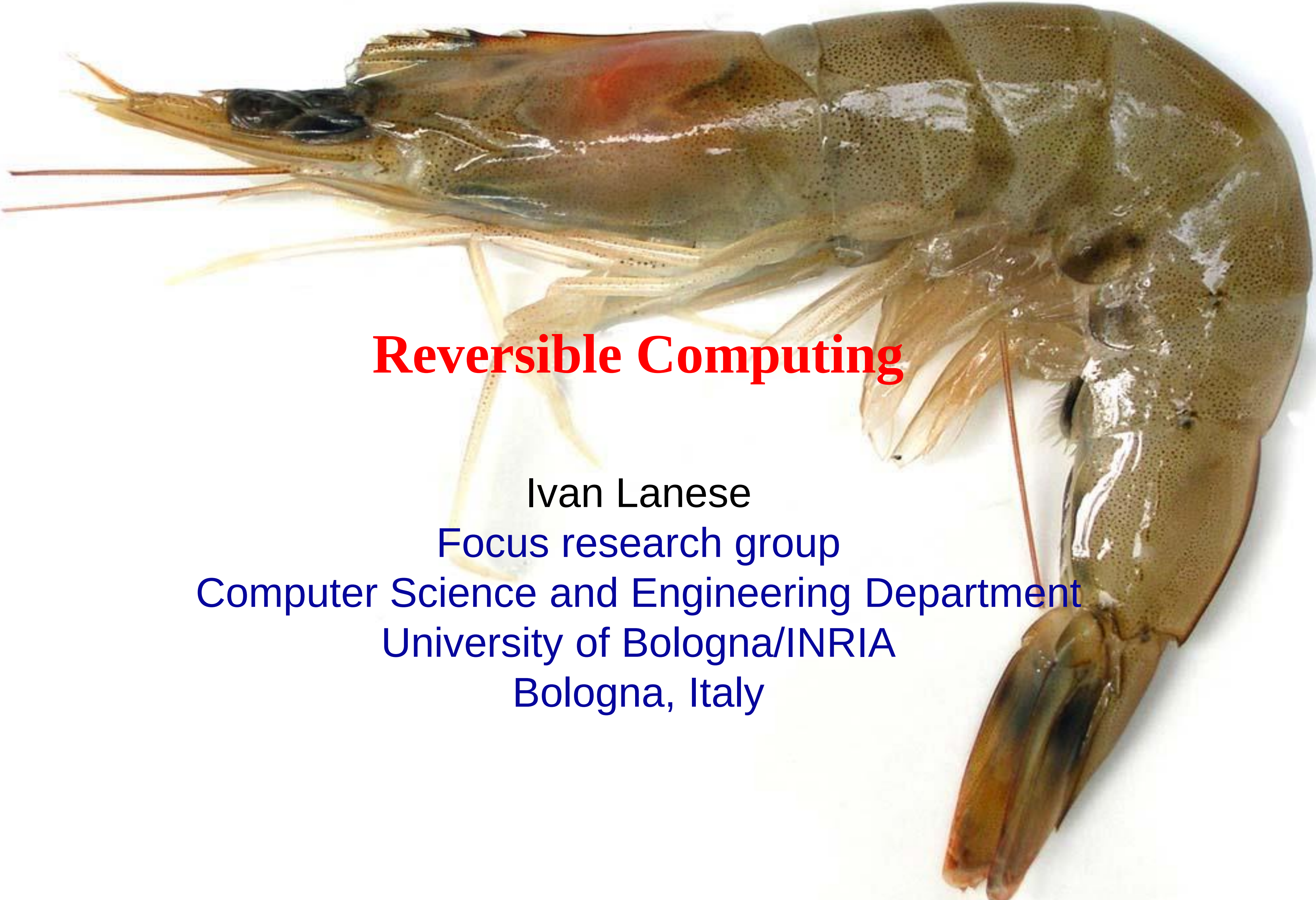
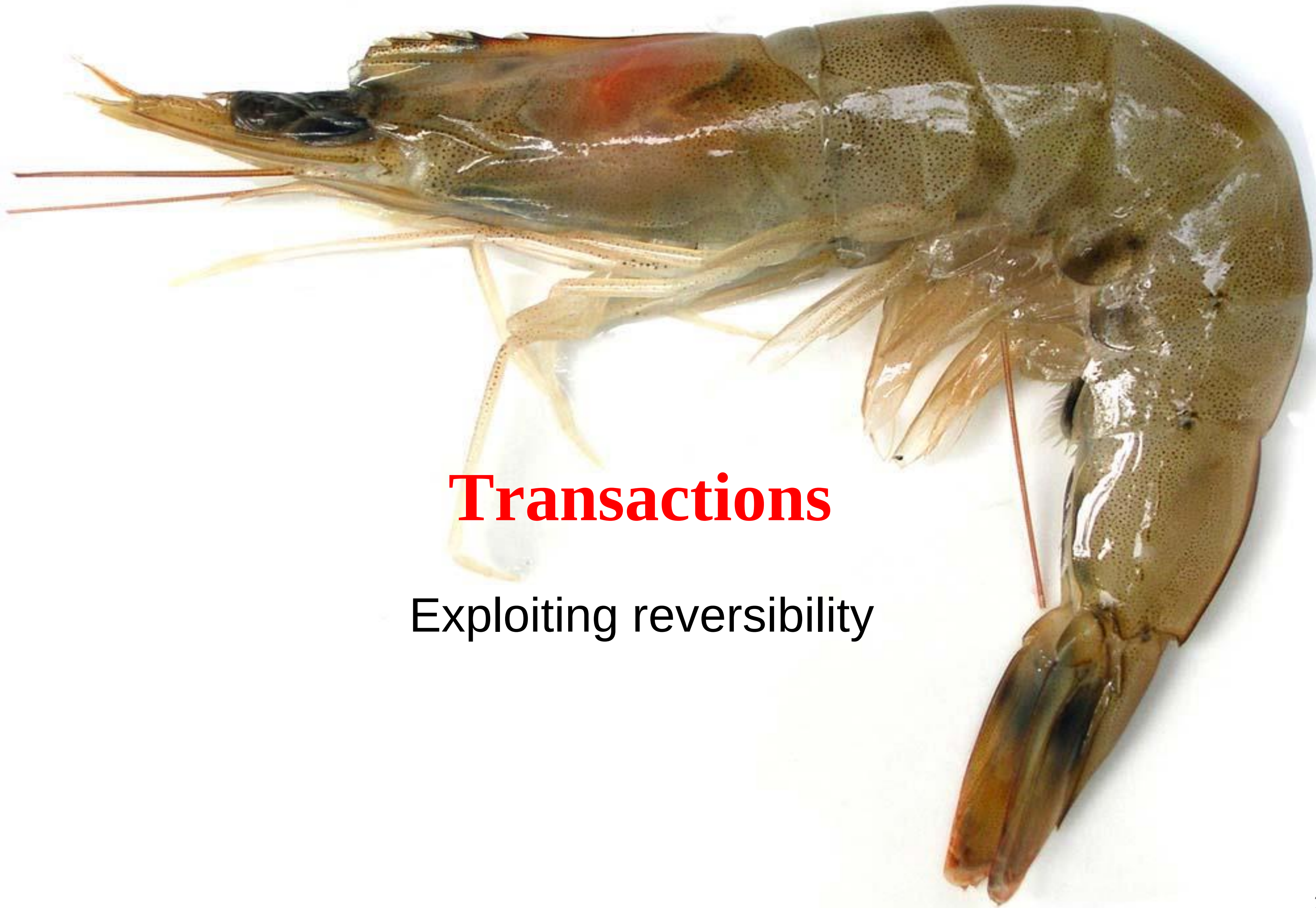


Reversible Computing

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



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
- In case of commit, only P remains
- In case of abort, the effects of P are undone, and only Q remains

Transactions in croll-π

- $[\![[P,Q]_\gamma]\!] =$
  $\nu a\ \nu c\ a\langle 0\rangle\%c\langle 0\rangle\%0 \mid a(X) \rhd_\gamma [\![P]\!] \mid c(Y) \rhd [\![Q]\!]$
- Abort is $roll\ \gamma$
- Commit is implicit: if there is no $roll\ \gamma$ then the compensation and the transaction machinery become garbage
- We simulate the transaction boundary with causality tracking
- Atomic transaction: P is executed all or nothing
  - If P aborts all its effects are undone
- Not isolated

Interacting transactions in TransCCS

- Syntax

$$P ::= \bar{a} \mid a.P \mid P|Q \mid \nu a\, P \mid 0 \mid [P \rhd_k Q] \mid co\ k$$

- Semantics

$$\bar{a} \mid a.P \rightarrow P$$

$$[P \rhd_k Q] \mid R \rightarrow [P \mid R \rhd_k Q \mid R] \quad \text{if } k \notin fn(R)$$

$$[P \mid co\ k \rhd_k Q] \rightarrow P$$

$$[P \rhd_k Q] \rightarrow Q$$

- Processes from the environment moved into the transaction to interact with it
  - Saved also in the compensation
- Implicit abort, explicit commit

Example: transactions interacting

- $[\bar{a} \rhd_k Q] \mid [a.P \rhd_h Q'] \to$
  $$\big[a.P \mid [\bar{a} \rhd_k Q] \rhd_h Q' \mid [\bar{a} \rhd_k Q]\big] \to$$
  $$\big[[\bar{a} \mid a.P \rhd_k Q \mid a.P] \rhd_h Q' \mid [\bar{a} \rhd_k Q]\big] \to$$
  $$\big[[P \rhd_k Q \mid a.P] \rhd_h Q' \mid [\bar{a} \rhd_k Q]\big]$$
- 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

- $\bar{a} \mid a.R \mid [P \rhd_k Q] \rightarrow$

$$[P \mid \bar{a} \mid a.R \rhd_k Q \mid \bar{a} \mid a.R] \rightarrow$$
$$[P \mid R \;\; \rhd_k Q \mid \bar{a} \mid a.R] \rightarrow$$
$$Q \mid \bar{a} \mid a.R$$

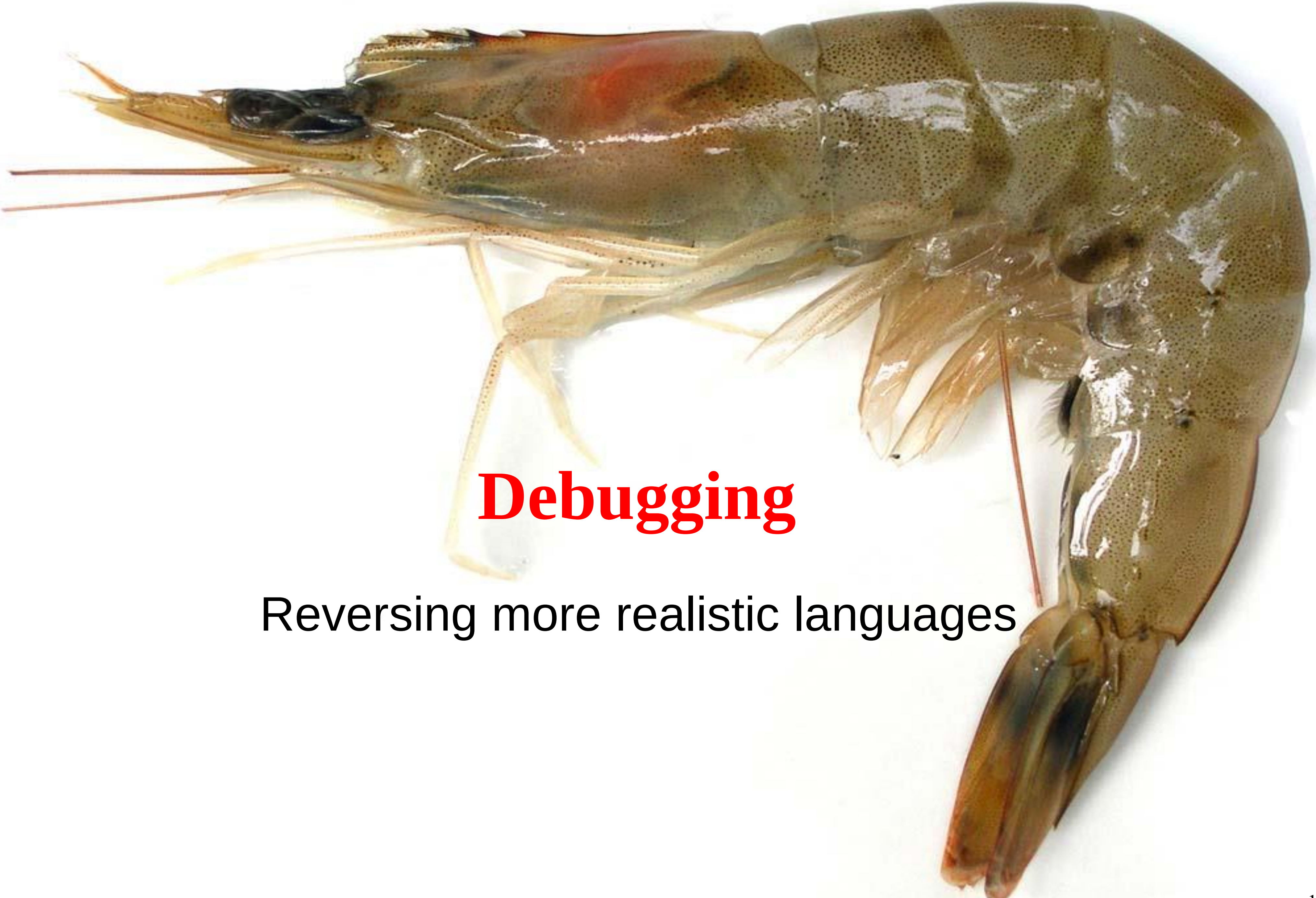
- Why undoing the synchronization on a?
- No reason for it to occur inside the transaction

Interacting transactions in croll-π

- $[\![[P \rhd_l Q]]\!] = [\nu l\ [\![P]\!] \mid l\langle roll\ \gamma\rangle \mid l(X) \rhd X, [\![Q]\!]]_\gamma$
- We simulate the automatic abort with a $roll$ that can be enabled at any moment
- $[\![co\ l]\!] = l(X) \rhd 0$
- A commit disables the abort

Comparing the two approaches

- $[\![[P \rhd_l Q]]\!] = [\nu l\ [\![P]\!] \mid l\langle roll\ \gamma\rangle \mid l(X) \rhd X, [\![Q]\!]]_\gamma$
- In croll-π only reductions depending on the transaction body are undone
  - In TransCCS other reductions are undone, and then redone
  - Difference due to a more precise causality tracking
- 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 $l\langle 0\rangle$ as commit



Debugging

Reversing more realistic languages

Debugging

- Going back and forward can help in finding a bug
- Some commercial debuggers provide the command "step back" in a sequential setting
  - For instance, gcc
- Our theory enables the definition of step back in a concurrent setting
  - The user specifies the thread to step back
  - Only threads which have no active consequences can step back
- Are there other commands we may add to a debugger to help the programmer to debug concurrent applications?
  - Based on our reversibility techniques

Which language to debug?

- No one programs in CCS or HOπ
- We would be very happy to build a debugger for Java, C++ or Erlang
  - For now, this requires too much effort
- We want to experiment on a simple programming language
  - Concurrent
  - Sharing features with more widespread languages
  - With a formal semantics
  - Sharing features with the calculi we can reverse
- We have chosen μOz

μOz

- A kernel language of Oz
  [P. Van Roy and S. Haridi. Concepts, Techniques and Models of Computer Programming. MIT Press, 2004]
- Oz is at the base of the Mozart language
- Higher-order language
  - Procedures can be communicated
- Thread-based concurrency
- Asynchronous communication via ports
- Variables are always created fresh and never modified
- Shared memory
  - Variable names are sent, not their content

μOz syntax

- S ::= [Statements]
  - skip [Empty statement]
  - S_1 S_2 [Sequence]
  - let x = v in S end [Variable declaration]
  - if x then S_1 else S_2 end [Conditional]
  - thread S end [Thread creation]
  - let x=c in S end [Procedure declaration]
  - {x x_1 … x_n} [Procedure call]
  - let x=Newport in S end [Port creation]
  - {Send x y} [Send]
  - let x ={Receive y} in S end [Receive]
- c ::= proc {x_1 … x_n} S end

μOz semantics

- Semantics defined by a stack-based abstract machine
- The abstract machine exploits a run-time syntax
- Each thread is a stack of instructions
  - The starting program is inserted into a stack
  - Thread creation creates new stacks
- Procedures are stored as closures
- Ports are queues of variables
- Semantics closed under
  - Contexts (for both code and state)
  - Structural congruence

μOz semantics: rules

R:skp $$\frac{\langle \textbf{skip}\ T\rangle}{0} \Bigg\| \frac{T}{0}$$

R:var $$\frac{\langle \textbf{let}\ x = v\ \textbf{in}\ S\ \textbf{end}\ T\rangle}{0} \Bigg\| \frac{\langle S\{^{x'}/_{x}\}\ T\rangle}{x' = v} \text{ if } x' \text{ fresh}$$

R:npr $$\frac{\langle \textbf{let}\ x = c\ \textbf{in}\ S\ \textbf{end}\ T\rangle}{0} \Bigg\| \frac{\langle S\{^{x'}/_{x}\}\ T\rangle}{x' = \xi \parallel \xi : c} \text{ if } x', \xi \text{ fresh}$$

R:npt $$\frac{\langle \textbf{let}\ x = \texttt{NewPort}\ \textbf{in}\ S\ \textbf{end}\ T\rangle}{0} \Bigg\| \frac{\langle S\{^{x'}/_{x}\}\ T\rangle}{x' = \xi \parallel \xi : \bot} \text{ if } x', \xi \text{ fresh}$$

R:if1 $$\frac{\langle \textbf{if}\ x\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \textbf{end}\ T\rangle}{x = \textbf{true}} \Bigg\| \frac{\langle S_1\ T\rangle}{x = \textbf{true}}$$

R:nth $$\frac{\langle \textbf{thread}\ S\ \textbf{end}\ T\rangle}{0} \Bigg\| \frac{T \parallel \langle S\ \langle\rangle\rangle}{0}$$

R:pc $$\frac{\langle \{\ x\ x_1 \ldots x_n\ \}\ T\rangle}{x = \xi \parallel \xi : \textbf{proc}\ \{\ y_1 \ldots y_n\ \}\ S\ \textbf{end}} \Bigg\| \frac{\langle S\{^{x_1}/_{y_1}\} \ldots \{^{x_n}/_{y_n}\}\ T\rangle}{x = \xi \parallel \xi : \textbf{proc}\ \{\ y_1 \ldots y_n\ \}\ S\ \textbf{end}}$$

R:snd $$\frac{\langle \{\ \texttt{Send}\ x\ y\ \}\ T\rangle}{x = \xi \parallel \xi : Q} \Bigg\| \frac{T}{x = \xi \parallel \xi : y; Q}$$

R:rcv $$\frac{\langle \textbf{let}\ x = \{\ \texttt{Receive}\ y\ \}\ \textbf{in}\ S\ \textbf{end}\ T\rangle}{y = \xi \parallel \xi : Q; z \parallel z = w} \Bigg\| \frac{\langle S\{^{x'}/_{x}\}\ T\rangle}{y = \xi \parallel \xi : Q \parallel z = w \parallel x' = w} \text{ if } x' \text{ fresh}$$

μOz reversible semantics

- We give unique names to threads
- We add histories to threads to remember past actions
- We add a delimiter to record when scopes end
  - For let
  - For procedure body
  - For if-then-else
- Ports have histories too
  - Should record also sender and receiver of each message
  - We do not want to change the order of communications

μOz reversible semantics: forward rules

R:fw:skp
$$\frac{t[H]\langle \textbf{skip}\ C\rangle}{0} \,\Bigg\|\, \frac{t[H\ \textbf{skip}]C}{0}$$

R:fw:var
$$\frac{t[H]\langle \textbf{let}\ x = v\ \textbf{in}\ S\ \textbf{end}\ C\rangle}{0} \,\Bigg\|\, \frac{t[H\ *x']\langle S\{^{x'}/_{x}\}\ \langle \textbf{esc}\ C\rangle\rangle}{x' = v} \text{ if } x' \text{ fresh}$$

R:fw:npr
$$\frac{t[H]\langle \textbf{let}\ x = c\ \textbf{in}\ S\ \textbf{end}\ C\rangle}{0} \,\Bigg\|\, \frac{t[H\ *x']\langle S\{^{x'}/_{x}\}\ \langle \textbf{esc}\ C\rangle\rangle}{x' = \xi \parallel \xi : c} \text{ if } x', \xi \text{ fresh}$$

R:fw:npt
$$\frac{t[H]\langle \textbf{let}\ x = \texttt{NewPort}\ \textbf{in}\ S\ \textbf{end}\ C\rangle}{0} \,\Bigg\|\, \frac{t[H\ *x']\langle S\{^{x'}/_{x}\}\ \langle \textbf{esc}\ C\rangle\rangle}{x' = \xi \parallel \xi : \bot|\bot} \text{ if } x', \xi \text{ fresh}$$

R:fw:if1
$$\frac{t[H]\langle \textbf{if}\ x\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \textbf{end}\ C\rangle}{x = \textbf{true}} \,\Bigg\|\, \frac{t[H\ \textbf{if}(x)S_2]\langle S_1\ \langle \textbf{esc}\ C\rangle\rangle}{x = \textbf{true}}$$

R:fw:nth
$$\frac{t[H]\langle \textbf{thread}\ S\ \textbf{end}\ C\rangle}{0} \,\Bigg\|\, \frac{t[H\ *t']C \parallel t'[\bot]\langle S\ \langle\rangle\rangle}{0} \text{ if } t' \text{ fresh}$$

R:fw:pc
$$\frac{t[H]\langle\{\ x\ (x_i)_1^n\ \}\ C\rangle}{x = \xi \parallel \xi : \textbf{proc}\ \{\ (y_i)_1^n\ \}\ S\ \textbf{end}} \,\Bigg\|\, \frac{t[H\ \{\ x\ (x_i)_1^n\ \}]\langle S(\{^{x_i}/_{y_i}\})_1^n\ \langle \textbf{esc}\ C\rangle\rangle}{x = \xi \parallel \xi : \textbf{proc}\ \{\ (y_i)_1^n\ \}\ S\ \textbf{end}}$$

R:fw:snd
$$\frac{t[H]\langle\{\ \texttt{Send}\ x\ y\ \}\ C\rangle}{x = \xi \parallel \xi : K|K_h} \,\Bigg\|\, \frac{t[H\ \uparrow x]C}{x = \xi \parallel \xi : t{:}y; K|K_h}$$

R:fw:rcv
$$\frac{t[H]\langle \textbf{let}\ y = \{\ \texttt{Receive}\ x\ \}\ \textbf{in}\ S\ \textbf{end}\ C\rangle}{\theta \parallel \xi : K; t'{:}z|K_h} \,\Bigg\|\, \frac{t[H\ \downarrow x(y')]\langle S\{^{y'}/_{y}\}\ \langle \textbf{esc}\ C\rangle\rangle}{\theta \parallel \xi : K|t'{:}z, t; K_h \parallel y' = w}$$
$$\text{if } y' \text{ fresh } \wedge\ \theta \triangleq x = \xi \parallel z = w$$

R:fw:scp
$$\frac{t[H]\langle \textbf{esc}\ C\rangle}{0} \,\Bigg\|\, \frac{t[H\ \textbf{esc}]C}{0}$$

μOz reversible semantics: backward rules

R:bk:skp
$$\frac{t[H\ \mathbf{skip}]C}{0} \;\Bigg\|\; \frac{t[H]\langle \mathbf{skip}\ C\rangle}{0}$$

R:bk:var
$$\frac{t[H\ \ *x]\langle S\ \langle \mathbf{esc}\ C\rangle\rangle}{x = v} \;\Bigg\|\; \frac{t[H]\langle \mathbf{let}\ x = v\ \mathbf{in}\ S\ \mathbf{end}\ C\rangle}{0}$$

R:bk:npr
$$\frac{t[H\ \ *x]\langle S\ \langle \mathbf{esc}\ C\rangle\rangle}{x = \xi \parallel \xi : c} \;\Bigg\|\; \frac{t[H]\langle \mathbf{let}\ x = c\ \mathbf{in}\ S\ \mathbf{end}\ C\rangle}{0}$$

R:bk:npt
$$\frac{t[H\ \ *x]\langle S\ \langle \mathbf{esc}\ C\rangle\rangle}{x = \xi \parallel \xi : \bot|\bot} \;\Bigg\|\; \frac{t[H]\langle \mathbf{let}\ x = \texttt{NewPort}\ \mathbf{in}\ S\ \mathbf{end}\ C\rangle}{0}$$

R:bk:if1
$$\frac{t[H\ \mathbf{if}(x)S_2]\langle S_1\ \langle \mathbf{esc}\ C\rangle\rangle}{x = \mathbf{true}} \;\Bigg\|\; \frac{t[H]\langle \mathbf{if}\ x\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2\ \mathbf{end}\ C\rangle}{x = \mathbf{true}}$$

R:bk:nth
$$\frac{t[H\ \ *t']C \parallel t'[\bot]\langle S\ \langle\rangle\rangle}{0} \;\Bigg\|\; \frac{t[H]\langle \mathbf{thread}\ S\ \mathbf{end}\ C\rangle}{0}$$

R:bk:pc
$$\frac{t[H\ \{\ x\ (x_i)_1^n\ \}]\langle S\ \langle \mathbf{esc}\ C\rangle\rangle}{0} \;\Bigg\|\; \frac{t[H]\langle\{\ x\ (x_i)_1^n\ \}\ C\rangle}{0}$$

R:bk:snd
$$\frac{t[H\ \ \uparrow x]C}{x = \xi \parallel \xi : t\!:\!y; K|K_h} \;\Bigg\|\; \frac{t[H]\langle\{\ \texttt{Send}\ x\ y\ \}\ C\rangle}{x = \xi \parallel \xi : K|K_h}$$

R:bk:rcv
$$\frac{t[H\ \ \downarrow x(z)]\langle S\ \langle \mathbf{esc}\ C\rangle\rangle}{z = w \parallel x = \xi \parallel \xi : K|t'\!:\!y, t; K_h} \;\Bigg\|\; \frac{t[H]\langle \mathbf{let}\ z = \{\ \texttt{Receive}\ x\ \}\ \mathbf{in}\ S\ \mathbf{end}\ C\rangle}{x = \xi \parallel \xi : K; t'\!:\!y|K_h}$$

R:bk:scp
$$\frac{t[H\ \mathbf{esc}]C}{0} \;\Bigg\|\; \frac{t[H]\langle \mathbf{esc}\ C\rangle}{0}$$

Debugging μOz

- An interpreter of the reversible semantics is nearly a reversible debugger
- A debugger needs the following commands
  - Commands to control execution
  - Commands to explore the configuration
    - Both code and state

Step commands

- Step forward
  - Standard
  - The user specifies the target thread
  - Step forward not enabled if waiting for resources
  - Receive from an empty queue
- Step backward
  - Only in reversible debuggers
  - The user specifies the target thread
  - Not enabled if waiting for dependencies to be undone
  - E.g, cannot step back the creation of a thread with not empty history

Other execution commands

- Run
  - Standard
  - Requires to define a scheduler
- Roll
  - Only in causal consistent reversible debuggers
  - Undo of a past action, including its consequences
  - May involve many threads
  - Should follow the dependencies

Configuration commands

- List of threads
  - Only in concurrent debuggers
- Display of the store
- Display of the code of a thread
- Display of the history of a thread
  - Only in reversible debuggers

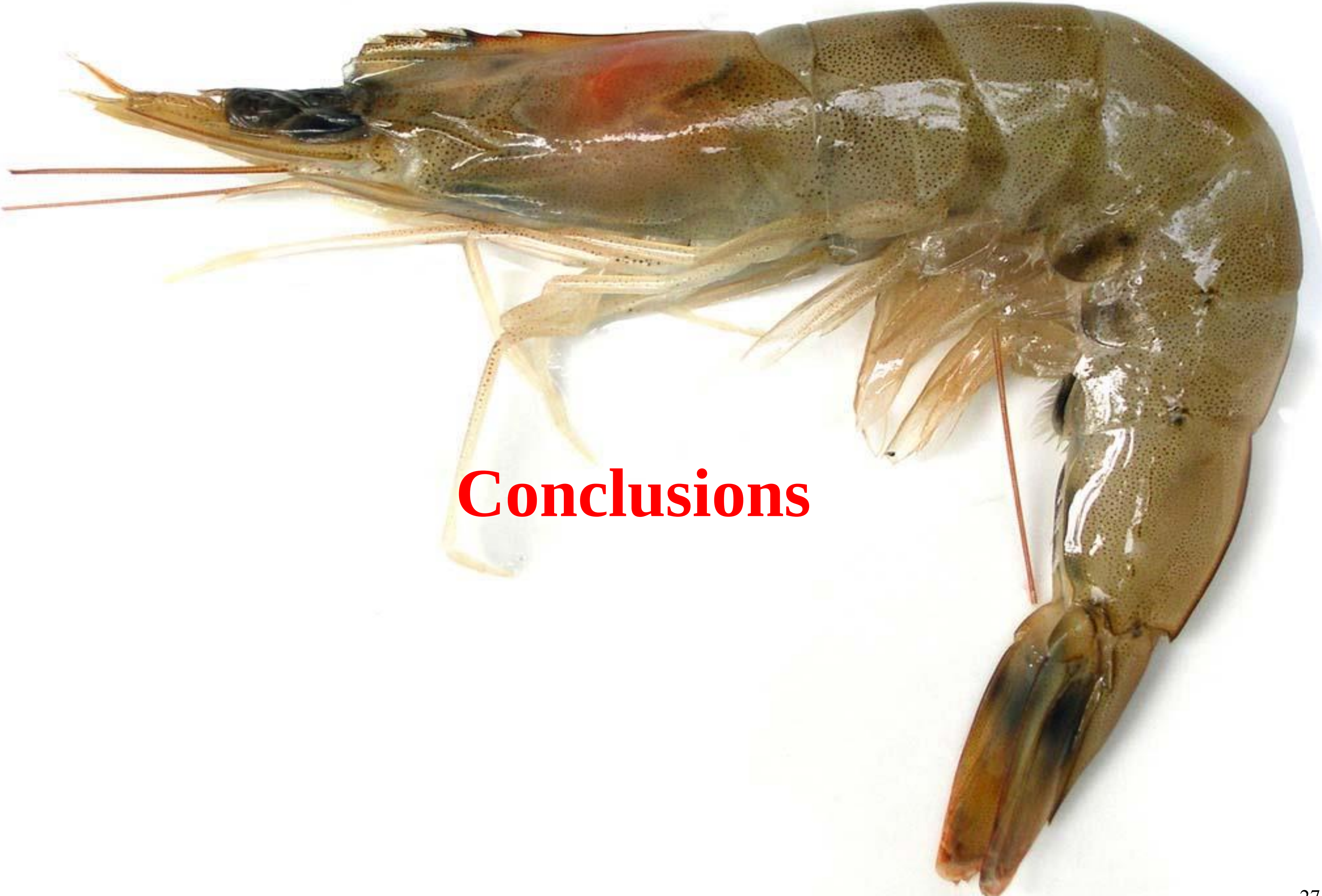
Dump and restore

- When debugging I may go back
- If the try is unsuccessful I may go forward again to the state I come from
- I normally do not record forward states
- Dump and restore solve the issue

Our prototype debugger

- Disclaimer: only a prototype
  - Quite unusable
  - Will improve in the future
- Written in Java
- Closely follows the semantics we have seen
- Available at `http://proton.inrialpes.fr/~mezzina/deb/`
- Starts with `java -jar deb.jar inputfile`

Conclusions



Summary

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

Future work: framework

- Many open questions
- Can we apply our techniques to mainstram concurrent languages?
  - Concurrent ML, Erlang, Java, ...
- Behavioral equivalences
  - How can we reason on reversible programs?
  - How to define compositional semantics?
- Implementation issues
  - Can we store histories in more efficient ways?
  - How much overhead do we have?
  - Trade-off between efficiency and granularity of reversibility

Future work: applications

- Can we find other killer applications?
  - Software transactional memories
  - Existing algorithms for distributed checkpointing
- Improving the debugger
  - Which are the commands we can provide?
  - Which debugging strategies they enable?
  - Which kind of bugs can they help to find?

Finally

Thanks!

Questions?