What makes guarded types tick?

Patrick Bahr, Bassel Mannaa, and Rasmus Ejlers Møgelberg

IT University of Copenhagen (paba,basm,mogel@itu.dk)

Abstract

We give an overview of the syntax and semantics of Clocked Type Theory (CloTT), a dependent type theory for guarded recursion with many clocks, in which one can encode coinductive types and capture the notion of productivity in types. The main novelty of CloTT is the notion of ticks, which allows one to open the delay type modality, and, e.g., define a dependent form of applicative functor action, which can be used for reasoning about coinductive data. In the talk we will give examples of programming and reasoning about guarded recursive and coinductive data in CloTT, and we will present the main syntactic results: Strong normalisation, canonicity and decidability of type checking. If time permits, we will also sketch the main ideas of the denotational semantics for CloTT. The results presented in this talk have previously been published in [2, 5].

Guarded recursion

The idea of guarded recursion [6], is to ensure productivity of recursive definitions by guarding all recursive calls by a delay type modality $\triangleright$ (pronounced *later*). We think of $\triangleright A$ as classifying data of type A that is available one time step from now. The assumptions on $\triangleright$ are a term $\mathsf{next} : A \to \triangleright A$ and a fixed point operator $\mathsf{fix} : (\triangleright A \to A) \to A$. The latter can be used in combination with guarded recursive types to define recursive programs. Suppose for example, that Str is a type of guarded streams satisfying the type equation $\mathsf{Str} \equiv \mathbb{N} \times \triangleright \mathsf{Str}$. One can then define a constant stream of zeros as $\mathsf{fix}(\lambda x.\,(0, x))$. For higher order programming with guarded streams, one needs to assume that $\triangleright$ is an applicative functor, with applicative action

$$\circledast : \triangleright(A \to B) \to \triangleright A \to \triangleright B$$

For example, given $f : \mathbb{N} \to \mathbb{N}$ one can define $\mathsf{map}\, f : \mathsf{Str} \to \mathsf{Str}$ as

$$\mathsf{fix}(\lambda g.\lambda\,(x, xs)\,.\,(f(x), g \circledast xs))$$

Ticks

Suppose now, that we want to reason about guarded streams using guarded recursion. As a simple motivating example, suppose we are given some predicate on $\mathbb{N}$ in the form of a dependent type $x : \mathbb{N} \vdash P(x)\,\mathsf{type}$. A lifting of P to guarded streams should be a dependent type $xs : \mathsf{Str} \vdash \hat{P}(xs)\,\mathsf{type}$ satisfying $\hat{P}\,(x, xs)$ iff $P(x)$ and "$\hat{P}(xs)$". However, the latter statement is not well typed, since xs has type $\triangleright\mathsf{Str}$ and $\hat{P}$ expects an element of Str. Rather, it should be true that one time step from now, $\hat{P}$ holds of the stream delivered at that time by xs. *Ticks* are evidence that time has passed, and they allow us to open elements of type $\triangleright A$. The rules for ticks can be described in a small dependently typed *tick calculus*:

$$\frac{\Gamma \vdash}{\Gamma, \alpha{:}\mathrm{tick} \vdash} \qquad \frac{\Gamma, \alpha{:}\mathrm{tick} \vdash A}{\Gamma \vdash \triangleright(\alpha{:}\mathrm{tick})A}$$

$$\frac{\Gamma, \alpha{:}\mathrm{tick} \vdash t : A}{\Gamma \vdash \lambda(\alpha{:}\mathrm{tick})t : \triangleright(\alpha{:}\mathrm{tick})A} \qquad \frac{\Gamma \vdash t : \triangleright(\alpha{:}\mathrm{tick})A}{\Gamma, \beta{:}\mathrm{tick}, \Gamma' \vdash t\,[\beta] : A[\beta/\alpha]}$$

A tick in a context $\Gamma, \beta{:}\mathsf{tick}, \Gamma'$ can be thought of as dividing the context into variables (Γ), that arrive before the tick, and variables (Γ') that arrive after. The rule for tick application $t\,[\beta]$ can be thought of as stating that t must be typed already before the tick β occurs. We write $\triangleright A$ for $\triangleright(\alpha{:}\mathsf{tick}).A$ whenever α does not occur in A, and similarly $\mathsf{next}\,t$ for $\lambda(\alpha{:}\mathsf{tick}).t$ whenever α does not occur in A.

The requirement for the lifting $\hat{P}$ can now be described as $\hat{P}\,(x, xs) \equiv P(x) \times \triangleright(\alpha{:}\mathsf{tick})\hat{P}(xs[\alpha])$. Moreover, one can generalise the applicative action to dependent types as follows:

$$\lambda f.\lambda y.\lambda(\alpha{:}\mathsf{tick}).f\,[\alpha](y\,[\alpha]) : \triangleright\left(\prod(x : A)\,.B\right) \to \prod(y : \triangleright A)\,.\triangleright(\alpha{:}\mathsf{tick}).B[y\,[\alpha]/x]$$

We can now use fix also for reasoning. For example a proof $p : \Pi(x{:}\mathsf{N})P$ can be lifted to a proof of $\Pi(y{:}\mathsf{Str})\hat{P}(y)$ as follows: Consider first

$$f : \triangleright(\Pi(y{:}\mathsf{Str})\hat{P}(y)) \to \Pi(y{:}\mathsf{Str})\hat{P}(y)$$
$$f\,q\,((x, xs)) \stackrel{\mathsf{def}}{=} (p(x), \lambda(\alpha{:}\mathsf{tick})q\,[\alpha](xs\,[\alpha]))$$

then $\mathsf{fix}(f) : \Pi(y{:}\mathsf{Str})\hat{P}(y)$.

Clocked Type Theory

The goal of *Clocked Type Theory* (CloTT) is to allow programming and reasoning with guarded recursive types but also with coinductive types. In order to work with coinductive types, CloTT combines ticks with multiple clocks and clock quantification as pioneered by Atkey & McBride [1]. To this end, the typing judgement includes a *clock context* Δ, which is a finite set of *clocks*. Moreover, each tick belongs to a clock κ; instead of $\alpha{:}\mathsf{tick}$, we write $\alpha{:}\kappa$ to indicate that α is a tick on the clock κ. Correspondingly, the later modality is written as $\triangleright(\alpha{:}\kappa).A$ and we write $\triangleright^\kappa A$ if α does not appear in A. The type of guarded streams over clock κ then becomes Str^κ, satisfying $\mathsf{Str}^\kappa \equiv \mathbb{N} \times \triangleright^\kappa \mathsf{Str}^\kappa$.

For guarded streams, we can construct head and tail functions as follows:

$$\mathsf{hd}^\kappa : \mathsf{Str}^\kappa \to \mathsf{N} \qquad \mathsf{tl}^\kappa : \mathsf{Str}^\kappa \to \triangleright^\kappa \mathsf{Str}^\kappa$$
$$\mathsf{hd}^\kappa \stackrel{\mathsf{def}}{=} \lambda x : \mathsf{Str}^\kappa.\pi_1\,x \qquad \mathsf{tl}^\kappa \stackrel{\mathsf{def}}{=} \lambda x : \mathsf{Str}^\kappa.\lambda(\alpha{:}\kappa).(\pi_2\,x)\,[\alpha]$$

We can use them to get the second element of a stream like this:

$$\lambda(xs : \mathsf{Str}^\kappa).\lambda(\alpha{:}\kappa).\mathsf{hd}^\kappa(\mathsf{tl}^\kappa\,xs\,[\alpha]) : \mathsf{Str}^\kappa \to \triangleright^\kappa \mathsf{N}$$

Note that the result type is $\triangleright^\kappa\mathsf{N}$ as opposed to N. To get the second element of the stream as a term of type N we have to consider coinductive streams.

To form coinductive types, we use clock quantification $\forall\kappa.A$. An element of type $\forall\kappa.A$ is a term of type A that can compute for an arbitrary number of ticks on clock κ. This intuition is realised in CloTT by the addition of a *tick constant* $\diamond : \kappa$, for any clock κ. The tick constant $\diamond$ can be used arbitrarily often for a term of type $\forall\kappa.A$, which is expressed in the typing rule for tick constant application $t\,[\diamond]$:

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

For the consistency of the calculus it is crucial that the typing rule for $t\,[\diamond]$ requires the clock κ to be fresh for the context Γ. While this rule provides sufficient expressivity of the calculus,

the rule has to be generalised slightly in order to obtain standard syntactic properties such as closure under substitution and subject reduction.

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

In this more general rule, the clock κ is substituted away in the conclusion.

Given the type of guarded streams Str^κ over κ, we can construct the type of coinductive streams by a simple clock quantification: $\mathsf{Str}^{\mathsf{c}} \stackrel{\mathsf{def}}{=} \forall\kappa.\mathsf{Str}^\kappa$. Returning to the example of obtaining the second element of a stream, the coinductive type allows us to get that second element of the stream of type N, because we may use the tick constant $\diamond$:

$$\mathsf{hd} : \mathsf{Str}^{\mathsf{c}} \to \mathsf{N} \qquad\qquad \mathsf{tl} : \mathsf{Str}^{\mathsf{c}} \to \mathsf{Str}^{\mathsf{c}}$$

$$\mathsf{hd} \stackrel{\mathsf{def}}{=} \lambda x : \mathsf{Str}.\mathsf{hd}^{\kappa_0}\,(x\,[\kappa_0]) \qquad\qquad \mathsf{tl} \stackrel{\mathsf{def}}{=} \lambda x : \mathsf{Str}^{\mathsf{c}}.\Lambda\kappa.(\mathsf{tl}^\kappa\,(x\,[\kappa]))\,[\diamond]$$

$$\lambda(xs : \mathsf{Str}^{\mathsf{c}}).\mathsf{hd}(\mathsf{tl}\,xs) : \mathsf{Str}^{\mathsf{c}} \to \mathsf{N}$$

Reduction semantics

The main motivation for introducing ticks is to be able to compute with guarded recursive dependent types. Bizjak et al. [4] introduced *delayed substitutions* as a means to combine guarded types with dependent types. However, delayed substitutions turn out to be inappropriate for devising a reduction semantics. With ticks, computations on later types can be expressed as straightforward beta and eta reductions:

$$(\lambda(\alpha{:}\kappa).t)\,[\alpha'] \to t\,[\alpha'/\alpha] \qquad\qquad \lambda(\alpha{:}\kappa).(t\,[\alpha]) \to t$$

$$(\Lambda\kappa.t)[\kappa'] \to t[\kappa'/\kappa] \qquad\qquad (\Lambda\kappa.t[\kappa]) \to t$$

However, this still leaves us with the fixed point combinator $\mathsf{fix}^\kappa : (\triangleright^\kappa A \to A) \to A$. Allowing arbitrary unfolding of fixed points by adding the rule $\mathsf{fix}^\kappa\, t \to t(\mathsf{next}^\kappa\,(\mathsf{fix}^\kappa\, t))$, would immediately yield a non-normalising reduction system.

Instead, we introduce a combinator $\mathsf{dfix}^\kappa : (\triangleright^\kappa A \to A) \to \triangleright^\kappa A$ as a primitive and derive fix^κ from it by defining $\mathsf{fix}^\kappa \stackrel{\mathsf{def}}{=} \lambda x.x(\mathsf{dfix}^\kappa\, x)$. Since a term $\mathsf{dfix}^\kappa t$ is of type $\triangleright^\kappa A$ rather than A, we are not forced to unfold dfix^κ for the sake of canonicity. We only have to unfold it if that term is applied to the tick constant $\diamond$ and thus forms a term of type A:

$$(\mathsf{dfix}^\kappa\, t)\,[\diamond] \to t\,(\mathsf{dfix}^\kappa\, t)$$

With the above rule we still maintain canonicity since any term $(\mathsf{dfix}^\kappa\, t)\,[\alpha]$, where α is a tick variable, is considered an open term. Moreover, the rule is restricted enough to ensure strong normalisation:

Theorem 1 (Strong normalisation & canonicity)**.**

1. If $\Gamma \vdash_\Delta t : A$, then t is strongly normalising.
2. If $\vdash_\Delta t : \mathsf{N}$, then $t \to^* \mathsf{suc}^n\ 0$ for some $n \in \mathbb{N}$.

Since the reduction semantics is confluent as well, we also obtain a decision procedure for the equational theory of the calculus. Details of CloTT and its reduction semantics can be found in Bahr et al. [2].

Denotational semantics

If time permits, we will also discuss the denotational semantics of CloTT. Dependent type theory with guarded recursion on a single clock can be given a denotational semantics in the topos of trees [3], modelling a closed type as a family of sets X_n indexed by natural numbers, together with restriction maps $r^n_X : X_{n+1} \to X_n$ for each n. The delay type operator is modelled as $(\triangleright X)_0 = 1$ (the singleton set) and $(\triangleright X)_{n+1} = X_n$. This allows the fixed point operator to be modelled using natural number induction. For example, the type of guarded streams satisfying $\mathsf{Str} \equiv \mathbb{N} \times \triangleright\mathsf{Str}$ can be modelled as $\mathsf{Str}_n = \mathbb{N}^{n+1} \times 1$ (associating the products to the right for strict equality of types), and the lifting of P satisfying $\hat{P}\,(x, xs) \equiv P(x) \times \triangleright(\alpha{:}\mathrm{tick})\hat{P}(xs[\alpha])$ can be modelled as

$$\hat{P}(x_n, (x_{n-1}, \dots, (x_0, \star)\dots) = \{(p_n, (p_{n-1}, \dots, (p_0, \star)\dots) \mid \forall i. p_i \in P(x_i)\}$$

There is no object of ticks in this model. Instead, ticks in contexts are modelled using the left adjoint $\triangleleft$ to $\triangleright$ defined as $(\triangleleft X)_n = X_{n+1}$ by defining $[\![\Gamma, \alpha : \mathrm{tick}]\!] = \triangleleft[\![\Gamma]\!]$. The multiclock case is more complex, but follows the same pattern: To model clocks, one needs a category with family (a standard notion of model of dependent type theory), an adjunction of endofunctors $L \dashv R$ on the underlying category, an extension of R to types and terms, and a projection $L \to \mathsf{id}$ to model tick weakening. The details of this model can be found in [5].

References

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

[2] P. Bahr, H. B. Grathwohl, and R. E. Møgelberg. The clocks are ticking: No more delays! In *LICS*, pages 1–12. IEEE, 2017.

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

[4] A. Bizjak, H. B. Grathwohl, R. Clouston, R. E. Møgelberg, and L. Birkedal. Guarded dependent type theory with coinductive types. In *FOSSACS*, pages 20–35, 2016.

[5] Bassel Mannaa and Rasmus Ejlers Møgelberg. The clocks they are adjunctions: Denotational semantics for clocked type theory. *arXiv preprint arXiv:1804.06687*, 2018.

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