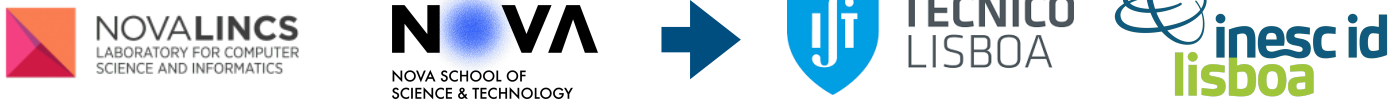
Shared state Session-based programs that can prove themselves

Luís Caires



Thanks F Pfenning, B Toninho, P Rocha, J A Perez

Open Problems in Concurrency Theory Bertinoro, June 2023

Propositions-as-Types for Concurrency

- *Bridge* between Logic, Programming Languages, and Computation.
- Programs are proofs in a logic, according to a Curry-Howard correspondence
  - program as a typed semantically well-behaved object (a function or a process)
  - proof simplification as computation ➡
    - preservation, progress, confluence
  - computation as cut-elimination ➡
    - logical relations semantics, termination
    - equational reasoning about observational equivalence

Propositions-as-Types for Concurrency

- *Bridge* between Logic, Programming Languages, and Computation.
- Programs are proofs in a logic, according to a Curry-Howard correspondence

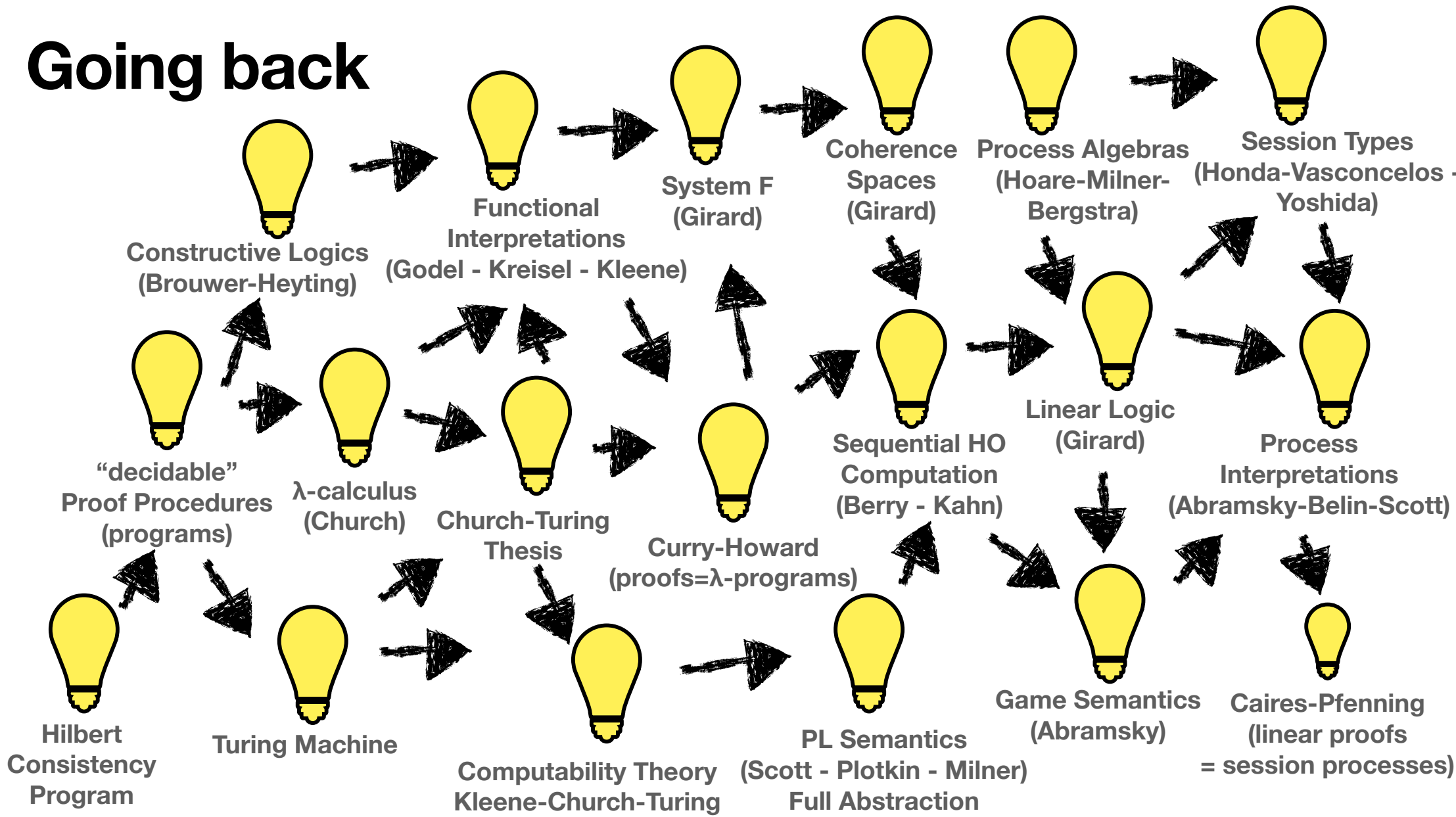
  - modular extensions (logically inspired connectives “automatically” socialize)
    - **polymorphism** (generics)
    - **dependent types** (assertions, certificates, …)
    - …
    - **shared state** concurrency and **non-determinism** (inspired in DILL [Erhard])

Propositions-as-Types for Concurrency

- *Bridge* between Logic, Programming Languages, and Computation.
- Programs are proofs in a logic, according to a Curry-Howard correspondence

  - connecting session types to the trunk "classical" of computation and PL theory
    - typed λ calculus: **sequential** ho computation with **pure values**
    - typed session calculus: **concurrent** ho computation with **linear resources**
    - former subsumes latter, via exponentials and sharing constructs
    - typed process-based infrastructure for safe concurrent programming

Going back


Constructive Logics
(Brouwer-Heyting)
Functional
Interpretations
(Godel - Kreisel - Kleene)
System F
(Girard)
Coherence
Spaces
(Girard)
Process Algebras
(Hoare-Milner-
Bergstra)
Session Types
(Honda-Vasconcelos -
Yoshida)
"decidable"
Proof Procedures
(programs)
λ-calculus
(Church)
Church-Turing
Thesis
Curry-Howard
(proofs=λ-programs)
Sequential HO
Computation
(Berry - Kahn)
Linear Logic
(Girard)
Process
Interpretations
(Abramsky-Belin-Scott)
Hilbert
Consistency
Program
Turing Machine
Computability Theory
Kleene-Church-Turing
PL Semantics
(Scott - Plotkin - Milner)
Full Abstraction
Game Semantics
(Abramsky)
Caires-Pfenning
(linear proofs
= session processes)

Propositions-as-Types for Concurrency

- *Bridge* between Logic, Programming Languages, and Computation.
- Programs are proofs in a logic, according to a Curry-Howard correspondence

  - connecting session types to the trunk "classical" of computation and PL theory
    - typed λ calculus: **sequential** ho computation with **pure values**
    - typed session calculus: **concurrent** ho computation with **linear resources**
    - former subsumes latter, via exponentials and sharing constructs
    - typed process-based infrastructure for safe concurrent programming

A Session Programming Language from Linear Logic

Process expressions (basic)

Logical Type	PL type	PL construct	
$\mathbf{1}$	close	close x	Close x.
$\bot$	wait	wait $x; P$	Wait on x, continue as P.
$A \,\&\, B$	offer $\{\vert$#inl : $\hat{A}$	case x $\{\vert$inl : P	Case on x: left and continue as P;
	$\vert$#inr : $\hat{B}\}$	$\vert$inr : $Q\}$	or right and continue as Q.
$A \oplus B$	case$\{\vert$#inl : $\hat{A}$	#inl $x; P$	Choose left on x, continue as P.
	$\vert$#inr : $\hat{B}\}$	#inr $x; P$	Choose right on x, continue as P.
$A \otimes B$	send $\hat{A}; \hat{B}$	send $x(y.P); Q$	Send y on x, continue as Q.
$A \parr B$	recv $\hat{A}; \hat{B}$	recv $x(y); P$	Receive y on x, continue as P.

Process expressions (basic)

Logical Type	PL type	PL construct	
$\mathbf{1}$	close	close x	Close x.
$\bot$	wait	wait $x; P$	Wait on x, continue as P.
$A \,\&\, B$	offer $\{\vert\#\text{inl} : \hat{A}$	case x $\{\vert\text{inl} : P$	Case on x: left and continue as P;
	$\vert\#\text{inr} : \hat{B}\}$	$\vert\text{inr} : Q\}$	or right and continue as Q.
$A \oplus B$	case$\{\vert\#\text{inl} : \hat{A}$	#inl $x; P$	Choose left on x, continue as P.
	$\vert\#\text{inr} : \hat{B}\}$	#inr $x; P$	Choose right on x, continue as P.
$A \otimes B$	send $\hat{A}; \hat{B}$	send $x(y.P); Q$	Send y on x, continue as Q.
$A \parr B$	recv $\hat{A}; \hat{B}$	recv $x(y); P$	Receive y on x, continue as P.

Process expressions (basic)

Logical Type	PL type	PL construct	
$\mathbf{1}$	close	close x	Close x.
$\bot$	wait	wait $x; P$	Wait on x, continue as P.
$A \,\&\, B$	offer $\{\vert\#\text{inl} : \hat{A}$	case x $\{\vert\text{inl} : P$	Case on x: left and continue as P;
	$\vert\#\text{inr} : \hat{B}\}$	$\vert\text{inr} : Q\}$	or right and continue as Q.
$A \oplus B$	case$\{\vert\#\text{inl} : \hat{A}$	#inl $x; P$	Choose left on x, continue as P.
	$\vert\#\text{inr} : \hat{B}\}$	#inr $x; P$	Choose right on x, continue as P.
$A \otimes B$	send $\hat{A}; \hat{B}$	send $x(y.P); Q$	Send y on x, continue as Q.
$A \parr B$	recv $\hat{A}; \hat{B}$	recv $x(y); P$	Receive y on x, continue as P.

Process expressions (basic)

Logical Type	PL type	PL construct	
$\mathbf{1}$	close	close x	Close x.
$\bot$	wait	wait $x; P$	Wait on x, continue as P.
$A \,\&\, B$	offer $\{\lvert\#\mathsf{inl} : \hat{A}$ $\lvert\#\mathsf{inr} : \hat{B}\}$	case x $\{\lvert\mathsf{inl} : P$ $\lvert\mathsf{inr} : Q\}$	Case on x: left and continue as P; or right and continue as Q.
$A \oplus B$	case$\{\lvert\#\mathsf{inl} : \hat{A}$ $\lvert\#\mathsf{inr} : \hat{B}\}$	$\#\mathsf{inl}\ x; P$	Choose left on x, continue as P.
		$\#\mathsf{inr}\ x; P$	Choose right on x, continue as P.
$A \otimes B$	send $\hat{A}; \hat{B}$	send $x(y.P); Q$	Send y on x, continue as Q.
$A \parr B$	recv $\hat{A}; \hat{B}$	recv $x(y); P$	Receive y on x, continue as P.

Process expressions (replication, affinity, state)

$!A$	$!\hat{A}$	$!x(y);P$	Replicated session on x with parameter y.
$?A$	$?\hat{A}$	$?x;P$	Make x unrestricted, continue as P.
		call $x(y);P$	Call x with input y, continue as P.
$\wedge A$	affine $\hat{A}$	affine $x;P$	Affine on x, continue as P.
$\vee A$	coaffine $\hat{A}$	discard x	Discard x.
		use $x;P$	Use x, continue as P.
$\mathbf{S}_f\ A$	state $\hat{A}$	cell $x(y.P)$	Cell on x storing y.
$\mathbf{U}_f\ A$	usage $\hat{A}$	release x	Release x.
		take $x(y);P$	Take y from x, continue as P.
$\mathbf{S}_e\ A$	statel $\hat{A}$	empty x	Empty cell on x.
$\mathbf{U}_e\ A$	usagel $\hat{A}$	put $x(y.P);Q$	Puts y in x, continues as Q.

Process expressions (replication, affinity, state)

$!A$	$!\hat{A}$	$!x(y);P$	Replicated session on x with parameter y.
$?A$	$?\hat{A}$	$?x;P$	Make x unrestricted, continue as P.
		call $x(y);P$	Call x with input y, continue as P.
$\wedge A$	affine $\hat{A}$	affine $x;P$	Affine on x, continue as P.
$\vee A$	coaffine $\hat{A}$	discard x	Discard x.
		use $x;P$	Use x, continue as P.
$\mathbf{S}_f\ A$	state $\hat{A}$	cell $x(y.P)$	Cell on x storing y.
$\mathbf{U}_f\ A$	usage $\hat{A}$	release x	Release x.
		take $x(y);P$	Take y from x, continue as P.
$\mathbf{S}_e\ A$	statel $\hat{A}$	empty x	Empty cell on x.
$\mathbf{U}_e\ A$	usagel $\hat{A}$	put $x(y.P);Q$	Puts y in x, continues as Q.

Process expressions (replication, affinity, state)

$!A$	$!\hat{A}$	$!x(y);P$	Replicated session on x with parameter y.
$?A$	$?\hat{A}$	$?x;P$	Make x unrestricted, continue as P.
		call $x(y);P$	Call x with input y, continue as P.
$\wedge A$	affine $\hat{A}$	affine $x;P$	Affine on x, continue as P.
$\vee A$	coaffine $\hat{A}$	discard x	Discard x.
		use $x;P$	Use x, continue as P.
$\mathbf{S}_f\ A$	state $\hat{A}$	cell $x(y.P)$	Cell on x storing y.
$\mathbf{U}_f\ A$	usage $\hat{A}$	release x	Release x.
		take $x(y);P$	Take y from x, continue as P.
$\mathbf{S}_e\ A$	statel $\hat{A}$	empty x	Empty cell on x.
$\mathbf{U}_e\ A$	usagel $\hat{A}$	put $x(y.P);Q$	Puts y in x, continues as Q.

Process expressions (replication, affinity, state)

$!A$	$!\hat{A}$	$!x(y);P$	Replicated session on x with parameter y.
$?A$	$?\hat{A}$	$?x;P$	Make x unrestricted, continue as P.
		call $x(y);P$	Call x with input y, continue as P.
$\wedge A$	affine $\hat{A}$	affine $x;P$	Affine on x, continue as P.
$\vee A$	coaffine $\hat{A}$	discard x	Discard x.
		use $x;P$	Use x, continue as P.
$\mathbf{S}_f\ A$	state $\hat{A}$	cell $x(y.P)$	Cell on x storing y.
$\mathbf{U}_f\ A$	usage $\hat{A}$	release x	Release x.
		take $x(y);P$	Take y from x, continue as P.
$\mathbf{S}_e\ A$	statel $\hat{A}$	empty x	Empty cell on x.
$\mathbf{U}_e\ A$	usagel $\hat{A}$	put $x(y.P);Q$	Puts y in x, continues as Q.

Standard affinity monad

Process expressions (replication, affinity, state)

$!A$	$!\hat{A}$	$!x(y);P$	Replicated session on x with parameter y.
$?A$	$?\hat{A}$	$?x;P$	Make x unrestricted, continue as P.
		call $x(y);P$	Call x with input y, continue as P.
$\wedge A$	affine $\hat{A}$	affine $x;P$	Affine on x, continue as P.
$\vee A$	coaffine $\hat{A}$	discard x	Discard x.
		use $x;P$	Use x, continue as P.
$\mathbf{S}_f\ A$	state $\hat{A}$	cell $x(y.P)$	Cell on x storing y.
$\mathbf{U}_f\ A$	usage $\hat{A}$	release x	Release x.
		take $x(y);P$	Take y from x, continue as P.
$\mathbf{S}_e\ A$	statel $\hat{A}$	empty x	Empty cell on x.
$\mathbf{U}_e\ A$	usagel $\hat{A}$	put $x(y.P);Q$	Puts y in x, continues as Q.

Process expressions (replication, affinity, state)

$!A$	$!\hat{A}$	$!x(y);P$	Replicated session on x with parameter y.
$?A$	$?\hat{A}$	$?x;P$	Make x unrestricted, continue as P.
		call $x(y);P$	Call x with input y, continue as P.
$\wedge A$	affine $\hat{A}$	affine $x;P$	Affine on x, continue as P.
$\vee A$	coaffine $\hat{A}$	discard x	Discard x.
		use $x;P$	Use x, continue as P.
$\mathbf{S}_f\ A$	state $\hat{A}$	cell $x(y.P)$	Cell on x storing y.
$\mathbf{U}_f\ A$	usage $\hat{A}$	release x	Release x.
		take $x(y);P$	Take y from x, continue as P.
$\mathbf{S}_e\ A$	statel $\hat{A}$	empty x	Empty cell on x.
$\mathbf{U}_e\ A$	usagel $\hat{A}$	put $x(y.P);Q$	Puts y in x, continues as Q.

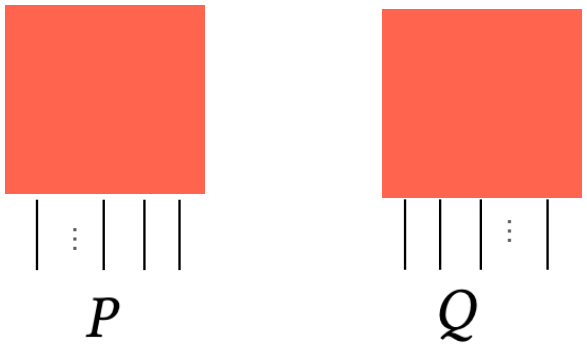
Inspired by Differential LL

Process expressions (recursion, corecursion)

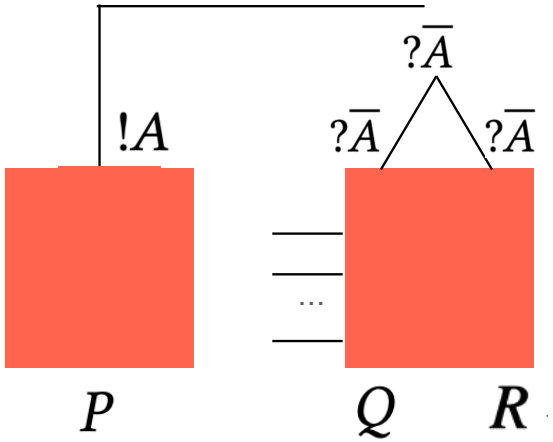
$\nu X.\ A$	corec $X(z, \vec{w}); P$
$\mu X.\ A$	unfold$_{\mu}$ $x; P$

Logical Composition Forms

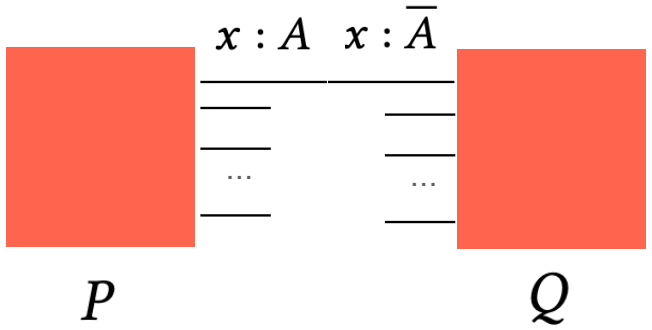
Mix



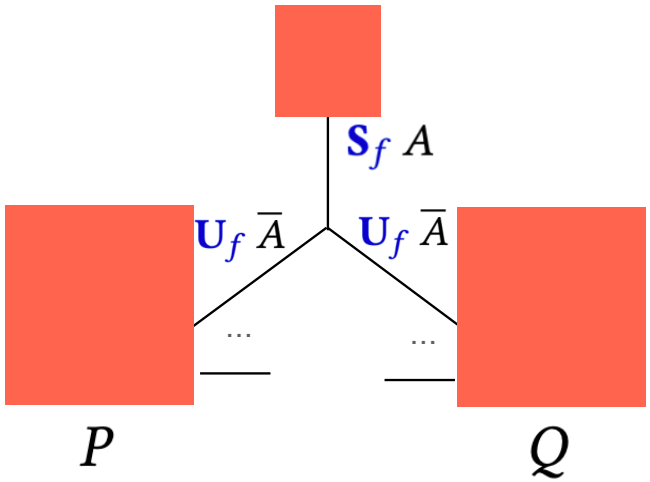
Contraction



Linear Cut

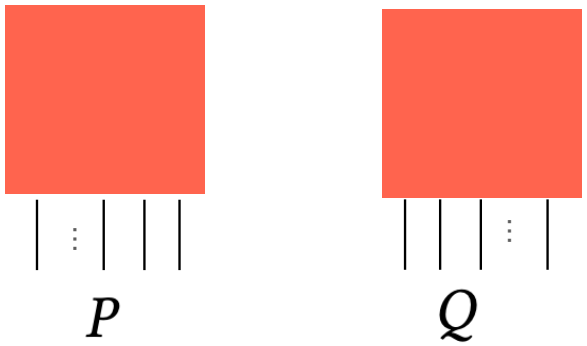


Co-Contraction

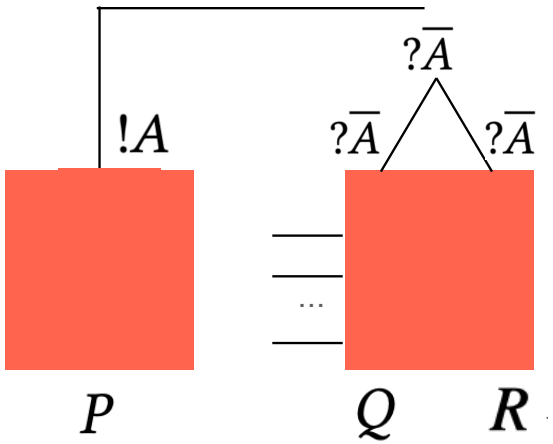


Logical Composition Forms

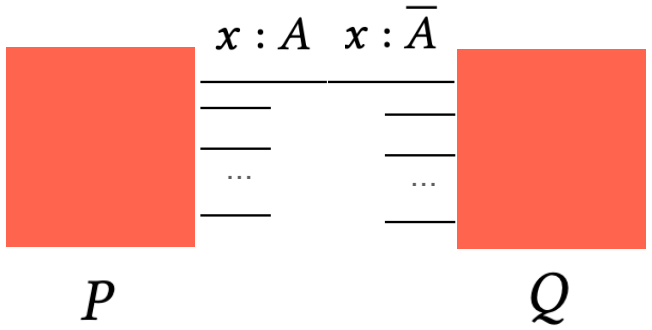
Mix par { *P* || *Q* }



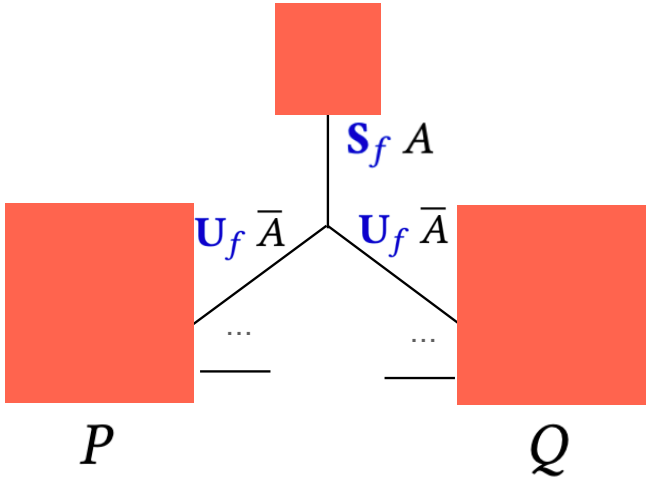
Contraction



Linear Cut

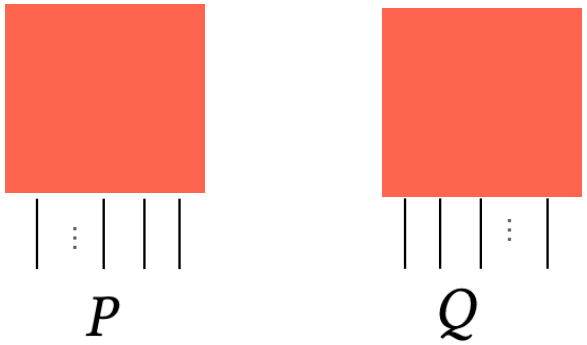


Co-Contraction

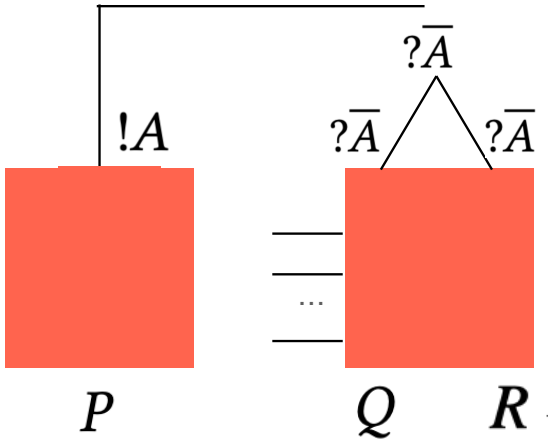


Logical Composition Forms

Mix

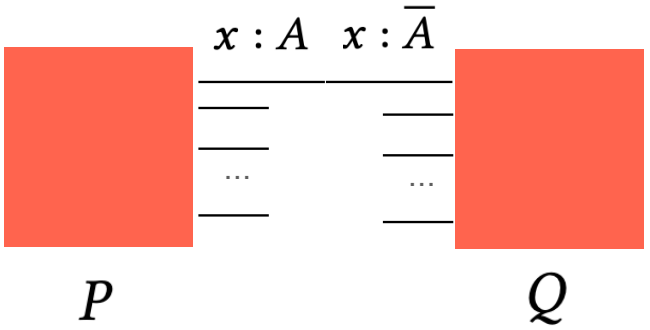


Contraction

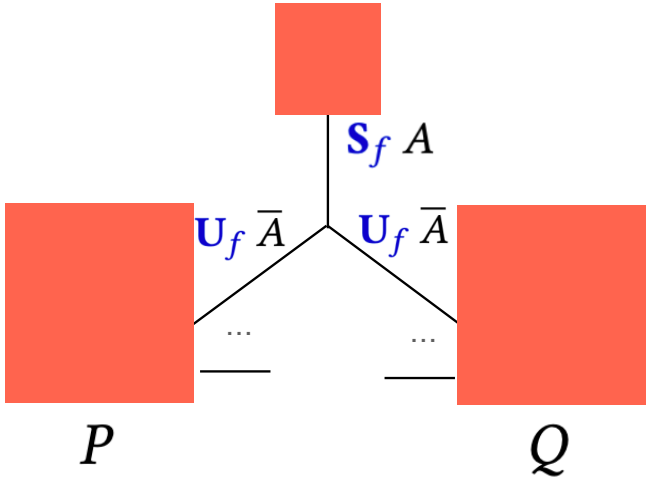


cut { *P* |*x*:A| *Q* }

Linear Cut

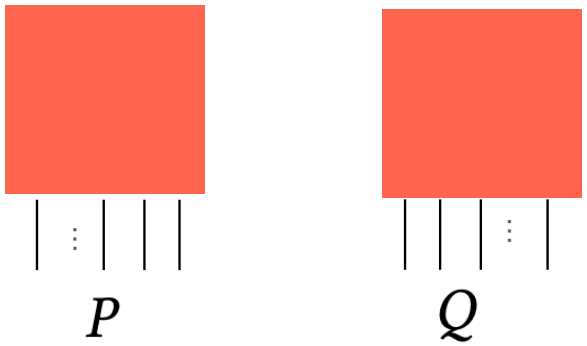


Co-Contraction

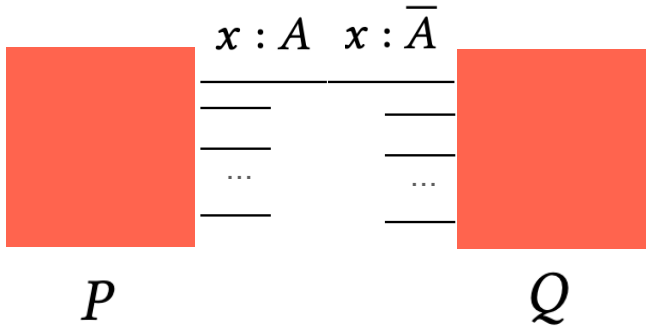


Logical Composition Forms

Mix

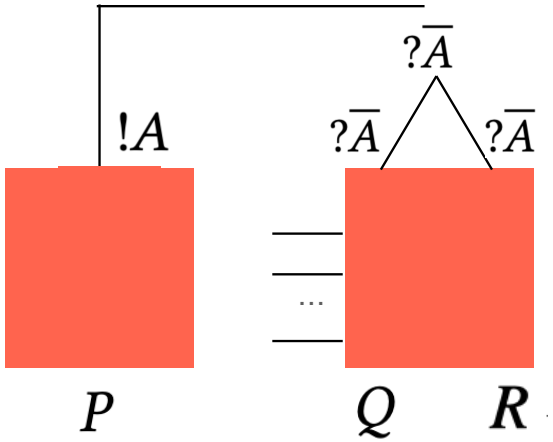


Linear Cut

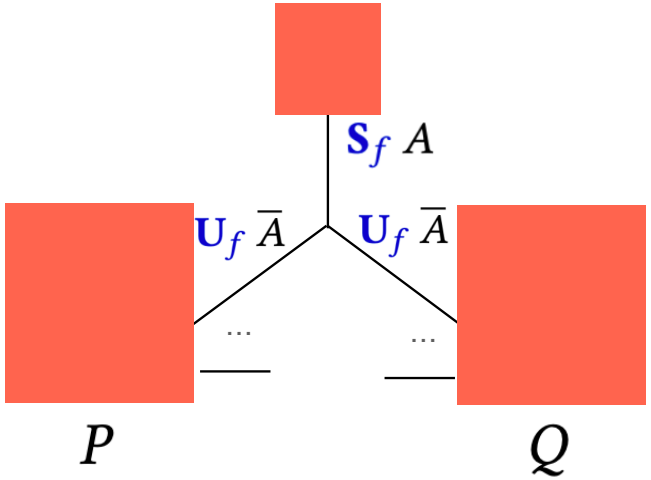


Contraction

?x; { Q || R }

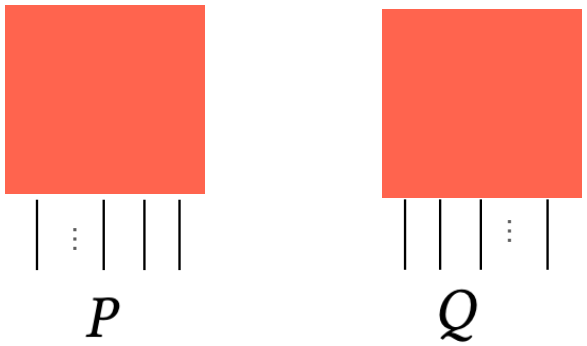


Co-Contraction

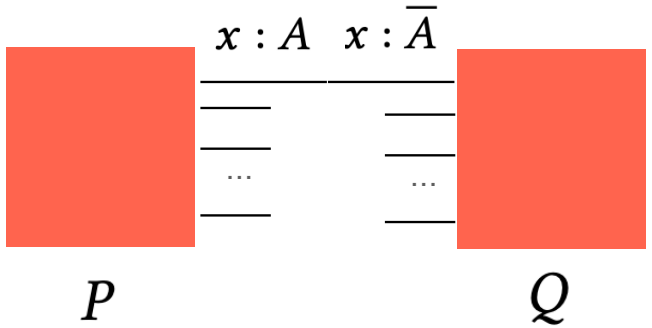


Logical Composition Forms

Mix

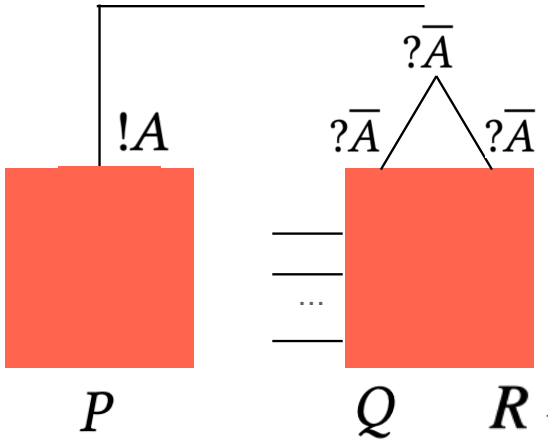


Linear Cut

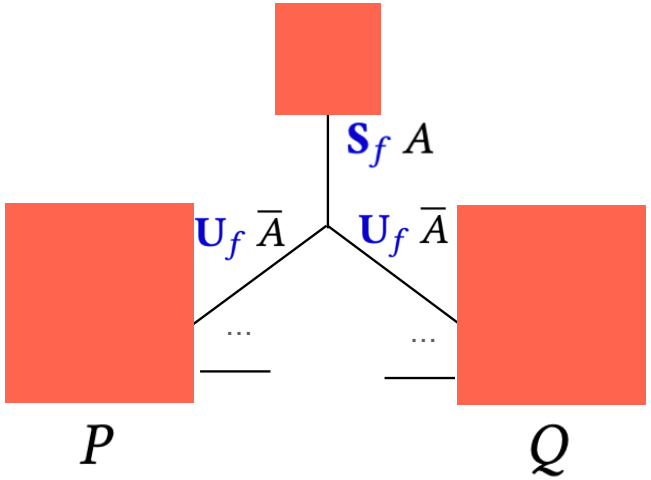


Contraction

?x; { Q |x:A| R }

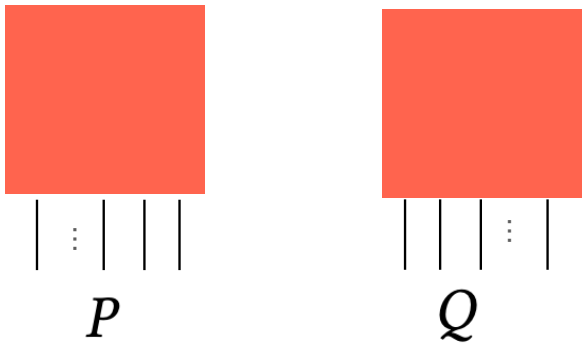


Co-Contraction

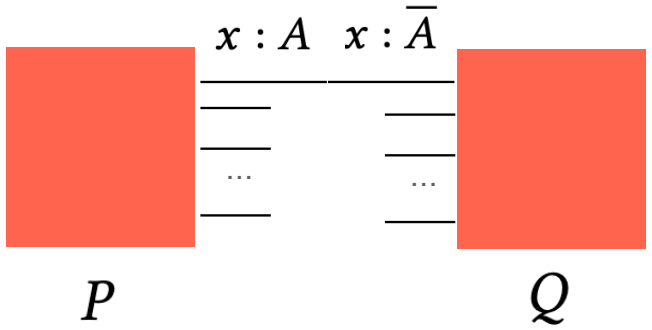


Logical Composition Forms

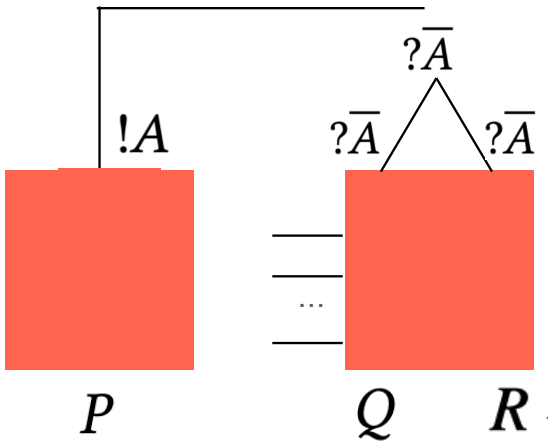
Mix



Linear Cut

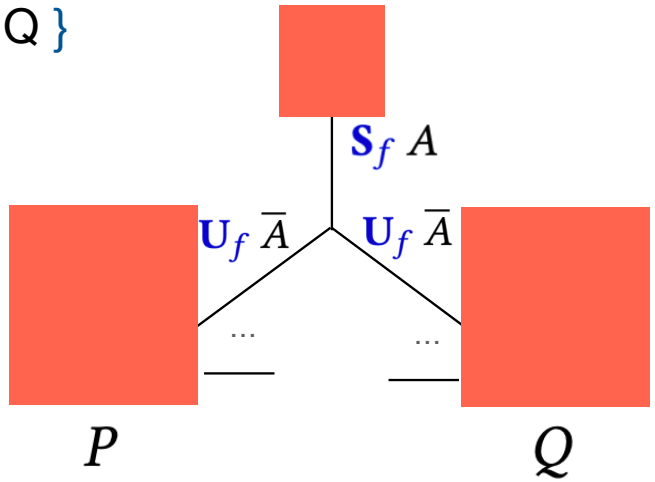


Contraction



Co-Contraction

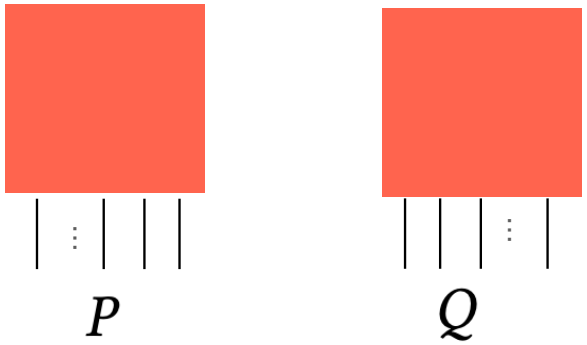
share x { P || Q }



Logical Composition Forms

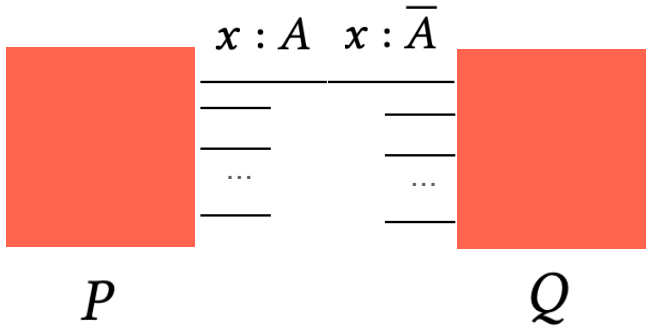
Mix

par { *P* || *Q* }



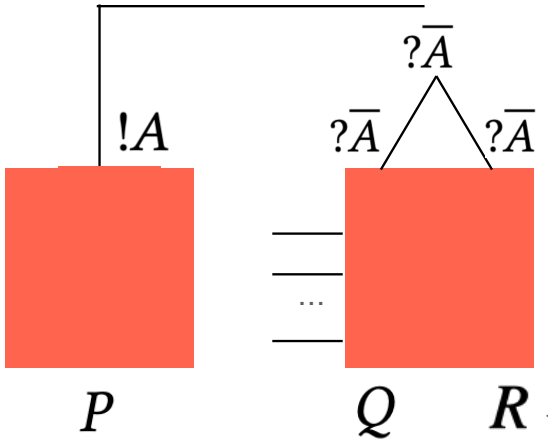
cut { *P* |*x*:A| *Q* }

Linear Cut



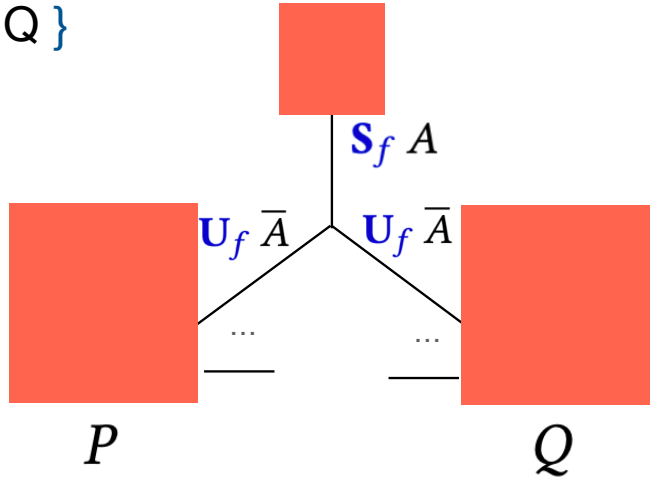
Contraction

?*x*; { Q |*x*:A| R }



share *x* { *P* || Q }

Co-Contraction



A Session Programming Language from Linear Logic

- Computational Interpretation of Linear Logic: congruence $\equiv$, reduction $\rightarrow$.
- Type Preservation: If $P \vdash \Delta; \Gamma$ and $P \rightarrow Q$, then $Q \vdash \Delta; \Gamma$.
- Deadlock-Freedom: Let $P \vdash \varnothing; \varnothing$ be a live process. Then, P reduces.
- Confluence (with sums): If $R \overset{*}{\leftarrow} P \overset{*}{\rightarrow} Q$, then exists S s.t. $R \equiv \overset{*}{\rightarrow} S \overset{*}{\leftarrow} \equiv Q$.
- Normalisation: If $P \vdash \Delta; \Gamma$, then exists a normal form Q s.t. $P \approx Q$.
- Strong Normalisation: If $P \vdash \varnothing; \varnothing$ then P is strongly normalising.

CLASS

- A session-typed language with shared state.
- The linear logical (lightweight) typing ensures:
    1. fidelity (resources are used according to safe protocols)
    2. deadlock absence (in the present of state sharing and locking)
    3. termination (all programs terminate)
    4. no null deferences
    5. no memory leaks
- Algorithmic type checking, (some) type and process reconstruction, basic data types

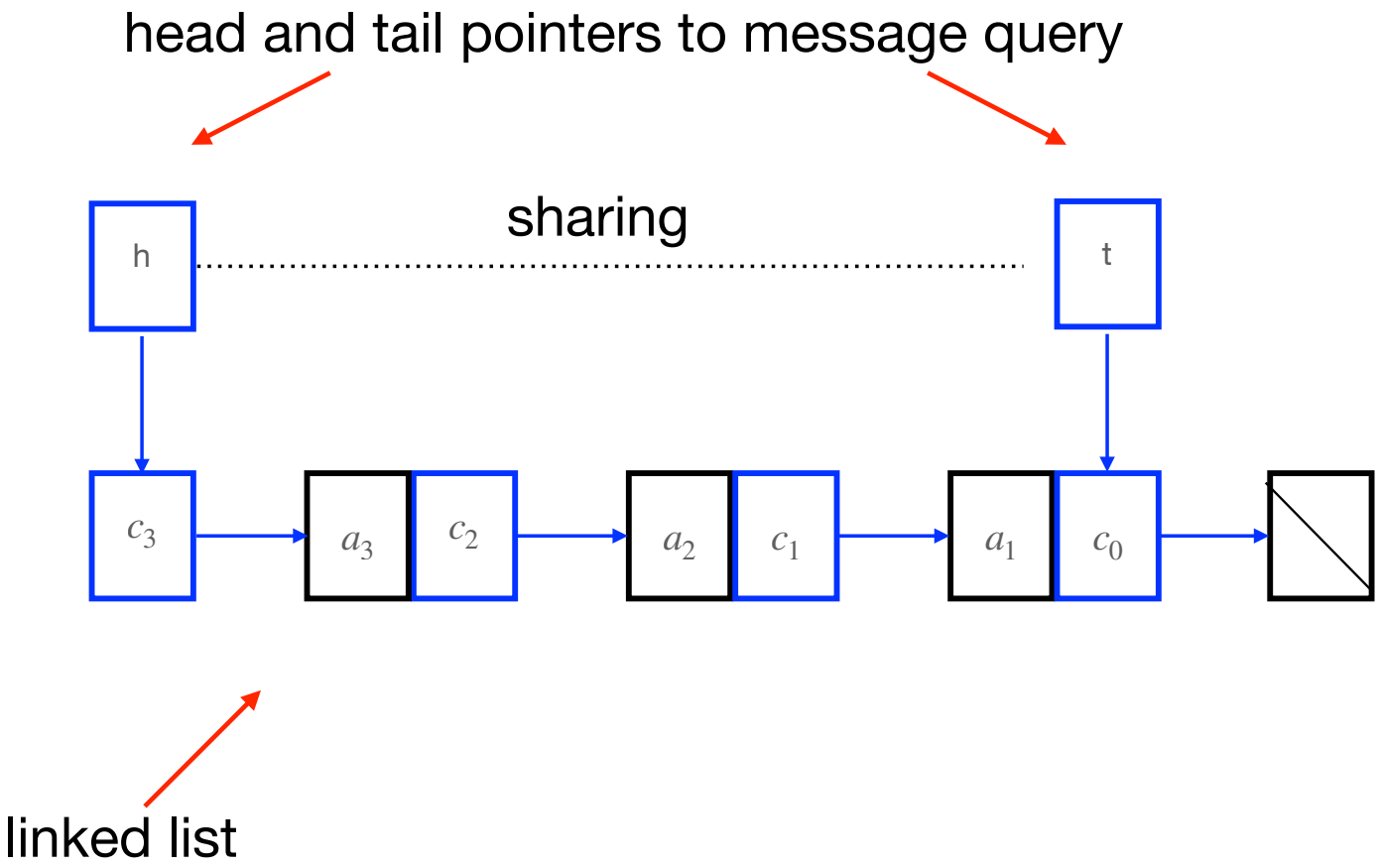
Examples

- Basic session based programming
- Higher-order polymorphic functional programmimg
- Shared state “hello world”
- Sharing linear behaviour
- Thread safe data structures (buffered channel with shared linked list)
- Dining Philosophers
- Barrier Abstraction
- Hoare monitors with conditions

CLASS Source code in the distribution

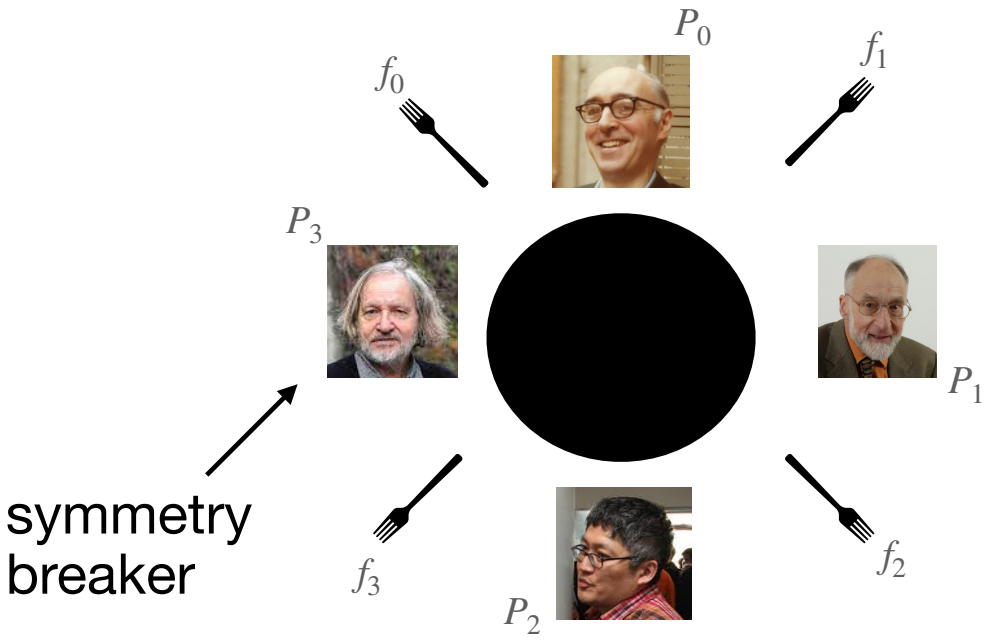
```
include "examples/pure/arithmetic-server.clls";;

include "examples/pure/recursion-for-free.clls";;

include "examples/state/toy.clls";;

include "examples/state/toggle.clls";;

include "examples/state/toggle-shared.nt.clls";;

include "examples/state/dining-philosophers.clls";;

include "examples/state/buffered-channel/tests.clls";;

include "examples/state/barrier.clls";;

include "examples/state/hoare-monitor2.clls";;
```

Buffered Channel




```
type rec List(A) {
    state Node(A)
}
and Node(A) {
    choice of {
        |#Null: close
        |#Next: send A; send List(A);
                close
    }
};;
```

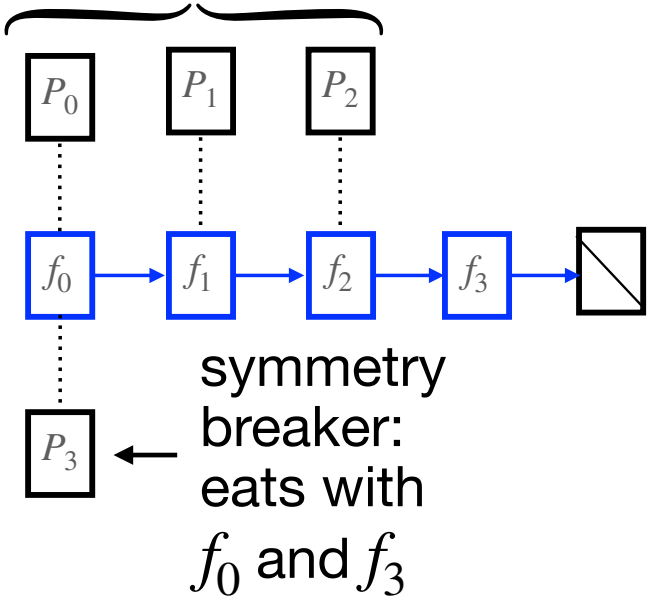
Dining Philosophers



Dijkstra's Solution: $f_0 < f_1 < f_2 < f_3$

encode resource acquisition order using linked data structures

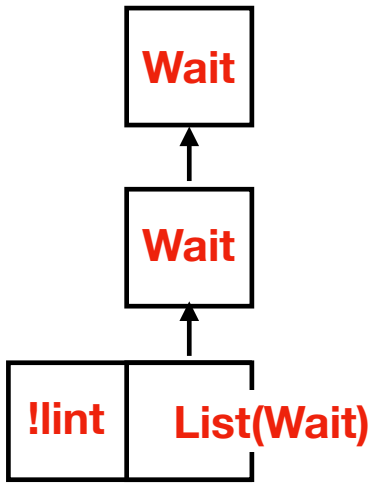
eat with consecutive forks



Barrier

Representation Types

```
type Wait {
    affine wait
};;

type Repr{
   send !Iint;
   affine List(Wait)
};;

type SState {
    state Repr
};;
```



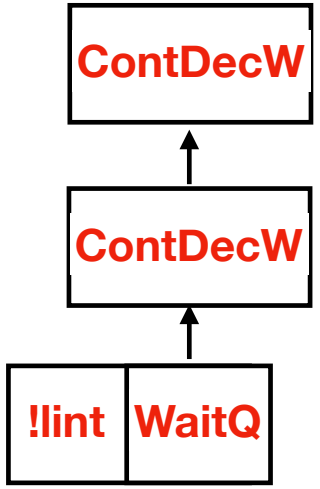
Hoare-style Monitor

Interface Types

```
type corec Incl {
   offer of {
   | #Inc: Incl
   | #End: wait
   }
} and corec Decl {
   coaffine
   offer of {
   | #Dec: coaffine recv ContDec; wait
   | #End: wait
   }
} and ContDec {
   coaffine send Decl; close
} and CounterI {
  recv Decl; Incl
};
```

Representation Types

```
type rec Rep {
   send !Iint; WaitQ
} and WaitQ {
   affine
   choice of {
   | #Next: NodeQ
   | #Null: close
   }
} and ContDecW {
   affine recv ~affine Rep; send affine Rep;
      recv ~Decl; wait
} and NodeQ {
      state send ContDecW; WaitQ
};;
```



CLASS Source code in the distribution

```
include "examples/pure/arithmetic-server.clls";;

include "examples/pure/recursion-for-free.clls";;

include "examples/state/toy.clls";;

include "examples/state/toggle.clls";;

include "examples/state/toggle-shared.nt.clls";;

include "examples/state/dining-philosophers.clls";;

include "examples/state/buffered-channel/tests.clls";;

include "examples/state/barrier.clls";;

include "examples/state/hoare-monitor2.clls";;
```

Some Remarks

- the session calculus as a fundamental language for concurrent computation with linear resources and shared state.
- programs are proofs (in linear logic) that themselves satisfy thread safety, memory safety, deadlock freedom, and termination (via CH and logical relations)
- session calculus considered adequate for mainstream concurrent programming
- we are developing CLASS, a PoC high-level language based on the model, implementation available in open source and bundled with lots of examples
- many challenges ahead

THANKS!