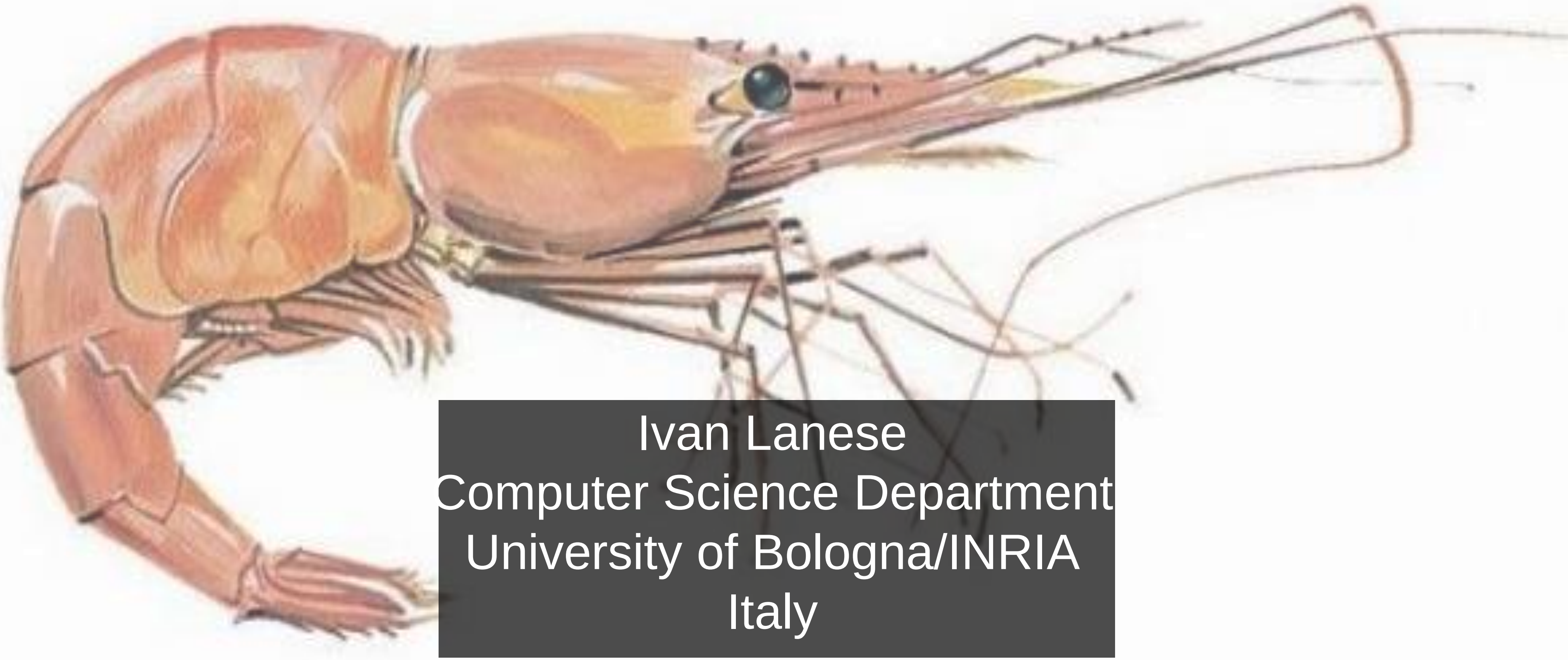
Causal-Consistent Reversibility in a Tuple-Based Language



Ivan Lanese
Computer Science Department
University of Bologna/INRIA
Italy

Joint work with Elena Giachino,
Claudio Antares Mezzina and Francesco Tiezzi

Map of the talk

- Reversibility
- Klaim
- Uncontrolled reversibility in Klaim
- Controlling reversibility: **roll** operator
- Conclusions

What is reversibility?

The possibility of executing a computation both in the standard, forward direction, and in the backward direction, going back to a past state

- Reversibility everywhere
  - chemistry/biology
  - quantum computing
  - state space exploration
  - debugging
  - …

Our aim

- We want to exploit reversibility for programming reliable concurrent and distributed systems
  - To make a system reliable we want to escape “bad” states
  - If a bad state is reached, reversibility allows one to go back to some past “good” state
- We think that reversibility is the key to
  - Understand existing patterns for programming reliable systems, e.g. checkpointing, rollback-recovery, transactions, …
  - Combine and improve them
  - Develop new patterns

Reverse execution of a sequential program

- Recursively undo the last action
  - Computations are undone in reverse order
  - To reverse A;B first reverse B, then reverse A

- We want the Loop Lemma to hold
  - From state S, doing A and then undoing A should lead back to S
  - From state S, undoing A (if A is in the past) and then redoing A should lead back to S

Avoiding loss of information

- Undoing computational actions may not be easy
  - Computational actions may cause loss of information
  - X = 5 causes the loss of the past value of X
- Restrict to languages that never lose information
  - X = X + 1 does not lose information
- Take languages that would lose information, and save this information
  - X = 5 becomes reversible by recording the old value of X

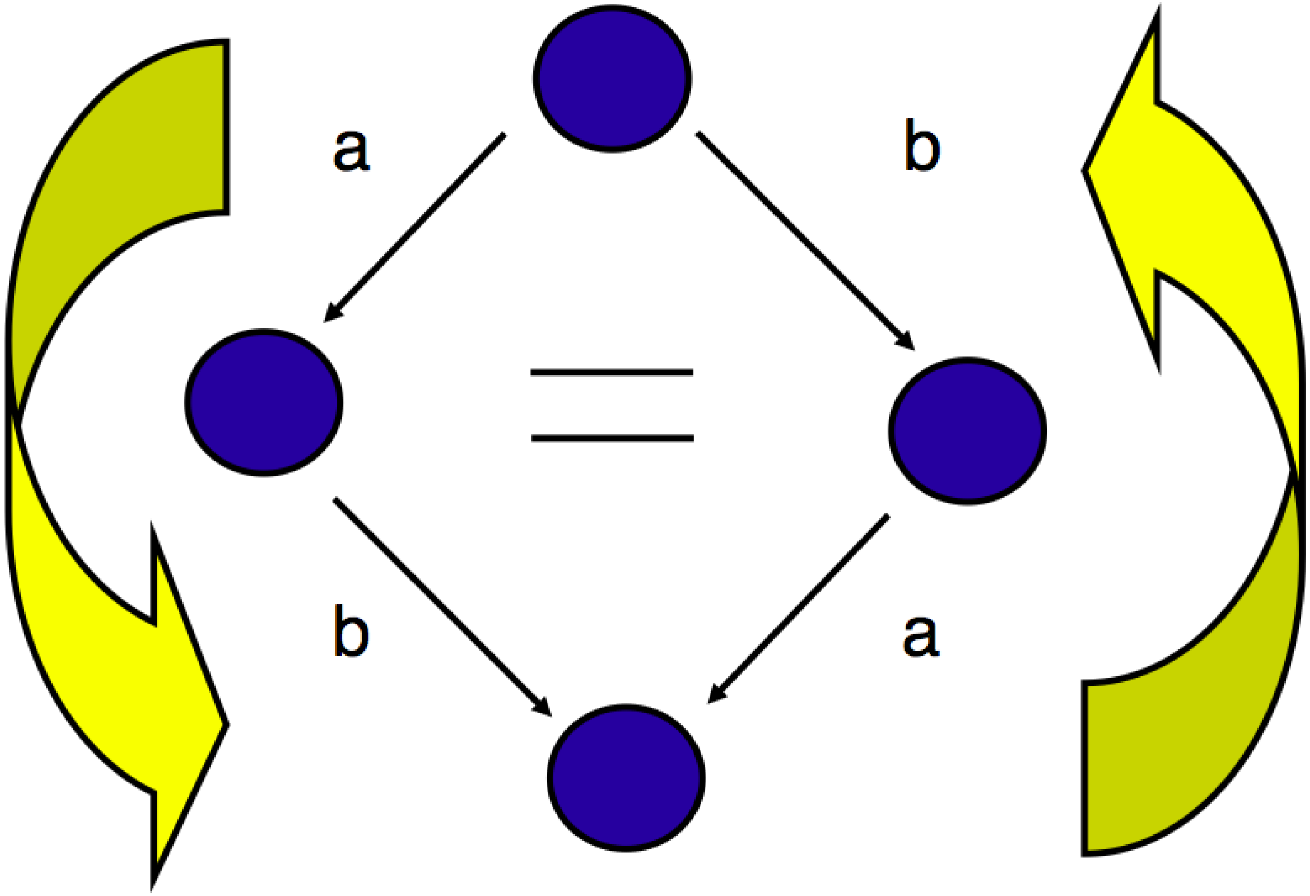
Reversibility and concurrency

- The sequential definition, recursively undo the last action, is no more applicable

- Which is the last action in a concurrent setting?
  - Executions of many actions may overlap
  - For sure, if an action A caused an action B, A could not be the last one

- Causal-consistent reversibility: recursively undo any action whose consequences (if any) have already been undone

Causal-consistent reversibility



Reversibility and concurrency

- Two sequential actions should be undone in reverse order

- Two concurrent actions can be undone in any order
  - Choosing an interleaving for them is an arbitrary choice
  - It should have no impact on the possible reverse behaviors

Map of the talk

- Reversibility
- Klaim
- Uncontrolled reversibility in Klaim
- Controlling reversibility: **roll** operator
- Conclusions

Klaim

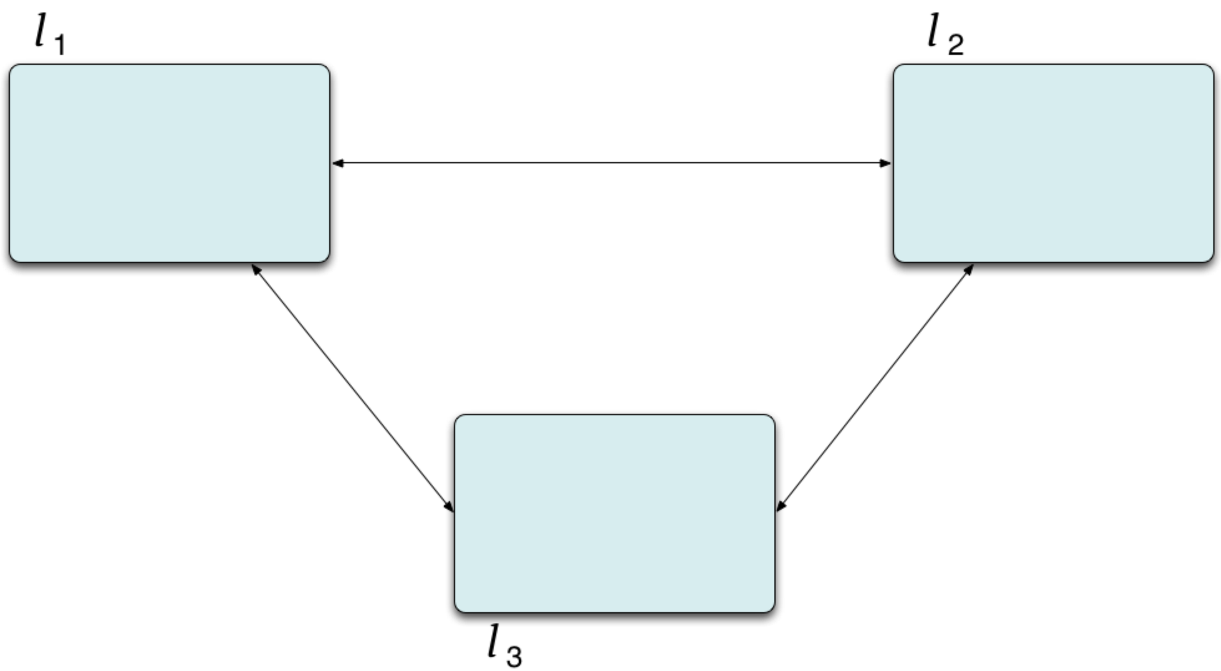
- Coordination language based on distributed tuple spaces
  - Linda operations for creating and accessing tuples
  - Tuples accessed via pattern-matching
- Klaim nets composed by distributed nodes containing processes and data (tuples)
- We consider a subset of Klaim called μKlaim

μKlaim syntax

$$
\begin{array}{rrcl}
\text{(Nets)} & N & ::= & \mathbf{0} \quad | \quad l :: C \quad | \quad N_1 \parallel N_2 \quad | \quad (\nu l)N \\
\text{(Components)} & C & ::= & \langle et \rangle \quad | \quad P \quad | \quad C_1 \mid C_2 \\
\text{(Processes)} & P & ::= & \mathbf{nil} \quad | \quad a.P \quad | \quad P_1 \mid P_2 \quad | \quad A \\
\text{(Actions)} & a & ::= & \mathbf{out}(t)@\ell \quad | \quad \mathbf{eval}(P)@\ell \\
 & & | & \mathbf{in}(T)@\ell \quad | \quad \mathbf{read}(T)@\ell \quad | \quad \mathbf{newloc}(l)
\end{array}
$$

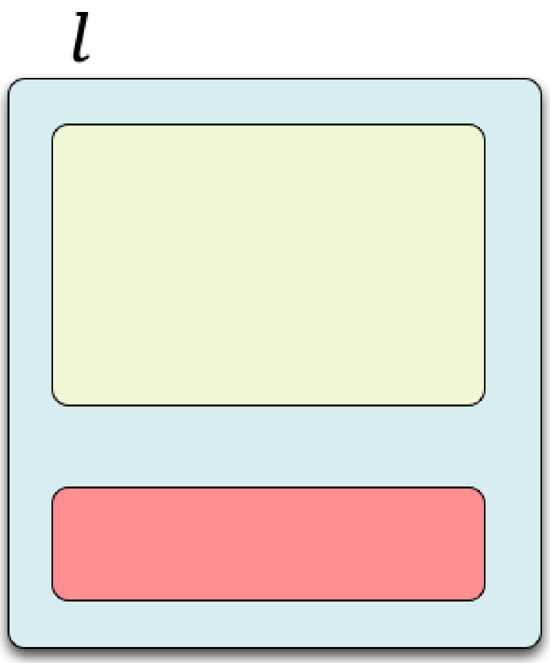
μKlaim syntax

(Nets)	N	$::=$	$\mathbf{0}$	$\mid$	$l :: C$	$\mid$	$N_1 \parallel N_2$	$\mid$	$(\nu l)N$
(Components)	C	$::=$	$\langle et \rangle$	$\mid$	P	$\mid$	$C_1 \mid C_2$		
(Processes)	P	$::=$	$\mathbf{nil}$	$\mid$	$a.P$	$\mid$	$P_1 \mid P_2$	$\mid$	A
(Actions)	a	$::=$	$\mathbf{out}(t)@\ell$	$\mid$	$\mathbf{eval}(P)@\ell$				
		$\mid$	$\mathbf{in}(T)@\ell$	$\mid$	$\mathbf{read}(T)@\ell$	$\mid$	$\mathbf{newloc}(l)$		



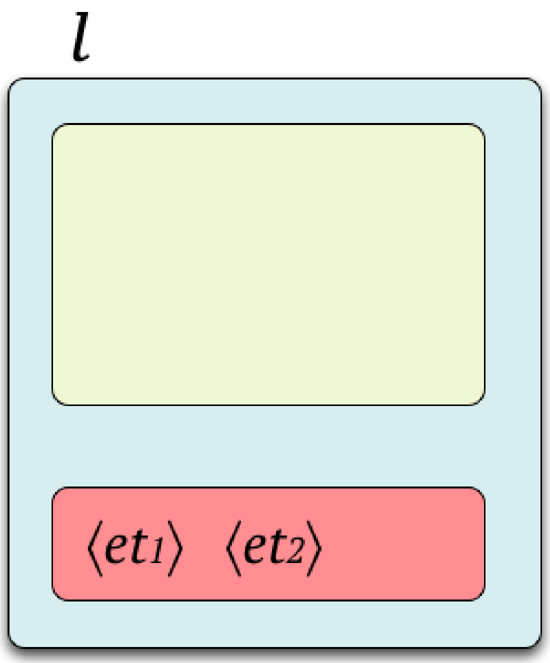
μKlaim syntax

(Nets)	N	$::=$	$\mathbf{0}$	$\mid$	$l :: C$	$\mid$	$N_1 \parallel N_2$	$\mid$ $(\nu l)N$
(Components)	C	$::=$	$\langle et \rangle$	$\mid$	P	$\mid$	$C_1 \mid C_2$	
(Processes)	P	$::=$	$\mathbf{nil}$	$\mid$	$a.P$	$\mid$	$P_1 \mid P_2$	$\mid$ A
(Actions)	a	$::=$	$\mathbf{out}(t)@\ell$	$\mid$	$\mathbf{eval}(P)@\ell$			
		$\mid$	$\mathbf{in}(T)@\ell$	$\mid$	$\mathbf{read}(T)@\ell$	$\mid$	$\mathbf{newloc}(l)$	



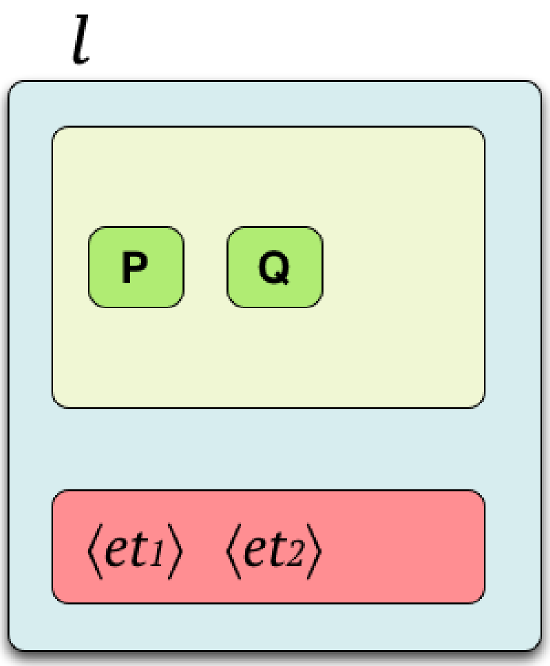
μKlaim syntax

$$
\begin{array}{llllllllll}
\text{(Nets)} & N & ::= & \mathbf{0} & \mid & l :: C & \mid & N_1 \parallel N_2 & \mid & (\nu l)N \\
\text{(Components)} & C & ::= & \langle et \rangle & \mid & P & \mid & C_1 \mid C_2 & & \\
\text{(Processes)} & P & ::= & \mathbf{nil} & \mid & a.P & \mid & P_1 \mid P_2 & \mid & A \\
\text{(Actions)} & a & ::= & \mathbf{out}(t)@\ell & \mid & \mathbf{eval}(P)@\ell & & & & \\
 & & \mid & \mathbf{in}(T)@\ell & \mid & \mathbf{read}(T)@\ell & \mid & \mathbf{newloc}(l) & &
\end{array}
$$



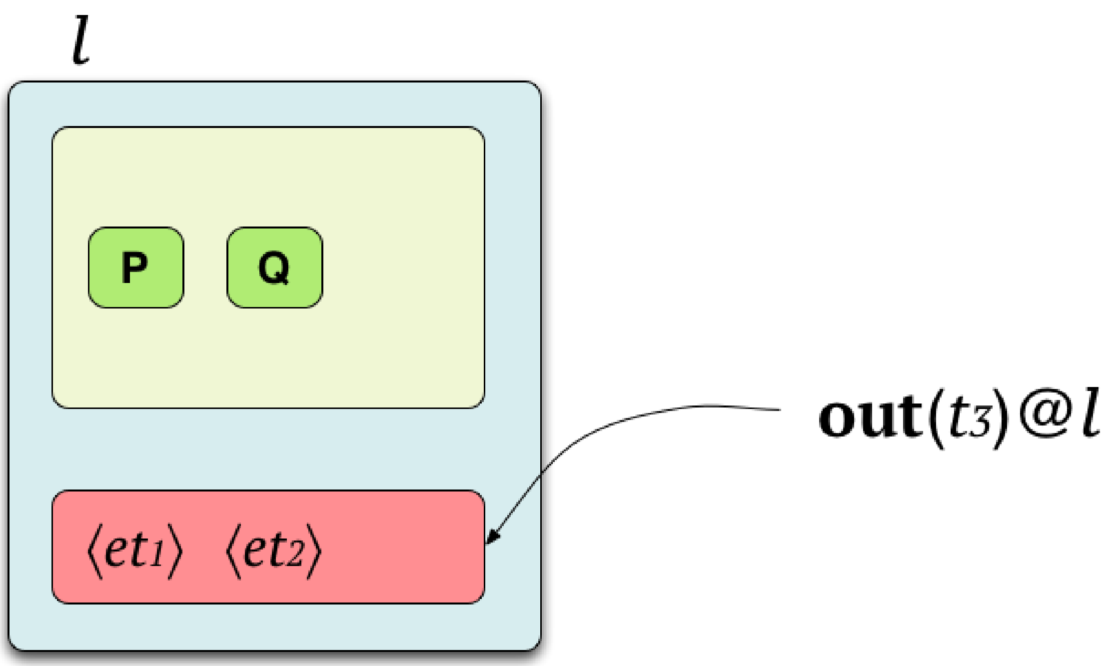
μKlaim syntax

$$
\begin{array}{rrcl}
\text{(Nets)} & N & ::= & \mathbf{0} \quad | \quad l :: C \quad | \quad N_1 \parallel N_2 \quad | \quad (\nu l)N \\
\text{(Components)} & C & ::= & \langle et \rangle \quad | \quad P \quad | \quad C_1 \mid C_2 \\
\text{(Processes)} & P & ::= & \mathbf{nil} \quad | \quad a.P \quad | \quad P_1 \mid P_2 \quad | \quad A \\
\text{(Actions)} & a & ::= & \mathbf{out}(t)@\ell \quad | \quad \mathbf{eval}(P)@\ell \\
 & & | & \mathbf{in}(T)@\ell \quad | \quad \mathbf{read}(T)@\ell \quad | \quad \mathbf{newloc}(l)
\end{array}
$$



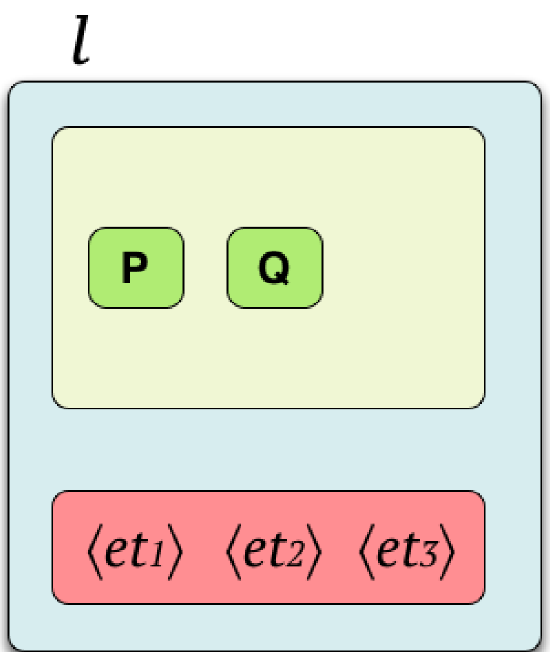
μKlaim syntax

$$
\begin{array}{lllllllll}
\text{(Nets)} & N & ::= & \mathbf{0} & \mid & l :: C & \mid & N_1 \parallel N_2 & \mid & (\nu l)N \\
\text{(Components)} & C & ::= & \langle et \rangle & \mid & P & \mid & C_1 \mid C_2 \\
\text{(Processes)} & P & ::= & \mathbf{nil} & \mid & a.P & \mid & P_1 \mid P_2 & \mid & A \\
\text{(Actions)} & a & ::= & \mathbf{out}(t)@\ell & \mid & \mathbf{eval}(P)@\ell \\
 & & \mid & \mathbf{in}(T)@\ell & \mid & \mathbf{read}(T)@\ell & \mid & \mathbf{newloc}(l)
\end{array}
$$



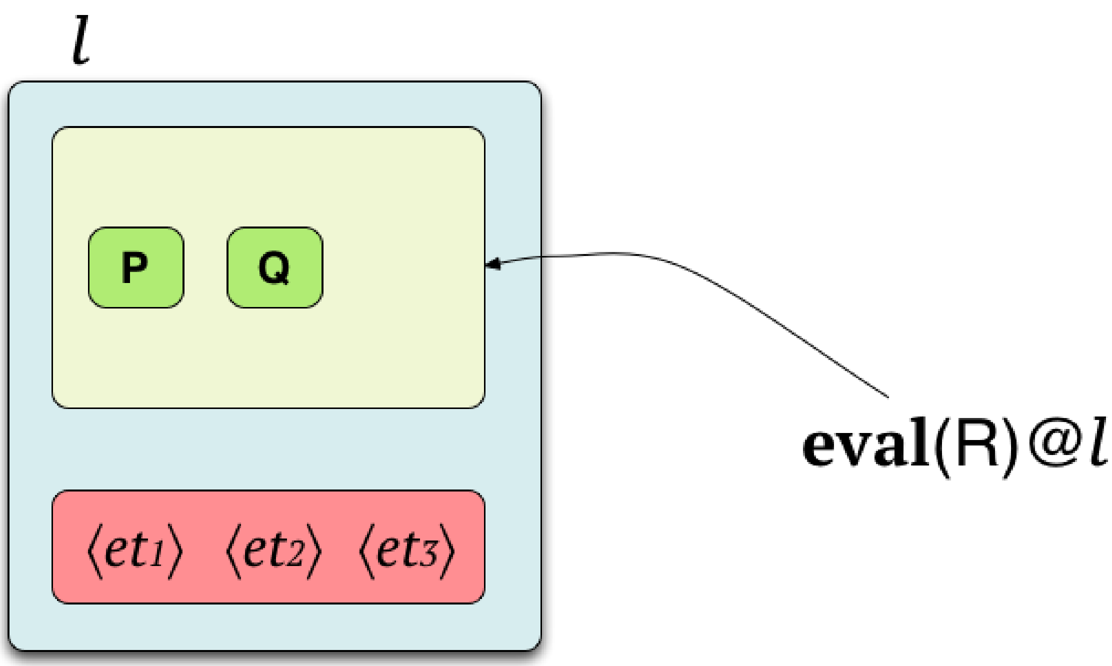
μKlaim syntax

$$
\begin{array}{llllllllll}
\text{(Nets)} & N & ::= & \mathbf{0} & | & l :: C & | & N_1 \parallel N_2 & | & (\nu l)N \\
\text{(Components)} & C & ::= & \langle et \rangle & | & P & | & C_1 \mid C_2 & & \\
\text{(Processes)} & P & ::= & \mathbf{nil} & | & a.P & | & P_1 \mid P_2 & | & A \\
\text{(Actions)} & a & ::= & \mathbf{out}(t)@\ell & | & \mathbf{eval}(P)@\ell & & & & \\
 & & | & \mathbf{in}(T)@\ell & | & \mathbf{read}(T)@\ell & | & \mathbf{newloc}(l) & &
\end{array}
$$



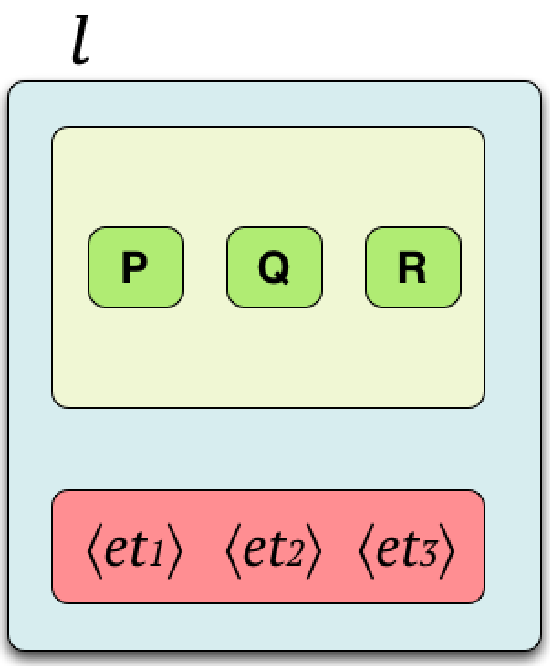
μKlaim syntax

$$
\begin{array}{rlllllllll}
\text{(Nets)} & N & ::= & \mathbf{0} & \mid & l :: C & \mid & N_1 \parallel N_2 & \mid & (\nu l)N \\
\text{(Components)} & C & ::= & \langle et \rangle & \mid & P & \mid & C_1 \mid C_2 & & \\
\text{(Processes)} & P & ::= & \mathbf{nil} & \mid & a.P & \mid & P_1 \mid P_2 & \mid & A \\
\text{(Actions)} & a & ::= & \mathbf{out}(t)@\ell & \mid & \mathbf{eval}(P)@\ell & & & & \\
 & & \mid & \mathbf{in}(T)@\ell & \mid & \mathbf{read}(T)@\ell & \mid & \mathbf{newloc}(l) & &
\end{array}
$$



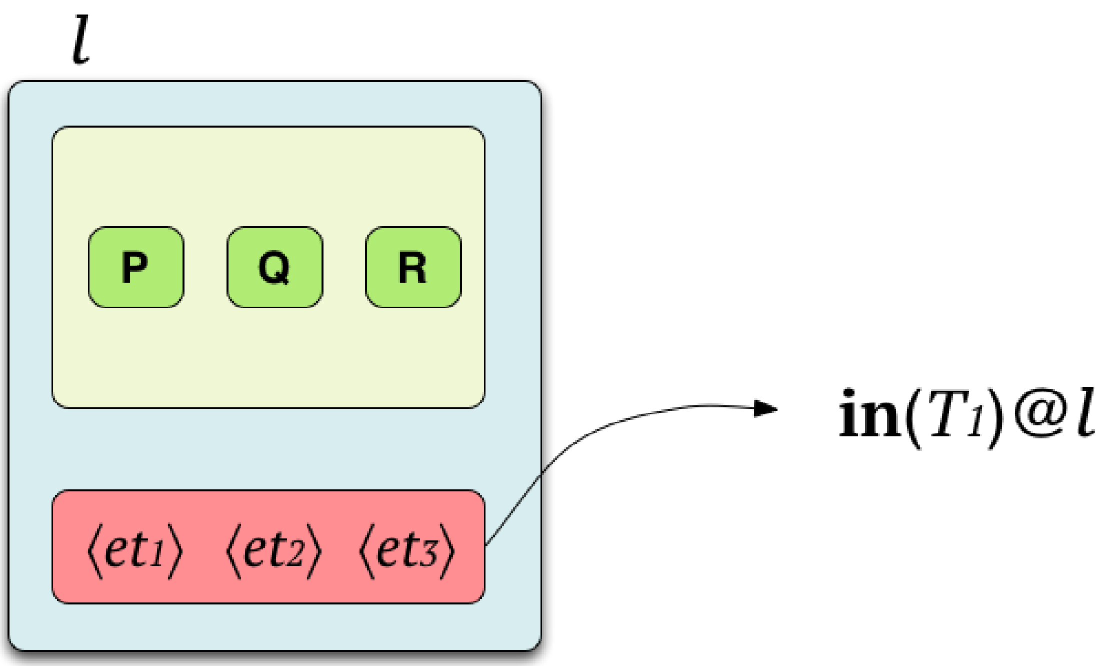
μKlaim syntax

$$
\begin{array}{rrcl}
\text{(Nets)} & N & ::= & \mathbf{0} \quad | \quad l :: C \quad | \quad N_1 \parallel N_2 \quad | \quad (\nu l)N \\
\text{(Components)} & C & ::= & \langle et \rangle \quad | \quad P \quad | \quad C_1 \mid C_2 \\
\text{(Processes)} & P & ::= & \mathbf{nil} \quad | \quad a.P \quad | \quad P_1 \mid P_2 \quad | \quad A \\
\text{(Actions)} & a & ::= & \mathbf{out}(t)@\ell \quad | \quad \mathbf{eval}(P)@\ell \\
 & & | & \mathbf{in}(T)@\ell \quad | \quad \mathbf{read}(T)@\ell \quad | \quad \mathbf{newloc}(l)
\end{array}
$$



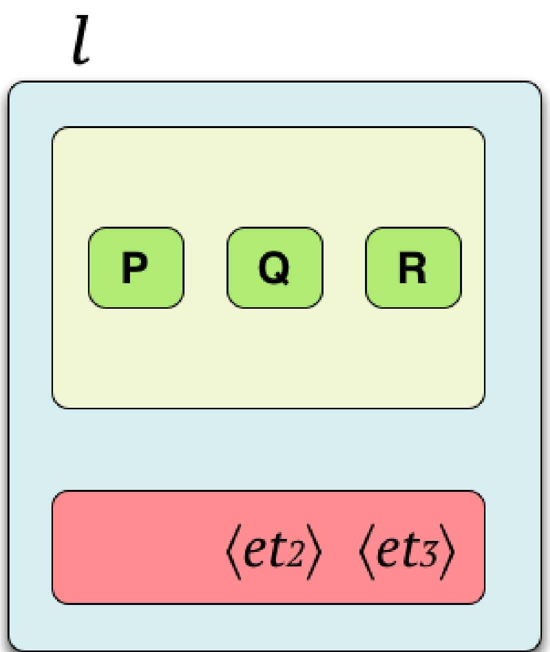
μKlaim syntax

$$
\begin{array}{rlllllllll}
\text{(Nets)} & N & ::= & \mathbf{0} & \mid & l :: C & \mid & N_1 \parallel N_2 & \mid & (\nu l)N \\
\text{(Components)} & C & ::= & \langle et \rangle & \mid & P & \mid & C_1 \mid C_2 & & \\
\text{(Processes)} & P & ::= & \mathbf{nil} & \mid & a.P & \mid & P_1 \mid P_2 & \mid & A \\
\text{(Actions)} & a & ::= & \mathbf{out}(t)@\ell & \mid & \mathbf{eval}(P)@\ell & & & & \\
 & & \mid & {\color{red}\mathbf{in}(T)@\ell} & \mid & \mathbf{read}(T)@\ell & \mid & \mathbf{newloc}(l) & &
\end{array}
$$



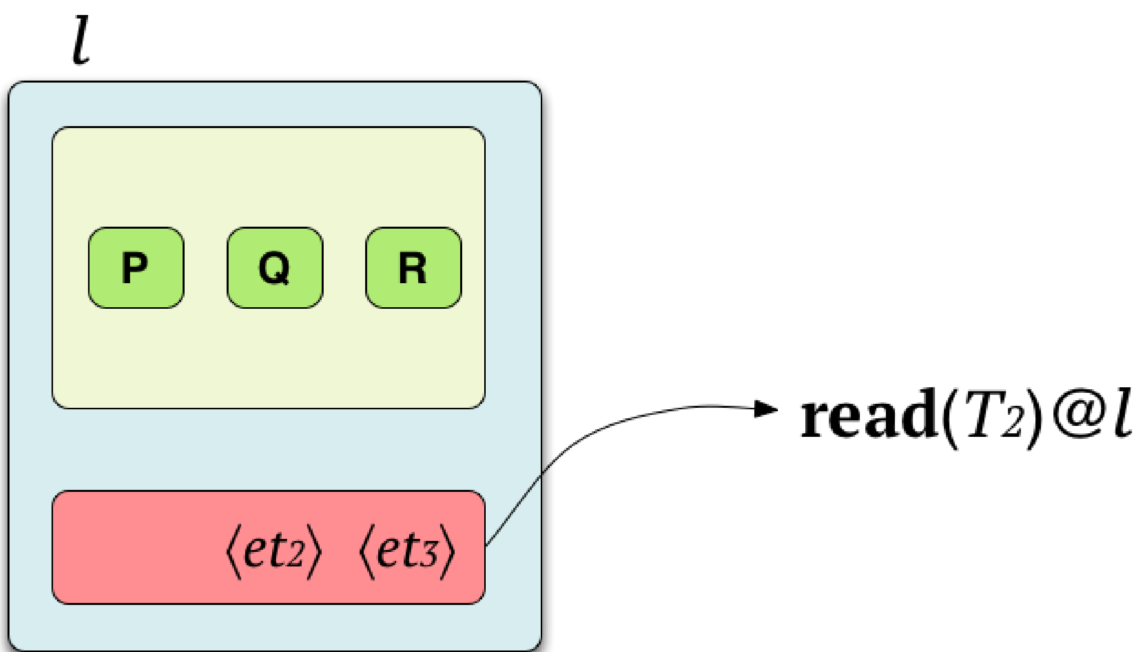
μKlaim syntax

$$
\begin{array}{rrcl}
\text{(Nets)} & N & ::= & \mathbf{0} \quad | \quad l :: C \quad | \quad N_1 \parallel N_2 \quad | \quad (\nu l)N \\
\text{(Components)} & C & ::= & \langle et \rangle \quad | \quad P \quad | \quad C_1 \mid C_2 \\
\text{(Processes)} & P & ::= & \mathbf{nil} \quad | \quad a.P \quad | \quad P_1 \mid P_2 \quad | \quad A \\
\text{(Actions)} & a & ::= & \mathbf{out}(t)@\ell \quad | \quad \mathbf{eval}(P)@\ell \\
 & & | & \mathbf{in}(T)@\ell \quad | \quad \mathbf{read}(T)@\ell \quad | \quad \mathbf{newloc}(l)
\end{array}
$$



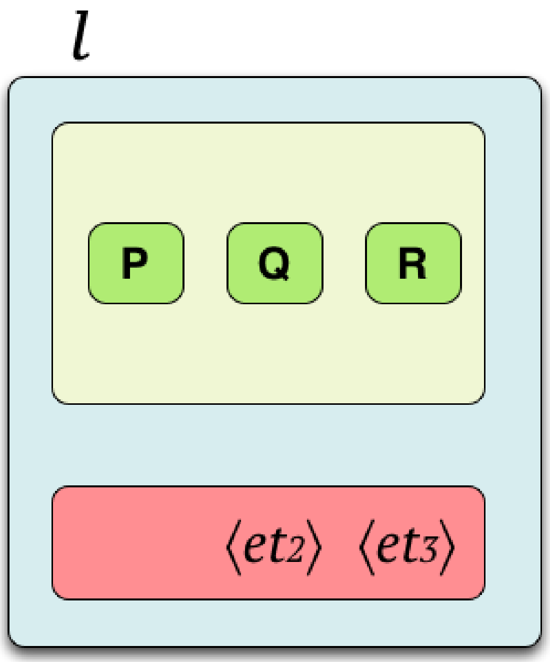
µKlaim syntax

$$
\begin{array}{llllllll}
\text{(Nets)} & N & ::= & \mathbf{0} & \mid & l :: C & \mid & N_1 \parallel N_2 & \mid & (\nu l)N \\
\text{(Components)} & C & ::= & \langle et \rangle & \mid & P & \mid & C_1 \mid C_2 \\
\text{(Processes)} & P & ::= & \mathbf{nil} & \mid & a.P & \mid & P_1 \mid P_2 & \mid & A \\
\text{(Actions)} & a & ::= & \mathbf{out}(t)@\ell & \mid & \mathbf{eval}(P)@\ell \\
& & \mid & \mathbf{in}(T)@\ell & \mid & \mathbf{read}(T)@\ell & \mid & \mathbf{newloc}(l)
\end{array}
$$



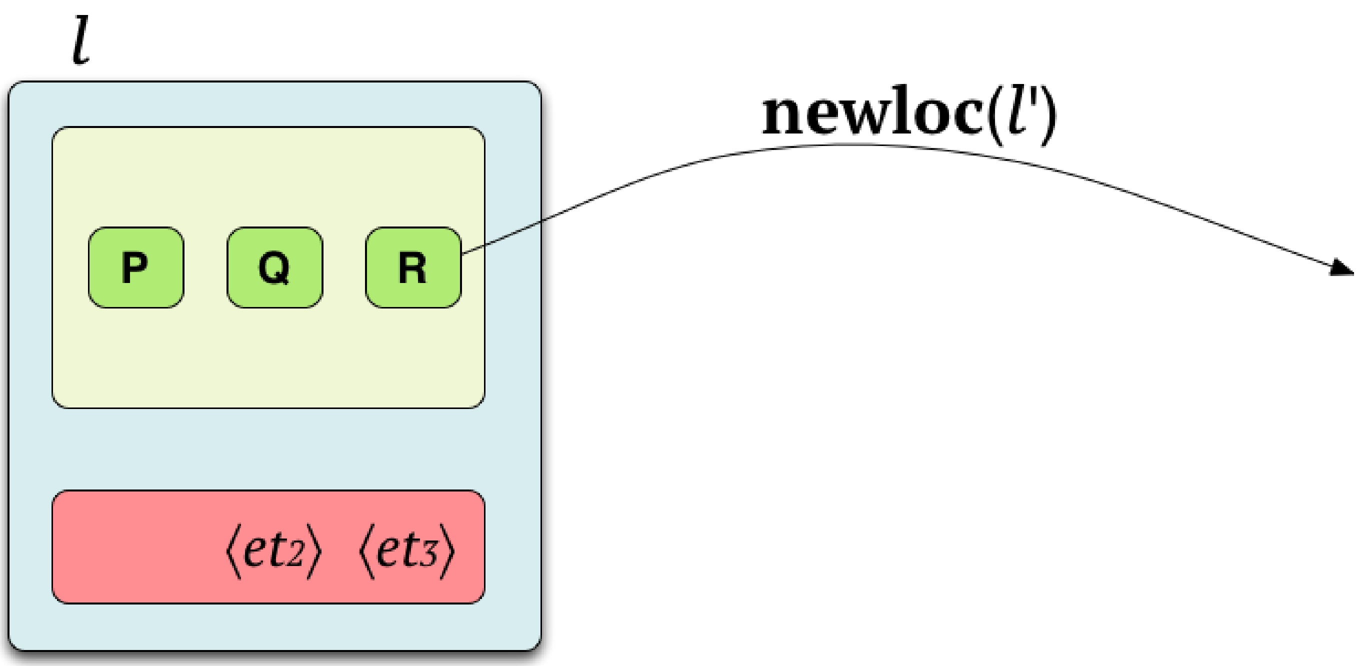
μKlaim syntax

$$
\begin{array}{rrcl}
\text{(Nets)} & N & ::= & \mathbf{0} \quad | \quad l :: C \quad | \quad N_1 \parallel N_2 \quad | \quad (\nu l)N \\
\text{(Components)} & C & ::= & \langle et \rangle \quad | \quad P \quad | \quad C_1 \mid C_2 \\
\text{(Processes)} & P & ::= & \mathbf{nil} \quad | \quad a.P \quad | \quad P_1 \mid P_2 \quad | \quad A \\
\text{(Actions)} & a & ::= & \mathbf{out}(t)@\ell \quad | \quad \mathbf{eval}(P)@\ell \\
 & & | & \mathbf{in}(T)@\ell \quad | \quad \mathbf{read}(T)@\ell \quad | \quad \mathbf{newloc}(l)
\end{array}
$$



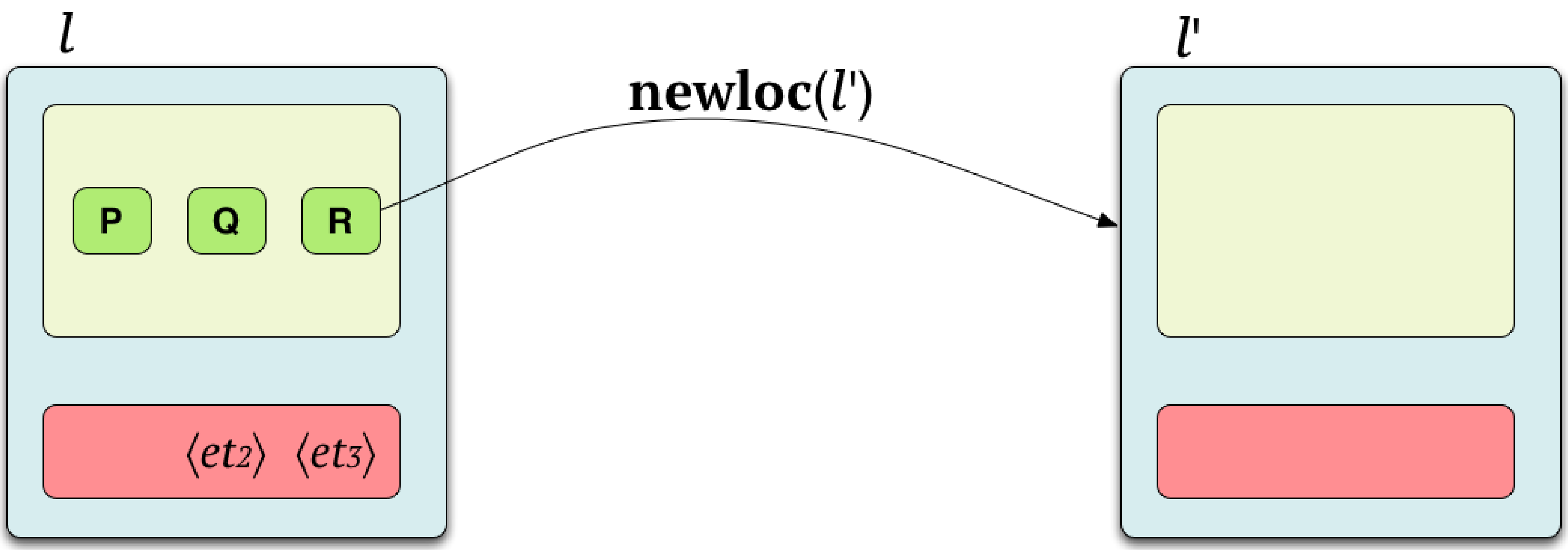
µKlaim syntax

$$
\begin{array}{lrcl}
\text{(Nets)} & N & ::= & \mathbf{0} \quad | \quad l :: C \quad | \quad N_1 \parallel N_2 \quad | \quad (\nu l)N \\
\text{(Components)} & C & ::= & \langle et \rangle \quad | \quad P \quad | \quad C_1 \mid C_2 \\
\text{(Processes)} & P & ::= & \mathbf{nil} \quad | \quad a.P \quad | \quad P_1 \mid P_2 \quad | \quad A \\
\text{(Actions)} & a & ::= & \mathbf{out}(t)@\ell \quad | \quad \mathbf{eval}(P)@\ell \\
 & & | & \mathbf{in}(T)@\ell \quad | \quad \mathbf{read}(T)@\ell \quad | \quad \mathbf{newloc}(l)
\end{array}
$$



μKlaim syntax

(Nets)	N	$::=$	$\mathbf{0}$	$\mid$	$l :: C$	$\mid$	$N_1 \parallel N_2$	$\mid$	$(\nu l)N$
(Components)	C	$::=$	$\langle et \rangle$	$\mid$	P	$\mid$	$C_1 \mid C_2$		
(Processes)	P	$::=$	$\mathbf{nil}$	$\mid$	$a.P$	$\mid$	$P_1 \mid P_2$	$\mid$	A
(Actions)	a	$::=$	$\mathbf{out}(t)@\ell$	$\mid$	$\mathbf{eval}(P)@\ell$				
		$\mid$	$\mathbf{in}(T)@\ell$	$\mid$	$\mathbf{read}(T)@\ell$	$\mid$	$\mathbf{newloc}(l)$		



Example

$$l_1 :: \langle foo \rangle \parallel l_2 :: \mathbf{read}(foo)@l_1.P \parallel l_3 :: \mathbf{read}(foo)@l_1.P'$$

$$l_1 :: \langle foo \rangle \parallel l_2 :: \mathbf{read}(foo)@l_1.P \parallel l_3 :: P'$$

$$l_1 :: \langle foo \rangle \parallel l_2 :: P \parallel l_3 :: \mathbf{read}(foo)@l_1.P'$$

$$l_1 :: \langle foo \rangle \parallel l_2 :: P \parallel l_3 :: P'$$

Map of the talk

- Reversibility
- Klaim
- Uncontrolled reversibility in Klaim
- Controlling reversibility: **roll** operator
- Conclusions

Making μKlaim reversible

- We define RμKlaim, an extension of μKlaim allowing:
  - *forward* actions, corresponding to μKlaim actions
  - *backward* actions, undoing them
- One has to trace history and causality information
  - We label evaluated tuples and processes with unique keys k
  - We use connectors $k_1 \prec (k_2, k_3)$ to store causality information
  - We use memories to store past actions
- Similarly to past works on other languages

Making μKlaim reversible

- We have to deal with some peculiarities of μKlaim causality structure
- Read dependencies
  - Two **read**s on the same tuple should not create dependences
  - If the **out** creating the tuple is undone then **read**s on the same tuple should be undone too
- Localities
  - Localities are resources and establish dependences
  - To undo a **newloc** one has to undo all the operations on the created locality

RμKlaim syntax

$$
\begin{array}{rrcl}
\text{(Nets)} & N & ::= & \mathbf{0} \quad | \quad l :: C \quad | \quad l :: \mathbf{empty} \quad | \quad N_1 \parallel N_2 \quad | \quad (\nu z)N \\
\text{(Components)} & C & ::= & k : \langle et \rangle \quad | \quad k : P \quad | \quad C_1 \mid C_2 \quad | \quad \mu \quad | \quad k_1 \prec (k_2, k_3) \\
\text{(Processes)} & P & ::= & \mathbf{nil} \quad | \quad a.P \quad | \quad P_1 \mid P_2 \quad | \quad A \\
\text{(Actions)} & a & ::= & \mathbf{out}(t)@\ell \quad | \quad \mathbf{eval}(P)@\ell \\
 & & | & \mathbf{in}(T)@\ell \quad | \quad \mathbf{read}(T)@\ell \quad | \quad \mathbf{newloc}(l) \\
\text{(Memories)} & \mu & ::= & [k : \mathbf{out}(t)@l; k''; k'] \quad | \quad [k : \mathbf{in}(T)@l.P; h : \langle et \rangle; k'] \\
 & & | & [k : \mathbf{read}(T)@l.P; h; k'] \quad | \quad [k : \mathbf{newloc}(l); k'] \\
 & & | & [k : \mathbf{eval}(Q)@l; k''; k']
\end{array}
$$

Example

$$l_1 :: k_1 : \langle foo \rangle \;\|\; l_2 :: k_2 : \mathbf{read}(foo)@l_1.P$$
$$\|\; l_3 :: k_3 : \mathbf{read}(foo)@l_1.P'$$

$$(\nu k_3')\; (\, l_1 :: k_1 : \langle foo \rangle \;\|\; l_2 :: k_2 : \mathbf{read}(foo)@l_1.P$$
$$\|\; l_3 :: k_3' : P' \mid [k_3 : \mathbf{read}(foo)@l_1.P'; k_1; k_3'] \,)$$

$$(\nu k_2')\; (\, l_1 :: k_1 : \langle foo \rangle$$
$$\|\; l_2 :: k_2' : P \mid [k_2 : \mathbf{read}(foo)@l_1.P; k_1; k_2']$$
$$\|\; l_3 :: k_3 : \mathbf{read}(foo)@l_1.P' \,)$$

$$(\nu k_2', k_3')\; (\, l_1 :: k_1 : \langle foo \rangle$$
$$\|\; l_2 :: k_2' : P \mid [k_2 : \mathbf{read}(foo)@l_1.P; k_1; k_2']$$
$$\|\; l_3 :: k_3' : P' \mid [k_3 : \mathbf{read}(foo)@l_1.P'; k_1; k_3'] \,)$$

Example

$$l_1 \;::\; k_1 : \langle foo \rangle \parallel \; l_2 \;::\; k_2 : \mathbf{in}(foo)@l_1.\mathbf{out}(foo)@l_1.P$$
$$\parallel \; l_3 \;::\; k_3 : \mathbf{in}(foo)@l_1.\mathbf{out}(foo)@l_1.P'$$

↓ execute **in** in l_2

↓ execute **out** in l_2

↓ execute **in** in l_3

↓ execute **out** in l_3

$$(\nu k_2', k_2'', k_2''', k_3', k_3'', k_3''')(l_1 \;::\; k_3''' : \langle foo \rangle$$
$$\parallel l_2 \;::\; k_2'' : P \mid [k_2 : \mathbf{in}(foo)@l_1.\mathbf{out}(foo)@l_1.P; k_1 : \langle foo \rangle; k_2']$$
$$\mid [k_2' : \mathbf{out}(foo)@l_1; k_2'''; k_2'']$$
$$\parallel l_3 \;::\; k_3'' : P' \mid [k_3 : \mathbf{in}(foo)@l_1.\mathbf{out}(foo)@l_1.P'; k_2''' : \langle foo \rangle; k_3']$$
$$\mid [k_3' : \mathbf{out}(foo)@l_1; k_3'''; k_3''] \,)$$

Example

$$
\begin{aligned}
l_1 :: k_1 : \langle foo \rangle \parallel\ & l_2 :: k_2 : \mathbf{in}(foo)@l_1.\mathbf{out}(foo)@l_1.P \\
\parallel\ & l_3 :: k_3 : \mathbf{in}(foo)@l_1.\mathbf{out}(foo)@l_1.P'
\end{aligned}
$$

↓ execute **in** in l_2

↓ execute **out** in l_2

↓ execute **in** in l_3

↓ execute **out** in l_3

it needs $k_2''' : \langle foo \rangle$ in l_1 to perform a backward step

$$
\begin{aligned}
(\nu k_2', k_2'', k_2''', k_3', k_3'', k_3''')(\, & l_1 :: k_3''' : \langle foo \rangle \\
\parallel\ & l_2 :: k_2'' : P \mid [k_2 : \mathbf{in}(foo)@l_1.\mathbf{out}(foo)@l_1.P; k_1 : \langle foo \rangle; k_2'] \\
& \qquad \mid [k_2' : \mathbf{out}(foo)@l_1; k_2'''; k_2''] \\
\parallel\ & l_3 :: k_3'' : P' \mid [k_3 : \mathbf{in}(foo)@l_1.\mathbf{out}(foo)@l_1.P'; k_2''' : \langle foo \rangle; k_3'] \\
& \qquad \mid [k_3' : \mathbf{out}(foo)@l_1; k_3'''; k_3''] \,)
\end{aligned}
$$

Properties

- The forward semantics of RμKlaim is an annotated version of the semantics of μKlaim
- The Loop Lemma holds
  - i.e., each reduction in RμKlaim has an inverse
- RμKlaim is causally consistent

Concurrency in RμKlaim

- Two transitions are concurrent unless
  - They use the same resource
  - At least one transition does not use it in read-only modality
- Resources defined by function λ on memories

$$
\begin{array}{rcl}
\lambda([k:\mathbf{out}(t)@l;k'';k']) & = & \{k,k',k'',\mathtt{r}(l)\} \\
\lambda([k:\mathbf{in}(T)@l.P;k'':\langle et\rangle;k']) & = & \{k,k',k'',\mathtt{r}(l)\} \\
\lambda([k:\mathbf{read}(T)@l.P;k'';k']) & = & \{k,\mathtt{r}(k''),k',\mathtt{r}(l)\} \\
\lambda([k:\mathbf{eval}(Q)@l;k'';k']) & = & \{k,k',k'',\mathtt{r}(l)\} \\
\lambda([k:\mathbf{newloc}(l);k']) & = & \{k,k',l\}
\end{array}
$$

- **Read** uses the tuple in read-only modality
- All primitives but **newloc** use the target locality in read-only modality

Causal consistency

- *Causal equivalence* identifies traces that differ only for
  - swaps of concurrent transitions
  - simplifications of inverse transitions

- *Casual consistency*: there is a unique way to go from one state to another up to causal equivalence
  - causal equivalent traces can be reversed in the same ways
  - traces which are not causal equivalent lead to distinct nets

Is uncontrolled reversibility enough?

- Uncontrolled reversibility is a suitable setting to understand and prove properties about reversibility
- ... but it is not suitable for programming (reliable) systems
  - Actions are done and undone nondeterministically
  - A program may diverge by doing and undoing the same action forever
  - No way to keep good results

Map of the talk

- Reversibility
- Klaim
- Uncontrolled reversibility in Klaim
- Controlling reversibility: **roll** operator
- Conclusions

Controlling reversibility

- In reliable systems
  - Normal execution is forward
  - Backward execution is used to escape bad states
- We add to μKlaim a **roll** operator
  - To undo a given past action
  - Together with all its consequences (and only them)
- We call CRμKlaim the extension of μKlaim with **roll**

CRμKlaim syntax

$$
\begin{array}{rlcl}
\text{(Nets)} & N & ::= & \mathbf{0} \quad | \quad l :: C \quad | \quad l :: \mathbf{empty} \quad | \quad N_1 \parallel N_2 \quad | \quad (\nu z)N \\
\text{(Components)} & C & ::= & k : \langle et \rangle \quad | \quad k : P \quad | \quad C_1 \mid C_2 \quad | \quad \mu \quad | \quad k_1 \prec (k_2, k_3) \\
\text{(Processes)} & P & ::= & \mathbf{nil} \quad | \quad a.P \quad | \quad P_1 \mid P_2 \quad | \quad A \quad | \quad \mathbf{roll}(\iota) \\
\text{(Actions)} & a & ::= & \mathbf{out}_\gamma(t)@\ell \quad | \quad \mathbf{eval}_\gamma(P)@\ell \quad | \quad \mathbf{in}_\gamma(T)@\ell \quad | \\
 & & & \mathbf{read}_\gamma(T)@\ell \quad | \quad \mathbf{newloc}_\gamma(l) \\
\text{(Memories)} & \mu & ::= & [k : \mathbf{out}_\gamma(t)@l.P; k''; k'] \quad | \quad [k : \mathbf{in}_\gamma(T)@l.P; h : \langle t \rangle; k'] \\
 & & | & [k : \mathbf{read}_\gamma(T)@l.P; h; k'] \quad | \quad [k : \mathbf{newloc}_\gamma(l).P; k'] \\
 & & | & [k : \mathbf{eval}_\gamma(Q)@l.P; k''; k']
\end{array}
$$

Example

- From

$$l :: k : \mathbf{out}_\gamma(foo)@l.\mathbf{roll}(\gamma) \parallel l' :: k' : \mathbf{in}(foo)@l.\mathbf{nil}$$

- We get

$$(\nu k'', k''', k'''')$$
$$(l :: k'' : \mathbf{roll}(k) \mid [k : \mathbf{out}_\gamma(foo)@l.\mathbf{roll}(\gamma); k'''; k'']$$
$$\parallel l' :: k'''' : \mathbf{nil} \mid [k' : \mathbf{in}(foo)@l.\mathbf{nil}; k''' : \langle foo \rangle; k''''])$$

- When we undo the **out** we need to restore the **in**

- The formal semantics is quite tricky

Results

- CRμKlaim is a controlled version of RμKlaim
- It inherits its properties

Map of the talk

- Reversibility
- Klaim
- Uncontrolled reversibility in Klaim
- Controlling reversibility: **roll** operator
- Conclusions

Summary

- We defined uncontrolled and controlled causal-consistent reversibility for μKlaim

- Two peculiar features taken into account
  - Read dependences
    - Allow to avoid spurious dependencies
  - Localities

Future work

- Defining a low-level controlled semantics closer to an actual implementation
- Study the relationships with patterns for reliability
- Using the controlled semantics to define a reversible debugger for μKlaim
- Extend the approach to mainstream languages
  - Interesting preliminary results for actor based languages

Thanks!

Questions?

μKlaim semantics

$$\frac{[\![t]\!] = et}{l :: \mathbf{out}(t)@l'.P \;\; \| \;\; l' :: \mathbf{nil} \;\mapsto\; l :: P \,\|\, l' :: \langle et \rangle} \; (Out)$$

μKlaim semantics

$$\frac{[\![t]\!] = et}{l :: \mathbf{out}(t)@l'.P \;\; || \;\; l' :: \mathbf{nil} \;\mapsto\; l :: P \,||\, l' :: \langle et \rangle} \; (Out)$$

$$\frac{match([\![T]\!], et) = \sigma}{l :: \mathbf{in}(T)@l'.P \;\; || \;\; l' :: \langle et \rangle \;\mapsto\; l :: P\sigma \,||\, l' :: \mathbf{nil}} \; (In)$$

μKlaim semantics

$$\frac{[\![t]\!] = et}{l :: \mathbf{out}(t)@l'.P \;\parallel\; l' :: \mathbf{nil} \;\mapsto\; l :: P \parallel l' :: \langle et \rangle} \quad (Out)$$

$$\frac{match([\![T]\!], et) = \sigma}{l :: \mathbf{in}(T)@l'.P \;\parallel\; l' :: \langle et \rangle \;\mapsto\; l :: P\sigma \parallel l' :: \mathbf{nil}} \quad (In)$$

$$\frac{match([\![T]\!], et) = \sigma}{l :: \mathbf{read}(T)@l'.P \;\parallel\; l' :: \langle et \rangle \;\mapsto\; l :: P\sigma \parallel l' :: \langle et \rangle} \quad (Read)$$

$$l :: \mathbf{newloc}(l').P \;\mapsto\; (\nu l')(l :: P \parallel l' :: \mathbf{nil}) \quad (New)$$

$$l :: \mathbf{eval}(Q)@l'.P \;\parallel\; l' :: \mathbf{nil} \;\mapsto\; l :: P \parallel l' :: Q \quad (Eval)$$

μKlaim semantics

Evaluation-closed relation

A relation is evaluation closed if it is closed under active contexts

$N1 \longmapsto N1'$ implies $N1 \parallel N2 \longmapsto N1' \parallel N2$ and $(\nu l)\, N1 \longmapsto (\nu l) N1'$

and under structural congruence

$N \equiv M \longmapsto M' \equiv N'$ implies $N \longmapsto N'$

μKlaim semantics

The μKlaim reduction relation $\longmapsto$ is the smallest evaluation-closed relation satisfying the rules in previous slide

CRμKlaim semantics

$$l :: k : \mathbf{eval}_\gamma(Q)@l'.P \parallel l' :: \mathbf{empty}$$
$$\mapsto_c (\nu k', k'')(\, l :: k' : P[k/\gamma] \mid [k : \mathbf{eval}_\gamma(Q)@l'.P; k''; k'] \parallel l' :: k'' : Q\,) \quad (Eval)$$

$$\frac{\begin{array}{c} M = (\nu \tilde{z}) l :: k' : \mathbf{roll}(k) \parallel l' :: [k : a.P; \xi] \parallel N \quad k <: M \quad \mathtt{complete}(M) \\ N_t = l'' :: h : \langle t \rangle \text{ if } a = \mathbf{in}_\gamma(T)@l'' \wedge \xi = h : \langle t \rangle; k'', \text{ otherwise } N_t = \mathbf{0} \\ N_l = \mathbf{0} \text{ if } k <:_M l, \text{ otherwise } N_l = l :: \mathbf{empty} \end{array}}{(\nu \tilde{z}) l :: k' : \mathbf{roll}(k) \parallel l' :: [k : a.P; \xi] \parallel N \rightsquigarrow_c l' :: k : a.P \parallel N_t \parallel N_l \parallel N \lightning_k} \quad (Roll)$$

- M is complete and depends on k
- N_t: if the undone action is an **in**, we should release the tuple
- N_l: we should not consume the **roll** locality, unless created by the undone computation
- $N\lightning_k$: resources consumed by the computation should be released