in Figure 8. An evaluation context E is called SN if every term occurring in E is in SN. That is, E is obtained from the grammar in Figure 8, where the form $E\,t$ is subject to the restriction that $t \in \mathsf{SN}$, and the forms $\mathsf{if}\,E\,t_1\,t_2$ and $\mathsf{rec}\,E\,t_1\,t_2$ are subject to the restriction $t_1, t_2 \in \mathsf{SN}$. A term is called *neutral* if it is of the form $E[x]$, $E[\mathsf{unfold}_\alpha\,t]$, or $E[(\mathsf{dfix}^\kappa\,t)\,[\alpha]]$, where $\alpha \in \mathsf{TV}$, E is SN, and $t \in \mathsf{SN}$. We write $\mathsf{Neu}(\Delta)$ for the set of neutral terms in $\mathsf{Terms}(\Delta)$.

The notions of evaluation contexts and weak head reduction

are standard: Weak head reduction contracts the β-redex in a head position. Our notion of neutral terms, on the other hand, is non-standard since we have to deal with free tick variables: As usual neutral terms are terms that are "stuck" because of a variable in head position. But in addition to ordinary term variables, we also have tick variables. Therefore, also $E[\mathsf{unfold}_\alpha\, t]$ and $E[(\mathsf{dfix}^\kappa\, t)\,[\alpha]]$ are neutral, since these terms are "stuck" due to the occurrence of a tick variable α. However, $E[t\,[\alpha]]$ is in general not neutral (only if $t = \mathsf{dfix}^\kappa\, u$), since $t\,[\alpha]$ can in fact be a β-redex, namely in case $t = \lambda(\alpha : \kappa).u$.

The side conditions in the definition of evaluation contexts and weak head reduction that require terms to be SN are included to ensure that all neutral terms are SN, and that the weak expansion of an SN term is itself SN. In addition both neutral terms and weak head reductions are closed under clock substitutions. For this to hold, it is crucial that SN itself is closed under clock substitution, which follows from Lemma VI.1. These two properties of weak head reduction and neutral terms are essential for the consistency of the saturation properties.

We now return to the saturation properties in Figure 7: (S1) states that $\mathcal{D}_{\Delta,\delta}$ is the domain of $\phi_{\Delta,\delta}$; (S2) and (S3) state that $\mathcal{D}_{\Delta,\delta}$ and $\phi_{\Delta,\delta}$ are closed under reduction (and expansion to an SN term); (S5) and (S6) state that $\phi_{\Delta,\delta}$ is closed under clock substitution and weak head expansion; and (S7) states that $\phi_{\Delta,\delta}$ includes at least all neutral terms. Finally, (S4) is the property we are actually interested in, namely that all terms in $\phi_{\Delta,\delta}(A)$ are SN.

We define a partial order $\leq$ on the set Sat of saturated families as follows:

$$(\mathcal{D}, \phi) \leq (\mathcal{D}', \phi') \quad \text{iff} \quad \begin{array}{l} \mathcal{D}_{\Delta,\delta} \subseteq \mathcal{D}'_{\Delta,\delta} \text{ and } \phi_{\Delta,\delta} \subseteq \phi'_{\Delta,\delta} \\ \text{for all objects } (\Delta, \delta) \text{ in } \mathcal{K} \end{array}$$

where $\phi_{\Delta,\delta} \subseteq \phi'_{\Delta,\delta}$ denotes graph inclusion, i.e. $\phi_{\Delta,\delta}(A) = \phi'_{\Delta,\delta}(A)$ for all $A \in \mathsf{dom}\,(\phi_{\Delta,\delta})$. One can easily check that $(\mathsf{Sat}, \leq)$ indeed forms a CPPO, with the least element $(\mathcal{D}, \phi)$, where $\mathcal{D}_{\Delta,\delta} = \emptyset$ and $\phi_{\Delta,\delta} = \emptyset$ for all Δ and $\delta\colon \Delta \to \mathbb{N}$.

The definition of the interpretation of types is stratified into two levels: First, we define the interpretation of codes, i.e. terms that inhabit the universe $\mathcal{U}$. To this end we define a monotone function T^0 on Sat. The fixed point of T^0 will give us an interpretation of $\mathcal{U}$. Then, in the second step, we define the interpretation of types using the interpretation of $\mathcal{U}$ that we have obtained as the fixed point of T^0. This second stage is given by a monotone function T^1 on Sat. The fixed point of T^1 will serve as the definition of $[\![\vdash_\Delta \cdot]\!]_\delta$.

In order to ensure that the definition respects the reduction relation in the sense of the saturation properties (S2) and (S3), both $\mathcal{D}$ and ϕ are defined in terms of normal forms. To this end, we write $A \to^*_{\mathsf{nf}} B$ to denote that B is a normal form of A, i.e. $A \to^* B$ and there is no reduction $B \to C$ for any C.

Definition VI.4. Let $T^0\colon \mathsf{Sat} \to \mathsf{Sat}$ be defined by $T^0(\mathcal{D}, \phi) = (\overline{\mathcal{D}'}, \overline{\phi'})$, where

$$\overline{\mathcal{D}'}_{\Delta,\delta} = \left\{ A \in \mathsf{SN}(\Delta) \;\middle|\; \exists B \in \mathcal{D}'_{\Delta,\delta}. A \to^*_{\mathsf{nf}} B \right\}$$
$$\overline{\phi'}_{\Delta,\delta}(A) = \phi'_{\Delta,\delta}(B), \text{ if } A \in \mathsf{SN}(\Delta) \text{ and } A \to^*_{\mathsf{nf}} B$$

and $\mathcal{D}', \phi'$ are defined on terms and types in normal form in Figure 9, where we use the notation $S^{\mathsf{wh}(\Delta)}$ to denote the closure of S by weak head expansion, i.e. the set $\{t \in \mathsf{Terms}(\Delta) \mid \exists s \in S.\, t \to^*_{\mathsf{WH}} s\}$.

The interpretation of $\hat{\Sigma}$, $\hat{\Pi}$ and base types is entirely standard. For the interpretation of neutral terms, we simply chose the set of all strongly normalising terms. Turning to $\hat{\forall}$ and $\hat{\triangleright}$, recall that the index δ keeps track of the time that is available on each clock. In the interpretation of $\hat{\forall}$, we are allowed to set the time arbitrarily high for any fresh clock. In the interpretation of $\hat{\triangleright}$, we require that time passes whenever the tick constant $\diamond$ is applied: The clock $\sigma(\kappa)$ is replaced by a clock κ' with a strictly smaller time counter according to δ'. The quantification over all morphisms $\sigma\colon (\Delta, \delta) \to ((\Delta', \sigma(\kappa)), \delta')$, is necessary in order to satisfy (S5), similarly to the interpretation of $\hat{\Pi}$.

Uniqueness of normal forms, which follows from Theorem III.2, ensures that T^0 is a well-defined function. It is also straightforward, if tedious, to check that T^0 maps saturated families to saturated families, and that T^0 is indeed monotone. Hence, according to Theorem VI.2, T^0 has a least fixed point, which we shall denote by $(\mathcal{D}^0, \phi^0)$. The purpose of T^0 is to provide us with $\mathcal{D}^0$, which serves as the interpretation of $\mathcal{U}$.

Definition VI.5. Let $T^1\colon \mathsf{Sat} \to \mathsf{Sat}$ be defined by $T^1(\mathcal{D}, \phi) = (\overline{\mathcal{D}'}, \overline{\phi'})$, where $\mathcal{D}', \phi'$ are defined on terms and types in normal form in way similar to Figure 9, with three changes: Code constructors $\hat{\Pi}, \hat{\Sigma}, \hat{\triangleright}, \hat{\forall}, \hat{\mathsf{Nat}}, \hat{\mathsf{Bool}}, \hat{1}$ are replaced by the corresponding type constructors $\Pi, \Sigma, \triangleright, \forall, \mathsf{Nat}, \mathsf{Bool}, 1$; $\mathcal{U}$ is included in each $\mathcal{D}'_{\Delta,\delta}$; and the equation $\phi'_{\Delta,\delta}(\mathcal{U}) = \mathcal{D}^0_{\Delta,\delta}$ is added to the definition of $\phi'_{\Delta,\delta}$.

Similarly to T^0, we can check that T^1 is indeed a mapping from Sat to Sat, and monotone. We write $(\mathcal{D}^1, \phi^1)$ to denote the least fixed point of T^1, according to Theorem VI.2. ϕ^1 is the interpretation function we are interested in, and we write $[\![\vdash_\Delta A]\!]_\delta$ instead of $\phi^1_{\Delta,\delta}(A)$.

The following lemma relates the two families of interpretation functions ϕ^0 and ϕ^1:

Lemma VI.6. *If $A \in \mathcal{D}^0_{\Delta,\delta}$, then*

(i) $\mathsf{El}\,(A) \in \mathcal{D}^1_{\Delta,\delta}$, *and*
(ii) $\phi^0_{\Delta,\delta}(A) = \phi^1_{\Delta,\delta}(\mathsf{El}\,(A))$.

Proof. Let $(\mathcal{D}^{0,\alpha}, \delta^{0,\alpha})$ be the transfinite sequence constructed as in Theorem VI.2 using the monotone function T^0. That is, there is some α such that $(\mathcal{D}^{0,\alpha}, \delta^{0,\alpha}) = (\mathcal{D}^0, \delta^0)$. Hence, the following is a generalisation of this lemma: For all ordinals α, if $A \in \mathcal{D}^{0,\alpha}_{\Delta,\delta}$, then (i) $\mathsf{El}\,(A) \in \mathcal{D}^1_{\Delta,\delta}$, and (ii) $\phi^{0,\alpha}_{\Delta,\delta}(A) = \phi^1_{\Delta,\delta}(\mathsf{El}\,(A))$. This property can be proved by transfinite induction on α. □

$$
\begin{aligned}
\mathcal{D}'_{\Delta,\delta} = {} & \left\{\hat{1}, \hat{\mathsf{Nat}}, \hat{\mathsf{Bool}}\right\} \cup \mathsf{Neu}(\Delta) \\
\cup {} & \left\{\hat{\Pi}x : A.\, B \,\middle|\, A \in \mathcal{D}_{\Delta,\delta}, \forall\sigma\colon (\Delta,\delta) \to (\Delta',\delta'), t \in \phi_{\Delta',\delta'}(A\,\sigma) : (B\,\sigma)\,[t/x] \in \mathcal{D}_{\Delta',\delta'}\right\} \\
\cup {} & \left\{\hat{\Sigma}x : A.\, B \,\middle|\, A \in \mathcal{D}_{\Delta,\delta}, \forall\sigma\colon (\Delta,\delta) \to (\Delta',\delta'), t \in \phi_{\Delta',\delta'}(A\,\sigma) : (B\,\sigma)\,[t/x] \in \mathcal{D}_{\Delta',\delta'}\right\} \\
\cup {} & \left\{\hat{\triangleright}\,\alpha : \kappa.A \,\middle|\, \begin{array}{l} \forall\alpha' \in \mathsf{TV} : A\,[\alpha'/\alpha] \in \mathcal{D}_{\Delta,\delta}; \\ \forall\sigma\colon (\Delta,\delta) \to ((\Delta',\sigma(\kappa)),\delta'), \kappa' \in \Delta' : \delta'(\kappa') < \delta'(\sigma(\kappa)) \\ \qquad \implies ((A\,\sigma)\,[\kappa'/\sigma(\kappa)])\,[\diamond/\alpha] \in \mathcal{D}_{\Delta',\delta'\restriction\Delta'} \end{array}\right\} \\
\cup {} & \left\{\hat{\forall}\kappa.A \,\middle|\, \forall\kappa' \notin \Delta, n \in \mathbb{N} : A\,[\kappa'/\kappa] \in \mathcal{D}_{(\Delta,\kappa'),\delta[\kappa' \mapsto n]}\right\}
\end{aligned}
$$

If $C \in \mathcal{D}'_{\Delta,\delta}$, then $\phi'_{\Delta,\delta}(C)$ is defined as follows:

$$
\begin{aligned}
\phi'_{\Delta,\delta}(\hat{1}) &= (\{\langle\rangle\} \cup \mathsf{Neu}(\Delta))^{\mathsf{wh}(\Delta)} \\
\phi'_{\Delta,\delta}(\hat{\mathsf{Bool}}) &= (\{\mathsf{true}, \mathsf{false}\} \cup \mathsf{Neu}(\Delta))^{\mathsf{wh}(\Delta)} \\
\phi'_{\Delta,\delta}(\hat{\mathsf{Nat}}) &= \mathcal{N}(\Delta) \\
\phi'_{\Delta,\delta}(\hat{\Pi}x : A.\, B) &= \{t \,|\, \forall\sigma\colon (\Delta,\delta) \to (\Delta',\delta'), s \in \phi_{\Delta',\delta'}(A\,\sigma).\, (t\,\sigma)s \in \phi_{\Delta',\delta'}((B\,\sigma)\ [s/x])\,\} \\
\phi'_{\Delta,\delta}(\hat{\Sigma}x : A.\, B) &= \{t \,|\, \pi_1\, t \in \phi_{\Delta,\delta}(A), \pi_2\, t \in \phi_{\Delta,\delta}(B\,[\pi_1\, t/x])\,\} \\
\phi'_{\Delta,\delta}(\hat{\triangleright}\,\alpha : \kappa.A) &= \left\{ t \,\middle|\, \begin{array}{l} \forall\alpha' \in \mathsf{TV} : t\,[\alpha'] \in \phi_{\Delta,\delta}(A\ [\alpha'/\alpha]); \\ \forall\sigma\colon (\Delta,\delta) \to ((\Delta',\sigma(\kappa)),\delta'), \kappa' \in \Delta' : \delta'(\kappa') < \delta'(\sigma(\kappa)) \\ \qquad \implies ((t\,\sigma)\,[\diamond])\,[\kappa'/\sigma(\kappa)] \in \phi_{\Delta',\delta\restriction\Delta'}(((A\,\sigma)\,[\kappa'/\sigma(\kappa)])\,[\diamond/\alpha]) \end{array}\right\} \\
\phi'_{\Delta,\delta}(\hat{\forall}\kappa.A) &= \left\{t \,\middle|\, \forall\kappa' \notin \Delta, n \in \mathbb{N}.t\,[\kappa'] \in \phi_{(\Delta,\kappa'),\delta[\kappa' \mapsto n]}(A\ [\kappa'/\kappa])\right\} \\
\phi'_{\Delta,\delta}(A) &= \mathsf{SN}(\Delta) \qquad \text{if } A \in \mathsf{Neu}(\Delta)
\end{aligned}
$$

where $\mathcal{N}(\Delta)$ is inductively defined as follows: (i) $0 \in \mathcal{N}(\Delta)$; (ii) $t \in \mathcal{N}(\Delta) \implies \mathsf{suc}\ t \in \mathcal{N}(\Delta)$; (iii) $\mathsf{Neu}(\Delta) \subseteq \mathcal{N}(\Delta)$; and (iv) $t \in \mathcal{N}(\Delta), s \in \mathsf{Terms}(\Delta), s \to_{\mathsf{WH}} t \implies s \in \mathcal{N}(\Delta)$.

Fig. 9. Definition of T^0

The strategy to establish strong normalisation is to prove that $\Gamma \vdash_\Delta t : A$ implies that $t \in [\![\vdash_\Delta A]\!]_\delta$ for all δ. Then $t \in \mathsf{SN}$ follows from saturation property (S4). However, to prove this by induction on the typing derivation, we need to generalise this property to work in the context of appropriate substitutions (for terms, ticks, and clocks). The generalisation to arbitrary clock substitutions is easy: If $\Gamma \vdash_\Delta t : A$, then $t\,\sigma \in [\![\vdash_{\Delta'} A\,\sigma]\!]_\delta$ for all $\sigma\colon \Delta \to \Delta'$ and $\delta\colon \Delta' \to \mathbb{N}$.

To generalise accordingly over term and tick substitutions, we need to lift the type interpretation $[\![\vdash_\Delta \cdot]\!]_\delta$ to typing contexts. Given a typing context $\Gamma \vdash_\Delta$, a clock substitution $\sigma\colon \Delta \to \Delta'$, and an object (Δ,δ) in $\mathcal{K}$, the interpretation of $\Gamma \vdash_\Delta$ w.r.t. σ, δ, written $[\![\Gamma \vdash_\Delta]\!]_{\sigma,\delta}$, is a set of finite mappings $\gamma\colon \mathsf{dom}\,(\Gamma) \to \mathsf{Terms} \cup \mathsf{TV} \cup \{\diamond\}$ inductively defined as follows:

(1) $!\colon \emptyset \to \mathsf{Terms} \cup \mathsf{TV} \cup \{\diamond\} \in [\![\cdot \vdash_\Delta]\!]_{\sigma,\delta}$.

(2) Given $\gamma \in [\![\Gamma \vdash_\Delta]\!]_{\sigma,\delta}$, then

(a) $\gamma\,[x \mapsto t] \in [\![\Gamma, x : A \vdash_\Delta]\!]_{\sigma,\delta}$, if $t \in [\![\vdash_{\Delta'} (A\,\sigma)\,\gamma]\!]_\delta$;

(b) $\gamma\,[\alpha \mapsto \alpha'] \in [\![\Gamma, \alpha : \kappa \vdash_\Delta]\!]_{\sigma,\delta}$, if $\kappa \in \Delta$ and $\alpha' \in \mathsf{TV}$;

(c) $(\gamma\ [\kappa'/\sigma(\kappa)])\,[\alpha \mapsto \diamond] \in [\![\Gamma, \alpha : \kappa \vdash_\Delta]\!]_{[\kappa'/\sigma(\kappa)]\circ\sigma,\delta\restriction\Delta''}$, if $\kappa' \in \Delta'$, $\delta(\kappa') < \delta(\sigma(\kappa))$, and $\Delta'' = \Delta' \setminus \{\sigma(\kappa)\}$.

Note that the interpretation of typing contexts is also indexed by clock substitutions σ. The σ index is used in the case where a substitution maps a tick variable to $\diamond$. The change of the clock substitution σ to $[\kappa'/\sigma(\kappa)] \circ \sigma$ ensures that such a mapping can only occur if the clock variable κ is fresh, which corresponds to the typing rule for $\diamond$ and the interpretation of $\triangleright$-types as given in Figure 9.

Now we are ready to state the fundamental property of $[\![\vdash_\Delta \cdot]\!]_\delta$:

Lemma VI.7 (Fundamental property)**.** *Let (Δ,δ) be an object in $\mathcal{K}$, $\sigma\colon \Delta \to \Delta'$, and $\gamma \in [\![\Gamma \vdash_\Delta]\!]_{\sigma,\delta}$.*

(i) If $\Gamma \vdash_\Delta A$ type, then $(A\,\sigma)\gamma \in \mathcal{D}^1_{\Delta',\delta}$.

(ii) If $\Gamma \vdash_\Delta t : A$, then $(t\,\sigma)\gamma \in [\![\vdash_{\Delta'} (A\,\sigma)\gamma]\!]_\delta$.

Proof sketch. The two properties are proved simultaneously by induction on the size of the derivation of $\Gamma \vdash_\Delta A$ type and $\Gamma \vdash_\Delta t : A$, respectively. We make use of the fact that $\Gamma \vdash_\Delta t : A$ implies $\Gamma \vdash_\Delta A$ type by a derivation of at most the same size. Hence, we may assume that $[\![\vdash_{\Delta'} (A\,\sigma)\gamma]\!]_\delta$ is defined when showing that $(t\,\sigma)\gamma \in [\![\vdash_{\Delta'} (A\,\sigma)\gamma]\!]_\delta$.

The argument for the code introduction rules is similar to the argument for the corresponding type introduction rules, because $[\![\vdash_\Delta \mathcal{U}]\!]_\delta = \mathcal{D}^0_{\Delta,\delta}$. In turn, the argument for type introduction rules is similar (but simpler) to the argument for the corresponding term introduction rule.

The case of the type conversion rule follows from (S2), (S3), and Theorem III.2. The case of the introduction rule for dfix^κ, follows by induction on $\delta(\kappa)$. The case of the introduction rule for $\mathsf{El}\,(\cdot)$ follows from Lemma VI.6. If $\gamma(\alpha) = \diamond$, then the cases for the introduction rules for unfold and fold follow from the induction hypothesis, (S6), and (S3) using the fact that $\mathsf{unfold}_\diamond\, t \to_{\mathsf{WH}} t$ respectively $\mathsf{fold}_\diamond\, t \to_{\mathsf{WH}} t$ as well as the fact that $\mathsf{El}\,(((\mathsf{dfix}^\kappa\, F)\,[\diamond])\,u) \to \mathsf{El}\,(F\,(\mathsf{dfix}^\kappa\, F)\,u)$. If $\gamma(\alpha) \neq \diamond$, then the case for unfold follows from (S7) and the fact that $\mathsf{unfold}_{\gamma(\alpha)}\, t$ is neutral, and the case for fold follows from the fact that $\mathsf{El}\,(((\mathsf{dfix}^\kappa\, F)\,[\gamma(\alpha)])\,u)$ is neutral.

The cases for the introduction of λ abstractions, for both terms and ticks, follows from the induction hypothesis and the definition of $[\![\Gamma \vdash_\Delta]\!]_{\sigma,\delta}$. The same holds for term application as well as tick application. □

From the above lemma, strong normalisation of closed well-typed terms follows immediately. To conclude strong normalisation of open terms, we only require that the identity substitution is contained in the interpretation of the corresponding typing context:

Lemma VI.8. *Let* id_Γ *be the identity on* $\mathsf{dom}\,(\Gamma)$. *Then* $\mathsf{id}_\Gamma \in [\![\Gamma \vdash_\Delta]\!]_{\sigma,\delta}$, *for all* $\sigma\colon \Delta \to \Delta', \delta\colon \Delta' \to \mathbb{N}$.

Proof. This follows by a straightforward induction on Γ. The case for tick variables is trivial, and the case for term variables follows from the saturation property (S7) and the fact that variables are neutral. □

Consequently, if $\Gamma \vdash_\Delta t : A$, then $t = (t\,\mathsf{id}_\Delta)\mathsf{id}_\Gamma \in [\![\vdash_\Delta (A\,\mathsf{id}_\Delta)\mathsf{id}_\Gamma]\!]_\delta$ for any $\delta\colon \Delta \to \mathbb{N}$, and by (S4), we may thus conclude that $t \in \mathsf{SN}$. Similarly, if $\Gamma \vdash_\Delta A$ type, then $A = (A\,\mathsf{id}_\Delta)\mathsf{id}_\Gamma \in \mathcal{D}^1_{\Delta,\delta}$, and by (S2) we have that $A \in \mathsf{SN}$.

VII. Conclusions and future work

We have introduced Clocked Type Theory, a new type theory for guarded recursion, and proved metatheoretic results for it including strong normalisation, confluence and canonicity. As a consequence it follows that all coinductive definitions are productive.

These results form the foundation for a type checking algorithm and an implementation in future work. The rule for application to $\diamond$ (cf. Figure 2, second rule on third line) may cause a challenge for efficient type checking, because of the unusual substitution in the term in the conclusion of the rule. However, the restricted rule (1) or the operator $\mathsf{force} : (\forall\kappa.\triangleright^\kappa A) \to \forall\kappa.A$ may be sufficiently expressive in practice and easier to type check.

Future work also includes denotational semantics of CloTT and extension with path types as outlined in Section V-C.

Acknowledgment

The authors would like to thank Aleš Bizjak for showing us the technique for interpreting universes, and Lars Birkedal and Ranald Clouston for helpful discussions.

This research was supported by The Danish Council for Independent Research for the Natural Sciences (FNU) (grant no. 4002-00442) and research grant 13156 from Villum Fonden.

References

[1] P. Martin-Löf, *Intuitionistic Type Theory*. Napoli: Bibliopolis, 1984.
[2] N. A. Danielsson, "Beating the productivity checker using embedded languages," in *PAR*, vol. 43, 2010, pp. 29–48.
[3] H. Nakano, "A modality for recursion," in *LICS*, 2000, pp. 255–266.
[4] A. Bizjak, H. B. Grathwohl, R. Clouston, R. E. Møgelberg, and L. Birkedal, "Guarded dependent type theory with coinductive types," in *FOSSACS*, 2016.
[5] R. Atkey and C. McBride, "Productive coprogramming with guarded recursion," in *ICFP*. ACM, 2013, pp. 197–208.
[6] R. E. Møgelberg and M. Paviotti, "Denotational semantics of recursive types in synthetic guarded domain theory," in *LICS*, 2016.
[7] T. Univalent Foundations Program, *Homotopy Type Theory: Univalent Foundations of Mathematics*, Institute for Advanced Study, 2013.
[8] 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*, vol. 8, no. 4, 2012.
[9] A. Bizjak, L. Birkedal, and M. Miculan, "A model of countable nondeterminism in guarded type theory," in *RTA-TLCA*, 2014, pp. 108–123.
[10] K. Svendsen and L. Birkedal, "Impredicative concurrent abstract predicates," in *ESOP*, 2014.
[11] A. W. Appel and D. A. McAllester, "An indexed model of recursive types for foundational proof-carrying code," *ACM Trans. Program. Lang. Syst*, vol. 23, no. 5, pp. 657–683, 2001.
[12] L. Birkedal, A. Bizjak, R. Clouston, H. B. Grathwohl, B. Spitters, and A. Vezzosi, "Guarded cubical type theory: Path equality for guarded recursion," in *CSL*, 2016.
[13] C. Cohen, T. Coquand, S. Huber, and A. Mörtberg, "Cubical type theory: a constructive interpretation of the univalence axiom," 2016. [Online]. Available: http://arxiv.org/abs/1611.02108
[14] R. Clouston, A. Bizjak, H. B. Grathwohl, and L. Birkedal, "Programming and reasoning with guarded recursion for coinductive types," in *FoSSaCS*, 2015.
[15] J. Cheney, "A dependent nominal type theory," *LMCS*, vol. 8, no. 1, 2012.
[16] A. Abel, B. Pientka, D. Thibodeau, and A. Setzer, "Copatterns: programming infinite structures by observations," in *POPL*, 2013.
[17] J. Hughes, L. Pareto, and A. Sabry, "Proving the correctness of reactive systems using sized types," in *POPL*, 1996.
[18] G. Barthe, M. J. Frade, E. Giménez, L. Pinto, and T. Uustalu, "Type-based termination of recursive definitions," *Mathematical Structures in Computer Science*, vol. 14, no. 1, pp. 97–141, 2004.
[19] A. Abel and B. Pientka, "Well-founded recursion with copatterns and sized types," *J. Funct. Program.*, vol. 26, p. e2, 2016.
[20] A. Abel and A. Vezzosi, "A formalized proof of strong normalization for guarded recursive types," in *APLAS*, 2014.
[21] P. Severi and F. de Vries, "Pure type systems with corecursion on streams: from finite to infinitary normalisation," in *ICFP*, 2012, pp. 141–152.
[22] A. Jeffrey, "Functional reactive types," in *CSL-LICS*. ACM, 2014, pp. 54:1–54:9.
[23] N. R. Krishnaswami, "Higher-order functional reactive programming without spacetime leaks," *ACM SIGPLAN Notices*, vol. 48, no. 9, pp. 221–232, Sep. 2013.
[24] A. Cave, F. Ferreira, P. Panangaden, and B. Pientka, "Fair reactive programming," in *POPL*. ACM, 2014, pp. 361–372.
[25] P. Bahr, H. B. Grathwohl, and R. E. Møgelberg, "The clocks are ticking: No more delays! – Technical appendix," available from authors' websites.
[26] J. Y. Girard, "Linear logic," *Theor. Comput. Sci.*, vol. 50, pp. 1–102, 1987.
[27] R. E. Møgelberg, "A type theory for productive coprogramming via guarded recursion," in *CSL-LICS*, 2014, pp. 71:1–71:10.
[28] M. Takahashi, "Parallel reductions in λ-calculus," *Information and Computation*, vol. 118, no. 1, pp. 120 – 127, 1995.
[29] W. W. Tait, "Intensional interpretations of functionals of finite type I," *J. Symbolic Logic*, vol. 32, no. 2, pp. 198–212, 1967.
[30] T. Coquand and J. Gallier, "A proof of strong normalization for the theory of constructions using a Kripke-like interpretation," in *First Annual Workshop on Logical Frameworks*, 1990.
[31] R. Harper, "Constructing type systems over an operational semantics," *Journal of Symbolic Computation*, vol. 14, no. 1, pp. 71 – 84, 1992.