The Reverse State Monad in Rocq (Work in Progress)

David Nowak (CNRS, Lille)
Vlad Rusu (Inria, Lille)

FROM Symposium, Iaşi, Romania, Sept. 17-19, 2025

Monads

- adding features (*effects*) to functional languages while keeping purity:
  - exceptions;
  - mutable state;
  - nondeterminism;
  - concurrency;
  - continuations;
  - ...
- pioneered by Haskell: rich library of monads and monad transformers;
- Reverse State Monad: effect is *backwards causality*;
- encoding the monad in Rocq - towards proving *reverse* programs `https://gitlab.inria.fr/haddock/revstate`.

Outline

Newcomb's Paradox

Game: player vs. host

- state of the game = 2 boxes : #1 transparent contains 1€; #2 opaque;
- rules of the game:
  - host discreetly (without player seeing) puts some money in box #2;
  - then player chooses: either both boxes, or just box #2;
- current game: host says is able to use backwards causality
  - if player's future choice = both boxes, host puts 1€ in box #2;
  - if player's future choice = box #2, host puts 1.000.000€ in it;
- strategy of the player determined by belief in backwards causality:
  - yes : choose box #2;
  - no : choose both boxes.

Newcomb's Paradox

Game: player vs. host

- state of the game = 2 boxes : #1 transparent contains 1€; #2 opaque;
- rules of the game:
  - host discreetly (without player seeing) puts some money in box #2;
  - then player chooses: either both boxes, or just box #2;
- current game: host says is able to use backwards causality
  - if player's future choice = both boxes, host puts 1€ in box #2;
  - if player's future choice = box #2, host puts 1.000.000€ in it;
- strategy of the player determined by belief in backwards causality:
  - yes : choose box #2;
  - no : choose both boxes.

Newcomb's Paradox

Game: player vs. host

- state of the game = 2 boxes : #1 transparent contains 1€; #2 opaque;
- rules of the game:
  - host discreetly (without player seeing) puts some money in box #2;
  - then player chooses: either both boxes, or just box #2;
- current game: host says is able to use backwards causality
  - if player's future choice = both boxes, host puts 1€ in box #2;
  - if player's future choice = box #2, host puts 1.000.000€ in it;
- strategy of the player determined by belief in backwards causality:
  - yes : choose box #2;
  - no : choose both boxes.

Newcomb's Paradox

Game: player vs. host

- state of the game = 2 boxes : #1 transparent contains 1€; #2 opaque;
- rules of the game:
  - host discreetly (without player seeing) puts some money in box #2;
  - then player chooses: either both boxes, or just box #2;
- current game: host says is able to use backwards causality
  - if player's future choice = both boxes, host puts 1€ in box #2;
  - if player's future choice = box #2, host puts 1.000.000€ in it;
- strategy of the player determined by belief in backwards causality:
  - yes : choose box #2;
  - no : choose both boxes.

Newcomb's Paradox

Game: player vs. host

- state of the game = 2 boxes : #1 transparent contains 1€; #2 opaque;
- rules of the game:
  - host discreetly (without player seeing) puts some money in box #2;
  - then player chooses: either both boxes, or just box #2;
- current game: host says is able to use backwards causality
  - if player's future choice = both boxes, host puts 1€ in box #2;
  - if player's future choice = box #2, host puts 1.000.000€ in it;
- strategy of the player determined by belief in backwards causality:
  - yes : choose box #2;
  - no : choose both boxes.

Newcomb's Paradox

Game: player vs. host

- state of the game = 2 boxes : #1 transparent contains 1€; #2 opaque;
- rules of the game:
  - host discreetly (without player seeing) puts some money in box #2;
  - then player chooses: either both boxes, or just box #2;
- current game: host says is able to use backwards causality
  - if player's future choice = both boxes, host puts 1€ in box #2;
  - if player's future choice = box #2, host puts 1.000.000€ in it;
- strategy of the player determined by belief in backwards causality:
  - yes : choose box #2;
  - no : choose both boxes.

Newcomb's Paradox

Game: player vs. host

- state of the game = 2 boxes : #1 transparent contains 1€; #2 opaque;
- rules of the game:
  - host discreetly (without player seeing) puts some money in box #2;
  - then player chooses: either both boxes, or just box #2;
- current game: host says is able to use backwards causality
  - if player's future choice = both boxes, host puts 1€ in box #2;
  - if player's future choice = box #2, host puts 1.000.000€ in it;
- strategy of the player determined by belief in backwards causality:
  - yes : choose box #2;
  - no : choose both boxes.

Backwards Causality with the Reverse State Monad

A more formal example:

- assume a *state* containing an infinite *Stream* over $\mathbb{N}$ with functions
  - $\_ :: \_ : \mathbb{N} \to \mathit{Stream} \to \mathit{Stream}$:
  - $\mathit{map} : (\mathbb{N} \to \mathbb{N}) \to \mathit{Stream} \to \mathit{Stream}$;
- $\mathit{do}\ x \leftarrow \mathit{get}$ reads the (whole) state & stores it in x;
- $\mathit{put}\ \ y$ changes the state to y.

Backwards Causality with the Reverse State Monad

A more formal example:

- assume a *state* containing an infinite *Stream* over $\mathbb{N}$ with functions
  - $\_ :: \_ : \mathbb{N} \to \mathit{Stream} \to \mathit{Stream}$:
  - $\mathit{map} : (\mathbb{N} \to \mathbb{N}) \to \mathit{Stream} \to \mathit{Stream}$;
- $\mathit{do}\ x \leftarrow \mathit{get}$ reads the (whole) state & stores it in x;
- $\mathit{put}\ y$ changes the state to y.

Backwards Causality with the Reverse State Monad

A more formal example:

- assume a *state* containing an infinite *Stream* over $\mathbb{N}$ with functions
  - $\_ :: \_ : \mathbb{N} \to \mathit{Stream} \to \mathit{Stream}$:
  - $\mathit{map} : (\mathbb{N} \to \mathbb{N}) \to \mathit{Stream} \to \mathit{Stream}$;
- $\mathit{do}\ x \leftarrow \mathit{get}$ reads the (whole) state & stores it in x;
- $\mathit{put}\ y$ changes the state to y.

Backwards Causality with the Reverse State Monad

What does $\mathit{do}\ x \leftarrow \mathit{get};\ \ \mathit{put}(0\ ::\ \mathit{map}\ (1+)\ x)$ do ?

- $\mathit{do}\ x \leftarrow \mathit{get}$ reads *future* state, *after* $\mathit{put}(0\ ::\ \mathit{map}\ (1+)\ x)$;
- hence state simultaneously contains x *and* $(0\ ::\ \mathit{map}\ (1+)\ x)$;
- hence $x = 0\ ::\ \mathit{map}\ (1+)\ x)$; 1-line program solves *fixpoint equation*, finds unique solution $x = 0\ ::\ 1\ ::\ 2\ ::\ \ldots$ there must be a trick!

  (. . . a fixpoint is hidden inside the program . . .)

- $\mathit{do}\ b \leftarrow \mathit{get};\ \ \mathit{put}\ \neg b$: equation $b = \neg b$ has no solution in Booleans . . . but has solution in *CPO* of Booleans;

- "fixpoint", "CPOs" : *domain theory* (& our library in Rocq)!

Backwards Causality with the Reverse State Monad

What does $do\ x \leftarrow get;\ \ put(0\ ::\ map\ (1+)\ x)$ do ?

- $do\ x \leftarrow get$ reads *future* state, *after* $put(0\ ::\ map\ (1+)\ x)$;
- hence state simultaneously contains x *and* $(0\ ::\ map\ (1+)\ x)$;
- hence $x = 0\ ::\ map\ (1+)\ x)$; 1-line program solves *fixpoint equation*, finds unique solution $x = 0\ ::\ 1\ ::\ 2\ ::\ \ldots$ there must be a trick!

  (. . . a fixpoint is hidden inside the program . . .)

- $do\ b \leftarrow get;\ \ put\ \neg b$: equation $b = \neg b$ has no solution in Booleans . . . but has solution in *CPO* of Booleans;
- "fixpoint", "CPOs" : *domain theory* (& our library in Rocq)!

Backwards Causality with the Reverse State Monad

What does $do\ x \leftarrow get;\ \ put(0\ ::\ map\ (1+)\ x)$ do ?

- $do\ x \leftarrow get$ reads *future* state, *after* $put(0\ ::\ map\ (1+)\ x)$;
- hence state simultaneously contains x *and* $(0\ ::\ map\ (1+)\ x)$;
- hence $x = 0\ ::\ map\ (1+)\ x)$; 1-line program solves *fixpoint equation*, finds unique solution $x = 0\ ::\ 1\ ::\ 2\ ::\ \ldots$ there must be a trick!
  (. . . a fixpoint is hidden inside the program . . .)
- $do\ b \leftarrow get;\ \ put\ \neg b$: equation $b = \neg b$ has no solution in Booleans . . . but has solution in *CPO* of Booleans;
- "fixpoint", "CPOs" : *domain theory* (& our library in Rocq)!

Backwards Causality with the Reverse State Monad

What does $\mathit{do}\ x \leftarrow \mathit{get};\ \ \mathit{put}(0\ ::\ \mathit{map}\ (1+)\ x)$ do ?

- $\mathit{do}\ x \leftarrow \mathit{get}$ reads *future* state, *after* $\mathit{put}(0\ ::\ \mathit{map}\ (1+)\ x)$;
- hence state simultaneously contains x *and* $(0\ ::\ \mathit{map}\ (1+)\ x)$;
- hence $x = 0\ ::\ \mathit{map}\ (1+)\ x)$; 1-line program solves *fixpoint equation*, finds unique solution $x = 0\ ::\ 1\ ::\ 2\ ::\ \ldots$ there must be a trick!

  (. . . a fixpoint is hidden inside the program . . .)

- $\mathit{do}\ b \leftarrow \mathit{get};\ \ \mathit{put}\ \neg b$: equation $b = \neg b$ has no solution in Booleans . . . but has solution in *CPO* of Booleans;
- "fixpoint", "CPOs" : *domain theory* (& our library in Rocq)!

Backwards Causality with the Reverse State Monad

What does $do\ x \leftarrow get;\ \ put(0\ ::\ map\ (1+)\ x)$ do ?

- $do\ x \leftarrow get$ reads *future* state, *after* $put(0\ ::\ map\ (1+)\ x)$;
- hence state simultaneously contains x *and* $(0\ ::\ map\ (1+)\ x)$;
- hence $x = 0\ ::\ map\ (1+)\ x)$; 1-line program solves *fixpoint equation*, finds unique solution $x = 0\ ::\ 1\ ::\ 2\ ::\ \ldots$ there must be a trick!

  (. . . a fixpoint is hidden inside the program . . .)

- $do\ b \leftarrow get;\ \ put\ \neg b$: equation $b = \neg b$ has no solution in Booleans . . . but has solution in *CPO* of Booleans;

- “fixpoint”, “CPOs” : *domain theory* (& our library in Rocq)!

Backwards Causality with the Reverse State Monad

What does $do\ x \leftarrow get;\ \ put(0\ ::\ map\ (1+)\ x)$ do ?

- $do\ x \leftarrow get$ reads *future* state, *after* $put(0\ ::\ map\ (1+)\ x)$;
- hence state simultaneously contains x *and* $(0\ ::\ map\ (1+)\ x)$;
- hence $x = 0\ ::\ map\ (1+)\ x)$; 1-line program solves *fixpoint equation*, finds unique solution $x = 0\ ::\ 1\ ::\ 2\ ::\ \ldots$ there must be a trick!

  (. . . a fixpoint is hidden inside the program . . .)

- $do\ b \leftarrow get;\ \ put\ \neg b$: equation $b = \neg b$ has no solution in Booleans . . . but has solution in *CPO* of Booleans;

- “fixpoint”, “CPOs” : *domain theory* (& our library in Rocq)!

Backwards Causality with the Reverse State Monad

What does $\mathit{do}\ x \leftarrow \mathit{get};\ \ \mathit{put}(0\ ::\ \mathit{map}\ (1+)\ x)$ do ?

- $\mathit{do}\ x \leftarrow \mathit{get}$ reads *future* state, *after* $\mathit{put}(0\ ::\ \mathit{map}\ (1+)\ x)$;
- hence state simultaneously contains x *and* $(0\ ::\ \mathit{map}\ (1+)\ x)$;
- hence $x = 0\ ::\ \mathit{map}\ (1+)\ x)$; 1-line program solves *fixpoint equation*, finds unique solution $x = 0\ ::\ 1\ ::\ 2\ ::\ \ldots$ there must be a trick!

  (. . . a fixpoint is hidden inside the program . . .)

- $\mathit{do}\ b \leftarrow \mathit{get};\ \ \mathit{put}\ \neg b$: equation $b = \neg b$ has no solution in Booleans . . . but has solution in *CPO* of Booleans;

- “fixpoint”, “CPOs” : *domain theory* (& our library in Rocq)!

Backwards Causality with the Reverse State Monad

What does $\mathit{do}\ x \leftarrow \mathit{get};\ \ \mathit{put}(0\ ::\ \mathit{map}\ (1+)\ x)$ do ?

- $\mathit{do}\ x \leftarrow \mathit{get}$ reads *future* state, *after* $\mathit{put}(0\ ::\ \mathit{map}\ (1+)\ x)$;
- hence state simultaneously contains x *and* $(0\ ::\ \mathit{map}\ (1+)\ x)$;
- hence $x = 0\ ::\ \mathit{map}\ (1+)\ x)$; 1-line program solves *fixpoint equation*, finds unique solution $x = 0\ ::\ 1\ ::\ 2\ ::\ \ldots$ there must be a trick!
  (. . . a fixpoint is hidden inside the program . . .)
- $\mathit{do}\ b \leftarrow \mathit{get};\ \ \mathit{put}\ \neg b$: equation $b = \neg b$ has no solution in Booleans . . . but has solution in *CPO* of Booleans;
- "fixpoint", "CPOs" : *domain theory* (& our library in Rocq)!

Outline

Outline

Complete Partial Orders (CPOs)

- $(C, \preceq, \perp)$ with set C, order $\preceq$ on C , $\perp$ least element of C;
- $\preceq$ interpreted as *definition order*, with $\perp$ interpreted as *undefined*;
- each *increasing sequence*[1] S has *least upper bound* $lub\ S$.

Examples:

- *flat* CPO $\mathbb{N} \cup \{\perp\}$: order restricted to $\mathbb{N}$ is equality;
- *Stream* over $\mathbb{N} \cup \{\perp\}$: *pointwise* order $s \sqsubseteq s'$ iff $\forall i \in \mathbb{N}, s[i] \preceq s'[i]$.

[1] Actually, each *directed set*.

Complete Partial Orders (CPOs)

- $(C, \preceq, \perp)$ with set C, order $\preceq$ on C , $\perp$ least element of C;
- $\preceq$ interpreted as *definition order*, with $\perp$ interpreted as *undefined*;
- each *increasing sequence*[1] S has *least upper bound* $lub\ S$.

Examples:

- *flat* CPO $\mathbb{N} \cup \{\perp\}$: order restricted to $\mathbb{N}$ is equality;
- *Stream* over $\mathbb{N} \cup \{\perp\}$: *pointwise* order $s \sqsubseteq s'$ iff $\forall i \in \mathbb{N}, s[i] \preceq s'[i]$.

[1] Actually, each *directed set*.

Complete Partial Orders (CPOs)

- $(C, \preceq, \bot)$ with set C, order $\preceq$ on C, $\bot$ least element of C;
- $\preceq$ interpreted as *definition order*, with $\bot$ interpreted as *undefined*;
- each *increasing sequence*[1] S has *least upper bound* $lub\ S$.

Examples:

- *flat* CPO $\mathbb{N} \cup \{\bot\}$: order restricted to $\mathbb{N}$ is equality;
- *Stream* over $\mathbb{N} \cup \{\bot\}$: *pointwise* order $s \sqsubseteq s'$ iff $\forall i \in \mathbb{N}, s[i] \preceq s'[i]$.

[1] Actually, each *directed set*.

Complete Partial Orders (CPOs)

- $(C, \preceq, \bot)$ with set C, order $\preceq$ on C , $\bot$ least element of C;
- $\preceq$ interpreted as *definition order*, with $\bot$ interpreted as *undefined*;
- each *increasing sequence*[1] S has *least upper bound* $lub\ S$.

Examples:

- *flat* CPO $\mathbb{N} \cup \{\bot\}$: order restricted to $\mathbb{N}$ is equality;
- *Stream* over $\mathbb{N} \cup \{\bot\}$: *pointwise* order $s \sqsubseteq s'$ iff $\forall i \in \mathbb{N}, s[i] \preceq s'[i]$.

[1] Actually, each *directed set*.

Continuous Functions

- for posets C, C': $f : C \to C'$ is *monotonic* iff $x \leq y$ implies $f\ x \leq' f\ y$;
- for CPOs C, C': $f : C \to C'$ is *continuous* iff f is monotonic & for all increasing sequence S , $f\ (lub\ S) = lub'\ (f\ S)$;
- notation $[C \to C']$ = set of continuous functions between CPOs C, C';
- examples : constant, identity, compositions of continuous functions;
- structure : CPOs + continuous functions = category **CPO**.

Continuous Functions

- for posets C, C': $f : C \to C'$ is *monotonic* iff $x \leq y$ implies $f\ x \leq' f\ y$;
- for CPOs C, C': $f : C \to C'$ is *continuous* iff f is monotonic & for all increasing sequence S , $f\ (lub\ S) = lub'\ (f\ S)$;
- notation $[C \to C']$ = set of continuous functions between CPOs C, C';
- examples : constant, identity, compositions of continuous functions;
- structure : CPOs + continuous functions = category **CPO**.

Continuous Functions

- for posets C, C': $f : C \rightarrow C'$ is *monotonic* iff $x \leq y$ implies $f\ x \leq' f\ y$;
- for CPOs C, C': $f : C \rightarrow C'$ is *continuous* iff f is monotonic & for all increasing sequence S , $f\ (lub\ S) = lub'\ (f\ S)$;
- notation $[C \rightarrow C']$ = set of continuous functions between CPOs C, C';
- examples : constant, identity, compositions of continuous functions;
- structure : CPOs + continuous functions = category **CPO**.

Continuous Functions

- for posets C, C': $f : C \to C'$ is *monotonic* iff $x \leq y$ implies $f\ x \leq' f\ y$;
- for CPOs C, C': $f : C \to C'$ is *continuous* iff f is monotonic & for all increasing sequence S , $f\ (lub\ S) = lub'\ (f\ S)$;
- notation $[C \to C']$ = set of continuous functions between CPOs C, C';
- examples : constant, identity, compositions of continuous functions;
- structure : CPOs + continuous functions = category **CPO**.

Continuous Functions

- for posets C, C': $f : C \to C'$ is *monotonic* iff $x \leq y$ implies $f\ x \leq' f\ y$;
- for CPOs C, C': $f : C \to C'$ is *continuous* iff f is monotonic & for all increasing sequence S , $f\ (lub\ S) = lub'\ (f\ S)$;
- notation $[C \to C']$ = set of continuous functions between CPOs C, C';
- examples : constant, identity, compositions of continuous functions;
- structure : CPOs + continuous functions = category **CPO**.

CPO is Cartesian Closed

For CPOs $(C' \leq, \bot)$, $(C', \leq', \bot')$, the following are CPOs:

- *product*: $(C \times C', \sqsubseteq, (\bot, \bot'))$ with pair-pointwise $\sqsubseteq$;
- *exponentiation*: $([C \to C'], \sqsubseteq, \lambda\_ \Rightarrow \bot')$ with function-pointwise $\sqsubseteq$.

Fixpoints

- Kleene: $f : [C \to C]$ has least fixpoint $\mathit{fix}\, f \triangleq \mathit{lub}\{f^{(n)}\bot \mid n \in \mathbb{N}\}$;
- fixpoints for several functions at once: *theorem of Bekić*;
- to prove continuity, compose elementary results:
  - $f : A \times B \to C$ is continuous iff it is so in each argument separately;
  - currying/uncurrying are continuous;
  - $\mathit{fix} : [C \to C] \to C$ is continuous;
  - and many more.

Fixpoints

- Kleene: $f : [C \to C]$ has least fixpoint $\mathit{fix}\, f \triangleq \mathit{lub}\{f^{(n)}\bot \mid n \in \mathbb{N}\}$;
- fixpoints for several functions at once: *theorem of Bekić*;
- to prove continuity, compose elementary results:
  - $f : A \times B \to C$ is continuous iff it is so in each argument separately;
  - currying/uncurrying are continuous;
  - $\mathit{fix} : [C \to C] \to C$ is continuous;
  - and many more.

Fixpoints

- Kleene: $f : [C \to C]$ has least fixpoint $\mathit{fix}\, f \triangleq \mathit{lub}\{f^{(n)}\bot \mid n \in \mathbb{N}\}$;
- fixpoints for several functions at once: *theorem of Bekić*;
- to prove continuity, compose elementary results:
  - $f : A \times B \to C$ is continuous iff it is so in each argument separately;
  - currying/uncurrying are continuous;
  - $\mathit{fix} : [C \to C] \to C$ is continuous;
  - and many more.

Outline

Monads in the Category **CPO**

$(M, ret, bind)$ where

- $M : \mathbf{CPO} \to \mathbf{CPO}$ is a functor;
- for all CPOs X, a function $ret_X : [X \to M\,X]$;
- for all CPOs X, Y a function $bind_{X,Y} : [M\,X \to [[X \to M\,Y] \to M\,Y]]$;
  notation: $do\ x \leftarrow m;\ m'$ for $bind_{X,Y}\ m\ (\lambda\, x \Rightarrow\ m')$;
- monad laws:
  - $do\ x \leftarrow m;\ ret\ x = m$;
  - $do\ x' \leftarrow ret\ x;\ f\ x' = f\ x$;
  - $do\ y \leftarrow (do\ x \leftarrow m;\ f\ x);\ g\ y = do\ x \leftarrow m;\ do\ y \leftarrow f\ x;\ g\ y$.

Monads in the Category **CPO**

$(M, ret, bind)$ where

- $M : \mathbf{CPO} \to \mathbf{CPO}$ is a functor;
- for all CPOs X, a function $ret_X : [X \to M\,X]$;
- for all CPOs X, Y a function $bind_{X,Y} : [M\,X \to [[X \to M\,Y] \to M\,Y]]$;
  notation: $do\ x \leftarrow m;\ m'$ for $bind_{X,Y}\ m\ (\lambda x \Rightarrow m')$;
- monad laws:
  - $do\ x \leftarrow m;\ ret\ x = m$;
  - $do\ x' \leftarrow ret\ x;\ f\ x' = f\ x$;
  - $do\ y \leftarrow (do\ x \leftarrow m;\ f\ x);\ g\ y = do\ x \leftarrow m;\ do\ y \leftarrow f\ x;\ g\ y$.

Monads in the Category **CPO**

$(M, ret, bind)$ where

- $M : \mathbf{CPO} \to \mathbf{CPO}$ is a functor;
- for all CPOs X, a function $ret_X : [X \to M\, X]$;
- for all CPOs X, Y a function $bind_{X,Y} : [M\, X \to [[X \to M\, Y] \to M\, Y]]$;
  notation: $do\ x \leftarrow m;\ m'$ for $bind_{X,Y}\ m\ (\lambda\, x \Rightarrow\ m')$;
- monad laws:
  - $do\ x \leftarrow m;\ ret\ x = m$;
  - $do\ x' \leftarrow ret\ x;\ f\ x' = f\ x$;
  - $do\ y \leftarrow (do\ x \leftarrow m;\ f\ x);\ g\ y = do\ x \leftarrow m;\ do\ y \leftarrow f\ x;\ g\ y$.

Monads in the Category **CPO**

$(M, ret, bind)$ where

- $M : \mathbf{CPO} \to \mathbf{CPO}$ is a functor;
- for all CPOs X, a function $ret_X : [X \to M\ X]$;
- for all CPOs X, Y a function $bind_{X,Y} : [M\ X \to [[X \to M\ Y] \to M\ Y]]$;
  notation: $do\ x \leftarrow m;\ m'$ for $bind_{X,Y}\ m\ (\lambda x \Rightarrow\ m')$;
- monad laws:
  - $do\ x \leftarrow m;\ ret\ x = m$;
  - $do\ x' \leftarrow ret\ x;\ f\ x' = f\ x$;
  - $do\ y \leftarrow (do\ x \leftarrow m;\ f\ x);\ g\ y = do\ x \leftarrow m;\ do\ y \leftarrow f\ x;\ g\ y$.

Monads in the Category **CPO**

$(M, ret, bind)$ where

- $M : \mathbf{CPO} \to \mathbf{CPO}$ is a functor;
- for all CPOs X, a function $ret_X : [X \to M\ X]$;
- for all CPOs X, Y a function $bind_{X,Y} : [M\ X \to [[X \to M\ Y] \to M\ Y]$;
  notation: $do\ x \leftarrow m;\ m'$ for $bind_{X,Y}\ m\ (\lambda x \Rightarrow\ m')$;
- monad laws:
  - $do\ x \leftarrow m;\ ret\ x = m$;
  - $do\ x' \leftarrow ret\ x;\ f\ x' = f\ x$;
  - $do\ y \leftarrow (do\ x \leftarrow m;\ f\ x);\ g\ y = do\ x \leftarrow m;\ do\ y \leftarrow f\ x;\ g\ y$.

Example: the Identity Monad

- identity functor $id : \mathbf{CPO} \to \mathbf{CPO}$;
- $\forall (A : \mathbf{CPO})(a : A),\ ret_A\ a = a$;
- $\forall (A\ B : \mathbf{CPO})(m : id\ A)(f : [A \to\ id\ B]),\ bind\ m\ f := \ f\ m$.

Example: the Continuation Monad Transformer

Parameterized by monad M and CPO R:

- $contT_R : \mathbf{CPO} \to \mathbf{CPO} = \lambda X \Rightarrow [[X \to M\ R] \to M\ R]$;
- $\forall (X : \mathbf{CPO})(x : X)$, $ret_X\ x = \lambda (k : [X \to M\ R]) \Rightarrow k\ x$;
- $\forall (X\ Y : \mathbf{CPO})(m : contT_R\ X)(f : [X \to contT_R\ Y])$,
  $bind\ m\ f = \lambda (k : [Y \to M\ R]) \Rightarrow m\,(\lambda (x : X) \Rightarrow f\ x\ k)$.

Encodes many other monads.

Example: the Continuation Monad Transformer

Parameterized by monad M and CPO R:

- $contT_R : \mathbf{CPO} \to \mathbf{CPO} = \lambda X \Rightarrow [[X \to M\ R] \to M\ R]$;
- $\forall (X : \mathbf{CPO})(x : X),\ ret_X\ x = \lambda (k : [X \to M\ R]) \Rightarrow k\ x$;
- $\forall (X\ Y : \mathbf{CPO})(m : contT_R\ X)(f : [X \to contT_R\ Y])$,
  $bind\ m\ f = \lambda (k : [Y \to M\ R]) \Rightarrow m\,(\lambda (x : X) \Rightarrow f\ x\ k)$.

Encodes many other monads.

Example: the Continuation Monad Transformer

Parameterized by monad M and CPO R:

- $contT_R : \mathbf{CPO} \to \mathbf{CPO} = \lambda X \Rightarrow [\,[X \to M\ R] \to M\ R];$
- $\forall (X : \mathbf{CPO})(x : X),\ ret_X\ x = \lambda (k : [X \to M\ R]) \Rightarrow k\ x;$
- $\forall (X\ Y : \mathbf{CPO})(m : contT_R\ X)(f : [X \to contT_R\ Y]),$
  $bind\ m\ f = \lambda (k : [Y \to M\ R]) \Rightarrow m\,(\lambda (x : X) \Rightarrow f\ x\ k).$

Encodes many other monads.

Example: the Continuation Monad Transformer

Parameterized by monad M and CPO R:

- $contT_R : \mathbf{CPO} \to \mathbf{CPO} = \lambda X \Rightarrow [\,[X \to M\ R] \to M\ R];$
- $\forall (X : \mathbf{CPO})(x : X),\ ret_X\ x = \lambda\,(k : [X \to M\ R]) \Rightarrow k\ x;$
- $\forall (X\ Y : \mathbf{CPO})(m : contT_R\ X)(f : [X \to contT_R\ Y]),$
  $bind\ m\ f = \lambda\,(k : [Y \to M\ R]) \Rightarrow m\,(\lambda\,(x : X) \Rightarrow f\ x\ k).$

Encodes many other monads.

Outline

The Reverse State Monad Transformer in Haskell[2]

Parameter : monad $(M, ret, bind)$

$revBind\ m\ f = \lambda\, s \Rightarrow mdo\ (x, s'') \leftarrow m\ s'; (x', s') \leftarrow f\ x\ s; ret\ (x', s'')$

- *mdo* "solves" mutually recursive equations thanks to lazy evaluation;
- *mdo* implemented using $mfix : [[X \rightarrow M\ X] \rightarrow M\ X]$;
- *mfix* defined as $\lambda\, f \Rightarrow fix\ (\lambda\, m \Rightarrow bind\ m\ f)$, right?
- we implemented *revRet* and *revBind* in Rocq, proved all required continuities, . . . but could not prove monad laws.

[2] https://hackage.haskell.org/package/rev-state-0.2.0.1

The Reverse State Monad Transformer in Haskell[2]

Parameter : monad $(M, ret, bind)$

$revBind\ m\ f = \lambda\, s \Rightarrow mdo\ (x, s'') \leftarrow m\ s'; (x', s') \leftarrow f\ x\ s; ret\ (x', s'')$

- *mdo* "solves" mutually recursive equations thanks to lazy evaluation;
- *mdo* implemented using $mfix : [[X \rightarrow M\ X] \rightarrow M\ X]$;
- *mfix* defined as $\lambda\, f \Rightarrow fix\ (\lambda\, m \Rightarrow bind\ m\ f)$, right?
- we implemented *revRet* and *revBind* in Rocq, proved all required continuities, . . . but could not prove monad laws.

[2] `https://hackage.haskell.org/package/rev-state-0.2.0.1`

The Reverse State Monad Transformer in Haskell[2]

Parameter : monad $(M, ret, bind)$

$revBind\ m\ f = \lambda\, s \Rightarrow mdo\ (x, s'') \leftarrow m\ s';\ (x', s') \leftarrow f\ x\ s;\ ret\ (x', s'')$

- *mdo* "solves" mutually recursive equations thanks to lazy evaluation;
- *mdo* implemented using $mfix : [[X \to M\ X] \to M\ X]$;
- *mfix* defined as $\lambda\, f \Rightarrow fix\ (\lambda\, m \Rightarrow bind\ m\ f)$, right?
- we implemented *revRet* and *revBind* in Rocq, proved all required continuities, . . . but could not prove monad laws.

[2] https://hackage.haskell.org/package/rev-state-0.2.0.1

The Reverse State Monad Transformer in Haskell[2]

Parameter : monad $(M, ret, bind)$

$revBind\ m\ f = \lambda\, s \Rightarrow mdo\ (x, s'') \leftarrow m\ s'; (x', s') \leftarrow f\ x\ s; ret\ (x', s'')$

- *mdo* "solves" mutually recursive equations thanks to lazy evaluation;
- *mdo* implemented using $mfix : [[X \to M\ X] \to M\ X]$;
- *mfix* defined as $\lambda\, f \Rightarrow fix\ (\lambda\, m \Rightarrow bind\ m\ f)$, right?
- we implemented *revRet* and *revBind* in Rocq, proved all required continuities, . . . but could not prove monad laws.

[2] `https://hackage.haskell.org/package/rev-state-0.2.0.1`

The Reverse State Monad Transformer in Haskell[2]

Parameter : monad $(M, ret, bind)$

$$revBind\ m\ f = \lambda\, s \Rightarrow mdo\ (x, s'') \leftarrow m\ s'; (x', s') \leftarrow f\ x\ s; ret\ (x', s'')$$

- *mdo* "solves" mutually recursive equations thanks to lazy evaluation;
- *mdo* implemented using $mfix : [[X \to M\ X] \to M\ X]$;
- *mfix* defined as $\lambda\, f \Rightarrow fix\ (\lambda\, m \Rightarrow bind\ m\ f)$, right?
- we implemented *revRet* and *revBind* in Rocq, proved all required continuities, ... but could not prove monad laws.

[2] `https://hackage.haskell.org/package/rev-state-0.2.0.1`

The Reverse State Monad Transformer in Haskell[2]

Parameter : monad $(M, ret, bind)$

$$revBind\ m\ f = \lambda\, s \Rightarrow mdo\ (x, s'') \leftarrow m\ s'; (x', s') \leftarrow f\ x\ s; ret\ (x', s'')$$

- *mdo* "solves" mutually recursive equations thanks to lazy evaluation;
- *mdo* implemented using $mfix : [[X \to M\ X] \to M\ X]$;
- *mfix* defined as $\lambda\, f \Rightarrow fix\ (\lambda\, m \Rightarrow bind\ m\ f)$, right?
- we implemented *revRet* and *revBind* in Rocq, proved all required continuities, . . . but could not prove monad laws.

[2] https://hackage.haskell.org/package/rev-state-0.2.0.1

What Went Wrong?

A closer look at Haskell code reveals that:

- parameter monad M must implement $\mathit{mfix} : [[X \to M\ X] \to M\ X]$;
- which is *not* defined as $\lambda f \Rightarrow \mathit{fix}\ (\lambda m \Rightarrow \mathit{bind}\ m\ f)$;
- no general definition for mfix, but specification as 4 axioms;
- implemented by most monads; e.g. for *identity*, $\mathit{mfix} = \mathit{fix}$;
- go back to Haskell's version?

What Went Wrong?

A closer look at Haskell code reveals that:

- parameter monad M must implement $mfix : [[X \to M\ X] \to M\ X]$;
- which is *not* defined as $\lambda f \Rightarrow fix\ (\lambda m \Rightarrow bind\ m\ f)$;
- no general definition for $mfix$, but specification as 4 axioms;
- implemented by most monads; e.g. for *identity*, $mfix = fix$;
- go back to Haskell's version?

What Went Wrong?

A closer look at Haskell code reveals that:

- parameter monad M must implement $mfix : [[X \to M\ X] \to M\ X]$;
- which is *not* defined as $\lambda f \Rightarrow fix\ (\lambda m \Rightarrow bind\ m\ f)$;
- no general definition for $mfix$, but specification as 4 axioms;
- implemented by most monads; e.g. for *identity*, $mfix = fix$;
- go back to Haskell's version?

What Went Wrong?

A closer look at Haskell code reveals that:

- parameter monad M must implement $mfix : [[X \to M\ X] \to M\ X]$;
- which is *not* defined as $\lambda f \Rightarrow fix\ (\lambda m \Rightarrow bind\ m\ f)$;
- no general definition for $mfix$, but specification as 4 axioms;
- implemented by most monads; e.g. for *identity*, $mfix = fix$;
- go back to Haskell's version?

What Went Wrong?

A closer look at Haskell code reveals that:

- parameter monad M must implement $\mathit{mfix} : [[X \to M\ X] \to M\ X]$;
- which is *not* defined as $\lambda f \Rightarrow \mathit{fix}\ (\lambda m \Rightarrow \mathit{bind}\ m\ f)$;
- no general definition for mfix, but specification as 4 axioms;
- implemented by most monads; e.g. for *identity*, $\mathit{mfix} = \mathit{fix}$;
- go back to Haskell's version?

What Went Wrong?

A closer look at Haskell code reveals that:

- parameter monad M must implement $\mathit{mfix} : [[X \to M\ X] \to M\ X]$;
- which is *not* defined as $\lambda f \Rightarrow \mathit{fix}\ (\lambda m \Rightarrow \mathit{bind}\ m\ f)$;
- no general definition for mfix, but specification as 4 axioms;
- implemented by most monads; e.g. for *identity*, $\mathit{mfix} = \mathit{fix}$;
- go back to Haskell's version?

$$\mathit{revBind}\ m\ f = \lambda s \Rightarrow \mathit{mdo}\ (x, s'') \leftarrow m\ s';\ (x', s') \leftarrow f\ x\ s;\ \mathit{ret}\ (x', s'')$$

What Went Wrong?

A closer look at Haskell code reveals that:

- parameter monad M must implement $mfix : [[X \to M\ X] \to M\ X]$;
- which is *not* defined as $\lambda\, f \Rightarrow fix\ (\lambda\, m \Rightarrow bind\ m\ f)$;
- no general definition for $mfix$, but specification as 4 axioms;
- implemented by most monads; e.g. for *identity*, $mfix = fix$;
- go back to Haskell's version?

$$revBind\ m\ f = \cancel{\lambda\, s \Rightarrow}\ \cancel{mdo}\ \cancel{(x, s'') \leftarrow m\ s'};\ \cancel{(x', s') \leftarrow f\ x\ s};\ \cancel{ret\ (x', s'')}$$

Alternative: Using the Continuation Monad Transformer

- define $revStateT\ M\ S$ as $\forall R.contT_R\ (M(S \times R))$; obtain $revRet$, $revBind$ satisfying monad laws;
- define $get = \lambda\,(k : [S \to M\,(S \times R)]) \Rightarrow mfix\ (k \circ fst)$;

  $put\ s = \lambda\,(k : [\mathbf{1} \to M\,(S \times R)]) \Rightarrow do\ (\_, r) \leftarrow k\ \bot_{\mathbf{1}};\ ret\ (s, r)$.

Alternative: Using the Continuation Monad Transformer

- define $\mathit{revStateT}\ M\ S$ as $\forall R.\mathit{contT}_R\ (M(S \times R))$; obtain revRet, $\mathit{revBind}$ satisfying monad laws;
- define $\mathit{get} = \lambda(k : [S \to M\ (S \times R)]) \Rightarrow \mathit{mfix}\ (k \circ \mathit{fst})$;

  $\mathit{put}\ s = \lambda(k : [\mathbf{1} \to M\ (S \times R)]) \Rightarrow \mathit{do}\ (\_, r) \leftarrow k\ \bot_{\mathbf{1}};\ \mathit{ret}\ (s, r)$.

Alternative: Using the Continuation Monad Transformer

- define $revStateT\ M\ S$ as $\forall R.contT_R\ (M(S \times R))$; obtain $revRet$, $revBind$ satisfying monad laws;
- define $get = \lambda\,(k : [S \to M\,(S \times R)]) \Rightarrow mfix\ (k \circ fst)$;

  $put\ s = \lambda\,(k : [\mathbf{1} \to M\,(S \times R)]) \Rightarrow do\ (\_, r) \leftarrow k\ \bot_{\mathbf{1}};\ ret\ (s, r)$.

Alternative: Using the Continuation Monad Transformer

- define $\mathit{revStateT}\ M\ S$ as $\forall R.\mathit{contT}_R\ (M(S \times R))$; obtain revRet, $\mathit{revBind}$ satisfying monad laws;
- define $\mathit{get} = \lambda\,(k : [S \to M\ (S \times R)]) \Rightarrow \mathit{mfix}\ (k \circ \mathit{fst})$;

  $\mathit{put}\ s = \lambda\,(k : [\mathbf{1} \to M\ (S \times R)]) \Rightarrow \mathit{do}\ (\_, r) \leftarrow k\ \bot_{\mathbf{1}};\ \mathit{ret}\ (s, r)$.

Why is this a Reverse State Monad Transformer?

We proved in Rocq, using the 4 laws of *mfix*:

- state flows backwards (unlike forwards state monad)::
  $put\ x; put\ y = put\ x;$
- output flows forward (like in forward state monad):
  $do\ x \leftarrow get; do\ y \leftarrow get; m\ x\ y = do\ x \leftarrow get; m\ x\ x;$
- for $f : [S \rightarrow S]$, $do\ x \leftarrow get; put\ (f\ x) = put\ (fix\ f)$
  fixpoints values determine whether backward causality is paradoxical:
  - $do\ x \leftarrow get;\ put(0 :: map\ (1+)\ x) = put(fix(\lambda\, x \Rightarrow (0 :: map\ (1+)\ x)))$
    $= put\ (0 :: 1 :: 2 :: \ldots)$: proper Stream value, no causality paradox;
  - $do\ b \leftarrow get;\ put(\neg b) = put(fix(\lambda\, b \Rightarrow (\neg b))) = put\ \bot$: undefined Boolean value, causality paradox.

Why is this a Reverse State Monad Transformer?

We proved in Rocq, using the 4 laws of *mfix*:

- state flows backwards (unlike forwards state monad)::
  $put\ x; put\ y = put\ x;$
- output flows forward (like in forward state monad):
  $do\ x \leftarrow get; do\ y \leftarrow get; m\ x\ y = do\ x \leftarrow get; m\ x\ x;$
- for $f : [S \rightarrow S]$, $do\ x \leftarrow get; put\ (f\ x) = put\ (fix\ f)$
  fixpoints values determine whether backward causality is paradoxical:
  - $do\ x \leftarrow get;\ put(0 :: map\ (1+)\ x) = put(fix(\lambda\, x \Rightarrow (0 :: map\ (1+)\ x)))$
    $= put\ (0 :: 1 :: 2 :: \ldots)$: proper Stream value, no causality paradox;
  - $do\ b \leftarrow get;\ put(\neg b) = put(fix(\lambda\, b \Rightarrow (\neg b))) = put\ \bot$: undefined Boolean value, causality paradox.

Why is this a Reverse State Monad Transformer?

We proved in Rocq, using the 4 laws of *mfix*:

- state flows backwards (unlike forwards state monad)::
  $put\ x; put\ y = put\ x;$
- output flows forward (like in forward state monad):
  $do\ x \leftarrow get; do\ y \leftarrow get; m\ x\ y = do\ x \leftarrow get; m\ x\ x;$
- for $f : [S \rightarrow S]$, $do\ x \leftarrow get; put\ (f\ x) = put\ (fix\ f)$
  fixpoints values determine whether backward causality is paradoxical:
  - $do\ x \leftarrow get;\ put(0 :: map\ (1+)\ x) = put(fix(\lambda\, x \Rightarrow (0 :: map\ (1+)\ x)))$
    $= put\ (0 :: 1 :: 2 :: \ldots)$: proper Stream value, no causality paradox;
  - $do\ b \leftarrow get;\ put(\neg b) = put(fix(\lambda\, b \Rightarrow (\neg b))) = put\ \bot$: undefined Boolean value, causality paradox.

Why is this a Reverse State Monad Transformer?

We proved in Rocq, using the 4 laws of *mfix*:

- state flows backwards (unlike forwards state monad)::
  $put\ x; put\ y = put\ x;$
- output flows forward (like in forward state monad):
  $do\ x \leftarrow get; do\ y \leftarrow get; m\ x\ y = do\ x \leftarrow get; m\ x\ x;$
- for $f : [S \to S]$, $do\ x \leftarrow get; put\ (f\ x) = put\ (fix\ f)$
  fixpoints values determine whether backward causality is paradoxical:
  - $do\ x \leftarrow get;\ put(0 :: map\ (1+)\ x) = put(fix(\lambda\, x \Rightarrow (0 :: map\ (1+)\ x)))$
    $= put\ (0 :: 1 :: 2 :: \ldots)$: proper Stream value, no causality paradox;
  - $do\ b \leftarrow get;\ put(\neg b) = put(fix(\lambda\, b \Rightarrow (\neg b))) = put\ \bot$: undefined Boolean value, causality paradox.

Why is this a Reverse State Monad Transformer?

We proved in Rocq, using the 4 laws of *mfix*:

- state flows backwards (unlike forwards state monad)::
  $put\ x; put\ y = put\ x;$
- output flows forward (like in forward state monad):
  $do\ x \leftarrow get; do\ y \leftarrow get; m\ x\ y = do\ x \leftarrow get; m\ x\ x;$
- for $f : [S \to S]$, $do\ x \leftarrow get; put\ (f\ x) = put\ (fix\ f)$
  fixpoints values determine whether backward causality is paradoxical:
  - $do\ x \leftarrow get; put(0 :: map\ (1+)\ x) = put(fix(\lambda\, x \Rightarrow (0 :: map\ (1+)\ x)))$ $= put\ (0 :: 1 :: 2 :: \ldots)$: proper Stream value, no causality paradox;
  - $do\ b \leftarrow get; put(\neg b) = put(fix(\lambda\, b \Rightarrow (\neg b))) = put\ \bot$: undefined Boolean value, causality paradox.

Why is this a Reverse State Monad Transformer?

We proved in Rocq, using the 4 laws of *mfix*:

- state flows backwards (unlike forwards state monad)::
  $put\ x; put\ y = put\ x;$
- output flows forward (like in forward state monad):
  $do\ x \leftarrow get; do\ y \leftarrow get; m\ x\ y = do\ x \leftarrow get; m\ x\ x;$
- for $f : [S \rightarrow S]$, $do\ x \leftarrow get; put\ (f\ x) = put\ (fix\ f)$
  fixpoints values determine whether backward causality is paradoxical:
  - $do\ x \leftarrow get;\ put(0 :: map\ (1+)\ x) = put(fix(\lambda\, x \Rightarrow (0 :: map\ (1+)\ x)))$
    = $put\ (0 :: 1 :: 2 :: \ldots)$: proper Stream value, no causality paradox;
  - $do\ b \leftarrow get;\ put(\neg b) = put(fix(\lambda\, b \Rightarrow (\neg b))) = put\ \bot$: undefined
    Boolean value, causality paradox.

Why is this a Reverse State Monad Transformer?

We proved in Rocq, using the 4 laws of *mfix*:

- state flows backwards (unlike forwards state monad)::
  $put\ x; put\ y = put\ x;$
- output flows forward (like in forward state monad):
  $do\ x \leftarrow get; do\ y \leftarrow get; m\ x\ y = do\ x \leftarrow get; m\ x\ x;$
- for $f : [S \rightarrow S]$, $do\ x \leftarrow get; put\ (f\ x) = put\ (fix\ f)$
  fixpoints values determine whether backward causality is paradoxical:
  - $do\ x \leftarrow get;\ put(0 :: map\ (1+)\ x) = put(fix(\lambda\, x \Rightarrow (0 :: map\ (1+)\ x)))$
    = $put\ (0 :: 1 :: 2 :: \ldots)$: proper Stream value, no causality paradox;
  - $do\ b \leftarrow get;\ put(\neg b) = put(fix(\lambda\, b \Rightarrow (\neg b))) = put\ \bot$: undefined Boolean value, causality paradox.

Outline

Still Work in Progress

- formalized Reverse State Monad Transformer in Rocq:
- for now, only 2 simple programs & equational reasoning.

- application: parsers; Reverse State Monad to deal with *lookahead*
- other proof techniques to be investigated (Hoare Logics, . . .).
- *there is a future in the past*[3].

[3] Christopher Nolan, *Tenet*, 2020.

Still Work in Progress

- formalized Reverse State Monad Transformer in Rocq:
- for now, only 2 simple programs & equational reasoning.

- application: parsers; Reverse State Monad to deal with *lookahead*
- other proof techniques to be investigated (Hoare Logics, . . .).
- *there is a future in the past*[3].

[3] Christopher Nolan, *Tenet*, 2020.

Still Work in Progress

- formalized Reverse State Monad Transformer in Rocq:
- for now, only 2 simple programs & equational reasoning.

- application: parsers; Reverse State Monad to deal with *lookahead*
- other proof techniques to be investigated (Hoare Logics, . . .).
- *there is a future in the past*[3].

[3] Christopher Nolan, *Tenet*, 2020.

Still Work in Progress

- formalized Reverse State Monad Transformer in Rocq:
- for now, only 2 simple programs & equational reasoning.

- application: parsers; Reverse State Monad to deal with *lookahead*
- other proof techniques to be investigated (Hoare Logics, . . .).
- *there is a future in the past*[3].

[3] Christopher Nolan, *Tenet*, 2020.