Verified Causal Broadcast with Liquid Haskell

Patrick Redmond             Gan Shen             Niki Vazou             Lindsey Kuper

Dagstuhl Seminar 23112: Unifying Formal Methods for Trustworthy Distributed Systems
13 March 2023

github.com/lsd-ucsc/cbcast-lh
Verified Causal Broadcast with Liquid Haskell

Patrick Redmond  Gan Shen  Niki Vazou  Lindsey Kuper

dagstuhl seminar 23112: unifying formal methods for trustworthy distributed systems
13 March 2023

University of California Santa Cruz

Institute of Software

github.com/lsd-ucsc/cbcast-lh
Verified Causal Broadcast with Liquid Haskell

Patrick Redmond  Gan Shen  Niki Vazou  Lindsey Kuper

github.com/lsd-ucsc/cbcast-lh
Verified Causal Broadcast with Liquid Haskell

Patrick Redmond  Gan Shen  Niki Vazou  Lindsey Kuper

Dagstuhl Seminar 23112:
Unifying Formal Methods for Trustworthy Distributed Systems
13 March 2023

github.com/lsd-ucsc/cbcast-lh
Verified Causal Broadcast with Liquid Haskell

Patrick Redmond  Gan Shen  Niki Vazou  Lindsey Kuper

github.com/lsd-ucsc/cbcast-lh
Verified Causal Broadcast with Liquid Haskell

Patrick Redmond  Gan Shen  Niki Vazou  Lindsey Kuper

dagstuhl seminar 23112:
unifying formal methods for trustworthy distributed systems
13 march 2023

github.com/lsd-ucsc/cbcast-lh
Verified Causal Broadcast with Liquid Haskell (IFL ’22)

Patrick Redmond    Gan Shen    Niki Vazou    Lindsey Kuper

dagsuhl Seminar 23112:
Unifying Formal Methods for Trustworthy Distributed Systems
13 March 2023

university of california
Santa Cruz

institute
idea
software

github.com/lsd-ucsc/cbcast-lh
Lost my ...
Lost my ... 

Found it!

2
Lost my ... 😕

Found it!
Lost my…

Found it!

😕

happens-before

2
Lost my ... 

Found it!

😕

happens-before

happens-before
Lost my …

Found it!

🙂

FIFO delivery

happens-before

Lost my …

Found it!

happens-before
Lost my …

Found it!

🙂

Yay!

FIFO delivery

Lost my …

happens-before

Found it!

happens-before

Yay!

FIFO delivery
Lost my... 😮

Found it! 😍

Yay!

FIFO delivery

happens-before

...

happens-before
Lost my

Found it!

🙂

😮

Yay!

FIFO delivery

happens-before

Lost my

Found it!

😮

happens-before
Lost my

Find it!

🙂

😮

Yay!

FIFO delivery

happens-before

2
Lost my …

Found it!

🙂

Yay!

FIFO delivery

Lost my …

Found it!

🙂

happens-before

🙂

2
Causal broadcast with vector clocks [Birman et al., 1991]
Causal broadcast with vector clocks [Birman et al., 1991]
Lost my book...

Causal broadcast with vector clocks [Birman et al., 1991]
A message is **deliverable** if its VC is:

- 1 greater than recipient’s VC in sender’s position
- ≤ recipient’s VC elsewhere

Causal broadcast with vector clocks [Birman et al., 1991]
A message is **deliverable** if its VC is:

- 1 greater than recipient’s VC in sender’s position
- ≤ recipient’s VC elsewhere

Causal broadcast with vector clocks [Birman et al., 1991]
A message is deliverable if its VC is:
- 1 greater than recipient’s VC in sender’s position
- ≤ recipient’s VC elsewhere

Causal broadcast with vector clocks [Birman et al., 1991]
A message is **deliverable** if its VC is:

- 1 greater than recipient’s VC in sender’s position
- ≤ recipient’s VC elsewhere

Causal broadcast with vector clocks [Birman et al., 1991]
A message is **deliverable** if its VC is:

- 1 greater than recipient’s VC in sender’s position
- ≤ recipient’s VC elsewhere

Causal broadcast with vector clocks [Birman et al., 1991]
A message is **deliverable** if its VC is:

- 1 greater than recipient’s VC in sender’s position
- \( \leq \) recipient’s VC elsewhere

Causal broadcast with vector clocks [Birman et al., 1991]
A message is **deliverable** if its VC is:

- 1 greater than recipient’s VC in sender’s position
- ≤ recipient’s VC elsewhere

Causal broadcast with vector clocks [Birman et al., 1991]
A message is **deliverable** if its VC is:
- 1 greater than recipient’s VC in sender’s position
- \( \leq \) recipient’s VC elsewhere

Causal broadcast with vector clocks [Birman et al., 1991]
A message is **deliverable** if its VC is:
- 1 greater than recipient’s VC in sender’s position
- ≤ recipient’s VC elsewhere

Causal broadcast with vector clocks [Birman et al., 1991]
A message is **deliverable** if its VC is:

- 1 greater than recipient’s VC in sender’s position
- \( \leq \) recipient’s VC elsewhere

Causal broadcast with vector clocks [Birman et al., 1991]
A message is **deliverable** if its VC is:

- 1 greater than recipient’s VC in sender’s position
- ≤ recipient’s VC elsewhere

Causal broadcast with vector clocks [Birman et al., 1991]
A message is **deliverable** if its VC is:
- 1 greater than recipient’s VC in sender’s position
- ≤ recipient’s VC elsewhere

Causal broadcast with vector clocks [Birman et al., 1991]
A message is **deliverable** if its VC is:
- 1 greater than recipient’s VC in sender’s position
- \( \leq \) recipient’s VC elsewhere

Causal broadcast with vector clocks [Birman et al., 1991]
A message is **deliverable** if its VC is:

- 1 greater than recipient’s VC in sender’s position
- ≤ recipient’s VC elsewhere

Causal broadcast with vector clocks [Birman et al., 1991]
A message is **deliverable** if its VC is:

- 1 greater than recipient’s VC in sender’s position
- ≤ recipient’s VC elsewhere

Causal broadcast with vector clocks [Birman et al., 1991]
A message is **deliverable** if its VC is:
- 1 greater than recipient’s VC in sender’s position
- ≤ recipient’s VC elsewhere

Causal broadcast with vector clocks [Birman et al., 1991]
Programmers should be able to...

express and prove interesting correctness properties
Programmers should be able to...

express and prove **interesting correctness properties**

...of **deployable implementations** of distributed systems
Programmers should be able to...

express and prove interesting correctness properties

...of deployable implementations of distributed systems

...using language-integrated verification tools
Programmers should be able to...

express and prove interesting correctness properties
...of deployable implementations of distributed systems
...using language-integrated verification tools (i.e., types!)
Refinement types
type Nat = { v:Int | v >= 0 }

Refinement types
type Nat = { v: Int | v >= 0 }
type Nat = \{ \text{v:} \text{Int} | \text{v} \geq 0 \} \\
\text{type } \text{VectorClock} = [\text{Nat}]
type Nat = { v:Int | v >= 0 }

val VectorClock = [Nat]

vcMerge :: VectorClock -> VectorClock -> VectorClock
type Nat = { v:Int | v >= 0 }

vcMerge :: VectorClock -> VectorClock -> VectorClock

vcMerge = zipWith max

e.g., vcMerge [1,0,0,0] [0,2,0,1] = [1,2,0,1]

Refinement types
type Nat = \{ v: \text{Int} \mid v \geq 0 \} \\
type VectorClock = [\text{Nat}]

cMerge :: VectorClock -> VectorClock -> VectorClock \\
cMerge = \text{zipWith max}

\begin{center}
\text{e.g., } cMerge \begin{bmatrix} 1,0,0,0 \end{bmatrix} \begin{bmatrix} 0,2,0,1 \end{bmatrix} = \begin{bmatrix} 1,2,0,1 \end{bmatrix}
\end{center}

type VC\text{ized} N = \{ vc: VectorClock \mid \text{len} vc == N \}
type Nat = { v:Int | v >= 0 }

type VectorClock = [Nat]

vcMerge :: VectorClock -> VectorClock -> VectorClock
vcMerge = zipWith max

e.g., vcMerge [1,0,0,0] [0,2,0,1] = [1,2,0,1]


type VCsize N = { vc:VectorClock | len vc == N }
type VCsameLength V = VCsize {len V}

Refinement types
type Nat = { v: Int | v >= 0 }

give VectorClock = [Nat]

define vcMerge :: VectorClock -> VectorClock -> VectorClock
define vcMerge = zipWith max

e.g., vcMerge [1,0,0,0] [0,2,0,1] = [1,2,0,1]

type VCsized N = { vc: VectorClock | len vc == N }
type VCsameLength V = VCsized {len V}

define vcMerge :: v: VectorClock -> VCsameLength {v} -> VCsameLength {v}

Refinement types
type Nat = { v:Int | v >= 0 }
type VectorClock = [Nat]

type VCsized N = { vc:VectorClock | len vc == N }
type VCsameLength V = VCsized {len V}

vcMerge :: v:VectorClock -> VCsameLength {v} -> VCsameLength {v}
vcMerge = zipWith max

Refinement types
type Nat = \{ v : \text{Int} \mid v \geq 0 \} 

type VectorClock = [\text{Nat}]

type VC\text{ sized} \text{ } N = \{ \text{vc : VectorClock} \mid \text{len} \text{ } \text{vc} == N \} 

type VCsameLength \text{ } V = \text{VC\text{ sized}} \text{ } \{\text{len} \text{ } V\}

vcMerge :: v : \text{VectorClock} \rightarrow \text{VCsameLength} \text{ } \{v\} \rightarrow \text{VCsameLength} \text{ } \{v\}

vcMerge = \text{zipWith} \text{ } \text{max}

\textbf{Refinement} \textit{reflection}
type Nat = { v: Int | v ≥= 0 }

type VectorClock = [Nat]

type VCsized N = { vc: VectorClock | len vc == N }

type VCsameLength V = VCsized {len V}

vcMerge :: v: VectorClock -> VCsameLength {v} -> VCsameLength {v}
vcMerge = zipWith max

Refinement reflection
type Nat = { v:Int | v >= 0 }
type VectorClock = [Nat]

type VC sized N = { vc:VectorClock | len vc == N }
type VCsameLength V = VCsized {len V}

vcMerge :: v:VectorClock -> VCsameLength {v} -> VCsameLength {v}
vcMerge = zipWith max

type Commutative a A = x:a -> y:a -> { _:Proof | A x y == A y x }

Refinement reflection
type Nat = { v: Int | v >= 0 }

type VectorClock = [Nat]

type VCsize N = { vc: VectorClock | len vc == N }

type VCSameLength V = VCsize {len V}

vcMerge :: v: VectorClock -> VCSameLength {v} -> VCSameLength {v}
vcMerge = zipWith max

type Commutative a A = x:a -> y:a -> { _:Proof | A x y == A y x }

Refinement reflection
type Nat = { v:Int | v >= 0 }
type VectorClock = [Nat]

type VC sized N = { vc:VectorClock | len vc == N }  
type VC sameLength V = VC sized {len V}

vcMerge :: v:VectorClock -> VC sameLength {v} -> VC sameLength {v}  
vcMerge = zipWith max


type Commutative a A = x:a -> y:a -> { _:Proof | A x y == A y x }
type Nat = { v: Int | v >= 0 }
type VectorClock = [Nat]

type VCsized N = { vc: VectorClock | len vc == N }
type VCsameLength V = VCsized {len V}

cMerge :: v: VectorClock -> VCsameLength {v} -> VCsameLength {v}
cMerge = zipWith max

type Commutative a A = x:a -> y:a -> { _:Proof | A x y == A y x }

cMergeComm :: n: Nat -> Commutative (VCsized n) vcMerge

Refinement reflection
type Nat = { v:Int | v >= 0 }

type VectorClock = [Nat]

type VCscoped N = { vc:VectorClock | len vc == N }

type VCsameLength V = VCscoped {len V}

vcMerge :: v:VectorClock -> VCsameLength {v} -> VCsameLength {v}
vcMerge = zipWith max

vcMergeComm :: n:Nat -> Commutative (VCscoped n) vcMerge

Refinement reflection
type Nat = { v: Int | v >= 0 }  

type VectorClock = [Nat]


type VC sized N = { vc: VectorClock | len vc == N }  
type VCsameLength V = VC sized {len V}


vcMerge :: v: VectorClock -> VCsameLength {v} -> VCsameLength {v}  
vcMerge = zipWith max


type Commutative a A = x: a -> y: a -> { _ : Proof | A x y == A y x }  

vcMergeComm :: n: Nat -> Commutative (VC sized n) vcMerge
vcMergeComm _ n [] [] = ()
vcMergeComm n (_x: xs) (_y: ys) = vcMergeComm (n - 1) xs ys


Refinement reflection
type Nat = \{ \text{v:Int} \mid \text{v} \geq 0 \} 

\text{type VectorClock} = [\text{Nat}]

type VCsized N = \{ \text{vc:VectorClock} \mid \text{len vc} == N \} 

\text{type VCsameLength V} = \text{VCsized} \{\text{len V}\}

\text{vcMerge} :: \text{v:VectorClock} \to \text{VCsameLength \{v\}} \to \text{VCsameLength \{v\}} 
\text{vcMerge} = \text{zipWith max}

\text{type Commutative a A = x:a} \to \text{y:a} \to \{\_:_\text{Proof} \mid \text{A x y} == \text{A y x}\}

\text{vcMergeComm} :: \text{n:Nat} \to \text{Commutative (VCsized n) vcMerge} 
\text{vcMergeComm}_n [\_] [\_] = () 
\text{vcMergeComm} n (_x:xs) (_y:ys) = \text{vcMergeComm} (n - 1) xs ys

\textbf{Refinement reflection}
type Nat = { v: Int | v >= 0 }

type VectorClock = [Nat]

type VCsized N = { vc: VectorClock | len vc == N }

type VCsameLength V = VCsized {len V}

vcMerge :: v: VectorClock -> VCsameLength {v} -> VCsameLength {v}
vcMerge = zipWith max

vcMergeComm :: n: Nat -> Commutative (VCsized n) vcMerge
vcMergeComm _n [] [] = ()
vcMergeComm n (_x:xs) (_y:ys) = vcMergeComm (n - 1) xs ys

Refinement reflection
type Nat = \{ \texttt{v:Int} \mid \texttt{v} \geq 0 \} 

type VectorClock = [Nat]

type VCsized N = \{ \texttt{vc:VectorClock} \mid \text{len} \texttt{vc} == N \} 


type VCsameLength V = VCsized \{\text{len } V\} 

vcMerge :: \texttt{v:VectorClock} \rightarrow \texttt{VCsameLength \{v\}} \rightarrow \texttt{VCsameLength \{v\}} 

vcMerge = \text{zipWith max} 

vcMergeComm :: \texttt{n:Nat} \rightarrow \texttt{Commutative (VCsized n) vcMerge} 

vcMergeComm _n [] [] = () 

vcMergeComm n (_x:xs) (_y:ys) = vcMergeComm (n - 1) xs ys 

Refinement \textit{reflection}
type Nat = { v: Int | v >= 0 }  

type VectorClock = [Nat]  


type VCsized N = { vc: VectorClock | len vc == N }  

type VCsameLength V = VCsized {len V}  

vcMerge :: v: VectorClock -> VCsameLength {v} -> VCsameLength {v}  
vcMerge = zipWith max

vcMergeComm :: n: Nat -> Commutative (VCsized n) vcMerge  
vcMergeComm _n [] [] = ()  
vcMergeComm n (_x:xs) (_y:ys) = vcMergeComm (n - 1) xs ys

Re
fi
me
nt
Re
fi
me
nt

Refinement reflection
(Local) causal delivery as a refinement type

User's process history ($pHist$):
[(Deliver “Lost my 📲”),
(Deliver “Found it!”),
(Broadcast “Yay!”),
...]

verification code
(Local) causal delivery as a refinement type

```haskell
type LocalCausalDelivery P =
  { m1 : Message | elem (Deliver (pID P) m1) (pHist P) }
-> { m2 : Message | elem (Deliver (pID P) m2) (pHist P)
      && vcLess (mVC m1) (mVC m2) }
-> { _: Proof | processOrder (pHist P) (Deliver (pID P) m1)
                     (Deliver (pID P) m2) }
```

’s process history (pHist):
[(Deliver "Lost my 📱"),
 (Deliver "Found it!"),
 (Broadcast "Yay!")]

(Local) causal delivery as a refinement type

```
type LocalCausalDelivery P =
    { m1 : Message | elem (Deliver (pID P) m1) (pHist P) }
    -> { m2 : Message | elem (Deliver (pID P) m2) (pHist P)
        && vcLess (mVC m1) (mVC m2) }
    -> { _: Proof | processOrder (pHist P) (Deliver (pID P) m1)
               (Deliver (pID P) m2) }
```

's process history (pHist):
[(Deliver "Lost my 📱"),
  (Deliver "Found it!")],
(Broadcast "Yay!")
[...]

verification code
(Local) causal delivery as a refinement type

**Verification code**

```haskell
type LocalCausalDelivery P =
    { m1 : Message | elem (Deliver (pID P) m1) (pHist P) }
-> { m2 : Message | elem (Deliver (pID P) m2) (pHist P)
    && vcLess (mVC m1) (mVC m2) }
-> { _: Proof | processOrder (pHist P) (Deliver (pID P) m1)
    (Deliver (pID P) m2) }
```

**Process history (pHist):**

```
[(Deliver “Lost my 📱”),
 (Deliver “Found it!”),
 (Broadcast “Yay!”),
 ...]
```
(Local) causal delivery as a refinement type

type LocalCausalDelivery P =
  { m1 : Message | elem (Deliver (pID P) m1) (pHist P) }
-> { m2 : Message | elem (Deliver (pID P) m2) (pHist P)
    && vcLess (mVC m1) (mVC m2) }
-> { _: Proof | processOrder (pHist P) (Deliver (pID P) m1)
    (Deliver (pID P) m2) }

’s process history (pHist):
[(Deliver “Lost my 📱”),
 (Deliver “Found it!”),
 (Broadcast “Yay!”),
 ...]
(Local) causal delivery as a refinement type

```plaintext
type LocalCausalDelivery P =
  { m1 : Message | elem (Deliver (pID P) m1) (pHist P) }
-> { m2 : Message | elem (Deliver (pID P) m2) (pHist P)
  && vcLess (mVC m1) (mVC m2) }
-> { _: Proof | processOrder (pHist P) (Deliver (pID P) m1)
  (Deliver (pID P) m2) }
```

P's process history (pHist):
[(Deliver “Lost my 📱”),
 (Deliver “Found it!”),
 (Broadcast “Yay!”),
 ...]
Running the protocol preserves (local) causal delivery
Running the protocol preserves (local) causal delivery

data Op r = OpBroadcast r | OpReceive (Message r) | OpDeliver

step :: Op r -> Process -> Process
step (OpBroadcast r) p = ...
step (OpReceive m) p = ...
step (OpDeliver) p = ...

application code

verification code
Running the protocol preserves (local) causal delivery

data Op r = OpBroadcast r | OpReceive (Message r) | OpDeliver

step :: Op r -> Process -> Process
step (OpBroadcast r) p = ...
step (OpReceive m)   p = ...
step (OpDeliver)     p = ...

application code

lcdStep :: op : Op r
       -> p : Process
       -> LocalCausalDelivery p
       -> LocalCausalDelivery (step p op)
lcdStep op p lcdp =
       case op ? step op p of
          OpBroadcast r -> ... -- short proof
          OpReceive m   -> ... -- short proof
          OpDeliver     -> ... -- long proof

verification code
Running the protocol preserves (local) causal delivery

data Op r = OpBroadcast r | OpReceive (Message r) | OpDeliver

step :: Op r -> Process -> Process
step (OpBroadcast r) p = ...
step (OpReceive m)    p = ...
step (OpDeliver)      p = ...

application code

lcdStep :: op : Op r
         -> p : Process
         -> LocalCausalDelivery p
         -> LocalCausalDelivery (step p op)

lcdStep op p lcdn =
case op ? step op p of
  OpBroadcast r -> ... -- short proof
  OpReceive   m -> ... -- short proof
  OpDeliver   -> ... -- long proof
Running the protocol preserves (local) causal delivery

\[
data \text{ Op } r = \text{OpBroadcast } r \mid \text{OpReceive (Message } r) \mid \text{OpDeliver}
\]

\[
\text{step :: Op } r \rightarrow \text{Process } \rightarrow \text{Process}
\]
\[
\text{step (OpBroadcast } r) \rightarrow \text{Process } \rightarrow \ldots
\]
\[
\text{step (OpReceive } m) \rightarrow \text{Process } \rightarrow \ldots
\]
\[
\text{step (OpDeliver)} \rightarrow \text{Process } \rightarrow \ldots
\]

\[\text{application code}\]

\[
\text{lcdStep :: op : Op } r
\]
\[
\rightarrow \text{Process}
\]
\[
\rightarrow \text{LocalCausalDelivery } p
\]
\[
\rightarrow \text{LocalCausalDelivery (step } p \text{ op)}
\]
\[
\text{lcdStep } op \rightarrow p \rightarrow \text{lcdp } =
\]
\[
\text{case } op \rightarrow \text{step } op \rightarrow p \rightarrow \text{of}
\]
\[
\text{OpBroadcast } r \rightarrow \ldots \text{ -- short proof}
\]
\[
\text{OpReceive } m \rightarrow \ldots \text{ -- short proof}
\]
\[
\text{OpDeliver} \rightarrow \ldots \text{ -- long proof}
\]

\[\text{verification code}\]
Running the protocol preserves (local) causal delivery

```haskell
data Op r = OpBroadcast r | OpReceive (Message r) | OpDeliver

step :: Op r -> Process -> Process
step (OpBroadcast r) p = ...
step (OpReceive m) p = ...
step (OpDeliver) p = ...

```

application code

```haskell
lcdStep :: op : Op r -> p : Process
        -> LocalCausalDelivery p
        -> LocalCausalDelivery (step p op)

lcdStep op p lcdp =
  case op ? step op p of
    OpBroadcast r -> ... -- short proof
    OpReceive m -> ... -- short proof
    OpDeliver -> ... -- long proof

```

verification code
Running the protocol for one step preserves local causal delivery

↓ = “relies on”
Running the protocol for one step preserves local causal delivery

\[ \downarrow = \text{“relies on”} \]
Running the protocol for one step preserves local causal delivery.

broadcast, receive, deliver each preserve local causal delivery (deliver is the hard part)
Running the protocol for one step preserves causal delivery

Running the protocol for one step preserves local causal delivery

broadcast, receive, deliver each preserve local causal delivery
(deliver is the hard part)

= “relies on”
Running the protocol for one step preserves causal delivery

broadcast, receive, deliver
each preserve local causal delivery
(deliver is the hard part)

Causal delivery [Birman et al., 1991]: \( m \to m' = \forall p: \text{deliver}_p(m) \xRightarrow{\text{P}} \text{deliver}_p(m') \)
Running the protocol for one step preserves causal delivery

Running the protocol for one step preserves local causal delivery

broadcast, receive, deliver
each preserve local causal delivery
(deliver is the hard part)

Causal delivery [Birman et al., 1991]: \( m \rightarrow m' = \forall p: \text{deliver}_p(m) \xrightarrow{p} \text{deliver}_p(m') \)

**verification code**

```haskell
type CausalDelivery X =
  pid : PID -- any pid in the domain of execution X
  |-> { m : Message | elem (Deliver pid m) (pHist (X pid)) }
  |-> { m' : Message | elem (Deliver pid m') (pHist (X pid))
      & happensBefore X (Broadcast m) (Broadcast m') }
  |-> { _: Proof | procOrder (pHist (X pid)) (Deliver pid m) (Deliver pid m') }
```
Running the protocol for one step preserves causal delivery

Running the protocol for one step preserves local causal delivery

broadcast, receive, deliver each preserve local causal delivery (deliver is the hard part)

Causal delivery [Birman et al., 1991]: \( m \rightarrow m' = \forall p: deliver_p(m) \xrightarrow{p} deliver_p(m') \)

type CausalDelivery X =
  pid : PID -- any pid in the domain of execution X
  -> { m : Message | elem (Deliver pid m) (pHist (X pid)) }
  -> { m' : Message | elem (Deliver pid m') (pHist (X pid))
    && happensBefore X (Broadcast m) (Broadcast m') }
  -> { _: Proof | procOrder (pHist (X pid)) (Deliver pid m) (Deliver pid m') }
Running the protocol for one step preserves causal delivery

Running the protocol for one step preserves local causal delivery

broadcast, receive, deliver
each preserve local causal delivery
(deliver is the hard part)

Causal delivery [Birman et al., 1991]: \[ m \rightarrow m' = \forall p: \text{deliver}_p(m) \xrightarrow{P} \text{deliver}_p(m') \]

**type** CausalDelivery X =

- pid : PID -- any pid in the domain of execution X
- \{ m : Message | elem (Deliver pid m) (pHist (X pid)) \}
- \{ m' : Message | elem (Deliver pid m') (pHist (X pid))
- \&\& \text{happensBefore} X (Broadcast m) (Broadcast m') \}
- \{ _ : Proof | \text{procOrder} (pHist (X pid)) (Deliver pid m) (Deliver pid m') \}
Running the protocol for one step preserves causal delivery.

Whole execution observes causal delivery $\rightarrow$ each process observes local causal delivery.

broadcast, receive, deliver
each preserve local causal delivery (deliver is the hard part)

Causal delivery [Birman et al., 1991]: $m \rightarrow m' = \forall p: deliver_p(m) \xrightarrow{p} deliver_p(m')$

```haskell
module CausalDelivery where

-- pid : PID -- any pid in the domain of execution X

type CausalDelivery X =
  pid : PID
  -> { m : Message | elem (Deliver pid m) (pHist (X pid)) }
  -> { m' : Message | elem (Deliver pid m') (pHist (X pid))
     && happensBefore X (Broadcast m) (Broadcast m') }
  -> { _ : Proof | procOrder (pHist (X pid)) (Deliver pid m) (Deliver pid m') }
```

= “relies on”
Running the protocol for one step preserves causal delivery

Running the protocol for one step preserves local causal delivery

whole execution observes causal delivery → each process observes local causal delivery

broadcast, receive, deliver each preserve local causal delivery
(deliver is the hard part)

Causal delivery [Birman et al., 1991]:\[ m \rightarrow m' = \forall p: deliver_p(m) \mathbin{\xrightarrow{p}} deliver_p(m') \]

### Type Definition

```
type CausalDelivery X =
    pid : PID -- any pid in the domain of execution X
  -> { m : Message | elem (Deliver pid m) (pHist (X pid)) }
  -> { m' : Message | elem (Deliver pid m') (pHist (X pid))
      && happensBefore X (Broadcast m) (Broadcast m') } 
  -> { _ : Proof | procOrder (pHist (X pid)) (Deliver pid m) (Deliver pid m') }
```
Running the protocol for one step preserves causal delivery

Running the protocol for one step preserves local causal delivery

whole execution observes causal delivery $\rightarrow$
each process observes local causal delivery

vector clocks reflect happens-before

broadcast, receive, deliver each preserve local causal delivery
(deliver is the hard part)

each process observes local causal delivery $\rightarrow$
whole execution observes causal delivery

Causal delivery [Birman et al., 1991]: $m \rightarrow m' = \forall p: \text{deliver}_p(m) \xrightarrow{P} \text{deliver}_p(m')$

**type** CausalDelivery X =

<table>
<thead>
<tr>
<th>pid : PID -- any pid in the domain of execution X</th>
</tr>
</thead>
<tbody>
<tr>
<td>--&gt; { m : Message</td>
</tr>
<tr>
<td>--&gt; { m' : Message</td>
</tr>
<tr>
<td>} &amp; happensBefore X (Broadcast m) (Broadcast m') }</td>
</tr>
<tr>
<td>--&gt; { _ : Proof</td>
</tr>
</tbody>
</table>

verification code
Running the protocol for one step preserves causal delivery

whole execution observes causal delivery → each process observes local causal delivery

vector clocks reflect happens-before

broadcast, receive, deliver each preserve local causal delivery (deliver is the hard part)

vector clocks preserve happens-before

each process observes local causal delivery → whole execution observes causal delivery

Causal delivery [Birman et al., 1991]: \[ m \rightarrow m' = \forall p: \text{deliver}_p(m) \xrightarrow{p} \text{deliver}_p(m') \]

**type** CausalDelivery X =

```
<table>
<thead>
<tr>
<th>pid : PID -- any pid in the domain of execution X</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ \rightarrow { m : Message</td>
</tr>
<tr>
<td>[ \rightarrow { m' : Message</td>
</tr>
<tr>
<td>&amp;&amp; [ \text{happensBefore X (Broadcast m) (Broadcast m')} ]</td>
</tr>
<tr>
<td>[ \rightarrow { _ : Proof</td>
</tr>
</tbody>
</table>
```

\[ \downarrow = \text{“relies on”} \]
Causal delivery [Birman et al., 1991]: \[ m \rightarrow m' = \forall p: \text{deliver}_p(m) \rightarrow \text{deliver}_p(m') \]

**type** CausalDelivery X =

\[
\begin{align*}
\text{pid} : \text{PID} & \quad \text{-- any pid in the domain of execution X} \\
\rightarrow & \{ m : \text{Message} \mid \text{elem (Deliver pid m ) (pHist (X pid))} \} \\
\rightarrow & \{ m' : \text{Message} \mid \text{elem (Deliver pid m') (pHist (X pid))} \} \\
& \quad \& \& \text{happensBefore X (Broadcast m) (Broadcast m')} \\
\rightarrow & \{ _: \text{Proof} \mid \text{procOrder (pHist (X pid)) (Deliver pid m) (Deliver pid m')} \}
\end{align*}
\]

**verification code**
Building apps with causal broadcast
Building apps with causal broadcast

- App logic
- App state
- Delay queue
- Message transport
- WAN

Connections:
- Broadcast from App logic to Delay queue
- Deliver to App state from Delay queue
- Receive to Message transport from Delay queue

Deliverable?
Building apps with causal broadcast
Building apps with causal broadcast
Building apps with causal broadcast
Building apps with causal broadcast

Credit: Matthew Weidner
Programmers should be able to...

express and prove interesting correctness properties
...of deployable implementations of distributed systems
...using language-integrated verification tools (i.e., types!)
Programmers should be able to...

express and prove **interesting correctness properties**

...of **deployable implementations** of distributed systems

...using **language-integrated verification tools** *(i.e., types!)*

---

**Toward Hole-Driven Development in Liquid Haskell**

PATRICK REDMOND, University of California, Santa Cruz, USA  
GAN SHEN, University of California, Santa Cruz, USA  
LINDSEY KUPER, University of California, Santa Cruz, USA

Liquid Haskell is an extension to the Haskell programming language that adds support for *refinement types*: data types augmented with SMT-decidable logical predicates that refine the set of values that can inhabit a type. Furthermore, Liquid Haskell’s support for *refinement reflection* enables the use of Haskell for general-purpose, mechanized theorem proving. A growing list of large-scale mechanized proof developments in Liquid Haskell take advantage of this capability. Adding theorem-proving capabilities to a "legacy" language like Haskell lets programmers directly verify properties of real-world Haskell programs (taking advantage of the existing highly tuned compiler, run-time system, and libraries), just by writing Haskell. However, more established proof assistants like Agda and Coq offer far better support for interactive proof development and insight into the proof state (for instance, what subgoals still need to be proved to finish a partially-complete proof). In contrast, Liquid Haskell provides only coarse-grained feedback to the user — either it reports a type error, or not — unfortunately hindering its usability as a theorem prover.

In this paper, we propose improving the usability of Liquid Haskell by extending it with support for Agda-style *typed holes* and interactive editing commands that take advantage of them. In Agda, typed holes allow programmers to indicate unfinished parts of a proof, and incrementally complete the proof in a dialogue with the compiler. While GHC Haskell already has its own Archimedes support for typed holes, we want...

[HATRA 2021]
Thank you!

Languages, Systems, and Data Lab: lsd.ucsc.edu
Lindsey’s research blog: decomposition.al

github.com/lsd-ucsc/cbcast-lh
Thank you!

Languages, Systems, and Data Lab: lsd.ucsc.edu
Lindsey’s research blog: decomposition.al

github.com/lsd-ucsc/cbcast-lh