Relative expressiveness of a stack of KLAIMS

Rosario Pugliese

joint work with R. De Nicola and D. Gorla

MIKADO meeting

Torino, January 2005

Aim

- Evaluate the theoretical impact of tuple-based communication
- Test the expressive power of KLAIM constructs
- Test relative expressiveness with the asynchronous π-calculus

Aim & Approach

- Evaluate the theoretical impact of tuple-based communication
- Test the expressive power of KLAIM constructs
- Test relative expressiveness with the asynchronous π-calculus

- Distilling from KLAIM a few, more and more, foundational calculi
- Studying the possibility of encoding each of the calculi in a more basilar one

Outline of the talk

- Four simpler and simpler calculi:
  - KLAIM
  - μKLAIM: micro-KLAIM
  - cKLAIM: core-KLAIM
  - l-cKLAIM: Local core-KLAIM

Outline of the talk

- Four simpler and simpler calculi:
  - KLAIM
  - μKLAIM: micro-KLAIM
  - CKLAIM: core-KLAIM
  - L-CKLAIM: Local core-KLAIM
- Encodings of high-level constructs in lower-level ones

Outline of the talk

- Four simpler and simpler calculi:
  - KLAIM
  - μKLAIM: micro-KLAIM
  - CKLAIM: core-KLAIM
  - L-CKLAIM: Local core-KLAIM
- Encodings of high-level constructs in lower-level ones
- Encodings of the asynchronous π-calculus in CKLAIM and of L-CKLAIM in the asynchronous π-calculus

Outline of the talk

- Four simpler and simpler calculi:
  - KLAIM
  - μKLAIM: micro-KLAIM
  - cKLAIM: core-KLAIM
  - l-cKLAIM: Local core-KLAIM
- Encodings of high-level constructs in lower-level ones
- Encodings of the asynchronous π-calculus in cKLAIM and of l-cKLAIM in the asynchronous π-calculus
- Assessment of the encodings

Outline of the talk

- Four simpler and simpler calculi:
  - KLAIM
  - μKLAIM: micro-KLAIM
  - cKLAIM: core-KLAIM
  - L-cKLAIM: Local core-KLAIM
- Encodings of high-level constructs in lower-level ones
- Encodings of the asynchronous π-calculus in cKLAIM and of L-cKLAIM in the asynchronous π-calculus
- Assessment of the encodings
- Conclusions

KLAIM

A **K**ernel **L**anguage for **A**gent **I**nteraction and **M**obility

- Process Calculus Flavored
  - Small set of basic combinator
  - Clean operational semantics

Klaim

A Kernel Language for Agent Interaction and Mobility

- Process Calculus Flavored
  - Small set of basic combinator
  - Clean operational semantics
- Linda-based communication model
  - Asynchronous communication
  - Shared tuple spaces
  - Pattern Matching

KLAIM

A Kernel Language for Agent Interaction and Mobility

- Process Calculus Flavored
  - Small set of basic combinator
  - Clean operational semantics
- Linda-based communication model
  - Asynchronous communication
  - Shared tuple spaces
  - Pattern Matching
- Explicit use of localities
  - Multiple distributed tuple spaces
  - Code and Process mobility

Tuples and Pattern Matching

- Tuples: $(\text{"}foo\text{"}, 10 + 5, true)$

Tuples and Pattern Matching

- Tuples: $(\text{“}foo\text{”}, 10 + 5, true)$
- Templates: $(!s, 15, !b)$
    - Formal Fields
    - Actual Fields

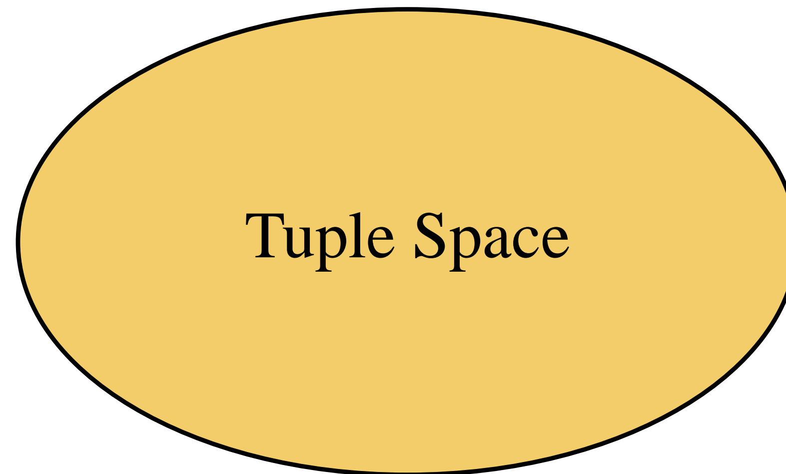
Tuples and Pattern Matching

- Tuples: $(``foo", 10+5, true)$
- Templates: $(!s, 15, !b)$
  - Formal Fields
  - Actual Fields
- Pattern Matching:
  - Formal fields match any field of the same type
  - Actual fields match if identical

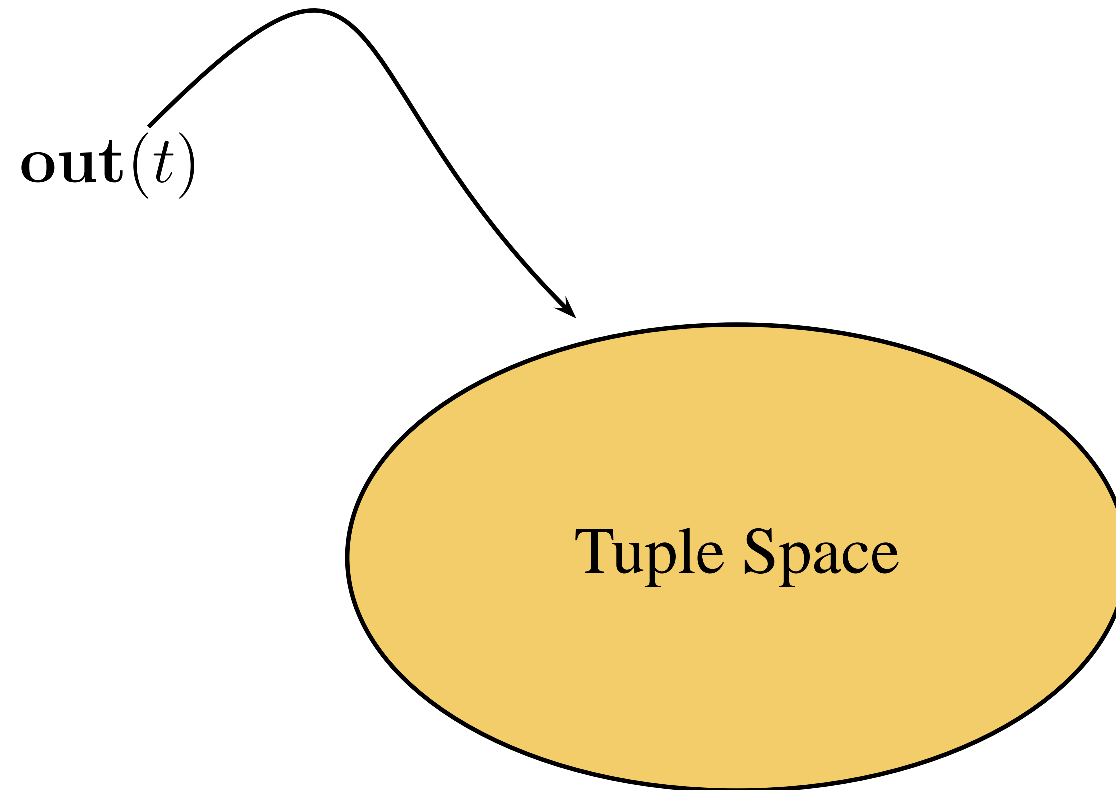
$(!s, 15, !b)$ matches $(``foo", 10+5, true)$

$(!s, 15, !b)$ does not match $(``foo", 10, true)$

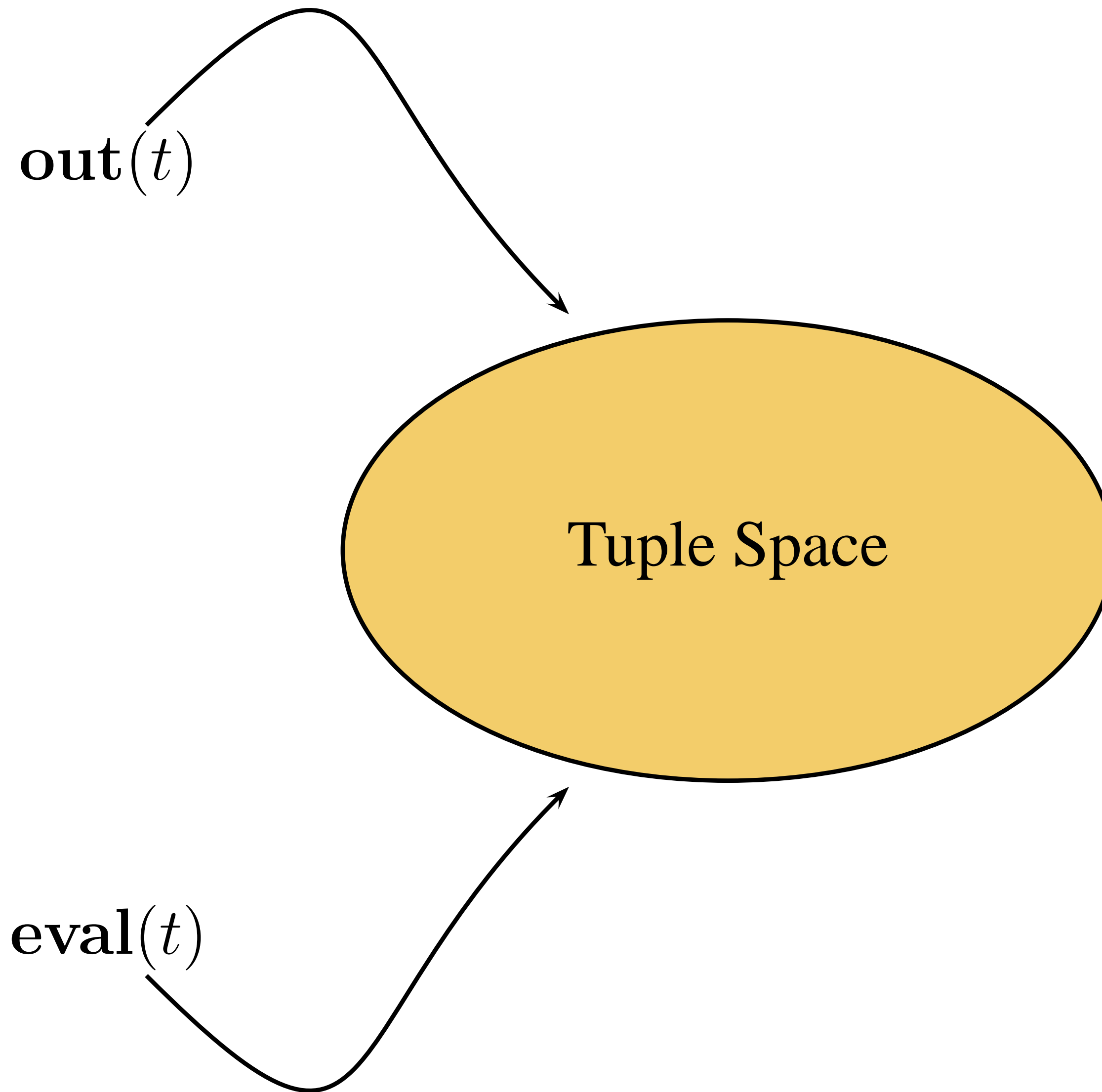
Linda communication model



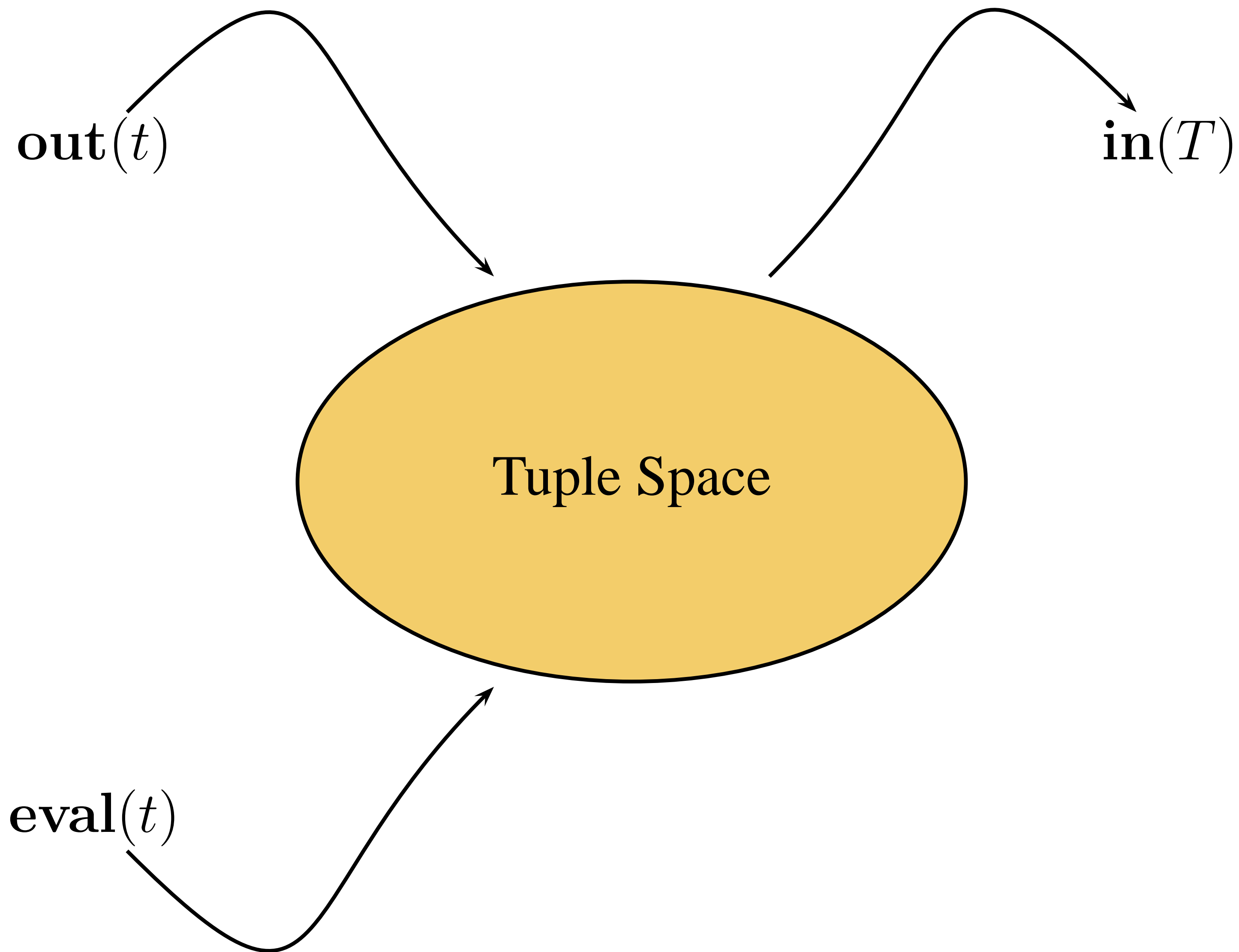
Linda communication model



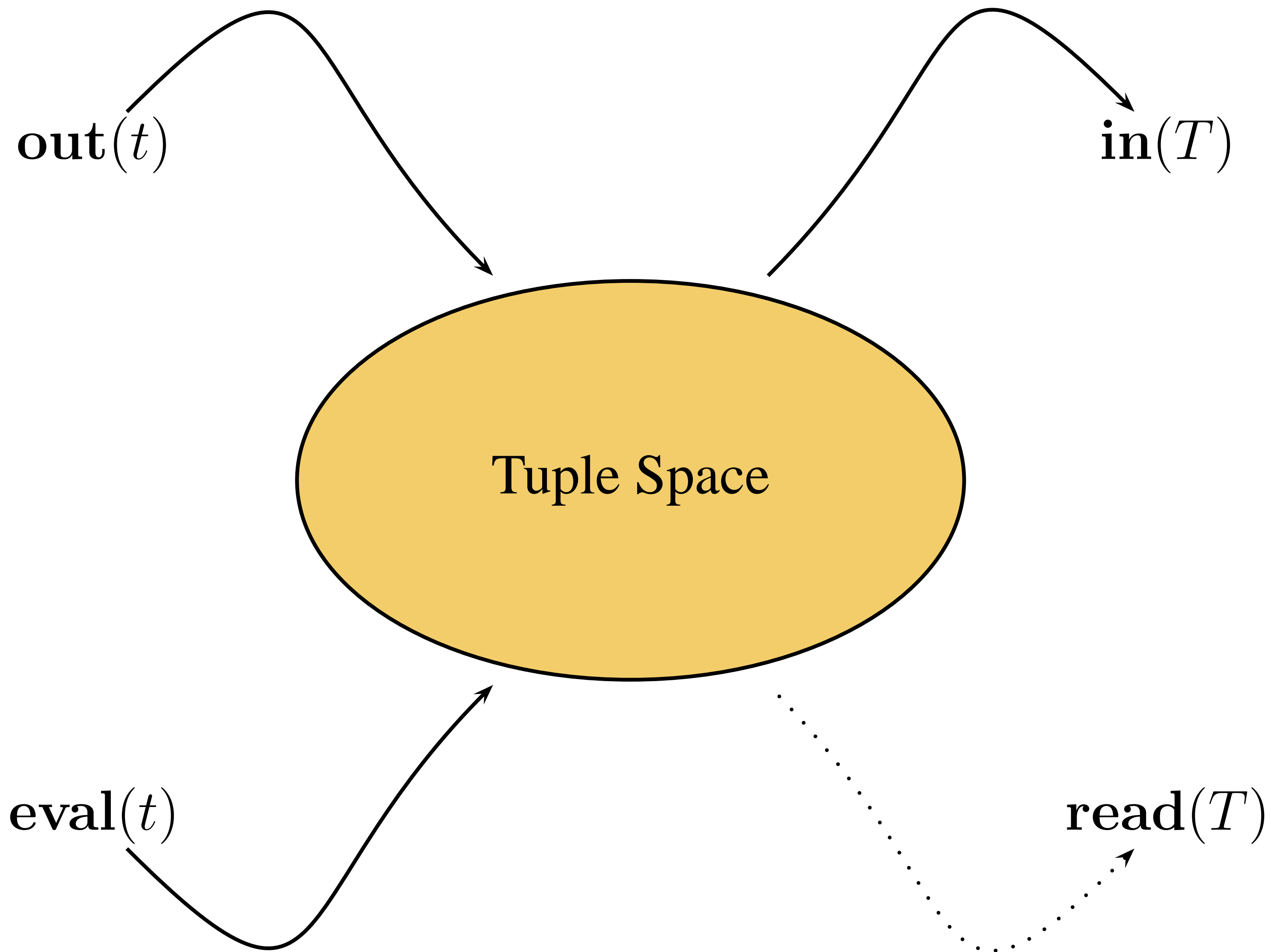
Linda communication model



Linda communication model



Linda communication model



From Linda to KLAIM

- Localities to model distribution
  - Physical localities (network addresses), i.e. names
  - Logical localities (aliases for addresses), i.e. variables
  - A distinct variable `self` indicates the node a process is on
  - Local and Remote Operations
    (to read/withdraw/write data, spawn processes, create nodes)

From Linda to KLAIM

- Localities to model distribution
  - Physical localities (network addresses), i.e. names
  - Logical localities (aliases for addresses), i.e. variables
  - A distinct variable `self` indicates the node a process is on
  - Local and Remote Operations
    (to read/withdraw/write data, spawn processes, create nodes)
- Allocation environments to associate physical to logical localities
  - Logical localities are local aliases for network addresses
  - Programmers may not know the exact topology

From Linda to KLAIM

- Localities to model distribution
  - Physical localities (network addresses), i.e. names
  - Logical localities (aliases for addresses), i.e. variables
  - A distinct variable `self` indicates the node a process is on
  - Local and Remote Operations
    (to read/withdraw/write data, spawn processes, create nodes)
- Allocation environments to associate physical to logical localities
  - Logical localities are local aliases for network addresses
  - Programmers may not know the exact topology
- Process Algebras Operators to specify behaviours
  - Action prefix
  - Parallel composition
  - Name creation

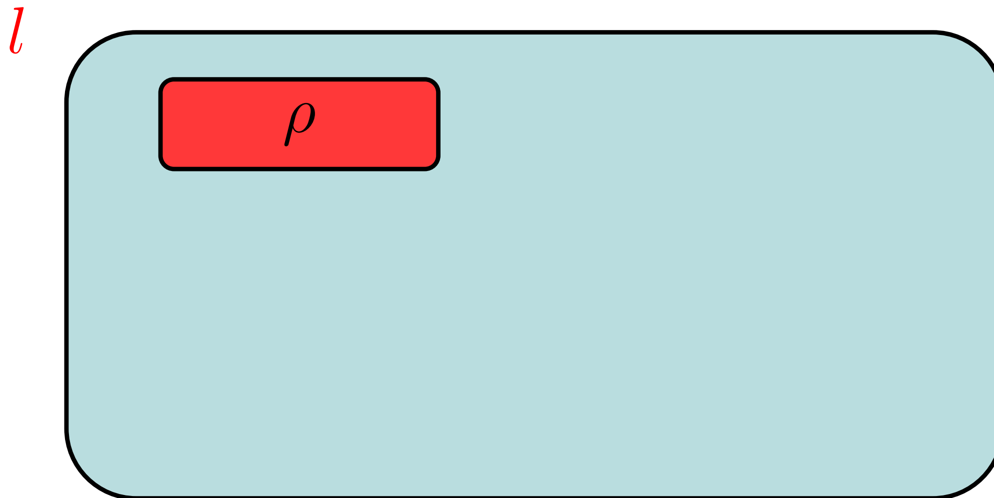
Klaim Nodes

KLAIM Nodes

- Locality name (address)

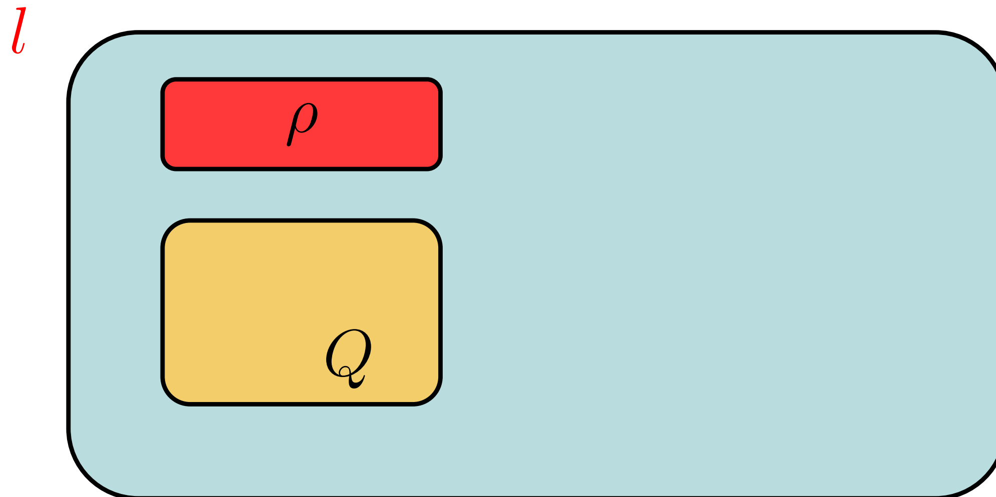
KLAIM **Nodes**

- Locality name (address)
- Allocation Environment



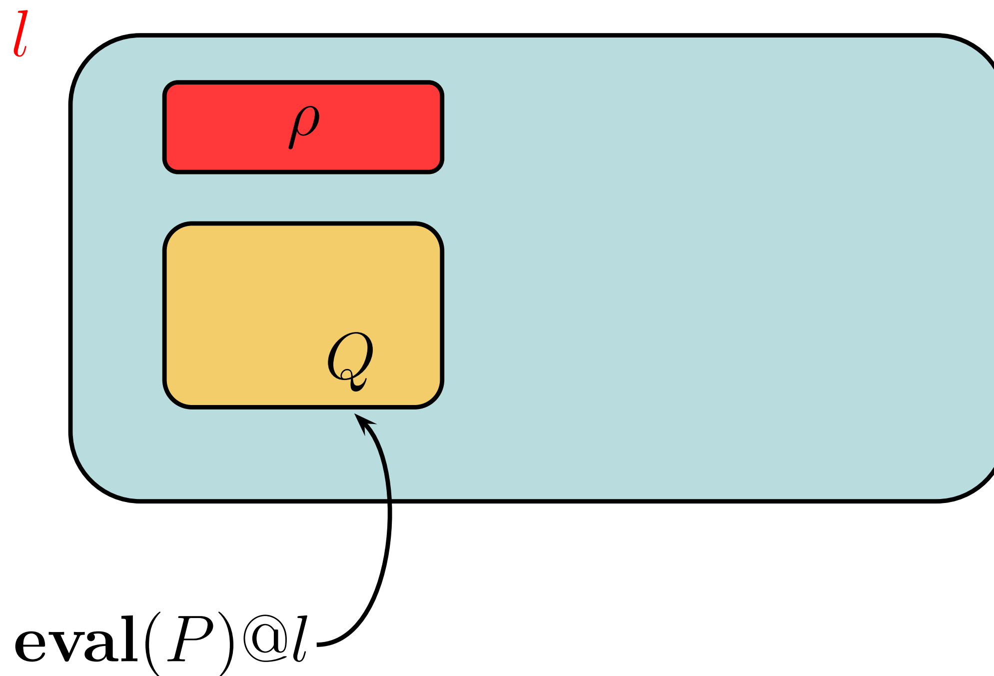
KLAIM Nodes

- Locality name (address)
- Allocation Environment
- Processes



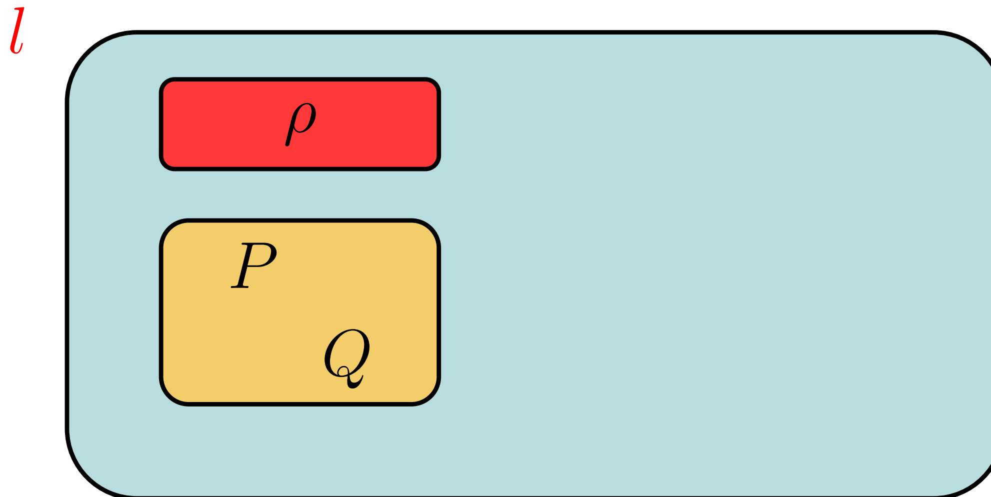
KLAIM Nodes

- Locality name (address)
- Allocation Environment
- Processes



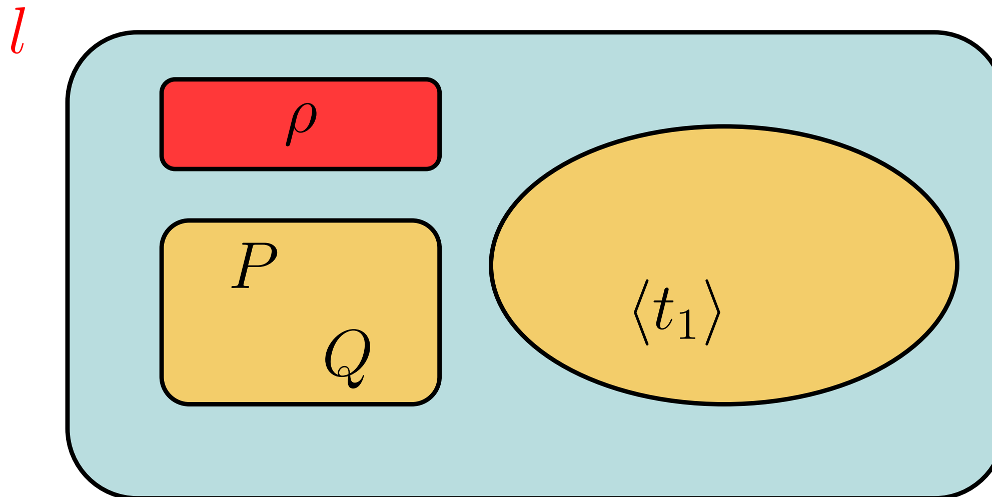
KLAIM Nodes

- Locality name (address)
- Allocation Environment
- Processes



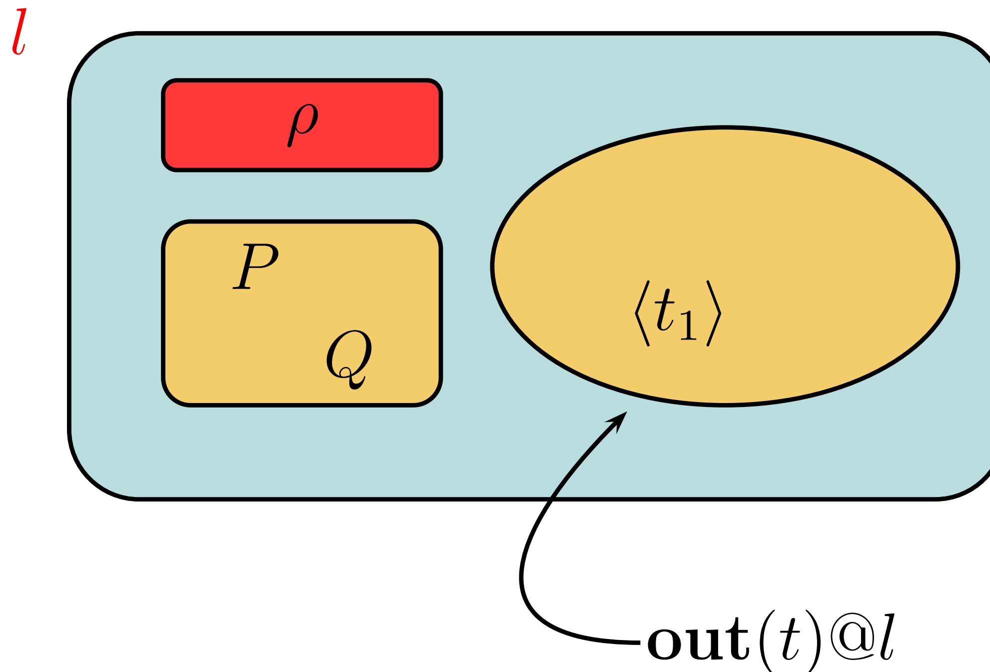
KLAIM Nodes

- Locality name (address)
- Allocation Environment
- Processes
- Tuple Space



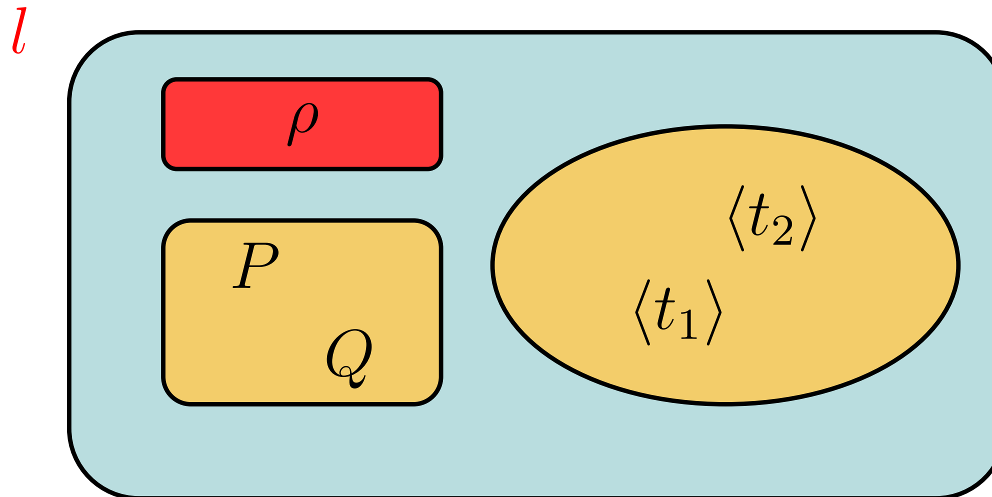
KLAIM Nodes

- Locality name (address)
- Allocation Environment
- Processes
- Tuple Space



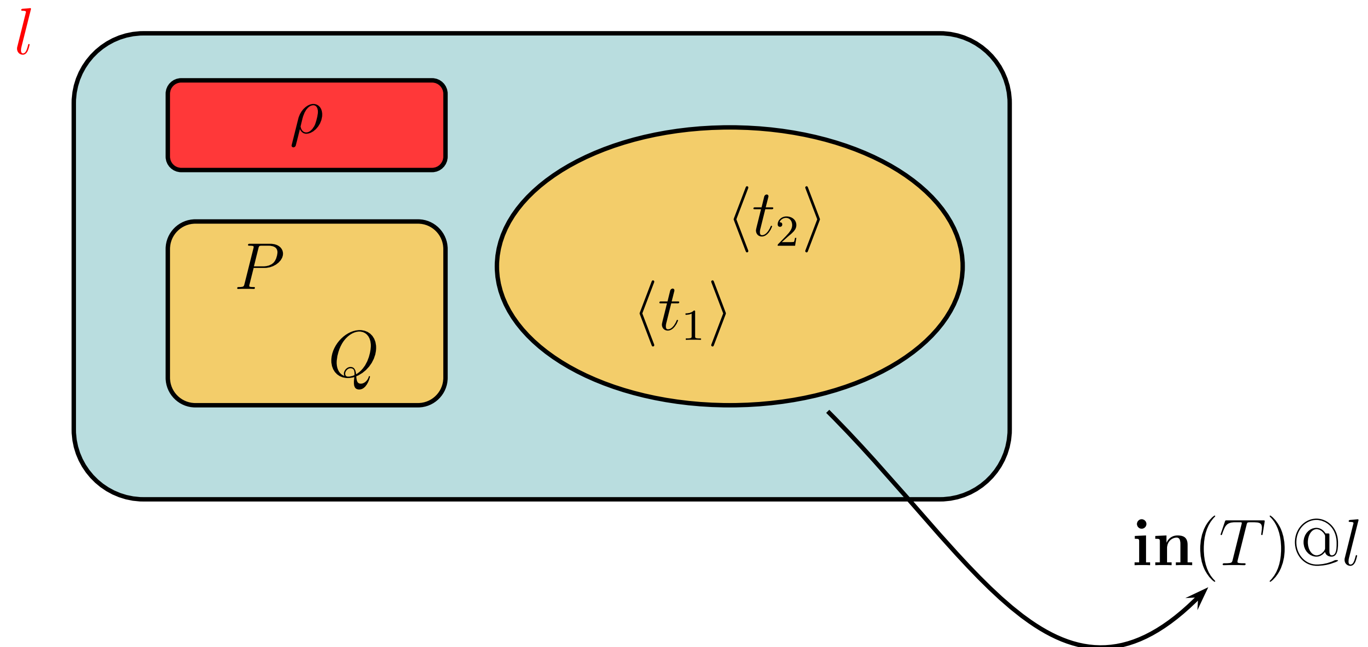
KLAIM Nodes

- Locality name (address)
- Allocation Environment
- Processes
- Tuple Space



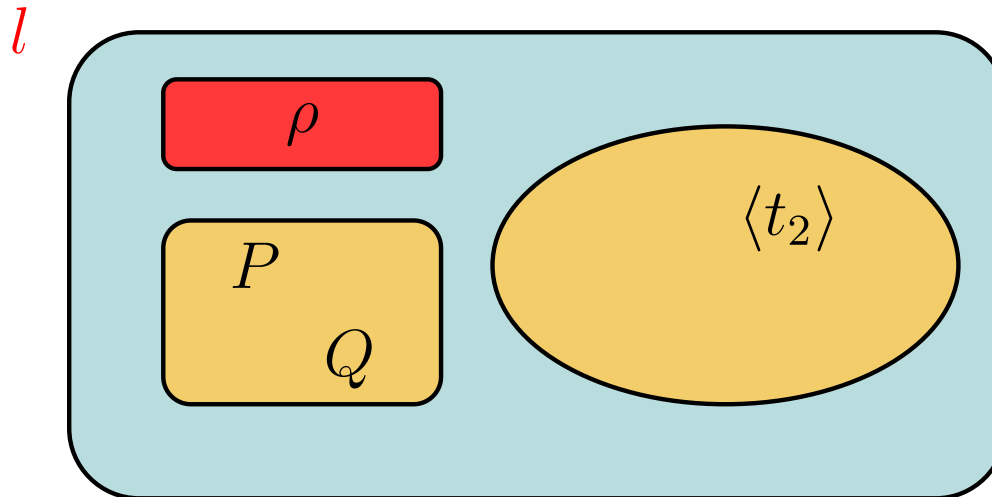
KLAIM Nodes

- Locality name (address)
- Allocation Environment
- Processes
- Tuple Space



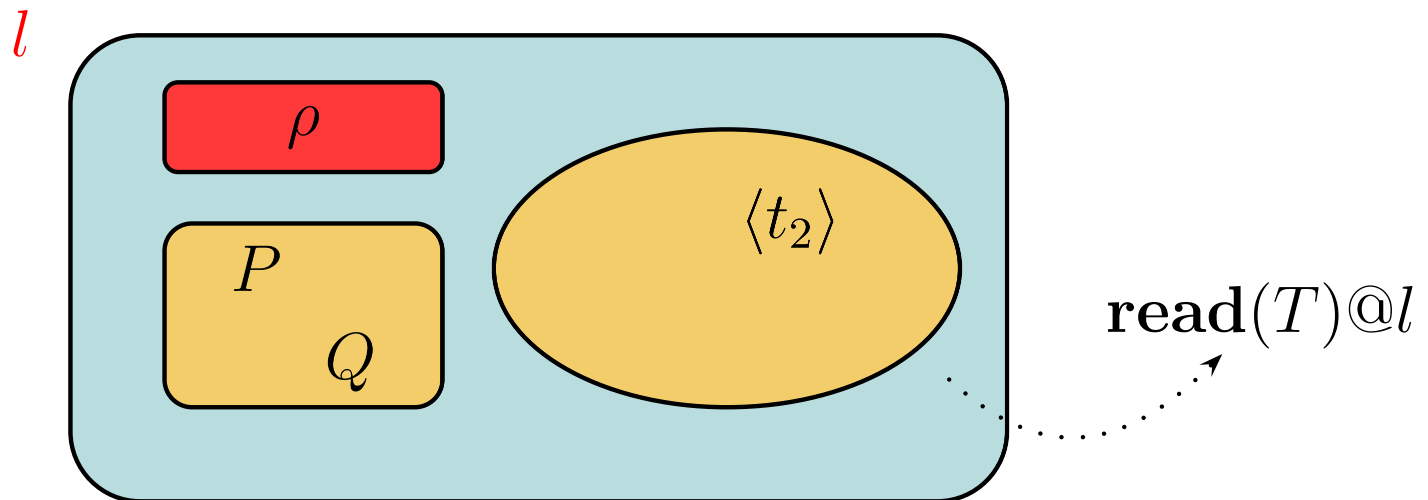
KLAIM Nodes

- Locality name (address)
- Allocation Environment
- Processes
- Tuple Space



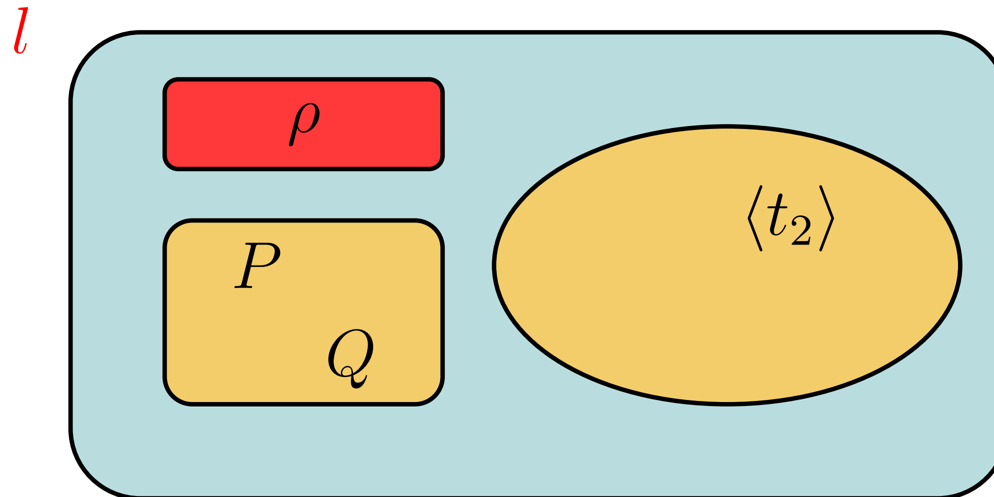
Klaim Nodes

- Locality name (address)
- Allocation Environment
- Processes
- Tuple Space



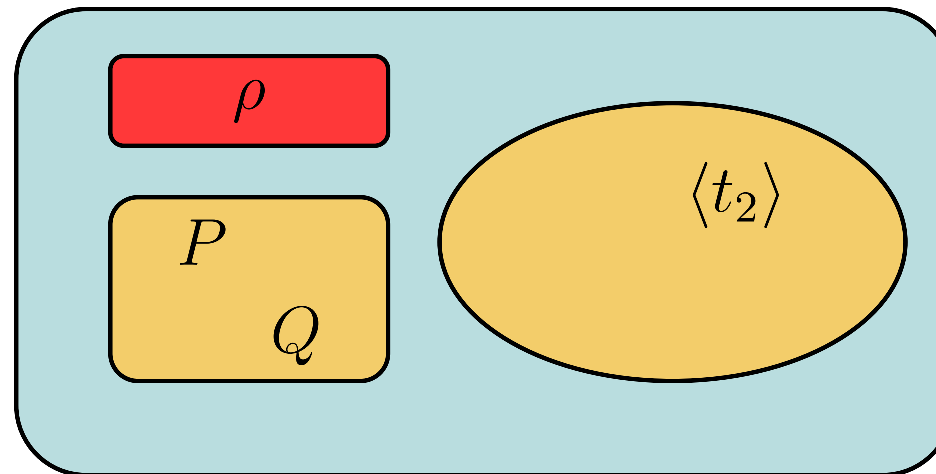
KLAIM Nodes

- Locality name (address)
- Allocation Environment
- Processes
- Tuple Space



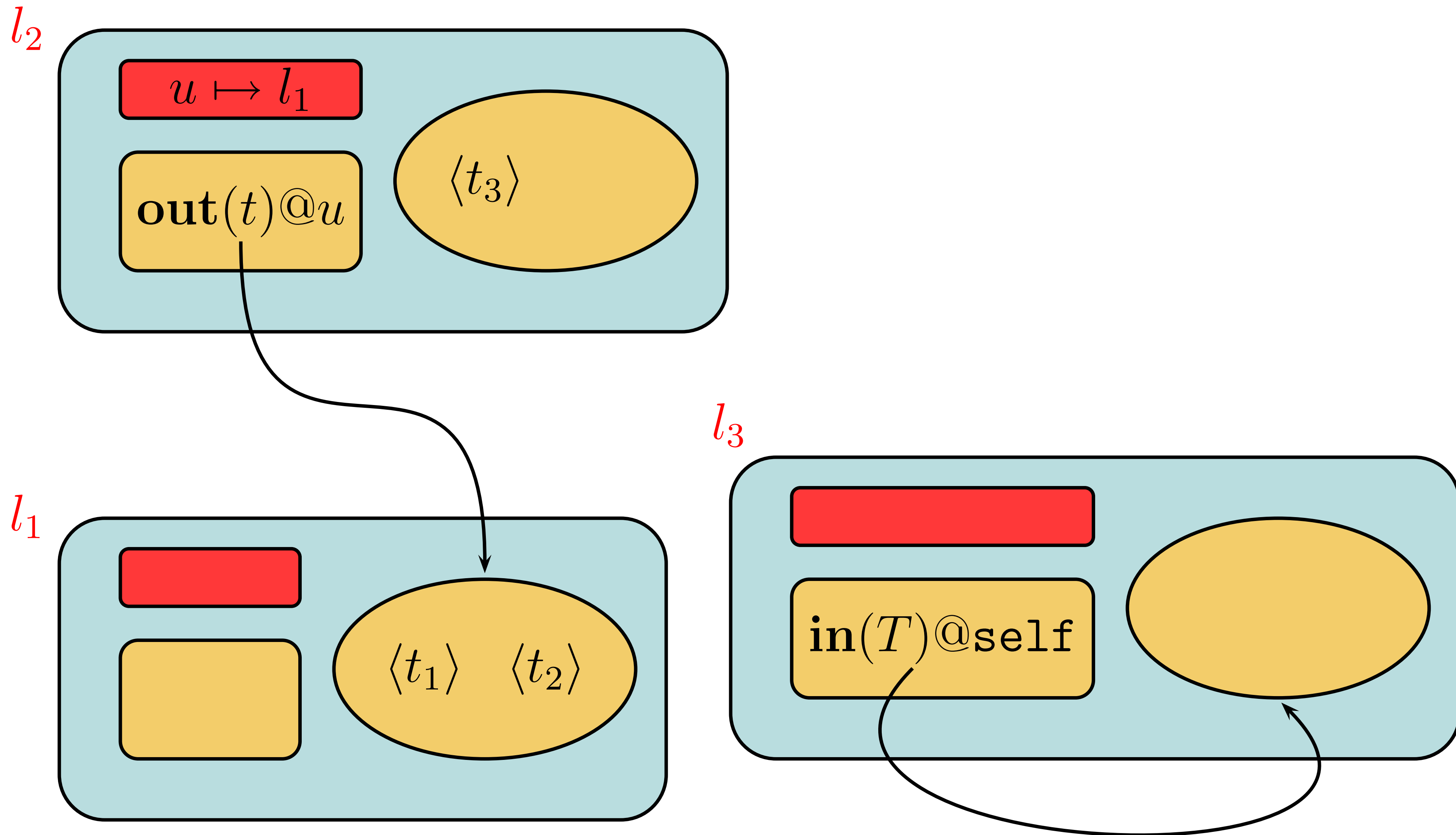
KLAIM Nodes

- Locality name (address)
- Allocation Environment
- Processes
- Tuple Space



$\mathbf{new}(u)$

KLAIM **Nets**



KLAIM: Syntax (1)

locality names: $l, l', \ldots$ *variables*: $\mathtt{self}, x, y, \ldots, X, Y, \ldots$

u will stand for a locality name or a locality variable

Klaim: Syntax (1)

locality names: $l, l', \ldots$ *variables*: $\texttt{self}, x, y, \ldots, X, Y, \ldots$

u will stand for a locality name or a locality variable

Tuples:

$$t \quad ::= \quad u \;\big|\; P \;\big|\; t_1, t_2$$

Templates:

$$T \quad ::= \quad u \;\big|\; !\,x \;\big|\; !\,X \;\big|\; T_1, T_2$$

KLAIM: Syntax (1)

locality names: $l, l', \ldots$ *variables*: $\texttt{self}, x, y, \ldots, X, Y, \ldots$

u will stand for a locality name or a locality variable

Tuples:

$$t \quad ::= \quad u \;\mid\; P \;\mid\; t_1, t_2$$

Templates:

$$T \quad ::= \quad u \;\mid\; !\,x \;\mid\; !\,X \;\mid\; T_1, T_2$$

Actions: exchanging data, spawning new processes, creating new nodes

$$a \;::=\; \mathbf{in}(T)@u \;\mid\; \mathbf{read}(T)@u \;\mid\; \mathbf{out}(t)@u \;\mid\; \mathbf{eval}(P)@u \;\mid\; \mathbf{new}(l)$$

KLAIM: Syntax (2)

Nets are plain collections of interconnected nodes. Nodes have a unique address, an allocation environment and host components.

$$N \quad ::= \quad \mathbf{0} \quad \big| \quad l ::_{\rho} C \quad \big| \quad N_1 \parallel N_2 \quad \big| \quad (\nu l)N$$

KLAIM: Syntax (2)

Nets are plain collections of interconnected nodes. Nodes have a unique address, an allocation environment and host components.

$$N \quad ::= \quad \mathbf{0} \quad | \quad l ::_{\rho} C \quad | \quad N_1 \parallel N_2 \quad | \quad (\nu l)N$$

Components are evaluated tuples or running processes.

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

KLAIM: Syntax (2)

Nets are plain collections of interconnected nodes. Nodes have a unique address, an allocation environment and host components.

$$N \quad ::= \quad \mathbf{0} \quad \big| \quad l ::_\rho C \quad \big| \quad N_1 \parallel N_2 \quad \big| \quad (\nu l)N$$

Components are evaluated tuples or running processes.

$$C \quad ::= \quad \langle t \rangle \quad \big| \quad P \quad \big| \quad C_1 \mid C_2$$

Processes are built up from the inert process via action prefixing, parallel composition and recursion.

$$P \quad ::= \quad \mathbf{nil} \quad \big| \quad a.P \quad \big| \quad P_1 \mid P_2 \quad \big| \quad X \quad \big| \quad \mathbf{rec}\, X.P$$

KLAIM: Syntax

Nets:

$$N \quad ::= \quad \mathbf{0} \quad | \quad l ::_{\rho} C \quad | \quad N_1 \parallel N_2 \quad | \quad (\nu l)N$$

Components:

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

Processes:

$$P \quad ::= \quad \mathbf{nil} \quad | \quad a.P \quad | \quad P_1 \mid P_2 \quad | \quad X \quad | \quad \mathbf{rec}\, X.P$$

Actions:

$$a \; ::= \; \mathbf{in}(T)@u \quad | \quad \mathbf{read}(T)@u \quad | \quad \mathbf{out}(t)@u \quad | \quad \mathbf{eval}(P)@u \quad | \quad \mathbf{new}(l)$$

Tuples: $t \quad ::= \quad u \quad | \quad P \quad | \quad t_1, t_2$

Templates: $T \quad ::= \quad u \quad | \quad !\,x \quad | \quad !\,X \quad | \quad T_1, T_2$

Binders

- Prefix $\mathbf{in}(\ldots, !x\ , \ldots, !X\ , \ldots)@u.P$ binds x , X in P (**read** acts similarly)
- Prefix $\mathbf{new}(l).P$ binds l in P
- Restriction $(\nu l)N$ binds l in N
- Recursion $\mathbf{rec}\, X.P$ binds X in P

KLAIM: Structural Congruence

Monoid laws for "$\|$", i.e.

$$N \parallel \mathbf{0} \equiv N \;,\; N_1 \parallel N_2 \equiv N_2 \parallel N_1 \;,\; (N_1 \parallel N_2) \parallel N_3 \equiv N_1 \parallel (N_2 \parallel N_3)$$

(ALPHA)	N	$\equiv$	N'	**if** $N =_\alpha N'$
(EXT)	$N_1 \parallel (\nu l)N_2$	$\equiv$	$(\nu l)(N_1 \parallel N_2)$	**if** $l \notin fn(N_1)$
(ABS)	$l ::_\rho C$	$\equiv$	$l ::_\rho (C \,\vert\, \mathbf{nil})$	
(CLONE)	$l ::_\rho C_1 \vert C_2$	$\equiv$	$l ::_\rho C_1 \parallel l ::_\rho C_2$	
(REC)	$l ::_\rho \mathbf{rec}\, X.P$	$\equiv$	$l ::_\rho P[^{\mathbf{rec}\, X.P}/_X]$	

KLAIM: Operational Semantics (1)

Producing data:

$$\frac{\rho(u) = l' \qquad \mathcal{E}[\![\, t \,]\!]_\rho = t'}{l ::_\rho \mathbf{out}(t)@u.P \;\|\; l' ::_{\rho'} \mathbf{nil} \longmapsto l ::_\rho P \,\|\, l' ::_{\rho'} \langle t' \rangle}$$

where

- the local environment ρ is used to determine the location (l') that is the target of the action
- function $\mathcal{E}[\![\, t \,]\!]\rho$ replaces the free locality variables occurring in t according to the *local* environment ρ yielding t'

  $\longrightarrow$ *static binding discipline*

Remark:
Free variables act as *aliases* that have to be replaced by *network addresses*

KLAIM: Operational Semantics (2)

Spawning processes:

$$\frac{\rho(u) = l'}{l ::_\rho \mathbf{eval}(Q)@u.P \;\|\; l' ::_{\rho'} \mathbf{nil} \longmapsto l ::_\rho P \,\|\, l' ::_{\rho'} Q}$$

No interpretation of **eval**'s argument is performed: free variables in Q will be translated according to the *remote* allocation environment

$\rightarrow$ *dynamic binding discipline*

KLAIM: Operational Semantics (3)

Consuming data:

$$\frac{\rho(u) = l' \qquad match(\mathcal{E}[\![\, T \,]\!]_\rho, t) = \sigma}{l ::_\rho \mathbf{in}(T)@u.P \;\|\; l' ::_{\rho'} \langle t \rangle \longmapsto l ::_\rho P\sigma \;\|\; l' ::_{\rho'} \mathbf{nil}}$$

where:

- function $match(T, t)$ checks compliance of t and $\mathcal{E}[\![\, T \,]\!]_\rho$ and associates values (i.e. names and processes) to variables bound by the template
- a tuple matches against a template if they have the same number of fields, and corresponding fields match
- σ is a substitution of names and processes for the free variables in T

KLAIM: Operational Semantics (4)

Accessing data:

$$\frac{\rho(u) = l' \qquad match(\mathcal{E}[\![\, T \,]\!]_\rho, t) = \sigma}{l ::_\rho \mathbf{read}(T)@u.P \;\|\; l' ::_{\rho'} \langle t \rangle \longmapsto l ::_\rho P\sigma \;\|\; l' ::_{\rho'} \langle t \rangle}$$

read is similar to **in**, but the accessed tuple is left in the tuple space

KLAIM: Operational Semantics (5)

Creating new nodes:

$$l ::_{\rho} \mathbf{new}(l').P \longmapsto (\nu l')(l ::_{\rho} P \parallel l' ::_{\rho[l'/\mathtt{self}]} \mathbf{nil})$$

where the environment of the creator is used to build up the environment for the new node (obviously, other choices are possible)

KLAIM: Operational Semantics (6)

Remaining rules:

$$\frac{N_1 \longmapsto N_1'}{N_1 \parallel N_2 \longmapsto N_1' \parallel N_2}$$

$$\frac{N \longmapsto N'}{(\nu l)N \longmapsto (\nu l)N'}$$

$$\frac{N \equiv M \longmapsto M' \equiv N'}{N \longmapsto N'}$$

μKLAIM: micro KLAIM

We strip off KLAIM:

1. higher-order data (i.e. tuples containing process code)
2. allocation environments (i.e. the distinction between logical and physical localities)

μKlaim: micro Klaim

We strip off Klaim:

1. higher-order data (i.e. tuples containing process code)
2. allocation environments (i.e. the distinction between logical and physical localities)

μKlaim: largest sub-calculus of Klaim where tuples do not contain processes, templates do not contain process variables, allocation environments are empty and processes do not contain free variables

μKLAIM: Syntax

$$
\begin{array}{lcl}
N & ::= & \mathbf{0} \mid l :: C \mid N_1 \parallel N_2 \mid (\nu l)N \\
C & ::= & \langle t \rangle \mid P \mid C_1 \mid C_2 \\
P & ::= & \mathbf{nil} \mid a.P \mid P_1 \mid P_2 \mid X \mid \mathbf{rec}\, X.P \\
a & ::= & \mathbf{in}(T)@u \mid \mathbf{read}(T)@u \mid \mathbf{out}(t)@u \mid \mathbf{eval}(P)@u \mid \mathbf{new}(l) \\
t & ::= & u \mid t_1, t_2 \\
T & ::= & u \mid !\,x \mid T_1, T_2
\end{array}
$$

$\mathcal{N}$ set of *names* ranged over by $l, l', \ldots, u, \ldots, x, y, \ldots, X, Y, \ldots$

μKLAIM: Operational Semantics

$$l :: \mathbf{out}(t)@l'.P \;\parallel\; l' :: \mathbf{nil} \;\longmapsto\; l :: P \parallel l' :: \langle t \rangle$$

$$l :: \mathbf{eval}(P_2)@l'.P_1 \;\parallel\; l' :: \mathbf{nil} \;\longmapsto\; l :: P_1 \parallel l' :: P_2$$

$$\frac{match(T,t) = \sigma}{l :: \mathbf{in}(T)@l'.P \;\parallel\; l' :: \langle t \rangle \;\longmapsto\; l :: P\sigma \parallel l' :: \mathbf{nil}}$$

$$\frac{match(T,t) = \sigma}{l :: \mathbf{read}(T)@l'.P \;\parallel\; l' :: \langle t \rangle \;\longmapsto\; l :: P\sigma \parallel l' :: \langle t \rangle}$$

$$l :: \mathbf{new}(l').P \;\longmapsto\; (\nu l')(l :: P \parallel l' :: \mathbf{nil})$$

Encoding Higher Order Messages in Pi-Calculus

Higher-order data can be handled by means of *triggers* like in [Sang:BisimHO-Pi:I & C,1996].

In π-calculus

$$\bar{a}\langle P\rangle \quad | \quad a(X).X$$

is translated to

$$(\nu c)(\bar{a}\langle c\rangle \mid *\ c().P') \quad | \quad a(x).\bar{x}\langle\rangle$$

where P' is the translation of P and $*$ denotes replication

The address of P is delivered and can be used by the interested processes to activate as many copies of P as needed

Encoding KLAIM in μKLAIM (1)

In KLAIM

$$l_1 :: \mathbf{out}(P)@l \;\parallel\; l_2 :: \mathbf{in}(!X)@l.X$$

is translated to

$$\begin{array}{l} \quad l_1 :: \mathbf{new}(l').\mathbf{eval}\Big(\mathbf{rec}\, X.\, \mathbf{in}(!y)@l'.\mathbf{eval}(P)@y.X\Big)@l'.\mathbf{out}(l')@l \\ \parallel\; l_2 :: \mathbf{in}(!x)@l.\mathbf{out}(l_2)@x \end{array}$$

that evolves to (a net “equivalent” to)

$$(\nu l')(l_1 :: \mathbf{nil} \;\parallel\; l_2 :: P)$$

A fresh node is used to deposit the process we want migrate (P),
its address is then delivered to be used by the interested processes
to activate as many copies of P as needed

Encoding KLAIM in μKLAIM (2)

Binding of free locality variables to names can be modelled by:

- storing the allocation environment of each node in the tuple space of a fresh node `env`
- prefixing each localized action with the "capture" of the free variables occurring in its argument and in its target by exploiting `env`

Encoding KLAIM in μKLAIM (2)

Binding of free locality variables to names can be modelled by:

- storing the allocation environment of each node in the tuple space of a fresh node `env`
- prefixing each localized action with the "capture" of the free variables occurring in its argument and in its target by exploiting `env`

Thus,

$$l ::_\rho \mathbf{out}(u_1, l, u_2)@l'.P$$

where ρ maps u_1 to l_1 and u_2 to l_2,

is translated to

$$\begin{array}{ll} & \mathtt{env} :: \langle l, u_1, l_1 \rangle \mid \langle l, u_2, l_2 \rangle \\ \| & l :: \mathbf{read}(l, u_1, !y_1)@\mathtt{env}.\mathbf{read}(l, u_2, !y_2)@\mathtt{env}.\mathbf{out}(y_1, l, y_2)@l'.P' \end{array}$$

where P' is the translation of P, and y_1 and y_2 are fresh names

cKlaim: core Klaim

We strip off μKlaim:

1. actions **read** for accessing data without removing them from tuple spaces

2. *polyadic* tuples, i.e. all tuples in cKlaim have just one field

cKLAIM: Syntax

$$
\begin{array}{rcl}
N & ::= & \mathbf{0} \quad | \quad l :: C \quad | \quad N_1 \parallel N_2 \quad | \quad (\nu l)N \\
C & ::= & \langle t \rangle \quad | \quad P \quad | \quad C_1 \mid C_2 \\
P & ::= & \mathbf{nil} \quad | \quad a.P \quad | \quad P_1 \mid P_2 \quad | \quad X \quad | \quad \mathbf{rec}\, X.P \\
a & ::= & \mathbf{in}(T)@u \quad | \quad \mathbf{out}(t)@u \quad | \quad \mathbf{eval}(P)@u \quad | \quad \mathbf{new}(l) \\
t & ::= & u \\
T & ::= & u \quad | \quad !\, x
\end{array}
$$

cKlaim: Syntax

$$N ::= \mathbf{0} \;\mid\; l :: C \;\mid\; N_1 \parallel N_2 \;\mid\; (\nu l)N$$

$$C ::= \langle t \rangle \;\mid\; P \;\mid\; C_1 \mid C_2$$

$$P ::= \mathbf{nil} \;\mid\; a.P \;\mid\; P_1 \mid P_2 \;\mid\; X \;\mid\; \mathbf{rec}\, X.P$$

$$a ::= \mathbf{in}(T)@u \;\mid\; \mathbf{out}(t)@u \;\mid\; \mathbf{eval}(P)@u \;\mid\; \mathbf{new}(l)$$

$$t ::= u$$

$$T ::= u \;\mid\; !\,x$$

cKlaim is a sub-calculus of μKlaim and inherits its operational semantics

Encoding μKLAIM in cKLAIM (1)

Actions **read** can be naturally encoded by performing action **in** followed by action **out** on the tuple that has been accessed

For example,

$$\mathbf{read}(!x, l', !y)@l$$

is translated to

$$\mathbf{in}(!x, l', !y)@l.\mathbf{out}(x, l', y)@l$$

Encoding μKLAIM in cKLAIM (1)

Actions **read** can be naturally encoded by performing action **in** followed by action **out** on the tuple that has been accessed

For example,

$$\mathbf{read}(!x, l', !y)@l$$

is translated to

$$\mathbf{in}(!x, l', !y)@l.\mathbf{out}(x, l', y)@l$$

Expressivity is not compromised, because of asynchrony. This is somehow related to the equation

$$a(x).\bar{a}\langle x\rangle \quad \approx \quad \mathbf{0}$$

established for the π_{a}-calculus

(see [AmadioCastellaniSangiorgi:BisimAsynchPi:TCS,1998])

Polyadic vs. Monadic π-calculus (1)

Synchronous π-calculus

With Milner's encoding for the *synchronous* π-calculus we have that:

$$\bar{a}\langle b, c\rangle \quad | \quad a(x, y)$$

becomes

$$(\nu n)(\bar{a}\langle n\rangle.\bar{n}\langle b\rangle.\bar{n}\langle c\rangle \quad | \quad a(z).z(x).z(y))$$

A fresh name (n) is exchanged by exploiting a common channel (a)
n is then used to pass the sequence of values

Polyadic vs. Monadic π-calculus (2)

Asynchronous π-calculus

A possible encoding for the *asynchronous* π-calculus (adapted from [HondaTokoro:ObjectCalcAsynchComm:ECOOP'91]):

$$\bar{a}\langle b, c\rangle \quad | \quad a(x, y)$$

is rendered as

$$(\nu n)\big(\bar{a}\langle n\rangle \quad | \quad n(n_1).(\ \bar{n}_1\langle b\rangle \ | \ n(n_2).\bar{n}_2\langle c\rangle\ )\big)$$

$$| \quad (\nu n_1, n_2)a(z).\big(\bar{z}\langle n_1\rangle \quad | \quad n_1(x).(\ \bar{z}\langle n_2\rangle \ | \ n_2(y)\ )\big)$$

The schema is similar to the one for the synchronous calculus
Since output sequentialization is not possible, different channels are needed to send the different values in the sequence

Encoding μKLAIM in cKLAIM (2)

In our setting, pattern-matching of values performed while retrieving tuples complicates the encoding

The key ideas for the encoding are the following:

- A polyadic tuple is implemented by a process that sequentially produces the fields of the tuple
- The receiving process accesses these (monadic) fields in an exclusive and ordered way
- If the i-th tuple field matches against the i-th template field, the retrieving procedure goes on; otherwise, it is stopped and the involved processes are rolled back

Encoding μKLAIM in cKLAIM (2)

We translate the interaction between the consuming action $\mathbf{in}(T)@l$ and a tuple t by exploiting the following protocol:

TUPLE CONSUMER		TUPLE HANDLER
Acquire the **lock** over a tuple		
Ask for tuple's **length**	$\longrightarrow$	
	$\twoheadleftarrow$	Provide t's length k
If $k = \|T\|$ proceed, otherwise release the lock and roll back the tuple handler		
Ask for tuple's **first field**	$\longrightarrow$	
	$\twoheadleftarrow$	Provide t's first field f_1
If T's first field matches f_1 proceed, otherwise release the lock and roll back the tuple handler		
...	...	...
Require tuple's **last field**	$\longrightarrow$	
	$\twoheadleftarrow$	Provide t's last field f_k
If T's last field matches f_k FINISH, otherwise release the lock and roll back the tuple handler		

l-cKlaim: local cKlaim

We strip off cKlaim remote inputs and outputs
(communications is only local and process migration is needed
to use remote resources)

l-cKlaim: local cKlaim

We strip off cKlaim remote inputs and outputs
(communications is only local and process migration is needed
to use remote resources)

Syntax:

$$a \quad ::= \quad \mathbf{in}(T) \;\;|\;\; \mathbf{out}(t) \;\;|\;\; \mathbf{eval}(P)@u \;\;|\;\; \mathbf{new}(l)$$

L-CKLAIM: local CKLAIM

We strip off CKLAIM remote inputs and outputs
(communications is only local and process migration is needed
to use remote resources)

Syntax:

$$a \quad ::= \quad \mathbf{in}(T) \;\Big|\; \mathbf{out}(t) \;\Big|\; \mathbf{eval}(P)@u \;\Big|\; \mathbf{new}(l)$$

Operational semantics: rules for the new (local) operations

$$l :: \mathbf{out}(l').P \longmapsto l :: P \,|\, \langle l' \rangle \qquad \frac{match(T, l') = \sigma}{l :: \mathbf{in}(T).P \,|\, \langle l' \rangle \longmapsto l :: P\sigma}$$

Encoding L-CKLAIM in CKLAIM

- action $\mathbf{out}(l)@l'.P$ is implemented as

$$\mathbf{eval}(\ \mathbf{out}(l)\ )@l'.P'$$

where P' is the encoding of P

Encoding L-CKLAIM in CKLAIM

- action $\mathbf{out}(l)@l'.P$ is implemented as

  $$\mathbf{eval}(\ \mathbf{out}(l)\ )@l'.P'$$

  where P' is the encoding of P

- action $\mathbf{in}(T)@l'.P$ at site l is implemented as

  $$\mathbf{eval}(\ \mathbf{in}(T).\mathbf{eval}(P')@l\ )@l'$$

  where P' is the encoding of P

Encoding the asynch. π-calculus into cKLAIM

Syntax:

$$p \quad ::= \quad \mathbf{0} \quad | \quad \bar{a}\langle b\rangle \quad | \quad a(b).p \quad | \quad p_1|p_2 \quad | \quad (\nu a)p \quad | \quad [a=b]p \quad | \quad !p$$

Encoding the asynch. π-calculus into cKlaim

Syntax:

$$p \;::=\; \mathbf{0} \;\mid\; \bar{a}\langle b\rangle \;\mid\; a(b).p \;\mid\; p_1|p_2 \;\mid\; (\nu a)p \;\mid\; [a=b]p \;\mid\; !p$$

Encoding π_{a}-calculus in cKlaim:

- each channel a is translated in the TS of a node named a
- the basic constructs are translated as follows:

$$\begin{aligned}
[\![(\nu a)p]\!] &= \mathbf{new}(a).[\![p]\!] \\
[\![\bar{a}\langle b\rangle]\!] &= \mathbf{out}(b)@a.\mathbf{nil} \\
[\![a(b).p]\!] &= \mathbf{in}(!b)@a.[\![p]\!] \\
[\![[a=b]p]\!] &= \mathbf{new}(l).\mathbf{out}(a)@l.\mathbf{in}(b)@l.[\![p]\!]
\end{aligned}$$

Encoding L-CKLAIM into the asynch. π-calculus

Encoding Nets:

$$(\![\mathbf{0}]\!) \triangleq\ !\,\overline{\mathtt{ex}}\langle\rangle \qquad (\![N_1 \parallel N_2]\!) \triangleq (\![N_1]\!) \mid (\![N_2]\!)$$

$$(\![(\nu l)N]\!) \triangleq (\nu l)((\![N]\!) \mid !\,\overline{\mathtt{ex}}\,l) \qquad (\![l :: C]\!) \triangleq (\![C]\!)_l \mid !\,\overline{\mathtt{ex}}\,l$$

Encoding Processes:

$$(\![\mathbf{nil}]\!)_u \triangleq \mathbf{0} \qquad (\![\langle l\rangle]\!)_u \triangleq \bar{u}\,l$$

$$(\![X]\!)_u \triangleq X \qquad (\![\mathbf{rec}\,X.P]\!)_u \triangleq \mathbf{rec}\,X.(\![P]\!)_u$$

$$(\![C_1|C_2]\!)_u \triangleq (\![C_1]\!)_u \mid (\![C_2]\!)_u \qquad (\![\mathbf{new}(l).P]\!)_u \triangleq (\nu l)((\![P]\!)_u \mid !\,\overline{\mathtt{ex}}\,l)$$

$$(\![\mathbf{out}(u').P]\!)_u \triangleq \bar{u}\,u' \mid (\![P]\!)_u \qquad (\![\mathbf{in}(!x).P]\!)_u \triangleq u(x).(\![P]\!)_u$$

$$(\![\mathbf{in}(u').P]\!)_u \triangleq \mathbf{rec}\,X.u(x).(\nu c)\Big(\ \bar{c} \mid [x = u']c.(\![P]\!)_u \mid c.(\bar{u}\,x \mid X)\Big)$$

$$(\![\mathbf{eval}(Q)@u'.P]\!)_u \triangleq \mathbf{rec}\,X.\mathtt{ex}(x).(\nu c)\Big(\ \bar{c} \mid [x = u']c.((\![P]\!)_u \mid (\![Q]\!)_{u'}) \mid c.X\ \Big)$$

x, X fresh in the last two cases

Encoding L-CKLAIM into the asynch. π-calculus

Locality names are rendered as channels

To take names that are addresses of network nodes and raw names apart, the encoding uses a reserved channel ex to record the formers

Process distribution: the encoding for processes, $(\![P]\!)_u$, is parameterized with the locality u where P is on

Name matching construct (akin the encoding of μKLAIM in CKLAIM): process $\mathbf{in}(l').P$ running at l is rendered as a process that first retrieves a datum at l and then checks if it is l'; if the check succeeds, the process goes on, otherwise it places back the accessed datum and looks for another one

Assessing the Encodings

Two well-established criteria:

Full Abstraction w.r.t. EQ**:**

$X_1 \; EQ_{\mathcal{X}} \; X_2$ if and only if $enc(X_1) \; EQ_{\mathcal{Y}} \; enc(X_2)$

Semantical Equivalence w.r.t. EQ**:**

$X \; EQ_{\mathcal{Z}} \; enc(X)$, for some language $\mathcal{Z}$ containing both $\mathcal{X}$ and $\mathcal{Y}$

Assessing the Encodings

Two well-established criteria:

Full Abstraction w.r.t. EQ:

$X_1 \; EQ_{\mathcal{X}} \; X_2$ if and only if $enc(X_1) \; EQ_{\mathcal{Y}} \; enc(X_2)$

Semantical Equivalence w.r.t. EQ:

$X \; EQ_{\mathcal{Z}} \; enc(X)$, for some language $\mathcal{Z}$ containing both $\mathcal{X}$ and $\mathcal{Y}$

N.B.:

- EQ is not a precise equivalence but a *family* of equivalences
  $EQ_{\mathcal{L}}$ is the equivalence EQ in the language $\mathcal{L}$
- A stronger equivalence guarantees a better encoding, in that it attests that the target calculus has expressive power closer to that of the source calculus
- Semantical equivalence w.r.t. EQ implies full abstraction w.r.t. EQ

A uniform notion of equivalence

Observation predicate, or *barb*:

$$N \downarrow l \text{ iff } N \equiv (\nu\widetilde{l})(N' \parallel l ::_\rho \langle t \rangle) \text{ for some } \widetilde{l}, N', \rho \text{ and } t \text{ s.t. } l \notin \widetilde{l}$$

A uniform notion of equivalence

Observation predicate, or *barb*:

$$N \downarrow l \text{ iff } N \equiv (\nu\widetilde{l})(N' \parallel l ::_\rho \langle t \rangle) \text{ for some } \widetilde{l}, N', \rho \text{ and } t \text{ s.t. } l \notin \widetilde{l}$$

Contexts: $\mathcal{C}[\cdot] \;::=\; [\cdot] \;\Big|\; N \parallel \mathcal{C}[\cdot] \;\Big|\; (\nu l)\mathcal{C}[\cdot]$

A uniform notion of equivalence

Observation predicate, or *barb*:

$$N \downarrow l \text{ iff } N \equiv (\nu\widetilde{l})(N' \parallel l ::_\rho \langle t \rangle) \text{ for some } \widetilde{l}, N', \rho \text{ and } t \text{ s.t. } l \notin \widetilde{l}$$

Contexts: $\mathcal{C}[\cdot] \ ::= \ [\cdot] \ \Big| \ N \parallel \mathcal{C}[\cdot] \ \Big| \ (\nu l)\mathcal{C}[\cdot]$

A binary relation $\Re$ between nets is

- *barb preserving*, if $N \Re M$ and $N \downarrow l$ imply $M \Downarrow l$
- *reduction closed*, if $N \Re M$ and $N \longmapsto N'$ imply $M \longmapsto^* M'$ and $N' \Re M'$
- *F-context closed*, if $N \Re M$ implies $\mathcal{C}[N] \Re \mathcal{C}[M]$ for every context $\mathcal{C}[\cdot]$ of F

A uniform notion of equivalence

Observation predicate, or *barb*:

$$N \downarrow l \text{ iff } N \equiv (\nu \widetilde{l})(N' \parallel l ::_\rho \langle t \rangle) \text{ for some } \widetilde{l}, N', \rho \text{ and } t \text{ s.t. } l \notin \widetilde{l}$$

Contexts: $\mathcal{C}[\cdot] \ ::= \ [\cdot] \ \Big| \ N \parallel \mathcal{C}[\cdot] \ \Big| \ (\nu l)\mathcal{C}[\cdot]$

A binary relation $\Re$ between nets is

- *barb preserving*, if $N \Re M$ and $N \downarrow l$ imply $M \Downarrow l$
- *reduction closed*, if $N \Re M$ and $N \longmapsto N'$ imply $M \longmapsto^* M'$ and $N' \Re M'$
- *F-context closed*, if $N \Re M$ implies $\mathcal{C}[N] \Re \mathcal{C}[M]$ for every context $\mathcal{C}[\cdot]$ of F

Barbed bisimilarity, $\dot{\cong}$, is the largest symmetric, barb preserving and reduction closed relation between nets

A uniform notion of equivalence

Observation predicate, or *barb*:

$$N \downarrow l \text{ iff } N \equiv (\nu\widetilde{l})(N' \parallel l ::_\rho \langle t \rangle) \text{ for some } \widetilde{l}, N', \rho \text{ and } t \text{ s.t. } l \notin \widetilde{l}$$

Contexts: $\mathcal{C}[\cdot] \; ::= \; [\cdot] \;\Big|\; N \parallel \mathcal{C}[\cdot] \;\Big|\; (\nu l)\mathcal{C}[\cdot]$

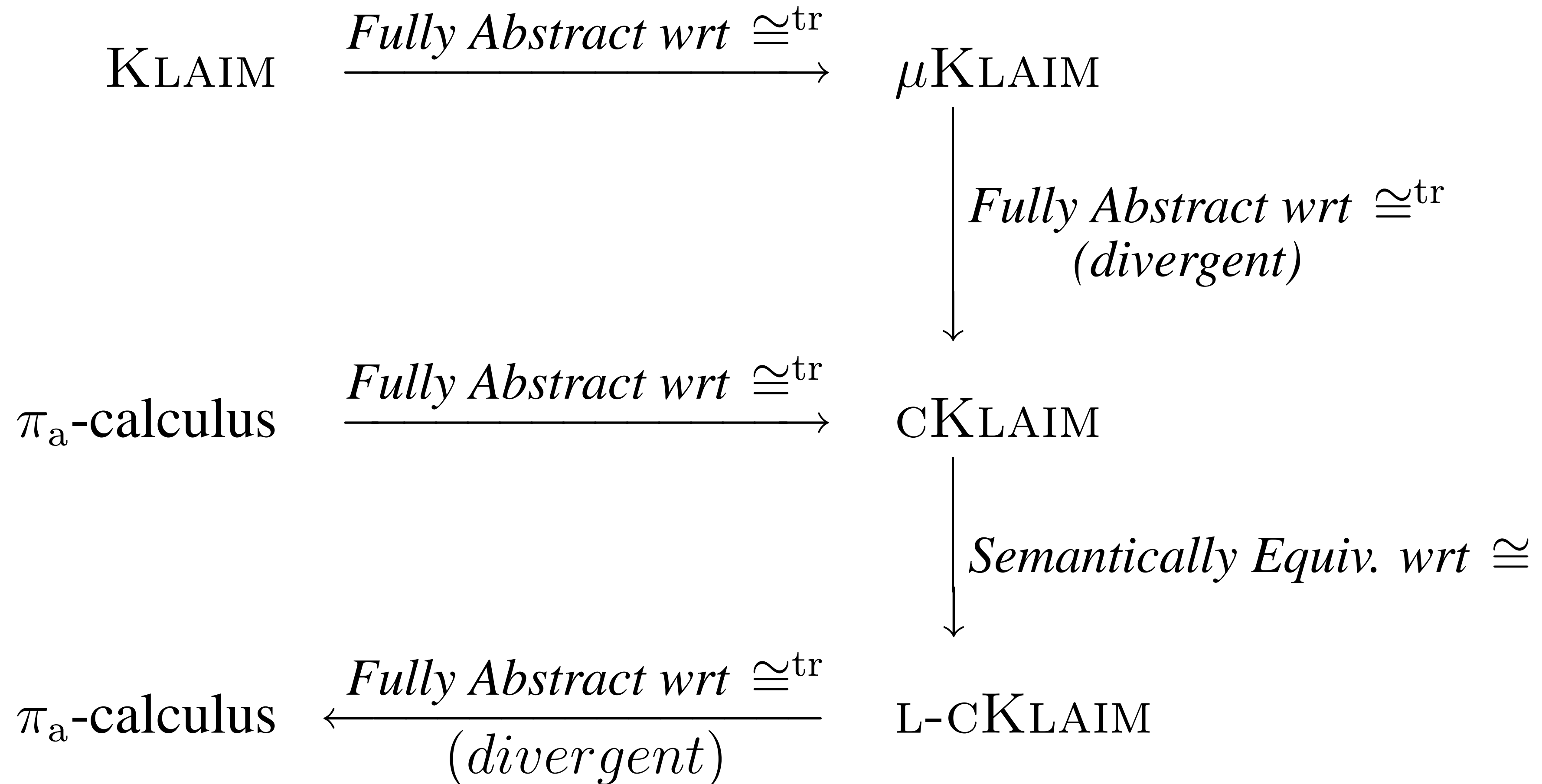
A binary relation $\Re$ between nets is

- *barb preserving*, if $N \,\Re\, M$ and $N \downarrow l$ imply $M \Downarrow l$
- *reduction closed*, if $N \,\Re\, M$ and $N \longmapsto N'$ imply $M \longmapsto^* M'$ and $N' \,\Re\, M'$
- *F-context closed*, if $N \,\Re\, M$ implies $\mathcal{C}[N] \,\Re\, \mathcal{C}[M]$ for every context $\mathcal{C}[\cdot]$ of F

Barbed bisimilarity, $\dot{\cong}$, is the largest symmetric, barb preserving and reduction closed relation between nets

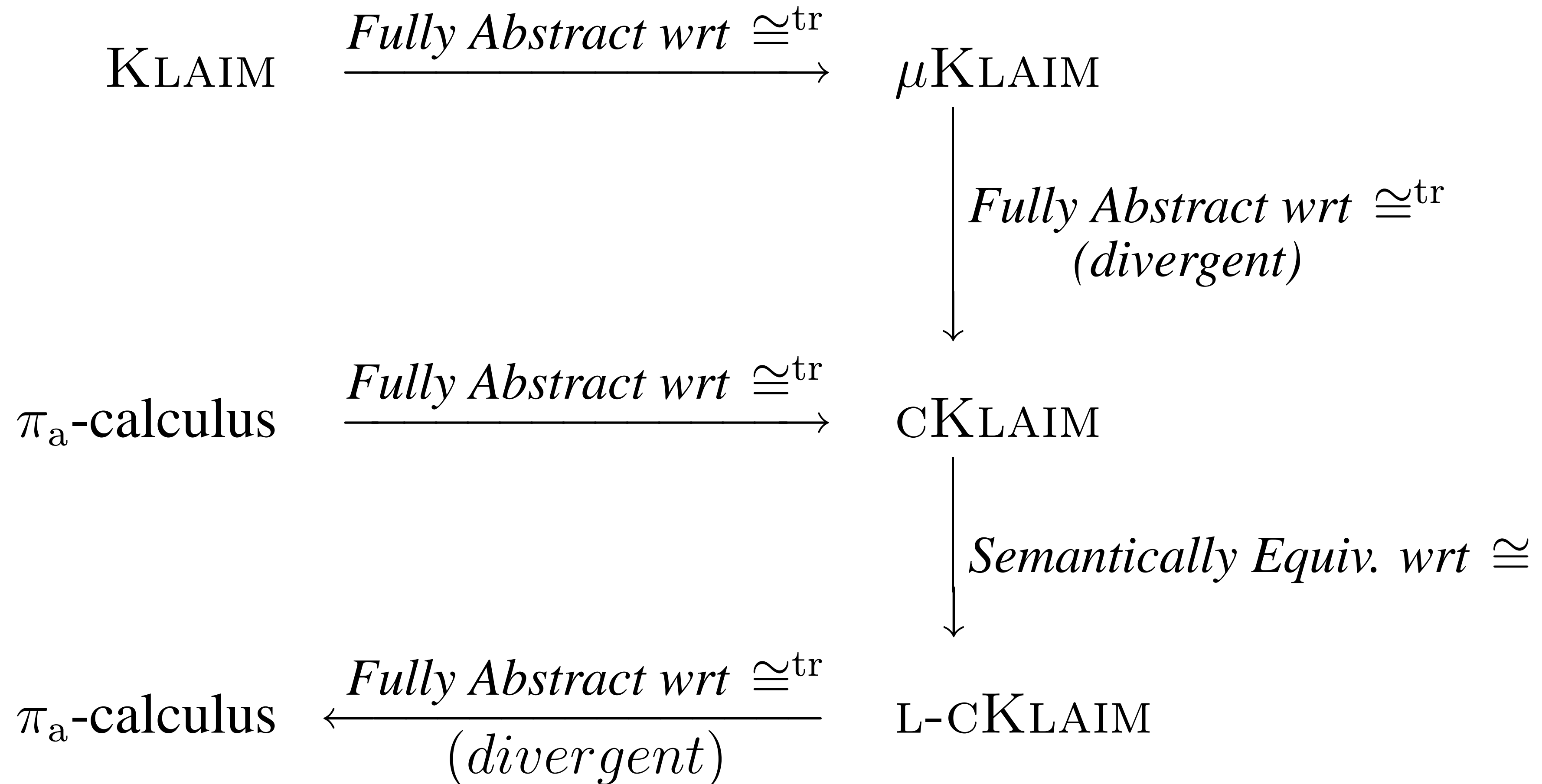
Barbed congruence, $\cong$, is the largest symmetric, barb preserving, reduction and context closed relation between nets

Properties of the Encodings



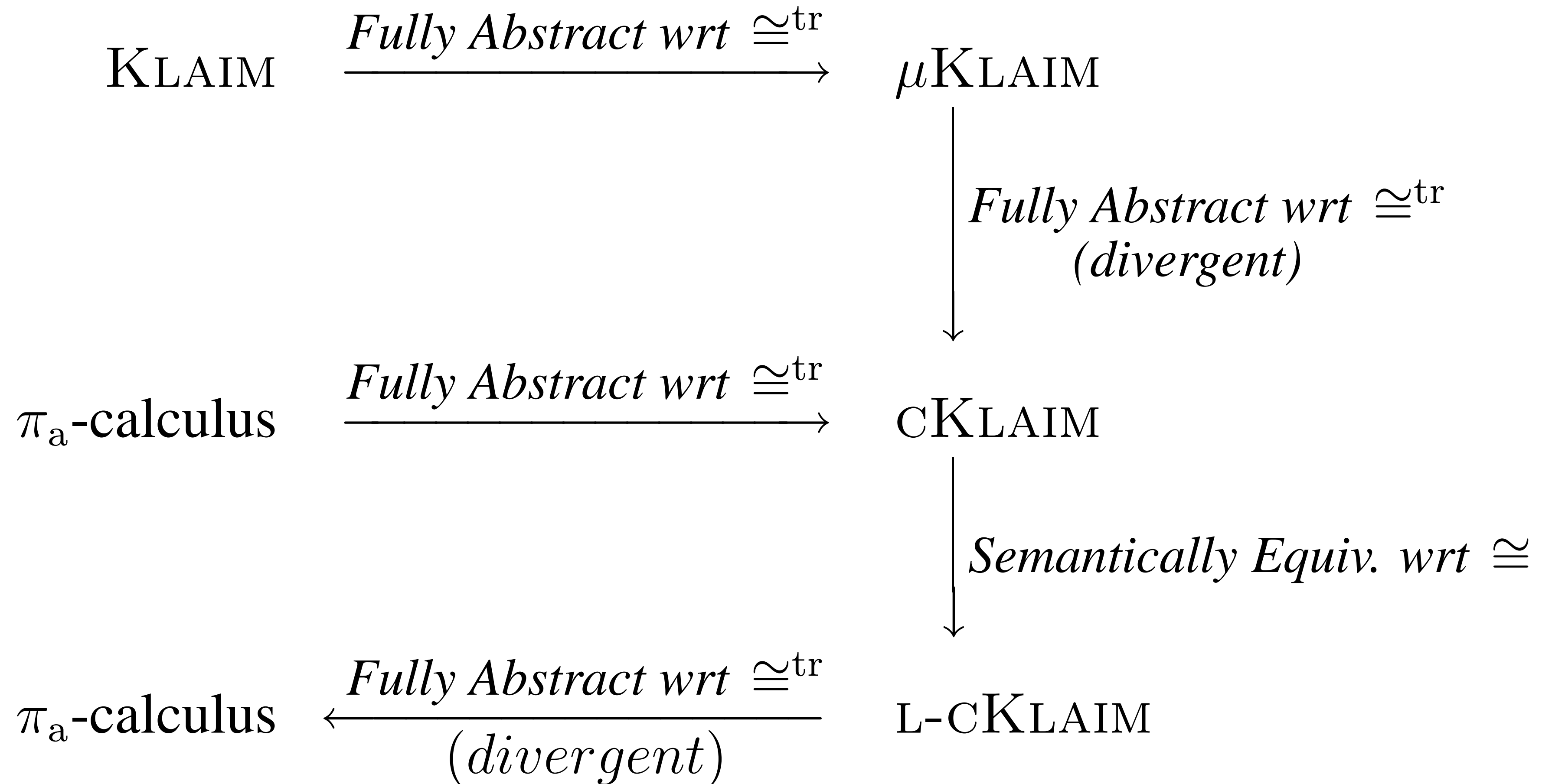
A labelled arrow between two calculi, $\mathcal{X} \xrightarrow{\mathcal{P}} \mathcal{Y}$, means that $\mathcal{X}$ can be encoded in $\mathcal{Y}$ and the encoding enjoys property $\mathcal{P}$

Properties of the Encodings



- $\cong$ is barbed congruence (i.e. $\dot{\cong}$ closed for all possible contexts)
- $\cong^{tr}$ is $\dot{\cong}$ closed for all contexts resulting from the translation (via the encoding) of source contexts

Properties of the Encodings



If we consider the encoding as a protocol (i.e. a precise sequence of message exchanges), translated contexts represent opponents conform to the protocol

The source calculus can be faithfully compiled in the target one

Comments

- The encoding of cKLAIM in L-cKLAIM is the best we can imagine:
  it does not introduce divergence and translates nets into barb.congr. ones
  The two calculi have exactly the same expressive power
  Remote communications are just a mean to simplify programming

Comments

- The encoding of cKLAIM in l-cKLAIM is the best we can imagine:
  it does not introduce divergence and translates nets into barb.congr. ones
  The two calculi have exactly the same expressive power
  Remote communications are just a mean to simplify programming
- The encodings of KLAIM in μKLAIM and of π_{a}-calculus in cKLAIM are satisfactory: the generated code is quite simple and the encodings enjoy the four properties pointed out in [Palamidessi:MSCS2003]
  Source and target calculi have similar expressive power

Comments

- The encoding of cKlaim in l-cKlaim is the best we can imagine:
  it does not introduce divergence and translates nets into barb.congr. ones
  The two calculi have exactly the same expressive power
  Remote communications are just a mean to simplify programming
- The encodings of Klaim in μKlaim and of π_a-calculus in cKlaim are satisfactory: the generated code is quite simple and the encodings enjoy the four properties pointed out in [Palamidessi:MSCS2003]
  Source and target calculi have similar expressive power
- The encodings of μKlaim in cKlaim and of l-cKlaim in π_a-calculus are less satisfactory because they may introduce divergence
  The encoding of polyadic communication in monadic communication is neither simple nor efficient: μKlaim is more expressive than cKlaim
  In a Linda-like framework, these two forms of communication are not interchangeable

π_a-calculus vs cKLAIM

A key design issue breaks full abstraction:

- In π_a-calculus, knowing a name implies that communication actions can be performed upon a channel with that name and these actions succeed whenever a parallel component performs a complementary action
- This is not the case in cKLAIM: it is *not* necessarily the case that each free name is associated to a locality

Claim: No 'reasonable' encoding of π_a-calculus in cKLAIM can be given: checking existence of nodes before firing an output is a too low-level feature that cannot be implemented in such an abstract setting as the π-calculus

To recover full abstraction we can e.g.

- make cKLAIM higher-level by adding struct. rule: $l :: \mathbf{nil} \equiv \mathbf{0}$ each name is *always* associated to a communication medium

Criteria for an encoding to be reasonable

According to [Palamidessi:MSCS2003], any 'reasonable' encoding $enc(\cdot)$ should guarantee the same the degree of parallelism, a close correspondence between the names of the used channels and the same semantics.

1. *be homomorphic w.r.t. the parallel operator*, i.e. $enc(N \parallel M) = enc(N) \parallel enc(M)$
2. *preserve renaming*, i.e. for every permutation of names σ in the source language there exists a permutation of names θ in the target language such that $enc(P\sigma) = (enc(P))\theta$
3. *preserve the basic observables*, i.e. it must preserve the visible behaviours of the encoded terms
4. *preserve termination*, i.e. it must turn each terminating term in a terminating term

Conclusions

We have presented:

- a family of process languages based on KLAIM
- a tight study of the their expressive power by encodings
- a comparison with the asynchronous π-calculus

Conclusions

We have presented:

- a family of process languages based on KLAIM
- a tight study of the their expressive power by encodings
- a comparison with the asynchronous π-calculus

Our results might be used to assess:

- dynamic naming disciplines vs. static ones
- languages based on LINDA
- GC languages with remote/local operations

Conclusions

We have presented:

- a family of process languages based on KLAIM
- a tight study of the their expressive power by encodings
- a comparison with the asynchronous π-calculus

Our results might be used to assess:

- dynamic naming disciplines vs. static ones
- languages based on LINDA
- GC languages with remote/local operations

To investigate more deeply the expressive power of pattern-matching:

- studying 'reasonable' encodings of calculi with communication based on pattern-matching into calculi with channel-based communications

References

- R. De Nicola, G. Ferrari, R. Pugliese. KLAIM*: a Kernel Language for Agents Interaction and Mobility*. **IEEE Trans. on Software Engineering**, 1998.
- D. Gorla and R. Pugliese. *A Semantic Theory for Global Computing Systems*. **Technical Report**, Univ. Firenze, 2004.
- R. De Nicola, D. Gorla, and R. Pugliese. *Basic observables for a calculus for global computing*. **Technical report** available at `http://www.dsi.uniroma1.it/~gorla/papers/bo4k-full.pdf`.
- R. De Nicola, D. Gorla and R. Pugliese. *On the Expressive Power of* KLAIM*-based Calculi*. **Proc. of Express 2004, ENTCS**.
  Full paper available at
  `http://www.dsi.uniroma1.it/~gorla/papers/expr4k-full.pdf`.

The End

- Thanks!
- Questions?