Infinitary proof theory of first order linear logic with fixed points

Farzaneh Derakhshan

fderakhs@andrew.cmu.edu

ASL annual meeting, 2020

PhD student, Carnegie Mellon University
Advisor: Frank Pfenning

Induction and coinduction

Induction	Coinduction - Bisimulation

Induction and coinduction

Termination, Progress; A property eventually holds	Productivity, Equality of streams; A property holds infinitely often
Induction	Coinduction - Bisimulation

Induction and coinduction

Finite data types	Infinite data types
Natural numbers, Lists, etc.	Streams, Infinite trees, etc.
Termination, Progress; A property eventually holds	Productivity, Equality of streams; A property holds infinitely often
Induction	Coinduction - Bisimulation

Induction and coinduction

Least fixed points	Greatest fixed points
Finite data types	Infinite data types
Natural numbers, Lists, etc.	Streams, Infinite trees, etc.
Termination, Progress; A property eventually holds	Productivity,Equality of streams; A property holds infinitely often
Induction	Coinduction - Bisimulation

Mutual least and greatest fixed points

1. Examples?
2. Induction/Coinduction?
3. Termination/productivity?

Prove theorems using induction and coinduction - Previous works

- Induction principle
- Bisimulation
- Coinduction principle [Kozen and Silva]
- An infinitary calculus for first-order logic with inductive definitions [Brotherston]
- A finitary calculus for least and greatest fixed points in linear logic [Baelde]
- Well founded recursion with copatterns and sized types [Abel and Pientka]

Our contribution

A first order calculus for proving properties about mutual least and greatest fixed points, in particular Session-typed processes

1. Add fixed points and assign priorities to them,
2. Use circular edges in the proof for inductive and coinductive steps,
3. Impose a validity condition to ensure soundness of this proof system.

We use priorities in the validity condition to ensure valid simultaneous induction and conduction.

Finite lists: Example of least fixed points

Natural numbers

$$\mathtt{nat} =^1_\mu \oplus\{\mathsf{zero} : 1, \mathsf{succ} : \mathtt{nat}\}$$

$$\overline{3} = \mathsf{succ}\ \mathsf{succ}\ \mathsf{succ}\ \mathsf{zero}$$

Lists of natural numbers

$$\mathtt{list}_{\mathsf{nat}} =^1_\mu \oplus\{\mathsf{nil} : 1, \mathsf{cons} : \mathsf{nat} \otimes \mathtt{list}_{\mathsf{nat}}\}$$

$$\overline{[3, 3]} = \mathsf{cons}(\overline{3}, \mathsf{cons}(\overline{3}, \mathsf{nil}))$$

Programming with finite lists

Terminating

Append two lists

$$
\begin{aligned}
& l \leftarrow \texttt{Append} \leftarrow l_1, l_2 = \\
& \qquad \textbf{case}\, l_1\, (\mu_{list} \Rightarrow \\
& \qquad\quad \textbf{case}\, l_1\, (nil \Rightarrow \textbf{wait}\, l_1; l \leftarrow l_2 \\
& \qquad\qquad cons \Rightarrow l.\mu_{list}; \\
& \qquad\qquad\quad x \leftarrow \textbf{recv}\, l; \\
& \qquad\qquad\quad l.cons; \textbf{send}\, l\, x; l \leftarrow \texttt{Append} \leftarrow l_1\, l_2))
\end{aligned}
$$

If l1 is an empty list (nil): forward l2 to l.

If l1=cons(x, --):
send x to l and call Append
on l2 and the remaining of l1.

I use linear binary session typed processes for programming examples. See [1,2] for more info.

Termination and List as first order predicates

$$List(l_1) \vdash Terminate(\_ \leftarrow \text{Append} \leftarrow l_1 \; \_)$$

$$Terminate(\_ \leftarrow \text{Append} \leftarrow \texttt{nil} \; \_) =^1_\mu 1$$
$$Terminate(\_ \leftarrow \text{Append} \leftarrow (\text{cons}(x) :: l_1') \; \_) =^1_\mu Terminate(\_ \leftarrow \text{Append} \leftarrow l_1' \; \_)$$

$$List(\texttt{nil}) =^1_\mu 1$$
$$List(\text{cons}(x) :: l_1') =^1_\mu List(l_1')$$

Append terminates - proof

$List(l_1) \vdash Terminate(\_ \leftarrow \text{Append} \leftarrow l_1\ \_)$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\cdot \vdash 1}\ 1R}{1 \vdash 1}\ 1L}{1 \vdash Terminate(\_ \leftarrow \text{Append} \leftarrow \texttt{nil}\ \_)}\ \mu R}{\dagger\ \ List(\texttt{nil}) \vdash Terminate(\_ \leftarrow \text{Append} \leftarrow \texttt{nil}\ \_)}\ \mu L$$

$$\dfrac{\dfrac{\dfrac{\dagger}{List(l_1') \vdash Terminate(\_ \leftarrow \text{Append} \leftarrow l_1'\ \_)}}{List(l_1') \vdash Terminate(\_ \leftarrow \text{Append} \leftarrow (\text{cons}(x) :: l_1')\ \_)}\ \mu R}{\dagger\ \ List(\text{cons}(x) :: l_1') \vdash Terminate(\_ \leftarrow \text{Append} \leftarrow (\text{cons}(x) :: l_1')\ \_)}\ \mu L$$

Programming with streams: example of greatest fixed points

Productive

merge Merge two streams into a single stream by alternatively outputting an element of each.

split_1 Return the odd elements of a stream.

split_2 Return the even elements of a stream.

$$\mathsf{merge}(\mathsf{split}_1(t), \mathsf{split}_2(t)) = t$$

Programming with streams: example of greatest fixed points

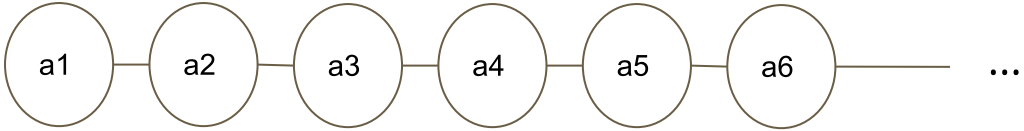
Productive

merge — Merge two streams into a single stream by alternatively outputting an element of each.

split_1 — Return the odd elements of a stream.

split_2 — Return the even elements of a stream.

$$\mathsf{merge}(\mathsf{split}_1(t), \mathsf{split}_2(t)) = t$$



Programming with streams: example of greatest fixed points

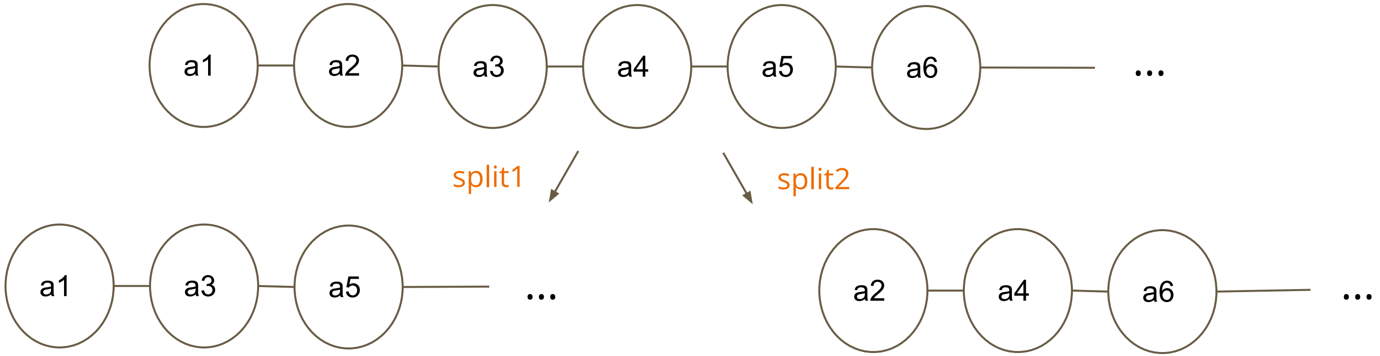
Productive

merge — Merge two streams into a single stream by alternatively outputting an element of each.

split_1 — Return the odd elements of a stream.

split_2 — Return the even elements of a stream.

$$\mathsf{merge}(\mathsf{split}_1(t), \mathsf{split}_2(t)) = t$$



Programming with streams: example of greatest fixed points

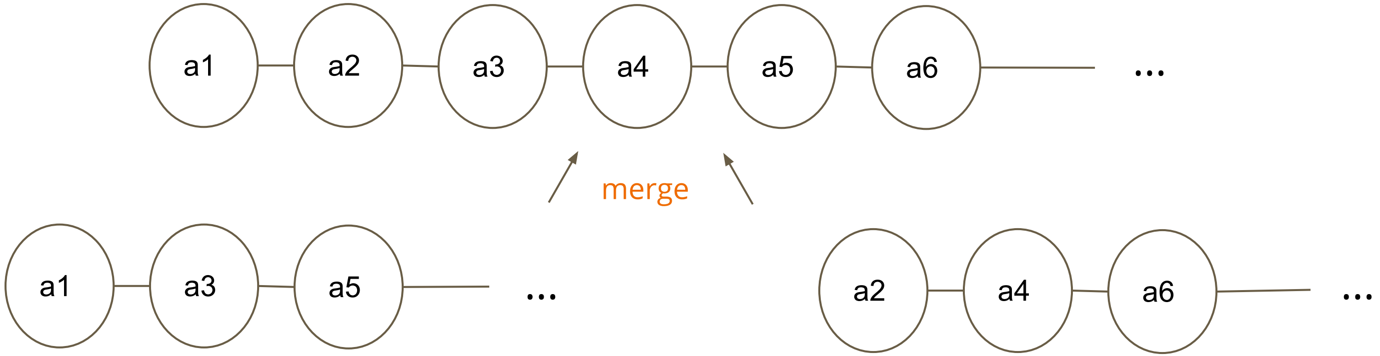
Productive

merge — Merge two streams into a single stream by alternatively outputting an element of each.

split_1 — Return the odd elements of a stream.

split_2 — Return the even elements of a stream.

$$\mathsf{merge}(\mathsf{split}_1(t), \mathsf{split}_2(t)) = t$$



Programming with streams

Define properties of merge and splits as:

$$
\begin{aligned}
\mathtt{Merge}(x, y, z) &=^1_\nu (\mathsf{hd}\, z = \mathsf{hd}\, x \,\&\, \mathtt{Merge}\,(y, \mathsf{tl}\, x, \mathsf{tl}\, z)) \\
\mathtt{Split}_1(x, y) &=^1_\nu (\mathsf{hd}\, y = \mathsf{hd}\, x \,\&\, \mathtt{Split}_2(\mathsf{tl}\, x, \mathsf{tl}\, y)) \\
\mathtt{Split}_2(x, y) &=^1_\nu (1 \,\&\, \mathtt{Split}_1(\mathsf{tl}\, x, y))
\end{aligned}
$$

Operations merge and split□ are inverses

$$\frac{}{\cdot \vdash \mathsf{hd}\,y_1 = \mathsf{hd}\,y_1}\;=R$$

$$\frac{\cdot \vdash \mathsf{hd}\,y_1 = \mathsf{hd}\,y_1}{\mathsf{hd}\,y_1 = \mathsf{hd}\,x \vdash \mathsf{hd}\,x = \mathsf{hd}\,y_1}\;=L$$

$$\frac{\mathsf{hd}\,y_1 = \mathsf{hd}\,x \vdash \mathsf{hd}\,x = \mathsf{hd}\,y_1}{\mathsf{hd}\,y_1 = \mathsf{hd}\,x, 1 \vdash \mathsf{hd}\,x = \mathsf{hd}\,y_1}\;1L$$

$$\frac{\mathsf{hd}\,y_1 = \mathsf{hd}\,x, 1 \vdash \mathsf{hd}\,x = \mathsf{hd}\,y_1}{\mathsf{hd}\,y_1 = \mathsf{hd}\,x, 1\&\mathtt{S}_1(\mathtt{tl}\,x, y_2) \vdash \mathsf{hd}\,x = \mathsf{hd}\,y_1}\;\&L$$

$$\frac{\mathsf{hd}\,y_1 = \mathsf{hd}\,x, 1\&\mathtt{S}_1(\mathtt{tl}\,x, y_2) \vdash \mathsf{hd}\,x = \mathsf{hd}\,y_1}{\mathsf{hd}y_1 = \mathsf{hd}x\&\mathtt{S}_2(\mathsf{tl}x, \mathsf{tl}y_1), 1\&\mathtt{S}_1(\mathtt{tl}x, y_2) \vdash \mathsf{hd}x = \mathsf{hd}y_1}\;\&L$$

$$\frac{\mathtt{S}_2(x, y_2), \mathtt{S}_1(x, y_1) \vdash \mathtt{M}(y_1, y_2, x)}{\mathtt{S}_2(\mathsf{tl}x, \mathsf{tl}y_1), \mathtt{S}_1(\mathtt{tl}x, y_2) \vdash \mathtt{M}(y_2, \mathsf{tl}y_1, \mathsf{tl}x)}\;\mathsf{Sub}_{[\mathsf{tl}x, \mathsf{tl}y_1, y_2 / x, y_2, y_1]}$$

$$\frac{\mathtt{S}_2(\mathsf{tl}x, \mathsf{tl}y_1), \mathtt{S}_1(\mathtt{tl}x, y_2) \vdash \mathtt{M}(y_2, \mathsf{tl}y_1, \mathsf{tl}x)}{\mathtt{S}_2(\mathsf{tl}x, \mathsf{tl}y_1), 1\&\mathtt{S}_1(\mathtt{tl}x, y_2) \vdash \mathtt{M}(y_2, \mathsf{tl}y_1, \mathsf{tl}x)}\;\&L$$

$$\frac{\mathtt{S}_2(\mathsf{tl}x, \mathsf{tl}y_1), 1\&\mathtt{S}_1(\mathtt{tl}x, y_2) \vdash \mathtt{M}(y_2, \mathsf{tl}y_1, \mathsf{tl}x)}{\mathsf{hd}y_1 = \mathsf{hd}x\&\mathtt{S}_2(\mathsf{tl}x, \mathsf{tl}y_1), 1\&\mathtt{S}_1(\mathtt{tl}x, y_2) \vdash \mathtt{M}(y_2, \mathsf{tl}y_1, \mathsf{tl}x)}\;\&L$$

$$\frac{\mathsf{hd}y_1 = \mathsf{hd}x\&\mathtt{S}_2(\mathsf{tl}x, \mathsf{tl}y_1), 1\&\mathtt{S}_1(\mathtt{tl}x, y_2) \vdash \mathsf{hd}x = \mathsf{hd}y_1 \qquad \mathsf{hd}y_1 = \mathsf{hd}x\&\mathtt{S}_2(\mathsf{tl}x, \mathsf{tl}y_1), 1\&\mathtt{S}_1(\mathtt{tl}x, y_2) \vdash \mathtt{M}(y_2, \mathsf{tl}y_1, \mathsf{tl}x)}{\mathsf{hd}\,y_1 = \mathsf{hd}\,x \,\&\, \mathtt{S}_2(\mathsf{tl}\,x, \mathsf{tl}\,y_1), 1\&\mathtt{S}_1(\mathtt{tl}\,x, y_2) \vdash \mathsf{hd}\,x = \mathsf{hd}\,y_1 \,\&\, \mathtt{M}(y_2, \mathsf{tl}\,y_1, \mathsf{tl}\,x)}\;\&R$$

$$\frac{\mathsf{hd}\,y_1 = \mathsf{hd}\,x \,\&\, \mathtt{S}_2(\mathsf{tl}\,x, \mathsf{tl}\,y_1), 1\&\mathtt{S}_1(\mathtt{tl}\,x, y_2) \vdash \mathsf{hd}\,x = \mathsf{hd}\,y_1 \,\&\, \mathtt{M}(y_2, \mathsf{tl}\,y_1, \mathsf{tl}\,x)}{\mathsf{hd}\,y_1 = \mathsf{hd}\,x \,\&\, \mathtt{S}_2(\mathsf{tl}\,x, \mathsf{tl}\,y_1), \mathtt{S}_2(x, y_2) \vdash \mathsf{hd}\,x = \mathsf{hd}\,y_1 \,\&\, \mathtt{M}(y_2, \mathsf{tl}\,y_1, \mathsf{tl}\,x)}\;\nu L$$

$$\frac{\mathsf{hd}\,y_1 = \mathsf{hd}\,x \,\&\, \mathtt{S}_2(\mathsf{tl}\,x, \mathsf{tl}\,y_1), \mathtt{S}_2(x, y_2) \vdash \mathsf{hd}\,x = \mathsf{hd}\,y_1 \,\&\, \mathtt{M}(y_2, \mathsf{tl}\,y_1, \mathsf{tl}\,x)}{\mathtt{S}_1(x, y_1), \mathtt{S}_2(x, y_2) \vdash \mathsf{hd}\,x = \mathsf{hd}\,y_1 \,\&\, \mathtt{M}(y_2, \mathsf{tl}\,y_1, \mathsf{tl}\,x)}\;\nu L$$

$$\frac{\mathtt{S}_1(x, y_1), \mathtt{S}_2(x, y_2) \vdash \mathsf{hd}\,x = \mathsf{hd}\,y_1 \,\&\, \mathtt{M}(y_2, \mathsf{tl}\,y_1, \mathsf{tl}\,x)}{\mathtt{S}_1(x, y_1), \mathtt{S}_2(x, y_2) \vdash \mathtt{M}(y_1, y_2, x)}\;\nu R$$

Programming with mutual least and greatest fixed points

run(x,t): A stream producer where x is the list of operations, and t is the output stream.

Skip one step and do nothing

Put z as the head of output stream and inserts the new list of operations x to the original one.

$$
\begin{aligned}
\mathtt{run}(\cdot\,, t) &=^1_\mu \quad 1 \\
\mathtt{run}(skip; x, t) &=^1_\mu \quad \mathtt{run}(x, t) \\
\mathtt{run}(put\langle x \rangle; y, t) &=^1_\mu \quad \mathtt{nrun}\,(x, y, t) \\
\mathtt{nrun}(x, y, t) &=^2_\nu \quad \mathtt{hd}\, t = \mathtt{o} \,\&\, \mathtt{run}(x; y, \mathtt{tl}\ t)
\end{aligned}
$$

Run on any list of operations produces a (possibly infinite) list of elements "o"

$$\begin{aligned}
\mathtt{run}(\cdot\,, t) &=^1_\mu 1 \\
\mathtt{run}(skip; x, t) &=^1_\mu \mathtt{run}(x, t) \\
\mathtt{run}(put\langle x\rangle; y, t) &=^1_\mu \mathtt{nrun}\,(x, y, t) \\
\mathtt{nrun}(x, y, t) &=^2_\nu \mathtt{hd}\, t = \mathtt{o} \,\&\, \mathtt{run}(x; y, \mathtt{tl}\, t)
\end{aligned}$$

$$\begin{aligned}
\mathtt{list}_\mathtt{o}(t) &=^1_\mu \oplus\{\mathsf{nil} : 1, \mathsf{next} : \mathtt{stream}_\mathtt{o}(t)\} \\
\mathtt{stream}_\mathtt{o}(t) &=^2_\nu \&\{\mathsf{hd} : \mathtt{hd}\, t = \mathtt{o}, \mathsf{tl} : \mathtt{list}_\mathtt{o}\,(\mathtt{tl}\, t)\}
\end{aligned}$$

$$(\dagger)\ \mathtt{run}(x, t) \vdash \mathtt{list}_\mathtt{o}(t)$$

$$(\star)\ \mathtt{nrun}(x, y, t) \vdash \mathtt{stream}_\mathtt{o}(t)$$

Run produces a list$_\circ$ - proof

$$
\begin{aligned}
\mathtt{run}(\cdot\,, t) &=^1_\mu 1\\
\mathtt{run}(\mathit{skip}; x, t) &=^1_\mu \mathtt{run}(x, t)\\
\mathtt{run}(\mathit{put}\langle x\rangle; y, t) &=^1_\mu \mathtt{nrun}\,(x, y, t)\\
\mathtt{nrun}(x, y, t) &=^2_\nu \mathsf{hd}\, t = \mathsf{o} \,\&\, \mathtt{run}(x; y, \mathtt{tl}\, t)
\end{aligned}
$$

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\cdot \vdash 1}\;1R}{\cdot \vdash \oplus\{\mathsf{nil} : 1, \mathsf{next} : \mathtt{stream}_\circ(t)\}}\;\oplus R}{\cdot \vdash \mathtt{list}_\circ(t)}\;\mu_{\mathtt{list}_\circ} R}{1 \vdash \mathtt{list}_\circ(t)}\;1L}{\dagger\, \mathtt{run}(\cdot\,, t) \vdash \mathtt{list}_\circ(t)}\;\mu_{\mathtt{run}} L
$$

$$
\dfrac{\dfrac{\dagger}{\mathtt{run}(x, t) \vdash \mathtt{list}_\circ(t)}}{\dagger\, \mathtt{run}((\mathit{skip}; x), t) \vdash \mathtt{list}_\circ(t)}\;\mu_{\mathtt{run}} L
$$

$$
\dfrac{\dfrac{\dfrac{\dfrac{\star}{\mathtt{nrun}\,(x, y, t) \vdash \mathtt{stream}_\circ(t)}}{\mathtt{nrun}\,(x, y, t) \vdash \oplus\{\mathsf{nil} : 1, \mathsf{next} : \mathtt{stream}_\circ(t)\}}\;\oplus R}{\mathtt{nrun}\,(x, y, t) \vdash \mathtt{list}_\circ(t)}\;\mu_{\mathtt{list}_\circ} R}{\dagger\, \mathtt{run}(\mathit{put}\langle x\rangle; y, t) \vdash \mathtt{list}_\circ(t)}\;\mu_{\mathtt{run}} L
$$

$$
\dfrac{\dfrac{\dfrac{\dfrac{}{\mathsf{hd}\, t = \mathsf{o} \vdash \mathsf{hd}\, t = \mathsf{o}}\;\mathtt{ID}}{\&\{\mathsf{hd} : \mathsf{hd}\, t = \mathsf{o}, \mathsf{tl} : \mathtt{run}(x; y, \mathtt{tl}\, t)\} \vdash \mathsf{hd}\, t = \mathsf{o}}\;\&L}{\mathtt{nrun}\,(x, y, t) \vdash \mathsf{hd}\, t = \mathsf{o}}\;\nu_{\mathtt{nrun}} L \qquad \dfrac{\dfrac{\dfrac{\dagger}{\mathtt{run}(x; y, \mathtt{tl}\, t) \vdash \mathtt{list}_\circ(\mathsf{tl}\, t)}}{\&\{\mathsf{hd} : \mathsf{hd}\, t = \mathsf{o}, \mathsf{tl} : \mathtt{run}(x; y, \mathtt{tl}\, t)\} \vdash \mathtt{list}_\circ(\mathsf{tl}\, t)}\;\&L}{\mathtt{nrun}\,(x, y, t) \vdash \mathtt{list}_\circ(\mathsf{tl}\, t)}\;\nu_{\mathtt{nrun}} L}{\dfrac{\mathtt{nrun}\,(x, y, t) \vdash \&\{\mathsf{hd} : \mathsf{hd}\, t = \mathsf{o},\ \mathsf{tl} : \mathtt{list}_\circ(\mathsf{tl}\, t)\}}{\star\, \mathtt{nrun}\,(x, y, t) \vdash \mathtt{stream}_\circ(t)}\;\nu_{\mathtt{stream}_\circ} R}\;\&R
$$

Strong progress and Validity condition

A process satisfies *strong progress*, if after *finite number of steps*, it either becomes *empty* or attempts to *communicate to the left or right* [2].

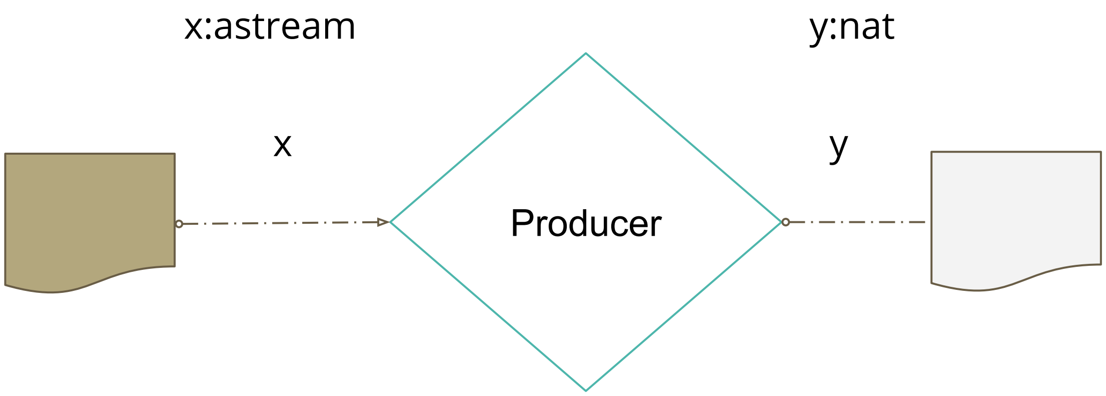
Theorem. Our *validity condition* on session-typed processes ensures *strong progress* [2].

We want to prove this directly using our calculus.

Producer/Idle: a locally valid program

$$\begin{aligned}\Sigma := \mathsf{ack} &=^1_\mu \oplus\{ack : \mathsf{astream}\},\\ \mathsf{astream} &=^2_\nu \&\{head : \mathsf{ack},\quad tail : \mathsf{astream}\},\\ \mathsf{nat} &=^3_\mu \oplus\{z : 1,\quad s : nat\}\end{aligned}$$

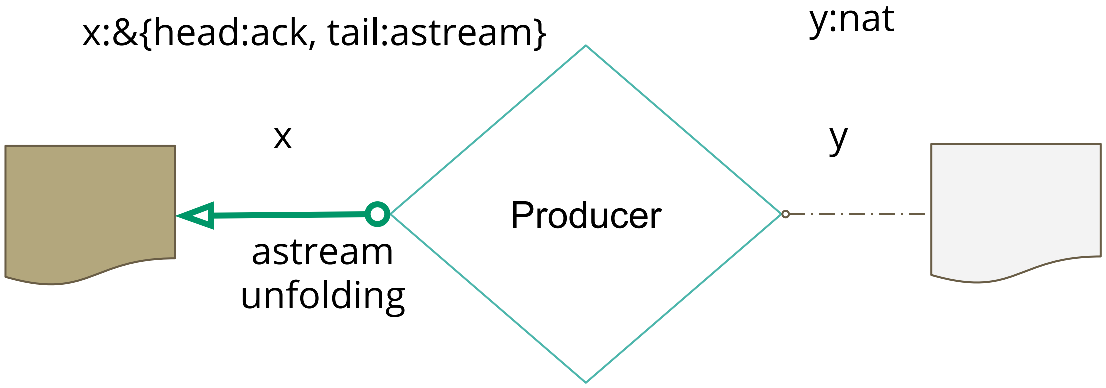
$$z : \mathsf{ack} \vdash w \leftarrow \mathtt{Idle} \leftarrow z :: (w : \mathsf{nat})$$
$$x : \mathsf{astream} \vdash y \leftarrow \mathtt{Producer} \leftarrow x :: (y : \mathsf{nat}),$$



Producer/Idle: a locally valid program

$$\begin{aligned}\Sigma := \mathsf{ack} &=^1_\mu \oplus\{ack : \mathsf{astream}\},\\ \mathsf{astream} &=^2_\nu \&\{head : \mathsf{ack},\quad tail : \mathsf{astream}\},\\ \mathsf{nat} &=^3_\mu \oplus\{z : 1,\quad s : nat\}\end{aligned}$$

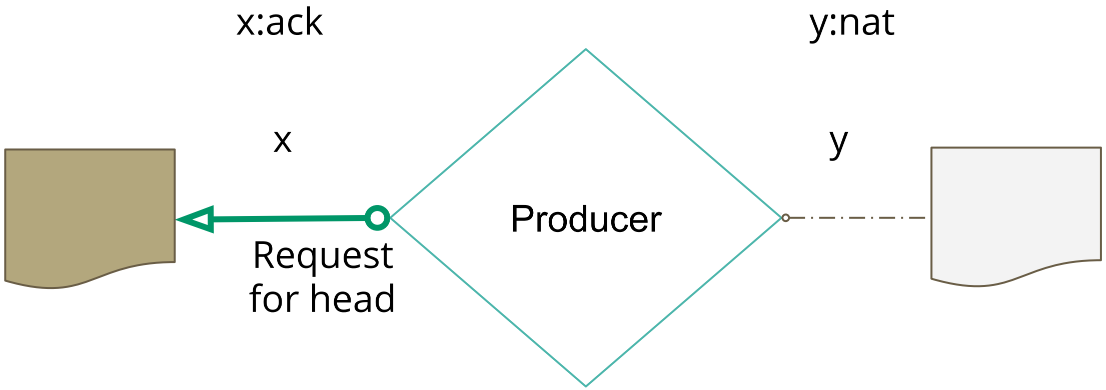
$$z : \mathsf{ack} \vdash w \leftarrow \mathtt{Idle} \leftarrow z :: (w : \mathsf{nat})$$
$$x : \mathsf{astream} \vdash y \leftarrow \mathtt{Producer} \leftarrow x :: (y : \mathsf{nat}),$$



Producer/Idle: a locally valid program

$$\Sigma := \mathsf{ack} =^1_\mu \oplus\{ack : \mathsf{astream}\},$$
$$\mathsf{astream} =^2_\nu \&\{head : \mathsf{ack},\quad tail : \mathsf{astream}\},$$
$$\mathsf{nat} =^3_\mu \oplus\{z : 1,\quad s : nat\}$$

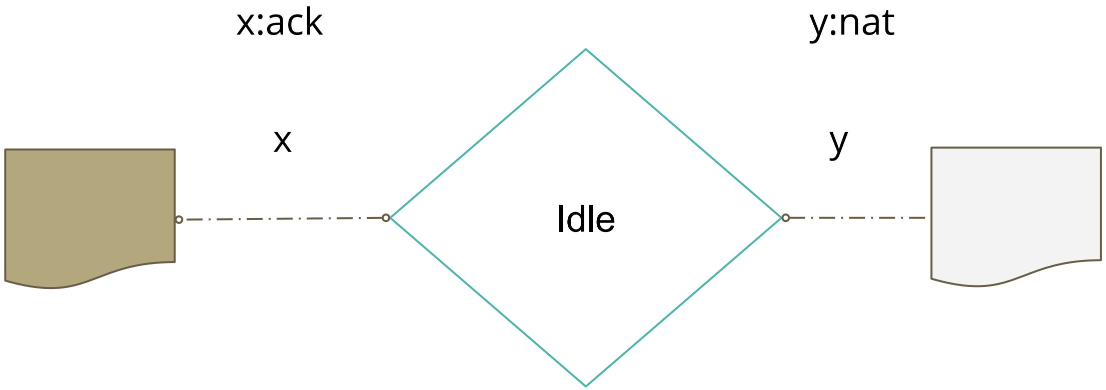
$$z : \mathsf{ack} \vdash w \leftarrow \texttt{Idle} \leftarrow z :: (w : \mathsf{nat})$$
$$x : \mathsf{astream} \vdash y \leftarrow \texttt{Producer} \leftarrow x :: (y : \mathsf{nat}),$$



Producer/Idle: a locally valid program

$$\Sigma := \mathsf{ack} =^1_\mu \oplus\{ack : \mathsf{astream}\},$$
$$\mathsf{astream} =^2_\nu \&\{head : \mathsf{ack}, \quad tail : \mathsf{astream}\},$$
$$\mathsf{nat} =^3_\mu \oplus\{z : 1, \quad s : nat\}$$

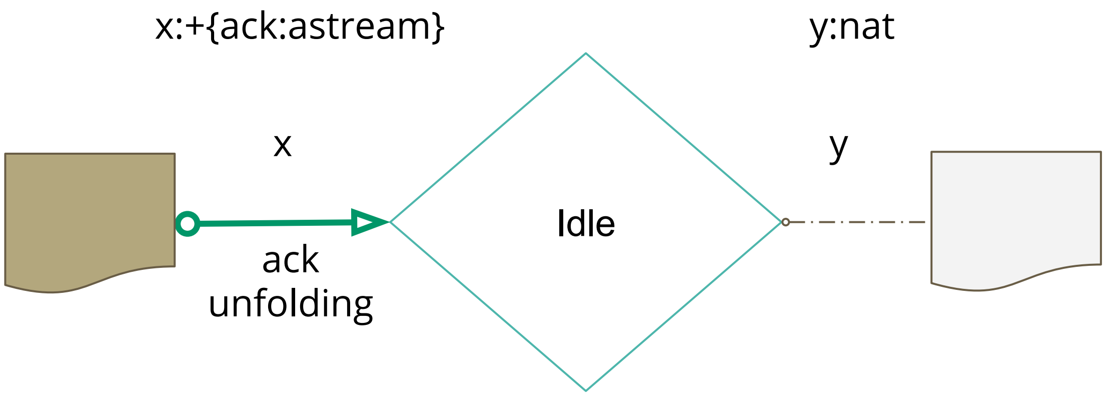
$$z : \mathsf{ack} \vdash w \leftarrow \mathtt{Idle} \leftarrow z :: (w : \mathsf{nat})$$
$$x : \mathsf{astream} \vdash y \leftarrow \mathtt{Producer} \leftarrow x :: (y : \mathsf{nat}),$$



Producer/Idle: a locally valid program

$$\Sigma := \mathsf{ack} =^1_\mu \oplus\{ack : \mathsf{astream}\},$$
$$\mathsf{astream} =^2_\nu \&\{head : \mathsf{ack}, \quad tail : \mathsf{astream}\},$$
$$\mathsf{nat} =^3_\mu \oplus\{z : 1, \quad s : nat\}$$

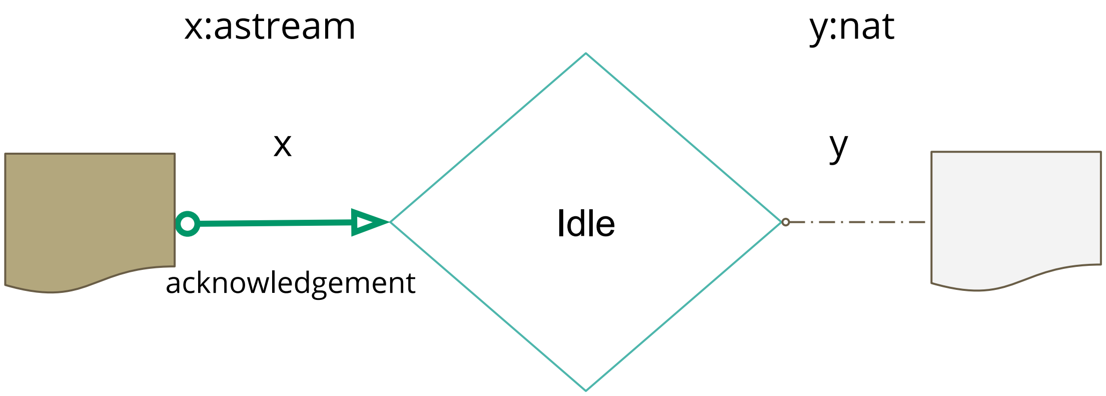
$$z : \mathsf{ack} \vdash w \leftarrow \mathtt{Idle} \leftarrow z :: (w : \mathsf{nat})$$
$$x : \mathsf{astream} \vdash y \leftarrow \mathtt{Producer} \leftarrow x :: (y : \mathsf{nat}),$$



Producer/Idle: a locally valid program

$$\Sigma := \mathsf{ack} =^1_\mu \oplus\{ack : \mathsf{astream}\},$$
$$\mathsf{astream} =^2_\nu \&\{head : \mathsf{ack},\quad tail : \mathsf{astream}\},$$
$$\mathsf{nat} =^3_\mu \oplus\{z : 1,\quad s : nat\}$$

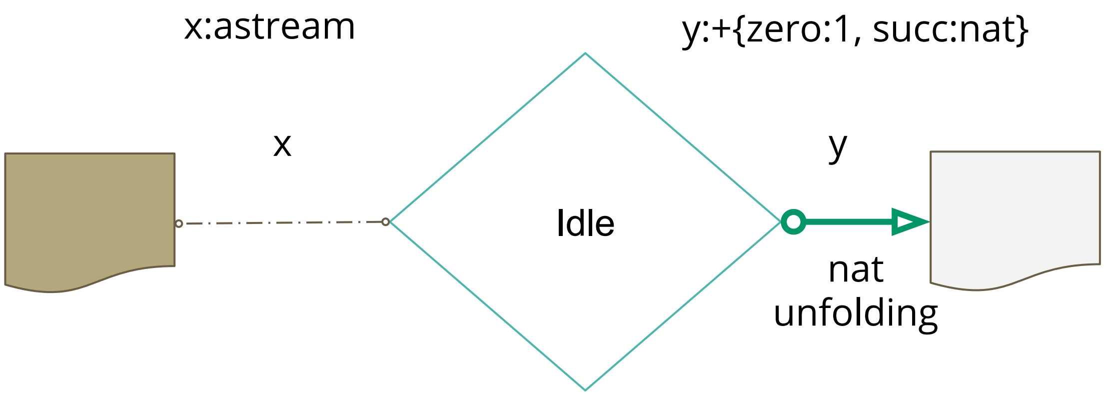
$$z : \mathsf{ack} \vdash w \leftarrow \mathtt{Idle} \leftarrow z :: (w : \mathsf{nat})$$
$$x : \mathsf{astream} \vdash y \leftarrow \mathtt{Producer} \leftarrow x :: (y : \mathsf{nat}),$$



Producer/Idle: a locally valid program

$$\Sigma := \mathsf{ack} =^1_\mu \oplus\{ack : \mathsf{astream}\},$$
$$\mathsf{astream} =^2_\nu \&\{head : \mathsf{ack}, \quad tail : \mathsf{astream}\},$$
$$\mathsf{nat} =^3_\mu \oplus\{z : 1, \quad s : nat\}$$

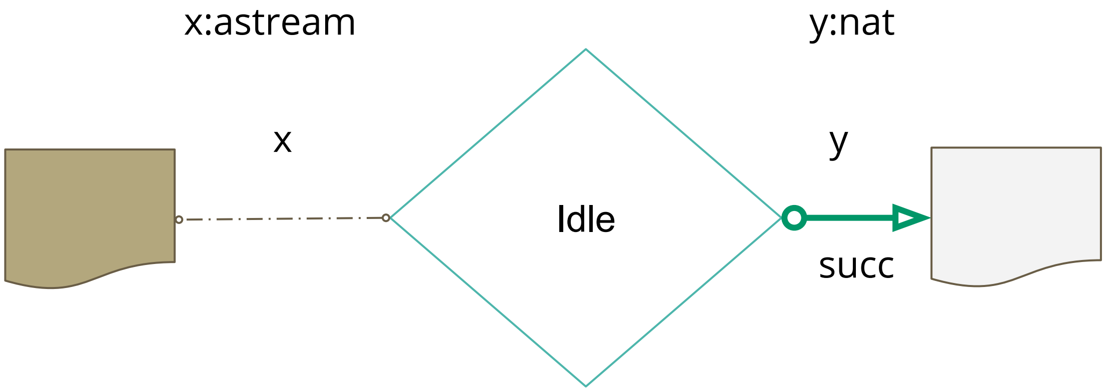
$$z : \mathsf{ack} \vdash w \leftarrow \mathtt{Idle} \leftarrow z :: (w : \mathsf{nat})$$
$$x : \mathsf{astream} \vdash y \leftarrow \mathtt{Producer} \leftarrow x :: (y : \mathsf{nat}),$$



Producer/Idle: a locally valid program

$$\begin{aligned}\Sigma := \ & \mathsf{ack} =^1_\mu \oplus\{ack : \mathsf{astream}\},\\ & \mathsf{astream} =^2_\nu \&\{head : \mathsf{ack},\quad tail : \mathsf{astream}\},\\ & \mathsf{nat} =^3_\mu \oplus\{z : 1,\quad s : nat\}\end{aligned}$$

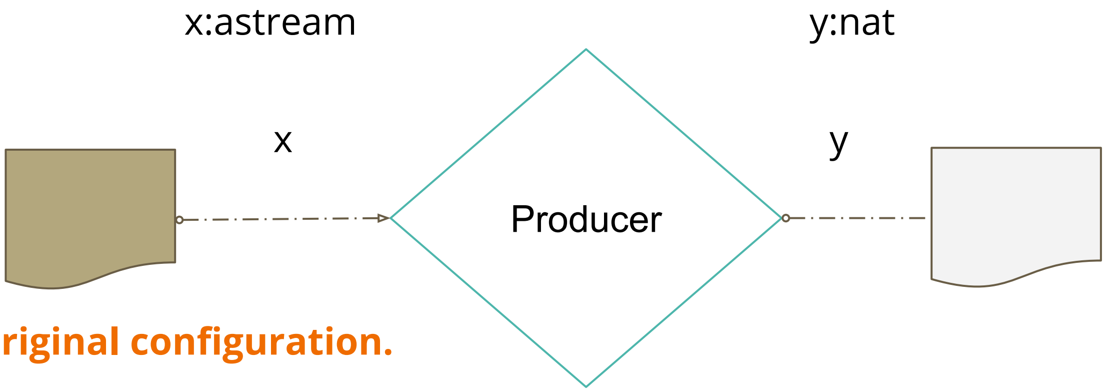
$$z : \mathsf{ack} \vdash w \leftarrow \mathtt{Idle} \leftarrow z :: (w : \mathsf{nat})$$
$$x : \mathsf{astream} \vdash y \leftarrow \mathtt{Producer} \leftarrow x :: (y : \mathsf{nat}),$$



Producer/Idle: a locally valid program

$$\Sigma := \mathsf{ack} =^1_\mu \oplus\{ack : \mathsf{astream}\},$$
$$\mathsf{astream} =^2_\nu \&\{head : \mathsf{ack}, \quad tail : \mathsf{astream}\},$$
$$\mathsf{nat} =^3_\mu \oplus\{z : 1, \quad s : nat\}$$

$$z : \mathsf{ack} \vdash w \leftarrow \mathtt{Idle} \leftarrow z :: (w : \mathsf{nat})$$
$$x : \mathsf{astream} \vdash y \leftarrow \mathtt{Producer} \leftarrow x :: (y : \mathsf{nat}),$$



Back to the original configuration.

Producer/Idle: a locally valid program - code

$$\Sigma := \text{ack} =^1_\mu \oplus\{ack : \text{astream}\},$$
$$\text{astream} =^2_\nu \&\{head : \text{ack}, \quad tail : \text{astream}\},$$
$$\text{nat} =^3_\mu \oplus\{z : 1, \quad s : nat\}$$

Eventually communicate with its external channels

$z : \text{ack} \vdash w \leftarrow \texttt{Idle} \leftarrow z :: (w : \text{nat})$
$x : \text{astream} \vdash y \leftarrow \texttt{Producer} \leftarrow x :: (y : \text{nat}),$

$$y^0 \leftarrow \texttt{Producer} \leftarrow x^0 = \\ \quad Lx^0.\nu_{astream}; \\ \quad\quad Lx^1.head; y^0 \leftarrow \texttt{Idle} \leftarrow x^1$$

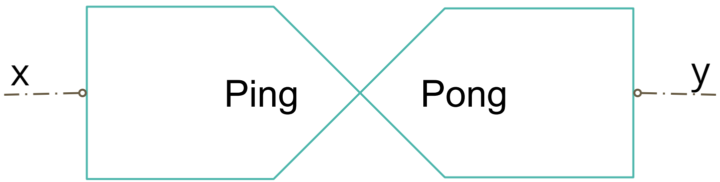
$$y^0 \leftarrow \texttt{Idle} \leftarrow x^1 = \\ \quad \mathbf{case}\ Lx^1\ (\mu_{ack} \Rightarrow \\ \quad\quad \mathbf{case}\ Lx^2\ (ack \Rightarrow Ry^0.\mu_{nat}; \\ \quad\quad\quad Ry^1.s; y^1 \leftarrow \texttt{Producer} \leftarrow x^2))$$

This example is adapted from [2].

Ping-Pong: an invalid program

$$\Sigma := \mathsf{ack} =^1_\mu \oplus\{ack : \mathsf{astream}\},$$
$$\mathsf{astream} =^2_\nu \&\{head : \mathsf{ack},\quad tail : \mathsf{astream}\},$$
$$\mathsf{nat} =^3_\mu \oplus\{z : 1,\quad s : nat\}$$

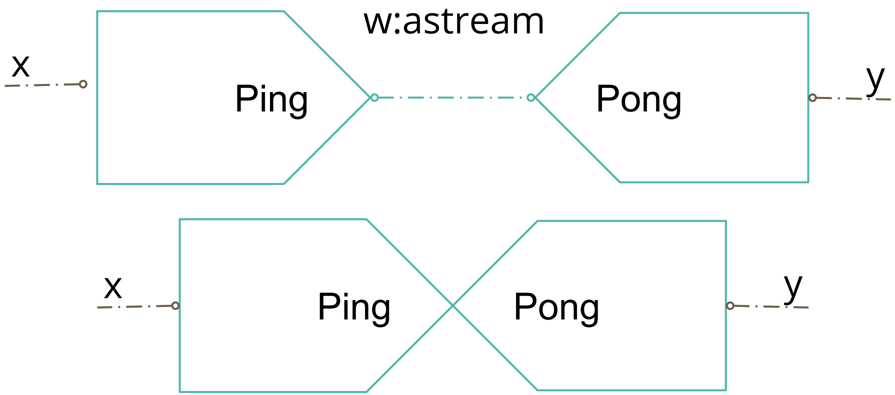
$$x : \mathsf{nat} \vdash \mathtt{Ping} :: (w : \mathsf{astream})$$
$$w : \mathsf{astream} \vdash \mathtt{Pong} :: (y : \mathsf{nat})$$
$$x : \mathsf{nat} \vdash \mathtt{PingPong} :: (y : \mathsf{nat})$$



Ping-Pong: an invalid program

$$\Sigma := \mathsf{ack} =^1_\mu \oplus\{ack : \mathsf{astream}\},$$
$$\mathsf{astream} =^2_\nu \&\{head : \mathsf{ack},\quad tail : \mathsf{astream}\},$$
$$\mathsf{nat} =^3_\mu \oplus\{z : 1,\quad s : nat\}$$

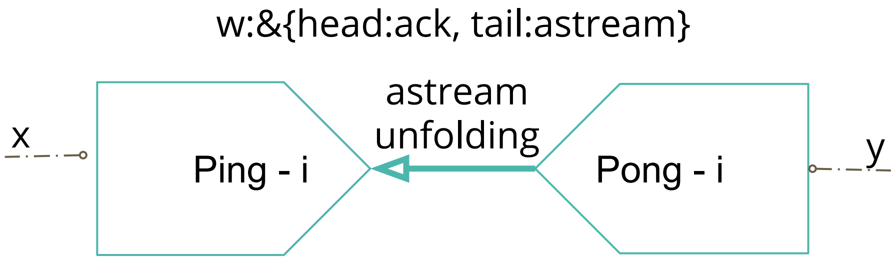
$$x : \mathsf{nat} \vdash \mathtt{Ping} :: (w : \mathsf{astream})$$
$$w : \mathsf{astream} \vdash \mathtt{Pong} :: (y : \mathsf{nat})$$
$$x : \mathsf{nat} \vdash \mathtt{PingPong} :: (y : \mathsf{nat})$$



Ping-Pong: an invalid program

$$\Sigma := \mathsf{ack} =^1_\mu \oplus\{ack : \mathsf{astream}\},$$
$$\mathsf{astream} =^2_\nu \&\{head : \mathsf{ack}, \quad tail : \mathsf{astream}\},$$
$$\mathsf{nat} =^3_\mu \oplus\{z : 1, \quad s : nat\}$$

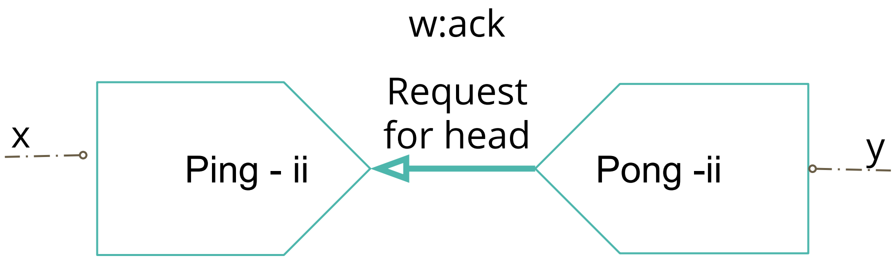
$$x : \mathsf{nat} \vdash \mathtt{Ping} :: (w : \mathsf{astream})$$
$$w : \mathsf{astream} \vdash \mathtt{Pong} :: (y : \mathsf{nat})$$
$$x : \mathsf{nat} \vdash \mathtt{PingPong} :: (y : \mathsf{nat})$$



Ping-Pong: an invalid program

$$\Sigma := \mathsf{ack} =^1_\mu \oplus\{ack : \mathsf{astream}\},$$
$$\mathsf{astream} =^2_\nu \&\{head : \mathsf{ack}, \quad tail : \mathsf{astream}\},$$
$$\mathsf{nat} =^3_\mu \oplus\{z : 1, \quad s : nat\}$$

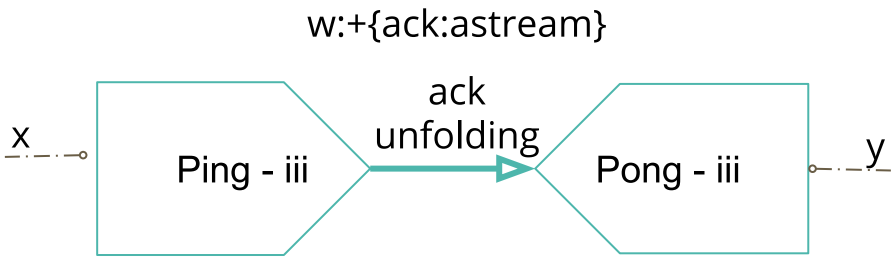
$$x : \mathsf{nat} \vdash \mathtt{Ping} :: (w : \mathsf{astream})$$
$$w : \mathsf{astream} \vdash \mathtt{Pong} :: (y : \mathsf{nat})$$
$$x : \mathsf{nat} \vdash \mathtt{PingPong} :: (y : \mathsf{nat})$$



Ping-Pong: an invalid program

$$\Sigma := \mathsf{ack} =^1_\mu \oplus\{ack : \mathsf{astream}\},$$
$$\mathsf{astream} =^2_\nu \&\{head : \mathsf{ack}, \quad tail : \mathsf{astream}\},$$
$$\mathsf{nat} =^3_\mu \oplus\{z : 1, \quad s : nat\}$$

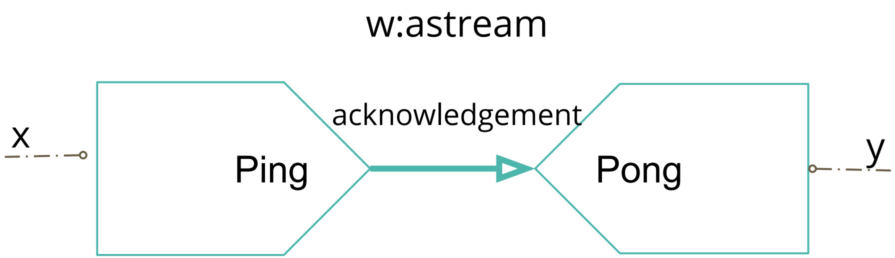
$$x : \mathsf{nat} \vdash \mathtt{Ping} :: (w : \mathsf{astream})$$
$$w : \mathsf{astream} \vdash \mathtt{Pong} :: (y : \mathsf{nat})$$
$$x : \mathsf{nat} \vdash \mathtt{PingPong} :: (y : \mathsf{nat})$$



Ping-Pong: an invalid program

$$\Sigma := \mathsf{ack} =^1_\mu \oplus\{ack : \mathsf{astream}\},$$
$$\mathsf{astream} =^2_\nu \&\{head : \mathsf{ack}, \quad tail : \mathsf{astream}\},$$
$$\mathsf{nat} =^3_\mu \oplus\{z : 1, \quad s : nat\}$$

$$x : \mathsf{nat} \vdash \mathtt{Ping} :: (w : \mathsf{astream})$$
$$w : \mathsf{astream} \vdash \mathtt{Pong} :: (y : \mathsf{nat})$$
$$x : \mathsf{nat} \vdash \mathtt{PingPong} :: (y : \mathsf{nat})$$



Back to the original configuration.

Ping-Pong: an invalid program - code

$y \leftarrow \texttt{PingPong} \leftarrow x =$

$\quad w \leftarrow \texttt{Ping} \leftarrow x;$ $\qquad \%\ \textit{spawn process}\ \texttt{Ping}_w$

$\quad\quad y \leftarrow \texttt{Pong} \leftarrow w$ $\qquad \%\ \textit{continue with a tail call}$

Keep calling itself without communicating with its external channels

$w \leftarrow \texttt{Ping} \leftarrow x =$	$[0,0,0,0,0,0]$
$\quad \textbf{case}\ Rw\ (\nu_{astream} \Rightarrow$	$[0,0,-1,0,0,0]$
$\quad\quad \textbf{case}\ Rw\ (head \Rightarrow Rw.\mu_{ack};$	$[0,1,-1,0,0,0]$
$\quad\quad\quad Rw.ack; w \leftarrow \texttt{Ping} \leftarrow x$	$[0,1,-1,0,0,0]$
$\quad\quad \mid tail \Rightarrow w \leftarrow \texttt{Ping} \leftarrow x))$	$[0,0,-1,0,0,0]$

$y \leftarrow \texttt{Pong} \leftarrow w =$	$[0,0,0,0,0,0]$
$\quad Lw.\nu_{astream};$	$[0,0,0,1,0,0]$
$\quad Lw.head;$	$[0,0,0,1,0,0]$
$\quad \textbf{case}\ Lw\ (\mu_{ack} \Rightarrow$	$[-1,0,0,1,0,0]$
$\quad\quad \textbf{case}\ Lw\ ($	$[-1,0,0,1,0,0]$
$\quad\quad\quad ack \Rightarrow Ry.\mu_{nat};$	$[-1,0,0,1,0,1]$
$\quad\quad\quad Ry.s;$	$[-1,0,0,1,0,1]$
$\quad\quad\quad y \leftarrow \texttt{Pong} \leftarrow w))$	$[-1,0,0,1,0,1]$

A valid configuration of processes satisfies strong progress

We define strong progress as a predicate

$$\mathcal{C} \in [\![x : A]\!]$$

$$\cdot \vdash \mathcal{C} :: (x{:}A) \quad \Longleftrightarrow_{\text{Bisimulation}} \quad \cdot \vdash \mathcal{C} \in [\![x : A]\!]$$

Theorem. *If configuration C is well-typed then there is an infinite derivation for its strong progress property. Moreover, if it C is valid, the infinite derivation is a proof.*

Conclusion

We introduced an infinitary sequent calculus for first order intuitionistic multiplicative additive linear logic with fixed points [2].

Our main motivation for introducing this calculus is to reason about programs behaviour. In particular we use this calculus to give a direct proof for the strong progress property of locally valid binary session typed processes [2]. The importance of a direct proof other than its elegance is that it can be adapted for a more general validity condition on processes without the need to prove cut elimination productivity for their underlying derivations.

Send me an Email!

fderakhs@andrew.cmu.edu

References

1. Frank Pfenning. Substructural logics. Lecture notes for course given at Carnegie Mellon University, Fall 2016, December 2016.
2. Farzaneh Derakhshan and Frank Pfenning. 2019. Circular Proofs as Session-Typed Processes: A Local Validity Condition. arXiv preprint arXiv:1908.01909 (2019).
3. Farzaneh Derakhshan and Frank Pfenning. 2020. Circular Proofs in First-Order Linear Logic with Least and Greatest Fixed Points. arXiv preprint arXiv:2001.05132 (2020).
4. Andreas Abel and Brigitte Pientka. 2016. Well-founded recursion with copatterns and sized types. Journal of Functional Programming 26(2016).
5. David Baelde and Dale Miller. 2007. Least and greatest fixed points in linear logic. In International Conference on Logic for Programming Artificial Intelligence and Reasoning. Springer, 92–106
6. James Brotherston. 2005. Cyclic proofs for first-order logic with inductive definitions. In International Conference on Automated Reasoning with Analytic Tableaux and Related Methods. Springer, 78–92.
7. Dexter Kozen and Alexandra Silva. 2017. Practical coinduction.Mathematical Structures in Computer Science 27, 7 (2017), 1132–1152.