Reversible Computing in Concurrent Interacting Systems

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

Plan of the course

1. Motivation
2. Reversible Computing in Sequential Systems
3. Causal-consistent Reversible Computing
4. Reversing Erlang

Motivation

What is reversible computing?

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

- What does it mean to go backward?
- If from state S_1 I go forward to state S_2, then from state S_2 I should be able to go back to state S_1

Why reversible computing in this course?

- Reversible computing can be seen as an emerging programming paradigm
- There are languages based on this paradigm
  - Janus, RFun, …
  - Only academic languages at the moment
  - Somehow related to Prolog
- There are tools, frameworks and libraries based on this paradigm
  - Debuggers (including GDB and Microsoft WinDbg)
  - Optimistic simulators
  - ...

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
  - Quantum computers
  - Green computing
- 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

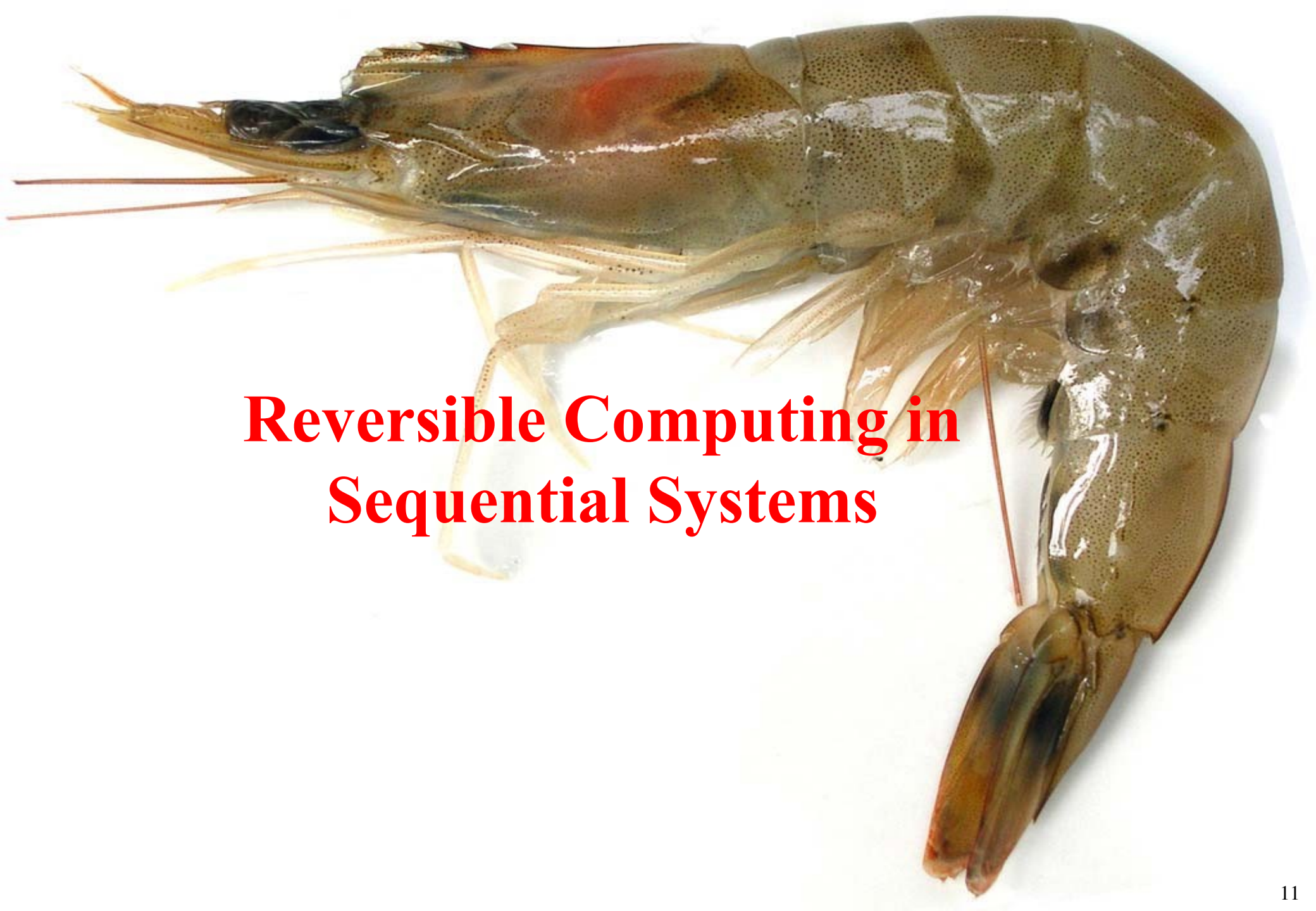
- While exploring a state space towards a solution one may get stuck
- Need to backtrack to find a better solution
- This is the standard mechanism in Prolog
- Also, while looking for an optimal solution one may find a local optimum
- State space exploration much easy in a reversible language
  - No 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



Reversible Computing in Sequential Systems

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 always possible
- Computation steps may cause loss of information
  - This means backward computation is not deterministic
- 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
- 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
- Undoing steps one by one
  - Limiting the language to constructs that are reversible
    - » Featuring only actions that cause no loss of information
    - » Janus approach
  - Adding history information so to make the language reversible
    - » One should save information on the past configurations
    - » X=5 becomes reversible by recording the old value of X

Janus programming language

- Imperative programming language
  - Naturally reversible
  - No history information
- Academic toy language
- Described in
  [Tetsuo Yokoyama, Robert Glück: A Reversible Programming Language and its Invertible Self-Interpreter. PEPM 2007]
- Playground at http://topps.diku.dk/pirc/?id=janusP

Janus essence

- Reversible assignment: x += e (also -= , ^=)
- Reversible conditional
  if e_1 then s_1 else s_2 fi e_2
  Exit assertion must hold if coming from then, must not hold if coming from else
- One can call procedures, and uncall them
  - Uncall invokes the inverse semantics
- One can declare local variables (initially 0)
  - Variables need to be undeclared – must be 0 before undeclaration
- While loops exist, but are quite complex

Janus syntax

- p ::= (procedure id(par*) s)$^{+}$
  s ::= x$\oplus$=e | x[e]$\oplus$=e | if e then s else s fi e |
  call id(e*) | uncall id(e*) | skip | s s |
  local int v | delocal int v |
  local int v[c] | delocal int v[c] |
  int v | int v[c]
- The program starts from procedure main
- Only main can declare global variables (no need to undeclare them)
- All variables passed by reference

Let us play

- We want to write a program that given n computes the n-th pair of Fibonacci number
  - A procedure fib(int n, int x1, int x2)
  - To be invoked with x1=x2=0

Janus features

- Programs can be executed in both directions
  - If I write a procedure, I get the reverse one for free
  - Reverse execution as efficient as forward one
- The inverse program can be derived with a simple local transformation
- $x \oplus= e \leftrightarrow x \oplus= e$

  $\text{if } e_1 \text{ then } s_1 \text{ else } s_2 \text{ fi } e_2 \leftrightarrow \text{if } e_2 \text{ then } s_1 \text{ else } s_2 \text{ fi } e_1$

  $\text{call id(e*)} \leftrightarrow \text{uncall id(e*)}$

  $\text{skip} \leftrightarrow \text{skip}$

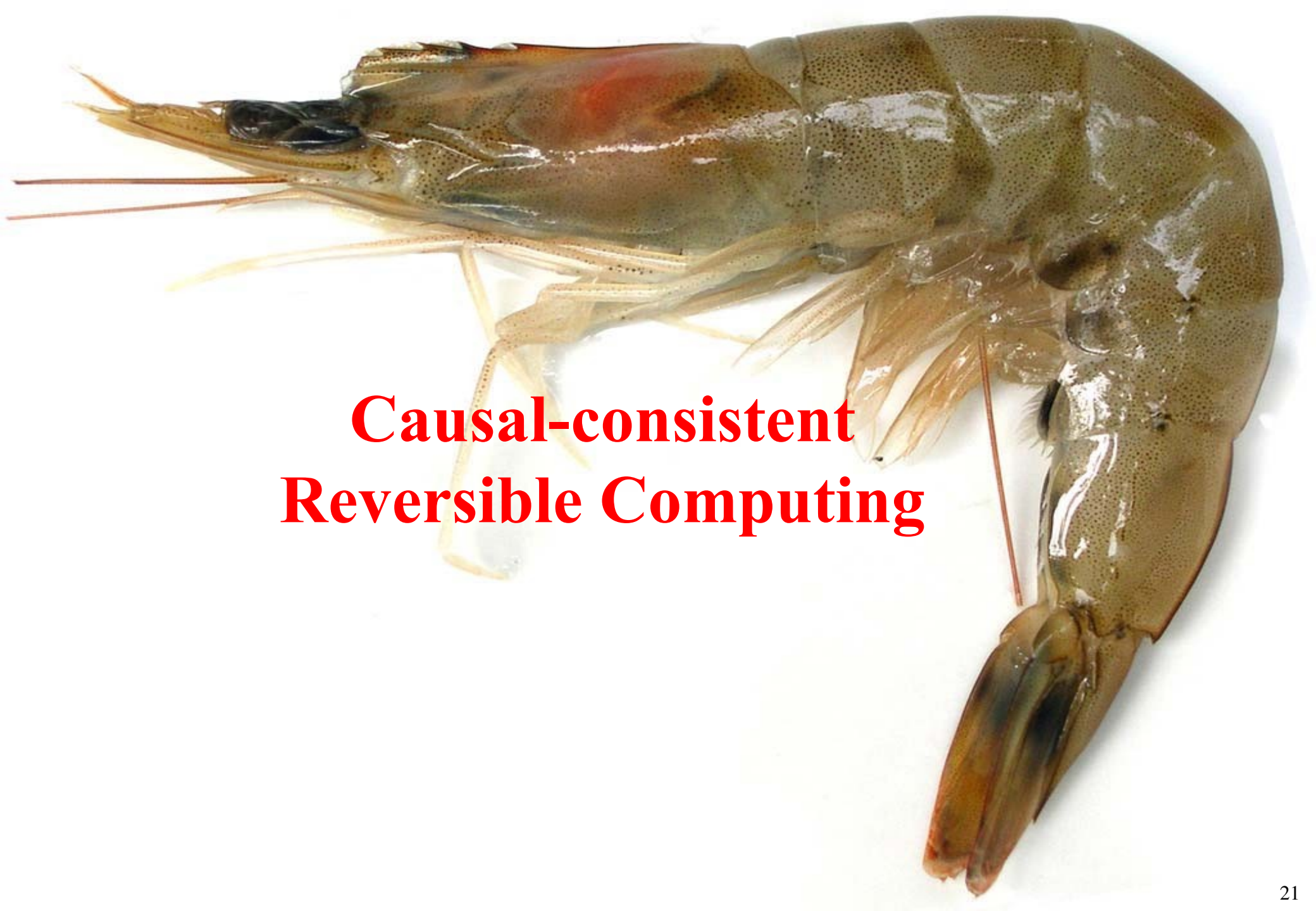
  $s_1\ s_2 \leftrightarrow s_2\ s_1$

  $\text{local int v} \leftrightarrow \text{delocal int v}$

Janus limitations

- Only bijective functions can be programmed
  - Any function can be made bijective
    $x \rightarrow f(x) \quad => \quad x \rightarrow (x,f(x))$

- Runtime exceptions possible
  - Assertions in conditionals (and while loops)
  - Delocal of variables
- Difficult to program with
  - Bijective functions need to be obtained by composing bijective functions
  - Try with factorial



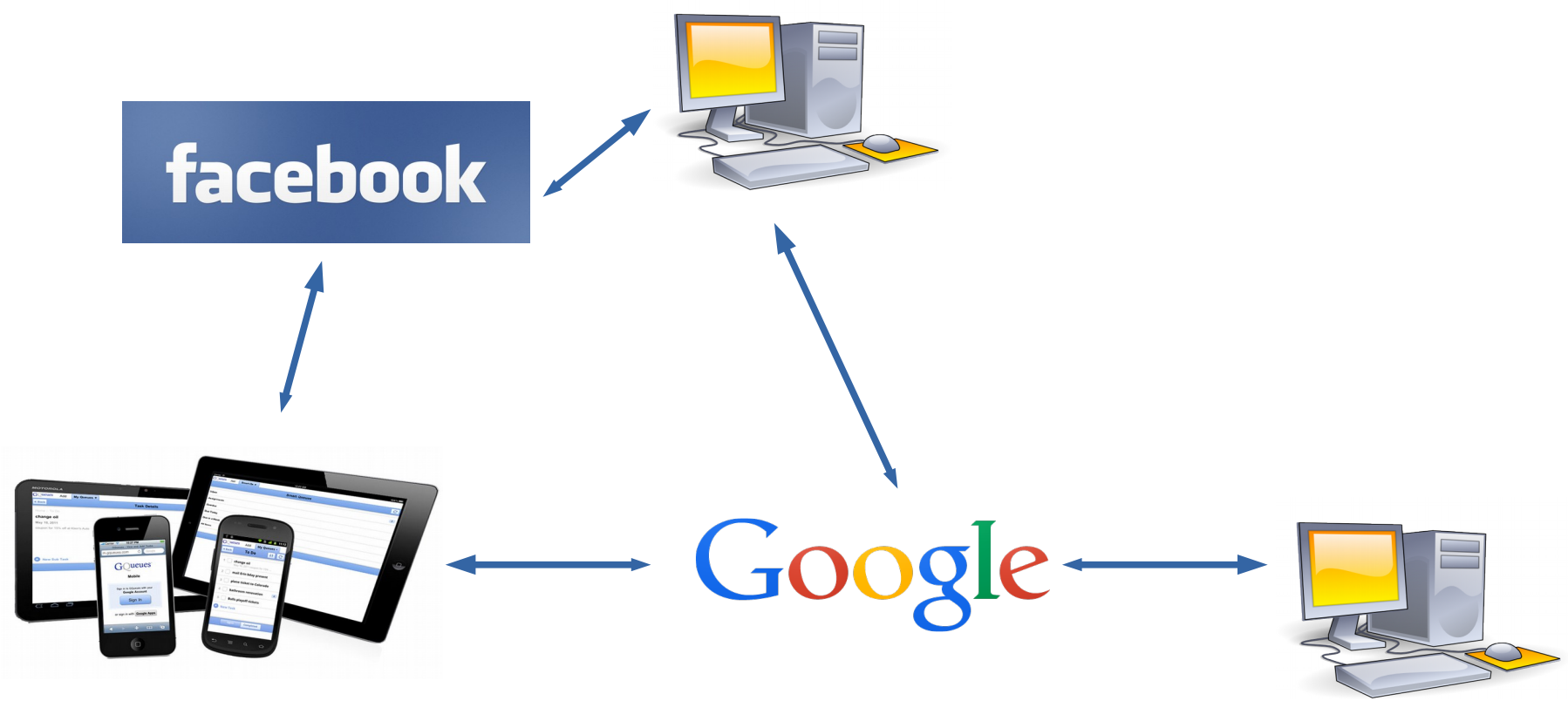
Causal-consistent Reversible Computing

Reversibility for concurrent interacting systems

- Janus is a sequential language
- Reversible debugging for sequential programs is well understood
  - See, e.g., GDB
- Reversible computation in a concurrent setting is far less explored
- We are particularly interested in this setting
- No Janus style approach available yet, we will use history information

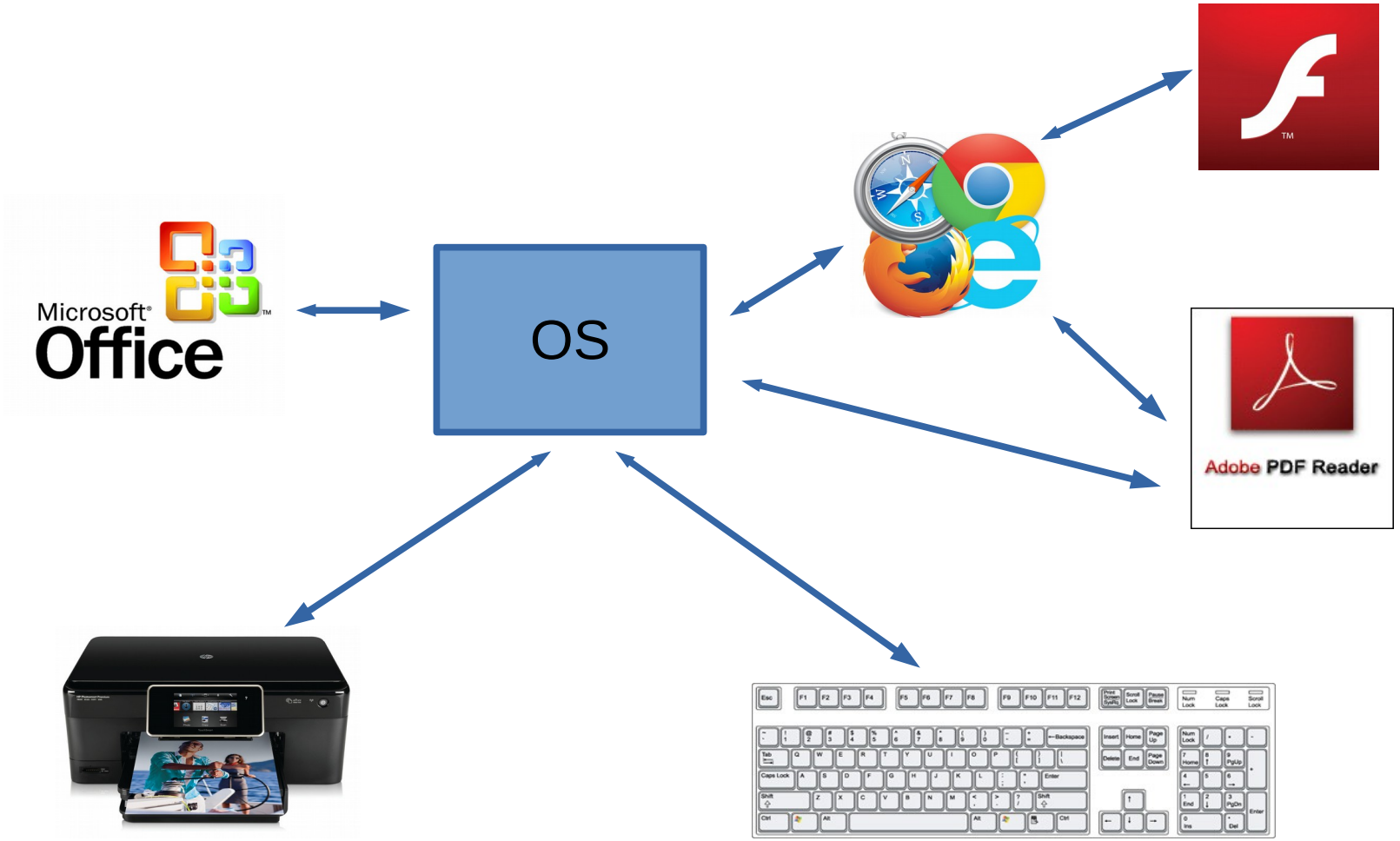
Concurrency and interaction everywhere

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



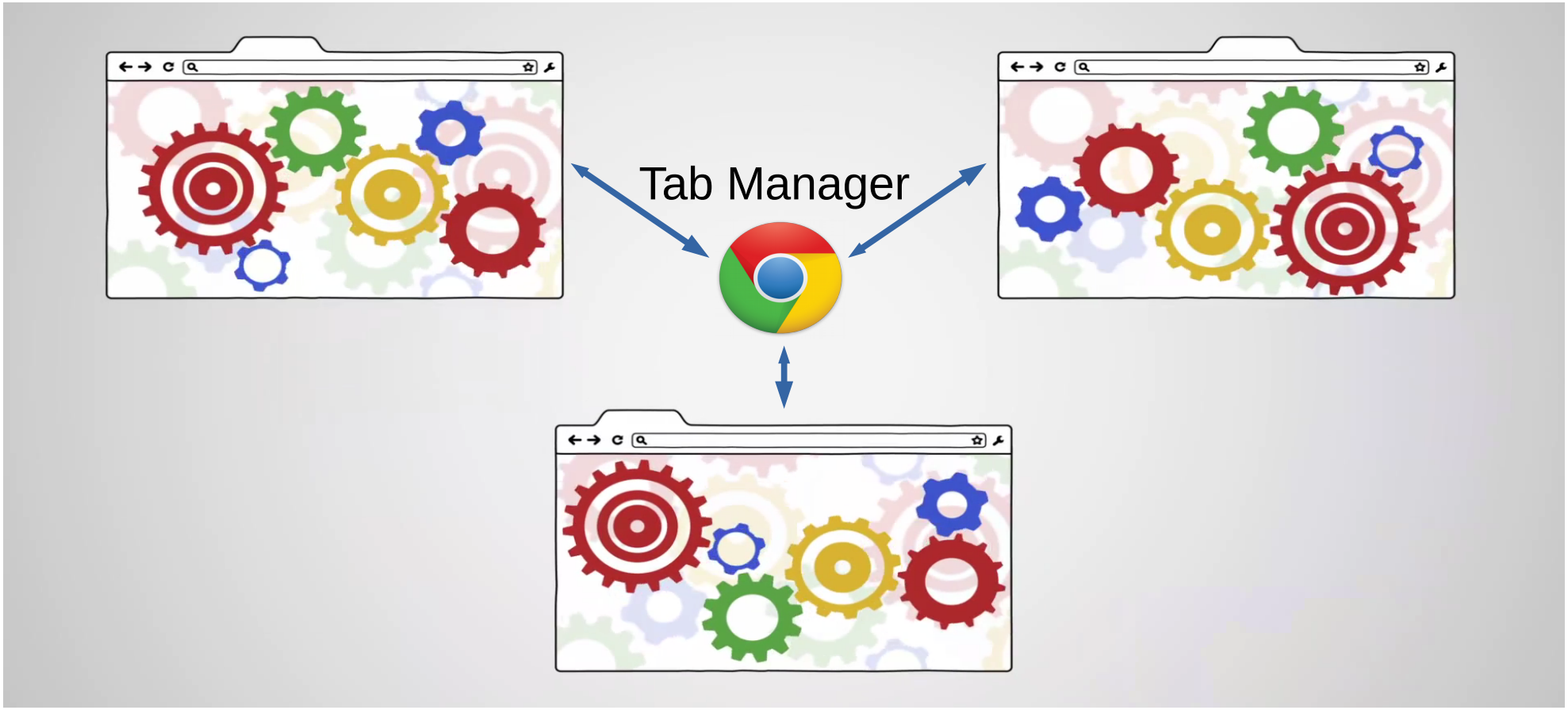
Concurrency and interaction everywhere

- Your computer features concurrency and interaction



Concurrency everywhere

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

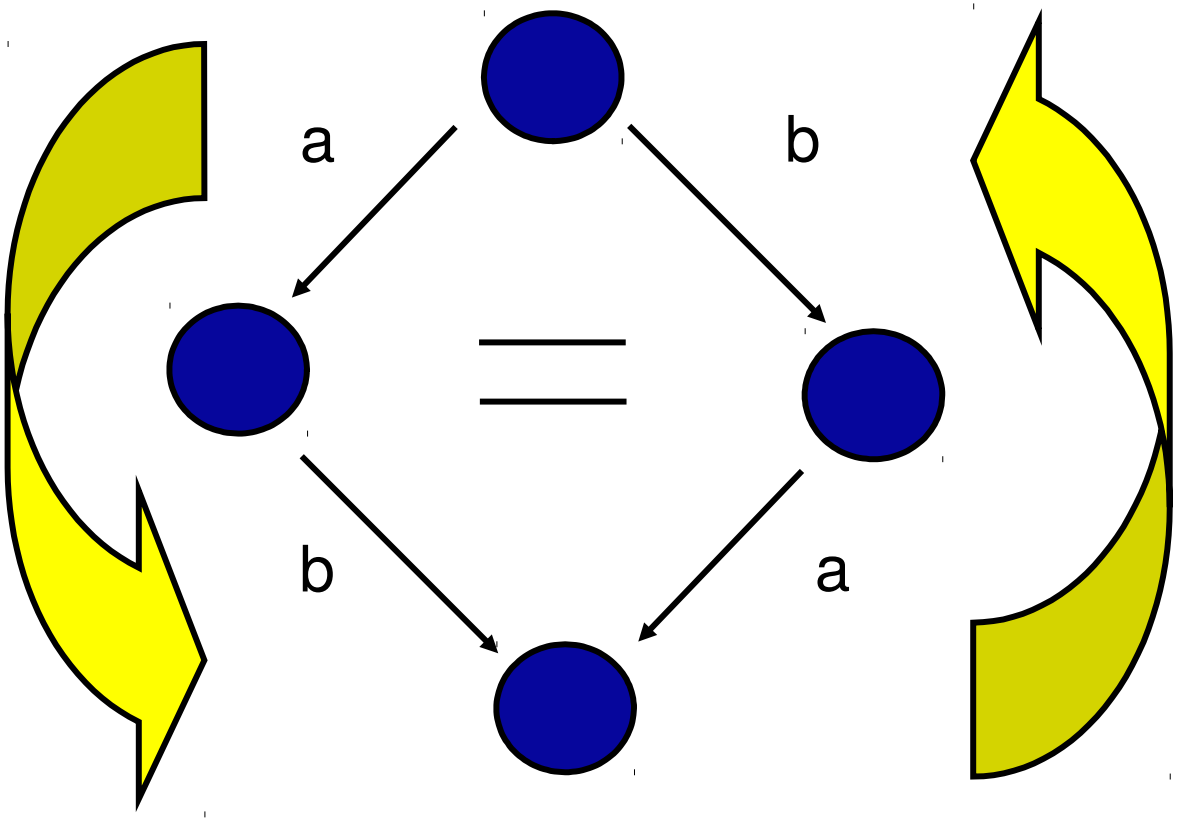


Reversibility and concurrency

- 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]

Causal-consistent reversibility



Why do 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)

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
- If we have two messages with the same value we need to distinguish them

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

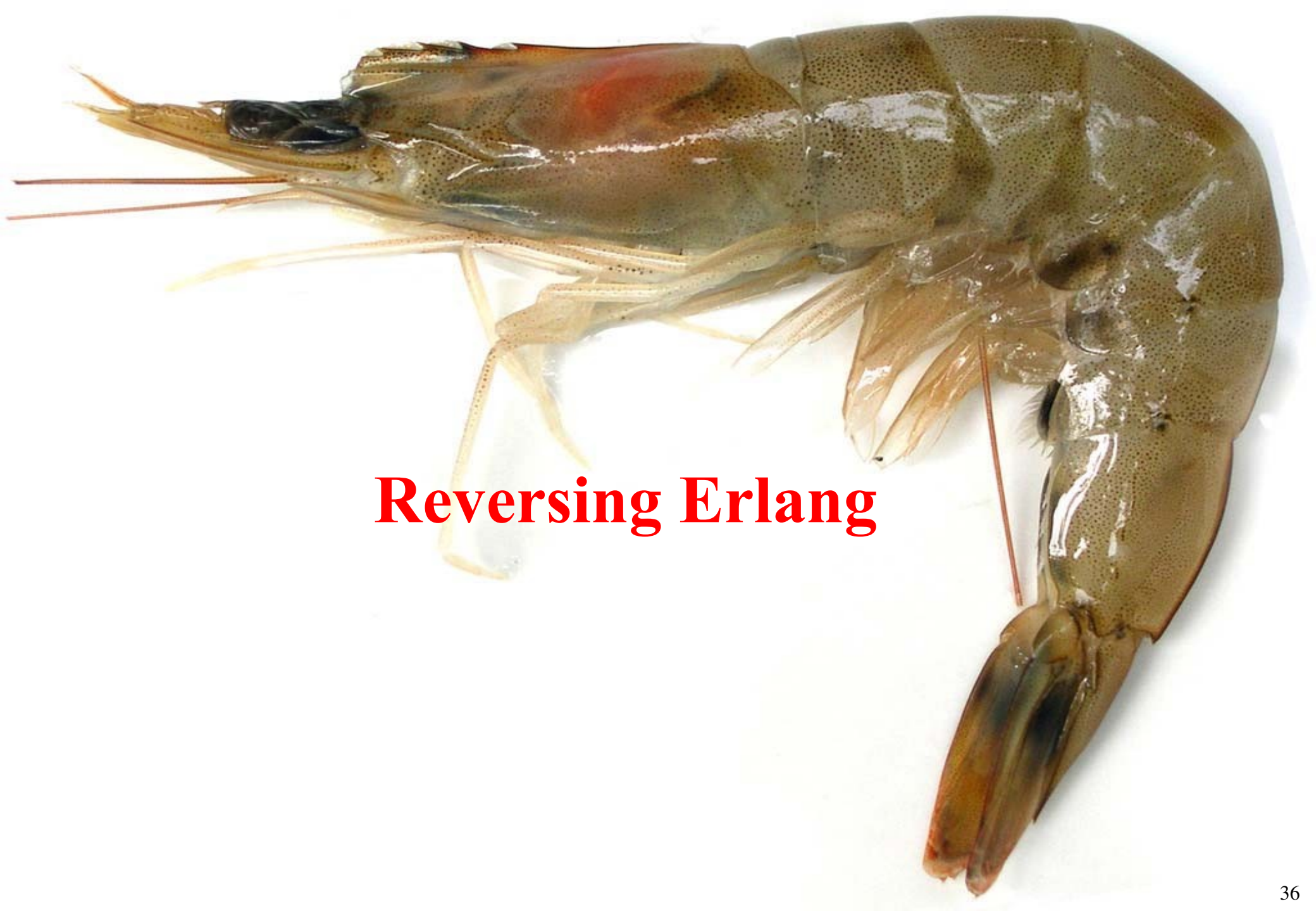
  endif
- Two possible computations
- The two possible computations lead to the same state
- From the causal consistency theorem we know that we need history information to distinguish them
  - At least we should trace the chosen branch

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
  - Shows that from an initial state we can reach only forward-reachable states

What we know

- We have some idea about how to take a concurrent language and define its causal-consistent reversible extension
  - We need to satisfy the Loop Lemma
  - We need to satisfy the Causal Consistency Theorem
- More technicalities are needed to do it
- We consider the case of Erlang

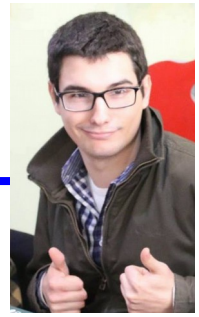


Reversing Erlang

Why Erlang?

- Real setting to test our ideas
- Built-in primitives for communication
- Simple concurrency model: the actor model
  - Enables a clear separation between few concurrency-relevant constructs and sequential constructs
  - We can deal with sequential constructs in a uniform way

CauDEr



- We illustrate the ideas that we present with CauDEr
- CauDEr is a Causal-consistent Debugger for Erlang
- Provides back and forward causal-consistent execution of Erlang systems
- Available at https://github.com/mistupv/cauder
- Developed by Adrian Palacios (Universitat Politecnica de Valencia)

From Erlang to Core Erlang

- Erlang is compiled into Core Erlang
  - More constrained and easy to deal with, yet equally expressive
  - We will consider Core Erlang
  - CauDEr translates Erlang programs into Core Erlang and works on the translation
- 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* = $fun_1, ..., fun_n$
- *fun* ::= *fname* = fun $(X_1, ..., X_n) \rightarrow expr$
- *fname* ::= *Atom/Integer*
- *lit* ::= *Atom* | *Integer* | *Float* | [ ]
- *expr* ::= *Var* | *lit* | *fname* | $[expr_1|expr_2]$ | $\{expr_1, ..., expr_n\}$
  | call *expr* $(expr_1, ..., expr_n)$ | apply *expr* $(expr_1, ..., expr_n)$
  | case *expr* of $clause_1, ..., clause_m$ end
  | let *Var* = $expr_1$ in $expr_2$
  | receive $clause_1, ..., clause_n$ end
  | spawn(*expr*, $[expr_1, ..., expr_n]$) | $expr_1$! $expr_2$ | self()
- *clause* ::= *pat* when $expr_1 \rightarrow expr_2$
- *pat* ::= *Var* | *lit* | $[pat_1|pat_2]$ | $\{pat_1, ..., pat_n\}$

Core Erlang at runtime

- At runtime a Core Erlang system is composed by:
  - A global mailbox Γ: messages travelling in the network
  - A set of processes
    - Each process has a unique name p, a state θ, an expression under evaluation e, and a queue of waiting messages q

Core Erlang actions

- Each evaluation step performed by a process is an action
- We have a further action, Sched, taking place when a message is moved from the global mailbox to the target local mailbox
  - Always enabled, if there is a message

Core Erlang reversible semantics

- Preliminary version in
  [Naoki Nishida, Adrián Palacios, Germán Vidal:
  A Reversible Semantics for Erlang. LOPSTR 2016]
- We add unique identifiers λ to messages
- We add histories *h* to processes 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

Causality

- Causal-consistent reversibility is based on causality
- We need to understand whether two 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 processes
  - In the same process 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 process are always in conflict

CauDEr as reversible simulator

- 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
- Using the manual modality only actions with no causal dependences can be undone

Controlling reversibility in Core Erlang

- When debugging you observe a misbehavior
  - E.g., variable x has value 5 and you expected 6
- You want to understand where the misbehavior comes from
- You want to undo the action causing it
  - E.g., the last assignment to variable x
- More in general, you want to undo an arbitrary past action

Undoing arbitrary past actions

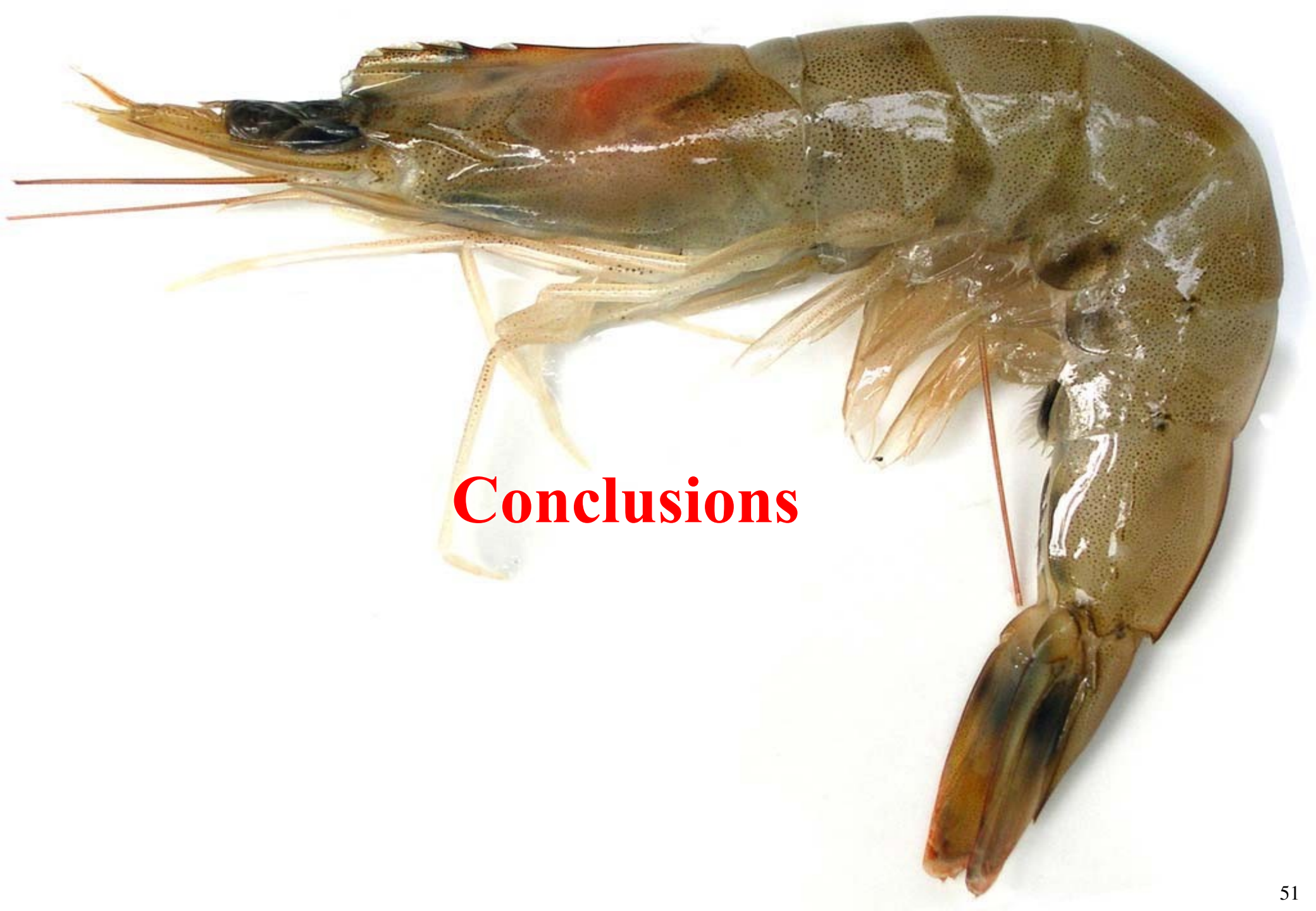
- According to causal-consistent reversibility, you need first to undo all its consequences
  - We want to undo only them for minimality
- One can put the desired process in rollback mode, undoing steps in reverse order
- When a dependency is found, rollback mode is propagated
  - E.g., if the send of a message must be undone, its receive (and sched) must be undone beforehand
- When the dependency is undone, rollback in the original process can restart

Undoing arbitrary past actions in CauDEr

- CauDEr provides causal-consistent undo of arbitrary past actions
- Different forms, according to the different forms of possible misbehaviors
- It reports information on which actions have been undone
  - Useful to spot bugs due to missing or undesired dependencies

Causal-consistent reversible debugging

- When a misbehavior is found, go back following its causes till you find the bug
- This may involve exploring multiple processes
- The debugger tells you on which processes you need to focus



Conclusions

Summary

- Reversibility has multiple application areas
- Sequential reversibility: recursively undo the last step
- Janus supports this without history information, but this is not easy to use
- Causal-consistent reversibility: undo any action, provided that its dependences (if any) are undone beforehand
- Causal-consistent reversibility suitable for debugging

Future work

- Many open questions
- Can we cover full Erlang?
  - Error handling model
- 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?