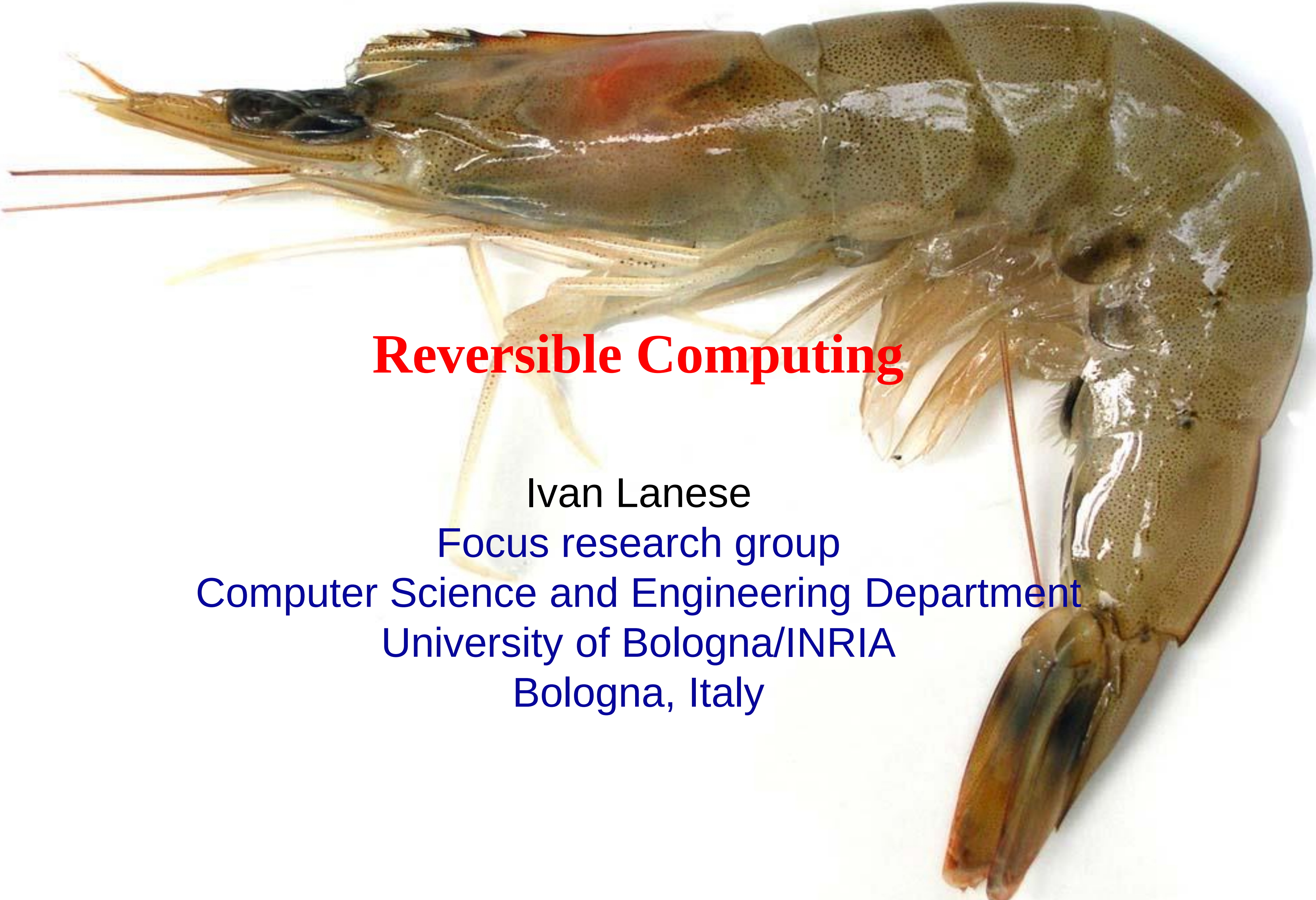


Reversible Computing

Ivan Lanese
Focus research group
Computer Science and Engineering Department
University of Bologna/INRIA
Bologna, Italy

Remember where we are

- Causal-consistent reversibility as the suitable way to do reversibility in a concurrent setting
- Uncontrolled reversibility as the simplest setting, but not very useful
- Different mechanisms allowing to control reversibility
- Which one do we choose?

Our approach

- The choice of the approach is based on the intended application field
- Our application field: programming reliable concurrent/distributed systems
- Normal computation should go forward
  - No backward computation without errors
- In case of error I should go back to a past state
  - We assume to be able to detect errors
- I should go to a state where the decision leading to the error has not been taken yet
  - The programmer should be able to find such a state

The kind of algorithm we want to write

- γ: take some choice
  ....
  if we reached a bad state
      roll γ
  else
      output the result
- The approach based on the **roll** operator is suitable to our aims
- Not necessarily the best in all the cases

A trade-off

- The approach based on **roll** tries to minimize the use of reversibility
  - Reversible computations only in case of error
  - The amount of computation to be undone is bound
  - Efficient strategy
- The programmer should find
  - The bad state
  - The decision leading to it
- Other approaches are less efficient, but rely less on the programmer skills
  - Irreversible actions only require to find the good state
  - Easier, but the approach is less efficient

Roll and loop

- With the **roll** approach
- We reach a bad state
- We go back to a past state
- We may choose again the same path
- We reach the same state again
- We go back again to the same past state
- We may choose again the same path
- …

Permanent and transient errors

- Going back to a past state forces us to forget everything we learned in the forward computation
  - We forget that a given path was not good
  - We may retry again and again the same path
- The approach is good for transient errors
  - Errors that may disappear by retrying
  - E.g., message loss on the Internet
- The approach is less suited for permanent errors
  - Errors that occur every time a state is reached
  - E.g., division by zero, null pointer exception
  - We can only hope to take a different branch in a choice

Non perfect reversibility

- In case of error I want to change path
  - Not possible in the current setting
  - The **roll** leads back to a past state
  - The same path will be available again
  - The programmer cannot avoid this
- I need to remember something from the past try
  - I should break the Loop Lemma
  - Reversibility should not be perfect

Alternatives

- I want to specify alternatives
- **Roll** causes the choice of a different alternative
- The programmer may declare alternatives so to avoid looping behaviors
  - I should rely on the programmer for a good definition and ordering of alternatives

Specifying alternatives

- Actions A%B
- Normally, A%B behaves like A
- If A%B is the target of a **roll**, it becomes B
- Intuitive meaning: try A, then try B
- Very simple alternative mechanism
- B may have alternatives too

Programming with alternatives

- We should find the actions that may lead to bad states
- We should replace them with actions with alternatives
- We need to find suitable alternatives
  - Retry
  - Retry with different resources
  - Give up and notify the user
  - Trace the outcome to drive future choices

Example

- Try to book a flight to Cagliari with Meridiana
- A Meridiana website error makes the booking fail
  - Retry: try again to book with Meridiana
  - Retry with different resources: try to book with Alitalia
  - Give up and notify the user: no possible booking, sorry
  - Trace the outcome to drive future choices: remember that Meridiana web site is prone to failure, next time try a different company first

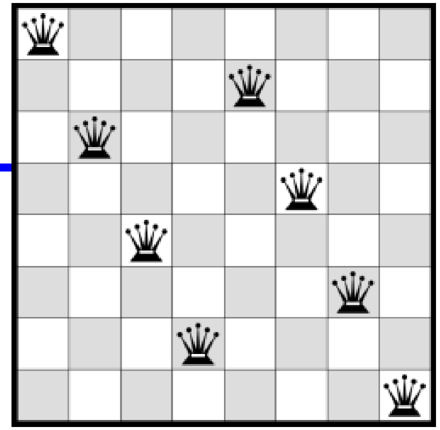
Other forms of alternatives

- Our alternatives are in sequence
  - Try A, then try B
- One can imagine to try A and B in parallel, when one of them succeeds the other computation is undone
  - Try both Meridiana and Alitalia
  - Book with the first one to give a good offer
  - This is called speculative parallelism

Is this enough?

- We have outlined a large piece of theory
  - Uncontrolled causal-consistent reversibility
  - A **roll** operator as control mechanism
  - Alternatives to avoid looping
- Did we find suitable abstractions for programming reliable systems?
- We need to put our constructs at work on a suitable benchmark
  - Still ongoing work

8 queens problem



- A classic state exploration program
  - 8 queens problem
  - Compact specification, concurrent algorithm
  - We need to improve efficiency
- Not innovative, but this is the first application programmed using a reversible causal-consistent calculus

Interacting transactions

- [Edsko de Vries, Vasileios Koutavas, Matthew Hennessy: Communicating Transactions. CONCUR 2010]
- Transactions that may interact with the environment and with other transactions while computing
- In case of abort one has to undo all the effects on the environment and on other transactions
  - To ensure atomicity

Interacting transactions via reversibility

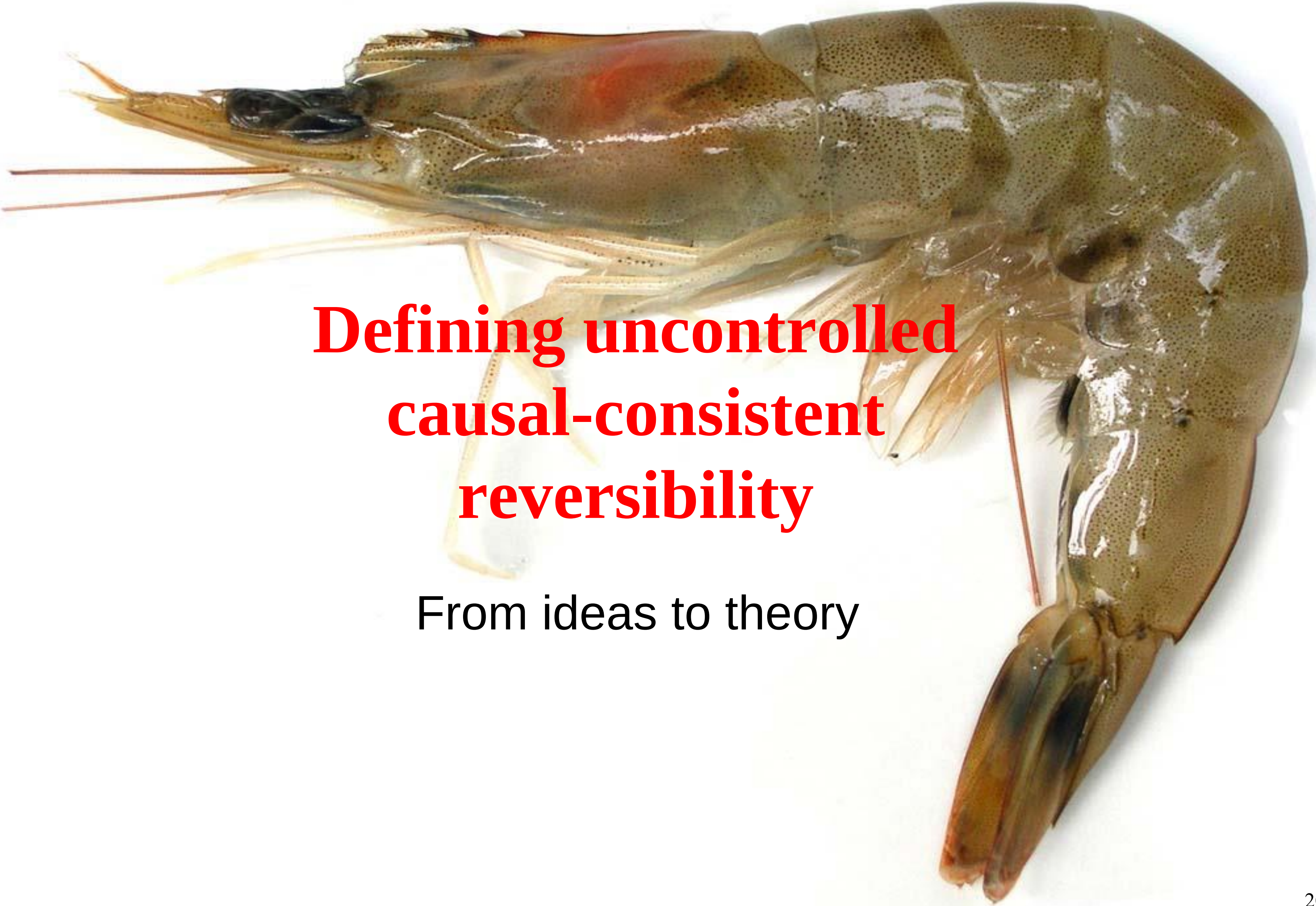
- We can encode interacting transactions
  - We label the start of the transaction with γ
  - An abort is a **roll** γ
  - The **roll** γ undoes all the effects of the transaction
  - A commit simply disables the **roll** γ
- The mapping is simple, the resulting code quite complex
  - We also need all the technical machinery for reversibility
- The encoding is more precise than the original semantics
  - We avoid some useless undo
  - Since our treatment of causality is more refined

Software transactional memories

- AtCCS
  [Lucia Acciai, Michele Boreale, Silvano Dal-Zilio: A Concurrent Calculus with Atomic Transactions. ESOP 2007]
  - A calculus for software transactional memories
  - We have been able to model most of it
  - In a compositional way

Long running transactions

- Used in the field of service oriented computing
- Computations that either succeed or are compensated
  - A compensation is an ad hoc piece of code
  - The compensation does an approximate undo of the effect of the transaction
- Atomicity is relaxed w.r.t. ACID transactions
- Reversibility does not help in encoding calculi for long running transactions
- However, reversibility may help the programmer in writing the compensation



Defining uncontrolled causal-consistent reversibility

From ideas to theory

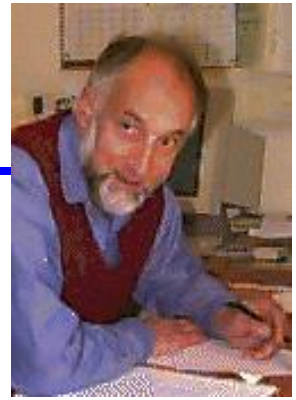
Our (theoretical) tools

- Process calculi
  - CCS, higher-order π
- Operational semantics
  - Mainly reduction semantics
- Programming languages
  - μOz

Why process calculi?

- Abstract view of programming languages
  - Focusing on interaction and communication
- Equipped with a well defined semantics
  - To clearly specify the intended behavior
- Equipped with suitable tools for reasoning
  - In particular behavioral equivalences
  - Allowing to prove our results
- When the basic issues have been understood we will move towards more realistic languages

CCS



- A calculus to model concurrent interacting systems
- One of the contributions for which Milner got the Turing award
- Syntax

$$P ::= a.\mathrm{P} \mid \overline{a}.P \mid P|P \mid P + P \mid 0 \mid \nu a\ P$$

- CCS normally includes other operators, but this is enough for our purposes
- We consider only guarded choice

Structural congruence

- Some terms are written in a different way, but have the same meaning
- Structural congruence $\equiv$ to equate them
  - Parallel composition and choice are associative, commutative and have 0 as neutral element
  - α-conversion: renaming of bound variables
  - $\nu a\ 0 \equiv 0$ $\qquad$ $\nu a\ (P|Q) \equiv (\nu a\ P)|Q$ if a not in $\mathrm{fn}(Q)$
  - $\nu a\ \nu b\ P \equiv \nu b\ \nu a\ P$
- As a consequence
  $\nu a\ P \equiv P$ if a not in $\mathrm{fn}(Q)$

Reduction semantics

- Defines the behavior of CCS terms
- One rule only

$$(\bar{a}.P + Q)|(a.P' + Q') \to P|P'$$

- Closed under structural congruence and contexts
  - Parallel composition and restriction

Making CCS reversible

- Structural congruence is already reversible
- The reduction rule loses lot of information
$$(\bar{a}.P + Q)|(a.P' + Q') \rightarrow P|P'$$
- We have lost a, Q and Q'
- We need to store this information
- We want a form of distributed storage
- First try
$$(\bar{a}.P + Q)|(a.P' + Q') \rightarrow P|P'|[a, Q, Q']$$
- We do not know where Q and Q' were attached
- Even worst if we have multiple processes and memories

Unique keys

- We need to relate the different parts
- We cannot refer them by description
  - Not memory efficient
  - Even worst, we cannot exchange equal terms with different histories
- We add unique keys to sequential processes
  - Processes beginning with prefix, choice or 0
  - Interaction is always between two sequential processes
- We have processes with keys such as $k: (a.P + Q)$

Reduction with keys

- Second try

  $$k\colon(\bar{a}.P + Q)|k'\colon(a.P' + Q') \rightarrow h\colon P|h'\colon P'|$$
  $$[a, Q, Q', k, k', h, h']$$

- The memory remembers that
  - the processes with key k and key k'
  - interacted on channel a (output on k)
  - discarding respectively processes Q and Q'
  - producing respectively continuations with key h and h'
- We have all the information to reverse the reduction
- Causality information: processes with key h and h' depend on processes with key k and key k'

Inventing keys

- At each step we invent two keys
  $k:(\bar{a}.P + Q)|k':(a.P' + Q') \rightarrow h:P|h':P'|$
  $[a, Q, Q', k, k', h, h']$
- To ensure uniqueness they have to be different from all the existing keys
- This is done by using restriction
- Third (and final) try
  $k:(\bar{a}.P + Q)|k':(a.P' + Q') \rightarrow \nu h, h' \;\; h:P|h':P'|$
  $[a, Q, Q', k, k', h, h']$

Undoing a step

- We have one backward reduction rule

$$h : P \mid h' : P' \mid [a, Q, Q', k, k', h, h'] \rightsquigarrow$$
$$k : (\overline{a}.P + Q) \mid k' : (a.P' + Q')$$

- Does the Loop Lemma holds?

$$k : (\overline{a}.P + Q) \mid k' : (a.P' + Q')$$
$$\rightarrow \nu h, h' \;\; h : P \mid h' : P' \mid [a, Q, Q', k, k', h, h']$$
$$\rightsquigarrow \nu h, h' k : (\overline{a}.P + Q) \mid k' : (a.P' + Q')$$

- Yes, up to structural congruence
- Other direction a bit more tricky

Invariants on keys

- Before reduction keys attached to sequential processes

$$k: (\bar{a}.P + Q) | k': (a.P' + Q') \rightarrow \nu h, h' \;\; h: P | h': P' | [a, Q, Q', k, k', h, h']$$

- And after?
- P and P' are arbitrary processes
- We want to find keys for the sequential processes
  - Otherwise they cannot reduce

Extending structural congruence

- We add two rules, one for restriction and one for parallel composition

  $$\mathrm{k}: \nu a \; \mathrm{P} \; \equiv \; \nu a \; k: P$$

  $$k: P|Q \equiv \nu k' \, \nu k'' \, k \prec k', k'' \mid k': P \mid k'': Q$$

- A connector $k \prec k', k''$ means that the process with index k has been split into processes with keys k' and k''
  - Again causality information
- Structural rules for restriction on names are extended to deal also with keys
- $k: P|k': 0 \equiv k: P$ does not hold