What makes guarded types tick?

Syntax and Semantics for Type Theory with Guarded Recursion and Ticks

Patrick Bahr Bassel Mannaa
Rasmus Møgelberg

IT University of Copenhagen

Overview

1. Intro: Guarded Recursion

2. Guarded Recursion + Dependent Types

3. Coinductive Types via Clocks

Overview

1. Intro: Guarded Recursion

Clocked Type Theory (CloTT)

2. Guarded Recursion + Dependent Types

3. Coinductive Types via Clocks

Overview

1. Intro: Guarded Recursion

Clocked Type Theory (CloTT)

2. Guarded Recursion + Dependent Types
3. Coinductive Types via Clocks
4. Reduction Semantics
5. Denotational Semantics

Guarded Recursion

H. Nakano. *A modality for recursion.* LICS 2000

Guarded recursion on one slide

- type modality $\triangleright$ (pronounced "later")
  - e.g. $t : \triangleright A$
  - t promises to evaluate to a value of type A in next time step

H. Nakano. *A modality for recursion.* LICS 2000

Guarded recursion on one slide

- type modality $\triangleright$ (pronounced “later”)
  - e.g. $t : \triangleright A$
  - t promises to evaluate to a value of type A in next time step
- instead of general fixed-point operator

$$\mathrm{fix} : (A \to A) \to A$$

H. Nakano. *A modality for recursion.* LICS 2000

Guarded recursion on one slide

- type modality $\triangleright$ (pronounced "later")
  - e.g. $t : \triangleright A$
  - t promises to evaluate to a value of type A in next time step
- instead of general fixed-point operator

$$\cancel{\mathrm{fix} : (A \to A) \to A}$$

- we have the guarded fixed-point operator

$$\mathrm{fix} : (\triangleright A \to A) \to A$$

H. Nakano. *A modality for recursion.* LICS 2000

What is guarded recursion?

- abstract form of step-indexing via $\triangleright$
- allows to add general recursive types without breaking consistency

What is it good for?

- For reasoning: construct models of programming languages and type systems.
- For programming: ensures productivity of coinductive definitions – in a modular way.

'Later' as applicative functor

- $\triangleright$ is an applicative functor [1]

$$\begin{aligned} \mathrm{next} &: A \to \triangleright A \\ \circledast &: \triangleright(A \to B) \to \triangleright A \to \triangleright B \end{aligned}$$

- guarded fixed-point operator
  $\mathrm{fix} : (\triangleright A \to A) \to A$ satisfies

$$\mathrm{fix}\, f = f(\mathrm{next}(\mathrm{fix}\, f))$$

[1] Atkey & McBride. Productive Coprogramming with Guarded Recursion, ICFP 2013

Example

Guarded recursive type of streams

$$\mathsf{Str} \cong \mathsf{Nat} \times \triangleright\mathsf{Str}$$

Example

Guarded recursive type of streams

$$\mathsf{Str} \cong \mathsf{Nat} \times \triangleright\mathsf{Str}$$

Example

Guarded recursive type of streams

$$\mathsf{Str} \cong \mathsf{Nat} \times \triangleright\mathsf{Str}$$

Let's write a function that increments each element of a stream:

$$\begin{aligned} &\mathsf{incr} : \mathsf{Str} \to \mathsf{Str} \\ &\mathsf{incr} \triangleq \mathsf{fix}\,\lambda g : \triangleright(\mathsf{Str} \to \mathsf{Str}). \\ &\qquad\qquad \lambda x : \mathsf{Str}.\, \langle \mathsf{suc}\,(\pi_1 x), g \circledast (\pi_2\, x)\rangle \end{aligned}$$

Guarded Recursion

$+$

Dependent Types

P. Bahr, H. Bugge Grathwohl and R.E. Møgelberg *The Clocks Are Ticking: No More Delays!* LICS 2017

Combining $\triangleright$ and dependent types

Simple types

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

Combining $\triangleright$ and dependent types

Simple types

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

Dependent types

$$\frac{\Gamma \vdash s : \Pi x : A.B \qquad \Gamma \vdash t : A}{\Gamma \vdash s\, t : B\,[t/x]}$$

Combining $\triangleright$ and dependent types

Simple types

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

Dependent types

$$\frac{\Gamma \vdash s : \Pi x : A.B \qquad \Gamma \vdash t : A}{\Gamma \vdash s\,t : B\,[t/x]}$$

$$\frac{\Gamma \vdash s : \triangleright(\Pi x : A.B) \qquad \Gamma \vdash t : \triangleright A}{\Gamma \vdash s \circledast t : ???}$$

Combining $\triangleright$ and dependent types

Simple types

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

Dependent types

$$\frac{\Gamma \vdash s : \Pi x : A.B \qquad \Gamma \vdash t : A}{\Gamma \vdash s\, t : B\,[t/x]}$$

$$\frac{\Gamma \vdash s : \triangleright(\Pi x : A.B) \qquad \Gamma \vdash t : \triangleright A}{\Gamma \vdash s \circledast t : \triangleright B\,[t/x]}$$

Combining $\triangleright$ and dependent types

Simple types

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

Dependent types

$$\frac{\Gamma \vdash s \colon \Pi x : A.B \qquad \Gamma \vdash t \colon A}{\Gamma \vdash s\, t \colon B\,[t/x]}$$

$$\frac{\Gamma \vdash s \colon \triangleright(\Pi x : A.B) \qquad \Gamma \vdash t \colon \triangleright A}{\Gamma \vdash s \circledast t \colon \triangleright B\,[t/x]}$$

Combining $\triangleright$ and dependent types

Simple types

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

Dependent types

$$\frac{\Gamma \vdash s \colon \Pi x : A.B \qquad \Gamma \vdash t \colon A}{\Gamma \vdash s\, t \colon B\,[t/x]}$$

$$\frac{\Gamma \vdash s \colon \triangleright(\Pi x : A.B) \qquad \Gamma \vdash t \colon \triangleright A}{\Gamma \vdash s \circledast t \colon \triangleright B\,[t/x]}$$

We need: eliminate $\triangleright$ in a controlled way

Delayed Substitutions

[Bizjak et al. FoSSaCS 2016]

Instead of $$\frac{\Gamma \vdash s\colon \triangleright(\Pi x : A.B) \qquad \Gamma \vdash t\colon \triangleright A}{\Gamma \vdash s \circledast t\colon \triangleright B\,[t/x]}$$

they have $$\frac{\Gamma \vdash s\colon \triangleright(\Pi x : A.B) \qquad \Gamma \vdash t\colon \triangleright A}{\Gamma \vdash s \circledast t\colon \triangleright[x \leftarrow t]\,.B}$$

Delayed Substitutions

[Bizjak et al. FoSSaCS 2016]

Instead of
$$\frac{\Gamma \vdash s : \triangleright(\Pi x : A.B) \qquad \Gamma \vdash t : \triangleright A}{\Gamma \vdash s \circledast t : \triangleright B\,[t/x]}$$

they have
$$\frac{\Gamma \vdash s : \triangleright(\Pi x : A.B) \qquad \Gamma \vdash t : \triangleright A}{\Gamma \vdash s \circledast t : \triangleright \underbrace{[x \leftarrow t]\,.B}_{\text{"let next } x = t \text{ in } B\text{"}}}$$

Delayed Substitutions

[Bizjak et al. FoSSaCS 2016]

Instead of
$$\frac{\Gamma \vdash s : \triangleright(\Pi x : A.B) \qquad \Gamma \vdash t : \triangleright A}{\Gamma \vdash s \circledast t : \triangleright B\,[t/x]}$$

they have
$$\frac{\Gamma \vdash s : \triangleright(\Pi x : A.B) \qquad \Gamma \vdash t : \triangleright A}{\Gamma \vdash s \circledast t : \triangleright \underbrace{[x \leftarrow t]\,.B}_{\text{"let next } x = t \text{ in } B\text{"}}}$$

In general

$$\triangleright\left[x_1 \leftarrow t_1, \ldots x_n \leftarrow t_n\right].A$$

$$\mathrm{next}\left[x_1 \leftarrow t_1, \ldots x_n \leftarrow t_n\right].t$$

Equalities

$$
\begin{aligned}
\triangleright^{\kappa}\xi\left[x \leftarrow \mathsf{next}\xi.u\right].A &= \triangleright^{\kappa}\xi.A\left[u/x\right] \\
\triangleright^{\kappa}\xi\left[x \leftarrow u\right].A &= \triangleright^{\kappa}\xi.A && \text{if } x \notin \mathsf{fv}(A) \\
\triangleright^{\kappa}\xi\left[x \leftarrow u, y \leftarrow v\right]\xi'.A &= \triangleright^{\kappa}\xi\left[y \leftarrow v, x \leftarrow u\right]\xi'.A && \text{if } \ldots
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{next}\xi\left[x \leftarrow \mathsf{next}\xi.u\right].t &= \mathsf{next}\xi.t\left[u/x\right] \\
\mathsf{next}\xi\left[x \leftarrow u\right].t &= \mathsf{next}\xi.t && \text{if } x \notin \mathsf{fv}(t) \\
\mathsf{next}\xi\left[x \leftarrow u, y \leftarrow v\right]\xi'.t &= \mathsf{next}\xi\left[y \leftarrow v, x \leftarrow u\right]\xi'.t && \text{if } \ldots \\
\mathsf{next}\xi\left[x \leftarrow t\right].x &= t
\end{aligned}
$$

Equalities

$$\begin{aligned}
\triangleright^{\kappa}\xi\,[x \leftarrow \mathsf{next}\xi.u]\,.A &= \triangleright^{\kappa}\xi.A\,[u/x] \\
\triangleright^{\kappa}\xi\,[x \leftarrow u]\,.A &= \triangleright^{\kappa}\xi.A && \text{if } x \notin \mathsf{fv}(A) \\
\triangleright^{\kappa}\xi\,[x \leftarrow u, y \leftarrow v]\,\xi'.A &= \triangleright^{\kappa}\xi\,[y \leftarrow v, x \leftarrow u]\,\xi'.A && \text{if } \ldots
\end{aligned}$$

$$\begin{aligned}
\mathsf{next}\xi\,[x \leftarrow \mathsf{next}\xi.u]\,.t &= \mathsf{next}\xi.t\,[u/x] \\
\mathsf{next}\xi\,[x \leftarrow u]\,.t &= \mathsf{next}\xi.t && \text{if } x \notin \mathsf{fv}(t) \\
\mathsf{next}\xi\,[x \leftarrow u, y \leftarrow v]\,\xi'.t &= \mathsf{next}\xi\,[y \leftarrow v, x \leftarrow u]\,\xi'.t && \text{if } \ldots \\
\mathsf{next}\xi\,[x \leftarrow t]\,.x &= t
\end{aligned}$$

Not clear how to devise a confluent & normalising reduction semantics that verify these equalities.

Making guarded types tick

- Treat $\triangleright A$ as function type "tick $\to A$"
- generalise to dependent function type:
  $\triangleright(\alpha : \mathsf{tick}).A$

Making guarded types tick

- Treat $\triangleright A$ as function type "tick $\to A$"
- generalise to dependent function type: $\triangleright(\alpha : \mathsf{tick}).A$

$$\frac{\Gamma, \alpha : \mathsf{tick} \vdash t : A}{\Gamma \vdash \lambda(\alpha : \mathsf{tick}).t : \triangleright(\alpha : \mathsf{tick}).A}$$

Making guarded types tick

- Treat $\triangleright A$ as function type "tick $\to A$"
- generalise to dependent function type: $\triangleright(\alpha : \mathsf{tick}).A$

$$\frac{\Gamma, \alpha : \mathsf{tick} \vdash t : A}{\Gamma \vdash \lambda(\alpha : \mathsf{tick}).t : \triangleright(\alpha : \mathsf{tick}).A}$$

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

Making guarded types tick

- Treat $\triangleright A$ as function type "tick $\to A$"
- generalise to dependent function type:
  $\triangleright(\alpha : \mathsf{tick}).A$

$$\frac{\Gamma, \alpha : \mathsf{tick} \vdash t : A}{\Gamma \vdash \lambda(\alpha : \mathsf{tick}).t : \triangleright(\alpha : \mathsf{tick}).A}$$

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

available before tick β occurred

Making guarded types tick

- Treat $\triangleright A$ as function type "tick $\to A$"
- generalise to dependent function type:
  $\triangleright(\alpha : \mathsf{tick}).A$

$$\frac{\Gamma, \alpha : \mathsf{tick} \vdash t : A}{\Gamma \vdash \lambda(\alpha : \mathsf{tick}).t : \triangleright(\alpha : \mathsf{tick}).A}$$

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

available before tick β occurred

available after tick β occurred

Making guarded types tick

- Treat $\triangleright A$ as function type "tick $\to A$"
- generalise to dependent function type: $\triangleright(\alpha : \mathsf{tick}).A$

$$\frac{\Gamma, \alpha : \mathsf{tick} \vdash t : A}{\Gamma \vdash \lambda(\alpha : \mathsf{tick}).t : \triangleright(\alpha : \mathsf{tick}).A}$$

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

Making guarded types tick

- Treat $\triangleright A$ as function type "tick $\to A$"
- generalise to dependent function type: $\triangleright(\alpha : \mathsf{tick}).A$

$$\frac{\Gamma, \alpha : \mathsf{tick} \vdash t : A}{\Gamma \vdash \lambda(\alpha : \mathsf{tick}).t : \triangleright(\alpha : \mathsf{tick}).A}$$

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

Dependent version of $\circledast$ can be defined with type:

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

Examples

Next

$$\lambda(x : A).\lambda(\alpha : \mathsf{tick}).x \ : \ A \to \ \triangleright A$$

Examples

Next

$$\lambda(x : A).\lambda(\alpha : \mathsf{tick}).x \; : \; A \to \; \triangleright A$$

$= \triangleright(\alpha : \mathsf{tick}).A$, for fresh α.

Examples

Next

$$\lambda(x : A).\lambda(\alpha : \mathsf{tick}).x \;:\; A \to \;\triangleright A$$

Applicative action

Given $s : \triangleright(\alpha : \mathsf{tick}).(\Pi x : A.B)$,
and $t : \triangleright(\alpha : \mathsf{tick}).A$

define $\qquad s \circledast t = \lambda(\alpha : \mathsf{tick}).(s\,[\alpha])(t\,[\alpha])$

Examples

Next

$$\lambda(x : A).\lambda(\alpha : \mathsf{tick}).x \;:\; A \to \;\triangleright A$$

Applicative action

Given $s : \triangleright(\alpha : \mathsf{tick}).(\Pi x : A.B)$,
and $t : \triangleright(\alpha : \mathsf{tick}).A$

define $\qquad s \circledast t = \lambda(\alpha : \mathsf{tick}).(s\,[\alpha])(t\,[\alpha])$
of type $\qquad \triangleright(\alpha : \mathsf{tick}).B\,[t\,[\alpha]/x]$

Examples

Next

$$\lambda(x : A).\lambda(\alpha : \mathsf{tick}).x \; : \; A \to \; \triangleright A$$

Applicative action

Given $s : \triangleright(\alpha : \mathsf{tick}).(\Pi x : A.B)$,
and $t : \triangleright(\alpha : \mathsf{tick}).A$

define $\quad s \circledast t = \lambda(\alpha : \mathsf{tick}).(s\,[\alpha])(t\,[\alpha])$
of type $\quad \triangleright(\alpha : \mathsf{tick}).B\,[t\,[\alpha]/x]$

Non-example

$\lambda(\alpha : \mathsf{tick}).t\,[\alpha]\,[\alpha]$ is not well typed!

Extended Example

- Given $x : \mathbb{N} \vdash P(x)$ type, i.e. a predicate on $\mathbb{N}$.
- Lift P to $xs : \mathsf{Str} \vdash \hat{P}(xs)$ type, i.e. a predicate on streams:

$$\hat{P}\,\langle x, xs\rangle \cong P(x) \times \triangleright(\alpha : \mathsf{tick}).\hat{P}(xs[\alpha])$$

Extended Example

- Given $x : \mathbb{N} \vdash P(x)$ type, i.e. a predicate on $\mathbb{N}$.
- Lift P to $xs : \mathsf{Str} \vdash \hat{P}(xs)$ type, i.e. a predicate on streams:

$$\hat{P}\,\langle x, xs\rangle \cong P(x) \times \triangleright(\alpha : \mathsf{tick}).\hat{P}(xs[\alpha])$$

- A proof $p : \Pi(x : \mathsf{Nat}).P(x)$ can be lifted to a proof of $\Pi(y : \mathsf{Str}).\hat{P}(y)$ using fix:

$$f : \triangleright(\Pi(y{:}\mathsf{Str}).\hat{P}(y)) \to \Pi(y{:}\mathsf{Str}).\hat{P}(y)$$

$$f\ q\,(\langle x, xs\rangle) \triangleq \langle p(x), \lambda(\alpha{:}\mathsf{tick}).(q\,[\alpha])(xs\,[\alpha])\rangle$$

Extended Example

- Given $x : \mathbb{N} \vdash P(x)$ type, i.e. a predicate on $\mathbb{N}$.
- Lift P to $xs : \mathsf{Str} \vdash \hat{P}(xs)$ type, i.e. a predicate on streams:

$$\hat{P}\,\langle x, xs\rangle \cong P(x) \times \triangleright(\alpha : \mathsf{tick}).\hat{P}(xs[\alpha])$$

- A proof $p : \Pi(x : \mathsf{Nat}).P(x)$ can be lifted to a proof of $\Pi(y : \mathsf{Str}).\hat{P}(y)$ using fix:

$$f : \triangleright(\Pi(y{:}\mathsf{Str}).\hat{P}(y)) \to \Pi(y{:}\mathsf{Str}).\hat{P}(y)$$

$$f\;q\,(\langle x, xs\rangle) \triangleq \langle p(x), \lambda(\alpha{:}\mathsf{tick}).(q\,[\alpha])(xs\,[\alpha])\rangle$$

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

Equational Theory of Ticks

$$
\begin{array}{rcll}
(\lambda(\alpha : \mathsf{tick}).t)\,[\alpha'] & = & t\,[\alpha'/\alpha] & \\
\lambda(\alpha : \mathsf{tick}).t\,[\alpha] & = & t & \text{if } \alpha \notin \mathsf{fv}(t)
\end{array}
$$

Coinductive Types via Clock Quantification

Atkey & McBride. Productive Coprogramming with Guarded Recursion, ICFP 2013

Recall: Guarded Recursive Types

Guarded streams:

$$\mathsf{Str} \cong \mathsf{Nat} \times \triangleright\mathsf{Str}$$

functions of type $\mathsf{Str} \to \mathsf{Str}$ are causal.

Recall: Guarded Recursive Types

Guarded streams:

$$\mathsf{Str} \cong \mathsf{Nat} \times \triangleright \mathsf{Str}$$

functions of type $\mathsf{Str} \to \mathsf{Str}$ are causal.

Example

We can write a function that increments each element:

$$\mathsf{incr} : \mathsf{Str} \to \mathsf{Str}$$

$$\mathsf{incr} \triangleq \mathsf{fix}\, \lambda g.\lambda x : \mathsf{Str}.\, \langle \mathsf{suc}\,(\pi_1 x), g \circledast (\pi_2\, x)\rangle$$

but <u>not</u> a function that skips every other element

$$\mathsf{skipEven} : \mathsf{Str} \to \mathsf{Str}$$

Coinductive types via clock quantification

- $\triangleright$ annotated with clock variables κ: $\triangleright^{\kappa} A$
- quantification over clocks: $\forall \kappa . A$
- eliminate $\triangleright^{\kappa}$ via $\mathsf{force} : (\forall \kappa . \triangleright^{\kappa} A) \to \forall \kappa . A$

Example

$$\mathsf{Str} \;\; \cong \mathsf{Nat} \times \triangleright\, \mathsf{Str}$$

Atkey & McBride. Productive Coprogramming with Guarded Recursion, ICFP 2013

Coinductive types via clock quantification

- $\triangleright$ annotated with clock variables κ: $\triangleright^{\kappa} A$
- quantification over clocks: $\forall \kappa . A$
- eliminate $\triangleright^{\kappa}$ via $\mathsf{force} : (\forall \kappa . \triangleright^{\kappa} A) \rightarrow \forall \kappa . A$

Example

$$\mathsf{Str} \;\; \cong \mathsf{Nat} \times \triangleright^{\kappa} \mathsf{Str}$$

Atkey & McBride. Productive Coprogramming with Guarded Recursion, ICFP 2013

Coinductive types via clock quantification

- $\triangleright$ annotated with clock variables κ: $\triangleright^{\kappa} A$
- quantification over clocks: $\forall \kappa . A$
- eliminate $\triangleright^{\kappa}$ via $\mathsf{force} : (\forall \kappa . \triangleright^{\kappa} A) \to \forall \kappa . A$

Example

$$\mathsf{Str}^{\kappa} \cong \mathsf{Nat} \times \triangleright^{\kappa} \mathsf{Str}^{\kappa}$$

Atkey & McBride. Productive Coprogramming with Guarded Recursion, ICFP 2013

Coinductive types via clock quantification

- $\triangleright$ annotated with clock variables κ: $\triangleright^\kappa A$
- quantification over clocks: $\forall\kappa.A$
- eliminate $\triangleright^\kappa$ via force : $(\forall\kappa.\triangleright^\kappa A) \to \forall\kappa.A$

Example

$$\mathsf{Str}^\kappa \cong \mathsf{Nat} \times \triangleright^\kappa \mathsf{Str}^\kappa$$

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

Atkey & McBride. Productive Coprogramming with Guarded Recursion, ICFP 2013

Coinductive types via clock quantification

- $\triangleright$ annotated with clock variables κ: $\triangleright^\kappa A$
- quantification over clocks: $\forall\kappa.A$
- eliminate $\triangleright^\kappa$ via force : $(\forall\kappa.\triangleright^\kappa A) \to \forall\kappa.A$

Example

$$\mathsf{Str}^\kappa \cong \mathsf{Nat} \times \triangleright^\kappa \mathsf{Str}^\kappa$$

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

Functions of type $\mathsf{Str}_\mathsf{C} \to \mathsf{Str}_\mathsf{C}$ are productive.

e.g. $\quad$ skipEven : $\mathsf{Str}_\mathsf{C} \to \mathsf{Str}_\mathsf{C}$

Atkey & McBride. Productive Coprogramming with Guarded Recursion, ICFP 2013

Typing Rules for Clock Quantification

Typing judgement extended with a clock context Δ.

Typing Rules for Clock Quantification

Typing judgement extended with a clock context Δ.

$$\frac{\Gamma \vdash_{\Delta,\kappa} t : A \qquad \Gamma \vdash_{\Delta}}{\Gamma \vdash_{\Delta} \Lambda\kappa.t : \forall\kappa.A}$$

$$\frac{\Gamma \vdash_{\Delta} t : \forall\kappa.A \qquad \kappa' \in \Delta}{\Gamma \vdash_{\Delta} t[\kappa'] : A\left[\kappa'/\kappa\right]}$$

Typing Rules for Clock Quantification

Typing judgement extended with a clock context Δ.

$$\frac{\Gamma \vdash_{\Delta,\kappa} t : A \qquad \Gamma \vdash_{\Delta}}{\Gamma \vdash_{\Delta} \Lambda\kappa.t : \forall\kappa.A}$$

$$\frac{\Gamma \vdash_{\Delta} t : \forall\kappa.A \qquad \kappa' \in \Delta}{\Gamma \vdash_{\Delta} t[\kappa'] : A\,[\kappa'/\kappa]}$$

Clocks in CloTT

Each tick α now belongs to a certain clock κ, written $\alpha : \kappa$ (instead of just α : tick).

Clocks in CloTT

Each tick α now belongs to a certain clock κ, written $\alpha : \kappa$ (instead of just α : tick).

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

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

Tick constant

In order to obtain

$$\mathsf{force} : \forall\kappa.\triangleright^{\kappa} A \to \forall\kappa.A$$

we introduce the tick constant $\diamond$:

Tick constant

In order to obtain

$$\mathsf{force} : \forall\kappa.\triangleright^{\kappa} A \to \forall\kappa.A$$

we introduce the tick constant $\diamond$:

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

- $\diamond$ can only be used in a context without free occurrences of κ.

Tick constant

In order to obtain

$$\mathsf{force} : \forall\kappa.\triangleright^{\kappa} A \to \forall\kappa.A$$

we introduce the tick constant $\diamond$:

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

- $\diamond$ can only be used in a context without free occurrences of κ.

Tick constant

In order to obtain

$$\mathsf{force} : \forall\kappa.\triangleright^{\kappa} A \to \forall\kappa.A$$

we introduce the tick constant $\diamond$:

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

- $\diamond$ can only be used in a context without free occurrences of κ.
- force is definable in terms of $\diamond$:

$$\mathsf{force}\; x \triangleq \Lambda\kappa.(x\,[\kappa])\,[\diamond]$$

Tick constant

In order to obtain

$$\mathsf{force} : \forall\kappa.\triangleright^{\kappa} A \to \forall\kappa.A$$

we introduce the tick constant $\diamond$:

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

- $\diamond$ can only be used in a context without free occurrences of κ.
- force is definable in terms of $\diamond$:

$$\mathsf{force} : \forall\kappa.\triangleright(\alpha : \kappa).A \to \forall\kappa.A\,[\diamond/\alpha]$$

$$\mathsf{force}\ x \triangleq \Lambda\kappa.(x\,[\kappa])\,[\diamond]$$

Reduction Semantics

Guarded fixed points

$\mathsf{fix}^\kappa \colon (\triangleright^\kappa A \to A) \to A$

$$\mathsf{fix}^\kappa f = f(\mathsf{next}^\kappa(\mathsf{fix}^\kappa f))$$

We need to restrict fixed point unfolding to obtain strong normalisation (while retaining canonicity).

Guarded fixed points

$\mathsf{fix}^\kappa \colon (\triangleright^\kappa A \to A) \to A$

$$\mathsf{fix}^\kappa f = f(\mathsf{next}^\kappa(\mathsf{fix}^\kappa f))$$

We need to restrict fixed point unfolding to obtain strong normalisation (while retaining canonicity).

Delayed fixed point

- $\mathsf{dfix}^\kappa \colon (\triangleright^\kappa A \to A) \to \triangleright^\kappa A$
- only unfolds if applied to tick constant $\diamond$

$$\begin{array}{lcll} (\mathsf{dfix}^\kappa f)\,[\alpha] & \not\to & f(\mathsf{dfix}^\kappa f) & \text{if } \alpha \text{ is tick variable} \\ (\mathsf{dfix}^\kappa f)\,[\diamond] & \to & f(\mathsf{dfix}^\kappa f) & \end{array}$$

- $\mathsf{fix}^\kappa f \triangleq f\,(\mathsf{dfix}^\kappa\, f)$

Reduction Semantics

$$
\begin{aligned}
(\lambda(x : A).t)s &\to t\,[s/x] \\
(\lambda(\alpha : \kappa).t)\,[\beta] &\to t\,[\beta/\alpha] \\
\lambda(\alpha : \kappa).(t\,[\alpha]) &\to t && \text{if } \alpha \notin \mathsf{fv}(t) \\
(\Lambda\kappa.t)[\kappa'] &\to t\,[\kappa'/\kappa] \\
(\Lambda\kappa.t[\kappa]) &\to t && \text{if } \kappa \notin \mathsf{fv}(t) \\
(\mathsf{dfix}^{\kappa}\, t)\,[\diamond] &\to t\,(\mathsf{dfix}^{\kappa}\, t)
\end{aligned}
$$

Syntactic Properties of CloTT

Theorem (Decidable equality)

- *Reduction relation $\rightarrow$ is confluent.*
- *Well-typed terms are strongly normalising.*

Syntactic Properties of CloTT

Theorem (Decidable equality)

- *Reduction relation $\to$ is confluent.*
- *Well-typed terms are strongly normalising.*

Theorem (Canonicity)

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

Syntactic Properties of CloTT

Theorem (Decidable equality)

- *Reduction relation $\rightarrow$ is confluent.*
- *Well-typed terms are strongly normalising.*

Theorem (Canonicity)

If $\vdash_\Delta t : \mathsf{Nat}$*, then* $t \rightarrow^* \mathsf{suc}^n\, 0$ *for some* $n \in \mathbb{N}$*.*

Corollary (Productivity)

Given $\vdash_\Delta t : \mathsf{Str}_\mathsf{C}$*, any element of the stream* t *can be computed with a finite number of reduction steps.*

(via a term $\mathsf{nth} : \mathsf{Nat} \to \mathsf{Str}_\mathsf{C} \to \mathsf{Nat}$*)*

Denotational Semantics

B. Mannaa and R.E. Møgelberg *The clocks they are adjunctions* FSCD 2018

The topos of trees ($\mathbf{Set}^{\omega^{\mathrm{op}}}$)

$$
\begin{tikzcd}
X \arrow[d, "\mathrm{next}"'] & : & X(0) \arrow[d] & X(1) \arrow[l, "r_0"'] \arrow[d, "r_1"'] & X(2) \arrow[l, "r_1"'] \arrow[d, "r_2"'] & \cdots \arrow[l] \\
\blacktriangleright X & : & 1 & X(0) \arrow[l] & X(1) \arrow[l, "r_0"'] & \cdots \arrow[l, "r_1"']
\end{tikzcd}
$$

- Example: $\mathsf{Str} \cong \mathsf{Nat} \times \triangleright \mathsf{Str}$

$$
\mathbb{N} \times 1 \xleftarrow{\pi} \mathbb{N}^2 \times 1 \xleftarrow{\pi} \mathbb{N}^3 \times 1 \xleftarrow{\pi} \ldots
$$

- No object of ticks!

A left adjoint to ▶

- Define $\blacktriangleleft X(n) = X(n+1)$
- Maps

$$\begin{tikzcd}
X \arrow[d] & : & X(0) \arrow[d] & X(1) \arrow[l] \arrow[d] & X(2) \arrow[l] \arrow[d] & \dots \arrow[l] \\
\blacktriangleright Y & : & 1 & Y(0) \arrow[l] & Y(1) \arrow[l] & \dots \arrow[l]
\end{tikzcd}$$

- Correspond to maps

$$\begin{tikzcd}
\blacktriangleleft X \arrow[d] & : & X(1) \arrow[d] & X(2) \arrow[l] \arrow[d] & X(3) \arrow[l] \arrow[d] & \dots \arrow[l] \\
Y & : & Y(0) & Y(1) \arrow[l] & Y(2) \arrow[l] & \dots \arrow[l]
\end{tikzcd}$$

Interpreting ticks

- Define $[\![\Gamma, \alpha : \mathsf{tick}]\!] = \blacktriangleleft[\![\Gamma]\!]$
- Interpret rule

$$\frac{\Gamma, \alpha : \mathsf{tick} \vdash_\Delta A \text{ type}}{\Gamma \vdash_\Delta \triangleright(\alpha : \mathsf{tick}).A \text{ type}}$$

- as pullback

$$\begin{tikzcd}
{[\![\Gamma \vdash_\Delta \triangleright(\alpha : \mathsf{tick}).A \text{ type}]\!]} \arrow[r] \arrow[d] \arrow[dr, phantom, "\lrcorner", very near start] & \blacktriangleright[\![\Gamma, \alpha : \mathsf{tick} \vdash_\Delta A \text{ type}]\!] \arrow[d] \\
{[\![\Gamma]\!]} \arrow[r, "\eta"] & \blacktriangleright[\![\Gamma, \alpha : \mathsf{tick}]\!]
\end{tikzcd}$$

- Where η is the unit of the adjunction

Clocks model $\mathbf{Set}^{\mathbb{T}}$

- $\mathbb{T}$ objects: (Θ, ϑ)

$$\Theta \subset_{\text{fin}} \mathrm{CV} \qquad \vartheta : \Theta \to \mathbb{N}$$

- Morphisms

$$\begin{tikzcd} \Theta \arrow[rr, "\sigma"] \arrow[dr, "\vartheta"'] & \arrow[d, phantom, "\geq"] & \Theta' \arrow[dl, "\vartheta'"] \\ & \mathbb{N} & \end{tikzcd}$$

- Modelling clock contexts

$$[\![\Delta]\!](\Theta, \vartheta) = \Theta^{\Delta}$$

- $\mathrm{GR}[\Delta] = \mathbf{Set}^{\int [\![\Delta]\!]}$

Modelling ticks

- Judgements in clock context Δ modelled in $\mathrm{GR}[\Delta]$
- Adjunction for each $\kappa \in \Delta$

$$
\begin{tikzcd}
\mathrm{GR}[\Delta] \arrow[r, shift left, "\blacktriangleleft^{\kappa}"] \arrow[r, phantom, "\perp"] & \mathrm{GR}[\Delta] \arrow[l, shift left, "\blacktriangleright^{\kappa}"]
\end{tikzcd}
$$

- Right adjoint:

$$
(\overset{\kappa}{\blacktriangleright} X)(\Theta; \vartheta; f) = \begin{cases} X(\Theta; \vartheta[f(\kappa) \mapsto n]; f) & \vartheta(f(\kappa)) = n + 1 \\ 1 & \vartheta(f(\kappa)) = 0 \end{cases}
$$

- where $\vartheta : \Theta \to \mathbb{N}$, $f : \Delta \to \Theta$

The left adjoint

- First attempt not a presheaf

$$(\overset{\kappa}{\blacktriangleleft} X)(\Theta; \vartheta; f) = X(\Theta; \vartheta[f(\kappa) \mapsto \vartheta(f(\kappa)) + 1]; f)$$

- Definition must take into account all pasts

$$\overset{\kappa}{\blacktriangleleft} X(\Theta; \vartheta; f) = \coprod_{\kappa \in Y \subset f^{-1}(f(\kappa))} X((\Theta; \vartheta; f)[Y, \kappa+])$$

where

$$(\Theta; \vartheta; f)[Y, \kappa+] = (\Theta, \#_\Theta; \vartheta[\#_\Theta \mapsto \vartheta(f(\kappa))+1]; f[Y \mapsto \#_\Theta])$$

Summary

Clocked Type Theory (CloTT)

dependent type theory featuring

- coinductive types via clock quantification
- guarded recursive reasoning via ticks

Summary

Clocked Type Theory (CloTT)

dependent type theory featuring

- coinductive types via clock quantification
- guarded recursive reasoning via ticks

Semantics

We have

- a denotational semantics
- a reduction semantics with
  - confluence
  - strong normalisation
  - canonicity

What makes guarded types tick?

Syntax and Semantics for Type Theory with Guarded Recursion and Ticks

Patrick Bahr   Bassel Mannaa
Rasmus Møgelberg

IT University of Copenhagen

Bonus Slides

Typing Rules

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

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

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

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

$$\frac{\Gamma \vdash_{\Delta} t : \triangleright^{\kappa} A \to A}{\Gamma \vdash_{\Delta} \mathsf{dfix}^{\kappa}\, t : \triangleright^{\kappa} A}$$

Typing Rules (cont.)

$$\frac{\Gamma, x : A \vdash_\Delta t : B}{\Gamma \vdash_\Delta \lambda(x : A).t : \Pi(x : A).\, B}$$

$$\frac{\Gamma \vdash_\Delta t : \Pi(x : A).\, B \qquad \Gamma \vdash_\Delta u : A}{\Gamma \vdash_\Delta t\, u : B\,[u/x]}$$

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

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

Reduction Semantics

$$(\lambda x : A.t)s \to t\,[s/x]$$
$$(\lambda(\alpha' : \kappa).t)\,[\alpha] \to t\,[\alpha/\alpha']$$
$$(\Lambda\kappa.t[\kappa]) \to t$$
$$\mathrm{fold}_\diamond t \to t$$
$$\mathrm{if\ true}\ t_1\ t_2 \to t_1$$
$$\mathrm{rec}\,(\mathrm{suc}\ \ t_1)\ t_2\ t_3 \to t_3\ t_1\,(\mathrm{rec}\ t_1\ t_2\ t_3)$$
$$(\mathrm{dfix}^\kappa\ t)\,[\diamond] \to t\,(\mathrm{dfix}^\kappa\ t)$$

$$(\Lambda\kappa.t)[\kappa'] \to t\,[\kappa'/\kappa]$$
$$\lambda(\alpha : \kappa).(t\,[\alpha]) \to t$$
$$\pi_i\,\langle t_1, t_2\rangle \to t_i$$
$$\mathrm{unfold}_\diamond t \to t$$
$$\mathrm{if\ false}\ t_1\ t_2 \to t_2$$
$$\mathrm{rec}\,0\ t\ s \to t$$

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