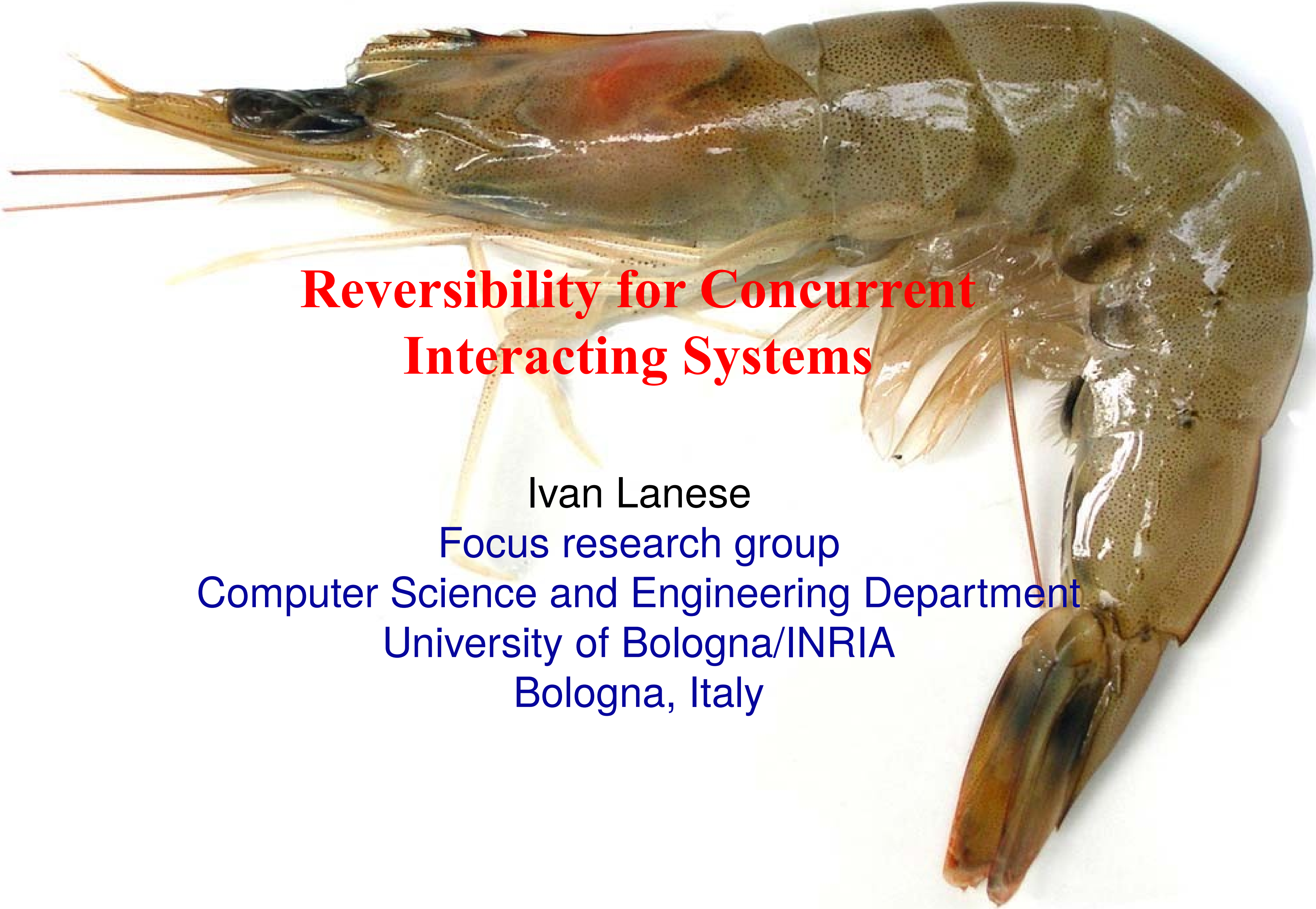
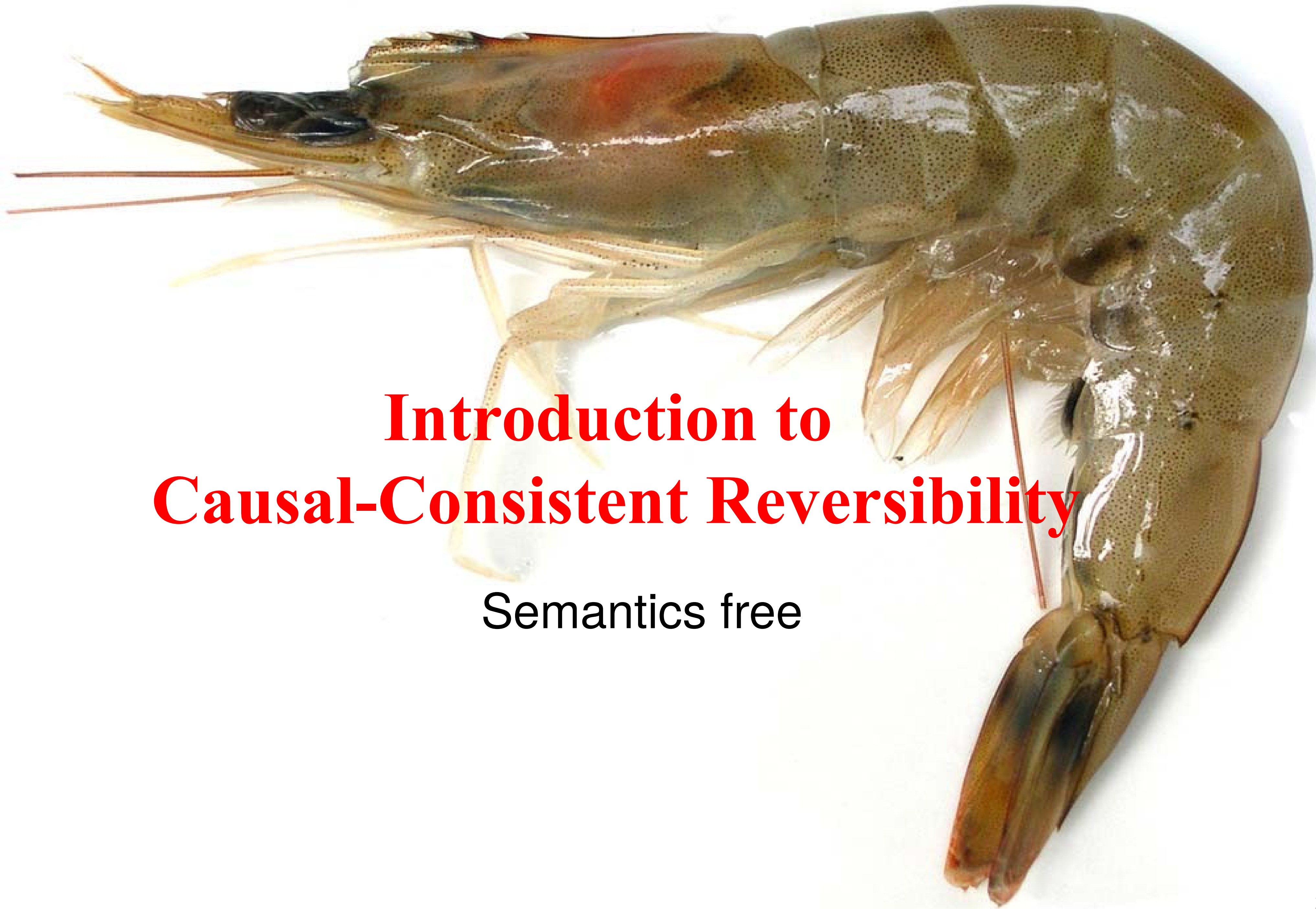


Reversibility for Concurrent Interacting Systems

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

Plan of the course

1. Introduction to causal-consistent reversibility
2. Defining uncontrolled causal-consistent reversibility
3. Controlling reversibility
4. Avoiding endless loops
5. An application: transactions
6. Reversing Erlang

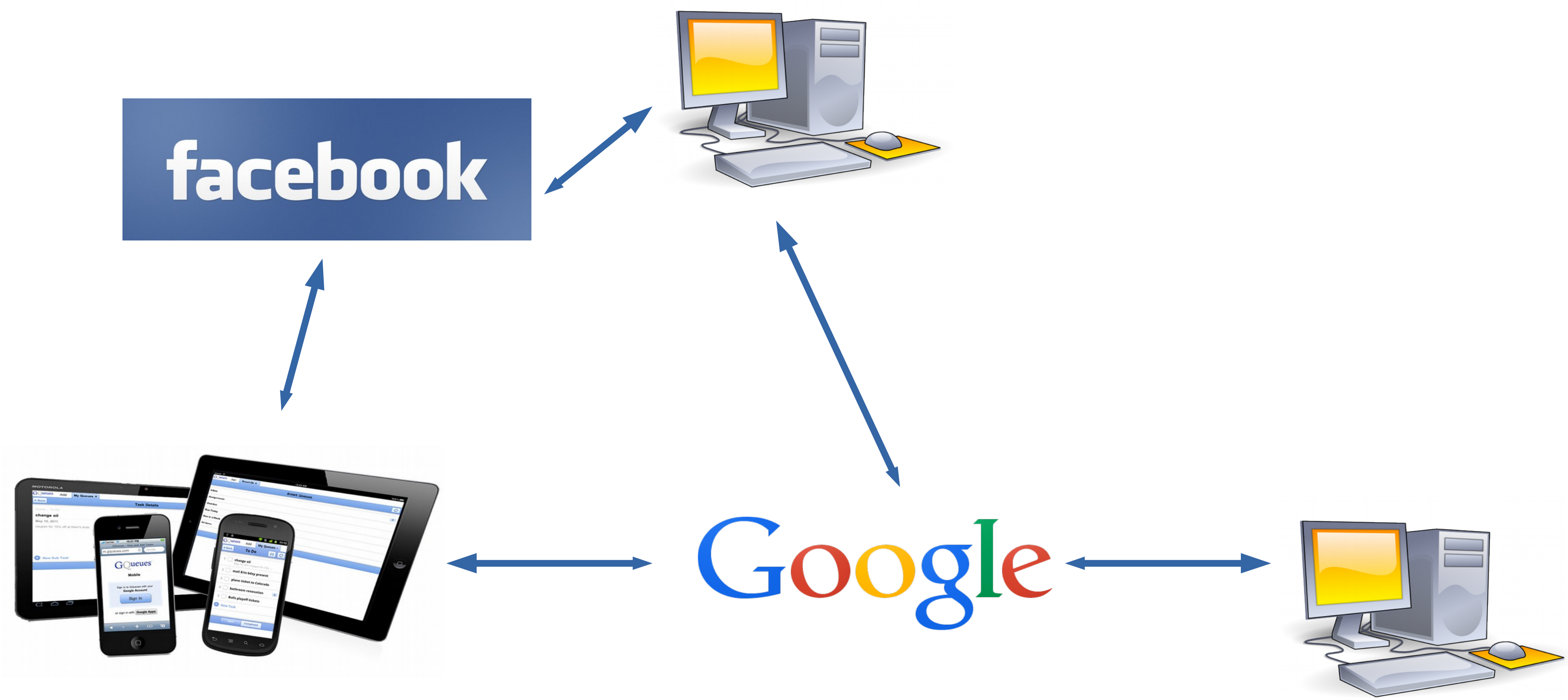


Introduction to Causal-Consistent Reversibility

Semantics free

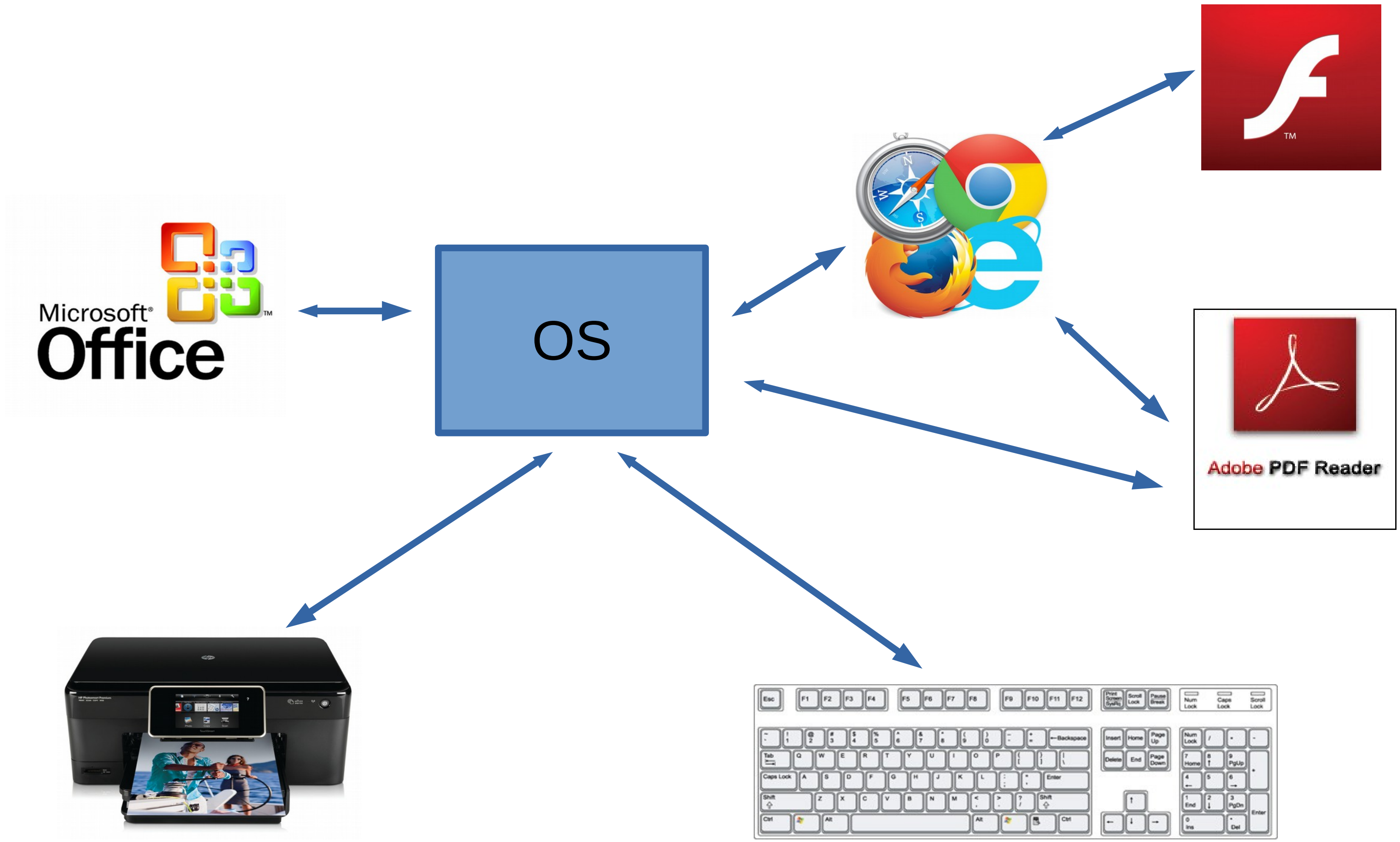
Concurrency and interaction everywhere

- Each distributed system is necessarily concurrent
  - E.g., the Internet



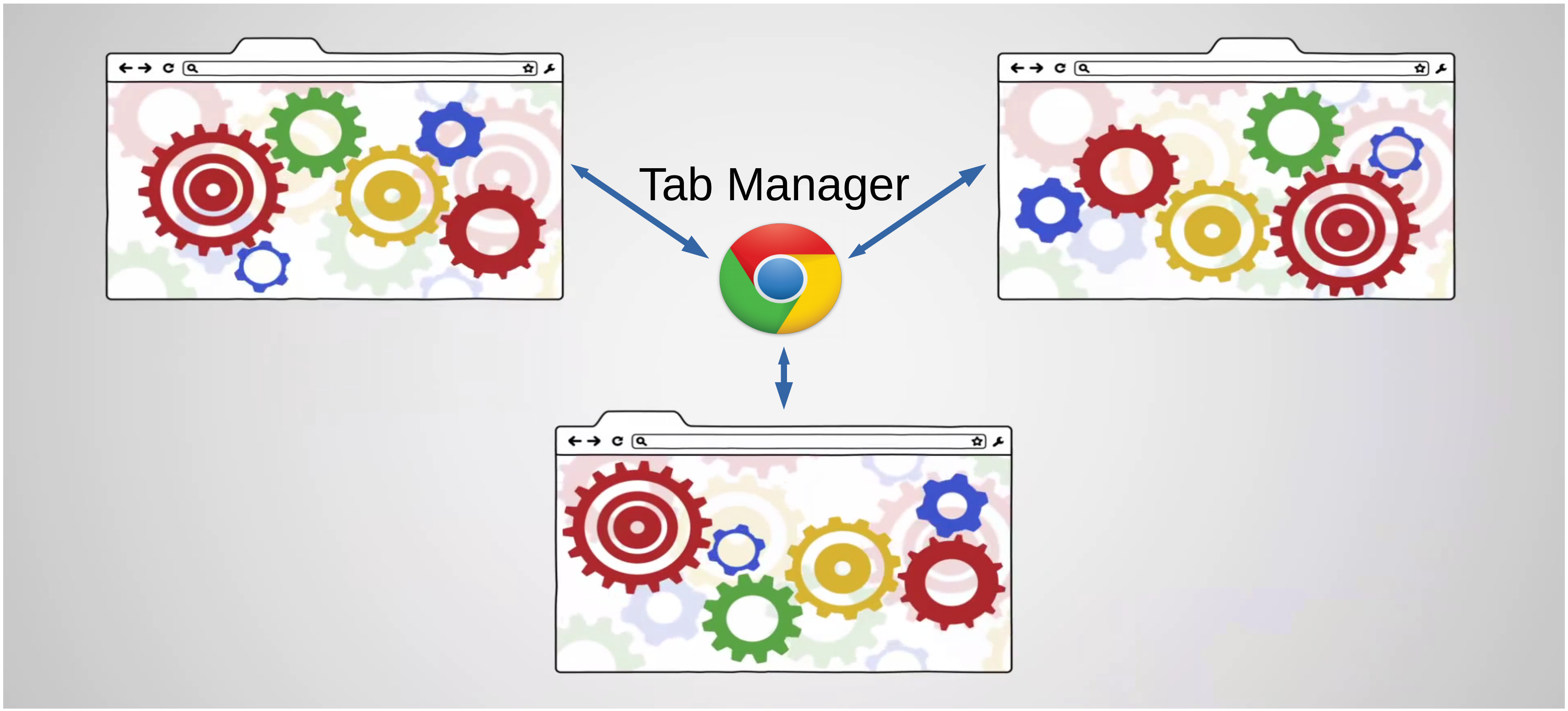
Concurrency and interaction everywhere

- Your computer features concurrency and interaction



Concurrency everywhere

- Single applications feature concurrency and interaction
  - E.g., google chrome



Reversibility everywhere

- Reversibility widespread in the world
  - Chemistry/biology, quantum phenomena
- Reversibility for modelling
- Reversibility widespread in computer science
  - Application undo, backup, svn, ...
- Reversibility for programming
  - State space exploration
  - View-update problem
  - Reliable systems (transactions, checkpoints)
  - Quantum computers
  - DNA circuits
- Reversibility for debugging

Reversibility in chemistry/biology

- Most of the chemical and/or biological phenomenons are reversible
- Direction of execution depends on environmental conditions such as temperature or pressure
- RCCS, the first reversible process calculus, was devised to model biological systems
  [Vincent Danos, Jean Krivine: Reversible Communicating Systems. CONCUR 2004]
- A reversible language for programming biological systems:
  [Luca Cardelli, Cosimo Laneve: Reversible structures. CMSB 2011]

State space exploration

- While exploring a state space towards a solution one may find a dead end
- Need to backtrack to find a solution
- This is the standard mechanism in Prolog
- State space exploration much easy in a reversible language
  - No need to program backtracking

View-update problem

- Views allow one to access (part of) a data structure
  - Views of databases
- The user may want to modify the view
- How to reflect the changes on the data structure?
- Easy if the view is generated by a reversible language
  - Lenses
- A survey of the approach is in [Benjamin C. Pierce et al.: Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. ACM Trans. Program. Lang. Syst. 29(3) (2007)]

Reversibility for reliability

- To make a system reliable we want to avoid “bad” states
- If a bad state is reached, reversibility allows one to go back to some past state
- Far enough, so that the decisions leading to the bad state have not been taken yet
- When we restart computing forward, we should try new directions

Reversibility for reliability:examples

- Checkpointing
  - We save the state of a program to restore it in case of errors
- Rollback-recovery
  - We combine checkpoints with logs to recover a program state
- Transactions
  - Computations which are executed all or nothing
  - In case of error their effect should be undone
  - Both in database systems (ACID transactions) and in service oriented computing (long running transactions)
- A reversible setting seems useful to study these patterns and to devise new ones

Reverse execution of a sequential program

- Recursively undo the last step
  - Computations are undone in reverse order
  - To reverse A;B reverse B, then reverse A
- First we need to undo single computation steps
- 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 the last executed action) and then redoing A should lead back to S

Undoing computational steps

- Not necessarily easy
- Computation steps may cause loss of information
- X=5 causes the loss of the past value of X
- X=X+Y causes no loss of information
  - Old value of X can be retrieved by doing X=X-Y
  - In general, Janus assignments and other Janus commands do not cause loss of information
- X=X*Y causes the loss of the value of X only if Y is 0

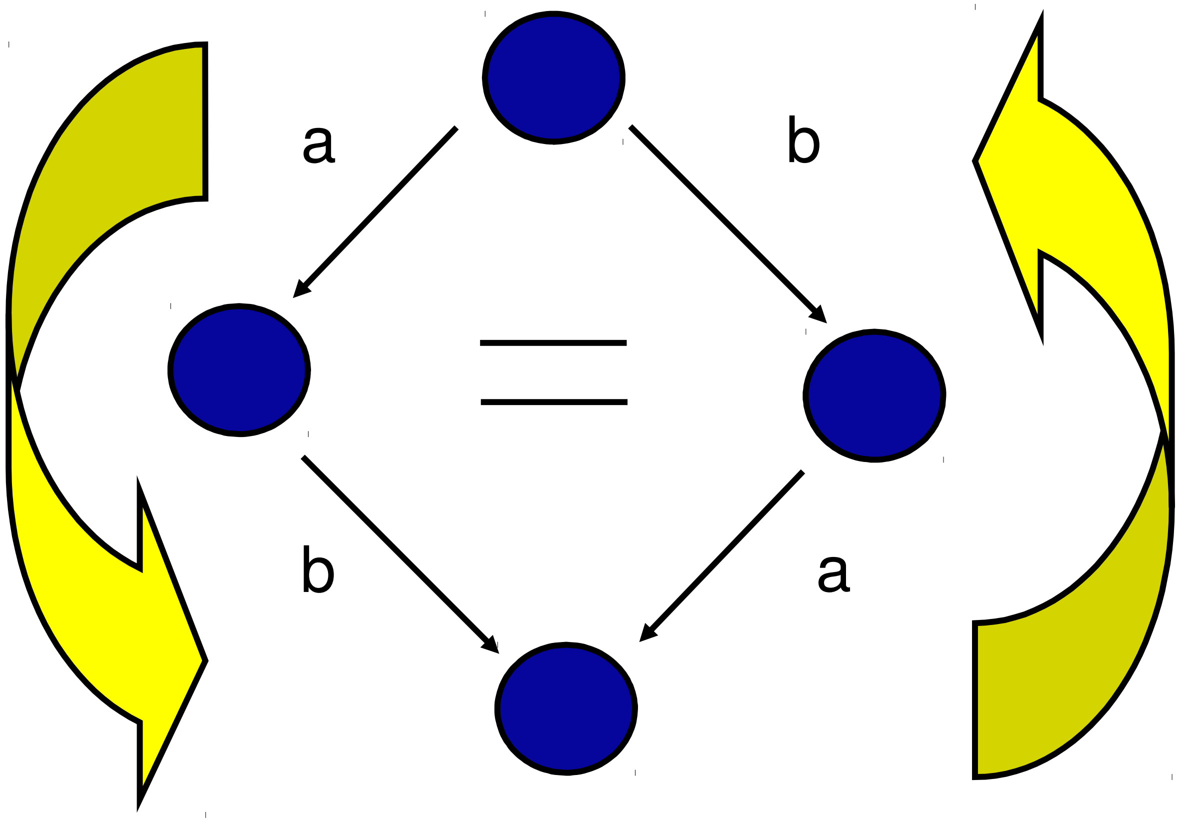
Different approaches to undo

- Saving a past state and redoing the same computation from there (rollback-recovery)
- Undoing steps one by one
  - Limiting the language to constructs that are reversible
    - Featuring only actions that cause no loss of information
    - Janus approach
  - Taking a language which is not reversible and make it reversible
    - One should save information on the past configurations
    - X=5 becomes reversible by recording the old value of X
- We concentrate on this last approach

Reversibility and concurrency


- In a sequential setting, recursively undo the last step
- Which is the last step in a concurrent setting?
- Not clear
- 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
  [Vincent Danos, Jean Krivine: Reversible Communicating Systems. CONCUR 2004]
  - Not backward deterministic (neither forward)

Causal-consistent reversibility



Why we want causal consistency?

- If we are not causal consistent we may undo a cause without undoing the consequence
- We reach a state where the consequence is in place, without any cause justifying it
- These are states that could not have been reached by forward execution
- Causal-consistent reversibility enables only the exploration of states reachable with a forward-only computation (when starting from an initial state)

Non-determinism versus concurrency

- In causal-consistent reversibility you need to distinguish concurrency from nondeterminism
- Two actions in a sequence whose order is chosen nondeterminstically need to be reversed in reverse order
- Two concurrent actions can be reversed in any order
  - Swapping concurrent actions should have no impact on the possible reverse behaviours

History information

- To reverse actions we need to store some history information
- Different threads are reversed independently
- It makes sense to attach history information to threads
- History information should trace where a thread comes from
  - X=5 destroys the old value of X
  - We need to store the old value of X to know the previous state of the thread

Causal history information

- We need to remember causality information
- If thread T_1 sent a message *m* to thread T_2 then T_1 cannot reverse the send before T_2 reverses the receive
  - Otherwise we would get a configuration where *m* has never been sent, but it has been received
- We need to remember that the send of *m* from T_1 caused the receive of *m* in T_2

Causal equivalence

- According to causal-consistent reversibility
  - Changing the order of execution of concurrent actions should not make a difference
  - Doing an action and then undoing it should not make a difference (Loop Lemma)
- Two computations are causal equivalent if they are equal up to the transformations above

Causal Consistency Theorem

- Two coinitial computations are causal equivalent iff they are cofinal
- Causal equivalent computations should
  - Lead to the same state
  - In particular, they produce the same history information
- Computations which are not causal equivalent
  - Should not lead to the same state
  - Otherwise we would erroneously reverse at least one of them in the wrong way
  - If in a non reversible setting they would lead to the same state, we should add history information to distinguish the states

Example

- If x>5 then
  y=6;x=2
  else
  x=2;y=6
  endif
- Two possible computations
- The two possible computations lead to the same state
- From the causal consistency theorem we know that we need history information to distinguish them
  - At least we should trace the chosen branch

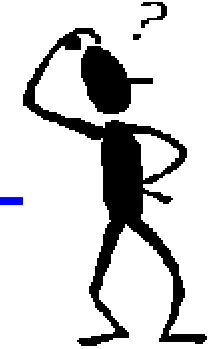
Parabolic Lemma

- Each computation is causally equivalent to a computation obtained by doing a backward computation followed by a forward computation
- Intuitively, we undo all what we have done and then compute only forward
  - Tries which are undone are not relevant
- Useful for proving the Causal Consistency Theorem

What we know

- We have some idea about how to define a causal-consistent reversible variant of a concurrent language
  - We need to satisfy the Loop Lemma
  - We need to satisfy the Causal Consistency Theorem
- More technicalities are needed to do it
- We continue to explore reversibility in an informal way

What we don’t know



- We know only uncontrolled reversibility
- We have a language able to go both back and forward
- When to go backward and when to go forward?
- Just non-deterministic is not a good idea
  - The program may go back and forward between the same states forever
  - If a good state is reached, the program may go back and lose the computed result
- We need some form of control for reversibility

Reversibility control

- One may imagine different ways of controlling reversibility
- We will show some possible alternatives
- We will try to categorize them
- We will try to understand the state space of the possible mechanisms
- The choice of the best mechanism depends on the intended application field

A taxonomy for reversibility control

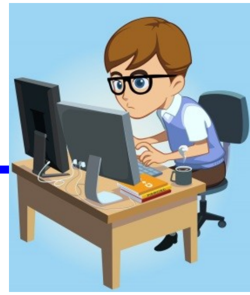
- Categorization according to who controls the reversibility
- Three different possibilities
  - Internal control: reversibility is controlled by the programmer
  - External control: reversibility is controlled by the environment
  - Semantic control: reversibility control is embedded in the semantics of the language

λ

Internal control

- Reversibility is controlled by the programmer
- Explicit operators to specify whether to go backward and whether to go forward
- We have different possibilities
  - Irreversible actions
    [Vincent Danos, Jean Krivine: Transactions in RCCS. CONCUR 2005]
  - **Roll** operator
    [Ivan Lanese, Claudio Antares Mezzina, Alan Schmitt, Jean-Bernard Stefani: Controlling Reversibility in Higher-Order Pi. CONCUR 2011]

Irreversible actions



- Execution is non-deterministically backward or forward
- Some actions, once done, cannot be undone
  - This allows to make a computed result permanent
  - They are a form of commit
- Still most programs are divergent
- Suitable to model biological systems
  - Most reactions are reversible
  - Some are not

Roll operator

- Normal execution is forward
- Backward computations are explicitly required using a dedicated command
- **Roll** γ, where γ is a reference to a past action
  - Undoes the action pointed by γ, and all its consequences
  - Go back n steps not meaningful in a concurrent setting
- γ is a form of checkpoint
- This allows to make a computed result permanent
  - If there is no **roll** pointing back past a given action, then the action is never undone
- Still most programs are divergent
- Suitable to program reliability patterns

External control

- Reversibility is controlled by something outside the program
- Again we have different possibilities
  - Controller processes
    [Iain Phillips, Irek Ulidowski, Shoji Yuen: A Reversible Process Calculus and the Modelling of the ERK Signalling Pathway. RC 2012]
  - Hierarchical component-based systems
  - Causal-consistent reversible debugger
    [see Claudio Mezzina's course]

Controller processes

- Two layered system
- A reversible slave process and a forward master process
- The slave process may execute only
  - Actions allowed by the master
  - In the direction allowed by the master
- Used to model biological systems
- Allows for non causal-consistent reversibility

Hierarchical component-based systems

- Systems featuring a hierarchy of components
- A generalization of the previous setting to multiple layers
- Each component controls the behavior of its children
  - Including the direction of their execution
- It needs information on the state of the children
  - E.g., each child should notify its errors
- Similar to Erlang error recovery style

Semantic control

- Reversibility policy embedded in the language
- Again we have different possibilities
  - Prolog
  - State-space exploration via heuristics
  - Energy-based control
    [Giorgio Bacci, Vincent Danos, Ohad Kammar: On the Statistical Thermodynamics of Reversible Communicating Processes. CALCO 2011]

Prolog backtracking

- Prolog tries to satisfy a given goal
- It explores deep-first the possible solutions
- When it reaches a dead end, it rollbacks and tries a different path
- The programmer may limit backtracking using cut
  - Not a pure semantic control, cut is internal control

State-space exploration via heuristics λ

- In general, there are different ways to explore a state space looking for a solution
- Strategy normally composed by a standard algorithm plus some heuristics driving it
- As before, if the algorithm reaches a dead end, it rollbacks and tries a different path
- Sample algorithm
  - count how many times each action has been done and undone
  - choose paths which have been tried less times

Energy-based control

λ

- We assume a world with a given amount of energy
- Forward and backward steps are taken subject to some probability
- The rates depend on the available amount of energy
- There is a lower bound on the amount of energy allowing to commit a forward computation in finite average time

Remember where we are

- Causal-consistent reversibility as a 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
- Let us go in a bit more details on the **roll** approach

More details on the **roll** 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 we should go back to a past state
  - We assume to be able to detect errors
- We 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 *b*
- We go back to a past good state *g*
- We may choose again the same path
- We reach the bad state *b* again
- We go back again to the same good state *g*
- 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 we would like 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
- We need to remember something from the past try
  - We should break the Loop Lemma
  - Reversibility should not be perfect

Alternatives

- We want to specify alternatives
- **Roll** causes the choice of a different alternative
- The programmer may declare alternatives so to avoid looping behaviors
  - We 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 Warsaw with Lufthansa
- A Lufthansa website error makes the booking fail
  - Retry: try again to book with Lufthansa
  - Retry with different resources: try to book with KLM
  - Give up and notify the user: no possible booking, sorry
  - Trace the outcome to drive future choices: remember that Lufthansa web site is prone to failure, next time try a different company first

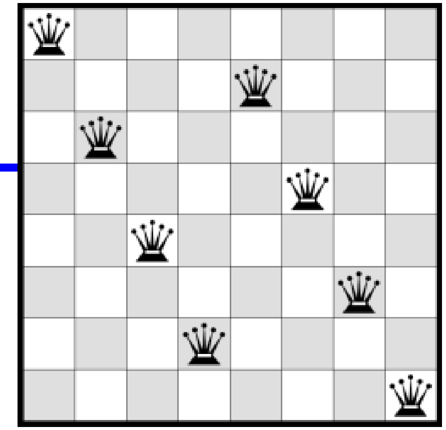
Other forms of alternatives

- 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 Lufthansa and KLM
  - 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
- How can we exploit this theory?
- We need to put our constructs at work on a suitable benchmark
  - We can look to the different application areas described at the beginning
  - We have some successful examples, but most of the work is still to be done
  - Debugging is one of these, see Claudio Mezzina's course

8 queens problem



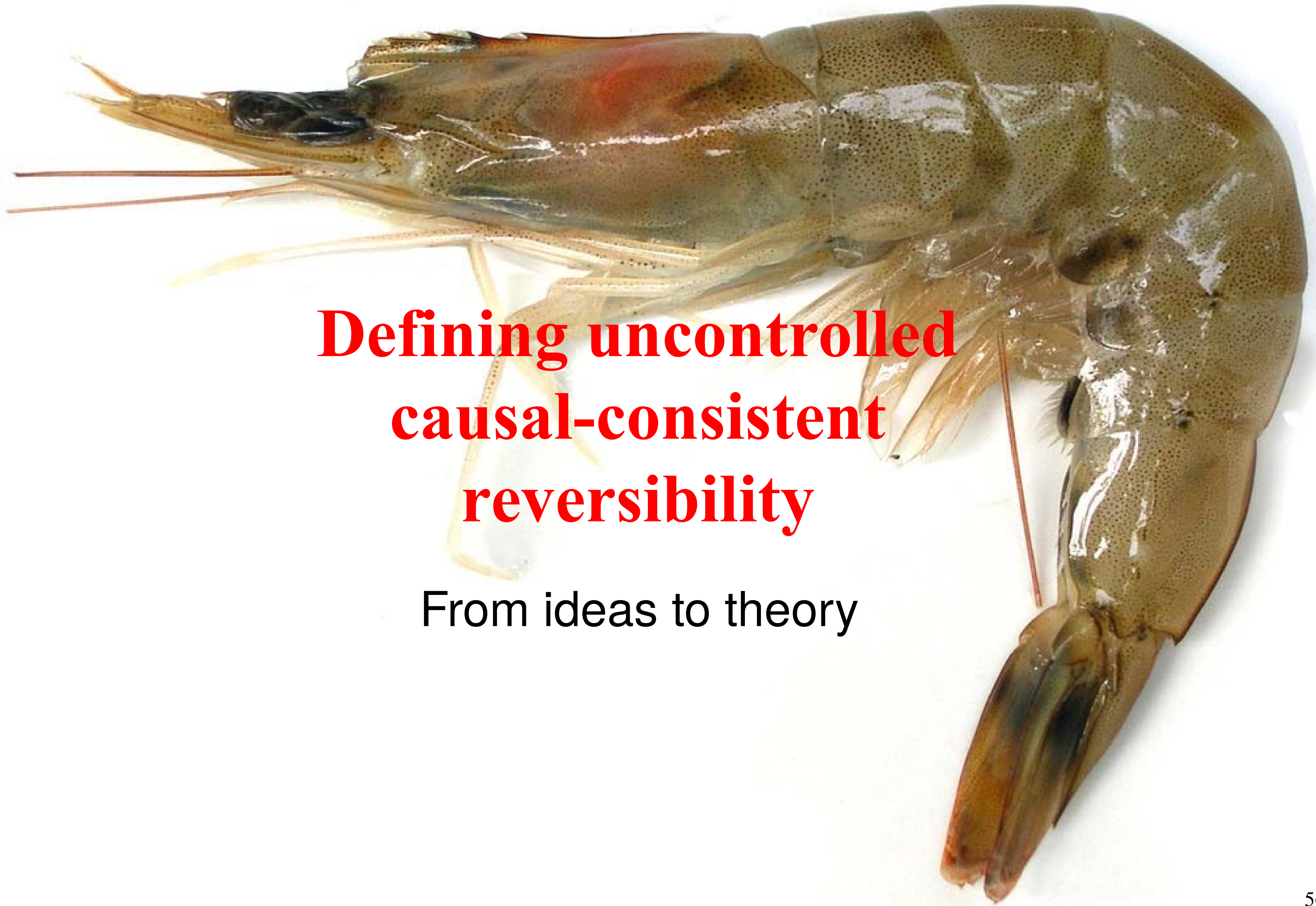
- A classic state exploration program
  - 8 queens problem
- Not innovative, but this is the first application programmed using a reversible causal-consistent calculus
- We will show the code later on
  - Compact, concurrent algorithm
  - Not very efficient

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 the other transactions
  - To avoid effects of aborted transactions

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



Defining uncontrolled causal-consistent reversibility

From ideas to theory

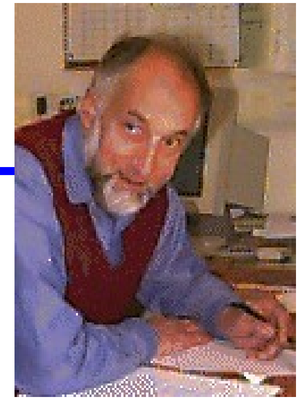
Our (theoretical) tools

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

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

$$P ::= a\,.\,P \,|\, \bar{a}\,.\,P \,|\, (P|Q) \,|\, P+Q \,|\, 0 \,|\, (\nu\, a)\, P$$

- 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
  - $(\nu a)0 \equiv 0$
  - $(\nu a)(P \mid Q) \equiv ((\nu a)P) \mid Q$ if a not in 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 fn(P)

Reduction semantics

- Defines the behavior of CCS terms
- One rule only

$$\left(\overline{a}.P+P'\right)\middle|\left(a.Q+Q'\right)\to P\middle|Q$$

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

Making CCS reversible

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

Unique keys

- We need to relate the different elements
- 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

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

- The memory remembers that
  - the processes with key *k* and key *k'*
  - interacted on channel *a* (output on *k*)
  - discarding respectively processes *P'* 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 key *h'* depend on processes with key *k* and key *k'*

Inventing keys

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

Undoing a step

- We have one backward reduction rule
  $$h:P|h':Q\|[a,P',Q',k,k',h,h'] \leftarrow$$
  $$k:(\overline{a}.P+P')|k':(a.Q+Q')$$
- Does the Loop Lemma holds?
  $$k:(\overline{a}.P+P')|k':(a.Q+Q') \rightarrow$$
  $$\nu h,h' \;\; h:P|h':Q\|[a,P',Q',k,k',h,h'] \leftarrow$$
  $$\nu h,h' \;\; k:(\overline{a}.P+P')|k':(a.Q+Q')$$
- Yes, up to structural congruence
- Other direction a bit more tricky

Managing keys

- Before reduction keys attached to sequential processes

$$k:(\overline{a}.(P_1|P_2)+P')|k':(a.Q+Q')\rightarrow$$

$$\nu h,h' \quad h:(P_1|P_2)|h':Q|[a,P',Q',k,k',h,h']$$

- And after?
- $P_1|P_2$ is a parallel process
- We want to derive keys for the sequential processes
  - Otherwise they cannot reduce

Extending structural congruence

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

$$k : \nu a\, P \equiv \nu a \;\; k : P$$

$$k : P | Q \equiv k \prec k_1 k_2 | k_1 : P | k_2 : Q$$

- A connector $k \prec k_1 k_2$ means that the process with key k has been split into processes with keys k_1 and k_2
  - 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

Example

$$k:\overline{a}.P|k':(a.b.0+a.c.0)|k'':\overline{b}.(Q|Q')\rightarrow$$
$$\nu h,h' \quad h:P|h':b.0\|[a,0,a.c.0,k,k',h,h']|k'':\overline{b}.(Q|Q')\rightarrow$$
$$\nu h,h',l,l' \quad h:P\|[a,0,a.c.0,k,k',h,h']\|$$
$$[b,0,0,h',k'',l,l']\|l:0|l':(Q|Q')\leftarrow$$
$$\nu h,h',l,l' \quad h:P|h':b.0\|[a,0,a.c.0,k,k',h,h']\|k'':\overline{b}.(Q|Q')\leftarrow$$
$$\nu h,h',l,l' \quad k:\overline{a}.P|k':(a.b.0+a.c.0)|k'':\overline{b}.(Q|Q')$$

ρCCS vs CCS

- Given a CCS process P we can generate a ρCCS configuration as $\nu k\ k{:}P$
  - No memories
  - No causal dependencies
- The programmer writes the CCS process and transforms it into a ρCCS configuration
- Given a ρCCS configuration one can generate a CCS process by removing all the additional information
- The two transformations form a Galois connection
  - α from ρCCS to CCS
  - c from CCS to ρCCS

ρCCS vs CCS, behaviorally

- Forward reductions of ρCCS configurations are CCS reductions
  - $M \rightarrow M'$ implies $\alpha(M) \rightarrow \alpha(M')$
- Given a CCS reduction, this can be done by any ρCCS configuration mapped to it
  - $P \rightarrow P'$ and $\alpha(M)=P$ implies $M \rightarrow M'$ and $\alpha(M')=P'$
  - History information has no impact on forward reductions

Valid configurations

- Not all the configurations are valid
- E.g., if the configuration contains a connector $k \prec k_1 k_2$ then k_1 and k_2 occur also as keys of a process, a memory or another connector
- Causality information should form a partial order
- A bit difficult to characterize syntactically valid configurations
- Semantic characterization: a configuration is valid iff it can be derived from a configuration of the form $\nu k\ k{:}P$

ρCCS in the literature

- You will not find any reference to ρCCS in the literature
- I defined it just for teaching purposes
- Based on the approach introduced for HOπ in
  [Ivan Lanese, Claudio Antares Mezzina, Jean-Bernard Stefani: Reversing Higher-Order Pi. CONCUR 2010]
  [Ivan Lanese, Claudio Antares Mezzina, Jean-Bernard Stefani: Reversibility in the higher-order π-calculus. Theor. Comput. Sci. 625 (2016)]

Causal-consistent CCS in the literature

- In the literature there are two other causal-consistent reversible CCS
  - RCCS
    [Vincent Danos, Jean Krivine: Reversible Communicating Systems. CONCUR 2004]
    Histories attached to threads
  - CCSk
    [Iain C. C. Phillips, Irek Ulidowski: Reversing Algebraic Process Calculi. FoSSaCS 2006]
    Process is not consumed, part of it is just annotated as no more active
- Both approaches are LTS based
- Rich literature built on both the approaches

ρCCS vs RCCS/CCSk

- Reduction-based vs LTS-based approach
- Reductions in ρCCS correspond to internal steps (τ moves) of RCCS/CCSk
- LTS-based: compositional, but more complex
- Reduction-based: not compositional, but simpler
  - Can be generalized to more complex languages (HOπ, Klaim, Erlang, ...)
- We will now discuss CCSk

CCS LTS semantics

- Defines the compositional behavior of CCS terms

$$\alpha.P \xrightarrow{\alpha} P \qquad \frac{P \xrightarrow{\alpha} P'}{P+Q \xrightarrow{\alpha} P'}$$

$$\frac{P \xrightarrow{\alpha} P'}{P|Q \xrightarrow{\alpha} P'|Q} \qquad \frac{P \xrightarrow{\alpha} P' \quad Q \xrightarrow{\bar{\alpha}} Q'}{P|Q \xrightarrow{\tau} P'|Q'}$$

$$\frac{P \xrightarrow{\alpha} P' \quad \alpha \neq a, \bar{a}}{\nu a P \xrightarrow{\alpha} \nu a P'}$$

CCSk approach

- Executed and discarded actions are not dropped
- Executed actions are marked with a fresh key
- Synchronizing actions have the same key
- We denote with X processes that may have executed actions, with P processes with no executed actions
- Reverse rules are just forward rules read in the opposite direction

CCSk semantics

$$\alpha.P \xrightarrow{\alpha[k]} \alpha[k].P$$

$$\frac{X \xrightarrow{\beta[h]} X' \quad k \neq h}{\alpha[k].X \xrightarrow{\beta[h]} \alpha[k].X'}$$

$$\frac{X \xrightarrow{\alpha[k]} X'}{X+Q \xrightarrow{\alpha[k]} X'+Q}$$

$$\frac{X \xrightarrow{\alpha[k]} X' \quad fresh(k,Y)}{X|Y \xrightarrow{\alpha[k]} X'|Y}$$

$$\frac{X \xrightarrow{\alpha[k]} X' \quad Y \xrightarrow{\overline{\alpha}[k]} Y'}{X|Y \xrightarrow{\tau} X'|Y'}$$

$$\frac{X \xrightarrow{\alpha[k]} X' \quad \alpha \neq a, \overline{a}}{\nu a\, X \xrightarrow{\alpha[k]} \nu a\, X'}$$

CCSk example

$$\nu a\left(a.c.d.P+b.Q\right)|\overline{a}.0|\overline{d}.R \xrightarrow{\tau}$$

$$\nu a\left(a[k].c.d.P+b.Q\right)|\overline{a}[k].0|\overline{d}.R \xrightarrow{c[h]}$$

$$\nu a\left(a[k].c[h].d.P+b.Q\right)|\overline{a}[k].0|\overline{d}.R \xrightarrow{\tau}$$

$$\nu a\left(a[k].c[h].d[l].P+b.Q\right)|\overline{a}[k].0|\overline{d}[l].R \xleftarrow{\tau}$$

$$\nu a\left(a[k].c[h].d.P+b.Q\right)|\overline{a}[k].0|\overline{d}.R \xleftarrow{c[h]}$$

$$\nu a\left(a[k].c.d.P+b.Q\right)|\overline{a}[k].0|\overline{d}.R \xleftarrow{\tau}$$

$$\nu a\left(a.c.d.P+b.Q\right)|\overline{a}.0|\overline{d}.R$$

CCSk main results

- Loop Lemma and causal consistency hold
- LTS semantics allows the composition of computations
- Bisimulation can be defined

What about RCCS?

- RCCS and CCSk are equivalent (ongoing work)
- Encoding from CCSk to RCCS and viceversa (correctness proof only for the forward direction) in
  [Doriana Medic, Claudio Antares Mezzina:
  Static VS Dynamic Reversibility in CCS. RC 2016]
- They provide the same runtime support for reversibility, in different ways

How many causal-consistent CCS do exist?

- Essentially one (ongoing work)
- There exists a unique way to define a causal-consistent extension of a given language
  - Satisfying the expected properties
  - For a fixed notion of causality

From ρCCS to ρπ

- CCS is not expressive enough
- We want to consider more expressive languages
- We choose higher-order π-calculus
  - CCS-style synchronization
  - During synchronizations processes are communicated (higher-order)

HOπ

- Syntax

$$P ::= a\langle P\rangle \mid a(X) \triangleright P \mid (P \mid Q) \mid X \mid 0 \mid (\nu a) P$$

- Higher-order communication
- Asynchronous calculus
- You can imagine structural congruence
- A reduction rule

$$a\langle P\rangle \mid a(X) \triangleright Q \rightarrow Q\left\{{}^{P}\!/_{X}\right\}$$

Infinite behaviors

- HOπ can implement infinite behaviors
  - No need for operators for replication or recursion
- $Q = a(X) \triangleright (P|X|a\langle X \rangle)$
  $Q|a\langle Q \rangle$ reduces to $P|Q|a\langle Q \rangle$
- This allows one to generate an infinite amount of copies of *P*

How to make HOπ reversible?

- The main novelty is given by substitutions
- In ρCCS we can take the continuations from the configuration
- In HOπ this is no more true
- From $Q\{P/X\}$ we cannot recover P nor Q
- Not even Q if we know P
  - $P|P$, $P|X$, $X|P$ and $X|X$ all produce the same result
- Not even P if we know Q
  - If Q does not contain X

ρπ

- Syntax:
  $$M ::= k : P \,|\, [\mu ; k] \,|\, k \prec k_1 k_2 \,|\, (M \,|\, M') \,|\, 0 \,|\, (\nu u) M$$
  $$\mu ::= k : a\langle P \rangle \,|\, k' : a(X) \triangleright Q$$
- Reduction rules:
  $$k : a\langle P \rangle \,|\, k' : a(X) \triangleright Q \to \nu k''\, k'' : Q\{{}^{P}/_{X}\} \,|\, [\mu ; k'']$$
  $$k'' : R \,|\, [\mu ; k''] \leftarrow \mu$$
- A unique continuation since the calculus is asynchronous
- We store the whole configuration
  - Not really memory efficient
  - But it works, and provides a simple semantics
  - One may optimize it

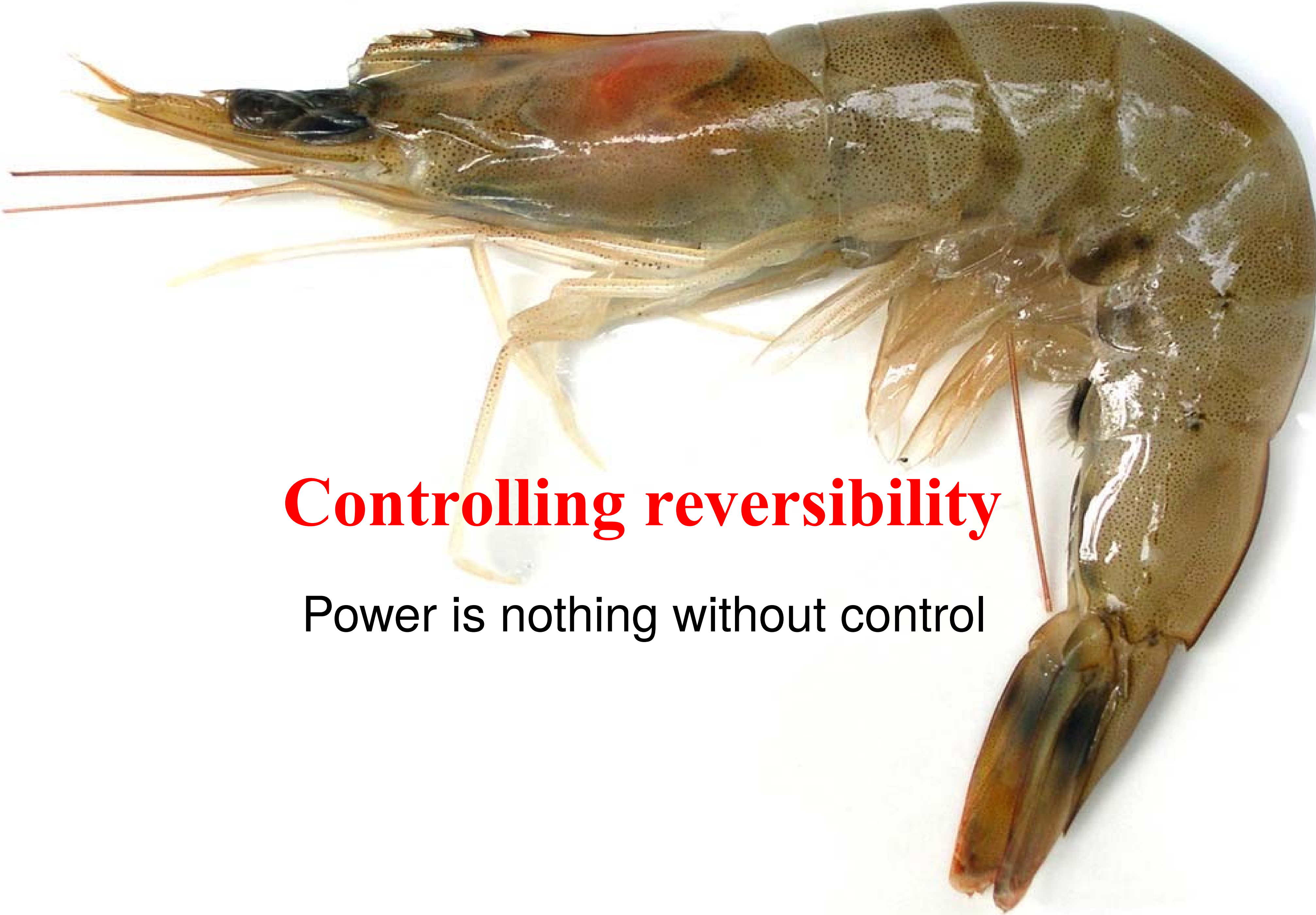
Restriction

- It seems we do not consider restriction
- Indeed, this is what we do
- We can do it!
- Try what happens with

$$k : a\langle \nu b \;\; c\langle b\langle Q\rangle .0\rangle\rangle | k' : a(X) \triangleright X | k'' : c(Y) \triangleright Y$$

Summarizing

- We have been able to define reversible CCS (reduction and LTS) and HOπ (reduction)
- All causal consistent
- For reductions, using almost the same technique
  - The technique can be applied to many other calculi as well
  - The technique for LTS is not so easy to generalize [Ioana Cristescu, Jean Krivine, Daniele Varacca: A compositional semantics for the reversible π-calculus, LICS 2013]
- But we are still at uncontrolled reversibility



Controlling reversibility

Power is nothing without control

Roll-π

- We want to use the **roll** operator to control reversibility in $\rho\pi$
- We have to attach labels γ to some actions
  - We choose triggers
  - Since triggers have a continuation
- The challenge is to define the semantics of the **roll** operator
  - It involves an unbounded number of processes
- We want to build on the uncontrolled semantics

Roll-π syntax

$$M ::= k : P \,|\, [\mu ; k] \,|\, k \prec k_1 k_2 \,|\, (M|M') \,|\, 0 \,|\, (\nu u) M$$

$$P ::= a\langle P\rangle \,|\, a(X) \triangleright_{\gamma} P \,|\, (P|Q) \,|\, X \,|\, 0 \,|\, (\nu a) P \,|\, roll\, \gamma \,|\, roll\, k$$

$$\mu ::= k : a\langle P\rangle \,|\, k' : a(X) \triangleright_{\gamma} Q$$

- Now γ attached to triggers
- The trigger is a binder for γ
- We do not want free occurrences of γ
- At run-time γ replaced by k

Roll-π semantics

- Little changes to the forward rule

$$k : a\langle P \rangle | k' : a(X) \triangleright_{\gamma} Q \rightarrow \nu k'' \;\; k'' : Q\{{}^{P}/_{X}\}\{{}^{k''}/_{\gamma}\} | [\mu ; k'']$$

- A new, complex, backward rule

$$\frac{M = k' : roll\, k | M' \quad M \leftarrow^{*} N \not\leftarrow \quad k < M' \quad complete(M)}{M \leftarrow_{r} N}$$

- The two last preconditions require to involve only processes which depend on *k*, and all of them
- We need to define the dependency relation

Exploiting causality

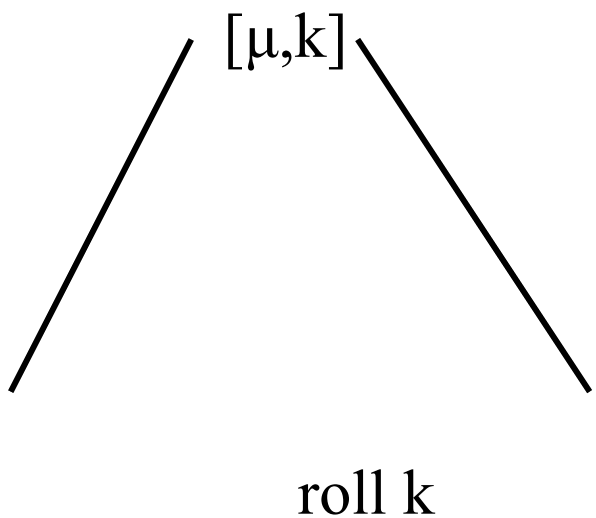
- Causal dependence: if in a configuration there is
  - $[k : a\langle P\rangle | k' : a(X) \triangleright Q ; k'']$ then $k > k''$ and $k' > k''$
  - $k \prec k' \, k''$ then $k > k'$ and $k > k''$
- $k > M$ if $k > h$ for all h:P, $[\mu;h]$ and $h \prec k' \, k''$ in M
- Completeness: if in a term I have
  - $[k : a\langle P\rangle | k' : a(X) \triangleright Q ; k'']$ then there is another occurrence of k''
  - $k \prec k' \, k''$ then there are other occurrences of k' and k''
- Completeness is essentially closure under consequences

Is roll-π a controlled ρπ?

- Let φ be a function that removes all γ and replaces all **roll**s with 0
  - Maps roll-π configurations to ρπ configurations
- $M \rightarrow_r M'$ iff $\varphi(M) \rightarrow \varphi(M')$
- If $M \leftarrow_r M'$ then $\varphi(M) \leftarrow^+ \varphi(M')$
  - The opposite implication holds only if a suitable **roll** exists

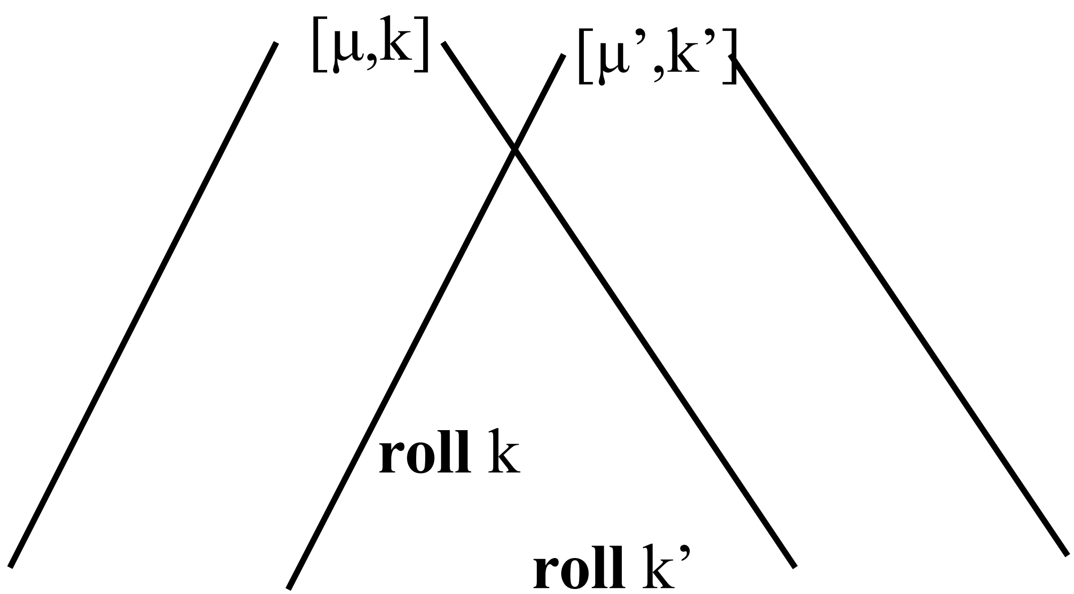
A graphical interpretation of **Roll**

- One can see the processes involved in a rollback as the tree of consequences of the key of the roll



Roll and concurrency

- Two **roll**s may interfere



- Executing one **roll** removes the other
- In a concurrent setting I would be able to execute both of them

Concurrent semantics for **Roll**

- We need to consider multiple **roll**s in the same step

$$\frac{M = M' \mid \prod_{i \in \{1 \ldots n\}} k'_i : roll\, k_i \quad M \leftarrow^{*} N \not\leftarrow \quad k_1 \ldots k_n < M' \quad complete(M)}{M \leftarrow_r N}$$

Going towards an implementation

- The rule defining the behavior of **roll** is not easy to implement
  - It involves an unbounded number of processes
- This semantics is a specification, not a guide to the implementation
- We can define a lower level semantics nearer to an implementation
- The lower level semantics should be equivalent to one of the more abstract semantics

A lower level semantics (simplified)

- A distributed algorithm based on message passing
- An active **roll** marks the target memory
- The marked memory sends messages “freeze” to all the descendants
  - The descendants forward the messages
  - If the descendant is a memory, the process(es) depending on the **roll** key are frozen
- When the message reaches a leaf, the leaf suicides by notifying its ancestors
  - If the leaf is a memory, non frozen processes are released
- The algorithm terminates when the marked memory is reached

Lower level semantics features

- Only binary interactions
- Easy to implement
- Indeed, we implemented it in Maude
- **Roll** execution is no more atomic
  - Loss of atomicity may create temporary inconsistencies
  - Inconsistencies are recovered when all **roll**s are done
  - A synchronization protocol is needed to avoid them
    - There is a bug on this point in the paper [Ivan Lanese, Claudio Antares Mezzina, Alan Schmitt, Jean-Bernard Stefani: Controlling Reversibility in Higher-Order Pi. CONCUR 2011]
  - Also, a **roll** execution may not terminate

Avoiding endless loops

No divergence please

Specifying alternatives in croll-π

- In roll-π every process featuring an executable **roll** has a divergent computation
- We want to give to the programmer tools to avoid this
- We use alternatives
- We add the simplest possible form of alternative
  - If something is simple and works, it is probably good (Occam razor)

Messages with alternative

- We attach alternatives only to messages
- Instead of messages $a\langle P\rangle$ we use messages with alternative
  - $a\langle P\rangle\%0$: try $a\langle P\rangle$, then stop trying
  - $a\langle P\rangle\%b\langle Q\rangle\%0$: try $a\langle P\rangle$, then $b\langle Q\rangle$, then stop trying
- If the message with alternative is the target of the **roll**, it is replaced by its alternative
- Very little change to the syntax
- Also the semantics is very similar
- The expressive power increases considerably

Croll-π syntax

$$M ::= k : P \mid [\mu ; k] \mid k \prec k_1 k_2 \mid (M \mid M') \mid 0 \mid (\nu u)\, M$$

$$P ::= a\langle P\rangle \% A \mid a(X) \triangleright_\gamma P \mid (P \mid Q) \mid X \mid 0 \mid (\nu a) P \mid roll\, \gamma \mid roll\, k$$

$$\mu ::= k : a\langle P\rangle \% A \mid k' : a(X) \triangleright_\gamma Q$$

$$A ::= 0 \mid b\langle Q\rangle \% 0$$

- Now messages have alternatives

Croll-π semantics

- Little changes to the forward rule

$$k : a\langle P\rangle \% A \mid k' : a(X) \triangleright_{\gamma} Q \rightarrow \nu k'' : Q\{^{P}/_{X}\}\{^{k''}/_{\gamma}\} \parallel [\mu ; k'']$$

- Little changes to the backward rule

$$\frac{M = k' : \mathit{roll}\, k \mid M' \quad \mathit{xtr}(M, k) \leftarrow^{*} N \not\leftarrow \quad k < M' \quad \mathit{complete}(M)}{M \leftarrow_{r} N}$$

- Function xtr is the identity but for

$$\mathit{xtr}([k : a\langle P\rangle \% A \mid k' : a(X) \triangleright_{\gamma} Q ; k''], k'') =$$
$$[k : A \mid k' : a(X) \triangleright_{\gamma} Q ; k'']$$

- It replaces the message target of the **roll** with its alternative

Arbitrary alternatives

- We only allow 0 and messages with 0 alternative as alternatives
  - Is this enough?
- We can encode arbitrary alternatives

$$\llbracket a\langle P\rangle\%Q\rrbracket = \nu c \quad a\langle\llbracket P\rrbracket\rangle\%c\langle\llbracket Q\rrbracket\rangle\%0|c(X)\triangleright X$$

- Q can even have alternatives
  - $a_1<P_1>\%\ldots\%a_n<P_n>\%0$ tries different options
  - By choosing $a_1=\ldots=a_n$ and $P_1=\ldots=P_n$ we try the same possibility n times before giving up

Endless retry

- We can retry the same alternative infinitely many times
  - This mimics roll-π messages

$$[\![a\langle P\rangle]\!] = \nu c \;\; Q | a\langle [\![P]\!]\rangle \% c\langle Q\rangle \% 0$$

$$Q = c(Z) \triangleright (Z | a\langle [\![P]\!]\rangle \% c\langle Z\rangle \% 0)$$

- As for replication, we can encode infinite behaviors using process duplication

Triggers with alternative

- We can attach alternatives to triggers instead of messages

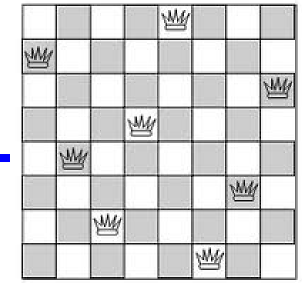
$$[\![a(X) \triangleright_{\gamma} Q \% b\langle R \rangle \% 0]\!] =$$
$$\nu c,d \;\; c\langle 0 \rangle \% d\langle 0 \rangle \% 0 |$$
$$(c(Y) \triangleright_{\gamma} a(X) \triangleright [\![Q]\!]) | (d(Z) \triangleright b\langle [\![R]\!] \rangle \% 0)$$

- We cannot mix triggers with alternative and messages with alternative

Expressive power

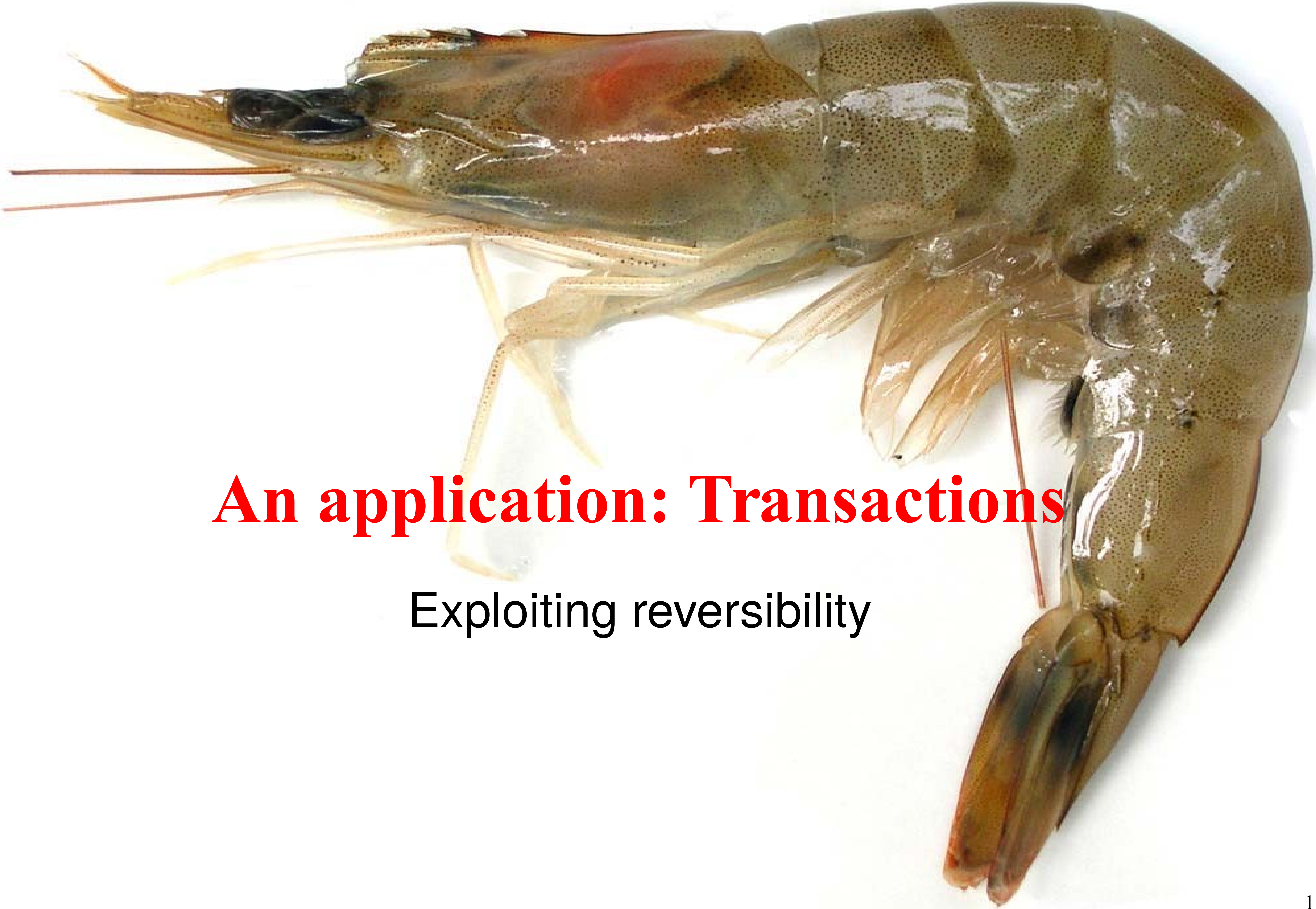
- Do alternatives increase the expressive power?
- Yes!
- We can prove this using encodings
- We can encode roll-π into croll-π
  - Using endless retry
- We cannot do the opposite, preserving both
  - Existence of a backward reduction
  - Termination
- The Loop Lemma does not hold in croll-π

The 8 queens



$$Q_i = act_i(Z) \triangleright p_i\langle i,1\rangle \% \ldots \% \; p_i\langle i,8\rangle \% f_i\langle 0\rangle \% 0 |$$
$$p_i(\boldsymbol{x_i}) \triangleright_{\gamma_i} (\, ! \, c_i\langle \boldsymbol{x_i}\rangle \% 0 | act_{i+1}\langle 0\rangle | f_{i+1}(Y) \triangleright roll\ \gamma_i |$$
$$\prod_{j=1}^{i-1} c_j(\boldsymbol{y_j}) \triangleright if\ err(\boldsymbol{x_i}, \boldsymbol{y_j})\ then\ \boldsymbol{roll}\ \gamma_i)$$

- ! denotes replication
  - We know we can encode it
- Compact and concurrent implementation



An application: Transactions

Exploiting reversibility

Interacting transactions

- We have been able to encode interacting transactions from
  [Edsko de Vries, Vasileios Koutavas, Matthew Hennessy: Communicating Transactions. CONCUR 2010]
- Improving on the original semantics
- Now we have the tools to understand why

Transactions with compensations

- They have the form $[P,Q]_\gamma$
- A transaction executing P, with compensation Q and with name γ
- Behaves as P
- It can either commit or abort
- In case of commit, the result is the same as executing P
- In case of abort, the effects of P are undone, and Q is executed
- Transactions are atomic: *P* is executed all or nothing
- Normally, transactions should not interact with each other (isolation)

Interacting transactions in TransCCS

- Syntax (subcalculus)
  $$P ::= \bar{a} \mid a.P \mid (P \mid Q) \mid 0 \mid (\nu a)P \mid [P \triangleright_k Q] \mid co\,k$$
- Semantics
  $$\bar{a} \mid a.P \rightarrow P$$
  $$[P \triangleright_k Q] \mid R \rightarrow [P \mid R \triangleright_k Q \mid R] \; if \; k \notin fn(R)$$
  $$[P \mid co\,k \triangleright_k Q] \rightarrow P$$
  $$[P \triangleright_k Q] \rightarrow Q$$
- Processes from the environment moved into the transaction to interact with it
  - Saved also in the compensation
- Implicit abort, explicit commit

Example: transactions interacting

$$[\bar{a} \rhd_k Q] \| [a.P \rhd_h Q'] \rightarrow$$

$$[[\bar{a} \rhd_k Q] | a.P \rhd_h [\bar{a} \rhd_k Q] | Q'] \rightarrow$$

$$[[\bar{a} | a.P \rhd_k Q | a.P] \rhd_h [\bar{a} \rhd_k Q] \| Q'] \rightarrow$$

$$[[P \rhd_k Q | a.P] \rhd_h [\bar{a} \rhd_k Q] \| Q']$$

- If both transactions commit we get *P*
- If both transactions abort we get Q|Q'
- Using the other embedding would have been fine too
- If other processes would be in the transaction *k* together with $\bar{a}$ then they would have entered the transaction *h* too

Example: external interactions aborted

$$\bar{a} \mid a\,.\,R \parallel [\, P \triangleright_k Q \,] \rightarrow$$
$$[\, \bar{a} \mid a\,.\,R \mid P \triangleright_k \bar{a} \mid a\,.\,R \mid Q \,] \rightarrow$$
$$[\, R \mid P \triangleright_k \bar{a} \mid a\,.\,R \mid Q \,] \rightarrow$$
$$\bar{a} \mid a\,.\,R \mid Q$$

- Why undoing the synchronization on *a*?
- No reason for it to occur inside the transaction

Transactions in croll-π

$$[\![[P, Q]_\gamma]\!] =$$

$$\nu a\, \nu c \;\; a\langle 0\rangle \% c\langle 0\rangle \% 0 \,|\, a(X) \triangleright_\gamma [\![P]\!] \,|\, c(Y) \triangleright [\![Q]\!]$$

- Abort is **roll** γ
- Commit is implicit: if there is no **roll** γ then the compensation and the transaction machinery become garbage
- We simulate the transaction boundary with causality tracking
- Atomic transaction
  - If *P* aborts all its effects are undone
- Not isolated

Interacting transactions in croll-π

$$[\![[P \triangleright_l Q]]\!] = [\nu l\, [\![P]\!] \mid l\langle \textbf{roll}\ \gamma\rangle \mid l(X) \triangleright X, [\![Q]\!]]_\gamma$$

$$[\![co\, l]\!] = l(X) \triangleright 0$$

- We simulate the automatic abort with a **roll** that can be enabled at any moment
- A commit disables the abort

Comparing the two approaches

$$[\![[P \rhd_l Q]]\!] = [\nu l\, [\![P]\!] | l\langle \boldsymbol{roll}\ \gamma \rangle | l(X) \rhd X, [\![Q]\!]]_\gamma$$

- In croll-π only reductions depending on the transaction body are undone
  - In TransCCS other reductions may be undone
  - Difference due to a more precise causality tracking in croll-π
- In croll-π abort is not atomic
  - First, commit becomes impossible
  - Then, abort is performed
- Atomicity problem solvable with choice
  - **roll** $\gamma + l(X) \rhd 0$
  - With l<0> as commit

Reversing Erlang

Welcome to the real world

Challenges of considering a real language

- Real languages are much bigger than CCS or HOπ
  - Around 100 constructs instead of around 10
- We need a semantics for them to work on
- We need to understand the causal semantics of each construct
- We need to understand how to reverse each construct

Erlang

- Functional, concurrent and distributed language from Ericsson
- Used in many relevant projects such as WhatsApp chat
- Based on the actor model
- Asynchronous message-passing communication

Facing the challenges

- We have chosen an actor-based language
  - Enables a clear separation between few concurrency-relevant constructs and sequential constructs
  - We can deal with sequential constructs in a uniform way
- Erlang is compiled into Core Erlang, which is more constrained and easy to deal with, yet equally expressive
  - We will consider Core Erlang
- Yet currently we do not support some more tricky features of Erlang
  - Mainly related to Erlang fault recovery model

Supported Core Erlang syntax

- $Module$ ::= module $Atom = fun_1, \ldots, fun_n$
- fun ::= $fname$ = fun $(X_1, \ldots, X_n) \rightarrow expr$
- $fname$ ::= $Atom/Integer$
- lit ::= $Atom \mid Integer \mid Float \mid [\,]$
- $expr$ ::= $Var \mid lit \mid fname \mid [expr_1|expr_2] \mid \{expr_1, \ldots, expr_n\}$
  | call $expr$ $(expr_1, \ldots, expr_n)$ | apply $expr$ $(expr_1, \ldots, expr_n)$
  | case $expr$ of $clause_1, \ldots, clause_m$ end
  | let $Var = expr_1$ in $expr_2$
  | receive $clause_1, \ldots, clause_n$ end
  | spawn($expr$, $[expr_1, \ldots, expr_n]$) | $expr_1$! $expr_2$ | self()
- $clause$ ::= pat when $expr_1 \rightarrow expr_2$
- pat ::= $Var \mid lit \mid [pat_1|pat_2] \mid \{pat_1, \ldots, pat_n\}$

Core Erlang semantics

- Two levels of semantics
  - A labelled semantics for expressions: labels describe side effects
  - An unlabelled semantics for systems
- The semantics exploits a run-time syntax
- A system is composed by:
  - A global mailbox Γ: messages travelling in the network
  - A set of threads
    - Each thread has a unique name p, a state θ, an expression under evaluation e, and a queue of waiting messages q

Core Erlang sequential expressions semantics

- Just a few sample rules

$$(Var)\ \frac{}{\theta, X \xrightarrow{\tau} \theta, \theta(X)}$$

$$(Tuple)\ \frac{\theta, e_i \xrightarrow{\ell} \theta', e_i'}{\theta, \{\overline{v_{1,i-1}}, e_i, \overline{e_{i+1,n}}\} \xrightarrow{\ell} \theta', \{\overline{v_{1,i-1}}, e_i', \overline{e_{i+1,n}}\}}$$

$$(Let2)\ \frac{}{\theta, \mathsf{let}\ X = v\ \mathsf{in}\ e \xrightarrow{\tau} \theta[X \mapsto v], e}$$

$$(Apply2)\ \frac{\mu(a/n) = \mathsf{fun}\ (X_1, \ldots, X_n) \to e}{\theta, \mathsf{apply}\ a/n\ (v_1, \ldots, v_n) \xrightarrow{\tau} \theta \cup \{X_1 \mapsto v_1, \ldots, X_n \mapsto v_n\}, e}$$

Core Erlang concurrent expressions semantics

$$(\mathit{Send1})\ \frac{\theta, e_1 \xrightarrow{\ell} \theta', e_1'}{\theta, e_1 \,!\, e_2 \xrightarrow{\ell} \theta', e_1' \,!\, e_2} \qquad (\mathit{Send2})\ \frac{\theta, e_2 \xrightarrow{\ell} \theta', e_2'}{\theta, v_1 \,!\, e_2 \xrightarrow{\ell} \theta', v_1 \,!\, e_2'}$$

$$(\mathit{Send3})\ \frac{}{\theta, v_1 \,!\, v_2 \xrightarrow{\mathsf{send}(v_1,v_2)} \theta, v_2}$$

$$(\mathit{Receive})\ \frac{}{\theta, \mathsf{receive}\ cl_1; \ldots; cl_n\ \mathsf{end} \xrightarrow{\mathsf{rec}(\kappa,\overline{cl_n})} \theta, \kappa}$$

$$(\mathit{Spawn})\ \frac{}{\theta, \mathsf{spawn}(a/n, [e_1, \ldots, e_n]) \xrightarrow{\mathsf{spawn}(\kappa,a/n,[\overline{e_n}])} \theta, \kappa}$$

$$(\mathit{Self})\ \frac{}{\theta, \mathsf{self}() \xrightarrow{\mathsf{self}(\kappa)} \theta, \kappa}$$

Core Erlang systems semantics

$$(Seq)\quad \frac{\theta, e \xrightarrow{\tau} \theta', e'}{\Gamma; \langle p, (\theta, e), q\rangle \mid \Pi \hookrightarrow \Gamma; \langle p, (\theta', e'), q\rangle \mid \Pi}$$

$$(Send)\quad \frac{\theta, e \xrightarrow{\mathsf{send}(p'', v)} \theta', e'}{\Gamma; \langle p, (\theta, e), q\rangle \mid \Pi \hookrightarrow \Gamma \cup (p'', v); \langle p, (\theta', e'), q\rangle \mid \Pi}$$

$$(Receive)\quad \frac{\theta, e \xrightarrow{\mathsf{rec}(\kappa, \overline{cl_n})} \theta', e' \quad \mathsf{matchrec}(\overline{cl_n}, q) = (\theta_i, e_i, v)}{\Gamma; \langle p, (\theta, e), q\rangle \mid \Pi \hookrightarrow \Gamma; \langle p, (\theta'\theta_i, e'\{\kappa \mapsto e_i\}), q \backslash\!\backslash v\rangle \mid \Pi}$$

$$(Spawn)\quad \frac{\theta, e \xrightarrow{\mathsf{spawn}(\kappa, a/n, [\overline{e_n}])} \theta', e' \quad p' \text{ is a fresh pid}}{\Gamma; \langle p, (\theta, e), q\rangle \mid \Pi \hookrightarrow \Gamma; \langle p, (\theta', e'\{\kappa \mapsto p'\}), q\rangle \mid \langle p', (\theta', \mathsf{apply}\ a/n\ (\overline{e_n})), [\,]\rangle \mid \Pi}$$

$$(Self)\quad \frac{\theta, e \xrightarrow{\mathsf{self}(\kappa)} \theta', e'}{\Gamma; \langle p, (\theta, e), q\rangle \mid \Pi \hookrightarrow \Gamma; \langle p, (\theta', e'\{\kappa \mapsto p\}), q\rangle \mid \Pi}$$

$$(Sched)\quad \frac{}{\Gamma \cup \{(p, v)\}; \langle p, (\theta, e), q\rangle \mid \Pi \hookrightarrow \Gamma; \langle p, (\theta, e), v : q\rangle \mid \Pi}$$

Core Erlang reversible semantics

- Preliminary version in
  [Naoki Nishida, Adrián Palacios, Germán Vidal:
  A Reversible Semantics for Erlang. LOPSTR 2016]
- We leave expressions semantics as it is
- We just change the systems semantics
- We add histories *h* to threads to remember past actions
  - Each history element stores (at least) the previous state and expression
  - We could optimize this, but this would make the semantics more complex
- We add unique identifiers λ to messages

Causality

- In order to define the reversible semantics we need to understand whether actions enabled at the same time are concurrent or in conflict
- Two concurrent actions can be executed in any order without changing the final result
  - Always true for actions in different threads
  - In the same thread two actions can be enabled together only if at least one is a Sched
  - E.g., a Sched and a Self are concurrent
  - Two Sched are not: the final queue depends on the order of execution
  - What about a Sched and a Receive?

Sched and Receive

- We can execute them in any order unless the Receive would read the message provided by the Sched
- This depends on the queue and on the patterns
- Very difficult to characterize
- We approximate by saying that a Sched and a Receive on the same thread are always in conflict

Reversible Core Erlang forward semantics

$$(Seq)\quad \frac{\theta, e \xrightarrow{\tau} \theta', e'}{\Gamma; \langle p, h, (\theta, e), q\rangle \mid \Pi \rightharpoonup \Gamma; \langle p, \tau(\theta, e) : h, (\theta', e'), q\rangle \mid \Pi}$$

$$(Send)\quad \frac{\theta, e \xrightarrow{\mathsf{send}(p'', v)} \theta', e' \quad \text{and} \quad \lambda \text{ is a fresh identifier}}{\Gamma; \langle p, h, (\theta, e), q\rangle \mid \Pi \rightharpoonup \Gamma \cup (p'', \{v, \lambda\}); \langle p, \mathsf{send}(\theta, e, p'', \{v, \lambda\}) : h, (\theta', e'), q\rangle \mid \Pi}$$

$$(Receive)\quad \frac{\theta, e \xrightarrow{\mathsf{rec}(\kappa, \overline{cl_n})} \theta', e' \quad \mathsf{matchrec}(\overline{cl_n}, q) = (\theta_i, e_i, \{v, \lambda\})}{\Gamma; \langle p, h, (\theta, e), q\rangle \mid \Pi \rightharpoonup \Gamma; \langle p, \mathsf{rec}(\theta, e, \{v, \lambda\}, q) : h, (\theta'\theta_i, e'\{\kappa \mapsto e_i\}), q \backslash\!\backslash \{v, \lambda\}\rangle \mid \Pi}$$

$$(Spawn)\quad \frac{\theta, e \xrightarrow{\mathsf{spawn}(\kappa, a/n, [\overline{e_n}])} \theta', e' \quad p' \text{ is a fresh pid}}{\begin{array}{ll}\Gamma; \langle p, h, (\theta, e), q\rangle \mid \Pi \rightharpoonup & \Gamma; \langle p, \mathsf{spawn}(\theta, e, p') : h, (\theta', e'\{\kappa \mapsto p'\}), q\rangle \\ & \mid \langle p', [\,], (\theta, \mathsf{apply}\ a/n\ (\overline{e_n})), [\,]\rangle \mid \Pi\end{array}}$$

$$(Self)\quad \frac{\theta, e \xrightarrow{\mathsf{self}(\kappa)} \theta', e'}{\Gamma; \langle p, h, (\theta, e), q\rangle \mid \Pi \rightharpoonup \Gamma; \langle p, \mathsf{self}(\theta, e) : h, (\theta', e'\{\kappa \mapsto p\}), q\rangle \mid \Pi}$$

$$(Sched)\quad \frac{}{\Gamma \cup \{(p, \{v, \lambda\})\}; \langle p, h, (\theta, e), q\rangle \mid \Pi \rightharpoonup \Gamma; \langle p, h, (\theta, e), \{v, \lambda\} : q\rangle \mid \Pi}$$

Reversible Core Erlang backward semantics

$$(\overline{Seq}) \quad \Gamma; \langle p, \tau(\theta, e) : h, (\theta', e'), q \rangle \mid \Pi \leftharpoondown \Gamma; \langle p, h, (\theta, e), q \rangle \mid \Pi$$

$$(\overline{Send}) \quad \Gamma \cup \{(p'', \{v, \lambda\})\}; \langle p, \mathsf{send}(\theta, e, p'', \{v, \lambda\}) : h, (\theta', e'), q \rangle \mid \Pi \leftharpoondown \Gamma; \langle p, h, (\theta, e), q \rangle \mid \Pi$$

$$(\overline{Receive}) \quad \Gamma; \langle p, \mathsf{rec}(\theta, e, \{v, \lambda\}, q) : h, (\theta', e'), q \backslash\!\!\backslash \{v, \lambda\} \rangle \mid \Pi \leftharpoondown \Gamma; \langle p, h, (\theta, e), q \rangle \mid \Pi$$

$$(\overline{Spawn}) \quad \begin{array}{l} \Gamma; \langle p, \mathsf{spawn}(\theta, e, p') : h, (\theta, e), q \rangle \mid \langle p', [\,], (\theta'', e''), [\,] \rangle \mid \Pi \\ \quad \leftharpoondown \Gamma; \langle p, h, (\theta, e), q \rangle \mid \Pi \end{array}$$

$$(\overline{Self}) \quad \Gamma; \langle p, \mathsf{self}(\theta, e) : h, (\theta', e'), q \rangle \mid \Pi \leftharpoondown \Gamma; \langle p, h, (\theta, e), q \rangle \mid \Pi$$

$$(\overline{Sched}) \quad \begin{array}{l} \Gamma; \langle p, h, (\theta, e), \{v, \lambda\} : q \rangle \mid \Pi \leftharpoondown \Gamma \cup (p, \{v, \lambda\}); \langle p, h, (\theta, e), q \rangle \mid \Pi \\ \quad \text{if the topmost } \mathsf{rec}(\ldots) \text{ item in } h \text{ (if any) has the} \\ \quad \text{form } \mathsf{rec}(\theta', e', \{v', \lambda'\}, q') \text{ with } q' \backslash\!\!\backslash \{v', \lambda'\} \neq \{v, \lambda\} : q \end{array}$$

Reversible Core Erlang simulator

- You can experiment with reversible Core Erlang
- A simulator is available at https://github.com/mistupv/rev-erlang
- Also installed in the virtual machine
- Developed by Adrian Palacios

Reversible Core Erlang simulator at work

- You can load an Erlang module
- It is automatically translated into Core Erlang
- You can select any function from the module and specify its parameters
- A starting system is created
- You can simulate its execution forward and backward

Demo time

Controlling Core Erlang

- Normal computation is forward
- We introduce checkpoints
- A checkpoint for an expression *expr* is obtained by replacing *expr* by let X = check(t) in *expr*
- Nondeterministically, a thread may rollback to a past checkpoint
- To ensure causal consistency, rollback is propagated to other threads when needed

Controlled Core Erlang at runtime

- Each thread is equipped with a set of active rollbacks
- If empty, the thread runs forward
- Rollbacks may be:
  - To a checkpoint
  - To the beginning of the thread
  - To the scheduling of a message
- The first form is introduced by the rule

$$(\overline{Undo}) \quad \Gamma; \lfloor \langle p, h, (\theta, e), q \rangle \rfloor_{\Psi} \mid \Pi \leftharpoondown \Gamma; \lfloor \langle p, h, (\theta, e), q \rangle \rfloor_{\Psi \cup \{\#^{\mathtt{t}}_{\mathsf{ch}}\}} \mid \Pi$$
$$\text{if } \mathsf{check}(\theta', e', \mathtt{t}) \text{ occurs in } h, \text{ for some } \theta' \text{ and } e'$$

- The two last forms are used to ensure causal consistency
- When the desired action is undone, the rollback is removed from the set

Controlled Core Erlang backward semantics

$$(\overline{Seq})\quad \Gamma; \lfloor\langle p, \tau(\theta,e)\!:\!h, (\theta',e'), q\rangle\rfloor_\Psi \mid \Pi \leftharpoondown \Gamma; \lfloor\langle p, h, (\theta,e), q\rangle\rfloor_\Psi \mid \Pi$$

$$(\overline{Check})\quad \Gamma; \lfloor\langle p, \mathsf{check}(\theta,e,\mathtt{t})\!:\!h, (\theta',e'), q\rangle\rfloor_\Psi \mid \Pi \leftharpoondown \Gamma; \lfloor\langle p, h, (\theta,e), q\rangle\rfloor_{\Psi\setminus\{\#^{\mathtt{t}}_{\mathsf{ch}}\}} \mid \Pi$$

$$(\overline{Send1})\quad \Gamma\cup\{(p',\{v,\lambda\})\}; \lfloor\langle p, \mathsf{send}(\theta,e,p',\{v,\lambda\})\!:\!h, (\theta',e'), q\rangle\rfloor_\Psi \mid \Pi \leftharpoondown \Gamma; \lfloor\langle p, h, (\theta,e), q\rangle\rfloor_\Psi \mid \Pi$$

$$(\overline{Send2})\quad \begin{array}{l} \Gamma; \lfloor\langle p, \mathsf{send}(\theta,e,p',\{v,\lambda\})\!:\!h, (\theta',e'), q\rangle\rfloor_\Psi \mid \lfloor\langle p', h', (\theta'',e''), q'\rangle\rfloor_{\Psi'} \mid \Pi \\ \leftharpoondown \Gamma; \lfloor\langle p, \mathsf{send}(\theta,e,p',\{v,\lambda\})\!:\!h, (\theta',e'), q\rangle\rfloor_\Psi \mid \lfloor\langle p', h', (\theta'',e''), q'\rangle\rfloor_{\Psi'\cup\{\#^\lambda_{\mathsf{sch}}\}} \mid \Pi \\ \qquad \text{if } (p',\{v,\lambda\}) \text{ does not occur in } \Gamma \text{ and } \#^\lambda_{\mathsf{sch}} \not\in \Psi' \end{array}$$

$$(\overline{Receive})\quad \Gamma; \lfloor\langle p, \mathsf{rec}(\theta,e,\{v,\lambda\},q)\!:\!h, (\theta',e'), q\setminus\!\!\setminus\{v,\lambda\}\rangle\rfloor_\Psi \mid \Pi \leftharpoondown \Gamma; \lfloor\langle p, h, (\theta,e), q\rangle\rfloor_\Psi \mid \Pi$$

$$(\overline{Spawn1})\quad \begin{array}{l} \Gamma; \lfloor\langle p, \mathsf{spawn}(\theta,e,p'')\!:\!h, (\theta,e), q\rangle\rfloor_\Psi \mid \lfloor\langle [\,], p'', (\theta'',e''), [\,]\rangle\rfloor_{\Psi'} \mid \Pi \\ \qquad \leftharpoondown \Gamma; \lfloor\langle p, h, (\theta,e), q\rangle\rfloor_\Psi \mid \Pi \end{array}$$

$$(\overline{Spawn2})\quad \begin{array}{l} \Gamma; \lfloor\langle p, \mathsf{spawn}(\theta,e,p'')\!:\!h, (\theta,e), q\rangle\rfloor_\Psi \mid \lfloor\langle p'', h'', (\theta'',e''), q''\rangle\rfloor_{\Psi'} \mid \Pi \\ \leftharpoondown \Gamma; \lfloor\langle p, \mathsf{spawn}(\theta,e,p'')\!:\!h, (\theta,e), q\rangle\rfloor_\Psi \mid \lfloor\langle p'', h'', (\theta'',e''), q''\rangle\rfloor_{\Psi'\cup\{\#_{\mathsf{sp}}\}} \mid \Pi \\ \qquad \text{if } h'' \neq [\,] \vee q'' \neq [\,] \text{ and } \#_{\mathsf{sp}} \not\in \Psi' \end{array}$$

$$(\overline{Self})\quad \Gamma; \lfloor\langle p, \mathsf{self}(\theta,e)\!:\!h, (\theta',e'), q\rangle\rfloor_\Psi \mid \Pi \leftharpoondown \Gamma; \lfloor\langle p, h, (\theta,e), q\rangle\rfloor_\Psi \mid \Pi$$

$$(\overline{Sched})\quad \begin{array}{l} \Gamma; \lfloor\langle p, h, (\theta,e), \{v,\lambda\}\!:\!q\rangle\rfloor_\Psi \mid \Pi \leftharpoondown \Gamma\cup(p,\{v,\lambda\}); \lfloor\langle p, h, (\theta,e), q\rangle\rfloor_{\Psi\setminus\{\#^\lambda_{\mathsf{sch}}\}} \mid \Pi \\ \qquad \text{if the topmost } \mathsf{rec}(\ldots) \text{ item in } h \text{ (if any) has the} \\ \qquad \text{form } \mathsf{rec}(\theta',e',\{v',\lambda'\},q') \text{ with } q'\setminus\!\!\setminus\{v',\lambda'\} \neq \{v,\lambda\}\!:\!q \end{array}$$

Controlled Core Erlang: proving properties

- One would like to prove properties of controlled Core Erlang
  - E.g., a rollback restores the state of the thread to the one before the selected checkpoint
- If you try to prove this directly, it is a mess
- First, prove standard properties of the uncontrolled semantics (loop lemma, causal consistency…)
- Then use these properties to prove properties of the controlled semantics

Conclusions

The journey is at the beginning

Summary

- Uncontrolled reversibility, for various calculi and languages
- Mechanisms for controlling reversibility
  - In particular using **roll** and checkpionts
- How to avoid looping using alternatives
- Some applications
  - State space exploration
  - Interacting transactions

Future work: uncontrolled reversibility

- Many open questions
- Can we cover full Erlang?
  - Error handling model
- Can we define a really distributed reversible Erlang?
- Can we deal with other languages?
  - Shared memory and complex data structures, classes and objects, ...
- Implementation issues
  - How can we store histories in more efficient ways?
  - How much overhead do we have?
  - Trade-off between efficiency and granularity of reversibility
- Can we have Janus style causal-consistent reversibility?

Future work: controlled reversibility

- Which ways of controlling reversibility are useful?
- Can we exploit reversibility to build high-level programming constructs?
  - Like we did for interacting transactions
  - How to do this in real languages?
  - Checkpoints are not the only option (and need to be refined)
- See Mezzina's course on the use of reversibility in debugging

Future work: applications

- Can we find some killer application for causal-consistent reversibility?
  - One of the current applications? Debugging, biological modelling?
  - Or some areas where reversibility is used, but not causal-consistent reversibility? Simulation, robots?
  - Or something where reversibility has not been used yet?

Future work: beyond causal consistency

- Out of causal order reversibility has been studied and applied, e.g., in biological modelling
- Can we build a coherent theory for it?
- Or can we just use causal-consistent reversibility with weaker causality notions?
  - Can we commute sequential but independent actions? E.g., x=x+1;y=y-1
  - Not concurrent moves commute in the space
- What about actions which are irreversible?
  - How to manage the interaction between reversible and irreversible systems?

Finally

Thanks!

Questions?