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 $\mu$Klaim
\( \mu \text{Klaim syntax} \)

<table>
<thead>
<tr>
<th>Nets</th>
<th>( N ::= 0 )</th>
<th>( l :: C )</th>
<th>( N_1 \parallel N_2 )</th>
<th>( (\nu l)N )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Components</td>
<td>( C ::= \langle et \rangle )</td>
<td>( P )</td>
<td>( C_1 )</td>
<td>( C_2 )</td>
</tr>
<tr>
<td>Processes</td>
<td>( P ::= \text{nil} )</td>
<td>( a.P )</td>
<td>( P_1 )</td>
<td>( P_2 )</td>
</tr>
<tr>
<td>Actions</td>
<td>( a ::= \text{out}(t)@l )</td>
<td>( \text{eval}(P)@l )</td>
<td>( \text{in}(T)@l )</td>
<td>( \text{read}(T)@l )</td>
</tr>
</tbody>
</table>
μKlaim syntax

(Nets) \( N ::= 0 \mid l :: C \mid N_1 || N_2 \mid (\nu l)N \)

(Component) \( C ::= \langle et \rangle \mid P \mid C_1 | C_2 \)

(Processes) \( P ::= \text{nil} \mid a.P \mid P_1 | P_2 \mid A \)

(Actions) \( a ::= \text{out}(t)@l \mid \text{eval}(P)@l \mid \text{in}(T)@l \mid \text{read}(T)@l \mid \text{newloc}(l) \)
\[ \text{\(\mu\text{Klaim syntax}\)} \]

(Nets) \( N ::= 0 \mid l :: C \mid N_1 \parallel N_2 \mid (\nu l)N \)

(Components) \( C ::= \langle et \rangle \mid P \mid C_1 \mid C_2 \)

(Processes) \( P ::= \text{nil} \mid a.P \mid P_1 \mid P_2 \mid A \)

(Actions) \( a ::= \text{out}(t)@l \mid \text{eval}(P)@l \mid \text{in}(T)@l \mid \text{read}(T)@l \mid \text{newloc}(l) \)
### μKlaim syntax

**Nets**

$$N ::= 0 \mid l :: C \mid N_1 \parallel N_2 \mid (\nu l)N$$

**Components**

$$C ::= \langle et \rangle \mid P \mid C_1 | C_2$$

**Processes**

$$P ::= \text{nil} \mid a.P \mid P_1 | P_2 \mid A$$

**Actions**

$$a ::= \text{out}(t)@l \mid \text{eval}(P)@l \mid \text{in}(T)@l \mid \text{read}(T)@l \mid \text{newloc}(l)$$

**locations**

\(l\)
μKlaim syntax

(Nets) \( N ::= 0 \mid l :: C \mid N_1 \parallel N_2 \mid (\nu l)N \)

(Components) \( C ::= \langle et \rangle \mid P \mid C_1 \mid C_2 \)

(Processes) \( P ::= \text{nil} \mid a.P \mid P_1 \mid P_2 \mid A \)

(Actions) \( a ::= \text{out}(t)@l \mid \text{eval}(P)@l \mid \text{in}(T)@l \mid \text{read}(T)@l \mid \text{newloc}(l) \)
μKlaim syntax

(Nets) \[ N ::= 0 \mid l :: C \mid N_1 \parallel N_2 \mid (vl)N \]

(Components) \[ C ::= \langle et \rangle \mid P \mid C_1 \mid C_2 \]

(Processes) \[ P ::= \text{n}il \mid a.P \mid P_1 \mid P_2 \mid A \]

(Actions) \[ a ::= \text{out}(t)@l \mid \text{eval}(P)@l \mid \text{in}(T)@l \mid \text{read}(T)@l \mid \text{newloc}(l) \]
μKlaim syntax

(Nets) \( N ::= 0 \mid l :: C \mid N_1 \parallel N_2 \mid (vl)N \)

(Components) \( C ::= \langle et \rangle \mid P \mid C_1 \mid C_2 \)

(Processes) \( P ::= \text{nil} \mid a.P \mid P_1 \mid P_2 \mid A \)

(Actions) \( a ::= \text{out}(t)@l \mid \text{eval}(P)@l \mid \text{in}(T)@l \mid \text{read}(T)@l \mid \text{newloc}(l) \)
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) \( P ::= \text{nil} \mid a.P \mid P_1 \mid P_2 \mid A \)

(Actions) \( a ::= \text{out}(t)@l \mid \text{eval}(P)@l \mid \text{in}(T)@l \mid \text{read}(T)@l \mid \text{newloc}(l) \)
μKlaim syntax

(Nets) \( N ::= \emptyset \mid l :: C \mid N_1 \parallel N_2 \mid (\nu l)N \)

Components) \( C ::= \langle et \rangle \mid P \mid C_1 \mid C_2 \)

(Processes) \( P ::= \text{nil} \mid a.P \mid P_1 \mid P_2 \mid A \)

(Actions) \( a ::= \text{out}(t)@l \mid \text{eval}(P)@l \mid \text{in}(T)@l \mid \text{read}(T)@l \mid \text{newloc}(l) \)

\( l \)
### μKlaim Syntax

#### Nets
\[
N ::= 0 \mid l :: C \mid N_1 \parallel N_2 \mid (vl)N
\]

#### Components
\[
C ::= \langle et \rangle \mid P \mid C_1 \mid C_2
\]

#### Processes
\[
P ::= \text{nil} \mid a.P \mid P_1 \mid P_2 \mid A
\]

#### Actions
\[
a ::= \text{out}(t)@l \mid \text{eval}(P)@l \mid \text{in}(T)@l \mid \text{read}(T)@l \mid \text{newloc}(l)
\]
μKlaim syntax

(Nets) \( N ::= 0 \mid l :: C \mid N_1 || N_2 \mid (\nu l)N \)

(Components) \( C ::= \langle et \rangle \mid P \mid C_1 \mid C_2 \)

(Processes) \( P ::= \text{nil} \mid a.P \mid P_1 \mid P_2 \mid A \)

(Actions) \( a ::= \text{out}(t)@l \mid \text{eval}(P)@l \mid \text{in}(T)@l \mid \text{read}(T)@l \mid \text{newloc}(l) \)
μKlaim syntax

(Nets) $N ::= 0 \mid l :: C \mid N_1 \parallel N_2 \mid (\nu l)N$

(Components) $C ::= \langle et \rangle \mid P \mid C_1 \mid C_2$

(Processes) $P ::= \text{nil} \mid a.P \mid P_1 \parallel P_2 \mid A$

(Actions) $a ::= \text{out}(t)@l \mid \text{eval}(P)@l \mid \text{in}(T)@l \mid \text{read}(T)@l \mid \text{newloc}(l)$

\[ l \]

\[ \langle et_2 \rangle \langle et_3 \rangle \]

\[ \text{read}(T_2)@l \]
μKlaim syntax

\[
\begin{align*}
(Nets) \quad N &::= 0 \quad | \quad l :: C \quad | \quad N_1 \parallel N_2 \quad | \quad (vl)N \\
(Components) \quad C &::= \langle et \rangle \quad | \quad P \quad | \quad C_1 \mid C_2 \\
(Processes) \quad P &::= \text{nil} \quad | \quad a.P \quad | \quad P_1 \mid P_2 \quad | \quad A \\
(Actions) \quad a &::= \text{out}(t)@l \quad | \quad \text{eval}(P)@l \quad | \quad \text{in}(T)@l \quad | \quad \text{read}(T)@l \quad | \quad \text{newloc}(l)
\end{align*}
\]
μKlaim syntax

(Nets) \( N ::= 0 \mid l :: C \mid N_1 || N_2 \mid (\nu l)N \)

(Components) \( C ::= \langle et \rangle \mid P \mid C_1 | C_2 \)

(Processes) \( P ::= \text{nil} \mid a.P \mid P_1 | P_2 \mid A \)

(Actions) \( a ::= \text{out}(t)@l \mid \text{eval}(P)@l \mid \text{in}(T)@l \mid \text{read}(T)@l \mid \text{newloc}(l) \)

\( l \)

\( \text{newloc}(l') \)

\( \langle et_2 \rangle \langle et_3 \rangle \)
μKlaim syntax

(Nets) \( N ::= 0 \mid l ::= C \mid N_1 \parallel N_2 \mid (\nu l)N \)

(Components) \( C ::= \langle et \rangle \mid P \mid C_1 \parallel C_2 \)

(Processes) \( P ::= \text{nil} \mid a.P \mid P_1 \parallel P_2 \mid A \)

(Actions) \( a ::= \text{out}(t)@l \mid \text{eval}(P)@l \mid \text{in}(T)@l \mid \text{read}(T)@l \mid \text{newloc}(l) \)

\[ l \quad \text{newloc}(l') \quad l' \]

\[ \langle et_2 \rangle \langle et_3 \rangle \quad \]
Example

\[ l_1 :: \langle \text{foo} \rangle \parallel l_2 :: \text{read}(\text{foo})@l_1.P \parallel l_3 :: \text{read}(\text{foo})@l_1.P' \]
Map of the talk

- Reversibility
- Klaim
- Uncontrolled reversibility in Klaim
- Controlling reversibility: roll operator
- Conclusions
Making $\mu$Klaim reversible

- We define $R\mu$Klaim, an extension of $\mu$Klaim allowing:
  - *forward* actions, corresponding to $\mu$Klaim 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 $k_1 < (k_2, k_3)$ to store causality information
  - We use memories to store past actions

- Similarly to past works on other languages
Making $\mu$Klaim reversible

- We have to deal with some peculiarities of $\mu$Klaim causality structure

- Read dependencies
  - Two reads on the same tuple should not create dependences
  - If the out creating the tuple is undone then reads 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
RᵦKlaim syntax

(Nets) $N ::= 0 \mid l :: C \mid l :: \text{empty} \mid N_1 || N_2 \mid (\nu z)N$

(Components) $C ::= k::\langle et \rangle \mid k::P \mid C_1 | C_2 \mid \mu \mid k_1 < (k_2, k_3)$

(Processes) $P ::= \text{nil} \mid a.P \mid P_1 | P_2 \mid A$

(Actions) $a ::= \text{out}(t)@l \mid \text{eval}(P)@l$
$\mid \text{in}(T)@l \mid \text{read}(T)@l \mid \text{newloc}(l)$

(Memories) $\mu ::= [k::\text{out}(t)@l; k''; k'] \mid [k::\text{in}(T)@l.P; h::\langle et \rangle; k']$
$\mid [k::\text{read}(T)@l.P; h; k'] \mid [k::\text{newloc}(l); k']$
$\mid [k::\text{eval}(Q)@l; k''; k']$
Example

\[
\begin{align*}
  l_1 &:: k_1: \langle \text{foo} \rangle \lor l_2:: k_2: \text{read} (\text{foo}) @ l_1.P \\
     &\lor l_3:: k_3: \text{read} (\text{foo}) @ l_1.P' \\
&\left( \nu k'_3 \right) ( l_1:: k_1: \langle \text{foo} \rangle \lor l_2:: k_2: \text{read} (\text{foo}) @ l_1.P \\
&\lor l_3:: k'_3: P' \lor [k_3: \text{read} (\text{foo}) @ l_1.P'; k_1; k'_3] ) \\
&\left( \nu k'_2 \right) ( l_1:: k_1: \langle \text{foo} \rangle \\
&\lor l_2:: k'_2: P \lor [k_2: \text{read} (\text{foo}) @ l_1.P; k_1; k'_2] \\
&\lor l_3:: k_3: \text{read} (\text{foo}) @ l_1.P' ) \\
&\left( \nu k'_2, k'_3 \right) ( l_1:: k_1: \langle \text{foo} \rangle \\
&\lor l_2:: k'_2: P \lor [k_2: \text{read} (\text{foo}) @ l_1.P; k_1; k'_2] \\
&\lor l_3:: k'_3: P' \lor [k_3: \text{read} (\text{foo}) @ l_1.P'; k_1; k'_3] )
\end{align*}
\]
Example

\[
\begin{align*}
  l_1 &:: k_1 : \langle \text{foo} \rangle & l_2 &:: k_2 : \text{in} (\text{foo}) @ l_1. \text{out} (\text{foo}) @ l_1. P \\
                           & \parallel l_3 &:: k_3 : \text{in} (\text{foo}) @ l_1. \text{out} (\text{foo}) @ l_1. P' 
\end{align*}
\]

- execute **in** in \( l_2 \)
- execute **out** in \( l_2 \)
- execute **in** in \( l_3 \)
- execute **out** in \( l_3 \)

\[
\begin{align*}
  (\nu k'_2, k''_2, k'''_2, k'_3, k''_3, k'''_3)( l_1 &:: k'''_3 : \langle \text{foo} \rangle \\
                           & \parallel l_2 &:: k''_2 : P | [k_2 : \text{in} (\text{foo}) @ l_1. \text{out} (\text{foo}) @ l_1. P; k_1 : \langle \text{foo} \rangle; k'_2] \\
                           & \parallel l_3 &:: k''_3 : P' | [k_3 : \text{in} (\text{foo}) @ l_1. \text{out} (\text{foo}) @ l_1. P'; k''_2 : \langle \text{foo} \rangle; k'_3] \\
                           &                  | [k'_3 : \text{out} (\text{foo}) @ l_1; k'''_3; k''_3] )
\end{align*}
\]
Example

\[ l_1 :: k_1 : \langle \text{foo} \rangle \parallel l_2 :: k_2 : \text{in}(\text{foo})@l_1.\text{out}(\text{foo})@l_1.P \parallel l_3 :: k_3 : \text{in}(\text{foo})@l_1.\text{out}(\text{foo})@l_1.P' \]

execute \text{in} in \( l_2 \)

execute \text{out} in \( l_2 \)

execute \text{in} in \( l_3 \)

execute \text{out} in \( l_3 \)

it needs \( k_2''' : \langle \text{foo} \rangle \) in \( l_1 \) to perform a backward step
Properties

- The forward semantics of $R_\mu$Klaim is an annotated version of the semantics of $\mu$Klaim

- The Loop Lemma holds
  - i.e., each reduction in $R_\mu$Klaim has an inverse

- $R_\mu$Klaim is causally consistent
Concurrency in $R\mu$Klaim

- 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 $\lambda$ on memories
  
  $$
  \lambda([k : \text{out}(t) @ l; k''; k']) = \{k, k', k'', r(l)\}
  
  \lambda([k : \text{in}(T) @ l.P; k'' : \langle et \rangle; k']) = \{k, k', k'', r(l)\}
  
  \lambda([k : \text{read}(T) @ l.P; k''; k']) = \{k, r(k''), k', r(l)\}
  
  \lambda([k : \text{eval}(Q) @ l; k''; k']) = \{k, k', k'', r(l)\}
  
  \lambda([k : \text{newloc}(l); k']) = \{k, k', l\}
  $$

- **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 $\mu$Klaim a **roll** operator
  - To undo a given past action
  - Together with all its consequences (and only them)
- We call CR$\mu$Klaim the extension of $\mu$Klaim with **roll**
CRµKlaim syntax

(Nets) \( N ::= 0 \mid \mathit{l :: C} \mid \mathit{l :: empty} \mid N_1 \parallel N_2 \mid (\nu z)N \)

(Components) \( C ::= k:\langle\mathit{et}\rangle \mid k:\mathit{P} \mid C_1 \mid C_2 \mid \mu \mid k_1 \prec (k_2, k_3) \)

(Processes) \( P ::= \mathit{nil} \mid a.P \mid P_1 \mid P_2 \mid A \mid \text{roll}(\nu) \)

(Actions) \( a ::= \mathit{out}_\gamma(t)@l \mid \mathit{eval}_\gamma(P)@l \mid \mathit{in}_\gamma(T)@l \mid \mathit{read}_\gamma(T)@l \mid \mathit{newloc}_\gamma(l) \)

(Memories) \( \mu ::= [k : \mathit{out}_\gamma(t)@l.P; k''; k'] \mid [k : \mathit{in}_\gamma(T)@l.P; h : \langle t \rangle; k'] \mid [k : \mathit{read}_\gamma(T)@l.P; h; k'] \mid [k : \mathit{newloc}_\gamma(l).P; k'] \mid [k : \mathit{eval}_\gamma(Q)@l.P; k''; k'] \)
Example

- From
  
  \[ l :: k : \text{out}_\gamma(\text{foo})@l.\text{roll}(\gamma) \parallel l' :: k' : \text{in}(\text{foo})@l.\text{nil} \]

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

- When we undo the \textbf{out} we need to restore the \textbf{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

\[
\frac{[t] = et}{l :: \text{out}(t)@l'.P || l' :: \text{nil} \leftrightarrow l :: P || l' :: \langle et \rangle} \quad (Out)
\]
\( [t] = et \)

\[
\begin{align*}
\text{(Out)} & \quad l :: \text{out}(t)@l'.P \parallel l' :: \text{nil} \rightarrow l :: P \parallel l' :: \langle et \rangle \\
\text{(In)} & \quad \text{match}(\llbracket T \rrbracket, et) = \sigma \\
& \quad l :: \text{in}(T)@l'.P \parallel l' :: \langle et \rangle \rightarrow l :: P\sigma \parallel l' :: \text{nil}
\end{align*}
\]
μKlaim semantics

\[
[t] = et
\]

\[
l ::\ out(t)@l'.P \parallel l' ::\ nil \rightarrow l :: P \parallel l' :: \langle et \rangle
\]

\[
\text{match}([T], et) = \sigma
\]

\[
l ::\ in(T)@l'.P \parallel l' :: \langle et \rangle \rightarrow l :: P\sigma \parallel l' ::\ nil
\]

\[
\text{match}([T], et) = \sigma
\]

\[
l ::\ read(T)@l'.P \parallel l' :: \langle et \rangle \rightarrow l :: P\sigma \parallel l' :: \langle et \rangle
\]

\[
l ::\ newloc(l').P \rightarrow (\nu l')(l :: P \parallel l' :: \nil) \quad \text{(New)}
\]

\[
l ::\ eval(Q)@l'.P \parallel l' :: \nil \rightarrow l :: P \parallel l' :: Q \quad \text{(Eval)}
\]
μKlaim semantics

Evaluation-closed relation

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

\[ \text{N1 } \leftrightarrow \text{N1'} \text{ implies N1 } \parallel \text{N2 } \leftrightarrow \text{N1'} \parallel \text{N2} \text{ and (vl) N1 } \leftrightarrow \text{(vl)N1'} \]

and under structural congruence

\[ \text{N} ≡ \text{M } \leftrightarrow \text{M'} ≡ \text{N'} \text{ implies } \text{N } \leftrightarrow \text{N'} \]

μKlaim semantics

The μKlaim reduction relation \( \leftrightarrow \) is the smallest evaluation-closed relation satisfying the rules in previous slide
CRμKlaim semantics

\[
\begin{aligned}
l &::= k : \text{eval}_\gamma(Q)@l'.P \parallel l' :: \text{empty} \\
\rightarrow_c & (vk', k'') \left( l :: k' :: P[k/\gamma] \mid [k : \text{eval}_\gamma(Q)@l'.P; k''; k'] \parallel l' :: k'' :: Q \right) \\
\end{aligned}
\]

(Eval)

\[
\begin{aligned}
M &= (\nu \tilde{z})l :: k' :: \text{roll}(k) \parallel l' :: [k : a.P; \xi] \parallel N \quad k <: M \quad \text{complete}(M) \\
N_t &= l'' :: h : \langle t \rangle \text{ if } a = \text{in}_\gamma(T)@l'' \land \xi = h : \langle t \rangle; k'', \text{ otherwise } N_t = 0 \\
N_i &= 0 \text{ if } k <:_{M} l, \text{ otherwise } N_i = l :: \text{empty} \\
\overline{(\nu \tilde{z})l :: k' :: \text{roll}(k) \parallel l' :: [k : a.P; \xi] \parallel N \rightarrow_c l' :: k : a.P \parallel N_t \parallel N_i \parallel N \nless_k}
\end{aligned}
\]

(Roll)

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