# A Recipe for the Semantics of Reversible Programming

Louis Lemonnier
University of Edinburgh

THE UNIVERSITY of EDINBURGH
informatics

PARTOUT seminar, LIX. 12th November 2024

# Reversible Programming

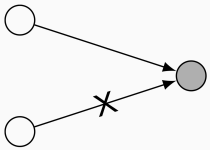
## Originally

- Landauer and Bennett, 1961: Reversible Computation and Energy Dissipation.
- Reversible programs: for a program $\mathtt{t}$, there is $\mathtt{t}^{-1}$ such that $\mathtt{t}; \mathtt{t}^{-1} = \mathtt{skip}$.
- Applications to quantum computing.

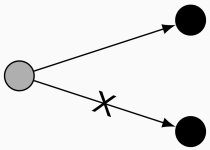
## What we do

- Reversibility, but not totality.
- Syntax for reversible functions.
- With enough expressivity.
- Through the categorical semantics.

Backward determinism



Forward determinism



[Kaarsgaard&Rennela21]

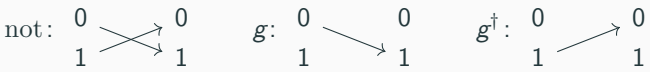
# A general framework: dagger categories

Origine: functional analysis where $\langle fx \mid y \rangle = \langle x \mid f^\dagger y \rangle$.

Category $\mathbf{C}$ equipped with a functor $(-)^\dagger \colon \mathbf{C}^{\mathrm{op}} \to \mathbf{C}$, such that:

- On objects, $A^\dagger = A$.
- On morphisms:
  - $(g \circ f)^\dagger = f^\dagger \circ g^\dagger$,
  - $f^{\dagger\dagger} = f$.

Example with partial injective functions between sets, here $\{0, 1\}$.




A very important class of morphisms: *partial* †*-isomorphism*. $ff^\dagger f = f$.

# Examples of relevant dagger categories

Sets and bijections.

# Examples of relevant dagger categories

Sets and bijections.


Sets and partial injections. $g$:


$g$ is undefined on 1.

# Examples of relevant dagger categories

Sets and bijections.


Sets and partial injections. $g$:


$g$ is undefined on 1.

Hilbert spaces and unitary maps.

$$|0\rangle \longrightarrow |+\rangle$$
$$|1\rangle \longrightarrow |-\rangle$$

where

$$|0\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \qquad |1\rangle = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \qquad |+\rangle = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix} \qquad |-\rangle = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ -\frac{1}{\sqrt{2}} \end{bmatrix}$$

# Examples of relevant dagger categories

Sets and bijections. $\begin{matrix} 0 & & 0 \\ & \searrow\!\nearrow & \\ 1 & & 1 \end{matrix}$

Sets and partial injections. $g\colon \begin{matrix} 0 & & 0 \\ & \searrow & \\ 1 & & 1 \end{matrix}$ $g$ is undefined on 1.

Hilbert spaces and unitary maps. $\begin{matrix} |0\rangle \longrightarrow |+\rangle \\ |1\rangle \longrightarrow |-\rangle \end{matrix}$ where

$$|0\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \qquad |1\rangle = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \qquad |+\rangle = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix} \qquad |-\rangle = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ -\frac{1}{\sqrt{2}} \end{bmatrix}$$

Hilbert spaces and contractions. $h\colon \begin{matrix} |0\rangle \longrightarrow |+\rangle \\ |1\rangle \qquad\quad |-\rangle \end{matrix}$ $h\,|1\rangle = 0.$

# How do you extract syntax from dagger categories?

## Cartesian closed category

# How do you extract syntax from dagger categories?

## Cartesian closed category

- Cartesian product $\times$.

# How do you extract syntax from dagger categories?

## Cartesian closed category

- Cartesian product $\times$.
  - Type $A \times B$.

# How do you extract syntax from dagger categories?

## Cartesian closed category

- Cartesian product $\times$.
  - Type $A \times B$.
  - Constructor $\langle t_1, t_2 \rangle$.

# How do you extract syntax from dagger categories?

## Cartesian closed category

- Cartesian product $\times$.
  - Type $A \times B$.
  - Constructor $\langle t_1, t_2 \rangle$.
- Right adjoint to the tensor $\to$.

# How do you extract syntax from dagger categories?

## Cartesian closed category

- Cartesian product $\times$.
  - Type $A \times B$.
  - Constructor $\langle t_1, t_2 \rangle$.
- Right adjoint to the tensor $\to$.
  - Type $A \to B$.

# How do you extract syntax from dagger categories?

## Cartesian closed category

- Cartesian product $\times$.
  - Type $A \times B$.
  - Constructor $\langle t_1, t_2 \rangle$.
- Right adjoint to the tensor $\to$.
  - Type $A \to B$.
  - Constructor $\lambda x.t$.

# How do you extract syntax from dagger categories?

## Cartesian closed category

- Cartesian product $\times$.
  - Type $A \times B$.
  - Constructor $\langle t_1, t_2 \rangle$.
- Right adjoint to the tensor $\to$.
  - Type $A \to B$.
  - Constructor $\lambda x.t$.

## Dagger category

# How do you extract syntax from dagger categories?

## Cartesian closed category

- Cartesian product $\times$.
  - Type $A \times B$.
  - Constructor $\langle t_1, t_2 \rangle$.
- Right adjoint to the tensor $\to$.
  - Type $A \to B$.
  - Constructor $\lambda x.t$.

## Dagger category

- Not cartesian, but often monoidal.

# How do you extract syntax from dagger categories?

## Cartesian closed category

- Cartesian product $\times$.
  - Type $A \times B$.
  - Constructor $\langle t_1, t_2 \rangle$.
- Right adjoint to the tensor $\to$.
  - Type $A \to B$.
  - Constructor $\lambda x.t$.

## Dagger category

- Not cartesian, but often monoidal.
  - Type $A \otimes B$.

# How do you extract syntax from dagger categories?

## Cartesian closed category

- Cartesian product $\times$.
  - Type $A \times B$.
  - Constructor $\langle t_1, t_2 \rangle$.
- Right adjoint to the tensor $\to$.
  - Type $A \to B$.
  - Constructor $\lambda x.t$.

## Dagger category

- Not cartesian, but often monoidal.
  - Type $A \otimes B$.
  - Linear type system.

# How do you extract syntax from dagger categories?

## Cartesian closed category

- Cartesian product $\times$.
  - Type $A \times B$.
  - Constructor $\langle t_1, t_2 \rangle$.
- Right adjoint to the tensor $\to$.
  - Type $A \to B$.
  - Constructor $\lambda x.t$.

## Dagger category

- Not cartesian, but often monoidal.
  - Type $A \otimes B$.
  - Linear type system.
- Not monoidal closed.

# How do you extract syntax from dagger categories?

## Cartesian closed category

- Cartesian product $\times$.
  - Type $A \times B$.
  - Constructor $\langle t_1, t_2 \rangle$.
- Right adjoint to the tensor $\to$.
  - Type $A \to B$.
  - Constructor $\lambda x.t$.

## Dagger category

- Not cartesian, but often monoidal.
  - Type $A \otimes B$.
  - Linear type system.
- Not monoidal closed.
  - No ground function type.

# How do you extract syntax from dagger categories?

## Cartesian closed category

- Cartesian product $\times$.
  - Type $A \times B$.
  - Constructor $\langle t_1, t_2 \rangle$.
- Right adjoint to the tensor $\to$.
  - Type $A \to B$.
  - Constructor $\lambda x.t$.

## Dagger category

- Not cartesian, but often monoidal.
  - Type $A \otimes B$.
  - Linear type system.
- Not monoidal closed.
  - No ground function type.
  - Is there a way to form functions?

# How do you extract syntax from dagger categories?

## Cartesian closed category

- Cartesian product $\times$.
  - Type $A \times B$.
  - Constructor $\langle t_1, t_2 \rangle$.
- Right adjoint to the tensor $\to$.
  - Type $A \to B$.
  - Constructor $\lambda x.t$.

## Dagger category

- Not cartesian, but often monoidal.
  - Type $A \otimes B$.
  - Linear type system.
- Not monoidal closed.
  - No ground function type.
  - Is there a way to form functions?

Hopefully, there is another way.

# Our new functions

We can cheat with our partial inverse!

# Our new functions

We can cheat with our partial inverse!

$$\begin{array}{lcl} [\![\Delta \vdash t\colon A]\!] & : & [\![\Delta]\!] \to [\![A]\!] \\ [\![\Delta \vdash t\colon A]\!]^{\dagger} & : & [\![A]\!] \to [\![\Delta]\!] \end{array}$$

# Our new functions

We can cheat with our partial inverse!

$$\begin{array}{lcl} [\![\Delta \vdash t\colon A]\!] & : & [\![\Delta]\!] \to [\![A]\!] \\ [\![\Delta \vdash t\colon A]\!]^{\dagger} & : & [\![A]\!] \to [\![\Delta]\!] \end{array}$$

What do we do with this?

# Our new functions

We can cheat with our partial inverse!

$$\begin{array}{lcl} [\![\Delta \vdash t \colon A]\!] & : & [\![\Delta]\!] \to [\![A]\!] \\ [\![\Delta \vdash t \colon A]\!]^{\dagger} & : & [\![A]\!] \to [\![\Delta]\!] \end{array}$$

What do we do with this?

Given $\quad \Delta \vdash t \colon A \qquad \Delta \vdash t' \colon B$

We form a function $t \mapsto t' \colon A \leftrightarrow B$, Whose semantics is

# Our new functions

We can cheat with our partial inverse!

$$\begin{array}{lcl} [\![\Delta \vdash t\colon A]\!] & : & [\![\Delta]\!] \to [\![A]\!] \\ [\![\Delta \vdash t\colon A]\!]^{\dagger} & : & [\![A]\!] \to [\![\Delta]\!] \end{array}$$

What do we do with this?

Given $\quad \Delta \vdash t\colon A \qquad \Delta \vdash t'\colon B$

We form a function $t \mapsto t'\colon A \leftrightarrow B$, Whose semantics is

$$[\![A]\!] \overset{[\![\Delta \vdash t\,:\,A]\!]^{\dagger}}{\longrightarrow} [\![\Delta]\!] \overset{[\![\Delta \vdash t'\,:\,B]\!]}{\longrightarrow} [\![B]\!]$$

# Our new functions

We can cheat with our partial inverse!

$$\begin{array}{lcl} [\![\Delta \vdash t\colon A]\!] & : & [\![\Delta]\!] \to [\![A]\!] \\ [\![\Delta \vdash t\colon A]\!]^{\dagger} & : & [\![A]\!] \to [\![\Delta]\!] \end{array}$$

What do we do with this?

Given $\quad \Delta \vdash t\colon A \qquad \Delta \vdash t'\colon B$

We form a function $t \mapsto t'\colon A \leftrightarrow B$, Whose semantics is

$$[\![A]\!] \xrightarrow{[\![\Delta \vdash t\,:\,A]\!]^{\dagger}} [\![\Delta]\!] \xrightarrow{[\![\Delta \vdash t'\,:\,B]\!]} [\![B]\!]$$

This is a reversible function! We have $(t \mapsto t')^{-1} = t' \mapsto t$, whose semantics is:

# Our new functions

We can cheat with our partial inverse!

$$\begin{array}{lcl} [\![\Delta \vdash t\colon A]\!] & : & [\![\Delta]\!] \to [\![A]\!] \\ [\![\Delta \vdash t\colon A]\!]^\dagger & : & [\![A]\!] \to [\![\Delta]\!] \end{array}$$

What do we do with this?

Given $\quad \Delta \vdash t\colon A \qquad \Delta \vdash t'\colon B$

We form a function $t \mapsto t'\colon A \leftrightarrow B$, Whose semantics is

$$[\![A]\!] \xrightarrow{[\![\Delta \vdash t\,:\,A]\!]^\dagger} [\![\Delta]\!] \xrightarrow{[\![\Delta \vdash t'\,:\,B]\!]} [\![B]\!]$$

This is a reversible function! We have $(t \mapsto t')^{-1} = t' \mapsto t$, whose semantics is:

$$[\![B]\!] \xrightarrow{[\![\Delta \vdash t'\,:\,B]\!]^\dagger} [\![\Delta]\!] \xrightarrow{[\![\Delta \vdash t\,:\,A]\!]} [\![A]\!]$$

# Together with pattern-matching

With a sum type $\oplus$:

$$\frac{\Delta \vdash t\colon A}{\Delta \vdash \mathtt{inj}_l\ t\colon A \oplus B} \qquad \frac{\Delta \vdash t\colon B}{\Delta \vdash \mathtt{inj}_r\ t\colon A \oplus B}$$

# Together with pattern-matching

With a sum type $\oplus$:

$$\frac{\Delta \vdash t\colon A}{\Delta \vdash \mathtt{inj}_l\ t\colon A \oplus B} \qquad \frac{\Delta \vdash t\colon B}{\Delta \vdash \mathtt{inj}_r\ t\colon A \oplus B}$$

We introduce orthogonality (to ensure reversibility):

# Together with pattern-matching

With a sum type $\oplus$:

$$\frac{\Delta \vdash t : A}{\Delta \vdash \mathtt{inj}_l\ t : A \oplus B} \qquad \frac{\Delta \vdash t : B}{\Delta \vdash \mathtt{inj}_r\ t : A \oplus B}$$

We introduce orthogonality (to ensure reversibility):

$$\frac{}{\mathtt{inj}_l\ t_1 \perp \mathtt{inj}_r\ t_2} \qquad \frac{t_1 \perp t_2}{C[t_1] \perp C[t_2]} \qquad \text{with} \begin{cases} [\![t_1]\!]^\dagger \circ [\![t_1]\!] = \mathrm{id} \\ [\![t_1]\!]^\dagger \circ [\![t_2]\!] = 0 \quad \text{when } t_1 \perp t_2 \end{cases}$$

# Together with pattern-matching

With a sum type $\oplus$:

$$\frac{\Delta \vdash t\colon A}{\Delta \vdash \mathtt{inj}_l\ t\colon A \oplus B} \qquad \frac{\Delta \vdash t\colon B}{\Delta \vdash \mathtt{inj}_r\ t\colon A \oplus B}$$

We introduce orthogonality (to ensure reversibility):

$$\frac{}{\mathtt{inj}_l\ t_1 \perp \mathtt{inj}_r\ t_2} \qquad \frac{t_1 \perp t_2}{C[t_1] \perp C[t_2]} \qquad \text{with} \begin{cases} [\![t_1]\!]^\dagger \circ [\![t_1]\!] = \mathrm{id} \\ [\![t_1]\!]^\dagger \circ [\![t_2]\!] = 0 \quad \text{when } t_1 \perp t_2 \end{cases}$$

Our functions are then:

$$\left\{ \begin{array}{ccc} t_1 & \mapsto & t'_1 \\ t_2 & \mapsto & t'_2 \\ & \vdots & \\ t_m & \mapsto & t'_m \end{array} \right\} : A \leftrightarrow B$$

whenever $\Delta_i \vdash t_i\colon A$ and $t_j \perp t_k$, $\quad \Delta_i \vdash t'_i\colon B$ and $t'_j \perp t'_k$.

# Example and semantics

$$\left\{ \begin{array}{lcl} \mathtt{inj}_l\, x & \mapsto & \mathtt{inj}_r\, x \\ \mathtt{inj}_r\, y & \mapsto & \mathtt{inj}_l\, y \end{array} \right\} : A \oplus B \leftrightarrow B \oplus A$$

Denotational semantics:

# Example and semantics

$$\left\{ \begin{array}{lcl} \mathtt{inj}_l\, x & \mapsto & \mathtt{inj}_r\, x \\ \mathtt{inj}_r\, y & \mapsto & \mathtt{inj}_l\, y \end{array} \right\} : A \oplus B \leftrightarrow B \oplus A$$

Denotational semantics:

$$\left[\!\!\left[ \left\{ \begin{array}{lcl} \mathtt{inj}_l\, x & \mapsto & \mathtt{inj}_r\, x \\ \mathtt{inj}_r\, y & \mapsto & \mathtt{inj}_l\, y \end{array} \right\} : A \oplus B \leftrightarrow B \oplus A \right]\!\!\right] = \left( \begin{array}{ll} [\![A]\!] & [\![B]\!] \\ [\![B]\!] & [\![A]\!] \end{array} \right) + \left( \begin{array}{ll} [\![A]\!] & [\![B]\!] \\ [\![B]\!] & [\![A]\!] \end{array} \right)$$

Operational semantics:

# Example and semantics

$$\left\{ \begin{array}{lcl} \mathtt{inj}_l\, x & \mapsto & \mathtt{inj}_r\, x \\ \mathtt{inj}_r\, y & \mapsto & \mathtt{inj}_l\, y \end{array} \right\} : A \oplus B \leftrightarrow B \oplus A$$

Denotational semantics:

$$\left[\!\!\left[ \left\{ \begin{array}{lcl} \mathtt{inj}_l\, x & \mapsto & \mathtt{inj}_r\, x \\ \mathtt{inj}_r\, y & \mapsto & \mathtt{inj}_l\, y \end{array} \right\} : A \oplus B \leftrightarrow B \oplus A \right]\!\!\right] = \left( \begin{array}{ccc} [\![A]\!] & & [\![B]\!] \\ [\![B]\!] & \searrow & [\![A]\!] \end{array} \right) + \left( \begin{array}{ccc} [\![A]\!] & & [\![B]\!] \\ [\![B]\!] & \nearrow & [\![A]\!] \end{array} \right)$$

Operational semantics:

$$\left\{ \begin{array}{lcl} \mathtt{inj}_l\, x & \mapsto & \mathtt{inj}_r\, x \\ \mathtt{inj}_r\, x & \mapsto & \mathtt{inj}_l\, x \end{array} \right\}\ \mathtt{inj}_r\, v \rightarrow$$

# Example and semantics

$$\left\{ \begin{array}{lcl} \mathtt{inj}_l\, x & \mapsto & \mathtt{inj}_r\, x \\ \mathtt{inj}_r\, y & \mapsto & \mathtt{inj}_l\, y \end{array} \right\} : A \oplus B \leftrightarrow B \oplus A$$

Denotational semantics:

$$\left[\!\!\left[ \left\{ \begin{array}{lcl} \mathtt{inj}_l\, x & \mapsto & \mathtt{inj}_r\, x \\ \mathtt{inj}_r\, y & \mapsto & \mathtt{inj}_l\, y \end{array} \right\} : A \oplus B \leftrightarrow B \oplus A \right]\!\!\right] = \left( \begin{array}{ll} [\![A]\!] & [\![B]\!] \\ [\![B]\!] & [\![A]\!] \end{array} \right) + \left( \begin{array}{ll} [\![A]\!] & [\![B]\!] \\ [\![B]\!] & [\![A]\!] \end{array} \right)$$

Operational semantics:

$$\left\{ \begin{array}{lcl} \mathtt{inj}_l\, x & \mapsto & \mathtt{inj}_r\, x \\ \mathtt{inj}_r\, x & \mapsto & \mathtt{inj}_l\, x \end{array} \right\} \; \mathtt{inj}_r\, v \to (\mathtt{inj}_l\, x)[v/x] \to$$

# Example and semantics

$$\left\{ \begin{array}{lcl} \mathtt{inj}_l\, x & \mapsto & \mathtt{inj}_r\, x \\ \mathtt{inj}_r\, y & \mapsto & \mathtt{inj}_l\, y \end{array} \right\} : A \oplus B \leftrightarrow B \oplus A$$

Denotational semantics:

$$\left[\!\!\left[ \left\{ \begin{array}{lcl} \mathtt{inj}_l\, x & \mapsto & \mathtt{inj}_r\, x \\ \mathtt{inj}_r\, y & \mapsto & \mathtt{inj}_l\, y \end{array} \right\} : A \oplus B \leftrightarrow B \oplus A \right]\!\!\right] = \left( \begin{array}{ll} [\![A]\!] & [\![B]\!] \\ [\![B]\!] & [\![A]\!] \end{array} \right) + \left( \begin{array}{ll} [\![A]\!] & [\![B]\!] \\ [\![B]\!] & [\![A]\!] \end{array} \right)$$

Operational semantics:

$$\left\{ \begin{array}{lcl} \mathtt{inj}_l\, x & \mapsto & \mathtt{inj}_r\, x \\ \mathtt{inj}_r\, x & \mapsto & \mathtt{inj}_l\, x \end{array} \right\} \; \mathtt{inj}_r\, v \to (\mathtt{inj}_l\, x)[v/x] \to \mathtt{inj}_l\, v$$

# The mathematical recipe /ˈɹɛs.ɹ.pi/

Our category **C** such that:

# The mathematical recipe /ˈɹɛs.ɪ.pi/

Our category **C** such that:

- Inverse category.
  - Partial inverse $(-)^{\dagger}$.

# The mathematical recipe /ˈɹɛs.ɪ.pi/

Our category **C** such that:

- Inverse category.
  - Partial inverse $(-)^{\dagger}$.
  - Takes care of pattern-matching.

# The mathematical recipe /ˈɹɛs.ɹ.pi/

Our category **C** such that:

- Inverse category.
  - Partial inverse $(-)^\dagger$.
  - Takes care of pattern-matching.
- Rig structure.
  - Usual monoidal product $\otimes$.

# The mathematical recipe /ˈɹɛs.ɪ.pi/

Our category **C** such that:

- Inverse category.
  - Partial inverse $(-)^\dagger$.
  - Takes care of pattern-matching.
- Rig structure.
  - Usual monoidal product $\otimes$.
  - Disjointness tensor $\oplus$ with jointly epic injections.

# The mathematical recipe /ˈɪɛs.ɪ.pi/

Our category **C** such that:

- Inverse category.
  - Partial inverse $(-)^\dagger$.
  - Takes care of pattern-matching.
- Rig structure.
  - Usual monoidal product $\otimes$.
  - Disjointness tensor $\oplus$ with jointly epic injections.
- Join structure.
  - Compatible morphisms on their domain and codomain admit a join.

# The mathematical recipe /ˈɪɛs.ɪ.pi/

Our category **C** such that:

- Inverse category.
  - Partial inverse $(-)^\dagger$.
  - Takes care of pattern-matching.
- Rig structure.
  - Usual monoidal product $\otimes$.
  - Disjointness tensor $\oplus$ with jointly epic injections.
- Join structure.
  - Compatible morphisms on their domain and codomain admit a join.
  - Sometimes, provides a nice structure on morphisms.

Examples:

- Sets and partial injective functions **PInj**.
- Hilbert spaces and contractions **Contr** (sometimes written $\mathbf{Hilb}_{\leq 1}$).

## The case of inverse categories (such as PInj)

# To Infinity and Beyond

Some reading: [Axelsen&Kaarsgaard16] + [Fiore04] + some calculations.

$\longrightarrow$ A suitable inverse category $\mathbf{C}$

# To Infinity and Beyond

Some reading: [Axelsen&Kaarsgaard16] + [Fiore04] + some calculations.

$\longrightarrow$ A suitable inverse category $\mathbf{C}$ is parameterised **DCPO**-algebraically $\omega$-compact.

# To Infinity and Beyond

Some reading: [Axelsen&Kaarsgaard16] + [Fiore04] + some calculations.

$\longrightarrow$ A suitable inverse category $\mathbf{C}$ can model infinite data types $\mu X.A$.

# To Infinity and Beyond

Some reading: [Axelsen&Kaarsgaard16] + [Fiore04] + some calculations.

$\longrightarrow$ A suitable inverse category $\mathbf{C}$ can model infinite data types $\mu X.A$.

Examples:

$$\texttt{Nat} = \mu X.1 \oplus X$$

# To Infinity and Beyond

Some reading: [Axelsen&Kaarsgaard16] + [Fiore04] + some calculations.

$\longrightarrow$ A suitable inverse category $\mathbf{C}$ can model infinite data types $\mu X.A$.

Examples:

$$\texttt{Nat} = \mu X.1 \oplus X \qquad [A] = \mu X.1 \oplus (A \otimes X)$$

And we want to parse those infinite types:

# To Infinity and Beyond

Some reading: [Axelsen&Kaarsgaard16] + [Fiore04] + some calculations.

$\longrightarrow$ A suitable inverse category $\mathbf{C}$ can model infinite data types $\mu X.A$.

Examples:

$$\mathtt{Nat} = \mu X.1 \oplus X \qquad [A] = \mu X.1 \oplus (A \otimes X)$$

And we want to parse those infinite types:

$$\mathtt{map}(\omega) = \textbf{fix}\ \mathsf{f}.\ \left\{ \begin{array}{ll} [\,] & \mapsto [\,] \\ h :: t & \mapsto (\omega\ h) :: (\mathsf{f}\ t) \end{array} \right\} : [A] \leftrightarrow [B]$$

# To Infinity and Beyond

Some reading: [Axelsen&Kaarsgaard16] + [Fiore04] + some calculations.

$\longrightarrow$ A suitable inverse category $\mathbf{C}$ can model infinite data types $\mu X.A$.

Examples:

$$\mathtt{Nat} = \mu X.1 \oplus X \qquad [A] = \mu X.1 \oplus (A \otimes X)$$

And we want to parse those infinite types:

$$\mathtt{map}(\omega) = \textbf{fix}\ \mathsf{f}.\ \left\{ \begin{array}{ll} [\,] & \mapsto [\,] \\ h :: t & \mapsto (\omega\ h) :: (\ \mathsf{f}\ \ t) \end{array} \right\} : [A] \leftrightarrow [B]$$

Works in inverse categories thanks to **DCPO**-enrichment: $g \leq f$ defined as $fg^\dagger g = g$.

# To Infinity and Beyond

Some reading: [Axelsen&Kaarsgaard16] + [Fiore04] + some calculations.

$\longrightarrow$ A suitable inverse category $\mathbf{C}$ can model infinite data types $\mu X.A$.

Examples:

$$\texttt{Nat} = \mu X.1 \oplus X \qquad [A] = \mu X.1 \oplus (A \otimes X)$$

And we want to parse those infinite types:

$$\texttt{map}(\omega) = \textbf{fix}\ \texttt{f}.\ \left\{ \begin{array}{ll} [\,] & \mapsto [\,] \\ h :: t & \mapsto (\omega\ h) :: (\texttt{f}\ t) \end{array} \right\} : [A] \leftrightarrow [B]$$

Works in inverse categories thanks to **DCPO**-enrichment: $g \leq f$ defined as $fg^\dagger g = g$.

$$\boxed{\text{fix}(F) = \sup_n \{F^n(\bot)\}}$$

# Summary of the language (mandatory slide)

$$
\begin{array}{lllll}
\text{(Ground types)} & A, B & ::= & \mathtt{I} \mid A \oplus B \mid A \otimes B \mid \\
\text{(Function types)} & T_1, T_2 & ::= & A \leftrightarrow B \mid \\
 & & & \\
\text{(Unit term)} & t, t_1, t_2 & ::= & * \\
\text{(Pairing)} & & & \mid t_1 \otimes t_2 \\
\text{(Injections)} & & & \mid \mathtt{inj}_l\ t \mid \mathtt{inj}_r\ t \\
\text{(Function application)} & & & \mid \omega\ t \\
 & & & \\
\text{(Abstraction)} & \omega & ::= & \{t_1 \mapsto t_1' \mid \cdots \mid t_m \mapsto t_m'\}
\end{array}
$$

# Summary of the language (mandatory slide)

$$
\begin{array}{lllll}
\text{(Ground types)} & A, B & ::= & \mathtt{I} \mid A \oplus B \mid A \otimes B \mid X \mid \mu X.A \\
\text{(Function types)} & T_1, T_2 & ::= & A \leftrightarrow B \mid \\
 & & & \\
\text{(Unit term)} & t, t_1, t_2 & ::= & * \\
\text{(Pairing)} & & & \mid t_1 \otimes t_2 \\
\text{(Injections)} & & & \mid \mathtt{inj}_l\ t \mid \mathtt{inj}_r\ t \\
\text{(Function application)} & & & \mid \omega\ t \\
\text{(Inductive terms)} & & & \mid \mathtt{fold}\ t \\
 & & & \\
\text{(Abstraction)} & \omega & ::= & \{t_1 \mapsto t_1' \mid \cdots \mid t_m \mapsto t_m'\} \\
\text{(Fixed points)} & & & \mid f \mid \textbf{fix}\ f.\omega
\end{array}
$$

# Summary of the language (mandatory slide)

$$
\begin{array}{lllll}
\text{(Ground types)} & A, B & ::= & \mathtt{I} \mid A \oplus B \mid A \otimes B \mid X \mid \mu X.A \\
\text{(Function types)} & T_1, T_2 & ::= & A \leftrightarrow B \mid T_1 \to T_2 \\
 & & & \\
\text{(Unit term)} & t, t_1, t_2 & ::= & * \\
\text{(Pairing)} & & & \mid t_1 \otimes t_2 \\
\text{(Injections)} & & & \mid \mathtt{inj}_l\ t \mid \mathtt{inj}_r\ t \\
\text{(Function application)} & & & \mid \omega\ t \\
\text{(Inductive terms)} & & & \mid \mathtt{fold}\ t \\
 & & & \\
\text{(Abstraction)} & \omega & ::= & \{t_1 \mapsto t_1' \mid \cdots \mid t_m \mapsto t_m'\} \\
\text{(Fixed points)} & & & \mid f \mid \mathbf{fix}\ f.\omega \\
\text{(Higher abstractions)} & & & \mid \lambda f.\omega \mid \omega_2\omega_1
\end{array}
$$

$\lambda$-calculus with fixed points thanks to **DCPO**-enrichment.

The language is Turing complete! (even if it is reversible)

# The language is Turing complete! (even if it is reversible)

⟵ ask this guy (Kostia Chardonnet, currently works in Nancy)

# The language is Turing complete! (even if it is reversible)

⟵ ask this guy (Kostia Chardonnet, currently works in Nancy)

## Roughly

- Reversible Turing Machines [Axelsen&Glück11].
  - Simulate your favourite Turing machines.
- Encode RTMs in our language:
  - Alphabet & states mapped to $\mathtt{I} \oplus \cdots \oplus \mathtt{I}$.
  - Tape as lists.
  - Functions simulating one-step transition of $\delta$.
  - Iterate until final state.

# A variety of diverging functions

## Usual

$$\mathsf{fix}\ f^T\ .\ f^T : T$$

# A variety of diverging functions

## Usual

$$\mathsf{fix}\ f^{T}\ .\ f^{T} \colon T$$

## Less usual

$$\mathsf{fix}\ f^{A \leftrightarrow B}\ .\ \{x \mapsto f\ x\} \colon A \leftrightarrow B$$

# A variety of diverging functions

## Usual

$$\mathsf{fix}\ f^{T}\ .\ f^{T} \colon T$$

## Less usual

$$\mathsf{fix}\ f^{A \leftrightarrow B}\ .\ \{x \mapsto f\,x\} \colon A \leftrightarrow B$$

$$\begin{array}{rcl} (\mathsf{fix}\ f\ .\ \{x \mapsto f\,x\})\ v & \to & \{x \mapsto (\mathsf{fix}\ f\ .\ \{x \mapsto f\,x\})\ x\}\ v \\ & \to & ((\mathsf{fix}\ f\ .\ \{x \mapsto f\,x\})\ x)[v/x] \\ & = & (\mathsf{fix}\ f\ .\ \{x \mapsto f\,x\})\ v \end{array}$$

# A variety of diverging functions

## Usual

$$\text{fix } f^T \,.\, f^T \colon T$$

## Less usual

$$\text{fix } f^{A \leftrightarrow B} \,.\, \{x \mapsto f\, x\} \colon A \leftrightarrow B$$

$$\begin{array}{rcl} (\text{fix } f \,.\, \{x \mapsto f\, x\})\ v & \to & \{x \mapsto (\text{fix } f \,.\, \{x \mapsto f\, x\})\ x\}\ v \\ & \to & ((\text{fix } f \,.\, \{x \mapsto f\, x\})\ x)[v/x] \\ & = & (\text{fix } f \,.\, \{x \mapsto f\, x\})\ v \end{array}$$

## New challenger

# A variety of diverging functions

## Usual

$$\mathsf{fix}\ f^{T}\ .\ f^{T} \colon T$$

## Less usual

$$\mathsf{fix}\ f^{A \leftrightarrow B}\ .\ \{x \mapsto f\,x\} \colon A \leftrightarrow B$$

$$\begin{array}{lcl} (\mathsf{fix}\ f\ .\ \{x \mapsto f\,x\})\ v & \to & \{x \mapsto (\mathsf{fix}\ f\ .\ \{x \mapsto f\,x\})\ x\}\ v \\ & \to & ((\mathsf{fix}\ f\ .\ \{x \mapsto f\,x\})\ x)[v/x] \\ & = & (\mathsf{fix}\ f\ .\ \{x \mapsto f\,x\})\ v \end{array}$$

## New challenger

$$\lambda g^{A_1 \leftrightarrow B_1}\ .\ \mathsf{fix}\ f^{A_2 \leftrightarrow B_2}\ .\ \left\{ \begin{array}{lcl} \mathtt{inj}_l\,x & \mapsto & \mathtt{inj}_l\,(g\,x) \\ \mathtt{inj}_r\,y & \mapsto & \mathtt{inj}_r\,(f\,y) \end{array} \right\} \colon A_1 \oplus A_2 \leftrightarrow B_1 \oplus B_2$$

# The (pure) quantum case

# The quantum troubles

**Contr** (Hilbert spaces and contractive linear maps) is not enriched in an interesting way.

Composition does not preserve any reasonable poset structure.

# The quantum troubles

**Contr** (Hilbert spaces and contractive linear maps) is not enriched in an interesting way.

Composition does not preserve any reasonable poset structure.

Is there a better category?

# The quantum troubles

**Contr** (Hilbert spaces and contractive linear maps) is not enriched in an interesting way.

Composition does not preserve any reasonable poset structure.

Is there a better category? I don't know.

# The quantum troubles

**Contr** (Hilbert spaces and contractive linear maps) is not enriched in an interesting way.

Composition does not preserve any reasonable poset structure.

Is there a better category? I don't know.

Can we find a way to mimic the story we have for classical reversibility?

# The quantum troubles

**Contr** (Hilbert spaces and contractive linear maps) is not enriched in an interesting way.

Composition does not preserve any reasonable poset structure.

Is there a better category? I don't know.

Can we find a way to mimic the story we have for classical reversibility?

A kind of solution with techniques adapted from guarded recursion.

# Computation in the topos of trees

Objects in the topos of trees are cochains in **Set**:

$$X(0) \xleftarrow[r_0]{} X(1) \xleftarrow[r_1]{} X(2) \longleftarrow \cdots$$

There is a functor $L\colon \mathbf{Set}^{\mathbb{N}^{\mathrm{op}}} \to \mathbf{Set}^{\mathbb{N}^{\mathrm{op}}}$, such that $LX$ is:

$$1 \xleftarrow[!]{} X(0) \xleftarrow[r_0]{} X(1) \xleftarrow[r_1]{} X(2) \longleftarrow \cdots$$

and a natural transformation $\nu\colon \mathrm{id} \Rightarrow L$, such that $\nu_X$ is:

$$\begin{array}{ccccccccc} X(0) & \xleftarrow[r_0]{} & X(1) & \xleftarrow[r_1]{} & X(2) & \xleftarrow[r_2]{} & X(3) & \longleftarrow & \cdots \\ \downarrow{\scriptstyle !} & & \downarrow{\scriptstyle r_0} & & \downarrow{\scriptstyle r_1} & & \downarrow{\scriptstyle r_2} & & \\ 1 & \xleftarrow[!]{} & X(0) & \xleftarrow[r_0]{} & X(1) & \xleftarrow[r_1]{} & X(2) & \longleftarrow & \cdots \end{array}$$

and a family of morphisms $\mathrm{fix}_X\colon [LX \to X] \to X$.

# A guarded category

Start with a dagger category $\mathbf{C}$.

Consider the category whose objects are cochains in $\mathbf{C}$:

$$X(0) \longleftarrow X(1) \longleftarrow X(2) \longleftarrow \cdots$$

And morphisms are natural transformations in $\mathbf{C}$:

$$\begin{array}{ccccccc} X(0) & \longleftarrow & X(1) & \longleftarrow & X(2) & \longleftarrow & \cdots \\ \downarrow & & \downarrow & & \downarrow & & \\ Y(0) & \longleftarrow & Y(1) & \longleftarrow & Y(2) & \longleftarrow & \cdots \end{array}$$

This category is enriched in the topos of trees $\mathbf{Set}^{\mathbb{N}^{\mathrm{op}}}$ (with its fixed point operator).

# Some mild conditions for guarded recursion

Guarded recursion enforces termination.

# Some mild conditions for guarded recursion

Guarded recursion enforces termination.

It also enforces to *advance* in the depth of the terms.

# Some mild conditions for guarded recursion

Guarded recursion enforces termination.

It also enforces to *advance* in the depth of the terms.

The size of the output cannot be smaller than the size of the input (and vice versa).

# Some mild conditions for guarded recursion

Guarded recursion enforces termination.

It also enforces to *advance* in the depth of the terms.

The size of the output cannot be smaller than the size of the input (and vice versa).

It still allows for the $\mathrm{map}$ function.

$$\mathtt{map}(\omega) = \textbf{fix}\ \mathsf{f}.\ \left\{ \begin{array}{ll} [\,] & \mapsto [\,] \\ h :: t & \mapsto (\omega\ h) :: (\ \mathsf{f}\ \ t) \end{array} \right\} : [A] \leftrightarrow [B]$$

# Back to Hilbert spaces

Once we have this diagram in **Contr**:

$$\begin{array}{ccccccc} X(0) & \longleftarrow & X(1) & \longleftarrow & X(2) & \longleftarrow & \cdots \\ \downarrow & & \downarrow & & \downarrow & & \\ Y(0) & \longleftarrow & Y(1) & \longleftarrow & Y(2) & \longleftarrow & \cdots \end{array}$$

# Back to Hilbert spaces

Once we have this diagram in **Contr**:

$$\begin{array}{ccccccc} X(0) & \longleftarrow & X(1) & \longleftarrow & X(2) & \longleftarrow & \cdots \\ \downarrow & & \downarrow & & \downarrow & & \\ Y(0) & \longleftarrow & Y(1) & \longleftarrow & Y(2) & \longleftarrow & \cdots \end{array}$$

We can take the limit of both cochains.

$$\begin{array}{ccccccccc} X(0) & \longleftarrow & X(1) & \longleftarrow & X(2) & \longleftarrow & \cdots & \longleftarrow & X_\infty \\ \downarrow & & \downarrow & & \downarrow & & & & \downarrow \\ Y(0) & \longleftarrow & Y(1) & \longleftarrow & Y(2) & \longleftarrow & \cdots & \longleftarrow & Y_\infty \end{array}$$

# Back to Hilbert spaces

Once we have this diagram in **Contr**:

$$\begin{array}{ccccccc} X(0) & \longleftarrow & X(1) & \longleftarrow & X(2) & \longleftarrow & \cdots \\ \downarrow & & \downarrow & & \downarrow & & \\ Y(0) & \longleftarrow & Y(1) & \longleftarrow & Y(2) & \longleftarrow & \cdots \end{array}$$

We can take the limit of both cochains.

$$\begin{array}{ccccccccc} X(0) & \longleftarrow & X(1) & \longleftarrow & X(2) & \longleftarrow & \cdots & \longleftarrow & X_\infty \\ \downarrow & & \downarrow & & \downarrow & & & & \downarrow \\ Y(0) & \longleftarrow & Y(1) & \longleftarrow & Y(2) & \longleftarrow & \cdots & \longleftarrow & Y_\infty \end{array}$$

But we lose the enrichment (and therefore the nice calculus that we could have on top).

# Conclusion

Take home message: no cartesian closure needed to have

# Conclusion

Take home message: no cartesian closure needed to have functions,

# Conclusion

Take home message: no cartesian closure needed to have functions, inductive types,

# Conclusion

Take home message: no cartesian closure needed to have functions, inductive types, recursion.

# Conclusion

Take home message: no cartesian closure needed to have functions, inductive types, recursion.

The situation is trickier for (pure) quantum computation.

# Conclusion

Take home message: no cartesian closure needed to have functions, inductive types, recursion.

The situation is trickier for (pure) quantum computation.

Thank you!